# RUNTIME TECHNIQUES FOR EFFICIENT EXECUTION OF VIRTUALIZED, MIGRATABLE MPI RANKS

BY

SAM WHITE

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kalé, Chair
Professor Bill Gropp
Professor Luke Olson
Dr. Atsushi Hori, National Institute of Informatics, Japan

## ABSTRACT

The Message Passing Interface (MPI) is the dominant programming system for scientific applications that run on distributed memory parallel computers. MPI is a library specification or standard maintained by the MPI Forum. The first MPI standard was ratified in 1994, with MPICH providing a reference software implementation. Over the nearly 30 years since, the MPI standard has continued to grow, with version 4.0 ratified in 2021. Still, most MPI implementations today trace their roots back to MPICH or other systems developed in the 1990s. At that time, nodes consisted of a single core, memory hierarchies were relatively flat, systems had very high reliability, and performance was generally predictable. Modern HPC systems share none of these characteristics. All nodes are multicore, with increasing on-node parallelism available year after year. Extreme scale systems may be reliable as a system but suffer from individual node and link failures, limiting their usefulness for long-running jobs at large scale. Finally, performance has become harder to predict due to many factors, including processor frequency scaling and contention over shared resources. At the same time, scientific applications have become more dynamic themselves through the use of adaptive mesh refinements, multiscale methods, and multiphysics capabilities in order to simulate particular areas of interest with higher fidelity. Our work addresses all of these issues through overdecomposition, creating more schedulable tasks than cores. We use Adaptive MPI (AMPI), an MPI implementation developed on top of Charm++'s asynchronous tasking runtime system, as the basis for all of our work. AMPI works by virtualizing MPI ranks as user-level, migratable threads rather than operating system processes.

In this thesis, we identify and overcome the issues associated with virtualizing MPI ranks as migratable user-level threads. These issues include problems of program correctness under virtualized execution, increased per-rank memory footprint, communication performance– both point-to-point and collective, in terms of latency, bandwidth, and asynchrony– and interoperability with other parallel programming systems commonly used on extreme scale systems. The resulting techniques and insights are applicable to other parallel programming systems and runtimes, while our AMPI implementation is as a result much more widely applicable and efficient for legacy MPI codes.

## ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Sanjay Kale. In 2014, I thought I was fortunate not only to get into this department but especially to work with a researcher as highly respected as Sanjay. During my undergraduate degree I was introduced to MPI, and had heard from a couple different people about Charm++ as an intriguing alternative. In hindsight it was more fortunate and rewarding than I could have possibly imagined at the time to be able to work on an MPI implementation on top of Charm++ with Sanjay, and to be able to bounce ideas off of him whenever I wanted, the conversations often going places I didn't expect. Thank you for providing me with such an enriching and supportive environment to grow as a software engineer, as an independent researcher, and as a person.

I would also like to thank my committee members–Bill Gropp, Luke Olson, and Atsushi Hori–for their inspiration, support, and guidance. My work owes greatly to their own, and I'm grateful to have had the opportunity to work with each of them.

My research group, the Parallel Programming Laboratory, also deserves a special thanks. To those who came before me, I thank you for setting expectations high and for leaving those after you a software ecosystem as rich for research exploration as Charm++. For those that I worked with– in particular, Abhinav Bhatele, Phil Miller, Ehsan Totoni, Harshitha Menon, Nikhil Jain, Bilge Acun, Seonmyeong Bak, Juan Galvez, Michael Robson, Eric Bohm, Ronak Buch, Eric Mikida, Nitin Bhat, Kavitha Chandrasekar, Jaemin Choi, Matthias Diener, Justin Szaday, and Zane Fink– thank you for sharing your ideas and ambitions, and for your efforts at furthering our group's joint work. I would also like to thank my colleagues at Charmworks Inc for their work on infrastructure and ideas that benefited my work greatly. In particular, Evan Ramos's work on the privatization techniques in Chapter 2 and Nitin Bhat's work on RDMA communication support in Chapter 3 helped to fully realize the promise of my work.

My undergraduate research advisor, Tia Newhall, also deserves particular mention. Tia introduced me to systems and parallel programming both, as well as Computer Science research more broadly. I never expected to major in CS when I started at Swarthmore, but Tia and the rest of the department there encouraged me to continue taking classes and pursue research after taking my first CS class on a whim– thank you.

I would also like to thank my friends and family for their love, support, and inspiration. To my parents, Carl and Marilyn, who instilled in my brothers and me a curiosity and love of learning that has resulted in all three of us earning doctoral degrees in different fields– thank you for everything. To my brothers, Mike and Craig– thank you for inspiring and

challenging me with your examples. To my mother-in-law, Karen, and her family– thank you for welcoming me into your family and inspiring me to be a better person every day.

Lastly, and most importantly, I would like to thank my wife, Carol-Lynn, for her love, support, and patience over the last eight years of graduate school. In particular, your own love of learning has always boosted mine when I felt discouraged, your dedication to your students has inspired me to work harder, and your perspective has helped refresh mine when I felt defeated. Your love has made the successes worth celebrating and the challenges worth enduring, and now I cannot wait to see what the rest of our life brings.

## TABLE OF CONTENTS

# CHAPTER 1: OVERVIEW

The Message Passing Interface (MPI) is the dominant parallel programming system for scientific applications that run on distributed memory parallel computers. While the MPI standard specifies many useful features, current implementations have constraints that restrict applications heading into the exascale era. In particular, as nodes are becoming increasingly parallel and subject to higher performance variability, the MPI-everywhere or MPI-only model is falling out of favor. MPI+X (where X is a shared memory programming system) is the major alternative. MPI+X alleviates issues related to per-rank memory footprint and interprocess communication, but also suffers from serialization around communication and the additional development effort required. With current MPI libraries assuming that ranks are equivalent to operating system processes, they are limited in what can be shared efficiently between the ranks that are co-located on a node.

Adaptive MPI (AMPI), unlike most MPI implementations, virtualizes MPI ranks as user-level threads, rather than operating system processes, allowing many ranks to inhabit the same address space. We refer to this technique of decomposing the problem domain into more ranks than execution units as overdecomposition. Overdecomposition empowers the runtime system to schedule ranks based on the availability of messages for them, overlapping communication of one rank with computation of others on its same processing element (PE) or kernel thread. Overdecomposition also provides opportunities to optimize for communication locality, to dynamically balance load during execution, and to ensure resilience to faults. Overdecomposition is achieved in AMPI through process virtualization. Virtualization is a powerful technique for bringing the benefits of overdecomposition to bear on legacy applications. However, it also brings with it potential overheads such as increased communication volume and higher memory usage, not to mention code modifications necessary to virtualize legacy applications. In this thesis we define and address the challenges associated with efficiently supporting the execution of virtualized, migratable MPI ranks on modern high performance computing systems. Our work builds on AMPI and its underlying runtime system, but the techniques and insights are applicable to other programming systems and their implementations as well.

Our work takes a holistic approach to creating an MPI implementation that effectively virtualizes MPI ranks as threads and optimizes for communication and memory usage through various runtime techniques. We develop new methods for automatic process virtualization which are portable and support shared memory execution and dynamic rank migration. Our improved runtime takes advantage of the shared address space between ranks on the same

node for fast, asynchronous communication– both point-to-point and collective–with low memory footprint. This requires rethinking the messaging semantics of AMPI's underlying task-based programming system in multiple ways in order to support efficient in-place data communication. We also pursue optimizations for more asynchronous MPI communication, with and without semantics changes to MPI, building on our runtime support for shared address space between ranks. Further, we reduce the per-rank memory footprint of both the runtime and the application data, allowing applications to scale to problem sizes previously unattainable. Finally, we consider interoperability of our runtime with others commonly used in high performance computing, studying the trade-offs and performance challenges while developing an integrated AMPI+OpenMP runtime system. We motivate and demonstrate the utility of this unique approach with applications and benchmarks along the way in order to show its effectiveness and potential.

## 1.1 BACKGROUND

### 1.1.1 MPI

The Message Passing Interface (MPI) is the predominantly used programming system in high performance computing. The MPI standard has been developed and evolved over the course of more than 25 years now. It specifies the syntax and semantics of everything from messaging to file I/O to interoperation with threading models and process creation. In all, it defines over 450 API routines to help application and library developers write portable, scalable code to run on parallel computers.

The Message Passing Interface was first standardized when large-scale systems consisted of distributed memory processors with a single core each, performance was predictable at various levels, and systems were highly reliable. Systems are now generally composed of nodes with tens or hundreds of CPU cores, each with variable frequency. Nodes also have specialized co-processors, deeper memory hierarchies, and are susceptible to failures at scale. While the MPI standard has evolved to address some of these trends, current MPI library implementations are limited by the assumption that an MPI rank is equated to an operating system process, each with its own separate address space.

The MPI standard has evolved to better support shared memory nodes in multiple ways: by defining multiple thread safety levels, through the addition of shared-memory windows between ranks on a node, by adding routines for thread-safe message reception, and, most recently, with the new partitioned communication APIs. There has also been much research on the limitations of current MPI libraries interoperating with threading models. These

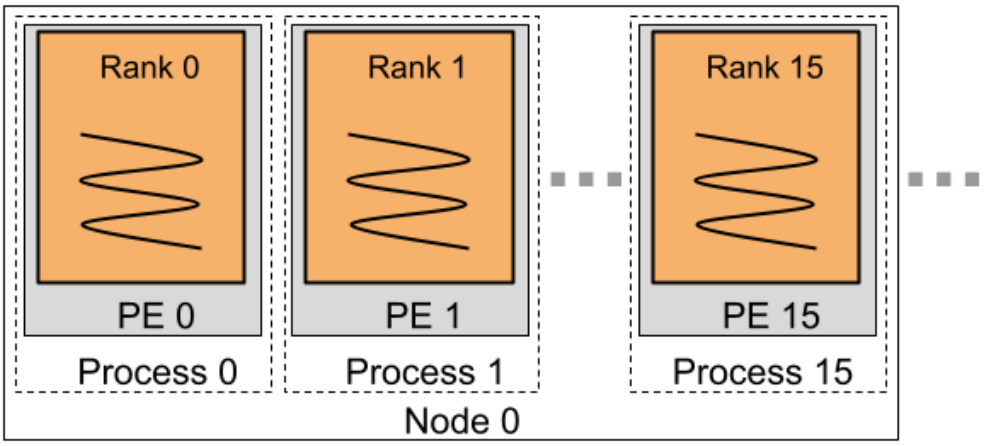



Figure 1.1: Traditional process-based MPI libraries implement the MPI-everywhere programming model on modern multicore nodes such that each rank is a separate process on a PE or core of a node. Note the process boundaries separating the address spaces of each rank in the node.

tend to focus on decreasing serialization and contention inside the MPI runtime when the application is using *MPI_THREAD_MULTIPLE* and calling into MPI simultaneously from different threads. This is typically achieved by using finer-grain locks around MPI's internal messaging data structures and by using locking schemes that minimize contention over shared resources as much as possible [1, 2, 3, 4]. An alternative approach examined the MPI messaging semantics to identify the sources of overhead and proposed relaxed semantics that are more amenable to shared memory parallelization [5]. Another has proposed refactoring applications to use communicators, windows, and tags to express logical parallelism between threads in a process and to exploit that in the communication runtime [6]. In spite of these advances, many applications still run most efficiently as MPI-everywhere or else are legacy codes without explicit shared memory multithreading support. If they do use MPI+X, it is often done using the *MPI_THREAD_FUNNELED* or *MPI_THREAD_SERIALIZED* threading levels, which move the serialization around communication from inside MPI up to the application code. This requires that the application be carefully developed and maintained to preserve data locality and affinity among threads, which are implicit in the message passing programming model [7, 8].

Even with OpenMP or another shared memory system, it is common to run with multiple processes per node still. In that case, or with MPI-everywhere, traditional MPI libraries equate MPI ranks with operating system processes, each with their own separate address space. Much work has been done to optimize communication between processes on the same multicore node in MPI using POSIX shared memory or kernel-assisted interprocess copy

mechanisms such as KNEM, LIMIC, CMA, and XPMEM [9, 10, 11, 12]. These mechanisms provide different kinds of shared memory access but have their own overheads, such as system call overhead, memory registration costs, and kernel-level locking around shared resources. As a consequence of these limitations, point-to-point and collective communication routines have been optimized separately for each [13]. They also require kernel support and so are not portable across systems.

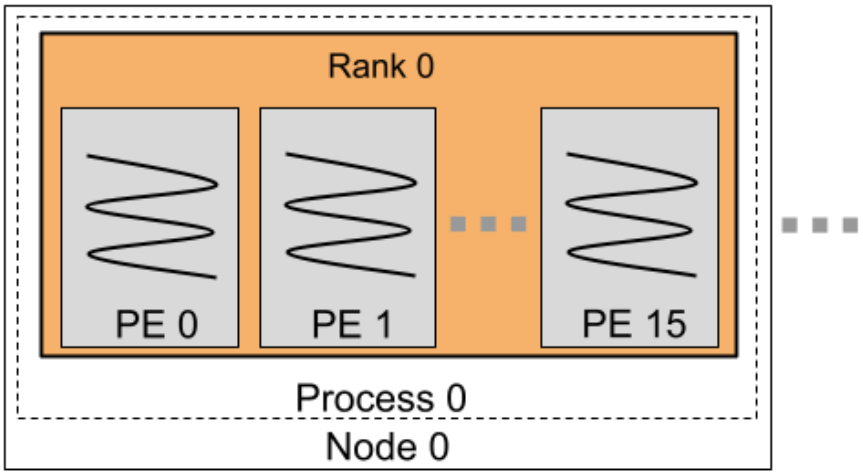



Figure 1.2: Hybrid MPI+X parallelism (where X is a shared memory programming model) typically exploits shared memory parallelism with multithreading within the shared address space of a single MPI rank, such that multiple threads share a single MPI rank or endpoint.

An alternative approach to the MPI+X one is to hoist the threading inside of the MPI runtime, such that a rank becomes associated with a thread rather than a process. This approach allows existing MPI-only codes to gain some of the benefits of shared memory execution without rewriting the application logic explicitly for it. This is how AMPI [14], MPC [15], FG-MPI [16], and other threaded MPI implementations work. They are similar to the MPI endpoints proposal in that multiple ranks (with their own private messaging resources) share the same address space [17]. With threads as ranks, no locking is needed around MPI's internal messaging data structures, as it is with *MPI_THREAD_MULTIPLE*. Additionally, messages sent between threads in the same address space can be optimized to take advantage of the user-space shared memory that comes naturally from a shared address space. There are no portability constraints on this shared address space, and none of the overheads of kernel-assisted interprocess copy mechanisms. The runtime is also free to use this shared address space for more asynchronous communication support, managing the issuance and completion of messages for many ranks in the node while those ranks concurrently execute on different cores. In this thesis, we focus on optimizations to MPI in

the context of ranks as threads, running on modern high performance computing systems composed of many shared memory nodes. First, we introduce AMPI's underlying runtime system, Charm++, before providing background on AMPI itself and outlining the structure of the thesis.

### 1.1.2 Charm++

Charm++ is a general purpose parallel programming system with a dynamic runtime system [18]. Users are responsible for decomposing the parallel computation into work and data units which are scheduled by the runtime system in a message-driven manner. The data units are special C++ objects called *chares*, which have special methods that can be remotely and asynchronously invoked by other chares. Chares can be organized into collections called chare arrays, which support collective communication routines as well. All communication is asynchronous and nonblocking, even collectives. Unlike in MPI, messages in Charm++ are in no way ordered; instead, they are executed in the order of their arrival. Chares are persistent and, while migratable across processing elements during runtime, are by default tethered to a single home processing element (either a core or a hyperthread) on which their method invocations are executed. This execution model serves to promote locality, since all methods of chare are executed on the same PE by default, while still enabling dynamic remapping of tasks at runtime for improved load balancing. Many scientific applications perform a spatial decomposition of the simulation domain and because the problems naturally exhibit high spatial locality, with any dynamic load imbalance evolving slowly over many timesteps. Charm++ separates the logical decomposition of the problem domain into objects from the mapping of objects to the cores and nodes of a parallel computer. This separation of logical work and data units from their mapping to cores or nodes is key to enabling runtime optimizations such as message-driven scheduling and dynamic load rebalancing without the need for invasive application code refactoring. Charm++ is based on C++, with some extensions needed to support globally visible objects, serialization of remotely invocable method parameters, and more. Charm++ is one example of task-based programming model based on an adaptive runtime system. Other examples include Chapel [19], HPX [20], and Legion [21].

The Charm++ runtime system supports shared memory systems by having a dedicated communication thread per process, which handles all the off-node messaging needs of the scheduler threads within it. This is illustrated in Figure 1.4 and is called "SMP mode" in Charm++. The communication thread simply polls the network for incoming messages and forwards these to the destination PE as required. It also handles all outgoing messages that

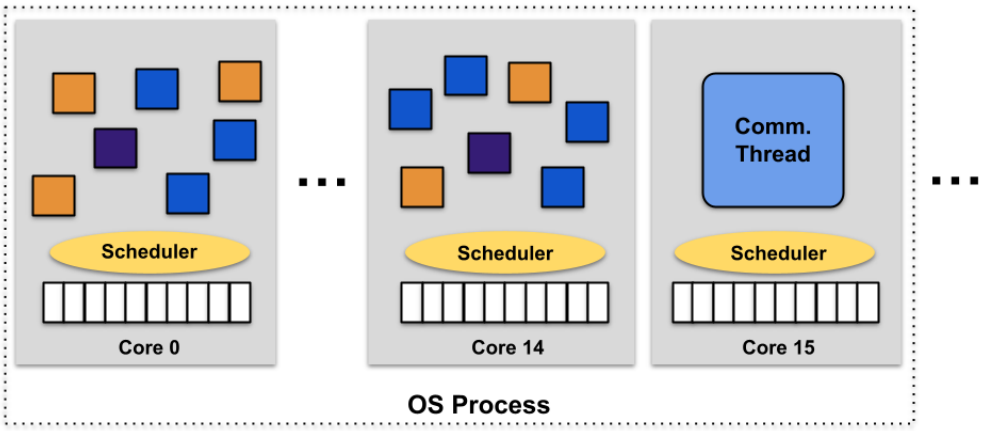



Figure 1.3: Charm++ in "SMP mode": this example shows multiple collections of parallel objects (chares) running within a single OS process on a node with 16 cores, one of which is dedicated to handling off-node communication.

must be sent outside the process. Messages in Charm++ are first-class objects, enabling chares to own messages and reuse their memory. When a chare sends a message to another chare that happens to be in the same process as it, the runtime system simply passes the message by pointer, avoiding any data copies of the message payload. When a chare communicates with a message in a separate process, the runtime enqueues the outgoing message in the communication thread's queue, and it is transferred asynchronously.

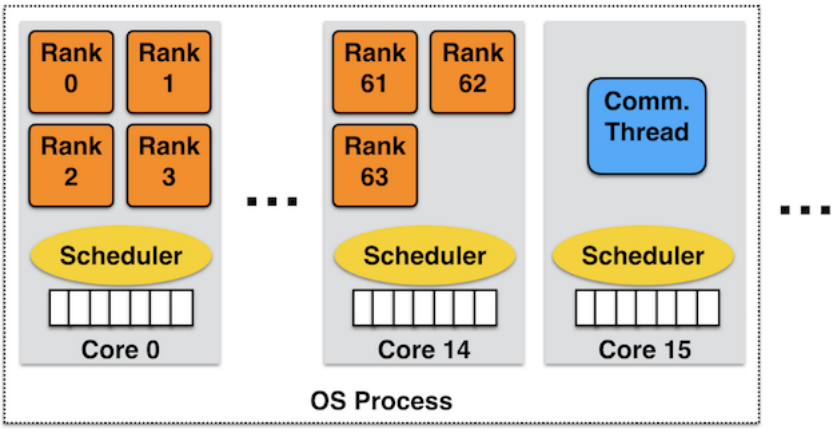



Figure 1.4: AMPI in "SMP mode": this example shows 64 MPI ranks running within a single OS process on a node with 16 cores, one of which is dedicated to handling network communication.

### 1.1.3 AMPI

Adaptive MPI is an implementation of the MPI standard on top of Charm++ [14]. Prior work by the Parallel Programming Laboratory initially developed AMPI to support a sub-

set of the MPI 1.1 standard that was most commonly used by applications at the time [22]. AMPI virtualizes ranks by implementing them as User-Level Threads (ULTs), which are associated with chare array elements in Charm++. Multiple AMPI ranks can be co-scheduled on the same PE, just as chare array elements can reside on the same PE. When AMPI reaches a blocking MPI call which cannot immediately be completed, it suspends the user-level thread and returns control to Charm++'s scheduler on that PE. The scheduler will then process any other messages its has in its queue, either for that rank or any others on that PE. Since ULT context switches are fast, this allows hiding the latency of one rank's communication with computation of other ranks on the same PE–without needing to implement double buffering or more complicated non-blocking communication schemes in the application logic.

ULTs are also migratable, enabling load balancing and fault tolerance in a manner transparent to the application. Dynamic load imbalance is typically difficult for MPI users to address at the application level, since doing so requires tracking execution times or modelling them, deciding if load rebalancing is necessary, then repartitioning the domain across all ranks or otherwise redistributing work and tracking which ranks own which parts of the domain. AMPI's execution model makes this much easier by making ranks transparently migratable and building on top of Charm++'s dynamic load balancing infrastructure. Since the runtime system manages the location of virtual rank using an efficient, distributed protocol, the application does not need to be aware of a rank's actual location. All communication still happens between logical rank identities. And since the load balancing strategy is entirely separate from the application code, tuning the rebalancing to account for different performance characteristics like communication, heterogeneous processor types, and more becomes easier. Rank migration also makes possible dynamic shrink/expand of the number of processors that a jobs is running on, and fault tolerance strategies such as checkpoint/restart.

AMPI load balancing works by having the runtime monitor idle time and optionally other metrics on each PE, and then migrating ranks around to rebalance the overall load. AMPI rank migration is automated by a memory allocator called Isomalloc, illustrated in Figure 1.5. Isomalloc reserves a unique slot in the global virtual memory space for each virtual rank, and maps all allocations from a particular rank into its slot. When migrating between nodes or address spaces, then, Isomalloc ensures that all data is migrated to the same virtual address on the destination node as it was on the source. This ensures that pointers work transparently across migrations without the need for users to be aware of it or to write their own serialization routines, which can be cumbersome for legacy applications. Isomalloc is used to allocate both the user-level thread stack and all heap data that a virtual

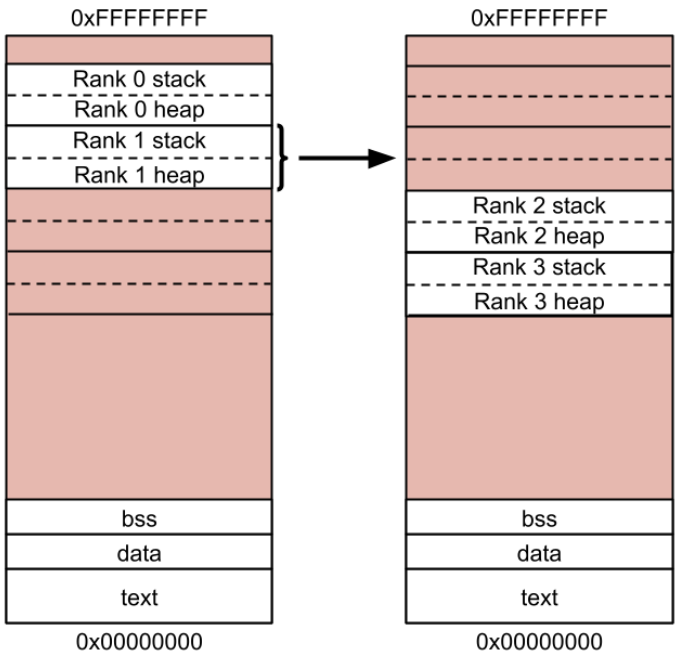



Figure 1.5: Isomalloc memory allocator automates transparent rank migration between address spaces. Here rank 1 is migrating from PE 0 to PE 1, and Isomalloc ensures its memory is mapped to its unique slice of the global virtual memory.

rank allocates. This enables both dynamic load balancing and automatic checkpoint/restart schemes for fault resilience, both without the user having to make pervasive code changes. A checkpoint can be thought of as a migration to storage. For load balancing or checkpointing, the user simply passes an *MPI_Info* object to an extension routine, *AMPI_Migrate()* which says whether they are requesting a checkpoint, load balancing, or something else.

Figure 1.6 shows a Gantt chart of the LULESH proxy application running on AMPI without virtualization (top) and with 8x virtualization (eight ranks per PE) and dynamic load balancing. Note the idle time in the unvirtualized run that arises during the *MPI_Allreduce* operation, with all PEs waiting on the slowest PE to catch up. This is how dynamic load imbalance often manifests in a bulk synchronous MPI application– a collective communication call appears to take longer than expected. Really, the collective communication call's latency is not the issue, but the imbalance leading to a straggler rank (here, rank 0) calling it after others is. Essentially, the program runs at the speed of the slowest PE. For runs across thousands of processors, this problem is amplified, and it can be caused by system noise, variations in hardware, or dynamic software behaviors such as particle movements, multiscale methods, or adaptive mesh refinements. With AMPI, rank virtualization allows co-scheduling multiple ranks on each PE in order to tolerate latency, and dynamic rank migration provides a means of rebalancing the overall load, as shown on the bottom. Note

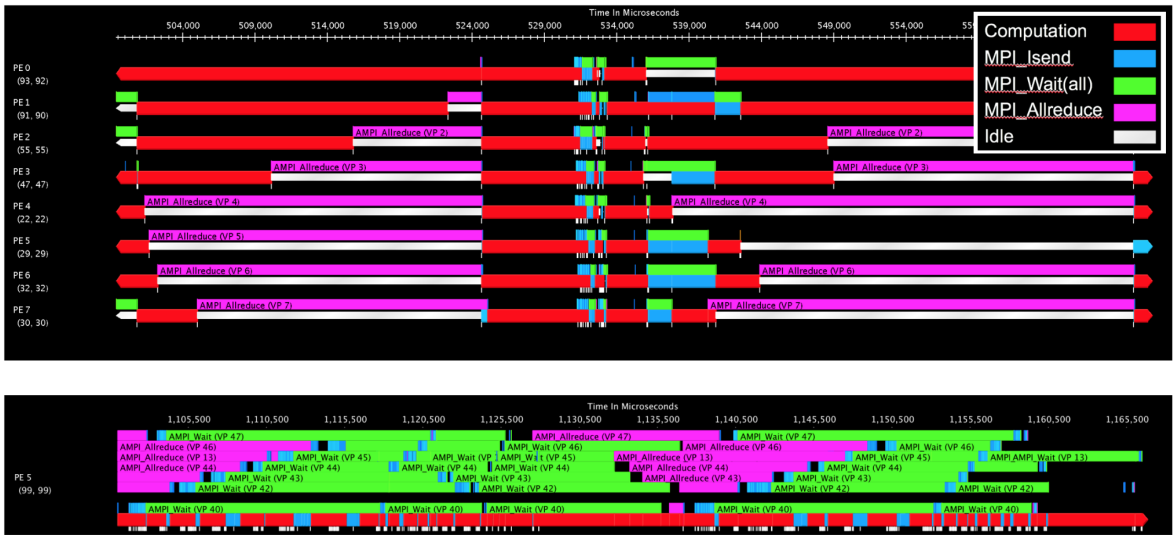



Figure 1.6: LULESH timelines of execution showing 8 ranks on 8 PEs (without virtualization or load balancing) on top compared to 8 virtual ranks being co-scheduled on 1 PE after load balancing.

that the bottom-most bar shows execution on the PE, while the the other bars show where each rank is executing or blocked.

Previous to our work, AMPI was proven on applications with dynamic load imbalance, and for recording the behavior of MPI applications to then simulate their performance on different kinds of systems [23, 24, 25, 26]. As part of our work, not the focus of this thesis, we have striven to make AMPI a complete MPI-3.1 implementation, to improve its usability as a drop-in replacement for popular MPI implementations such as MPICH and OpenMPI, and to make it competitive with vendor-tuned MPI libraries in terms of communication latency and bandwidth. The end result of that effort makes our work in this thesis more broadly applicable to existing MPI applications.

## 1.2 THESIS OBJECTIVE

Overdecomposition is a powerful technique for addressing common parallel performance pitfalls such as dynamic load imbalance and synchronization costs. Achieving overdecomposition in existing MPI applications can be challenging, but process virtualization based on user-level threads has the potential to provide legacy applications those benefits, as well as efficient shared memory execution, without the need for invasive code refactoring. However, virtualization also brings with it overheads, such as increased communication volume, higher memory usage, and more ranks participating in collective communication calls, not

to mention code modifications necessary to privatize any mutable global state.

This thesis aims to identify and overcome challenges associated with the virtualization of MPI ranks as user-level, migratable threads in order to improve the performance of high performance computing applications. From automating process virtualization to minimizing the memory overheads and optimizing communication within and across shared address spaces, we develop runtime techniques for the efficient execution of MPI ranks as threads on modern supercomputing systems.

## 1.3 ORGANIZATION

The rest of this thesis is organized as follows:

Chapter 2 deals with process virtualization, a core technique to making AMPI applicable to legacy MPI codes in an automated fashion. Process virtualization is necessary for correctness of legacy codes running on AMPI's multithreaded execution model, in which global and static variables are shared across virtual ranks in the same address space. We identify shortcomings in existing models–mainly portability, degree of automation, and support for advanced runtime features such as dynamic rank migration and support for arbitrary degrees of overdecomposition. We develop support for three new runtime techniques, one of which advances the state of the art for runtimes depending on process virtualization such as AMPI. We explore the performance implications of this technique in depth.

Chapter 3 identifies memory consumption as a limitation of AMPI's virtualization approach, but provides solutions to minimize it and make runtime memory usage more scalable. We investigate the memory usage of an AMPI application to motivate the need for several changes. First, we demonstrate that the communication semantics of AMPI's underlying runtime system cause inefficiencies in its implementation and develop new asynchronous in-place communication interfaces. Then, we apply similar techniques to rank migration and its transient memory overheads. Finally, we look at minimizing the per-rank memory footprint of the runtime system, taking advantage of the shared address space between ranks on a node. Overall, our work enables applications to scale further and to run with larger problem sizes than possible previously.

Chapter 4 discusses point-to-point communication issues in order to motivate several optimizations for fast, asynchronous communication. We make use of the shared address space between ranks on a node for not only faster transfers but also for more asynchronous and concurrent communication. We optimize for communication locality between ranks on the same PE or in the same address space whenever possible, and optimize the scheduling costs of our runtime. We also explore optimizations for MPI's full messaging semantics as well

as a relaxed set of semantics that permits further optimizations. This necessitates moving beyond the object-oriented, message-driven scheduling approach of Charm++ in favor of more directly making use of the shared address space across a process.

Chapter 5 evaluates collective communication routines for inefficiencies with respect to AMPI's virtualization and support for rank migration before proposing improved designs and novel adaptive algorithms. We consider different message sizes, scalability, and the impact of migration on various collective routines such as broadcast and allreduce, with and without non-commutative operations. We problematize non-commutative reduction operations as being particularly challenging to optimize within AMPI's execution model of virtualized, migratable ranks. We develop more adaptive algorithms that can optimize for rank ordering when ranks are mapped in the natural, block fashion to PEs but can also adapt gracefully to disordered mappings by opportunistically combining messages, taking advantage of associativity of the reduction operator.

Chapter 6 focuses on interoperability of AMPI with other commonly used parallel programming models such as OpenMP, CUDA, and Charm++. We explore OpenMP interoperation for not only shared memory parallelism but for transient load balancing by integrating OpenMP scheduling into our runtime. This allows cooperatively scheduling OpenMP parallel loop iterations with AMPI ranks on the same PEs. We then optimize GPU point-to-point communication in AMPI as well as providing for asynchronous kernel launch and completion. Finally, we develop support for AMPI-Charm++ interoperability, wherein users can independently set the number of virtual ranks and chare array elements and have them be co-scheduled and to communicate between the two models. This will enable new application workflows and make using MPI libraries easier from Charm++ applications.

Finally, Chapter 7 concludes with a summarization of the thesis's contributions and future directions.

# CHAPTER 2: AUTOMATIC PROCESS VIRTUALIZATION

## 2.1 INTRODUCTION

As high performance computing systems continue to evolve into exascale, application developers are being asked difficult questions. Can their code scale up to exascale? Can it make use of heterogeneous compute nodes? What about different memory technologies or network interconnects? Can it tolerate network latency given the increased relative costs of data movement compared to floating point operations? Can it use multiscale or other dynamic methods to increase simulation resolution only where needed, in areas of interest? What happens to your code if a node fails during the run?

Concurrent with these hardware advances, asynchronous many-task runtime systems are proving their ability to manage resources dynamically in response to changing application and hardware behaviors. These programming models and runtime systems address the need identified in a recent US Department of Energy Office of Science report titled "Reimagining Codesign for Advanced Scientific Computing: Report for the ASCR Workshop on Reimagining Codesign" for decoupling algorithms from the mapping and scheduling of computational work onto the hardware system, writing that the authors "recognized the need for more intelligent, dynamic runtime systems that can schedule and map computation to appropriate resources in an intricate heterogeneous system with complex memory hierarchies" [27]. Task-based programming systems such as Legion [21], HPX [20], Charm++ [18], and Chapel [19] are able to monitor performance during execution and introspectively adapt execution on the fly. They do so in part by taking advantage of different programming models that have been designed from the ground up with asynchrony, heterogeneity, and fault resilience in mind. Their unique programming models also dictate, however, that porting any existing code to them requires significant programmer effort. Serial portions of code that constitute the lowest level tasks can usually be left as is, but the parallel control flow must be rewritten. For legacy codes, this is often a non-starter due to code complexity and the developer effort required. Consequently, effort has been put into making MPI interoperation work with these tasking models, though that still requires part of the application to be rewritten, and then execution has to be handed off between MPI and the tasking runtime's scheduler which requires synchronization [28].

An alternative approach is to reimagine the execution model of existing parallel programming models or libraries, such that the existing code can be run on the tasking runtime system. MPC [29] and AMPI [14] are examples of such an approach applied to the MPI

programming interface [30]. MPI is widely used in the high performance computing domain and as of now lacks high-level, easy to use support for some of the key features that tasking runtimes offer such as efficient integration with shared memory programming models, dynamic load balancing, and fault resilience. In order to fulfill this promise, these runtimes rely on the technique of *process virtualization*.

Process virtualization, as we define it, refers to the abstraction of operating system processes such that code meant to run as a process can instead be run on a thread. For example, a code that uses a mutable global variable (such as in Figure 2.1) cannot be virtualized as is, since the global variable itself will be shared among multiple threads. We refer to the process of converting a code, either manually or automatically, to a virtualizable code as *privatization*. Process virtualization allows abstracting the notion of an MPI rank, which typically in a library-based MPI implementation is equivalent to a process, instead creating and running multiple MPI ranks as user-level threads within each process. The runtime system then manages the scheduling and communication between ranks, even supporting dynamic migration of ranks across address spaces at runtime. Process virtualization is, consequently, key to enabling adaptive runtime features to work on a legacy MPI application, and fully automatic privatization support is a common goal of runtimes like AMPI or MPC.

Previous work has attempted to automate process virtualization, with differing degrees of success. We summarize the state of the art privatization methods in order to motivate three novel runtime approaches, the last of which achieves a new degree of applicability to legacy codes with greater portability across compilers and linkers, high performance in multiple aspects, and support for advanced features such as dynamic rank migration. This work is critical in making the advanced runtime features of AMPI and others like it available to the legacy applications that need them in order to scale efficiently to the next generation of high performance computing systems.

To make privatization more concrete, Figure 2.1 provides a small code sample of an unsafe MPI program and a possible output of executing it in a virtualized manner (with multiple ranks in the same OS process). In the example output here, note that the zeroth rank sets the global variable *my_rank*'s value to its rank number 0, then blocks in the *MPI_Barrier()* call. Next, rank 1 will be scheduled and set *my_rank*'s value to its rank number 1 before suspending in the barrier. When both ranks are awakened from the barrier's completion in some ordering, they will both print the value of the last rank's number instead of their own, as shown in Figure 2.2. Any MPI user would expect this program to output "rank: 0" and "rank: 1" in some order, and this discrepancy would lead to correctness issues in a more complex code. This is because the variable is global, and global variables are defined in a per-process manner. Static variables are similarly defined per-process and suffer from the

```
#include <mpi.h>
#include <stdio.h>

int my_rank;
int num_ranks;

int main(int argc, char** argv)
{
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &num_ranks);

  MPI_Barrier(MPI_COMM_WORLD);
  printf("rank: %d\n", my_rank);

  MPI_Finalize();
  return 0;
}
```

Figure 2.1: MPI hello world program in C with global variables.

```
\$ ./hello_world +vp 2
rank: 1
rank: 1
```

Figure 2.2: Possible output from executing the MPI hello world program (above) with 2 Virtual Processors (VPs) in 1 OS process.

same issues in terms of virtualization, as shown in Figure 2.3.

Examples of unsafe variables are, in C/C++, non-const global and static variables. In Fortran, implicit or explicit save variables are static, and non-parameter module variables and common blocks are examples of global variables. We note that global variables whose value is written only once to the same value across all ranks are actually safe, since their value can be shared across all ranks. This is true of *num_ranks* in Figure 2.1. Thread unsafe virtualization issues arise when ranks are writing different values to the same variable, and that is why privatization is needed. And since legacy MPI applications and libraries can contain hundreds or thousands of such mutable global/static variables spread throughout the code in a pervasive manner, automatic privatization is essential.

Different approaches to process virtualization have been studied in the past. These approaches vary in degree of automation, applicability to different programming languages and kinds of variables, portability across compilers, linkers, operating systems, and architectures, performance in terms of runtime and memory overhead, and in support for migratability of data.

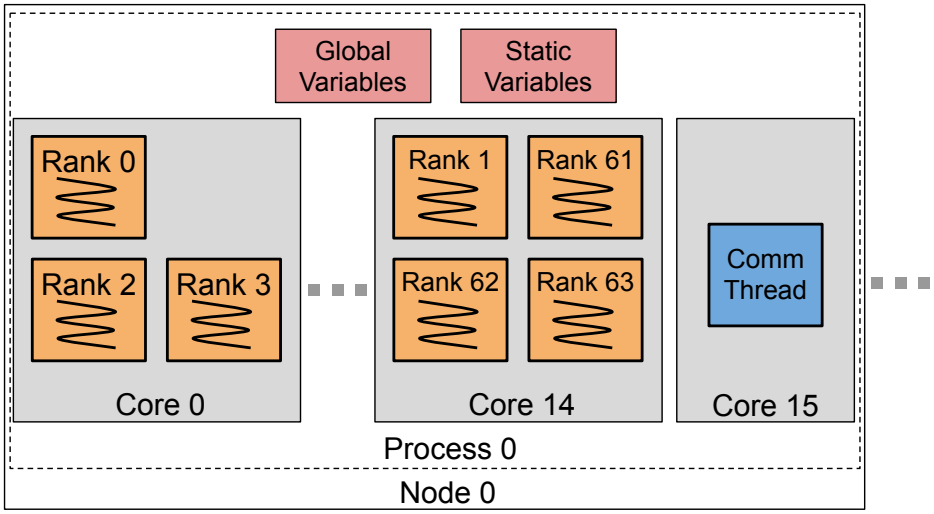



Figure 2.3: AMPI applications typically run with more virtual ranks than cores or PEs. Each virtual rank has its own user-level thread stack and heap memory. Notice that all ranks here share the same set of global and static variables by default. Privatization is necessary to provide each rank with their own separate copy of these variables. Also note that dynamic load balancing can alter the mapping of ranks to cores during execution. Here rank 1 has migrated from core 0 to core 15 on node 0. In practice, there are often multiple virtual ranks per core, and one OS process per socket or node. Ranks can even migrate across nodes.

### Manual code refactoring

Manual code refactoring is what we call the process of rewriting a code so that it does not use mutable global state. This usually requires encapsulating all global/static variables in an application into one or more structures which can then be allocated on the stack or heap and pointers to it passed around to all functions that reference the state. It also involves avoiding the use of thread unsafe library calls, such as C/C++ *strtok* and *getopt*. If a code only contains a few such variables that are rarely referenced, this manual code refactoring process can be doable, since the changes themselves are simple and mostly mechanical to make. However, oftentimes in legacy codes the effort required is significant due to the use of hundreds or thousands of global/static variables being used throughout the code.

### Source-to-source code refactoring tools

Source-to-source code refactoring tools can automate the tedious refactoring process described above. Photran [31] [32] was developed for Fortran codes as an Eclipse plug-in that worked on Abstract Syntax Trees of the code. It encapsulated all global/static variable references into a single Fortran derived type (equivalent to a C structure), and passed that

structure to all functions that referenced the global/static variables. These methods have much promise, since altering the source code remains the most portable method of privatization, so long as the solution can work on various programming languages and ideally incorporate programmer input on grouping variables into multiple structures rather than a single large structure for the entire codebase. We also note that this method can be combined with other methods which are semi-automatic, such as TLSglobals (described below).

### Swapglobals

The Swapglobals method relies on details of the ELF object format to automatically privatize global variables [33]. ELF maintains a Global Offset Table of all global variables, and the table can be swapped out when context switching ranks at runtime. This method does not require any changes to the source code and works with Fortran and C/C++ code. However, it does not handle static variables, since they are not stored in the global offset table. It only works on x86 architectures that fully support ELF, and it requires either a version of the ld linker 2.23 or older or a patched version of ld 2.24 or newer in order to avoid the linker optimizing out the GOT pointer reference at each global variable access. Additionally, it does not work in AMPI's SMP mode (illustrated in Figure 4.1), since there can only be 1 GOT actively in use per OS process at a given time, whereas SMP mode has multiple user-level schedulers running concurrently on each core within an OS process. The non-SMP mode restricts AMPI to having one process per core, rendering some of its optimizations for shared address space communication ineffective [34]. This, combined with its limited portability and applicability to static variables, led to Swapglobals being deprecated.

### TLSglobals

TLSglobals depends on the user tagging their mutable global/static variable declarations with the *thread_local* attribute [33]. The runtime then switches out the TLS segment pointer at each user-level thread context switch. This method works on C/C++ and Fortran (using the *__thread* attribute in C, *thread_local* in C++, and OpenMP's *threadprivate* directive in Fortran), and on static and global variables alike. So far, it only works on Linux and Mac operating systems and on x86 architectures, though it could work on others. It also requires GCC or a recent version of the Clang compiler (v10.0+), because it requires support for their *-mno-tls-direct-seg-refs* option which forces the compiler to access TLS variables through the segment pointer always. As such, it introduces an indirection to each privatized

variable access, which can result in performance degradation. We have extended TLSglobals to work with shared object linking and to work on ARM and Power architectures as well.

### Compiler automated TLS variable tagging

Compiler automation has been developed for MPC with support in the Intel compiler and with patched version of GCC available [35]. The user specifies a compiler option (*-fmpc-privatize*) which tells the compiler to automatically treat all global/static variables as if they were declared as a thread-local variables. It removes the need for users to identify and tag all unsafe variable declarations, and otherwise performs at runtime like TLSglobals. The runtime performance in turns depends on TLS variable access being as fast as access to unprivatized variables, which is architecture specific. This method also requires access to all dependent libraries in source code format so they can be recompiled with the special compiler support. MPC also includes support for hierarchical local storage, with additional attributes defined for data that needs to be privatized to varying levels of the node, core, ULT, or task hierarchy in order to minimize memory overhead [36].

### Process-in-Process (PIP)

PIP is a user-level library developed by Hori et al that can be used to create a shared address space between processes [37]. It has been used primarily to share memory at the user-level between multiple MPI processes resident on the same node for fast intranode communication. To the best of our knowledge it has not been used for MPI rank virtualization or overdecomposition in which multiple ranks are co-scheduled on each PE and can dynamically migrate between nodes, but we recognized its applicability to this execution model. It relies on compiling all code into a Position Independent Executable (PIE). Position independent executables define their contents (including global and static variables) as offsets from the instruction pointer, so that the executable can be loaded into an arbitrary location in virtual memory. This is the default on most modern operating systems for security reasons. PIP then relies on the glibc-specific, non-standard function *dlmopen* which can be called on the PIE binary with a unique namespace index to duplicate all code segments, including the global variables. This method has no compiler requirements (except for support for PIE, which is ubiquitous) and does not require any programmer effort, however, it does require glibc and a patched version of glibc at that in order to create more than 12 namespace indices (virtualized entities) per OS process. This is a seemingly arbitrary limit inside glibc's implementation and so PIP distributes patched versions of glibc along with its source code

Table 2.1: Summary of existing privatization methods and their features.

| Method | Automation | Portability | SMP Mode Support | Migration Support |
|---|---|---|---|---|
| Manual refactoring | Poor | Good | Yes | Yes |
| Photran | Fortran-specific | Good | Yes | Yes |
| Swapglobals | No static vars | Linker-specific | No | Yes |
| TLSglobals | Mediocre | Compiler-specific | Yes | Yes |
| *-fmpc-privatize* | Good | Compiler/linker-specific | Yes | Not implemented, but possible |
| Process-in-Process | Good | Requires GNU libc extension | Limited w/o patched glibc | Unknown |

to get around this limitation.

Existing approaches fall broadly under source code modification tools, compiler-based approaches, or runtime methods, with some combining elements of both. We can categorize and evaluate the existing privatization methods based on several criteria. One being the degree of automation or amount of application programmer effort required; another being portability across compilers, linkers, and operating systems; third being support for many scheduler threads per OS process, each with their own virtualized entities; and finally, support for runtime migratability of virtualized entities between address spaces.

Table 2.1 summarizes our review of existing privatization methods by rating each in terms of these criteria. Which criteria are important to a particular user will vary by their requirements.

## 2.2 AUTOMATIC RUNTIME PRIVATIZATION TECHNIQUES

Based on our desire to avoid portability restrictions such as requiring specific compilers while fully automating privatization, we found the runtime techniques at the heart of the Process-in-Process library appealing. This led to us developing support for three new privatization methods. All three compile the application as a Position Independent Executable and duplicate the code segments for each virtual rank in a process. They differ primarily in how they duplicate the code, and this has significant consequences for portability, performance, and the ability to dynamically migrate ranks. The third technique, PIEglobals, achieves what we believe to be the best yet combination of user-friendliness, performance, portability, and support for dynamic runtime capabilities.

### 2.2.1 PIPglobals

We initially looked into integrating the Process-in-Process library into AMPI for the purpose of automatic global/static variable privatization. After discussion with its developers,

we found that AMPI did not need to use the PIP library directly, and instead we have applied concepts from PIP and implemented the parts that we need from it for our purposes inside AMPI. Doing so also allowed us to target more architectures than PIP did at the time and to streamline the startup process by tailoring PIP's internals to AMPI's specific needs. We call this method PIPglobals.

This allowed us to construct a prototype that works for AMPI privatization. It works by compiling the user program as a Position Independent Executable and linking it against a special shim of function pointers. We cannot simply compile the application as a PIE and then call *dlmopen*, because that would lead to the AMPI runtime system being privatized along with the application code. Instead, we need to privatize only the application while running multiple copies of its code on a single copy of the runtime per OS process. To do so, we refactored AMPI's headers into a function pointer shim library that the application links against. At startup, a small loader utility then calls the glibc-specific function *dlmopen* on the user's PIE binary with a unique namespace index for each virtual rank. The loader uses *dlsym* to locate a special function linked with the user's binary, passes it a structure with pointers to the entire AMPI API in order to populate the PIE binary's function pointers. Then it locates and calls the entry point. This *dlmopen* and *dlsym* process repeats for each rank. As soon as execution jumps into the PIE binary, any global variables referenced within it appear privatized. This is because PIE binaries locate the global data segment immediately after the code segment so that PIE global variables are accessed relative to the instruction pointer, and because *dlmopen* creates a separate view of these segments in memory for each unique namespace index. This also means that there is no work to be done at user-level thread context switch time, and the cost of accessing global data should be the same as in the unprivatized code. We anticipated the startup overheads being insignificant for practical degrees of virtualization (typically in the range of tens of cores per OS process and roughly ten virtual ranks per core).

We did encounter two limitations. First, we cannot support high degrees of virtualization without using the patched version of glibc provided with the PIP library. This particularly limits the utility of the method in SMP mode. Second, we have not been able to implement support for runtime rank migration, which means an AMPI program virtualized via PIPglobals cannot perform dynamic load balancing, checkpoint/restart-based fault tolerance, etc. This is because we cannot intercept the *mmap* calls that happen from inside *ld-linux.so* in order to allocate them via Isomalloc, AMPI's migratable memory allocator. Also we are restricted to GNU/Linux systems due to the reliance on non-POSIX-standard *dlmopen*.

### 2.2.2 FSglobals

FSglobals takes the same idea of PIPglobals but instead of relocating the code in memory we copy it onto a shared file system. This has two main benefits, and one drawback. It makes this method portable beyond GNU/Linux systems and removes the limit of creating 12 virtual ranks per OS process. At the same time, it does require a shared file system and space on that file system for each virtualized rank's copy of the binary. This can cause FSglobals to be slow during startup as well, with all the I/O involved. In particular, we did not expect it to scale well to large runs with many virtual ranks. Overall, it works similarly to PIPglobals but instead of calling *dlmopen* with namespaces we create copies of the PIE binary on the file system and call the POSIX-standard *dlopen*.

We also note that shared objects are currently not supported by FSglobals due to the extra overhead of iterating through all dependencies and copying each one per virtual rank while avoiding system components, plus the complexity of ensuring each rank's program binary sees the proper set of objects.

FSglobals unfortunately suffers from the same issue as PIPglobals of not being able to intercept the code segment copies during initialization. This means that FSglobals does not support dynamic rank migration either.

### 2.2.3 PIEglobals

We developed a third related privatization method in order to support migration with privatization, which we call PIEglobals. We consider it the most fully automated method we have so far. As with PIPglobals, this method builds the user's MPI program as a shared object in Position Independent Executable mode. After initialization of the AMPI runtime, execution is handed off to a loader utility that performs PIEglobals setup. It first opens the PIE shared object using the system's dynamic linking capabilities, and then calls the glibc extension *dl_iterate_phdr* before and after the *dlopen* call in order to determine the location of the PIE binary's code and data segments in memory. This is useful because PIE binaries access global variables relative to the instruction pointer, and they locate the data segment immediately after the code segment. Our PIEglobals loader makes a copy of the program's code and data segments for each AMPI rank in the job via the Isomalloc allocator, thereby privatizing their global state while also ensuring that memory can be migrated across address spaces. It then constructs a synthetic function pointer for each rank at its new locations and calls it.

No further effort is required of the user to achieve global variable privatization beyond

building their program through AMPI's toolchain wrappers with the *-pieglobals* compiler option. Migration (for load balancing or fault tolerance) works because the code and data segments have already been allocated via Isomalloc. Any libraries and shared objects compiled as PIE will also be privatized. The technique is broadly portable to GNU/Linux systems, since all necessary glibc extensions have existed in stable releases of it since 2005. We have validated it on x86, ARM, and POWER architectures.

This method did present a series of development challenges. For one, shared objects maintain tables of function addresses (the Global Offset Table) used by the machine code for indirect lookups to other code, no matter where in memory the code is located, and they must be updated when PIEglobals moves it to a different location. Currently this is done by scanning memory inside the data segment boundaries identified by glibc for contents that look like pointers to the code's original location, which we intend to replace with a more robust method unaffected by false positives. Similarly, C++ codes can contain global variables that are initialized at startup using class constructor methods, which sometimes make heap allocations. With PIEglobals, this takes place at the time *dlopen* is called on the user's binary, before any interception and privatization can take place. These allocations must be logged, replicated for each privatized rank created, their contents copied. It is possible for any arbitrary data written by these constructors to contain pointers to other globals or heap allocated data, as well as function pointers (particularly in classes with virtual functions), which must also be updated. We also found that it is important to open the shared objects only once per OS process, rather than once per virtual rank, in SMP mode to avoid crashes in glibc due to interactions between *dlopen* and pthreads. Finally, we had to ensure that TLS variables inside applications and system libraries are privatized correctly with PIEglobals to each virtual rank by combining the method with TLSglobals.

Another challenge arose inside AMPI when using PIEglobals: anywhere that AMPI previously relied on a function pointer being the same across ranks would break now that each rank had its own unique copy of the code. AMPI implemented user-defined custom reduction operators by simply calling the same user function pointer on whichever core it may need to. With PIEglobals, we had to modify AMPI to subtract the base address from the user function address during *MPI_Op* creation, to store that offset in the op, and to then apply that offset to some rank's base address whenever applying the reduction operator. Since virtual ranks can migrate around the system arbitrarily, however, it is possible for a core to have no virtual ranks assigned to it at a given time. It is possible then that the runtime would be processing a user-defined reduction and a core that has no virtual ranks resident on it would like to combine reduction messages. While we could modify AMPI's reduction algorithms to be aware of empty cores and avoid processing reduction contributions on them,

instead forwarding the messages, since it is rare in practice to have an empty core, we instead require that all cores have at least virtual rank assigned to them during reduction processing with PIEglobals enabled and otherwise throw a runtime error when the reduction can't be processed.

Debugging code privatized with PIEglobals can also be difficult because debuggers such as GDB and LLDB do not know what debug symbols correspond to the manually copied code segments, leaving backtraces mostly mysterious. For this reason we provide a faculty to assist in debugging with virtualization, called *pieglobalsfind*. This can be called at runtime from within a debugger to translate a privatized address back to its original location as allocated by the system's runtime linker, thereby associating it with any debug symbols included in the binary.

Performance-wise, we had three initial concerns when developing PIEglobals. First, we theorized that copies of the code segment might cause instruction cache misses due to redundant copies of code being used separately by each rank. Second, we were concerned that startup costs would be significant due to the need for copying all code segments and then scanning for function pointers in any heap allocated static objects and updating them to point to the privatized code segment. Third, we knew that for large codes the migration overhead would be increased since the code segments must be migrated along with all the rank's heap-allocated memory and its user-level thread stack. In turn, this could make load balancing more costly. We discuss these aspects further in the following performance evaluation.

### 2.2.4 Results

We looked at the runtime performance of our three methods and compared them against other existing methods. We note that runtime performance is but one criteria– among portability, developer effort, suitability to dynamic migration, and maintainability of code– to consider when evaluating privatization methods for an application, but since our three new methods are all runtime-based, the overheads must be kept reasonably low in order for them to be effective.

We break down performance into multiple different aspects: startup or initialization time, context switch overhead, privatized variable access overhead, and migration overhead. Startup or initialization time is a one-time cost for each program execution, while privatized variable access and context switch times are expected to be paid many times per program execution. Migration is typically infrequent– done in reaction to dynamic load imbalance or hard faults– so its price is expected to be paid less often than, say, privatized variable

access. We show microbenchmark results for all of these characteristics as well as results for two production applications.

We used the Bridges-2 supercomputer at the Pittsburgh Supercomputing Center for all of our performance measurements. Bridges-2 is comprised of three different node type partitions: regular memory, extreme memory, and GPU. We use the "regular memory" nodes, each node having 2 AMD EPYC 7742 CPUs with 64 cores and 256 GB of memory. We used GCC v10.2.0 and Charm++'s MPI networking layer (OpenMPI v4.0.5) on the Mellanox Infiniband network interconnect. We compare the performance of AMPI's existing method TLSglobals against our three new ones: PIPglobals, FSglobals, and PIEglobals. We were unable to get Swapglobals working on this system for comparison.

### Startup overhead

We first measure the time spent in AMPI initialization. For the various privatization methods, we generally expected the runtime-based ones to incur higher overhead here. In particular, our three new methods duplicate the code segments of the application binary once per virtual rank in each OS process at startup. Compared to TLSglobals, which only has to copy the TLS segment once per virtual rank per process, we expected our new methods to perform worse. That being said, since the initialization or startup overhead is only paid once per job, some overhead can be tolerated as long as it does not scale up with the node count.

Figure 2.5 shows the startup time performance for each privatization method for eight virtual ranks per process. The worst of our new methods performs 9% worse than the baseline without privatization. We note that with the exception of FSglobals, which relies on a shared file system, the cost is constant per-process and does not increase with node counts.

### Context switch overhead

We next measured the time spent per user-level thread context switch. This is important because AMPI and runtimes like it use overdecomposition to hide latency via message-driven scheduling– increases in scheduling overhead can harm strong scaling performance and limit the degree of profitable overdecomposition. Higher degrees of overdecomposition are desirable for performing efficient dynamic load balancing.

We wrote a microbenchmark that context switches between two different user-level threads with each different privatization technique. Figure 2.6 shows the results averaged over

100,000 context switches. They demonstrate that TLSglobals and PIEglobals perform worst, but we note that all methods measured are within 12 nanoseconds of the baseline (without any privatization technique). This overhead is very small and does not increase based on the number of global variables, code size, or degree of virtualization for any of the methods. Note that the time includes scheduling costs as well, since each time a ULT yields, control returns to the scheduler which then context switches to the next ULT that is ready to execute. TLSglobals requires updating the TLS segment pointer at each context switch, while the rest of the privatization techniques do not rely on any additional work at context switch time, since their global variables are defined relative to the instruction pointer. We do note that PIEglobals includes use of TLSglobals where supported, so it includes the overhead of updating the TLS segment pointer. The other methods should also include TLSglobals support but do not currently. Hence, it is not surprising that TLSglobals and PIEglobals perform worst here, but we deem their minor overhead of nanoseconds acceptably low for our purposes of co-scheduling virtual MPI ranks.

**Privatized variable access overhead**

Another important characteristic of a privatization method is that the time spent accessing a privatized variable should not ideally increase when the variable is privatized. Per-access overheads can cause significant overheads if those variables are referenced in the innermost loops of computation. In order to validate that none of our methods exhibit this overhead, we ran a three dimensional Jacobi solver where all of the variables in the innermost computational loops are privatized. Figure 2.7 shows that indeed there are no hidden costs to accessing privatized variables compared to unprivatized variables. We have seen privatized variable access incur overheads with TLSglobals in the past, but we were not able to replicate it here. We hypothesize that any overhead can be optimized away by compilers when compiling with optimizations as we have.

**Migration overhead**

One of the main benefits of AMPI compared to traditional MPI libraries is its runtime support for dynamic load balancing, without the need for intrusive application code changes. The efficiency of dynamic load balancing depends in part on the cost of migrating ranks in AMPI. Since PIPglobals and FSglobals do not support migration at all, they are not shown here. Ideally privatization would only increase the migration time proportionally to the cost of communicating the size of the privatized variables. With PIEglobals, however, we

must communicate not only the globals themselves but the entire code segments as well. Of course the cost of migration then depends on the code size. Figure 2.8 shows results for ADCIRC, a production simulation code of approximately 50,000 source lines of code (described below), which has code size of approximately 14 MB that must be additionally migrated under PIEglobals but not TLSglobals. For reference, our Jacobi-3D standalone benchmark is around 100 lines of code and has a PIEglobals code segment size of 3 MB. Accordingly, as the (heap) memory per rank increases from 1 MB to 100 MB the proportional impact of PIEglobals on migration time decreases since the code segment consumes less of the rank's memory proportionally. This migration cost could potentially limit performance for fine-grained applications or when strong scaling with dynamic load balancing, but since dynamic rank migration is, in practice, relatively infrequent in applications using dynamic load balancing, we consider this cost high but acceptable. Furthermore, we discuss ideas for minimizing the migration cost in our future work section.

**Instruction cache misses**

Another concern we had when implementing our three new methods was that the code duplication would result in unnecessary instruction cache misses. This could potentially affect the performance of all code, not just privatized variables, and slow down the entire application's execution. We used the PAPI performance monitoring library [38] to track instruction cache misses, but found the results surprising: on Bridges2 PIEglobals had 22% fewer L1 instruction cache misses than TLSglobals on our a Jacobi-3D benchmark. This was unexpected, so we ran the same benchmark on TACC's Stampede2 supercomputer, using its Intel Xeon Ice Lake nodes, and there TLSglobals had 15% fewer L1 instruction cache misses. Consequently, we are unable to draw a strong conclusion from PAPI counters on the instruction cache behavior of PIEglobals at this time– more investigation is needed, although application results suggest there is no significant overhead here.

**Application demonstration: ADCIRC**

We also looked at overall execution time of a production application on PIEglobals. Demonstrating PIEglobals on a full-scale application code is important because the size of the code segments can increase the memory footprint and migration times as we have seen. In order to validate the technique, it must be applicable to large legacy codebases. Of our three novel methods, we only consider PIEglobals production-worthy because of its support for dynamic rank migration.

Table 2.2: ADCIRC speedup of best performing virtualization ratio over OpenMPI 4.0.2.

| Cores | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|
| Speedup % | 12 | 58 | 78 | 70 | 43 | 24 | 18 |

ADCIRC is a Fortran90 MPI code used to simulate storm surge flooding in real-time. It is used by the US Army Corp of Engineers, the Federal Emergency Management Agency, and the National Oceanic and Atmospheric Administration to predict the surge of rising ocean waters over floodplains, through low-lying marshes, and into communities during natural disasters such as hurricanes [39].

The ADCIRC code base originally contained hundred of mutable global variables across nearly 50,000 source lines of code. Privatizing all global state manually would be cumbersome, and the code is used on many different systems by users of varying degrees of HPC expertise, meaning requiring modifications to compilers or other system components would be burdensome to maintain and package. PIEglobals addresses these concerns with its portability and ease of use.

To validate PIEglobal's performance, we ran ADCIRC on Bridges2 with varying degrees of virtualization and with load balancing. Dynamic load balancing is particularly effective for ADCIRC since the computationally intensive parts of the domain follow the flow of water as it spreads over and around obstacles in its path. For dry areas, there is little to no computational work. We use the built-in GreedyRefineLB load rebalancing strategy and run with different levels of overdecomposition, and compare against OpenMPI 4.0.2.

We note that PIEglobals successfully privatizes this large Fortran code base with hundreds of global variables, and it does so efficiently and portably while supporting dynamic rank migration. The overall result is that ADCIRC is able to perform between 78% and 12% better than the baseline without virtualization and load balancing thanks to PIEglobals. Even at the limits of strong scaling for this input case, where communication tends to dominate performance, we see an 18% improvement, and we expect more tuning of load balancing frequency and strategy can yield greater speedups as well.

### Application demonstration: LAMMPS

LAMMPS is a widely used production molecular dynamics simulation code developed at Sandia National Laboratories. It supports all kinds of different materials modelling simulations ranging from atomic to macroscopic systems in two or three dimensions with boundary

conditions. It is written primarily in C++ and MPI, and scales from laptops to the largest supercomputers in the world. LAMMPS is mostly thread-safe, having been made thread-safe to enable calling it as a library from multithreaded applications, although it still contains some mutable static and global variables. We have previously been able to tag these unsafe variable declarations using TLSglobals, but to validate PIEglobals and compare its performance we have also run LAMMPS on PIEglobals.

We used the USER-MESO user package of LAMMPS to perform simulations of oil/shale gas recovery. This simulation uses the many-body dissipative particle dynamics numerical method, which combined with the complex geometry, highly non-uniform spatial distribution of particles, and non-uniform inter-particle force/potential calculations cause this type of simulation to exhibit highly dynamic load imbalance [40]. LAMMPS includes its own support for dynamic load balancing through repartitioning, but its built-in rebalancer is far from effective for such problems with complex geometries. LAMMPS estimates the computational load of each particle and tries to dynamically repartition the domain using a recursive bisection method. AMPI, on the other hand, instruments execution to measure the actual load of each rank, and can rebalance load using different strategies ranging from centralized to fully distributed. The distributed load balancers offer low overhead rebalancing, and we use Charm++'s built-in DistributedLB strategy here.

Figure 2.11 shows the execution time of LAMMPS for various configurations. What we see is that PIEglobals is able to correctly virtualize LAMMPS and provide dynamic rank migration on par with TLSglobals. PIEglobals is within 2% of TLSglobals for all runs, and it does not require the hours of development time required to manually find and tag all mutable global variable declarations with the thread_local attribute. Additionally, AMPI's built-in load balancing is able to outperform LAMMPS's Recursive Coordinate Bisection rebalancer that has been developed in concert with the LAMMPS code. Further work could pass knowledge from LAMMPS into AMPI's load balancing framework and a custom strategy could be developed for this user package or for LAMMPS more generally.

## 2.3 CONCLUSION

With the emergence of exascale class systems and cloud computing platforms, HPC application developers are facing a variety of challenges in evolving their codes forward to new levels of performance and simulation capabilities, all while ensuring correctness and maintainability. At the same time, task-based programming models are growing in appeal with their automated scheduling capabilities, asynchronous data movement support, and dynamic resource management features. However, since the investment in production software

Table 2.3: Summary of privatization methods and their features, including our three novel runtime methods.

| Method | Automation | Portability | SMP Mode | Migration |
|---|---|---|---|---|
| Refactoring | Poor | Good | Yes | Yes |
| Photran | Fortran-only | Good | Yes | Yes |
| Swapglobals | No static vars | Linker-specific | No | Yes |
| TLSglobals | Mediocre | Compiler-specific | Yes | Yes |
| *-fmpc-privatize* | Good | Compiler-specific | Yes | Possible |
| PIPglobals | Good | Requires glibc extension | Limited w/o patched glibc | No |
| FSglobals | Good | Shared FS needed | Yes | No |
| PIEglobals | Good | Implemented w/ glibc extension | Yes | Yes |

is large– often sustained over decades– the prospect of rewriting it for a new programming model can be daunting. Virtualization of existing codes is one approach to facilitate this transition, with fully automatic privatization being the ideal method of accomplishing it.

In this chapter we defined the process virtualization issue, motivated why automation is required, and summarized the current state of the art approaches to code privatization, before detailing our three new runtime methods and evaluating them for performance. We believe that one of our new methods, PIEglobals, is the best privatization method developed yet in terms of portability across different architectures and compilers, applicability to both C/C++ and Fortran codes, runtime performance, and support for runtime migration of virtualized entities. It enables running legacy applications that we could not practically virtualize before on top of AMPI for its dynamic runtime support. Of course, for a particular runtime, application, and execution environment the importance of these criteria will be weighed differently. We place particular importance on the degree of automation, the amount of developer effort and expertise needed to apply it, support for migratability, portability across popular systems, and performance, especially minimizing context switch overhead and privatized variable access costs.

For future work, we plan on continuing to validate and test PIEglobals against production application codebases. We plan on exploring the use of dynamic binary instrumentation tools for scanning heap allocated static objects at startup for pointers that need updating to the privatized code segment. We also plan on adding support for Mac OS and on investigating memory optimizations. In particular, we are looking into reducing the code bloat issue of memory usage in PIEglobals by either skipping the memory copy and using mmap directly for each virtual rank or by mapping the code segments into virtual memory from a single file descriptor using *mmap*. Both of these would require deeper changes to Isomalloc and

AMPI's startup process. Further, we could potentially reduce its migration memory overhead by changing Isomalloc to only migrate segments of code that differ across different ranks. Having a way to detect read-only (or written-once) variables, possibly through compiler support, and not duplicate them could reduce memory footprint per-rank as well. In all, we see this as the most promising process virtualization technique yet developed in terms of portability, performance, and support for arbitrary levels of overdecomposition and dynamic rank migration.

```
// In ampi_functions.h:
AMPI_FUNC (int, MPI_Send, const void *msg, int count, MPI_Datatype type, \
           int dest, int tag, MPI_Comm comm)

// In mpi.h:
#ifdef AMPI_USE_FUNCPTR
 #define AMPI_FUNC(return_type, func_name, ...) \
 extern return_type (* func_name)(__VA_ARGS__);
#else
 #define AMPI_FUNC(return_type, func_name, ...) \
 extern return_type func_name(__VA_ARGS__);
#endif
#include "ampi_functions.h"

// In ampi_funcptr.h:
struct AMPI_FuncPtr_Transport {
 #define AMPI_FUNC(return_type, func_name, ...) \
 return_type (* func_name)(__VA_ARGS__);
 #include "ampi_functions.h"
};

// In ampi_funcptr_loader.C (linked with AMPI runtime):
void AMPI_FuncPtr_Pack (struct AMPI_FuncPtr_Transport *x) {
 #define AMPI_FUNC(return_type, func_name, ...) \
 x->func_name = func_name;
 #include "ampi_functions.h"
}

// In ampi_funcptr_shim.C (linked with MPI user program):
void AMPI_FuncPtr_Unpack (struct AMPI_FuncPtr_Transport *x) {
 #define AMPI_FUNC(return_type, func_name, ...) \
 func_name = x->func_name;
 #include "ampi_functions.h"
}
```

Figure 2.4: AMPI's headers had to be refactored as a function pointer shim library to avoid privatizing its runtime along with the user application code.

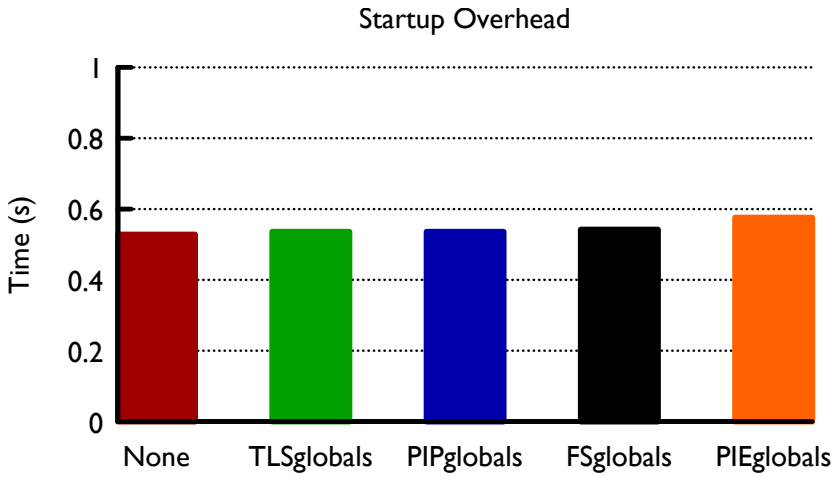



Figure 2.5: Startup or initialization overhead for different privatization methods with 8x virtualization (lower is better).

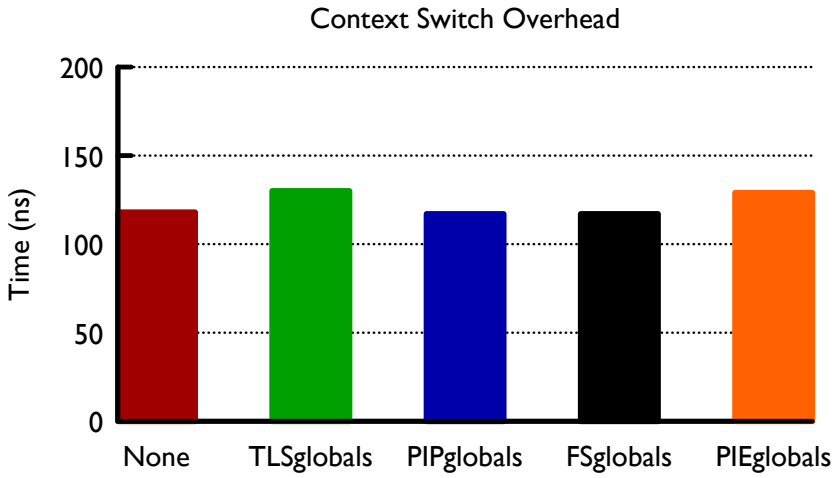



Figure 2.6: User-level thread context switch time in nanoseconds for each privatization method (lower is better).

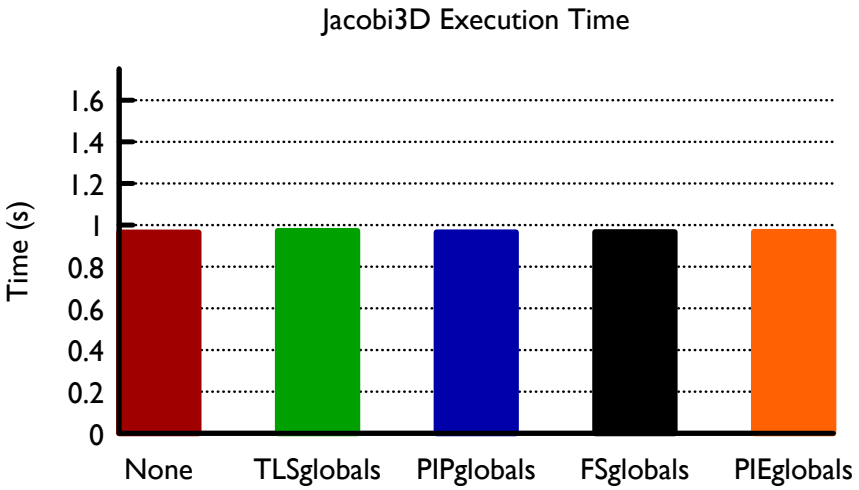



Figure 2.7: Execution time of Jacobi-3D where all variables accessed in the innermost loop are privatized global variables (lower is better).

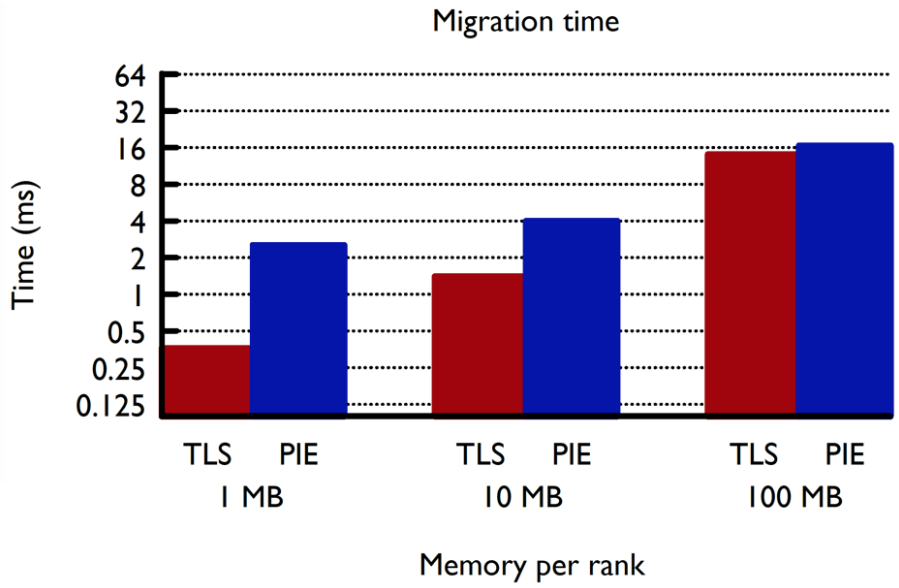



Figure 2.8: Migration time of virtual ranks with different sizes of memory allocated, comparing TLSglobals to PIEglobals (lower is better).

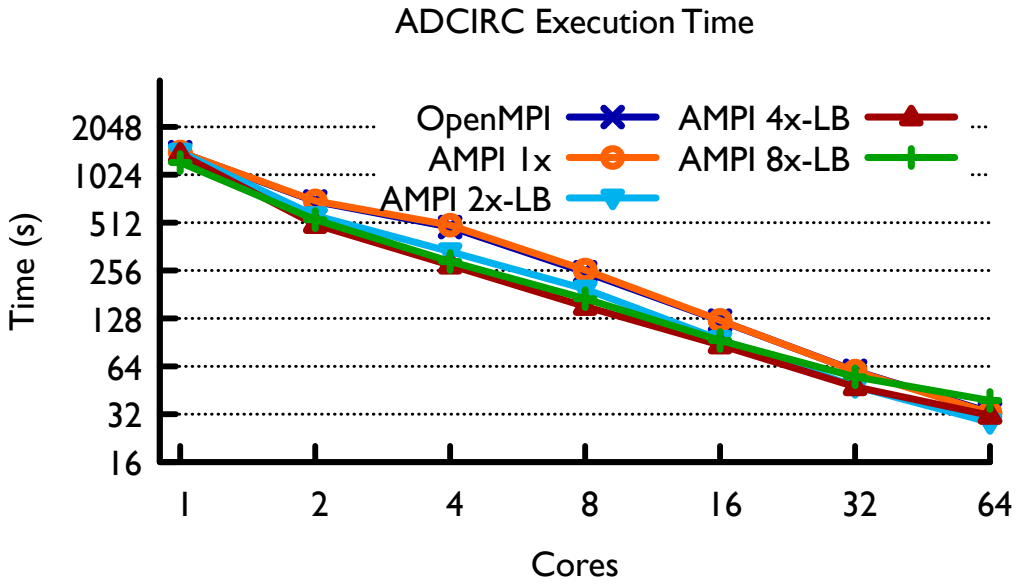



Figure 2.9: Strong scaling execution time for ADCIRC with varying degrees of virtualization and with dynamic load balancing. Lower is better.

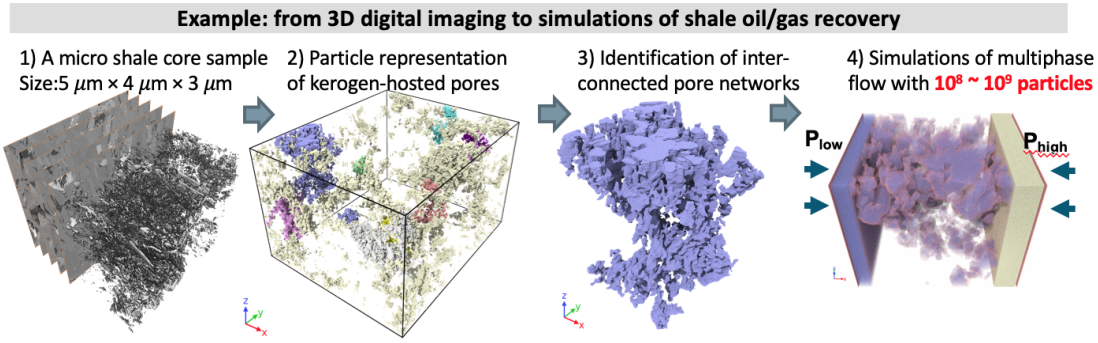



Figure 2.10: This shows a 3D image of a micro shale core sample on the left compared to different LAMMPS particle simulations to the right. The rightmost case with multiphase flow is what we run on AMPI with dynamic load balancing.

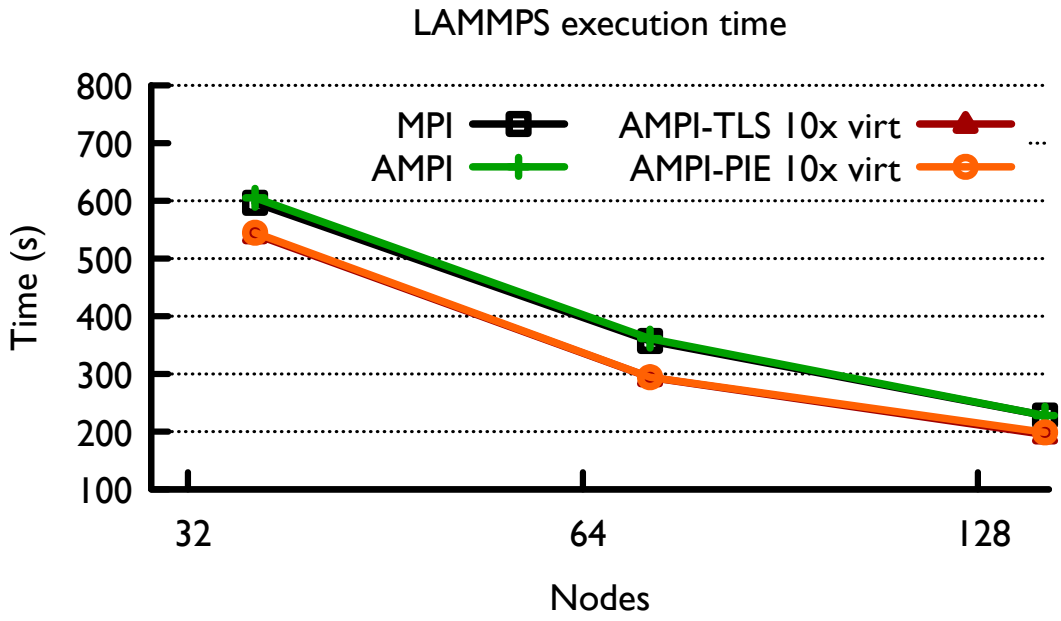



Figure 2.11: LAMMPS strong scaling results for a system with 3.75 million particles simulating fluid flow through shale pores. Lower is better.

# CHAPTER 3: MEMORY USAGE OPTIMIZATIONS

## 3.1 BACKGROUND

While node architectures are becoming increasingly parallel, total memory per node is increasing at a much slower rate. This gives applications less memory per core, and is one of the apparent downsides to an MPI-everywhere approach. Whereas message passing generally operates between ranks with their own separate address spaces, OpenMP and other shared memory parallel programming models operate directly on shared memory. Given AMPI's virtualization of ranks, reducing the amount of memory per rank, both inside AMPI and at the application level, becomes important. We propose to study the memory overheads of AMPI in detail, to bound its per-rank memory usage for scalable applications, to minimize AMPI's internal memory consumption, and to limit peak memory usage during communication and load balancing phases.

Studies have been conducted in the past on the scalability of the MPI standard itself and of MPI implementations of certain features [41]. MPI libraries typically set the eager/rendezvous protocol threshold as a trade-off in terms of not only latency and bandwidth but also memory usage and synchronization costs. Goodell et al identified parts of the MPICH2 library which had memory scaling linearly with the number of ranks [42]. MPI_Groups have been the target of various memory-saving optimizations, since if unoptimized they can consume $O(P)$ memory at each rank [43]. The MPI forum has also adapted the standard to address unscalable memory usage in certain features. This includes the addition of the distributed graph virtual topology, effectively replacing the unscalable graph topology, as well as the creation of shared memory windows. Shared memory windows can be used by applications to minimize the per-rank memory footprint of its own data structures [44]. Because this feature is relatively new to the standard, however, it is not yet widely used in applications and libraries. Moreover, implementations do not effectively use shared memory to its fullest extent for their own internal storage among ranks co-located on a node.

In this chapter, we first study the memory usage of an AMPI application and the runtime in order to motivate several memory optimizations. We categorize memory usage in terms of application versus runtime usage, transient versus persistent lifetimes, read-only versus mutable state, and by its usage for communication, migration, etc.

We inspect a three-dimensional stencil code called Jacobi-3D and plot the memory usage across four iterations, including one round of dynamic load balancing, on one node, both with and without virtualization. Figure 3.1 shows the result for two iterations, load re-

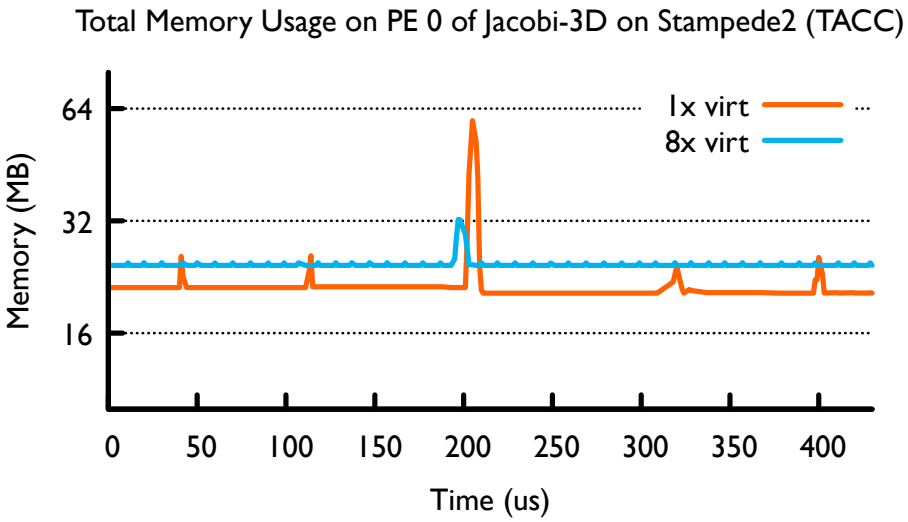



Figure 3.1: Memory usage on PE 0 of a three dimensional stencil benchmark on the Skylake partition of Stampede2 (TACC) both without and with 8x rank virtualization.

balancing, and then two more iterations. We notice the difference between runs with and without virtualization, and the various peaks of memory usage. This implementation uses persistently allocated message buffers for the halo exchange, so there is no dynamic memory allocation coming from the application during the plotted runtime. Without virtualization, each rank owns a 128 x 128 x 128 subdomain of the overall volume. With 8x virtualization, each owns a 64 x 64 x 64 subdomain. The code implements a 7-point stencil with periodic boundary conditions so each rank communicates with 6 neighbors. Without virtualization each rank allocates roughly 19.1 MB of total memory, 1.6 MB of which are message buffers. With 8x virtualization, this scales to 2.7 MB total memory per rank, 393.2 KB of which is in message buffers. In total memory per PE, this means 8x virtualization allocates 21.5 MB of application memory, compared to 19.1 MB of application memory without virtualization. However, the runtime also allocates memory per rank, as in for each user-level thread stack and for internal consumption such as buffering messages or storing information about MPI objects such as communicators. By default, AMPI allocates 1 MB user-level thread stacks. Thread stacks are statically allocated at runtime, and must be large enough to avoid the application overflowing it and failing. For many applications this default is more than enough, with some (such as our Jacobi benchmark) able to get by with a stack size as low as 64 KB. However, we have also seen legacy Fortran applications that extensively use AUTOMATIC arrays and which run fastest on the Intel compiler using its -no-heap-arrays option. This option allocates all AUTOMATIC arrays on the stack rather than the heap. Such applications can require 100s of MB of stack space per rank, which can obviously greatly limit the

degree of virtualization possible on most systems.

Aside from the memory footprint differences, we also notice differences in the dynamic memory usage over time. This memory usage comes from AMPI internally, for communication and for migration. Without virtualization, there are 5 peaks: the first two correspond to the halo exchange communication phase in each Jacobi iteration, the third to dynamic load balancing, and the last two to more halo exchanges. We observe that the communication phase spikes are smaller than the dynamic load balancing ones, and that the peaks are higher proportionally without virtualization than they are with 8x virtualization. This is due to two factors: the size of messages are decreased and the spreading of communication over the timestep. Communication spreading is a natural consequence of rank-based overdecomposition, in that ranks are co-scheduled in a fashion where one rank will perform its local computational work and then post its communication before blocking on completion of those requests (in an MPI_Waitall), at which time another rank will be scheduled to do the same. Thus, each rank will post its message buffers at different times throughout the overall timestep.

We also notice that while the floor of memory usage is higher with virtualization, the peaks during both communication phases and load balancing are both lower for this application. The memory usage during load balancing is lower with virtualization because each rank's memory footprint is less, meaning migration of one rank requires less memory. In either case, the high watermark of memory utilization is determined by rank migration, where it rises up to nearly 3x the amount of steady state memory usage. Ideally we can achieve a tighter bound on the maximum memory usage needed to run an AMPI application with load balancing and we can minimize the per-rank memory footprint in order to enable applications to run with data sizes closer to the total memory per node.

## 3.2 COMMUNICATION MEMORY USAGE

As seen in the Jacobi-3D memory utilization plot, communication– for application messaging and rank migration– is a limiting factor of the size at which users run applications on AMPI. We include rank migration in communication generally because rank migration is communication of a rank's data between PEs or nodes at runtime, using the same underlying communication infrastructure as we do for point-to-point communication. We have observed that the overall memory size that a user can run an AMPI application is limited by the use of dynamic load balancing and by communication on top of the memory footprint per rank, with virtualization increasing the memory footprint. Minimizing the spikes in memory usage during communication and migration phases of applications, then, will allow users to run

larger problem sizes. In this section we motivate a new model of messaging in Charm++ which enables libraries, such as AMPI, to perform in-place communication and migration of user-owned data.

Historically, we note that Charm++, the tasking runtime on which AMPI is implemented, has been used primarily for dynamic applications which overdecompose the domain into many more work and data units, chares, than execution units. Chares communicate asynchronously with each other while being migratable across PEs or nodes during execution. This overdecomposition tends to lead to the creation of more frequent, smaller messages than without overdecomposition, as we have seen in the Jacobi-3D example in Figure 3.1. Charm++'s messaging semantics can be seen as a product of this development lineage.

In Charm++, interactions between chares are performed using entry method invocations, which are carried out traditionally using two messaging models: a) Parameter Marshalling, and b) Custom Messages. Parameter marshalling is used when parameters are passed in an entry method invocation and a message is created internally by the runtime system on behalf of the user. On the sender side, the marshalling of these parameters requires the runtime system to copy individual parameters passed by the user into a single contiguous buffer which is sent across the network as a message. This is done to ensure safe reuse or freeing of the passed parameters after the entry method call. On the receiver side, the parameters are unmarshalled out of the message and passed to the entry method. This allows the runtime system to use pointers directly to contiguous parameters in the message and pass them to the entry method without copying them. To ensure safe memory management of this message, i.e. to avoid a potential memory leak, the message is freed by the runtime system after the entry method completes. For this reason, these parameters have a lifetime only until the end of the scope of the entry method. In order to use the received data beyond this scope, the user must copy the data into their own data structure. Therefore, two copies are required for sending data from a chare to another remote chare: one at the sender side and another at the receiver side.

The other messaging model is to use Custom messages. Custom messages are data structures that inherit from a base message data structure and encapsulate all the parameters required by an entry method. The key difference between parameter marshalling and custom messages is with regard to ownership of the buffer. On the sender side, after invoking an entry method with a custom message, the runtime system takes ownership of the message and the user cannot access the encapsulated parameters or reuse that buffer. Additionally, unlike parameter marshalling, the semantics of not allowing buffer reuse allow the runtime system to avoid making copies on the sender side. On the receiver side, the ownership of the received message is given to the user. This allows the user to directly use the received

message as if it were the user's data structure without requiring an additional copy. However, if the user forwards the received message to another chare, the message ownership is again handed back to the runtime system and the buffer is inaccessible. For iterative applications, using this model requires that the user make a new allocation and copy of the custom message for each iteration, which in turn adds to the cost of communication.

These traditional messaging models in Charm++ have their advantages and limitations: they allow applications that can use Charm++ message objects as their buffers to communicate via ownership passing with-in a node, and the message ownership model lends to a highly asynchronous implementation. However, neither of these messaging interfaces allow for communicating data "in-place" generally. The effects of this design can be seen in AMPI and other libraries that wish to operate on user-owned buffers, which must incur extra allocations and copies that are implicit in the model, resulting in higher latency and increased memory footprint. We propose and develop a new zero copy messaging model in Charm++ to address these messaging latency and memory consumption issues caused by the existing messaging semantics by building on top of RDMA support in modern HPC networks. This will facilitate reuse of user buffers, and eliminate the need to make additional large allocations and copies while still taking advantage of the asynchronous execution model at the heart of Charm++ and AMPI.

Charm++ is implemented on top of of Converse, a lower level runtime that supports one-sided active message exchanges between PEs, and LRTS, a low-level communication runtime that implements communication over a variety of native networking APIs, such as Cray uGNI, IBM PAMI, Infiniband, OFI, UCX, MPI, and Ethernet. The goals for our new API are to enable in-place communication of user-owned buffers with low latency and memory overhead, while maintaining asynchronous execution opportunities and fitting into the existing Charm++ programming model as much as possible. Based on this, we created multiple APIs for communication, varying from low level put or get operations on buffers registered with the runtime to entry method based APIs that hide the details of buffer registration from the user. We support point-to-point as well as collective routines, in addition to using these new APIs for migration and other functionality inside the runtime. We then make use of these in AMPI's implementation to improve the performance of MPI applications.

We first describe our lower-level point-to-point communication interface before discussing a higher-level interface, as well as collectives and migration support. We refer to these APIs as "zero copy" because they allow direct communication between user-owned buffers with zero extra copies in-between.

### 3.2.1 Direct API

The zero copy Direct API is built around CkNcpyBuffer objects which describe a contiguous memory region and enable registering or pinning that memory. This low-level API allows users to explicitly invoke a standard set of methods (put or get) on the CkNcpyBuffer objects to avoid both sender and receiver side copies for point-to-point messages. It is the Charm++ equivalent of the Converse level API[45], where Charm++ entry methods are used instead of Converse message handlers. To use the Direct API, the user creates a local CkNcpyBuffer object and sends it to the other participating chare in a remote entry method invocation as illustrated in Figure 3.2.

```
// Inside an entry method...
CkCallback srcCb(CkIndex_Ping1::sourceDone(), thisProxy[thisIndex]);
CkNcpyBuffer source(myBuffer, size * sizeof(int), srcCb);

// Invoke a remote method
// passing my CkNcpyBuffer object
arrProxy[1].recvNcpySrcObj(source);
```

Figure 3.2: Direct API object creation and handover

```
void recvNcpySrcObj(CkNcpyBuffer source) {
  CkCallback destCb(CkIndex_Ping1::destDone(), thisProxy[thisIndex]);
  CkNcpyBuffer dest(myBuffer, size * sizeof(int), destCb);

  // Call get on local dest object
  // passing the received source object
  dest.get(source);
}
```

Figure 3.3: Direct API performing Get operation

On receiving the remote CkNcpyBuffer object, the other participating chare creates a local CkNcpyBuffer object and calls the standard get method on it by passing the remote object to perform a zero copy read operation as shown in Figure 3.3. After the completion of the get operation, the callbacks specified in both the objects are invoked. Inside the source callback, sourceDone, the source buffer can be safely modified or freed. Similarly, inside the destination callback, destDone, the user is guaranteed that the data transfer into the destination buffer is complete and the user can begin operating on the newly received data. These callback functions are illustrated in Figure 3.4.

Using this API, after the preliminary handover of one of the CkNcpyBuffer objects to the other end, the user can exploit the persistent nature of iterative applications to perform zero

```
void sourceDone(CkDataMsg *msg) {
  delete myBuffer;  // free the buffer
}

void destDone(CkDataMsg *msg) {
  // received data, begin computing
  computeValues();
}
```

Figure 3.4: Direct API source and destination callbacks

copy operations using the same buffer information objects across iteration boundaries. The implementation of the Charm++ Direct API is an extension of the Converse zero copy API where the same functionality is used[45] and Charm++ callbacks are supported instead of handler functions because of the use of CkNcpyBuffer objects. These callbacks are invoked inline if the callback function has been marked as such, and the callback object can store a reference number or other data which gets passed into the CkDataMsg that is received after invocation of that callback. This enables users to send a tag along with buffer metadata information, and to retrieve that tag after completion in order to know which buffer has been completed.

### 3.2.2 Entry Method API

The zero copy Entry Method API extends the capability of the existing entry methods in Charm++ with slight modifications in order to send and receive buffers without copies. It supports both point-to-point and optimized broadcast operations and allows users to send and receive previously received copy based buffers as special "zero copy" buffers.

To send a buffer using the Entry Method API, the user is required to annotate the buffer parameter as nocopypost in the entry method declaration in the .ci charm interface file as shown in Figure 3.5.

```
// .ci declaration
entry void recvBuffer(int size, nocopypost int buffer[size]);
```

Figure 3.5: Entry Method API: Method Declaration

On the sender side, the user needs to wrap the buffer and an optional callback object inside a CkSendBuffer wrapper and invoke the remote entry method as shown in Figure 3.6. Figure 3.6 illustrates a point-to-point invocation. A broadcast call can be made in a similar

manner by using the entire chare array proxy arrProxy instead of a chare array element proxy like arrProxy[1].

```
// Create a callback
CkCallback srcCb(CkIndex_Ping1::sourceDone(), thisProxy[thisIndex]);

// Invoke the remote method
// passing myBuffer in CkSendBuffer
arrProxy[1].recvBuffer(size, CkSendBuffer(myBuffer, srcCb));
```

Figure 3.6: Entry Method API: Remote Invocation

On the receiver side, the user is required to have two overloaded definitions of the same entry method. The first definition, called the Post Entry Method uses the same argument list with an additional CkNcpyBufferPost $*$ parameter. The Post Entry Method is invoked first, allowing the user to match the sender/source buffers with corresponding receiver/destination buffer using tags. This is done using the CkMatchBuffer call where the user supplies the CkNcpyBufferPost $*$ pointer along with the index of the operation and a user provided integer tag. The index corresponds to the index of the nocopypost parameter among multiple nocopypost parameters, i.e. the first nocopypost parameter will have an index of 0, the next will be 1 and so on. This is illustrated in Figure 3.7.

```
// post entry method
void recvBuffer(int size, int *buffer, CkNcpyBufferPost *post) {
  // Match 0th source buffer with tag1
  CkMatchBuffer(ncpyPost, 0, tag1);
}
```

Figure 3.7: Entry Method API: CkMatchBuffer call inside Post Entry Method

For every CkMatchBuffer call with a tag, there should be corresponding CkPostBuffer call with the same tag that is used to post the receiver/destination buffer. This call can be made from any entry method: before, after, or inside the Post Entry Method. This is similar to an MPI_Irecv call that is made when the receiver is ready to receive a buffer. This is illustrated in Figure 3.8.

```
// in some other entry method...
// ready to post buffer
CkPostBuffer(myBuffer, mySize, tag1);
```

Figure 3.8: Entry Method API: CkPostBuffer call

After the execution of a CkPostBuffer (occurring from any entry method) and the corresponding CkMatchBuffer (executed inside the Post Entry Method), the runtime system performs the zero copy operation. On completion of all the zero copy operations of a particular entry method, the actual entry method is invoked. The actual entry method is the other overloaded definition of the same entry method, without the CkNcpyBufferPost * parameter. Inside the actual entry method, it is guaranteed that all the posted buffers have received the data from the send buffers. This is illustrated in Figure 3.9. Similar to the Direct API, the source/sender callback is invoked to signal that the buffer is ready to be reused or freed.

```
// actual entry method
void recvBuffer(int size, int *buffer) {
  computeValues();  // data ready in buffer
}
```

Figure 3.9: Entry Method API: Regular Entry Method

The implementation of the Entry Method API primarily relies on the tag matching functionality and the source-to-source code generation that uses the .ci file. On the sender side, the user invoked CkSendBuffer is converted to CkNcpyBuffer using a simple macro. The implementation uses the source-to-source code generator to generate the marshalling code on the sender side and the unmarshalling code on the receiver side. On the sender side, this generated marshalling code for the Entry Method API copies the smaller CkNcpyBuffer information object into the message, as opposed to copying the entire buffer (which is much larger in size) as done in the case of the regular messaging API. On the arrival of the message, the generated unmarshalling code first executes the Post Entry Method allowing the user to match the receiver buffer with a tag. The CkMatchBuffer call uses the tag to check if the receiver has already posted a buffer with the same tag. This is done by searching a hash table postedBuffMap that is used to store any posted receiver buffer information with the tag as the key when the user calls CkPostBuffer. If the receiver buffer has already been posted, a Get operation is issued by internally calling LrtsIssueRget. If the receiver buffer is not posted, the matching source buffer information is stored in another hash table matchedBuffMap with the tag as the key. On the user calling CkPostBuffer at a later point in time, matchedBuffMap is searched to find the source buffer information and a Get operation is issued in the same manner. After the Get operation is completed, the source callback method is invoked and the received message is enqueued again in order to execute the actual entry method to signal to the receiver that the data transfer is complete. This design allows us to flexibly tag match and pair any CkMatchBuffer and CkPostBuffer calls. Currently, the

hash table is managed on a per-PE basis to avoid any locking overheads, but in Chapter 4 we discuss changes to AMPI that could be brought down into this implementation in order to achieve greater concurrency and asynchrony in the implementation.

For implementing zero copy transfers in a broadcast call, we use a spanning tree, where each node of the spanning tree represents a process. The spanning tree is rooted at the process that contains the source buffer and all the other nodes represent the recipient processes. Get calls are performed in a top-down order, where each parent node serves as the source process for Get calls made by its immediate children. The root node's source callback is invoked when the first level of child nodes have completed their Get calls, expediting completion back up the tree. Similarly, the entry method on each non-root parent node is invoked when its immediate child nodes have completed their Get calls. The entire broadcast call is complete when all the leaf nodes of the spanning tree have received the source buffer.

### 3.2.3 Pup Buffer API for Migrations

Aside from inter-task communication, another major source of communication in task-based programs is that of moving the persistent data owned by migratable objects, be they AMPI ranks or Charm++ chares. Charm++ provides a Pack-UnPack (pup) API that enables users to write a single simple routine per chare class that handles both sides of the migration process. For each migration, the size of the chare's data must be assessed, a message must be allocated to that size, the chare's state copied into the message, and then the message transferred and unpacked on the destination PE. As a consequence of the existing pup API, during migrations the memory usage of a Charm++ program can transiently spike if many chares are relocating simultaneously, as is common in greedy rebalancing algorithms. Consequently, we sought to use the zero copy infrastructure we have built for the purpose of efficient migrations. The current pup API proved to be limiting in terms of not separating the allocation and transfer of data from the completion of the transfer. Thus, we added a new pup API called "pup_buffer" which operates asynchronously on the unpacking side of the protocol. This allows users to mix regular pup and pup_buffer objects in the same chare, using only pup_buffer for large arrays of data.

To use this API, the user has to call pup_buffer on the PUP::er object inside the chare's pup method. The pup method is the standard method that is written with a PUP::er object as an argument. This method is called by the runtime system when the chare is about to be migrated (for sizing and packing) or when the chare has just been migrated (for unpacking). The pup_buffer method as shown in Figure 3.10 takes two arguments: buffer pointer and size. Optionally, the user can also pass a custom allocator and deallocator in this call. AMPI

hides all of these details of migration from the user, instead linking its Isomalloc memory allocator into the application. These custom allocation hooks are used by Isomalloc to map the memory into a rank's reserved slot in the global virtual memory space, as illustrated in Figure 1.5.

```
// standard pup routine
void pup(PUP::er &p) {
  p | iteration; // pup using the copy-based scheme
  p.pup_buffer(buffer, size);
}
```

Figure 3.10: Pup Buffer API

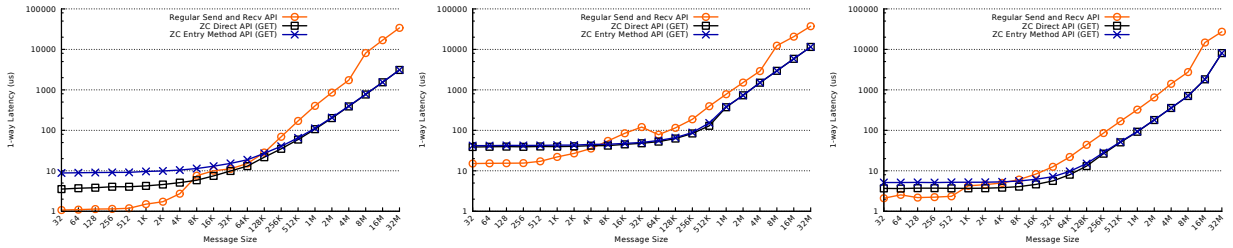
Our implementation uses a similar approach to the Entry Method API, where instead of packing the entire buffer, a CmiNcpyBuffer object is created out of it and packed instead. Similarly, on the receiver side, the source buffer's CmiNcpyBuffer object is unpacked and a Get is invoked into the newly allocated buffer. CmiNcpyBuffer is used over CkNcpyBuffer because there is no use of the additional CkCallback object added to CkNcpyBuffer. Unlike the copy based pup API, since the pup_buffer API executes asynchronously, there is no guarantee of the data transfer being complete during unpacking. To avoid entry methods of a chare with an active pup_buffer call executing on incomplete data, we buffer messages targeted to this chare in the runtime system until the issued Get completes. On the completion of the Get call, all buffered messages are released to execute the appropriate entry methods. The source buffer on the previous home PE (where the chare migrated from) is deallocated.

### 3.2.4 Performance Evaluation

Table 3.1: Benchmarking machines and their configuration

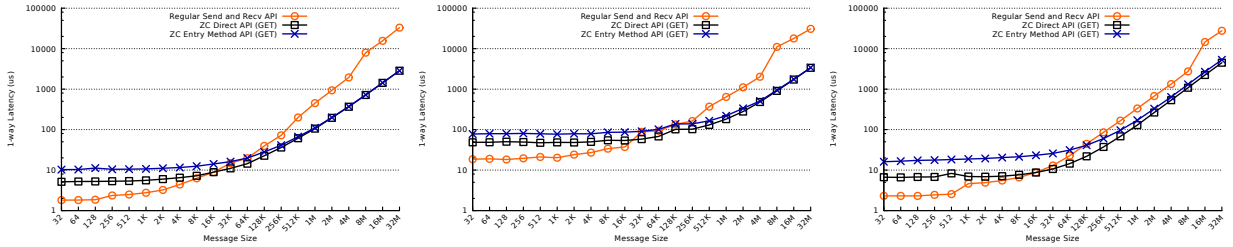
| Machine | Cores/Node | Memory/Node | Network | Charm Build |
|---|---|---|---|---|
| iForge | 40 | 192 GB | Infiniband | ucx |
| Stampede2 | 68 | 96 GB | Omni-Path | ofi |
| Cori | 32 | 128 GB | Aries | gni |
| Quartz | 36 | 128 GB | Omni-Path | ofi |

We use four HPC machines for all our performance experiments. These details including the Charm build used is summarized in Table 3.1.




(a) iForge (b) Stampede2 (c) Cori

Figure 3.11: Comparison of intra-node latency between regular and zero copy messaging API




(a) iForge (b) Stampede2 (c) Cori

Figure 3.12: Comparison of inter-node latency between regular and zero copy messaging API

To evaluate the performance of our new communication model, we developed two benchmarks that compare the performance of the regular messaging model with the zero copy messaging model.

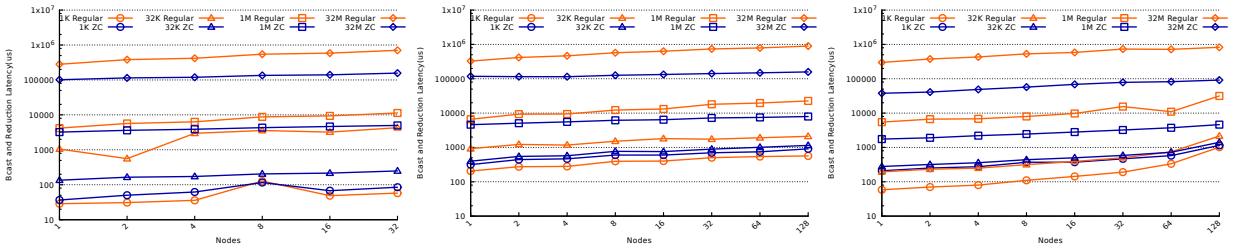
### Point-to-Point Performance

We use a ping-pong benchmark for the evaluation of point-to-point messaging performance in Charm++. The benchmark exchanges messages for a fixed number of iterations (1000 iterations for up to 256 KB and 100 iterations for 512 KB to 32 MB) and measures the one-way messaging latency to send and receive data from user buffers of two chares on two different PEs. The one-way latency is determined by averaging out the total time across all iterations and dividing that value by 2. This entire process is repeated for different message sizes. Using this benchmark, since we aim to determine the time taken for send and receive directly from user buffers, in the Regular API we make an explicit copy from the received message into the user buffer. On the other hand, this is unnecessary for the zero copy API because this direct transfer happens implicitly.

Improvements in intranode and inter-node latency with zero copy Direct API and zero

Table 3.2: Improvement in point-to-point latency with zero copy messaging API.

| Improvement | Metric | Intra-node | | | Inter-node | | |
|---|---|---|---|---|---|---|---|
| | | iForge | Stampede2 | Cori | iForge | Stampede2 | Cori |
| **ZC Direct API** | SpeedUp | 1.2x – 10.9x | 1.3x – 4.2x | 1.15x – 8x | 1.3x – 11.5x | 1.5x – 9.1x | 1.2x – 6.5x |
| | % Improvement | 22% – 90% | 23% – 69% | 13% – 70% | 25% – 9% | 33% – 89% | 18% – 83% |
| | Threshold Size | 8K | 8K | 1K | 32K | 32K | 16K |
| **ZC Entry Method API** | SpeedUp | 1.1x – 10.9x | 1.2x – 4.2x | 1.1x – 8x | 1.4x – 11.5x | 1.2x – 9x | 1.1x – 5.5x |
| | % Improvement | 5% – 90% | 19% – 69% | 9% – 70% | 28% – 91% | 18% – 90% | 8% – 81% |
| | Threshold Size | 128K | 8K | 8K | 128K | 256K | 128K |




(a) iForge (b) Stampede2 (c) Cori

Figure 3.13: Comparison of Broadcast and Reduction Latency between regular and zero copy messaging API

copy Entry Method API on three different machines are illustrated in Figures 3.11 and 3.12. As seen in all the latency plots, for small messages, the regular messaging API performs better than the zero copy API because of the extra memory allocations and copies being inexpensive in comparison to the time taken to send the metadata message for the zero copy API. However, we see that the zero copy API begins to outperform the regular API for medium and large messages, with the improvement increasing with message size. This is because of the metadata message latency remaining constant, whereas the cost of additional allocations and copies increases proportionally with message size.

The small performance difference seen between the Direct API and Entry Method API in all p2p latency plots can be attributed to two additional overheads in the entry method API. First, memory registration and deregistration is performed for every iteration. Second, there is some additional processing which includes tag matching and packing/unpacking. These overheads are not incurred in the Direct API because it only requires memory registration and deregistration once, and separately, there is no requirement for tag matching or packing/unpacking. Cross Memory Attach (CMA) is supported on both Stampede2 and Cori. The zero copy API executions on these machines take advantage of this for intranode transfers and this results into a smaller performance difference between Direct API and

Table 3.3: Improvement in bcast latency with zero copy messaging API.

| **Metric** | **iForge** | **Stampede2** | **Cori** |
|---|---|---|---|
| SpeedUp | 1.3x – 17x | 1.8x – 5.6x | 2.9x – 9.2x |
| % Improvement | 23% – 94% | 30% – 82% | 67% – 89% |

Entry Method API as seen in 3.11b and 3.11c. For these transfers, since registrations are not required, the Entry Method API only incurs the overhead associated with the additional processing. The range of speedups, percentage improvements, and threshold message sizes over which the the two variants of point-to-point zero copy API outperform the regular API are summarized in Table 3.2.

### Broadcast Performance

To evaluate the performance of broadcast operations, we use a ping-all and reduce benchmark written in Charm++. The benchmark measures the latency for a broadcast and reduction across all PEs for different message sizes. The average time for a single broadcast and reduction operation is determined by averaging the total time across many iterations (100 iterations up to 256 KB and 10 iterations for 512 KB to 32 MB). Similar to the ping-pong benchmark, since we aim to determine the time taken for send and receive directly form user buffers, in the Regular API we make an explicit copy from the received message into the user buffer.

Figure 3.13 illustrates the weak scaling performance of the broadcast version of the zero copy Entry Method API over the regular API. The figure plots broadcast and reduction latency for four different message sizes on three different machines. As seen in the plots, the improvement achieved with the zero copy API increases for the larger message sizes. This can be explained with the same analysis conducted for the point-to-point ping-pong experiments i.e. as the message size increases, the cost of the extra allocation and copy increases, making the regular API perform poorly for large messages. Additionally, it is also seen that in most cases, the improvement increases proportionally to the number of nodes or PEs. This indicates that zero copy entry method API scales better than the regular API. On all machines, the regular API performs better for the 1K size but the zero copy API performs better for larger message sizes in most cases as seen in Figure 3.13. Unlike iForge and Stampede2, it can be observed that the regular API performs better than the zero copy API on Cori for a 32K message for up to 32 nodes. We believe that this is primarily due to

the relatively expensive memory registration and deregistration operations on GNI, that are performed for every iteration of the zero copy API. These operations outweigh the benefits of the zero copy API at 32K message size. A similar pattern can be seen in the point-to-point case in Figure 3.12c. The range of speedups and percentage improvements achieved by the zero copy API over the regular API are summarized in Table 3.3.

Internally AMPI now uses the zero copy Direct API and maintains a pin-down cache of registered memory buffers for each rank. We support migrations of virtual ranks during execution, so long as they do not have any pending messages. Normally load balancing is conducted at a synchronization point at the end of timestep, so this is not a problem, though handling migrations leads to complications in the rendezvous protocol.

AMPI now chooses its communication protocol based on both the message size and the expected locality of the receiver from the sender. We say expected because Charm++ uses a distributed location management protocol that does not generally guarantee knowledge of all object's places at a given time on any given PE. It does guarantee eventual delivery of messages, but the receiver may not be where the sender initially thinks it is if it has recently migrated. Consequently, we must handle the case where we expect a receiver is on our same node but has actually migrated away. In this case we choose not to pin the memory upfront for same-node transfers in order to avoid the memory registration cost, and fall back to a slower put-based protocol where the receiver pins its memory and sends back a CkNcpyBuffer object to the sender, who then uses it to perform a put of the data after pinning its own buffer. Otherwise, if the receiver is where the sender expected, the receiver does a get from the sender's buffer to its own. Because Charm++'s distributed location management is generally only ever out of date the first iteration after load balancing, and load balancing is usually infrequent, the slower protocol is rarely used in practice but is necessary for correctness.

Figure 3.14 shows the results of the OSU MPI point-to-point latency benchmark for original AMPI compared to the new one with the zero copy API. The benefit of the zero copy API is seen when we switch from an eager protocol using Charm++ custom messages to a rendezvous one with the Direct API, avoiding unnecessary memory copies. We examine AMPI point-to-point communication more in Chapter 4.

We also modified AMPI to use the pup_buffer API for migrations. AMPI's memory allocator, Isomalloc, ensures that all stack and heap data are migratable by allocating each virtual rank's data from within unique slices of the global virtual memory address space. This ensures that after migrating a virtual rank's data, all pointers remain valid because all memory remains allocated at the same virtual address. We also modified the PUP infrastructure and Isomalloc to avoid copies entirely when ranks migrate within the shared

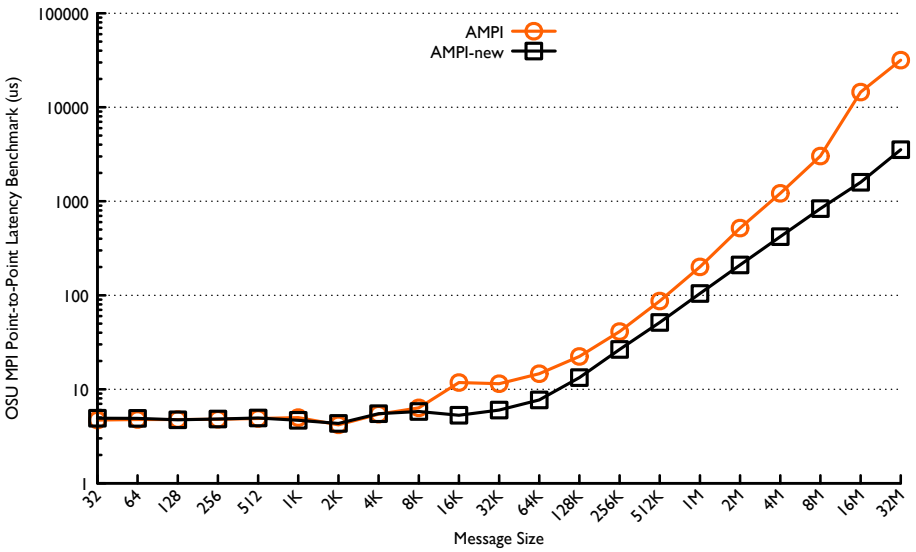



Figure 3.14: OSU MPI Point-to-point latency benchmark on Quartz at LLNL. Lower is better.

address space of a node. This makes intranode migrations cheap, only updating metadata related to Charm++'s location management and transferring Isomalloc metadata.

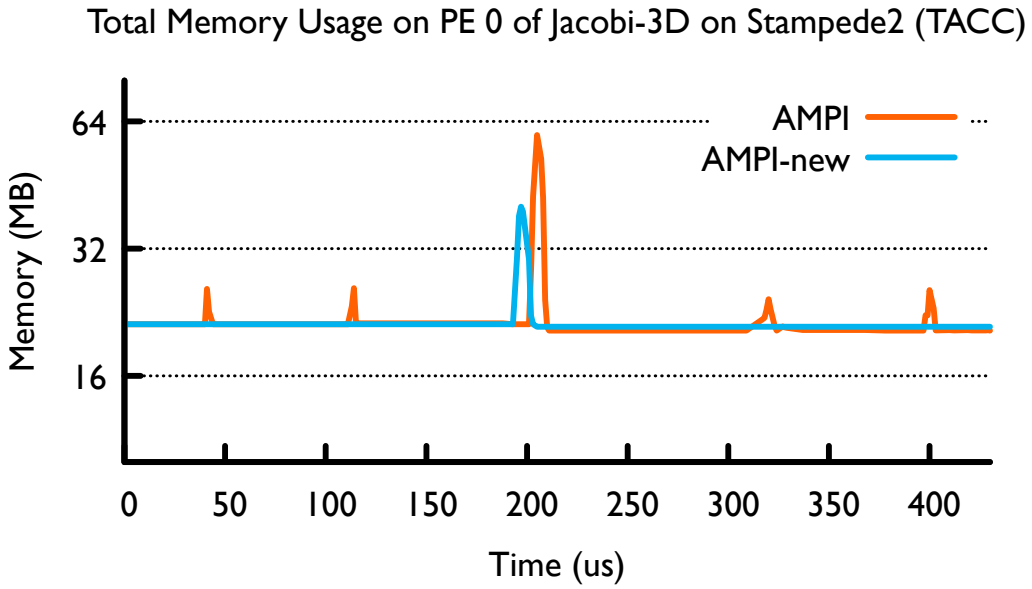



Figure 3.15: Memory usage on PE 0 of a three dimensional stencil benchmark without virtualization where each rank owns a 128 x 128 x 128 subdomain. AMPI-new uses the zero copy direct API for communication and the pup_buffer API for migration. This is run on the Skylake partition of Stampede2 (TACC).

Figure 3.15 shows the the memory usage over time on PE 0 of a three dimensional Jacobi solver run on AMPI with 8x overdecomposition on 8 Skylake nodes of TACC's Stampede2 machine. The Jacobi solver is essentially a stencil computation solving the Poisson equation using halo exchange with neighbors across distributed memory. The plot shows four timesteps of execution, with dynamic load balancing happening between the second and third iterations. AMPI-new uses the zero copy Direct API for communication and the pup_buffer API for migration, meaning all communication in this run happens in-place and ranks are migrated in-place as well. Here one rank migrates off of PE 0 at the same time as another rank migrates on to PE 0. The memory optimizations, taken together, allow users to run applications with larger memory sizes without running out of memory during rebalancing. The overall result is a 7% faster run time of Jacobi-3D with 34% lower peak memory usage.

## 3.3 PER-RANK MEMORY FOOTPRINT

Another source of memory overhead in MPI libraries is that used for storage of the opaque objects that users interact with, such as MPI_Groups and MPI_Datatypes. The storage requirements for these objects can range from a few bytes to megabytes in size. For instance, storage of an MPI_Op is typically small (16 bytes in AMPI), while MPI_Groups can vary in size depending on their size and complexity. For small groups, the constituent ranks can be stored as integers in an array. For large groups, this is impractical in terms of memory usage, so implementations typically try to optimize their storage by storing metadata describing the pattern of ranks if possible, or by storing their differences from a parent group using a bitmap. Groups are implicitly stored for all communicators, but users are also free to create groups as first-class objects, from which they can create a communicator. Groups are thus stored on a per-rank basis inside MPI libraries, as are other opaque MPI structures such as MPI_Datatypes, MPI_Ops, etc. Per-rank storage of these objects is typically not overly cumbersome in traditional MPI libraries on most modern HPC systems, which are composed of thousands of nodes with tens of CPU cores each, but with AMPI the per-rank memory overhead of storing these objects can be more acutely felt by end users wanting to run with virtualization and dynamic load balancing since an overdecomposition factor of around 10x is typically used for effective rebalancing. Load balancing can more finely redistribute work when virtualization is done to a higher degree, since each rank on average will represent a smaller fraction of the overall load. This makes it desirable to limit the per-rank memory footprint. We expect that future large scale systems might also push traditional MPI implementations to their memory limits as well.

At the same time, AMPI's shared address space execution model provides a potential

solution to storage of these objects: we can hoist the storage to the PE or node-level and share objects across ranks. We observe that some of these opaque objects are read-only for the duration of the program and they are the same across all ranks. Other opaque objects are dynamically created but are immutable after creation and can have the same value across ranks although each rank is free to define their own and will have unique handles to them. Still others are mutable. We distinguish between these cases and store them differently in order to maximize sharing of resources across the address space while not harming performance significantly.

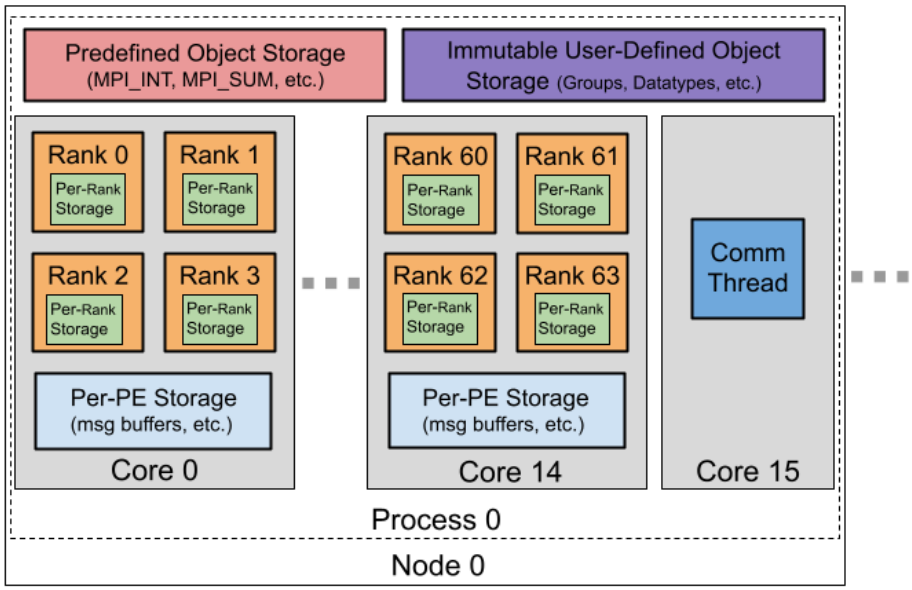



Figure 3.16: AMPI's internal storage of opaque objects differs depending on mutability, expected size, and whether we expect the values to be the same across ranks or not.

Figure 3.15 shows the the memory usage over time on PE 0 of a three dimensional Jacobi solver run on AMPI with 8x overdecomposition on 8 Skylake nodes of TACC's Stampede2 machine. The plot shows four timesteps of execution, with dynamic load balancing happening between the second and third iterations. AMPI-new uses the zero copy Direct API for communication and the pup_buffer API for migration, meaning all communication in this run happens in-place and ranks are migrated in-place as well. Examples of predefined, immutable MPI objects are those representing MPI constants or predefined objects, such as MPI_INT, MPI_DOUBLE, MPI_MAX, MPI_SUM, and the groups for MPI_COMM_WORLD and MPI_COMM_SELF. These built-in objects are the same across all ranks, and users are not responsible for allocating or freeing them. These semantics allow us to hoist their storage to the node-level, rather than at the rank-level. Since they are read-only and accessed within the shared address space, we do not need to lock around access to

them, so efficient access is preserved. In AMPI, we share a single definition of all predefined MPI_Datatypes, MPI_Ops, and MPI_Groups per node. For predefined types, the handles to the opaque objects are also the same on every rank, so there is no need to maintain separate handles for each rank as well. AMPI's internal representation of predefined datatype objects currently consumes around 256 bytes per type, and there are 56 predefined types. If there are, for example, 4096 ranks per node (i.e. 64 ranks per PE, 64 PEs per node), then we save over 58 MB in storage just on predefined datatypes alone. If we scaled that run out to 4096 nodes we would have 16 million virtual ranks in total, and an MPI_Group object could consume up to 64 MB of memory. In practice we do not persistently store the full array of integer rank values unless there is no discernible pattern to the ranks that compose the group. This hoisting of predefined objects to the node-level incurs no runtime overheads since there is no creation/deletion of them, no modifying their values, and the values are by definition the same across all ranks.

The next class of opaque objects we optimize for are those that are user-created, immutable, have arbitrary size, and an expectation that different ranks are likely to independently create objects with the same values, and whose access time is more important to performance than creation or deletion times. Examples include custom MPI_Datatypes and custom MPI_Groups. These objects may or may not be the same across ranks, but we expect that for many applications different ranks on the same node will create objects with the same values. Exceptions would be custom datatypes with absolute addressing rather than relative addressing, or groups that represent very sparse or small communicators in an otherwise large run. MPI_Datatypes are not completely immutable, since users can add a string name or arbitrary attributes to datatypes, so we split storage of user-defined datatypes into the immutable portion of state, which is cached at the node-level, and the mutable portion which is stored on a per-rank basis. For MPI_Groups, which are immutable, we cache their storage on a per-node basis, rather than per-rank, in order to minimize storage of redundant objects that are the same across ranks in the same address space. The main complications arising from caching these objects come from dynamic creation/deletion, the need for private per-rank handles to the objects, and migration of ranks. We maintain per-rank handle management since these objects can be created independently on each rank, and when creating an object we first create the internal object then check if that object is equal to any others in the node cache. If so, we increment the cached object's reference counter, which is atomic, and we free the object we just created, replacing its pointer with the cached one in our per-rank table that translates from MPI opaque handles to the internal object. We store these objects using linked lists with a mutex lock per list. We store user-defined datatypes separate from user-defined groups, and we store different kinds of datatypes separately as

well (contiguous, vector, indexed, indexed_block, struct, etc.) in order to limit the number of comparisons needed when inserting an object into the node-level cache. When migrating a rank, AMPI must serialize all of its internal state per-rank in order to ensure correctness, so we serialize a copy of the cached object and decrement the reference counter when done packing state on the sender side, and add that copy to the new cache on the destination side when unpacking. We only use the lock when creating or freeing an object, not when accessing it. Consequently, our design ensures that access to these objects can still be done directly through the pointer since we ensure reference stability, and that only creation and deletion incur the overheads of locking. We expect that MPI Group and Datatype creation are not performance critical operations and that they are typically created at startup and used repeatedly over many iterations, amortizing the cost of creation over many accesses, though this is in no way guaranteed.

For other objects we maintain per-rank storage as the default. This includes objects that are particular to a certain rank, such as custom MPI_Ops (which store a function pointer) or virtual topologies (where we store a list of neighbors, typically different for each rank), or objects that are mutable, such as MPI_Infos. MPI_Request objects are another example of mutable objects particular to a certain rank. Mutability matters because two ranks could, for example, create the same info and if we were caching them internally as the same object and one rank modified the object we would need to create a new object while maintaining the same opaque handle. We use per-rank storage for objects (such as MPI_Infos, MPI_Ops) that we expect to be small in size typically as well. Figure 3.16 illustrates the different kinds of storage in AMPI's runtime for different types of objects.

Figure 3.17 shows all three improvements to memory usage: reduction in the steady state or floor memory footprint, reduction in the peak of memory usage by migrating ranks in-place, and avoidance of dynamic memory allocation for communication enabled by our new communication interfaces and runtime support. Note that this plot shows the memory usage per node so as to capture the total memory savings across all virtual ranks in the same shared address space. In all, peak memory usage is reduced by nearly 250 MB and the average memory usage is reduced and steadier.

## 3.4 APPLICATION RESULTS

These memory overhead improvements have enabled us to run AMPI applications at unprecedented scale and to run new applications which previously were infeasible. PlasCom2, a plasma-coupled combustion simulation code developed by the Center for Exascale Simulation of Plasma-Coupled Combustion at the University of Illinois at Urbana-Champaign,

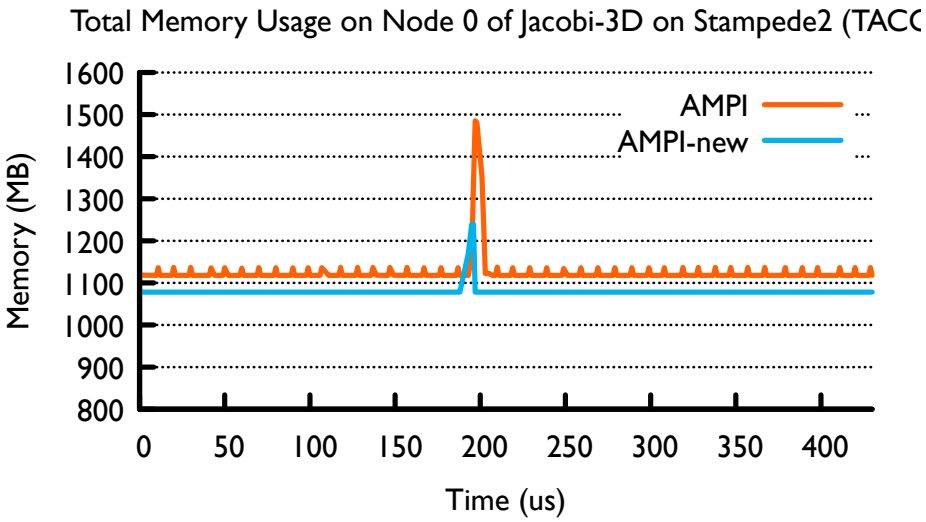



Figure 3.17: Memory usage on Node 0 of a three dimensional stencil benchmark on the Skylake partition of Stampede2 (TACC) with 8x rank virtualization.

has been able to scale across both partitions (Intel Xeon Phi and Xeon) of Stampede2, with AMPI providing load balancing between the two different node architectures. For this, we used a processor-speed aware Proportional Mapper to distribute virtual ranks at startup based on the estimated processor speeds. Then we used a refinement load balancing strategy that takes into account current processor placement. Combined, PropMap and RefineLB provided a 20% speedup with no application changes necessary (other than periodically calling AMPI_Migrate) when running over 2048 nodes with more than 1.3 million virtual ranks (12x virtualization). Trying to run this simulation without support for in-place migration resulted in out-of-memory failures during load rebalancing. Our per-rank memory footprint reductions also saved 42.2 MB per node in static memory usage per process.

## 3.5 CONCLUSION

We have shown that care attention to communication semantics are critical to performance, and that the shared address space between the many ranks in a process can be taken advantage of for memory savings in multiple different ways. We also demonstrated why and how memory usage can be a limiting factor in application runs on top of a virtualized runtime system such as AMPI. The memory overhead of each virtualized entity is particularly important for a runtime supporting overdecomposition, since the degree of overdecomposition tends to have several effects on performance: with higher degrees of overdecomposition,

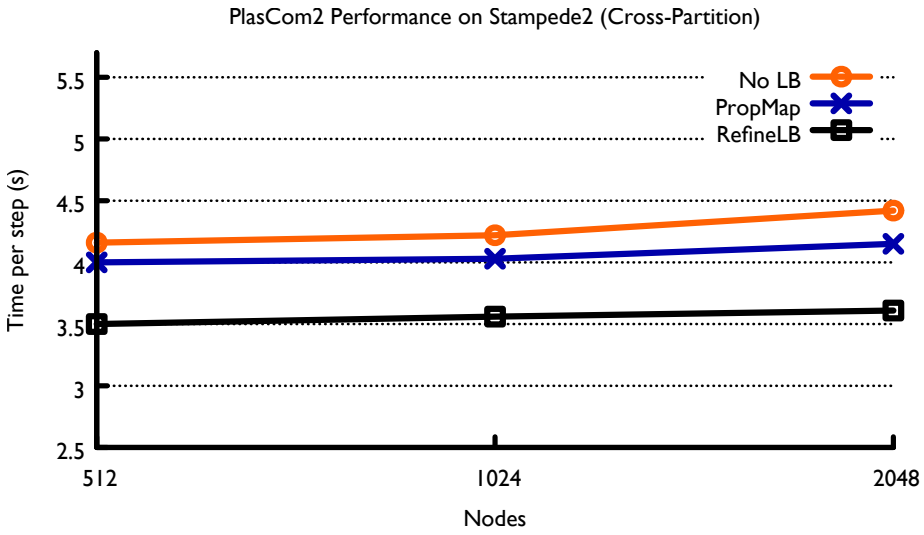



Figure 3.18: Weak scaling of PlasCom2 across partitions of the TACC Stampede2 supercomputer, shown with PropMap alone and PropMap with RefineLB for dynamic load balancing. Lower is better.

each overdecomposed entity represents a smaller fraction of the overall load, which allows finer grain load rebalancing. Our work also decreased the memory overhead of virtualization so that applications can effectively make use of more of the total memory per node by running with larger problem sizes. That work required a reimagining of AMPI's underlying task-based programming model with respect to its communication semantics. Instead of system-owned message objects, Charm++ now also supports communication of user-owned buffers. We added multiple communication APIs for high and low levels of control over the memory registration, and support in-place collectives and migration of chares and virtual AMPI ranks. In particular, migration memory overhead was identified as a limiting factor for applications, and we bound the memory usage by migrating ranks in-place across processes. Migrations within the shared address space now also can avoid copies entirely, encouraging the use of hierarchical load balancing strategies that minimize internode migrations. Altogether, this work enables running new applications on AMPI which are more demanding in terms of memory usage while still being able to perform dynamic load balancing and other runtime optimizations.

For future work, we would like to pursue memory-aware load balancing strategies and to explore use of MPI shared memory windows to minimize the per-rank memory usage at the application level. Shared memory windows require changes to the application and were new to the MPI-3.0 standard, so they are not yet commonly used in MPI libraries and applications. In AMPI, we can support efficient shared memory window access across

a node using the shared address space between ranks. However, rank migrations alter the locality of ranks, so the application must be informed of any migrations. We have added support for querying if a communicator created via MPI_Comm_split_type is valid for this purpose, but more work is needed on applications. For load balancing, currently we can restrict rebalancing to only happen within a node but internode balancing is sometimes necessary to achieve better overall balance. We could also stage multiple rank migrations over time in order to minimize peak memory usage spikes during rebalancing. Currently, there is no such control and many ranks can be migrated concurrently. A consequence of this is that greedy load balancing strategies are avoided when running near the memory capacity. Greedy algorithms can achieve very high quality of load distribution, and so are often desirable.

# CHAPTER 4: POINT-TO-POINT COMMUNICATION OPTIMIZATIONS

## 4.1 OVERVIEW

In comparison to MPI-everywhere and MPI+X (where X is a shared memory programming model), our runtime's virtualized execution model affords unique opportunities for communication optimizations. In this chapter, we consider AMPI's communication performance as a multithreaded MPI or thread-based MPI implementation, where ranks are virtualized and associated with threads rather than processes. As a consequence, each thread has its own private MPI endpoint for communication. MPC [46] and AMPI [14] are two implementations of this model. The key difference from MPI+X is that since each thread has its own rank or endpoint, no serialization around communication is needed (as in *MPI_THREAD_FUNNELED*) and contention around shared resources inside the communication runtime can be avoided (as in *MPI_THREAD_MULTIPLE*). If process virtualization can be automated, as in PIEglobals, no application refactoring to a shared memory programming model is needed, making this model attractive to legacy MPI codes.

In this chapter, we first optimize intranode transfers taking advantage of shared address space. Next, we examine AMPI and its implementation of point-to-point communication with an eye toward asynchrony and concurrency within the runtime, optimizing its performance for the full MPI messaging semantics as well as relaxed semantics. We identify limitations in AMPI's current implementation of point-to-point messaging which limit its asynchrony and concurrency. We propose and implement solutions to better schedule work within AMPI and to improve the degree of concurrency within it. We identify semantic inefficiencies in AMPI's endpoint model and build on top of previous research to exploit relaxed communication semantics in AMPI, with novel optimizations for communication locality. We show results for a range of benchmarks, mini-apps, and applications with different communication patterns to understand the performance implications of our optimizations.

## 4.2 SHARED ADDRESS SPACE COMMUNICATION

### 4.2.1 Background

With modern systems trending toward wider and wider shared memory nodes, optimizing for communication within a node has become more and more important to overall application performance. Many important HPC applications exhibit persistence in their communication

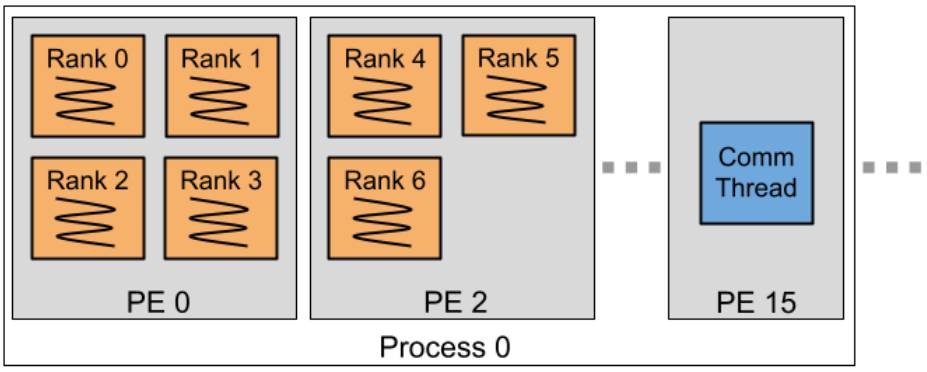



Figure 4.1: AMPI applications typically run with more virtual ranks than cores or PEs and one OS process per socket or node. Ranks on the same PE are co-scheduled in a message-driven manner.

pattern, wherein each rank communicates with the same fixed set of other ranks timestep after timestep. This exposes an opportunity to map those sets of ranks that talk to each other most often close together, either in the same shared memory node or on nodes nearby in the network topology. AMPI's support for user-space shared memory between ranks on the same node, as show in Figure 4.1, exposes a further opportunity for us to optimize for communication locality.

Existing MPI implementations use different methods 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 copy-in/copy-out method is used because registering shared memory pages across processes can be expensive compared to the cost of memory copy operations. 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. These mechanisms allow copying directly from the user's send buffer to the user's receive buffer with any intermediate copy. They have been used for optimization of both point-to-point and collective communication routines. Examples of kernel-assisted interprocess copy mechanisms include KNEM, CMA, LiMIC2, and XPMEM [9, 10, 11, 12]. 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, without the need for special operating system support, calls into the kernel, or the need for registration of shared memory pages.

Other MPI implementations have also included support for direct communication in shared memory. 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 [47]. MPC-

MPI is, similar to AMPI, a threads-based MPI implementation, and includes support for multiple ranks inhabiting the same address space [15, 29]. The PIP library has recently been incorporated with MPI to provide user-level shared memory between ranks on the same node [37]. AMPI differs from these efforts mainly by its high-level features (overdecomposition, load balancing, fault tolerance, etc). Our work is also relevant to any implementation of the MPI endpoints proposal.

Even though AMPI already built and ran on Charm++'s SMP mode, it failed to take advantage of the shared-memory semantics it provides. This is due to conflicting message buffer ownership semantics in MPI and Charm++, as discussed in Chapter 3. In MPI, message buffers are owned by the application. In Charm++, 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 Chapter 3, we circumvented the extra copies needed in AMPI for communication. Here, we show optimizations of that API for messages within a shared address space. This includes communication using derived datatypes, and our method is completely portable without need for kernel-assisted interprocess copy mechanisms.

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, and because we want to exploit communication locality to its fullest.

Figure 4.2 shows the small message latency on a single 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 shared address space between ranks that it could exploit, AMPI did not perform well compared to the other MPI implementations.

For small messages, AMPI P2 consistently has the highest latency, with the exception

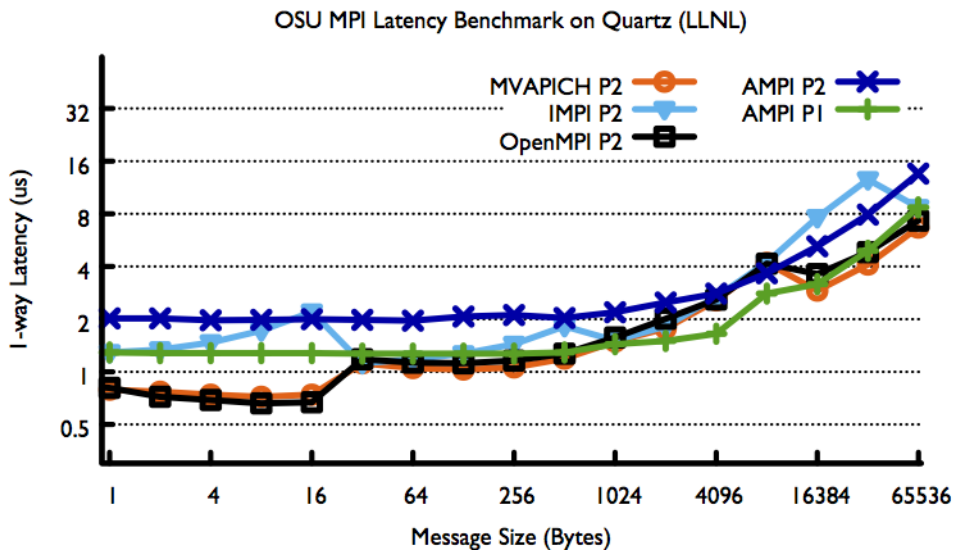



Figure 4.2: The existing AMPI implementation performed poorly in shared memory. MVAPICH2/2.2 and OpenMPI 2.0 perform the best.

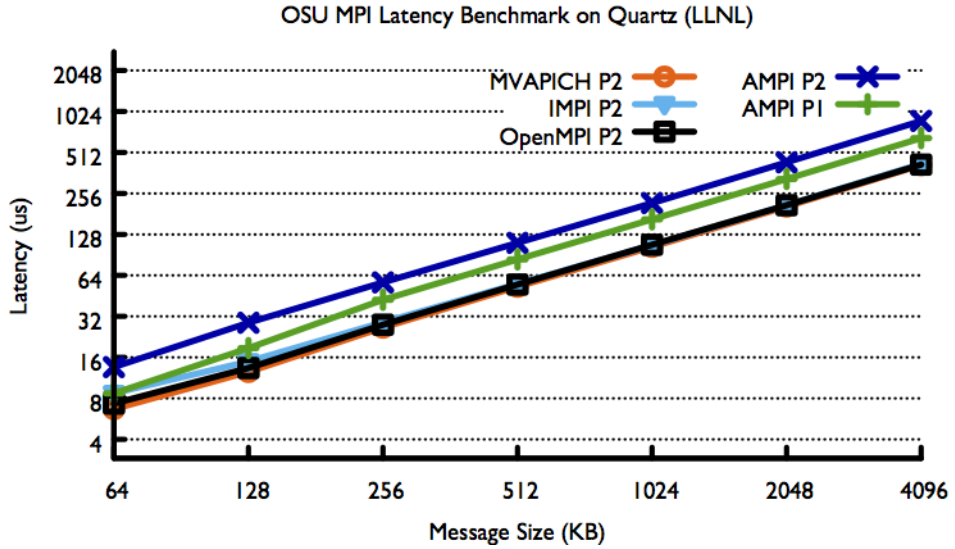



Figure 4.3: For messages sized 64KB to 4MB, all three process-based MPI implementation attain similar performance while AMPI P2 is consistently around 2x slower.

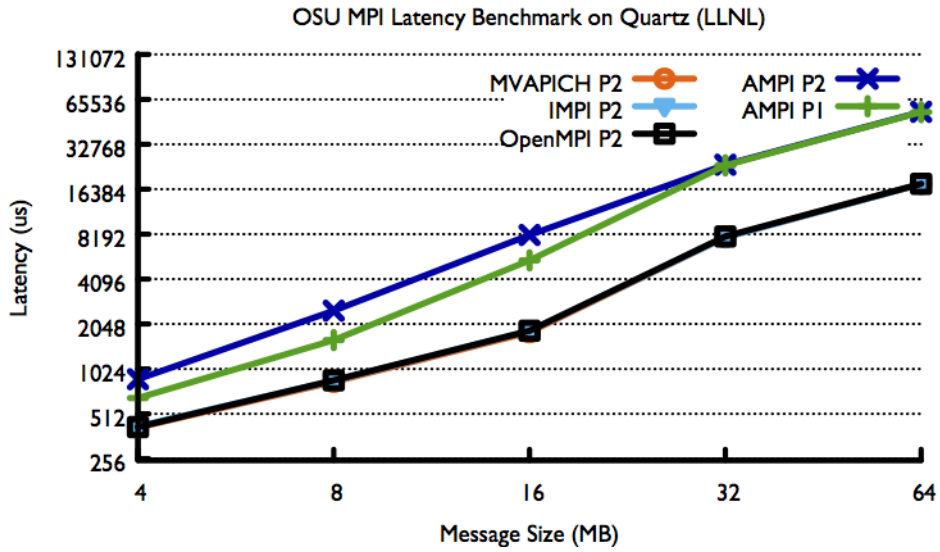



Figure 4.4: Large message latency suffers from an intermediate copy in AMPI P1 and AMPI P2.

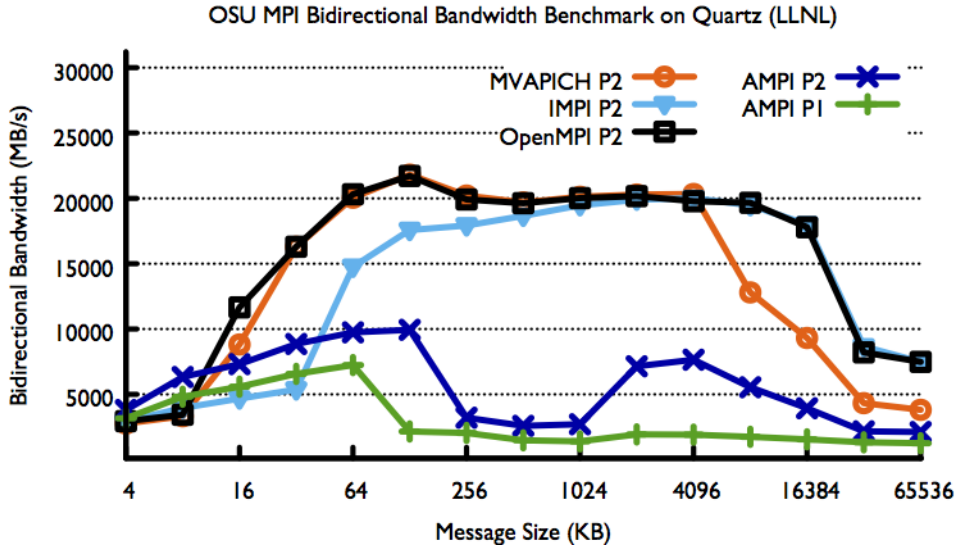



Figure 4.5: Bidirectional bandwidth results for messages sized 4KB to 64MB.

Table 4.1: Overhead per message in microseconds ($\mu$s). 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.

| **Overhead per message ($\mu$s)** | **0-B message** | **1-MB message** |
|---|---|---|
| **Scheduling** | 1.02 | 1.04 |
| **Memory copy** | 0.00 | 162.86 |
| **Other** | 0.25 | 1.31 |

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 4.3 and 4.4, 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 in Figure 4.5, AMPI fares no better, doing 2x worse at its peak bidirectional bandwidth usage than MVAPICH2/2.2. We expect that AMPI should be able to match the peak main memory bandwidth that is available to two cores on a node in the case where ranks are on separate PEs (P2). Using the STREAM benchmark, we measured memory bandwidth at 25,926 MB/s.

### 4.2.2 Performance Optimizations

In order to understand why AMPI is performing so poorly, we profile its performance on the messaging latency benchmark. Table 4.2.2 breaks down 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 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.

**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, QuickThreads, the (deprecated) POSIX ucontext interfaces, and a jump-buffer based implementation. By default Charm++ uses the ucontext ULT implementation on most Linux-based platforms. We added support for Boost context threads, which are the fastest on Quartz and all systems we have tested. Boost uses assembly instructions to save and restore only the registers that are needed. This brings the scheduling time down from 1.02us to 350ns. 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 decreased Charm++'s scheduler overhead by optimizing it for common cases and for AMPI's needs. Charm++'s low-level runtime system Converse supports conditional and periodic callback objects alike. Conditional callbacks are triggered by special events in the system, such as the beginning or end of idle time on that PE. Periodic callbacks are triggered roughly every so often in some unit of time, such as every 10 us or 100 ms. Periodic callbacks require the runtime to call a timer and check if an interval has passed during which a user callback should have been triggered. Previously Converse would call a timer every time through the scheduler loop regardless of if the user had registered any periodic callbacks. We optimized this away for the common case of there being no active periodic callbacks. We also disabled features in Charm++ that AMPI does not use, such as non-FIFO message queuing strategies. Together these changes saved another 80ns, bringing the scheduling overhead down to 270ns.

Third, we reduce the number of context switches needed when executing MPI_Waitall, which is used in the bidirectional bandwidth benchmark. Multiple completion routines such as Waitall allow the runtime to overlap completion of multiple messages, rather than the programmer scheduling individual calls to Wait on specific requests. 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. This means a thread that issues $m$ non-blocking operations and then blocks on $n$ specific requests using waitall could potentially be awoken $m$ times. Semantically, MPI_Waitall says that we only need to unblock the thread once.

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 message'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 over 2x, from 1.29 $\mu$s to 0.44 $\mu$s on one execution unit of Quartz.

### Memory Pooling

A significant portion of the cost for small messages 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 0.56 $\mu$s to 0.44 $\mu$s for messages sent and received on the same execution unit, and from 1.14 $\mu$s to 0.81 $\mu$s for messages between execution units in the same address space.

**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. In chapter 3, we discussed our development of new in-place communication interfaces in Charm++. Building on top of the direct API, we were able to implement a rendezvous protocol that makes use of user-space *memcpy* for transfers within the shared address space. There is no memory registration or pinning required, and AMPI can automatically take advantage of high bandwidth memory if the user buffers are already allocated there since there is no intermediate copy.

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. Avoid-

ing 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.

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 directly. In order to synchronize concurrent accesses to the messaging data structures, we rely on Charm++'s message-driven scheduling. In the next section, we pursue concurrent message matching for relaxed communication semantics. Here, we rely on Charm++ message scheduling queues 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

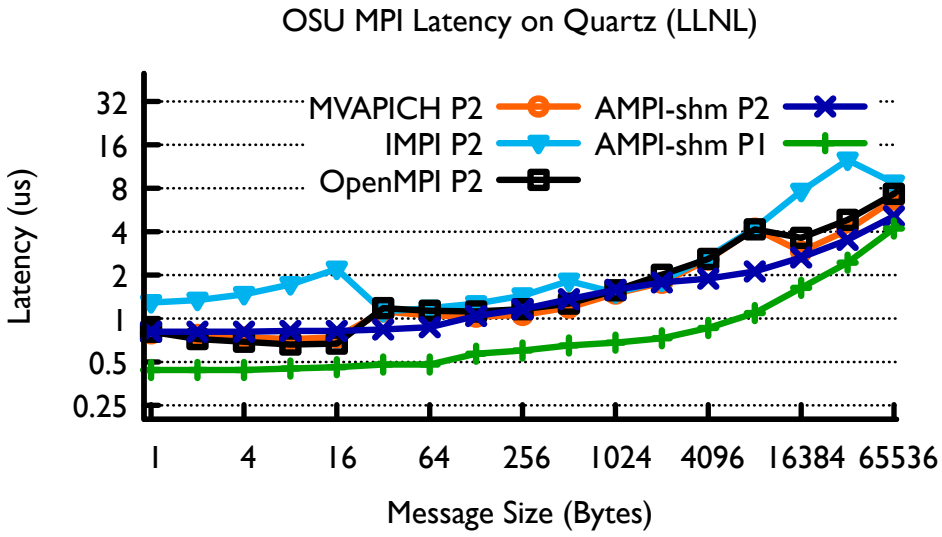



Figure 4.6: For small messages, our design brings AMPI within 5% or better than the best process-based MPI implementation for all message sizes.

find that on Quartz, the cross-over point between the eager and rendezvous protocols is 4 KB in terms of latency within the shared address space.

### 4.2.3 Results

We call our shared memory-aware implementation 'AMPI-shm'. Figure 4.6 shows that on Quartz AMPI-shm has 2x lower latency than the previous AMPI implementation, and is now faster than all process-based MPI libraries on the system for nearly all message sizes. 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. For small messages between endpoints co-located on the same execution unit, AMPI-shm has 58% lower latency than AMPI.

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 4.9). 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 [48], 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.

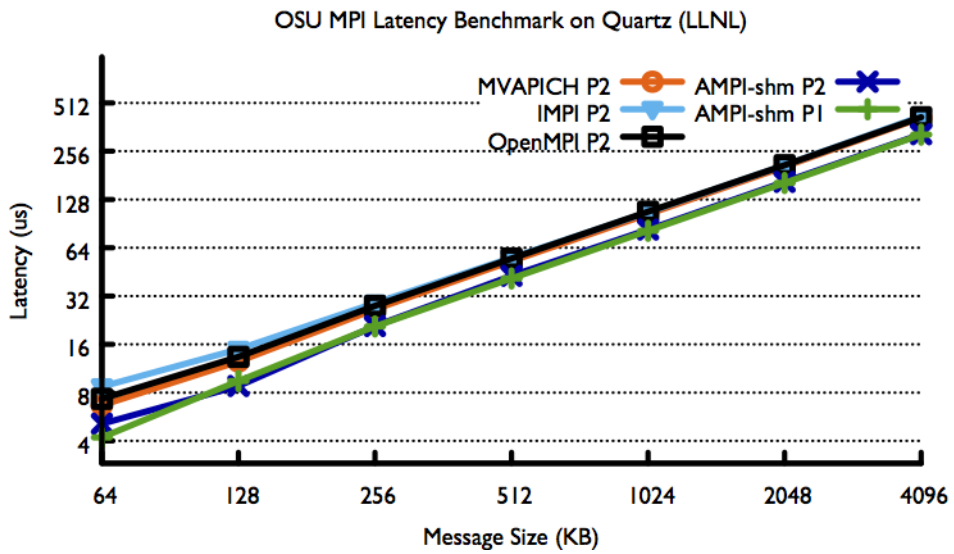



Figure 4.7: For large messages, the user-space single copy method over shared address space achieves lowest latency.

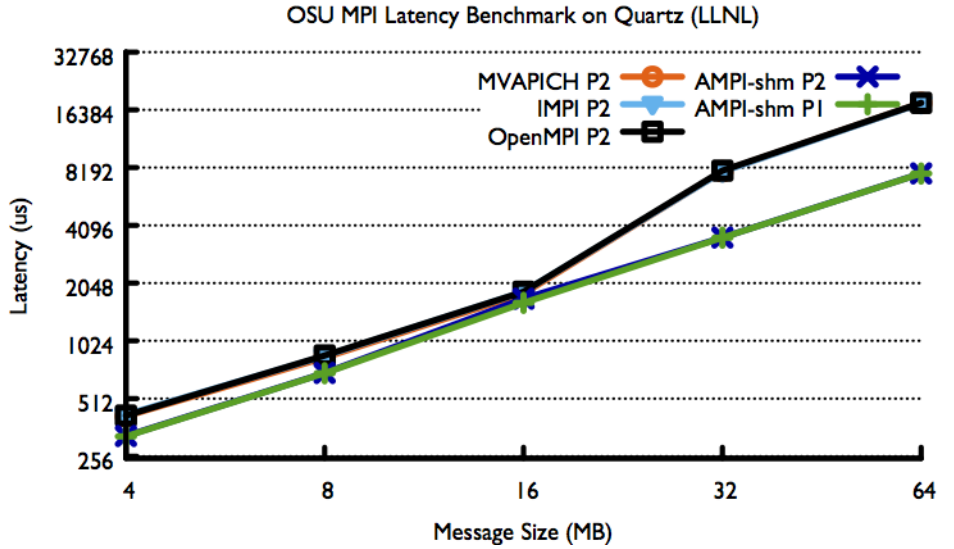



Figure 4.8: At the largest message sizes of 32 and 64 MB, AMPI-shm P1 and P2 have 2.33x better latency than the process-based MPI implementations.

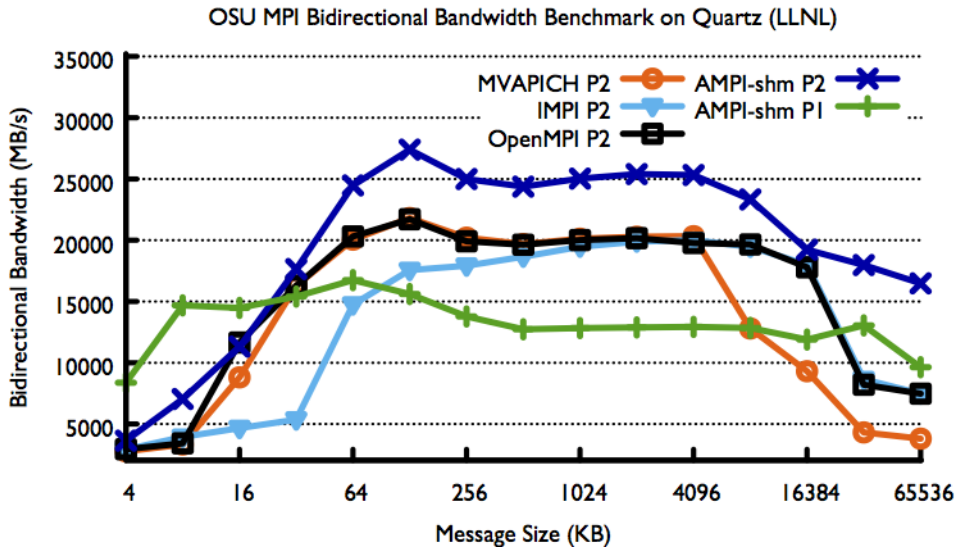



Figure 4.9: The shared memory-aware implementation of AMPI outperforms process-based MPI implementations in terms of bidirectional bandwidth. STREAM copy achieves a bandwidth of 25,926 MB/s on two cores of Quartz.

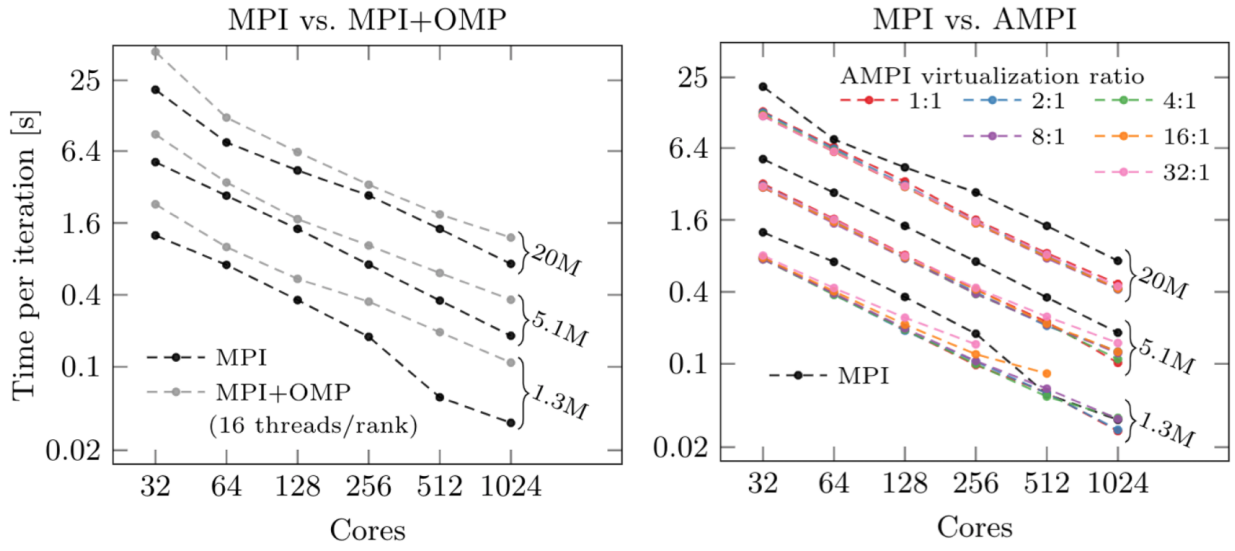



Figure 4.10: Performance of PlasCom2 on Quartz at LLNL, comparing AMPI (with and without overdecomposition) to MVAPICH2/2.2 (with and without OpenMP). Lower is better.

We also looked at the performance of a full-scale application: PlasCom2, a plasma-coupled combustion simulation code developed by the PSAAPII Center for Exascale Simulation of Plasma-Coupled Combustion at the University of Illinois at Urbana-Champaign. PlasCom2 uses MPI for distributed memory parallelism using spatial decomposition, with optional

OpenMP support for shared-memory parallelism. It is a multiphysics code that works on overset meshes with optional support for chemistry, uncertainty quantification, and more. The problem case being run here uses 3 overset meshes to simulate laser-induced breakdown of a high-temperature plasma under turbulent condition, as in a jet engine.

Figure 4.10 shows scaling performance on the Quartz cluster at LLNL for three different problem sizes and for various configurations of MPI, MPI+OpenMP, and AMPI with varying degrees of virtualization. The MPI implementation is MVAPICH2/2.2. The results show AMPI providing up to 2x speedup over the MPI-only runs with MVAPICH2/2.2 and up to 4x speedup over the MPI+OpenMP hybrid version, even without using overdecomposition or load balancing. With enough work per rank, virtualization provides additional speedup, whereas at the strong scaling limits it results in overhead.

## 4.3 ASYNCHRONOUS MESSAGING OPTIMIZATIONS

While network hardware and software have evolved to support low latency and high bandwidth communication, hiding latency effectively remains both important for performance and difficult to attain for many applications. Nonblocking communication routines separate the initiation and completion of message transfers semantically. This allows the MPI implementation to potentially overlap the underlying communication with the application's work. Simply using non-blocking point-to-point communication routines can enable enough overlap for some codes, but for others latency is more difficult to hide. This depends on the communication pattern, message size, the MPI library's messaging protocols, and the amount of work the application has to overlap its communication with. Developers are often faced with trade-offs in the maintainability of their code and implementing more complicated messaging patterns such as double-buffering which can expose more asynchrony. MPI libraries are allowed by the standard to only make progress on outstanding communication events only when they are called into, if they so choose. All of these factors combine to make attaining latency tolerance in MPI applications often difficult.

The MPI Forum has subtly acknowledged the importance of progress to end users by evolving the standard's explanation of it over time. The standard still guarantees only weak progress of communication events, meaning that the MPI library is free to only make progress on outstanding messages when the application calls into it. But it has evolved from saying, in MPI-2.2, "Different implementations reflect these different interpretations. While this ambiguity is unfortunate, it does not seem to affect many real codes. The MPI Forum decided not to decide which interpretation of the standard is the correct one, since the issue is contentious, and a decision would have much impact on implementers but less

impact on users." to, in MPI-3.1, only "Different implementations reflect these different interpretations. While this ambiguity is unfortunate, the MPI Forum decided not to define which interpretation of the standard is the correct one, since the issue is contentious." This stance allows MPI implementations to be single threaded internally, simplifying the runtime. But the shift in MPI-3.1's language serves as an acknowledgement that having weak or strong progress does have a real affect on users. Application developers may and sometimes do already insert extraneous calls to MPI routines (i.e. *MPI_Test*) into their programs in order to prod the library's progress engine along.

AMPI primarily provides latency tolerance through overdecomposition and message-driven scheduling. That is, if there are multiple ranks on the same PE, then one rank's blocking communication can be overlapped with the computation of another rank on the same PE. This kind of overlap is achieved by message-driven scheduling, without requiring the application developer to expose all of the asynchrony. But, as we will show, it is limited with respect to asynchrony because of both its reliance on message-driven scheduling for synchronization, and by the MPI point-to-point communication semantics.

### 4.3.1 Background

First, we explain the limitations of AMPI's existing point-to-point communication implementation with respect to asynchrony before exploring related work in the MPI community that helps motivate our own novel design.

#### Point-to-Point Communication in AMPI

AMPI implements point-to-point communication using Charm++'s support for communication between persistent, globally addressable C++ objects called *chares*. AMPI allocates a one dimensional array of chares for each communicator the user creates. Chares are by default anchored to a particular PE, though they can migrate across PEs and nodes of the system at runtime. All communication between chares is non-blocking, one-sided, and unordered.

In MPI, point-to-point messages may be blocking or non-blocking, are two-sided operations with both a send and a receive, and all messages between a pair of ranks are guaranteed to match in the order in which they are sent. In order to implement MPI's messaging semantics on top of Charm++, AMPI must maintain extra state to buffer out-of-order messages and match messages to requests. To do so, AMPI maintains sequence numbers between each pair of communicating ranks and stamps messages with these sequence numbers. As is typical

in MPI implementations, AMPI also manages two queues for each rank: one for unexpected messages, and one for posted requests. On the receive-side, when a message arrives to a chare (which is associated with a particular rank and communicator), we first check if it is in-order. If so, we check if a receive request has been preposted and match it if so. If not, we stash the message in the unexpected message queue to be matched later when a corresponding receive or probe call is performed by the receiver. Each AMPI rank also has associated with it a count of the number of requests it is currently blocked on, and each request is marked as being currently blocked on or not, so that as messages are matched we decrement the counter until it reaches 0, and only then do we awaken the corresponding user-level thread for that rank. This means that for multiple request completion routines such as *MPI_Waitall*, which might block on the receipt of $n$ requests, we only context switch one time, rather than $n$ times.

To make this more concrete, we can consider an example. Figure 4.11 shows a Gantt chart for the execution over time of 3 PEs in a single node. This node is connected to a network with RDMA capabilities. For simplicity, we do not show remote nodes. On these 3 PEs, AMPI can launch 1 process with 3 kernel threads, one dedicated to handling off-node communication and the other two worker threads on which virtual ranks can be scheduled. Here we have ranks 0 and 1 on PE 0 and ranks 2 and 3 on PE 1. We show the execution of these ranks split into user code execution, idle time, and AMPI message handling. We distinguish between eager and rendezvous messages to illustrate their differences. Each rank here executes some user code and then calls *MPI_Recv* with the source being some rank on another node.

The key to understanding this Gantt chart and the performance issue it illustrates is understanding the different queues inside AMPI and Charm++. These are illustrated in Figure 4.12. First, each AMPI rank has associated with it two message matching queues used to ensure MPI messaging semantics, as well as additional data structures for message ordering and tracking how many requests are currently blocked on. Second, each worker thread or PE has a First-In, First-Out (FIFO) scheduling queue of remote method invocations on its local chares. Third, each PE has a multiple-producer, single-consumer (MPSC) queue that it regularly checks for messages enqueued by the communication thread or other worker threads in the same process. When it finds a message there, it dequeues it from that queue and enqueues it into its local scheduler queue. The communication thread polls the network for messages, checks which PE they are meant for, and simply forwards those messages via the MPSC queue to the recipient PE. The communication thread also issues RDMA read and write calls and forwards the notification of completion of those calls as callbacks. Note that on network layers that support it, worker threads can issue RDMA read/write commands

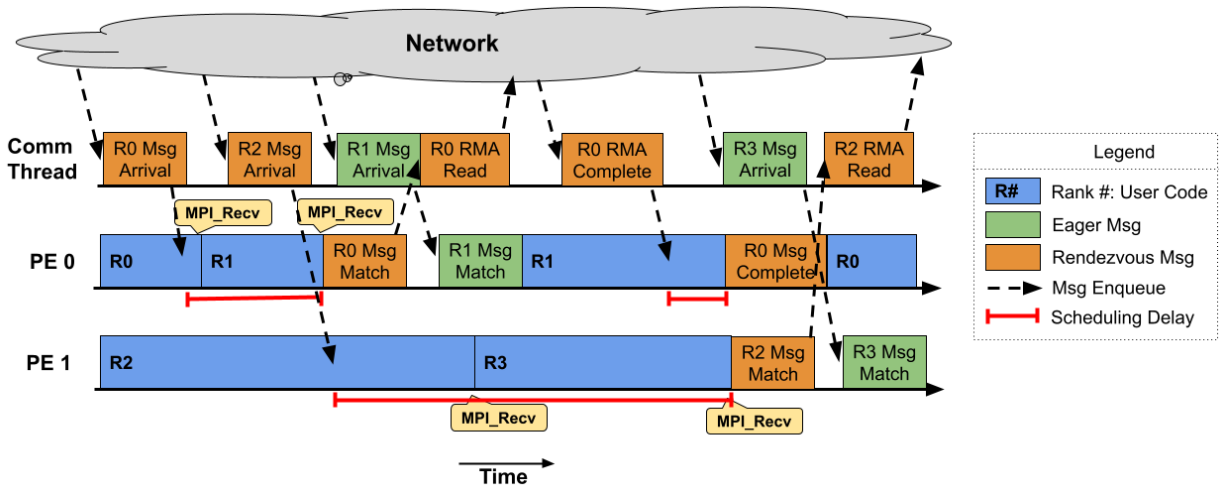



Figure 4.11: A Gantt chart showing execution of 4 ranks on a single node which are each receiving one message from off-node. Note the scheduling delays in particular: these are where AMPI's current design results in delayed handling of communication events on the critical path.

directly.

In the example, note the scheduling delays where a message sits in a queue for some time until it can be executed. These occur where messages are forwarded from the communication thread to the PE of a particular rank. The scheduler must itself run in order to pick messages from the MPSC queue and enqueue them into the local scheduling queue. Some time later, that message will be scheduled and execute on that PE. Those messages represent either eager or rendezvous protocol transfers. For an eager message, the payload is part of the Charm++ message along with metadata about the sender, tag, and communicator. For a rendezvous exchange, the message only contains metadata about the sender's buffer and its rank, tag, and communicator. Upon matching a rendezvous message the receiving chare will then initiate an RDMA read (or remote get) operation. Completion of that RDMA operation is asynchronously triggered through the communication thread via a Charm++ callback. The callback must again travel through the MPSC queue to the PE on which its chare resides, and then be scheduled for execution. The callback method marks the corresponding request object as complete and awakens the user-level thread if this was the last request that the rank was blocked on.

AMPI's current message-driven design means that a message that has arrived to the communication thread or even the PE on which the rank is running is effectively invisible to that rank until that message can actually be scheduled on that PE. This requires that the current rank yield control to the scheduler and that any other work in front of it in the scheduler's

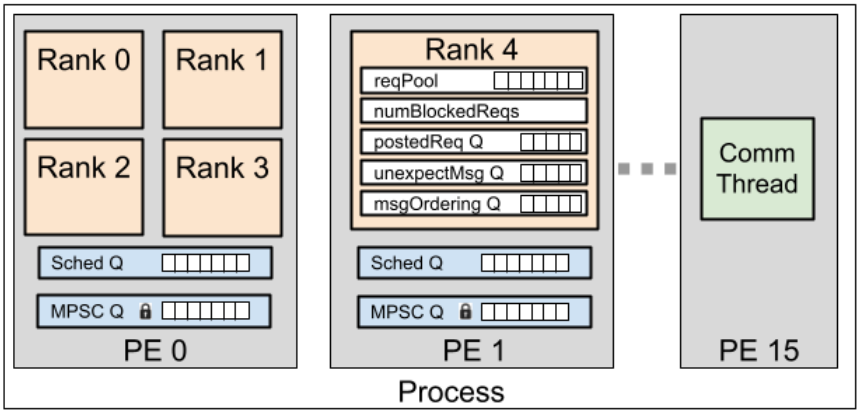



Figure 4.12: AMPI implements message matching and ordering at the endpoints, and relies on Charm++'s support for message-driven scheduling for progress. Only rank 4's internal data structures are shown here for simplicity. Note that PEs have separate local scheduling queues and concurrent MPSC queues for intranode communication. The only concurrent data structure here (signified by the lock icon) is the Multiple Producers Single Consumer queue on each PE.

queue is run first before it is sequenced and possibly matched. This forces unnecessary context switches and scheduling delays in matching messages when a message has arrived to a PE but cannot yet be matched because it is in the MPSC queue or the scheduler queue. These delays can propagate further for rendezvous messages, where matching the message is necessary before issuing the RDMA operation to asynchronously complete the payload transfer.

In the example, the first delay on PE 0 happens because the message can't be matched until its method invocation gets scheduled on PE 0, despite the message having been delivered to the node and PE 0 before the matching *MPI_Recv* call was made. Consequently, the corresponding RDMA operation is delayed. The scheduling delay on PE 1 is similar. The second delay on PE 0 happens when the RDMA read completes and the communication thread enqueues a callback to rank 0 in order to mark the receive request complete and awaken rank 0's thread. The delay here does not have the same kind of knock-on effect that delays in matching rendezvous messages does, since the only thing to possibly do after completing a request is to awaken the thread. This means that this delay can be overlapped with useful work by another rank on the same PE, as happens in this example. However, we note that given more ranks on each core (it is common in AMPI applications to run with 4-32 ranks per core for efficient dynamic load balancing) and AMPI's current design, all of these delays can be lengthened.

There are several important performance consequences of AMPI's current design as it

relates to point-to-point communication. The affinity of ranks and all of their communicator chares to the same PE means that we do not need to lock around access to the internal messaging data structures (note that AMPI does not currently support the MPI thread level *MPI_THREAD_MULTIPLE*). However, it also means that all message matching has to take place on the same PE on which a rank is itself running, since messages to chares arrive through the same scheduler queue that services all other ranks on the same PE. This design has several downsides that cause inefficiencies in terms of context switches (and resultant cache turnover) and synchronization or scheduling overhead for all messages between ranks not on the same PE:

1. A thread must yield to the scheduler whenever it is going to return from a non-blocking request completion routine (i.e. *MPI_Test*{*any,some,all*}) with the flag argument false. This is necessary to enable the incoming message to be scheduled on the same PE. It is even true when the message that will match the request we are testing is already in the scheduler's PE queue but not yet in the communicator-specific message matching queues owned by the chare. This results in extra context switches which in turn mean another rank may be scheduled and bring its own data into the cache while evicting the state of the suspended rank(s).

2. Internode communication must always be forwarded by the dedicated communication thread through MPSC queue and then the scheduler's queue before being matched. This requires synchronization, possible delays waiting in each queue while other ranks run, and possibly extra context switches as messages that have arrived on the node but not yet been matched at the chare are treated as not having arrived yet.

3. Intranode messages must travel through the PE scheduling queues before being matched, despite sender and receiver sharing the same address space. That again means synchronization realized through the MPSC queue and scheduling, which can delay processing.

**Asynchronous Point-to-Point Communication in MPI**

Fundamentally, we would like to make AMPI's implementation more fully asynchronous and to make better use of the shared address space among the many ranks in each node. Prior research has investigated multiple optimization possibilities in the face of similar problems in the context of MPI(+X) and asynchronous progress for MPI libraries.

For MPI+X (where X is a shared memory parallel programming model, such as OpenMP), researchers have explored finer grain locking around the different message matching and ordering data structures inside MPI [49] [50]. The use of different kinds of locks can help

minimize contention and favor work on the critical path or optimize for data locality [51] [52]. Optimization of the internal locking minimizes serialization and contention, and is necessary to implement the full MPI messaging semantics where messages between a given pair of ranks cannot overtake one another and wildcard receives are allowed. The MPI semantics for message ordering and support for wildcard receives make efficient concurrent two-sided communication operations difficult to realize. But other research has shown drastic improvements by relaxing the semantics [5]. Dang et al demonstrated dramatic speedups over existing *MPI_THREAD_MULTIPLE* implementations by dropping support for wildcard receives and message ordering both. The key insight here is that without wildcard receives and message sequencing, receive requests match to a single known sender's message. Taking advantage of that, matching can be done efficiently using a concurrent hash table where message matching times are constant rather than linear in the number of requests, as they generally are with the full semantics and queue-based implementations. The MPI 4.0 standard, recently ratified, includes standard *MPI_Info* strings that applications may set on communicators to tell the runtime about any semantics they do not require and which the runtime may be able to optimize for. If the application can tolerate overtaking messages, they can set "mpi_assert_allow_overtaking". If they will not use wildcards, they can similarly set "mpi_assert_no_any_tag" and "mpi_assert_no_any_source".

Previous work on asynchronous progress has focused on two different models of progression: with dedicated resources for progression and without. In general, the MPI standard does not guarantee so-called "strong" progress of communication. Instead, it is valid for the MPI library to only make progress when the user has called into the library. This means that users sometimes insert periodic calls to MPI_Test into their applications so as to drive the progress engine manually. Dedicating a resource such as a thread to running the progress engine has the obvious downside of consuming one execution stream per process (which is especially costly for MPI-only codes), but it provides good responsiveness to the network [53]. If both the progress thread and the main thread can call into the progress engine, locking is required around the shared data structures, which can hurt message latency. Some MPI implementations already support this model of thread-based asynchronous progress, and the approach has also been demonstrated using separate ghost processes as the offload engines [54] or specialized hardware on the Network Interface Card (NIC) [55]. Without dedicated resources, asynchronous progress has been implemented through the use of interrupts and/or thread preemption [56] [57]. These approaches tend to suffer from context switching overheads or other contention issues while not providing as much network responsiveness as the dedicated approach.

Another approach has been to rewrite applications to exploit the independence of commu-

nication over different communicators, windows, or with different tags from threads in each process [6]. Essentially, if each thread ID in each process uses its own private communicator unique to that thread ID, the MPI runtime can then associate communication resources to the thread rather than sharing across the process. This requires applications to be rewritten, and to use communicators or other logical MPI resources in an unconventional way, but the runtime optimizations are applicable to our work as well.

We explore multiple possible solutions to more intelligently schedule work inside AMPI and to enable concurrent access to the internal MPI data structures in an efficient manner. We pursue communication optimizations for the full messaging semantics as well as more relaxed ones, shedding light on the semantic overheads for a threads-based MPI implementation supporting overdecomposed endpoints such as AMPI.

### 4.3.2 Asynchronous Communication Support

#### Asynchronous Request Completion

First, we notice that completion of rendezvous transfers is inefficient in AMPI's design. Currently, after a rendezvous metadata message is matched at a chare (endpoint), either an RDMA read is initiated for off-node data or a memcpy is performed for intranode transfers. For an RDMA operation, completion of that operation is signalled asynchronously to the caller via a Charm++ callback routine on the chare. That completion is polled for by the communication thread and then forwarded to the chare's PE, travelling through the MPSC queue and the local scheduling queue before running. Once invoked, that callback routine is itself very simple. The callback is invoked with a Charm++ message as its argument which contains the MPI_Request (an integer index) of the corresponding request object to mark as complete. In the routine we lookup the request, mark a boolean "complete" variable as true, and check if the request is currently blocked on (another boolean value in the request structure), and if so we resume or awaken the thread. That thread resumption is currently done inline, so we immediately context switch to running that thread from wherever it last blocked. The completion process is essentially the same on both sender and receiver ranks after the RDMA operation. For intranode transfers, which use user-space memcpy within the shared address space, these completion callback routines are called inline after the memcpy since that happens synchronously on the CPU.

In order to achieve better overlap of rendezvous transfers, we can instead perform the request completion process directly through shared memory from the communication thread for an RDMA transfer or from another PE in the same node for a memcpy transfer. This

requires changes to AMPI's control flow and to its request objects and the count of requests currently blocked on in order to make shared access safe. We also pad the request objects out to be at least the size of a cache line to avoid false sharing. We protect each request object with a mutex lock and make the number of requests a rank is currently blocked on atomic. Then control flow changes so that the Charm++ callback object invoked after RDMA completion happens on the communication thread. We achieve that by creating a Charm++ Node Group (defined as having one instance per process) and making the callback take an additional pointer to the rank at which it is targeted. The callback routine is marked with Charm++'s immediate attribute which enables the runtime to invoke it on any PE of a node, including the communication thread.

### Prioritized Scheduling

Next, we notice that AMPI's current scheduling policy fails to prioritize work in an efficient manner and results in sub-optimal message matching. Charm++ provides support for prioritized scheduling of remote method invocations in addition to different policies such as first-in first out or last-in first-out. The sender (whoever is invoking the method on a chare) sets the priority as either an integer or a bitvector. AMPI had been skipping the prioritized message queue for all its work and relying on the FIFO scheduling policy.

Looking at AMPI's runtime design, we note that when a rendezvous protocol ready-to-send message is matched by a receiving chare, it then issues an RDMA read operation which will complete asynchronously. We would expect that the earlier we can issue these RDMA reads the sooner they will finish and unblock a rank waiting on its completion. Thus, we can switch AMPI to prioritize rendezvous ready-to-send messages above all other Charm++ messages (eager messages and rendezvous completion callbacks). Further, we separate rendezvous ready-to-send messages into off-node and on-node transfers, realizing that on-node rendezvous messages perform the payload transfer using a synchronous memcpy call which could delay processing of other remote RDMA operations in the scheduling queue. This three-level prioritization scheme allows us to prioritize 1) matching of messages that will lead to RDMA operations, above 2) matching messages that will lead to intranode memory copies, above 3) eager messages of all kinds and rendezvous completion callbacks.

### Locality-Aware Thread Resumption

In addition to prioritized scheduling, we notice that thread resumption can benefit from communication protocol or locality awareness. Currently in AMPI all thread resumption

happens inline. This avoids a trip through the scheduler and preserves locality by not allowing another rank to to be scheduled and run between the time the last message being waited on for completion actually finishes and that thread is resumed. For eager messages, the payload is brought into the receiving rank's CPU cache when it is copied from the internal message buffer to the user's receive buffer, so resuming the receiving rank's thread inline can help preserve locality if the receiver reads the receive buffer soon afterwards, which is often the case. On the other hand, rendezvous messages are only brought into cache if they are within-node; otherwise the RDMA read operation does not typically bring the payload into cache on its own. As a consequence, we can perform thread resumption after RDMA transfers through the scheduler instead of inline without adverse cache effects. This then allows the scheduler to run and perform high priority work (message matching, posting of RDMA operations, etc.) now rather than after the resumed rank runs and next calls a blocking MPI routine, which may be a considerable time later.

**Per-Rank Scheduling Queues**

We observe that messages that have arrived on the node and not yet been scheduled are invisible to AMPI's message matching. This includes messages just arriving on the communication thread but not yet forwarded through the MPSC queue to the destination PE and messages sitting in the MPSC queue but not yet moved to the scheduling queue. Ideally, any message already in the node could be sequenced and matched. AMPI's internal queue structure and scheduling prevents that, although relying on them does ensure proper synchronization within shared memory.

Without requiring concurrent access to AMPI's message sequencing and matching queues, we can peek into the local scheduling queue. If we do that directly, however, we need to check multiple queues (for prioritized messages) and each queue can contain messages for any and all ranks on that PE. With many ranks on each PE, we expect the queues can contain many messages during the communication phases of a bulk-synchronous application. Peeking into those queues then could be expensive. Instead, we can split the one scheduling queue per PE into multiple queues, one per rank, which the scheduler manages by forwarding messages to the proper queues and then dequeuing messages from the multiple rank-queues. The scheduler drains each rank's priority queue completely before moving to the next rank's queue in a round-robin fashion, in order to preserve locality between multiple messages for the same rank. This is done using enhancements to Charm++'s support for object-based scheduling queues. We note that this could be done per communicator instead, to avoid peeking at messages meant for other communicators of the same rank, but that could

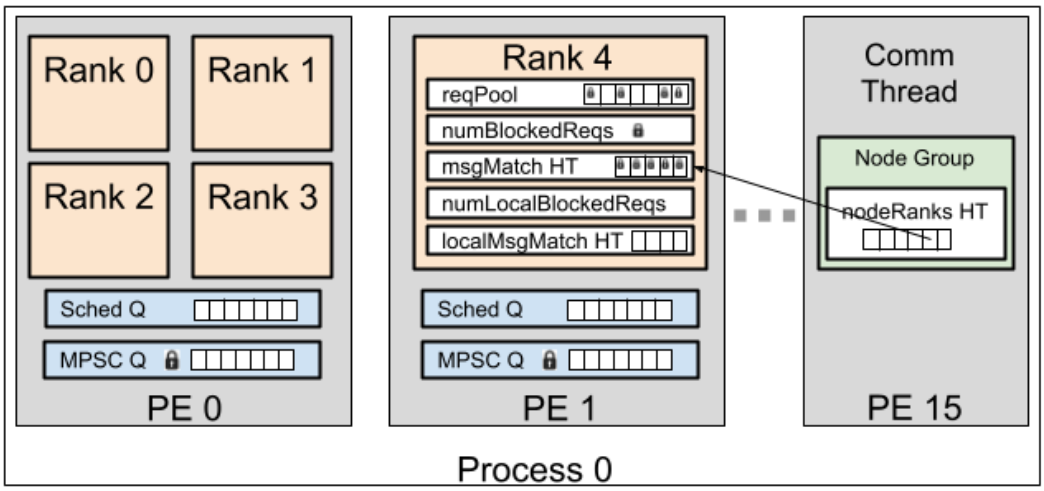



Figure 4.13: AMPI runtime design for taking advantage of relaxed communication semantics, with locality-aware concurrent message matching and asynchronous request completion. Compare this with Figure 2, AMPI's original design.

greatly increase the number of queues for certain applications.

### Concurrent Message Matching

With the prior optimizations for asynchronous request completion and prioritized scheduling in place, the remaining remote method invocations that are tied to a particular PE are related to message matching. As discussed previously, the semantics around MPI message matching make parallelizing the matching process difficult, since messages must be matched in the order they are sent for any given pair of ranks and wildcard receives can match to different messages.

Solutions for concurrent message matching in the context of MPI+X with thread level *MPI_THREAD_MULTIPLE* have ranged from using one global lock around access to all internal MPI data structures to using finer granularity locks around each separate data structure. Different kinds of locks and concurrent data structures have also been explored to optimize locality, avoid contention, and favor work on the critical path. The MPI-4.0 standard's support for partitioned communication and proposed enhancements to it can be seen as attempts to take message matching off the critical path. Other work motivated the inclusion of info hints on communicators for relaxed messaging semantics in MPI-4.0. We implement support in AMPI for this last approach, building on the prior work of Dang et al [5] but applying their concurrent message matching support to AMPI and its execution model with many endpoints in each process. We also differ in our support for multiple completion

routines and collectives. We do not pursue the global mutex or other fine grained locking solutions at this time due to the inefficiency of the global mutex approach and the complexity of finer grain locking.

A key insight in [5] was that if wildcard receives are not used and messages are allowed to overtake one another, each request will match to one message with a known sender. This allows matching messages in constant time using a concurrent hash table. Requests and arriving messages alike are inserted into the hash table, and whichever arrives second (the posting of the request or the incoming message) removes the matching first entry from the hash table. There is no need for separate message sequencing or handling of wildcard receives. In their work, there is one hash table per process, with many lightweight threads able to communicate concurrently with threads in other processes.

In AMPI, with many ranks sharing the same address space, each its own separate communication endpoint, we maintain one hash table per rank. The message matching hash table is accessible by any PE in the node including the communication thread. We also maintain a (non-concurrent) hash table on each PE (including the communication thread) that translates from a rank to its chare object pointer for all ranks in that node. This allows fast direct access to the internal state of all ranks in a node. When a message arrives from off-node, the communication thread uses the destination rank in the message to look up the corresponding concurrent message matching hash table and access it. If a matching receive request was preposted, the communication thread marks the request object complete and checks if that request is currently blocked on, in which case it decrements the atomic counter of blocked on requests for that rank. When the counter hits zero, it awakens that rank's thread by enqueuing the thread resumption in that PE's MPSC queue to then be scheduled. Note that for intranode transfers, ranks can access the message matching hash table of other ranks directly without going through the communication thread. All thread resumption from the communication thread and other PEs in the node is done through the scheduler.

This design achieves concurrent message matching for the relaxed semantics of no wildcards and overtaking messages, but PE-local messages now incur the overheads of shared memory access without the need for it. Ideally, communication that is PE-local would be fast in order to minimize the overhead of virtualization for scalable applications that exhibit good spatial locality. We realize that the relaxed messaging semantics can help here too: because each request will match to a known sender, and because a receiver knows for certain if a sender is on its same PE, we can optimize messaging for communication locality. First, we maintain a separate non-concurrent hash table at each rank for PE-local transfers. Second, we avoid the use of locks when accessing request objects that are PE-local. This is possible

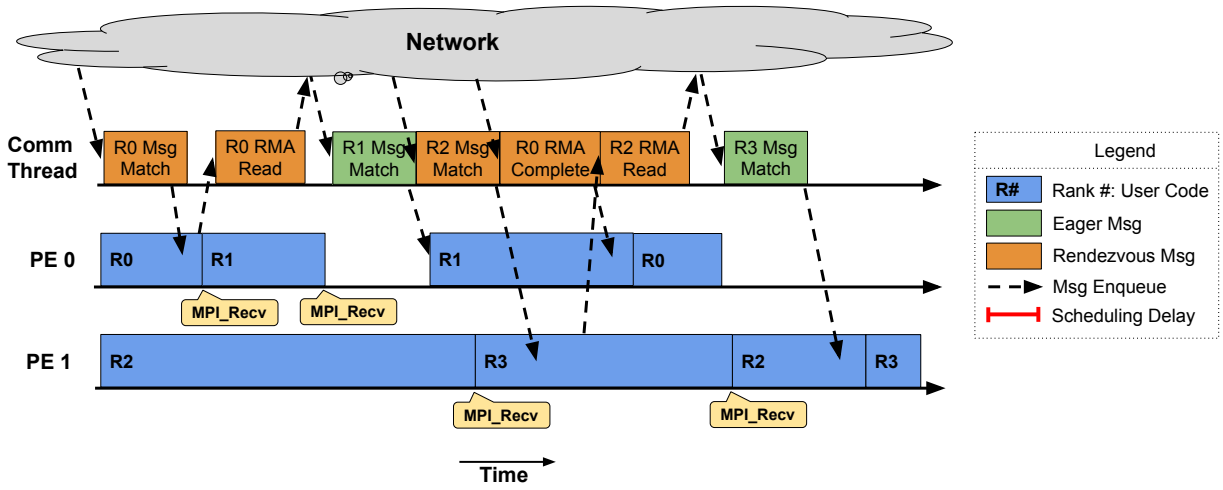



Figure 4.14: A Gantt chart showing execution of 4 ranks on a single node which are each receiving one message from off-node. Compare this to the original AMPI design to see where scheduling delays are avoided through asynchronous message matching and request completion.

because we know when creating and accessing a request the locality of the other rank that will fulfill it. Finally, we keep a separate non-atomic count of the number of PE-local requests blocked on. When a PE-local request is completed, we decrement that counter instead of the atomic one, until the PE-local count hits zero, at which point we subtract the total number of PE-local requests from the atomic count. When blocked inside a multiple completion routine such as MPI_Waitall for a combination of $n$ PE-local requests and $m$ non-PE-local requests, rather than $n + m$ atomic decrements this optimization results in at most $m+1$ atomic updates by batching the PE-local updates into a single atomic subtraction. We refer to this as locality-aware message matching, and note that the idea could be extended for awareness of other locality domains where concurrent access to the hash table might be able to be optimized differently: for instance, one could imagine separating the concurrent hash table out further, handling intranode transfers independently from internode ones, with possible hardware (SmartNIC) or software (interprocess shared memory) support for matching either.

Overall, this solution provides for concurrent message matching and asynchronous request completion for applications that can tolerate the relaxed semantics of no wildcard receives and overtaking messages. It allows intranode and internode messages to match and complete independently of one another, with optimizations for PE-local messaging as well. Potential drawbacks include increased single message latency based on overheads associated with increased concurrent shared memory data accesses, increased load on the communi-

cation thread, and loss of locality for intranode rendezvous messages when the receive has been preposted and the sender then completes the payload transfer synchronously using an inline memcpy.

### 4.3.3 Results

We compared the performance of our various runtime optimizations on a set of benchmarks, mini-apps, and production applications that have different communication patterns. We looked at overall execution time as well as communication time with varying degrees of overdecomposition (the number of ranks per PE). We compare AMPI (original implementation) against AMPI-async (scheduling and asynchronous request completion) and AMPI-relaxed (concurrent message matching and asynchronous request completion).

We used the Cori supercomputer at NERSC. Cori is a Cray XC40 supercomputer comprised of 2388 Intel Haswell nodes and 9688 Intel Knights Landing nodes. We use the Haswell nodes only. Each Haswell node contains 128 GB of memory and 32 cores split between two sockets. We used Charm++'s GNI networking layer on the Cray Aries interconnect, GCC version 11.2.0, and the SMP build of AMPI to run with one process per socket.

For benchmarks, we used the OSU MPI Benchmark suite version 5.9 as well as a range of mini-apps with different communication patterns, as well as a production application. We selected the benchmarks based on them not using wildcard receives and not requiring non-overtaking message semantics, as well as having a diverse set of communication patterns. The mini- or proxy-applications we use are Kripke, LULESH, MiniMD, MiniFE, and MiniGhost. Kripke and LULESH are both proxy applications developed at Livermore National Laboratory to mimic the computational workload and communication patterns of production applications. Kripke is a proxy for 3D Sn deterministic particle transport codes. LULESH approximates the performance characteristics of an Unstructured Lagrangian Explicit Shock Hydrodynamics code. MiniMD, MiniFE, and MiniGhost are mini-applications distributed as part of the Mantevo benchmark suite. MiniMD is a molecular dynamics mini-app derived from the LAMMPS application. MiniFE is a proxy application for unstructured implicit finite element codes. MiniGhost is a mini-app implementing a 3D nearest neighbor halo exchange, a common pattern in spatially decomposed HPC codes. Our production application showcase is PlasCom2, a plasma-coupled combustion simulation code developed by the DOE PSAAPII Center for Exascale Simulation of Plasma-Coupled Combustion at the University of Illinois at Urbana-Champaign. It implements a 27-point stencil with support for non-periodic boundary conditions.

### Point-to-Point Message Latency

We first measure the latency of a single point-to-point message transfer for various message sizes and localities of sender/receiver. We show results for message transfers between ranks on the same PE (P1) and across nodes (P2), as well as for the original AMPI implementation (AMPI) and both the optimized versions for the full semantics (AMPI-async) and relaxed semantics (AMPI-relaxed). We note that our optimizations are not targeting latency improvement. Still, the results show that our work does not harm single message latency significantly, the largest slowdown being 30% or 280 nanoseconds slower for very small messages between ranks on the same PE.

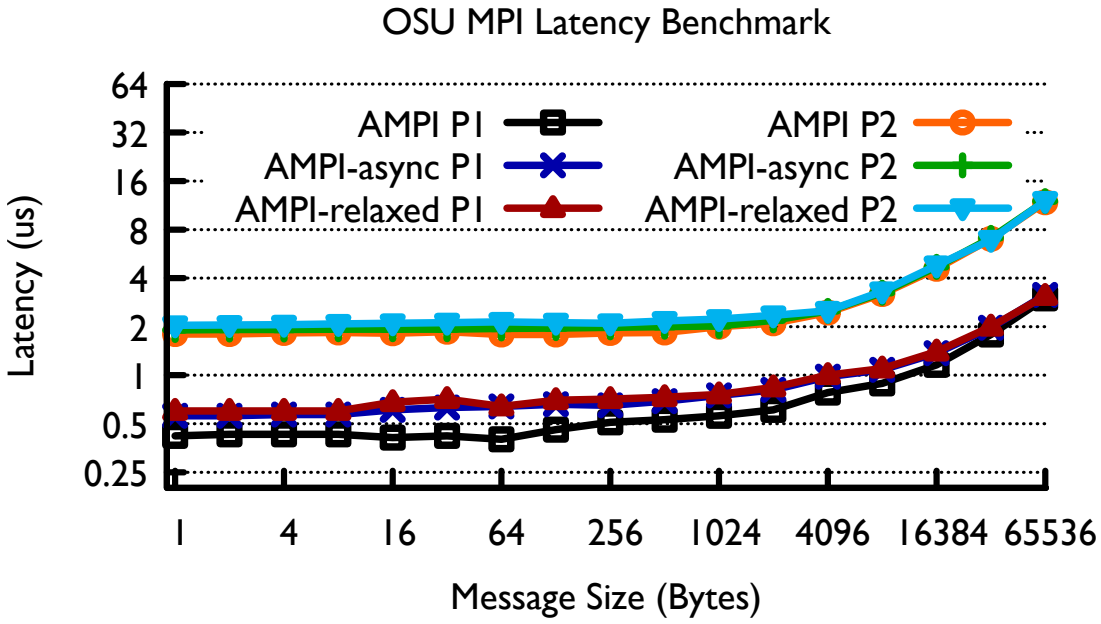



Figure 4.15: One-way MPI point-to-point message latency for AMPI (original), AMPI-async (asynchronous request completion and scheduling optimizations), and AMPI-relaxed (concurrent message matching, asynchronous request completion, locality-aware optimizations). Lower is better.

### Point-to-Point Message Rate

We next measured the message rate for various message sizes on 2 PEs. Each of the sending ranks sends a fixed number of messages (64) with different tags back-to-back to the paired receiving rank before waiting for a reply from the receiver rank. For AMPI-relaxed, the main benefit here comes for rendezvous messages (which are 13% faster at 16 KB), where message matching is done directly from the communication thread (eliminating a trip through the MPSC and scheduler queues) and doesn't require message ordering or walking

the message matching queues. AMPI-async closely matches the baseline, which aligns with our expectations given the benchmark's lack of computational work to overlap and all in-flight messages using the same communication protocol, nullifying the impact of prioritized scheduling.

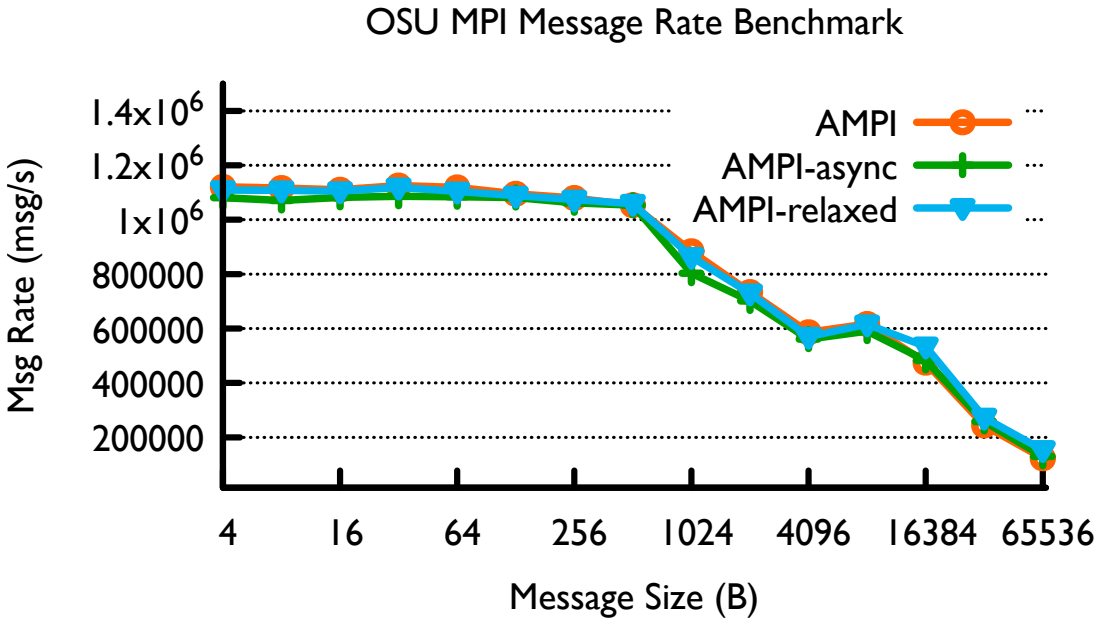



Figure 4.16: MPI point-to-point message rate for 2 ranks in separate processes (P2). Higher is better.

### Proxy Application Performance

We next look at overall application performance for the proxy applications and our different runtime optimizations. We run on 8 nodes of Cori, with two processes per node (one per socket). We compare performance of both optimized versions of AMPI against the baseline implementation, but this time we show results broken down for each optimization, and for runs without virtualization and with 8x virtualization (8 ranks per PE).

The results for our optimizations are mixed when running without virtualization. Kripke and MiniMD close to the same or worse. Kripke has a unique communication pattern with few communication neighbors (send to three neighbors, receive from three others) and a long critical path that makes it latency-bound. MiniMD uses blocking send/receive operations and no multiple completion routines, meaning the runtime has little opportunity to overlap multiple messages in-flight simultaneously for each rank. The other proxy applications see modest benefits without virtualization or overdecomposition. MiniFE's communication pattern is composed of blocking sends and non-blocking receives with single completion

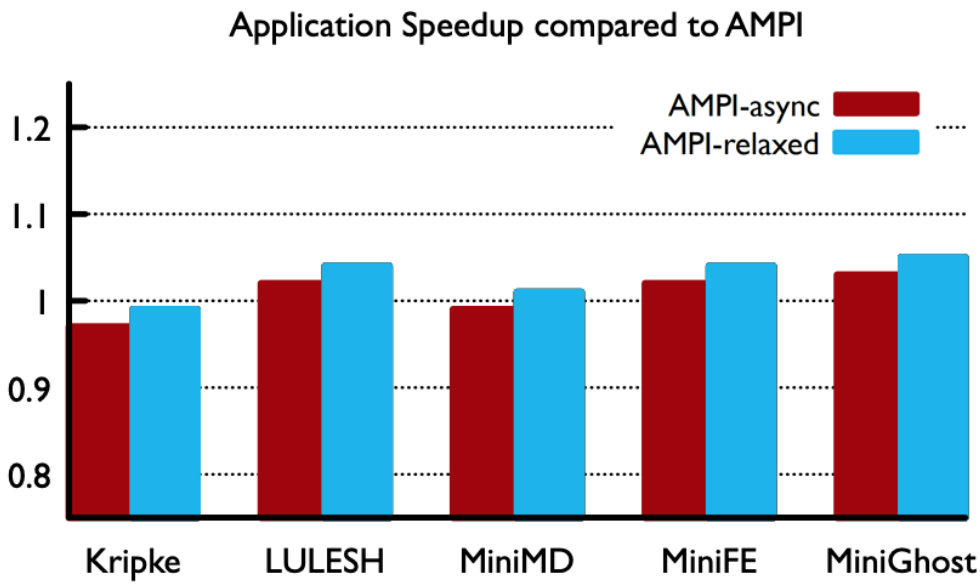



Figure 4.17: Speedup of AMPI-async and AMPI-relaxed normalized to baseline AMPI for mini-apps without virtualization. Higher is better.

routines (*MPI_Wait*): this allows at least the receive-side matching and completions for different messages to overlap. MiniGhost and LULESH see larger improvements, up to 5% for AMPI-relaxed without virtualization.

With virtualization, the results are more encouraging. With more ranks on each core, the runtime system has more computational work to overlap communication with on each PE. In the baseline AMPI, however, messages are not prioritized over running other ranks, which meant messages could sit in queues for extended periods of time before being matched. Now with AMPI-async optimizations such as prioritized scheduling and locality-aware thread resumption, rendezvous messages are prioritized over eager messages and running other ranks that are ready to resume on the same PE. Overall, we see AMPI-async and AMPI-relaxed both improve performance for all applications with 8x virtualization, with AMPI-relaxed performing best: up to 12% faster for LULESH.

Looking into the performance more, we observe that speedup generally tends to increase as the number of messages simultaneously in-flight increases. This points to faster matching and better overlap. MiniGhost and LULESH see the biggest performance improvements. MiniGhost has six neighbors that it communicates with all at once, while LULESH has 26. MiniGhost uses non-blocking sends and receives and iterates over calls to *MPI_Waitany* for completion of the up to 12 requests. LULESH uses non-blocking sends and receives, with a single *MPI_Waitall* call to complete all 26 send requests and individual *MPI_Wait* calls to complete each of the up to 26 receives.

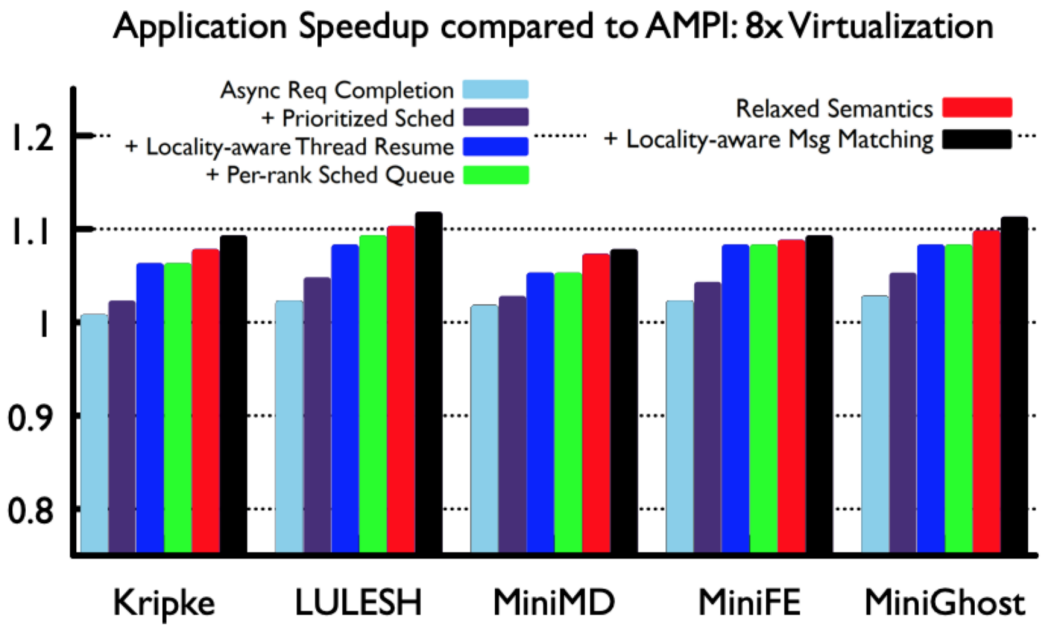



Figure 4.18: Speedup of AMPI-async and AMPI-relaxed normalized to baseline AMPI for mini-apps with 8x virtualization. We further breakdown performance based on optimizations included in AMPI-async and AMPI-relaxed to show their effects. Higher is better.

We also looked at the number of user-level thread context switches that occur during program execution. Part of the promise of making the runtime more asynchronous and concurrent is that if messages can arrive and match earlier, then they can potentially complete immediately upon posting a receive. MiniGhost sees the biggest gain here: with 8x virtualization, AMPI-relaxed incurs 7% fewer context switches than baseline AMPI. We believe the use of Waitany for completion allows AMPI-relaxed the best chance to have any one message pre-arrived, whereas use of Waitall or Wait makes context switch avoidance more difficult.

### Production Application Performance

Lastly, we look in more depth at the performance of two production simulation codes with respect to our various communication runtime optimizations. We explore the effects of scaling and look at how varying the degree of virtualization affects performance.

Our first production application is PlasCom2, which we described previously in Section 4.2.3 as a 27-point stencil code with support for overset meshes, non-periodic boundary conditions, as well as chemistry uncertainty quantification, and other multiphysics capabilities.

We look at PlasCom2's strong scaling performance with AMPI, AMPI-async, and AMPI-relaxed for an input case with 7.2 million grid-points. We notice that without virtualization

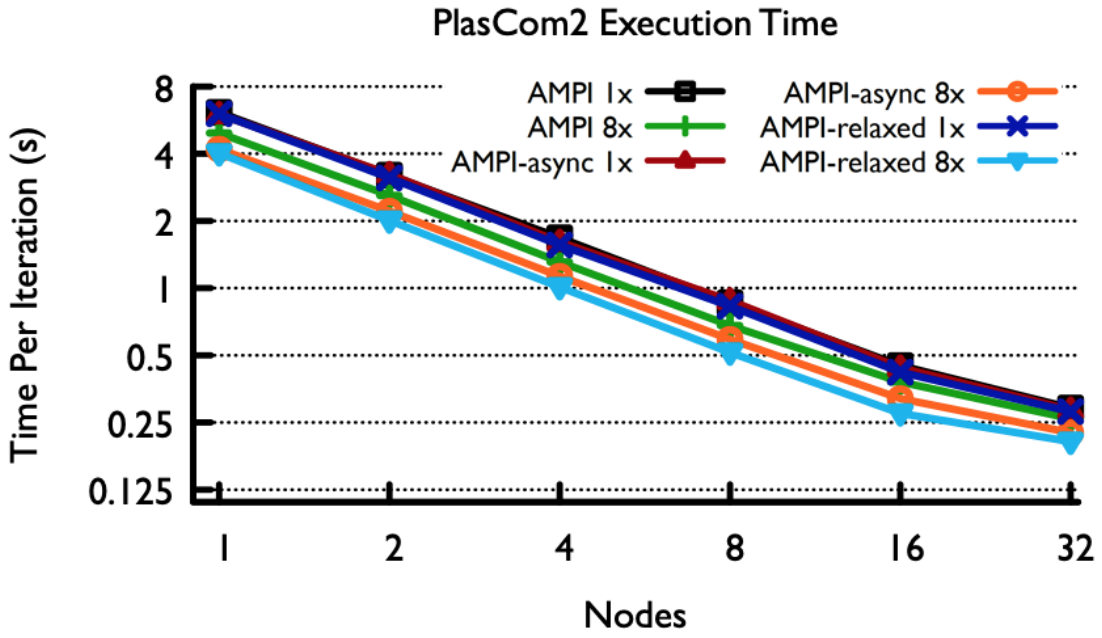



Figure 4.19: Strong scaling of PlasCom2 for baseline AMPI, AMPI-async, and AMPI-relaxed with 1x and 8x virtualization. Lower is better.

(1x), performance is improved by up to 2% for AMPI-async and 5% AMPI-relaxed over the baseline, but that the improvements mostly disappear as the problem size per rank decreases through strong scaling. With 8x virtualization, the performance improvement is more dramatic. This is similar to what we saw in the proxy applications, but the improvement is even greater in PlasCom2. The biggest differences between PlasCom2 and MiniGhost, a proxy application for stencil codes, are the halo exchange and boundary conditions. PlasCom2's halo exchange works by posting up to 26 non-blocking sends and up to 26 non-blocking receives before calling *MPI_Waitall* on all 52 requests. This differs from MiniGhost in the number of requests (52 vs 12) and in how they are waited on for completion, with PlasCom2 using Waitall. We observe that the performance improvement is likely not coming primarily from fewer context switches since PlasCom2 has only 2% fewer with AMPI-relaxed than AMPI.

Instead, we find that our asynchronous runtime variants are better at dealing with load imbalance arising from non-periodic boundary conditions. Load imbalance typically manifests in bulk-synchronous codes as idle time at synchronization points on underloaded PEs. This knock-on effect can be seen in the halo exchange, for example, when a rendezvous message's receive-side matching is delayed on an overloaded PE, which in turn then blocks the sender on an underloaded PE. Prioritized scheduling helps mitigate these effects, and concurrent message matching avoids it further by matching messages out of band on the communica-

tion thread. This is confirmed by looking at the idle time (time during which no rank is running and no Charm++ entry methods are executing) on each PE. In baseline AMPI, the maximum idle time on any PE is 18%, while the average idle time across all PEs is 13%. In AMPI-async, the maximum is 13% with the average 9%, while for AMPI-relaxed the maximum is 13% and the average is 10%. This suggests that both optimized runtime versions are able to minimize the effects of load imbalance through more asynchronous communication handling.

In all, PlasCom2 runs up to 1.68x faster on AMPI-relaxed with 8x virtualization than it does on the original AMPI without virtualization. Compared to baseline AMPI with 8x virtualization, it is 1.32x faster through more efficient, asynchronous communication. This much more significant speedup compared to the proxy applications is due to PlasCom2's bulk completion of all halo exchange requests using a single Waitall statement, load imbalance across ranks, and communication with up to 26 neighbors.

Our second production application is LAMMPS. In chapter 2, we compared PIEglobals performance to that of TLSglobals using a shale oil/gas recovery simulation with dynamic load imbalance. Here we run the same input case on AMPI-async to compare its performance to baseline AMPI. LAMMPS does not work with the relaxed semantics. All runs with virtualization are also running with dynamic load balancing using our built-in DistributedLB strategy. We show results for 1, 10, and 20 times virtualization. On baseline AMPI, 10x virtualization performs best, but with AMPI-async, we see 20x virtualization outperform that by 5%. Looking at context switch counts, AMPI-async performs 12% fewer ULT context switches at 20x virtualization than AMPI at 144 cores. In terms of load balance, average utilization across all PEs increases from 66% to 72%.

## 4.4 CONCLUSION

The increasingly powerful nodes of exascale class supercomputers are putting a renewed emphasis on efficient data movement support in programming models and distributed runtime systems in order to keep up with their computational speeds. Latency, bandwidth, asynchrony, and concurrency are all important factors in the communication performance. For multi-threaded communication runtime developers these issues present unique challenges in terms of trade-offs in one dimension versus another, as well as ease of implementation and use by applications.

In this chapter, we have optimized AMPI point-to-point communication by taking advantage of the shared address space between ranks on a node. We have examined the consequences of MPI's messaging semantics on a particular thread-based MPI implementation,

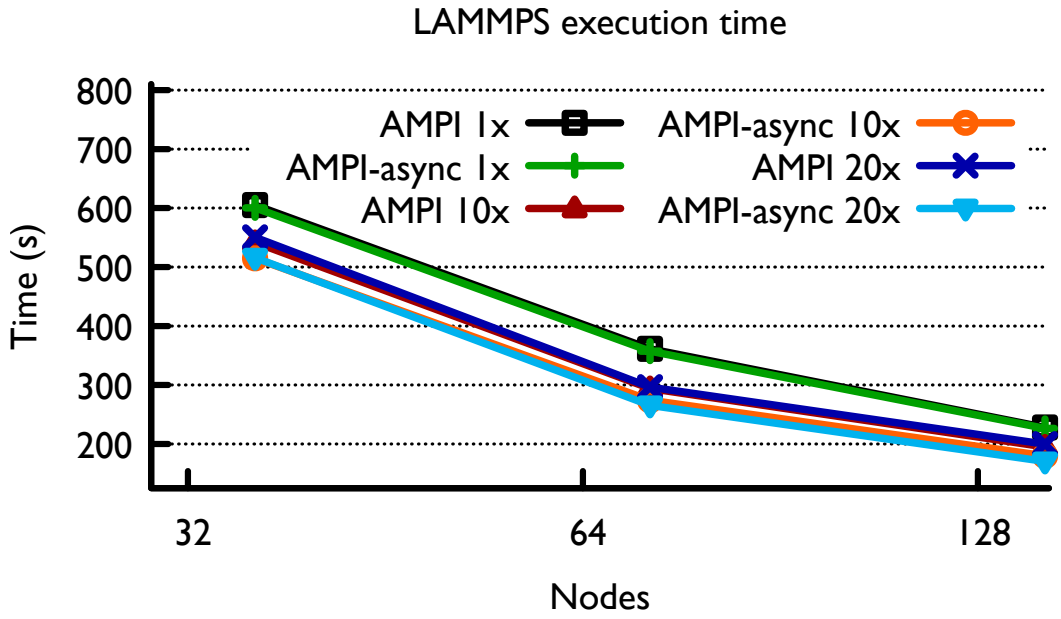



Figure 4.20: Strong scaling of LAMMPS USER-MESO for baseline AMPI and AMPI-async with 1, 10, and 20 times overdecomposition. Lower is better.

AMPI, and investigated how its design can be improved separately for applications that do and do not require the full MPI messaging semantics. Within the full semantics, making use of the shared address space to complete RDMA operations asynchronously through the dedicated communication thread and optimizing scheduling to prioritize rendezvous message matching improve are profitable for most applications we tested and for all when run with overdecomposition.

We also showed that the challenges for implementing more asynchronous and concurrent message matching in AMPI reduce to similar problems first identified in the context of efficient MPI+X support. There the focus was on efficient support for thread level *MPI_THREAD_MULTIPLE* and asynchronous progress, while here we build on that work to improve AMPI's support for asynchrony and concurrent communication handling. Taking advantage of relaxed messaging semantics, we implement efficient support for concurrent message matching while still optimizing for communication locality. Overall, we see improved communication asynchrony resulting in a production application speedup of up to 1.68x for runs with 8x rank virtualization compared to the baseline without virtualization, without requiring any code changes to the application.

For future work, we plan on making these semantic optimizations on a per-communicator basis, in accordance with the MPI 4.0 standard's info objects. This would allow more ap-

plications to take advantage of relaxed semantics for certain phases of an application or in libraries, and would expand the set of codes to which we could apply our optimized runtime system. This would allow us to support *MPI_Dist_graph_create*, which currently relies internally on wildcard message receives. We also plan to support performance visualization of AMPI-relaxed programs: currently all of the shared memory optimizations are effectively invisible to Charm++'s runtime tracing. Lastly, we would like to push the concurrency in AMPI communication downwards through Charm++. Currently, Charm++ is restricted to having one dedicated communication thread per process. With AMPI performing more work directly on the communication thread, we expect that this thread could become overloaded and so benefit from multiple such threads per process, though more performance analysis is needed. We can also push the concurrent hash table down into Charm++ so that it can implement the zero copy Post API more efficiently (AMPI uses the lower-level Direct API currently). The Post API's messaging semantics already do not allow wildcard tags and do not enforce ordering of messages between chares.

# CHAPTER 5: COLLECTIVE COMMUNICATION OPTIMIZATIONS

## 5.1 INTRODUCTION

Optimization of MPI collective communication routines has been well studied. Most of this work has focused on particular routines and scalable algorithms for them, optimization for particular networks, different node architectures, and message sizes [58] [59] [60] [61] [62] [63]. There has also been previous work on the effect of different (static) process placements on collective communication performance as well [64] [65] [66]. Since most MPI implementations are process-based libraries, much of this work has concentrated on that traditional execution model wherein each rank has its own separate address space.

In this chapter, we focus on collective communication routines in the face of virtualized, migratable ranks. Each rank is its own separate MPI endpoint, and there they may be tens, hundreds, or even thousands per node with overdecomposition. Further, each rank may migrate from node to node dynamically during execution. We study collective communication in AMPI's execution environment in order to identify and ameliorate performance issues caused the model. However, the issues are relevant to any parallel programming system that supports both migratable work units and non-commutative collective operations on them. First, we evaluate the costs of rank virtualization with respect to different collective routines and design virtualization and shared address space aware collective algorithms. Next, we identify the costs of dynamic rank migration as they relate to collective communication, particularly non-commutative reduction operations. Then, we demonstrate the shortcomings of various non-commutative allreduce algorithms in the face of dynamic rank migration. Finally, we propose and implement support for rank-placement adaptive, shared-memory aware non-commutative allreduce operations in AMPI, and demonstrate performance improvements using microbenchmarks.

As mentioned previously, AMPI supports virtualization of AMPI ranks as well as dynamic rank migration. Rank virtualization naturally leads to there being more ranks with their own communication endpoints. Typically virtualization is used for performance reasons– to tolerate communication latency, for transparent cache blocking benefits, or to enable dynamic load balancing. Rank migration involves serializing all of a rank's data– its user-level thread stack, all heap-allocated memory belonging to it, and the MPI library's state associated with it– sending that over the network, and unpacking it before resuming execution on a different core or node. AMPI links its own memory allocator, called Isomalloc, into the application to accomplish this. Isomalloc reserves a unique slice of the global virtual memory for each

virtual rank in each process. This enables pointers to work transparently across migrations between nodes, since all memory is re-allocated at the same virtual memory addresses it was at on the previous node.

Rank migration is used primarily for dynamic load balancing, with the runtime system monitoring load and invoking a user-selected load rebalancing strategy when imbalance goes over a threshold. Load balancing strategies can be centralized, distributed, or hierarchical. Centralized strategies can make use of accurate global information but require synchronization which can be costly. Distributed rebalancing algorithms sacrifice some global knowledge for reduced synchronization costs. In production use, refinement-based strategies are common, because they take into account the current mapping of ranks and seek to move as few ranks as possible while minimizing the load imbalance. This is in contrast to greedy algorithms for rebalancing, which generally try to maximize balance without regard to initial placement of ranks. Load balancing is typically infrequent in scientific simulations, where load imbalance tends to evolve slowly over the course of many timesteps.

In addition to load balancing, AMPI leverages its rank migration capability for supporting fault tolerance, automatic checkpoint/restart, power/energy optimizations, and resource elasticity by allowing changing the number of nodes assigned to an AMPI job.

AMPI relies on Charm++'s dynamic location management protocol for routing messages to the correct places in the face of migration. Essentially Charm++ maintains a distributed hash table of locations, and has a protocol for forwarding messages to the correct location when they do arrive late to a node which no longer contains the recipient rank. AMPI and Charm++ optimize communication for locality and shared memory, though they fall back to slower protocols when their location records are out of date. Typically the location records are stale only the first time-step directly following load balancing for ranks that migrated and the location caches are updated then. We also note that location records are always up to date for PE-local ranks, and a node-level cache that is always up to date is possible to implement, though not supported by Charm++ currently since its location management is done at the PE or core level.

We have also added support to AMPI to read the initial mapping of virtual ranks to cores from a user-defined file at startup. Both this and dynamic load balancing can lead to the natural ordering of ranks to cores being permuted, and this change–its effect on the performance of collective communication routines– is what we examine here. We use the file mapping method to simulate the effects of different dynamic load balancing scenarios.

Collective communication routines allow applications to express their communication patterns succinctly, at a higher abstraction level than point-to-point routines, and they afford the runtime system the ability to optimize the operation for different inputs and hardware

systems. AMPI implements all of the MPI standard's collective communication routines. For routines that it can implement on top of Charm++ collective communication primitives, that is preferred. All communication in Charm++ is non-blocking, including collectives. Charm++ supports tree-based collectives such as broadcasts, reductions, and gather operations. For collective operations such as broadcast and commutative allreduce, where the output is the same across all ranks and the ordering of contributions is irrelevant, performance is not significantly affected by migrations. This follows logically from the fact that all ranks receive the same output data, and no special care has to be taken to reorder messages that contribute to the result.

For routines that Charm++ does not natively support (such as scatter and all-to-all), AMPI uses point-to-point messages for its implementation. Implementing MPI collectives in AMPI on point-to-point communication routines is generally not favored because it involves MPI message matching, which requires synchronization at the endpoints as compared to a tree-based collective routine which can more easily offload the work to the communication thread in each process using asynchronous Charm++ messages. Synchronizing at the endpoints means messages must be routed to the PE and then be inserted into the scheduler's message queue in order to eventually be processed on that PE, as discussed in Chapter 4.

## 5.2 VIRTUALIZATION AND SHARED ADDRESS SPACE AWARE COLLECTIVES

We first look at the effects of virtualization before migration. For reductions, the MPI standard defines various predefined reduction operations and the ability for users to create their own custom operators. MPI_Op's can be used with a variety of routines (MPI_Allreduce, MPI_Reduce, MPI_Reduce_scatter, MPI_Scan, MPI_Accumulate, etc.) defined by the standard. All MPI_Op's must be associative. All predefined MPI_Op's are commutative. But when the user creates an operator they can tell the MPI library whether or not it is commutative. There is no such option for associativity. This is due to the parallel performance considerations of reduction operations. If an operation were not associative, the operation would have to be performed serially. If an operation is associative but non-commutative, it affords the MPI implementation the ability to combine ordered subsets of contributions, as in a tree. If an operation is both commutative and associative, the runtime can combine contributions on the fly no without regard to the ordering. Accordingly, the MPI standard encourages users to prefer the use of predefined or commutative operators where possible for the best performance, but still defines support for non-commutative reduction operations because they can be optimized by the runtime in parallel.

Table 5.1: Allreduce latency in microseconds ($\mu$s) for different reduction operations (commutative vs. non-commutative), degrees of virtualization (1 rank per core vs. 8x), and virtual rank mappings (block vs. random) on 32 nodes of Cori (NERSC) for a 4 byte message size.

| **Reduction Op** | **1x Block Mapping** | **8x Block Mapping** | **1x Random Mapping** | **8x Random Mapping** |
|---|---|---|---|---|
| **Commutative** | 59.14 | 67.08 | 61.36 | 68.94 |
| **Non-commutative** | 161.32 | 193.86 | 233.37 | 391.54 |

Based on these definitions, we hypothesize that permutations to the rank ordering should effect the performance of non-commutative reduction operations but not commutative ones.

Table 1 shows the performance of a commutative allreduce compared to a non-commutative allreduce in AMPI for two different mappings: block and random. For simplicity, we use MPI_SUM for the commutative operation and we create a user-defined operation that also computes the sum for the non-commutative operation. The mapping is kept constant throughout the benchmark's main iterative loop, to avoid measuring migration overheads themselves. Block mapping is the natural mapping that most all applications start from, while random can be considered an extreme example of a mapping after a greedy rebalancing strategy has run. it also shows the effect of virtualization, with either 1 rank per core or 8 ranks per core. The non-commutative allreduce uses recursive doubling implemented on top of AMPI point-to-point messaging routines. Recursive doubling is a commonly used, bandwidth optimal algorithm for Allreduce which behaves well for commutative and non-commutative operations alike [67]. The commutative reduction operation, on the other hand, is implemented using a Charm++ reduction, which optimizes for commutativity by combining messages opportunistically at each PE first, then at the node-level, and finally using a spanning tree across nodes.

The results show that for commutative reduction operations such as the predefined ones, performance is not substantially affected by changes in the mapping. However, for a non-commutative operation implemented on point-to-point messages, performance relies heavily on the mapping: whereas a commutative reduction suffers a roughly 3% slowdown going from a block to random mapping at 8x virtualization, a non-commutative reduction suffers a more than 2x slowdown for the same comparison! This result is expected for a point-to-point based implementation which does not well optimize for locality between ranks and requires synchronization at the virtual rank endpoints at each step of the algorithm.

This supports our hypothesis that commutative reductions are mostly impervious to changes in the rank mapping, since messages can be combined out of order, while non-commutative operations can suffer substantial performance degradation under mapping

changes.

We also note that Charm++ does not include support for non-commutative reductions or ordered gather operations, though they can be implemented on top of Charm++ as an unordered gather operation followed by sorting and applying the contributions in rank-order. This requires sending along the rank of the contributor for each data contribution, in order to be able to sort contributions by rank order at the root. For this purpose, Charm++ supports so-called "tuple" reductions which can perform different reduction operations on different data fields within the same message.

AMPI's execution model, compared to traditional process-based MPI implementations, provides unique opportunities for shared memory parallelism. Because there are multiple virtual ranks on each PE, and multiple PEs per process, tens or hundreds of AMPI ranks will often share the same address space. This enables fast shared memory access between ranks. We previously optimized AMPI's implementation for in-place collective communication support in Chapter 3 and for fast intraprocess point-to-point communication in Chapter 4.

For collective operations, we focus first on the commonly used broadcast and allreduce operations. These operations are not personalized and the results in no way depend on the ordering of rank contributions for predefined or commutative user-defined reduction operators, unlike in scatter, gather, alltoall, or non-commutative allreduce. Allreduce is commonly used in scientific simulation codes and as such has seen extensive research on optimizing its performance for different networks, architectures, message sizes, and node counts. Here, we optimize for shared address space between ranks.

For broadcast operations, we realized that Charm++ messaging semantics were limiting performance. For large messages, we developed support for in-place broadcast operations on both the sender and receiver sides, essentially adding completion callbacks to the interfaces so that users can get control back from the runtime after the runtime is finished with the buffer. Without this, Charm++ semantics necessitated copies of the buffer on both sender and receiver. For small and medium messages, where we can tolerate buffering the message, the problem is different: On the receive-side, Charm++ semantics require the runtime to hand off ownership of the buffer to each recipient chare. Each chare is free to then modify the contents of the message as it sees fit. This requires the runtime to deliver unique copies of the message object to each chare. For $nc$ chares on $p$ PEs of $n$ nodes, this requires the allocation, copy, and delivery of $c$ unique messages. Theoretically, since the output buffer is the same across all chares, we should be able to create only $n$ messages if the application does not require the ability to modify the message buffer. This required changing the reference count field of message objects in Charm++ to be atomic and defining new semantics around

message modification. We could have added a new message attribute type to Charm++, but we found that we could strengthen the semantics of the *nokeep* entry method attribute to specify not only that the application did not want or need to take ownership of the message object but that it would not modify the contents of the message as well. In practice, *nokeep* entry methods are most often used within Charm++'s implementation when the application is using parameter marshalled entry methods or else will be copying the contents of the message into another data structure and immediately releasing or freeing the message itself. Taking advantage of this new semantic and atomic reference counting, we developed support for making only $n$ total copies of a broadcast message that is being delivered to $c$ chares on $p$ PEs of $n$ nodes.

In AMPI, we then implemented small and medium message broadcast using a broadcast to a *nokeep* entry method of the chare array associated with a communicator. This results in one message object being allocated per process and a pointer to that message being delivered to each rank's chare within that process. From there AMPI deserializes the contents of the message into the user's output buffer. This effectively minimizes memory allocations and copies associated with per-rank message creation. We still use the separate zero copy broadcast API for large messages in order to avoid the one intermediate copy per node associated with the *nokeep* method. Figure 5.1 shows the latency of broadcasts for different message sizes on 32 nodes of the Quartz cluster at Lawrence Livermore National Laboratory with and without virtualization of 16 times the number of PEs.

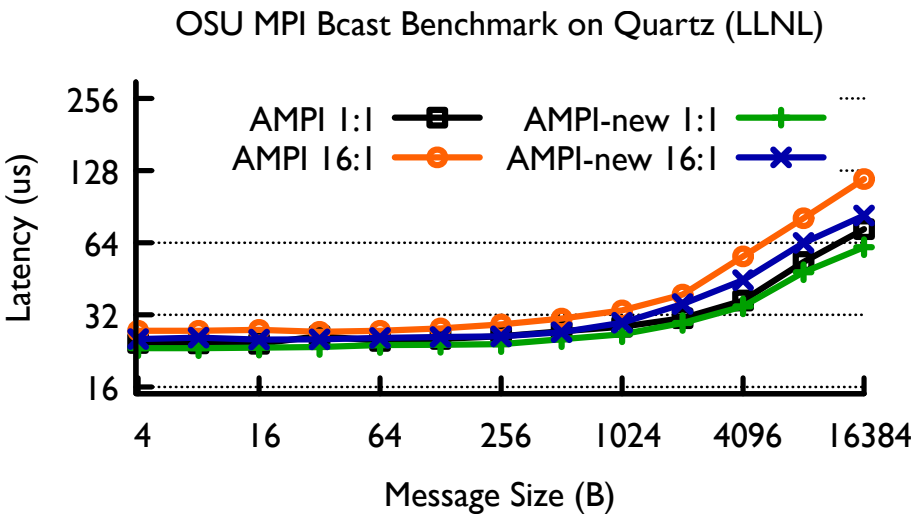



Figure 5.1: OSU MPI Broadcast benchmark results on 32 nodes of Quartz for various message sizes, both with and without 16x rank virtualization and our shared address space optimizations. Lower is better.

For commutative allreduce operations, the same optimization principle applies. The difference with reduction is that we need to apply the operator. For process-based MPI libraries, most interprocess copy mechanisms such as CMA, KNEM, and LIMIC require a copy rather than being able to directly access the shared buffer to apply the reduction operator. XPMEM does support this mode of shared memory access, but it requires explicit registration and deregistration of the shared buffers, which results in system calls and registration overhead. In AMPI, we can directly exploit the shared memory between ranks to apply the operators directly from their source buffers to a result buffer, and there is no need for system calls to register the memory upfront or to keep track of which memory regions are registered.

For predefined MPI_Op's, which are most commonly used, all operations are both commutative and associative, so the contributions can be combined in any order. We exploit both of these properties by applying the operators first within each PE. We do this directly from the user input buffer. This required changes to Charm++'s reduction interface, which is based on all reduction buffers being encapsulated inside *CkReductionMsg* objects, resulting in extraneous intermediate memory allocations and copies. After we reduce locally within a PE, we then reduce to a single buffer within the shared address space. For small messages, we use a single mutex lock, while for large buffers we use one lock per segment of the buffer, with the size of the segment being a tunable parameter, and stagger PEs so that they start from different segment offsets in the large buffer and wrap around it. This is all done directly through the shared address space rather than relying on the PE-level scheduler queues for inter-PE synchronization. Figure 5.2 shows results for an intranode allreduce on Quartz with 10 virtual ranks per PE. Our virtualization- and shared address space-aware approach is 32% faster than previous for small messages (less than 64 bytes), and up to 94% faster for messages larger than 16 KB.

## 5.3 NON-COMMUTATIVE REDUCTION OPERATIONS

Non-commutative allreduce operations pose an interesting challenge to our runtime in that we expect the mapping of ranks to cores to impact their performance. We have already seen a comparison of AMPI's current commutative reduction implementation to the point-to-point based recursive doubling implementation for non-commutative operations. Table 1 illustrates the overheads of virtualization, specifically how much more expensive virtualization is for non-commutative allreduce than commutative operations. This follows from the fact that commutative reduction contributions from multiple ranks on the same PE can be combined in any order first before performing the intranode and then internode allreduce. However, we need to separate out the cost of non-commutativity from the difference between point-

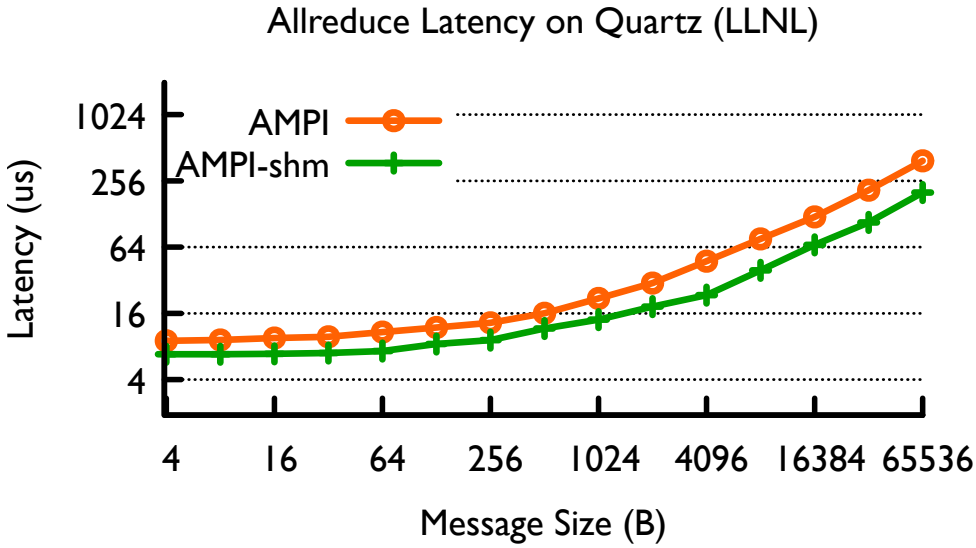



Figure 5.2: OSU MPI Allreduce latency comparison of shared memory-aware and baseline AMPI on 1 node of Quartz at LLNL. Lower is better.

to-point and shared memory-aware implementations.

For non-commutative reductions, we can make the implementation of recursive doubling node-aware, so that it takes advantage of shared memory between all virtual ranks in a node. To make more effective use of shared memory, we had to modify Charm++'s reduction interface to allow us to avoid copying messages multiple times within each process. The reduction API in Charm++ is non-blocking and one-sided, requiring the runtime system to copy the message and return from the reduction call immediately. We added a user callback to the Charm++ reduction interface that is invoked when the source buffer is complete, allowing us to delay the copy until its preceding rank's contribution is known and the message can be combined directly from the user's message buffer without making an intermediate copy. Even with a block mapping, there is no guarantee that reduction contributions on each PE will happen in rank-order, so the ability to delay or avoid the copy is critical. For example, if a PE hosts ranks 50 - 60, if the first two contributions come from rank 52 and 55, they cannot be combined until contributions from rank 53 and 54 are also available. Our shared address space-aware implementation opportunistically combines contributions from adjacent ranks within each PE first, before then combining messages across PEs in the shared address space.

Figure 5.3 compares the latency of a point-to-point based implementation of recursive doubling to our shared memory-aware version. It shows that performance is improved significantly, up to 3.5x, over the point-to-point implementation, which performs poorly as

messages must be copied and sent explicitly between all virtual ranks before applying the reduction operator.

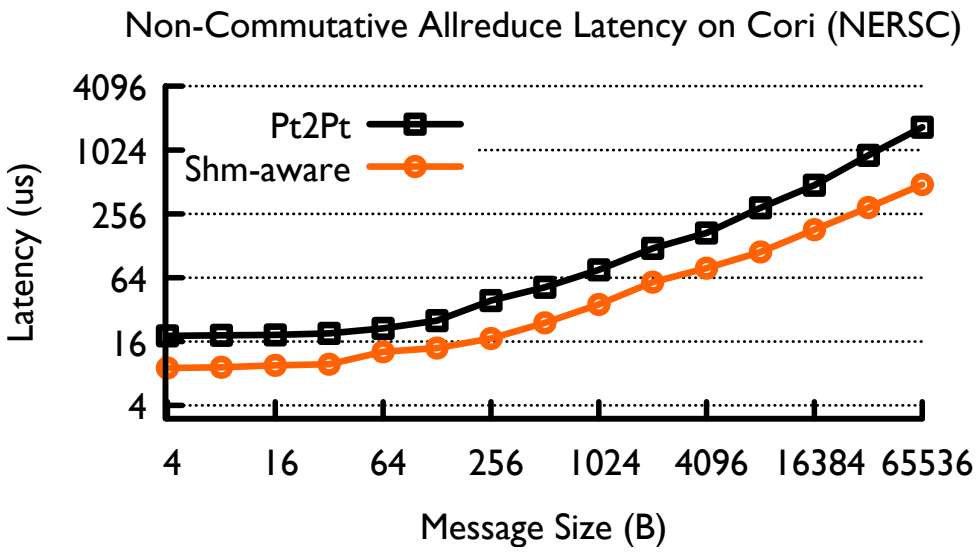



Figure 5.3: Comparison of shared memory-aware and point-to-point based recursive doubling for non-commutative allreduce on 1 node of Cori's Haswell partition. Lower is better.

This shared memory-aware implementation on its own will not implement non-commutative reductions correctly if ranks have migrated out of the natural block mapping across nodes. If ranks are out of block mapping, then we need to delay application of the reduction operator further, until those contributions can be processed with their neighboring ranks' contributions. And for that, our internode reduction algorithm needs to be able to adapt to out-of-order contributions.

### 5.3.1 Placement-Adaptive Non-Commutative Allreduce Algorithms

Building on the shared memory optimizations, we note that if ranks are not block mapped, we can still optimize for partial ordering. Since refinement-based load rebalancing strategies are common in practice, we expect that ranks will tend to stay close to a block mapping even if some migrate between nodes. We also want to optimize for the case where ranks migrate across PEs in the same node: this kind of intranode migration is cheaper than internode migration, and load balancing strategies that are topology-aware will favor it. Partial ordering of ranks then may be worth optimizing for in practice. We call an algorithm which can adapt to the degree of dynamic mapping change "adaptive" if it can optimize performance based on the degree of mapping change from block mapping.

In order to make our non-commutative allreduce implementation adaptive to mapping changes, we extend the shared memory optimization idea of delaying the message copy by extending the idea across nodes: if a message can not be combined with its predecessor's contribution since that rank is on a different node, we can forward its message, along with metadata identifying which rank's contribution it represents, with the other combined messages.

### Gather-Sort-Broadcast

An alternative approach is to implement non-commutative reductions using tuple reductions for a gather before sorting and applying reduction operation in rank-order before broadcasting the output buffer to all ranks. This approach minimizes the overheads associated with point-to-point communication from the many virtualized ranks that typically make up an AMPI application run. It also means that the efficiency of the operation is minimally affected by migrations skewing the rank ordering across nodes, since it does not do any message combining during the gather operation. However, that also means that this algorithm fails to take advantage of the case where ranks, or subsets of ranks, are actually ordered. In practice, with the use of refinement-based load balancing strategies, we expect for rank-ordering to still persist in part. Additionally, this approach also fails to make use of most cores and nodes while the data is gathered to the root, sorted and operators applied, and then broadcast back out. The sorting and application of the reduction operator becomes a serial bottleneck, with all other cores and nodes waiting for the result. We implement it as a baseline of sorts to compare the following two algorithms against.

### Recursive Doubling

Recursive doubling is a well-known algorithm for allreduce and allgather operations that avoids the serial bottleneck of the gather-broadcast pattern [67]. It does so by spreading the communication over more ranks which can communicate in parallel. It has been used in practice for commutative and non-commutative allreduce alike because it is bandwidth optimal. The total cost of a recursive doubling allreduce is $T = logp\alpha + nlogp\beta + nlogp\gamma$, where $p$ is the number of processors, $\alpha$ is the message latency, $\beta$ is the message bandwidth, $n$ is the message size, and $\gamma$ is the cost of the reduction operator.

It can be used for non-commutative allreduce because the communication pattern results in messages arriving in rank-order, allowing the implementation to apply the reduction operator at each step of the algorithm. For applications that use the default rank ordering

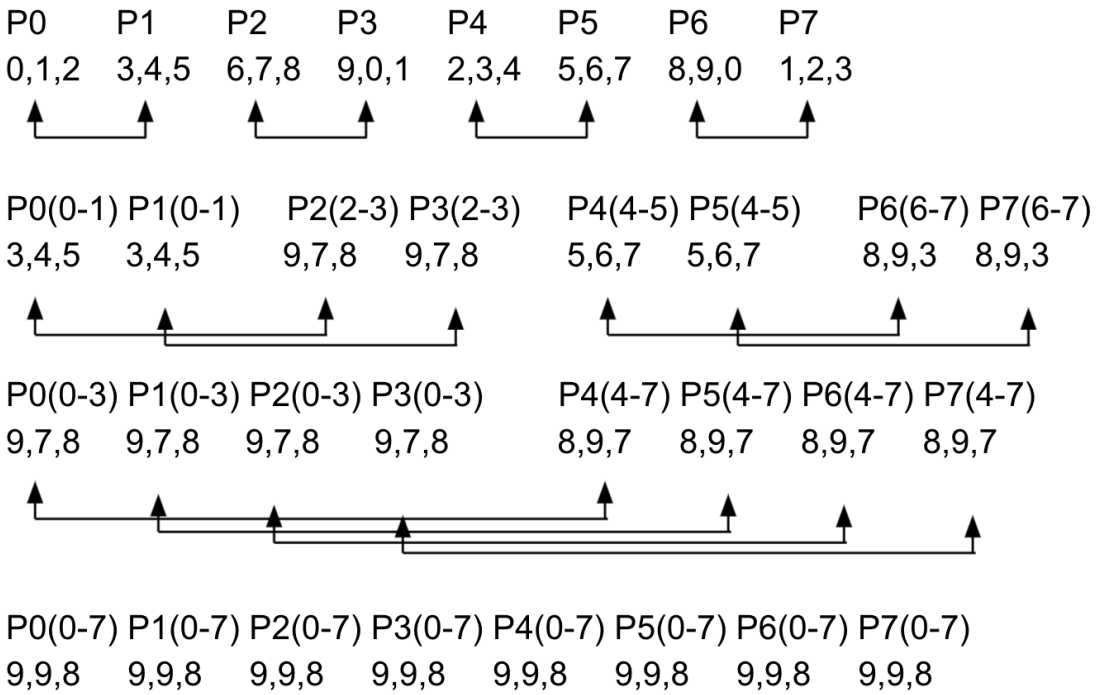



Figure 5.4: **Recursive doubling** communication pattern illustrated for 8 ranks in 3 steps. P0 is rank 0. P0(0-3) means that rank 0 has data from ranks 0 through 3. The three integers below each rank constitute their buffer at each step of the algorithm. Here we use the MPI_MAX operation for simplicity. Note than even if the operation were not commutative, the reduction operator can be applied at each step because messages arrive in rank-order.

and do not use any dynamic rank migration, we expect recursive doubling to perform well. This is illustrated in Figure 5.4.

For use cases where ranks are not nicely ordered across the cores and nodes of the system, as is the case after dynamic load balancing, we modify the recursive doubling algorithm to buffer out-of-order messages and send them along at each step. At each step of the algorithm we also attempt to combine as many subsets of contributions as possible. To do so, we maintain an extra buffer which indicates which ranks' data are represented by each buffer. This includes two integers for each contribution: the first representing the first rank whose contribution is contained in the buffer and the second representing the last. For example, if rank 7 has combined its buffer with that from ranks 4, 5, and 6, its tuple is denoted [4-7]. We refer to this metadata the "range encoding". This is depicted in Figure 5.4 as well. This opportunistic combining of messages results in the message size staying constant at each step of the algorithm so long as ranks are in their natural block mapping. Additionally, the algorithm can adapt to more randomized mappings as well, with graceful performance degradation. We also note that a full 32-bit integer is not always necessary to

store ranks in the range encoding: for instance, 16-bit integers can be used when the size of the communicator is less than or equal to 65,536.

### Recursive Halving

Recursive halving (or recursive doubling distance halving) reverses the communication pattern of recursive doubling. The key difference being that, for a gather operation, this results in exchanging the first messages, which are the smallest ones, to the farthest away partner node. At each step of the algorithm, messages double in size while the distance they are sent is halved. Consequently, this algorithm remains bandwidth-optimal (just as recursive doubling), yet achieves reduced network contention particularly for large messages, as shown in [60].

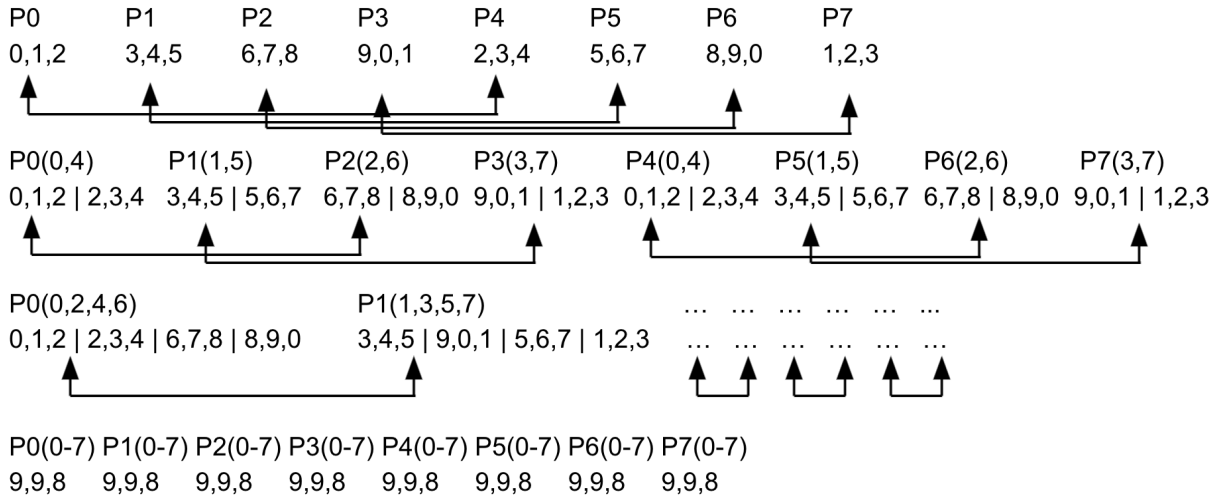



Figure 5.5: **Recursive halving** communication pattern illustrated for 8 ranks in 3 steps. Here we use the MPI_MAX operation for simplicity but illustrate the algorithm as if it were a non-commutative operation. This means that the operator cannot be applied at each step; instead, the runtime must buffer the out-of-order messages until they can be processed in rank-order later. Buffered messages are separated by vertical bars (—), and some ranks are excluded in the third step for space reasons.

MPI library developers have traditionally dismissed recursive halving as an implementation for non-commutative allreduce operations because its reversal of recursive doubling's communication pattern means that contributions arrive out of rank-order. As a result, the MPI library has to buffer those contributions until it receives all contributions or at least a contiguous range of contributions. This same logic applies to AMPI, but we note that rank migration can already nullify the natural rank-ordering that makes recursive doubling

attractive for non-commutative allreduce. With repeated load balancing or a heavily permuted mapping of ranks to cores, logically we might expect recursive halving to perform better than recursive doubling when the mapping of ranks to cores is either random or highly perturbed from rank-ordering.

Similar to recursive doubling, we implement recursive halving at the node-level in AMPI. This means that after all local contributions have been reduced as much as possible using the range encoding technique described in section B above, we use recursive halving between nodes to exchange data. Along the way, we reduce the data as much as possible at each step in the algorithm for contiguous ranges of contributions using the range encoding, in order to minimize the internode message sizes. Figure 5.5 illustrates the communication pattern, and how the message sizes increase as contributions arrive out-of-order.

We use the Cori supercomputer at NERSC for our performance measurements. Cori is a Cray XC40 supercomputer comprised of 2388 Intel Haswell nodes and 9688 Intel Knights Landing nodes. We use the Haswell nodes. Each contains 32 cores and 128 GB of memory [68]. We used Charm++'s GNI networking layer on the Cray Aries interconnect, and the SMP version of AMPI, which enables running with a dedicated communication thread per process, with multiple scheduler threads in each process. We run with 1 process per socket. We run with 8x virtualization, meaning there are 8 ranks per core, unless otherwise indicated.

We use the OSU MPI benchmark suite to measure the performance of Allreduce, only changing the MPI_SUM predefined op to a non-commutative op, which for consistency and simplicity, also computes the sum. In practice, a non-commutative operator may be more expensive to compute than this.

We first looked at scaling of the different algorithms for large messages with different mappings. Figure 5.6 shows the scaling results for the three algorithms for a 64KB size non-commutative allreduce using a block mapping. Here we see gather-sort-broadcast performs well at small scales but does not scale out as nicely as recursive doubling. Recursive doubling performs up to 5x better than both other algorithms at 32 nodes. This dramatic difference is due to the message size remaining constant for recursive doubling, while the others accumulate larger variable sized messages requiring dynamic memory allocation at each step of the algorithm.

Figure 5.7 shows the same scaling as Figure 5.6 but using a randomized mapping this time. The results suggest that recursive halving performs better at scale for this mapping and message sizes. Again gather-sort-broadcast performs well at small scales. With the random mapping recursive halving performs worst at scale, not being able to take advantage of message combining as effectively.

Next we look closer at the performance of our algorithms across more nuanced mappings

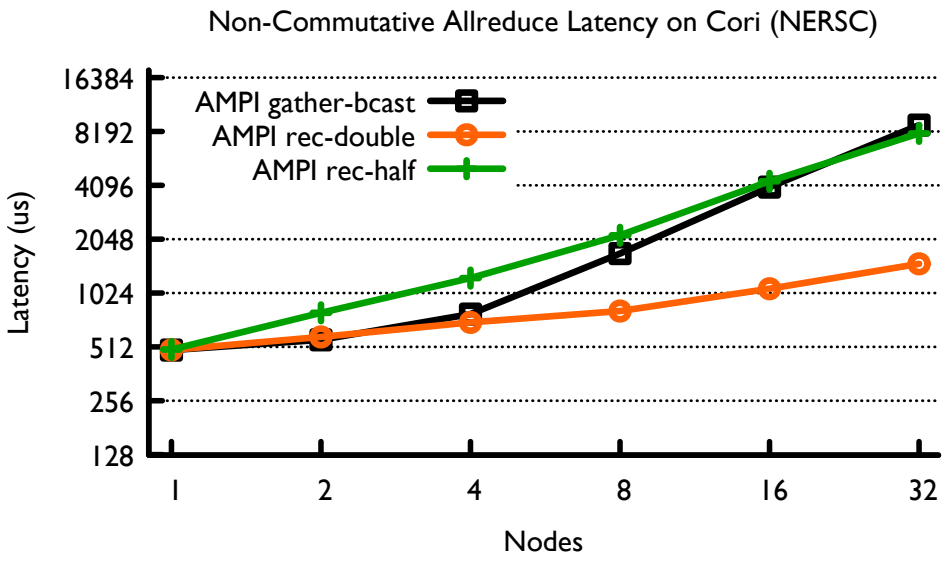



Figure 5.6: **Block Map Scaling**: scaling comparison for non-commutative allreduce of 64KB message size for different algorithms on 1 to 32 nodes of Cori's Haswell partition using the natural block mapping of ranks to cores. Lower is better.

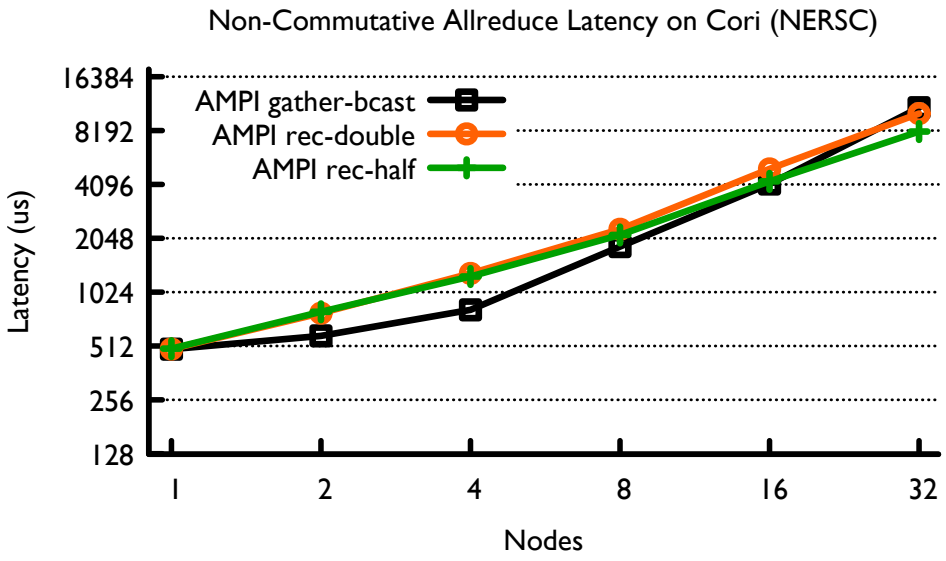



Figure 5.7: **Random Map Scaling**: scaling comparison for non-commutative allreduce of 64KB message size for different algorithms on 1 to 32 nodes of Cori's Haswell partition using a fully randomized mapping of ranks to cores. Lower is better.

and across various message sizes.

Figure 5.8 shows the performance of our different non-commutative allreduce algorithms on a block mapping of ranks to cores. This is the natural ordering of ranks for most ap-

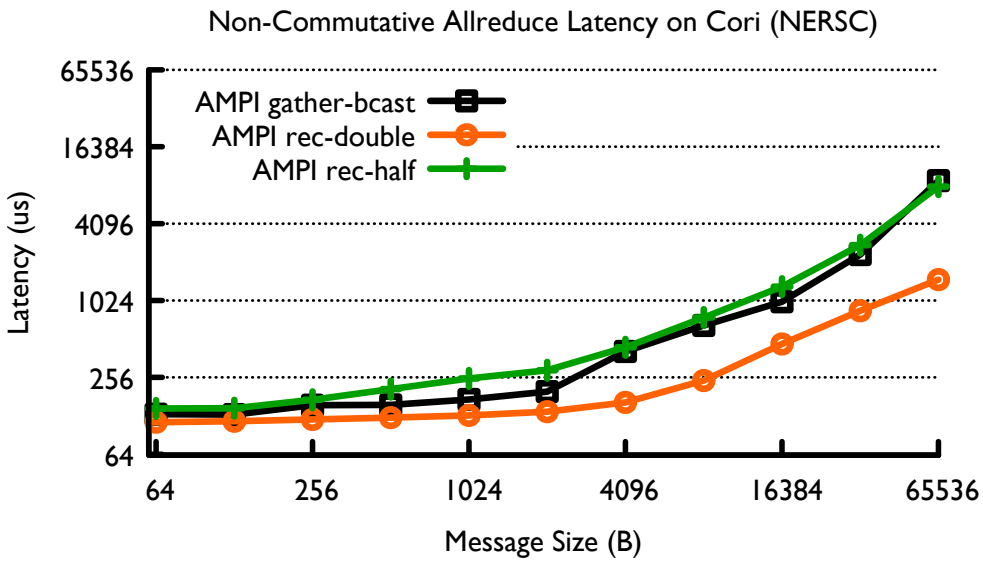



Figure 5.8: **Block Map**: comparison of non-commutative allreduce latency for different algorithms on 32 nodes of Cori's Haswell partition using the natural block mapping of ranks to cores. Lower is better.

plications without any load balancing or migration of ranks. As expected, we see recursive doubling performs best for medium and large message sizes. Recursive halving, meanwhile, performs poorly due to not being able to combine out-of-sequence messages at each step of the internode reduction.

Figure 5.9 shows the performance of our different non-commutative allreduce algorithms on a randomized mapping of ranks to cores. In essence, this simulates the effect of aggressive load balancing, particularly greedy-based rebalancing strategies that do not take into account the the current placement of ranks. In contrast to the block mapping results in Figure 5.8, we see recursive doubling perform similarly to both recursive halving and the gather-broadcast method, with recursive halving slightly outperforming both others for the biggest message sizes of 32KB and larger. This is due to the improved network contention characteristics of recursive-halving for large messages compared to recursive doubling in a more allgather-like operation with the rank-ordering being permuted by the random mapping. Compared to block mapping, recursive doubling performs up to 4.5x worse with a random mapping: a large performance penalty, and one which an application user might not expect to see after turning on load balancing. The gather-broadcast algorithm is unaffected because it does not optimize for rank-ordering at all. But we note that it does slightly outperform the two adaptive algorithms for small messages over the random mapping.

Figure 5.10 shows the performance of our different non-commutative allreduce algorithms

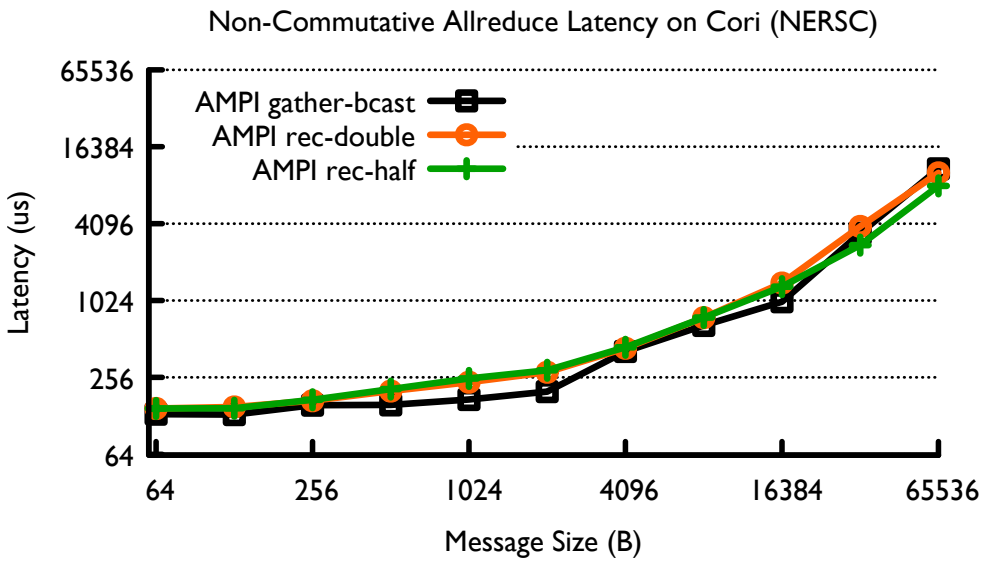



Figure 5.9: **Random Map**: comparison of non-commutative allreduce latency for different algorithms on 32 nodes of Cori's Haswell partition using a random mapping of ranks to cores. Lower is better.

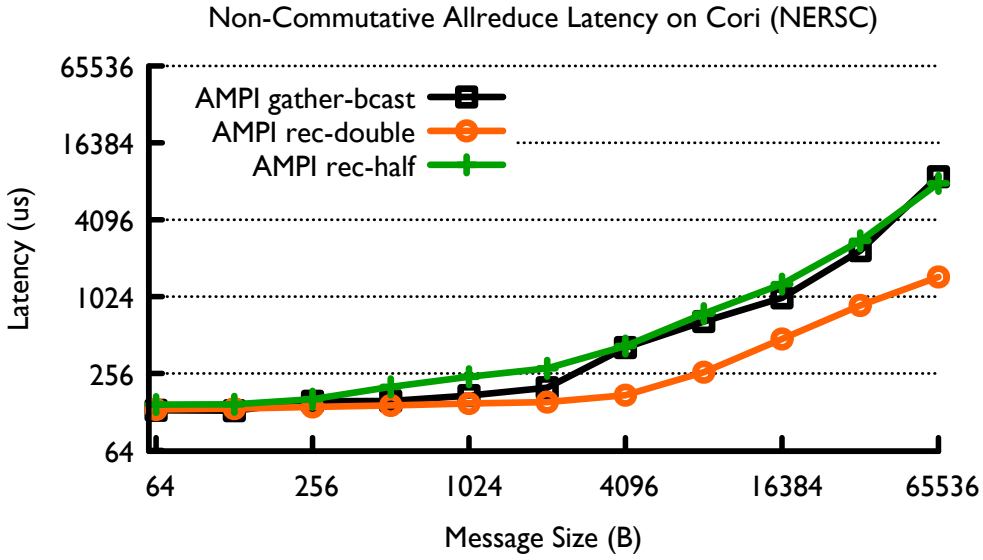



Figure 5.10: **10% Neighbor Map**: comparison of non-commutative allreduce latency for different algorithms on 32 nodes of Cori's Haswell partition using a mapping where 10% of all ranks have migrated to a neighboring node. Lower is better.

on a mapping where 10% of ranks have migrated to a neighboring node. This simulates the effects of a topology-aware refinement-based load balancer which can be highly effective in practice. Here we see recursive doubling perform the best, only paying a slight 3-10%

performance penalty from the block mapping. This is because if migrations are minimal and ranks only move to adjacent nodes then neighboring nodes's contributions can be combined in the first step of the internode algorithm. Consequently, the results are very similar to that of the block mapping.

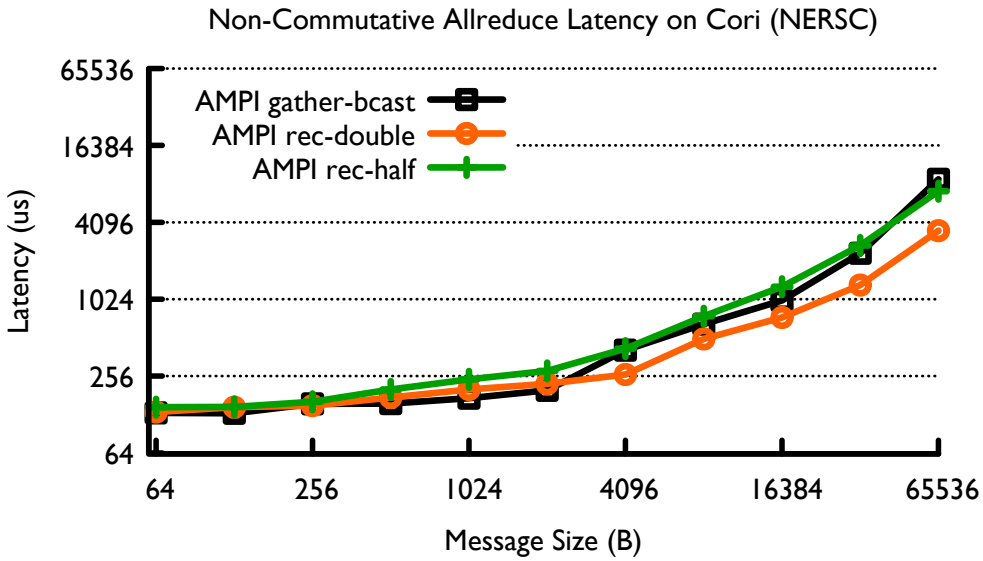



Figure 5.11: **10% Random Map**: comparison of non-commutative allreduce latency for different algorithms on 32 nodes of Cori's Haswell partition using mapping where 10% of ranks have migrated to a random other node. Lower is better.

Figure 5.11 shows the performance of our different non-commutative allreduce algorithms on a mapping where 10% of all ranks have migrated to a random other node. This simulates a topology-unaware refinement-based load balancer which has migrated a small number of ranks to balance load without considering how far they migrate. Recursive doubling pays a larger penalty here, since the rank ordering is disturbed enough that message sizes are increased throughout the algorithm, rather than only the first step as in the neighboring case above.

Overall, our results suggest that our adaptive algorithms with shared memory awareness and opportunistic message combining are able to tolerate permutations of the rank-ordering gracefully, with performance degrading as ranks become more out of their "natural" block mapping. Our adaptive recursive doubling algorithm is able to perform nearly 4.5x faster with the block mapping compared to a fully randomized mapping, and its performance only degrades slightly from the block mapping for scenarios that are realistic for refinement-based dynamic load balancing. We also see that recursive halving can outperform recursive doubling when the message size is large and the mapping is randomized. Moreover, the

gather-sort-broadcast method offers consistent performance across different mappings, but only performs better than the other two algorithms when messages are small or the node count is small.

## 5.4 CONCLUSION

Support for MPI rank virtualization and dynamic migration enables latency tolerance and dynamic load balancing, but can have drawbacks as well. Rank virtualization naturally increases the number of ranks that participate in a collective. Rank migration can also permute the ordering of ranks across the cores or nodes of the system. In this chapter, we identified rank migration as problematic for non-commutative reduction operations and sought to limit the performance degradation as much as possible given a mapping of ranks to cores. We also sought to limit the overheads associated with rank virtualization for commutative reductions and for broadcast operations taking advantage of the shared address space between ranks on the same node.

MPI collectives are well studied. In particular optimizations for different hardware configurations including network architectures and topologies [63] [62] as well as node architectures [59] [61]. Performance trade-offs, in terms of latency and bandwidth, have also been explored extensively [58] [60] [67]. Allreduce is especially well studied given its importance to the performance of many parallel applications.

Non-commutative reductions have received significantly less attention since they are not used as much as the predefined reduction operations, which are all commutative. Thakur et al discuss the algorithmic trade-offs for commutative and non-commutative allreduce in [58]. Traff develops efficient support for non-commutative reduce_scatter operations, where the result vector is split across processes, in [69]. Applications are rightfully encouraged by the MPI standard to prefer the use of commutative operations because they can be optimized further than non-commutative ones [70]. But non-commutative reductions can be an improvement to the user implementing their own similar functionality using either point-to-point communication routines or an (all)gather before applying the reduction operation themselves. Associativity of the reduction operator is key to enabling that performance improvement, as our work has shown.

The effect that process placement or mapping has on different communication patterns has also been considered in various contexts. [64] investigated how process placement affects collective communication performance. They relied on collectives being implemented on top of point-to-point routines. [66] looked at how process mapping affects collective communication over subcommunicators and how to optimize the mapping. Our work approaches a

similar problem from the opposite direction: how to optimize the collective implementation given a less than optimal mapping.

Our work has focused on AMPI's implementation, but our ideas are applicable to other MPI implementations and parallel programming runtimes. Any runtime that supports custom mappings of processes will encounter performance degradation when performing non-commutative reduction operations in non-block mappings, even if those mappings are not subject to dynamic changes as in AMPI. For example, the adaptive message combining based on range encoding and shared memory optimization techniques described here would apply to MPI-only runs with multiple processes per node. In that scenario, intranode reductions could be handled using an efficient interprocess copy mechanism such as XPMEM [71]. Non-commutative reductions over subcommunicators could also benefit from the same optimizations. Threads-based MPI implementations such as MPC [46] are more directly analogous to AMPI, while task based runtimes such as VT [72] or Legion [21] could potentially also make use of non-commutative operations over subsets of indexed tasks.

We implemented versions of recursive doubling and halving that opportunistically combine messages as much as possible at each step of the algorithm. Taking advantage of AMPI's shared address space across ranks in the same node, we optimized within-node reductions as well. Overall, our findings suggest that rank placement matters, that the degree to which ranks are out of their natural ordering affects performance, and that algorithmic changes in the collective implementation or load balancing strategy can effectively minimize the overhead by adapting to the ordering as much as possible. We demonstrated that our rank placement adaptive algorithms could optimize latency of non-commutative allreduce up to 4.5x for a block mapping compared to a random mapping. In practice, fully randomized mapping is unlikely, as most load balancing strategies take into account the current location of ranks and seek to minimize the number of migrations. Accordingly, we showed that if migrations are limited to around 10% of all ranks, and if migrations are limited to either the same node or neighboring nodes then the performance impact on non-commutative allreduce latency can be limited to less than 10% slowdown compared to a block mapping.

For future work, we would like to implement dynamic algorithm selection. This technique has been used to choose MPI collective algorithms for various factors such as message sizes, communicator sizes, communication pattern, and network hardware [73]. Our selection criteria for allreduce would include measures of migration frequency, migration pattern, and use of the non-commutative reduction operations. We would also like to explore use of non-commutative reductions in applications which require dynamic load balancing or rank migration for another reason.

Another avenue for future exploration is to make load balancing strategies more friendly

to non-commutative reductions. Load balancing strategies could be made aware of the use of non-commutative reductions in order to favor within-node migrations or else shifting ranks only at the edges of each node, so as to preserve rank contiguity within each node as much as possible. We would also like to explore different algorithms for other collective operations that share similarities to non-commutative reduction operations, such as (all)gather(v). These only require ordering of the contributions in rank order, but there is no opportunity to shrink message sizes by combining contributions along the way.

# CHAPTER 6: INTEROPERABILITY WITH OTHER PARALLEL PROGRAMMING MODELS

## 6.1 INTRODUCTION

So far we have discussed our runtime as a virtualized extension of the MPI-everywhere approach or as an alternative to hybrid MPI+X parallelism where X is a shared memory parallel programming model. With AMPI's virtualization of ranks as user-level threads, the multithreading that might traditionally be done using OpenMP or pthreads is hoisted into the runtime and each rank maintains its own private communication endpoint. This is in contrast to the typical MPI thread levels such as *MPI_THREAD_FUNNELED*, *_SERIALIZED*, or *_MULTIPLE*, where multiple threads share a single communication context. While this can be an efficient alternative for many applications, others which do not scale well in terms of the number of ranks may not see benefits. This includes iterative solvers that may take more iterations to converge when run over more ranks, or applications that rely on personalized all-to-all collective communication routines. For these applications, it may be more efficient to run with fewer MPI ranks and to interoperate with a shared memory parallel programming model for efficient parallel computation within each node. Other applications may also use a shared memory parallel programming model in order to dynamically balance computational load within each node, i.e. using OpenMP's dynamic loop scheduling capability.

Other uses for interoperability include compatibility with existing libraries and the use of accelerators such as GPUs. In such cases, AMPI should ideally interoperate seamlessly and efficiently with these other programming models. In practice, it is challenging to interoperate efficiently with other programming models, even for widely used programming models such as MPI+OpenMP or MPI+CUDA. For instance, MPI+OpenMP libraries that are called from MPI-only code can result in the operating system scheduling OpenMP and MPI code on the same core. AMPI's virtualized execution model can complicate these even further, if attention is not paid to competing schedulers. In this chapter, we study interoperation of AMPI with other parallel programming models commonly used in high performance computing, and implement solutions to overcome the challenges identified.

## 6.2 OPENMP

The performance challenges around MPI+OpenMP hybrid parallelism have been well studied. These range from thread locality and affinity to serialization around communication

and load balance. There are several reasons to adopt OpenMP into an MPI application. The memory overheads associated with inter-rank message passing can be avoided when using shared memory directly. The fewer number of ranks in an MPI+OpenMP job can translate to better scaling in terms of collectives and parallel I/O operations. OpenMP dynamic scheduling can help mitigate load imbalance coming from either software or hardware sources of variability. For some applications, there are algorithmic advantages to running with fewer ranks. Naturally, one might want to combine AMPI with OpenMP to attain any of these benefits as well.

We consider AMPI+OpenMP interoperation in two different configurations: PE-level virtualization and node-level virtualization. PE-level virtualization is what we call AMPI's typical run configuration with multiple virtual ranks being co-scheduled on each PE. Node-level virtualization is closer to the typical MPI+OpenMP scheme but with rank virtualization on the master thread. For example, an AMPI+OpenMP application running with PE-level virtualization might have 1 process per node, with 16 PEs in that process, each PE having 8 virtual ranks for a total of 128 ranks in the process. An AMPI+OpenMP code run with node-level virtualization, on the other hand, might have 1 process per node, with 8 virtual ranks being co-scheduled on 1 PE. In the case of PE-level virtualization, typical OpenMP implementations will not interoperate nicely with AMPI, since AMPI and OpenMP will both try to schedule work on each PE. In the node-level virtualization scheme, traditional OpenMP runtimes can compose nicely with AMPI from a scheduling perspective, since AMPI will not compete to schedule work on the other PEs in the node. To address the scheduling issues associated with PE-level virtualization, we motivate and develop an integrated OpenMP runtime with AMPI and compare it to the node-level virtualization approach.

### 6.2.1 OpenMP Runtime Integration

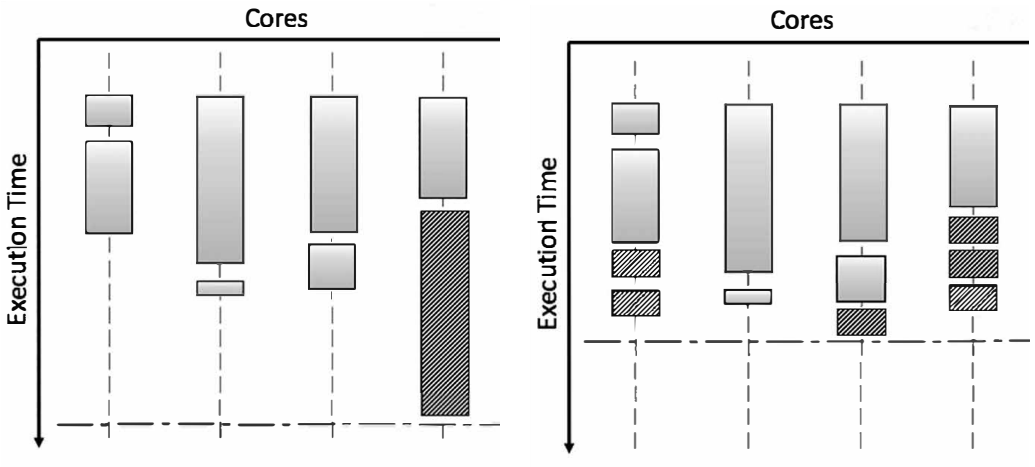
Many parallel applications no longer operate in a regime where work and data can be neatly divided into uniform chunks distributed to each processor. This trend encompasses unstructured computations, data-dependent iterative methods, variable resolution, multi-physics simulations, multi-phase execution, and many other developments that trade reduced total work or increased accuracy for more complicated and less predictable execution. Even applications that do offer simple structured decompositions can be made imbalanced by hardware heterogeneity. Load balancing in various forms can be applied to aid these applications, but it too must be scalable, which often means coarsening the problem to the node level to avoid considering an excessive number of cores. Discrete units of work assignment, heuristic algorithms, and unpredictable processor performance also prevent perfect uniformity. Sup-

plementary within-node balancing can help make up for these short-falls, as illustrated in Figure 6.1. As shown in the figure, the initial distribution of work items, represented by blocks, results in load imbalance. Using supplementary within-node load balancing helps redistribute only the excess work, as represented by the shaded blocks from the overloaded cores.

Even with very balanced work assignment across nodes and individual cores, execution may not proceed at a perfectly uniform pace. Network contention can delay some messages more than others. System noise from OS processes can also non-uniformly interfere with execution [74], with hard to predict knock-on effects [75]. Dynamic work redistribution can greatly help in mitigating these effects [76].

All of these pressures lead to a conclusion that multiple cores within each node must share data and work to sustain continued scalability in problem size and performance. At the same time, any sharing mechanism ideally should not compromise data locality or introduce excessive new bottlenecks or overheads. To address these desires, we introduce a design that combines the AMPI distributed programming model with a modified OpenMP runtime system. AMPI intermittently performs coarse load balancing in terms of objects that encapsulate associated work and data together, and assigns them to particular cores with good balance among nodes. These objects then adaptively share work with other cores in the same process, exposing fine-grained tasks only to the extent that otherwise idle cores are available to help execute them. Thus, our design ensures locality as well as low and proportionate scheduling overhead.

The main challenge of our integrated runtime approach is to balance load across PEs while




(a) Without intra-node LB. (b) With intra-node LB.

Figure 6.1: The potential benefits of intra-node work sharing on reducing load imbalance.

managing locality. A pure task model with randomized work stealing, or a pure dynamic schedule in OpenMP, sacrifices locality significantly to an extent that often eliminates the benefits of dynamic load balancing [77, 78]. Dynamic load balancing strategies are used to balance the load and redistribute the work at runtime. These load balancing strategies can incur significant overhead due to the cost of computing a new assignment and the consequent data movement. If done less frequently, the overhead is reduced and locality is maintained, but dynamically emerging load imbalance may last longer before being corrected. With increasing number of cores within a node, intra-node load balancing will become an effective way to reduce load imbalance.

The approach we propose is to utilize a relatively infrequent periodic assignment of work to cores based on load measurement, combined with user assisted creation of potential tasks from the work assigned to each core that the runtime can choose to make available to other cores. The idea is to utilize the idle cycles on other cores on a node to execute tasks belonging to the overloaded cores. We also need to make sure we do not incur task creation overhead when tasks are not needed. Figure 6.1 shows a schematic diagram of such a scenario where most of the computations are executed on the core they are assigned to, but the load imbalance towards the end triggers the dynamic creation of fine-grained tasks which are distributed across different cores.

Initially, we implemented this integrated runtime using GNU OpenMP [79], but have since implemented it using the LLVM OpenMP runtime. The key idea is that we still allow AMPI virtual ranks to be virtualized on each PE, but when a rank enters an OpenMP parallel region, instead of creating threads spread across the PEs on the node to execute that OpenMP parallel work, we instead enqueue that work locally on a work stealing task queue. Then when other PEs become idle, they try to steal randomly from other PE's work stealing task queues.

The scheduler on each PE polls the local task queue and the message queue for messages. We chose not to have a centralized task queue at the node level because then we lose locality information and there could be potential contention for the centralized queue. We have a separate task queue on each PE, which is a single producer multiple consumer queue for the fine-grained tasks. Whenever a PE becomes idle, it randomly chooses a PE and steals tasks from that PE's task queue. This is similar to Cilk's workstealing [80], except that our scheduler also polls other queues, including a PE-specific message queue for messages to chares assigned to that PE by the periodic load balancer.

The task queue is implemented using the Chase-Lev [81] non-blocking algorithm. The task queue is a double-ended queue. A *push(t)* call enqueues a task at the tail of the queue. A *pop()* call dequeues a task from the tail of the queue. A *steal()* call dequeues from the

head of the queue. The queue is a cyclic array of task pointers with non-wrapping head and tail indices, called H and T. A worker does a *push(t)* by adding the task at the tail of the queue and increments T, the tail pointer. A worker does a *pop()* by decrementing T. If it detects that there could be a conflict, then it uses compare and swap (CAS) to handle the conflict. A thief reads H and T and uses CAS to atomically increment H and obtains the task.

The task descriptor contains details about the task such as the object pointer, function pointer, parameters and an atomic variable. The message enqueued into the task queue contains range parameters and a pointer to the common task descriptor. To minimize the overheads of creating messages and task descriptors, we keep a pool of task messages and descriptors which are reused.

To minimize the overhead, we adopted two heuristics. Each node maintains an atomic counter to keep track of idle PEs within a node and each PE keeps a history vector of how many OpenMP tasks have been stolen by other idle PEs. Using these two heuristics, we can create OpenMP tasks only when there are idle PEs and the fine-grained parallelism is beneficial.

The initial implementation using the GNU OpenMP runtime still had creation overhead to some degree and had only limited support for OpenMP directives, because it was implemented using stackless Charm++ messages. First, it only supports barriers at the end of each OpenMP region. OpenMP has implicit and explicit barriers within a region, and can use multiple barriers within each region. For example, "omp for" has an implicit barrier in the end of each "omp for" pragma and "omp single" may have an implicit barrier if the variable updated within "omp single" is accessed outside the pragma. In addition, many synchronization pragmas such as "omp barrier" are used for correctness and verification. These barriers could not be implemented because of the use of stackless messages.

To implement barriers, the OpenMP tasks should be able to be suspended and resumed, and all the data for each OpenMP task should be maintained when they are resumed on other PEs. In addition, the stackless messages incur unnecessary overhead for each OpenMP region. Most OpenMP runtimes maintain a pool of threads which are suspended and can be resumed for the upcoming OpenMP regions, such that an OpenMP thread is initialized only when it is created in the beginning of the first OpenMP region, and is suspended and resumed until the runtime is exiting. Our initial implementation did the initialization for each OpenMP region because threads could not be suspended and resumed.

We adopted user-level threads to resume and suspend OpenMP tasks on top of the Charm++ runtime. Now, each OpenMP region creates user-level threads which can be scheduled by the Charm++ runtime scheduler. These user-level threads are pushed to the

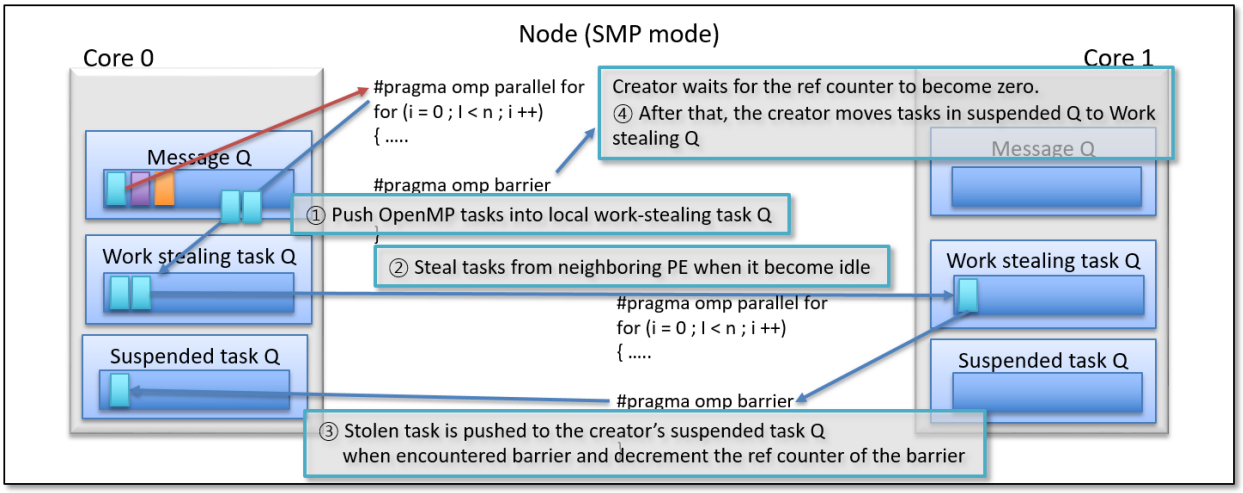



Figure 6.2: Implementation of OpenMP for Charm++ using user-level threads

same work-stealing queue as in the first queue and minimize the overhead of fine-grained parallelism using the same heuristics in the prior implementation. Each user level thread has its own stack and is migratable across different kernel-level threads.

However, even with the user-level threads, there are still several issues to implement suspend and resume of OpenMP threads. The first issue is how to schedule suspended OpenMP tasks which are stolen by thieves. Thieves cannot continue to work on this because they can be idle temporarily while waiting for messages from other PEs. So, these suspended tasks should be pushed to the creator's queue. The second issue is that the suspended tasks cannot be pushed to the creator's work-stealing queue by thieves because the work-stealing queue supports one producer and multiple consumers to minimize the usage of atomic operations. To resolve this issue, we implemented a separate queue for suspended tasks on each PE which supports multiple producers and consumers.

Figure 6.2 shows how the current implementation of OpenMP interoperates with Charm++ on a node with 2 PEs. First, the integrated OpenMP creates OpenMP tasks on OpenMP region which are user-level threads migratable across PEs on Charm++ runtime. Each OpenMP region keeps atomic counter for each barrier within the region. Created OpenMP tasks decrement the counter when encountered barriers within in each OpenMP region and they are pushed to the creator's suspended task queue if they are executed in PEs other than the creator. The creator of the OpenMP tasks waits for the counter to become zero and move suspended tasks from suspended task queue to work stealing queue after that. In this way, the integrated OpenMP resolve load imbalance across PEs within a node and implements synchronization and worksharing directives of OpenMP on top of Charm++ runtime system.

We initially modified the GNU OpenMP runtime for our work but we migrated to LLVM OpenMP runtime for better compatibility which works with common compilers such as icc, gcc, and clang. In addition to better compatibility, the LLVM OpenMP runtime has fine-grained optimizations such as frequent usage of padding for shared variables and assembly instructions for synchronization routines.

The adoption of user level threads brings several advantages over the initial implementation. First, multiple OpenMP parallel regions can be coalesced into one bigger OpenMP region. In the start of an OpenMP region, the runtime incurs an overhead and loses locality if there are short OpenMP regions successively because the same data can be accessed by different PEs. Implementation of barriers resolves this issue by coalescing short OpenMP regions into a bigger region. In addition, we can avoid some of the initialization of each OpenMP task mentioned above because OpenMP tasks can suspend and resume within a while loop. Each PE keeps a pool of user-level threads for OpenMP and resumes those threads only with initialization of function pointers to each OpenMP region.

To demonstrate the performance of our integrated OpenMP runtime, we ran the Kripke proxy application on AMPI. Kripke [82] is an LLNL proxy application for parallel deterministic transport codes. It is written using MPI and, optionally, OpenMP for parallelism. Kripke implements the key computation and communication aspects of a production transport simulation application. Such codes are used to deterministically solve for the flux of neutral particles within a volume of interest. Kripke implements parallel sweeps through a 3D domain. The domain is decomposed into spatial zones, and subdomains are distributed to MPI ranks.

Parallel sweeps are vital communication kernels for the performance of deterministic transport codes. A sweep is a sequential traversal through a domain. Because of the sequential dependencies through the domain, and because the domain is decomposed spatially, scaling sweeps efficiently is challenging. Consequently, Kripke pipelines successive sweeps over the different energy groups and directions in the problem to attain higher efficiency. In addition to the sweep, a reduction is performed every iteration to test the global particle count for convergence.

Our OpenMP runtime can be used with AMPI+OpenMP programs the same way it is with Charm++ applications. This allows users to run an AMPI code on a node with $N$ PEs using $N$ or more AMPI ranks per node with each rank using up to $N$ OpenMP threads, without actually oversubscribing the physical resources on the system.

All of the tests below were performed on Theta, using 64 cores per node. We use the default input parameters for Kripke version 1.1. No changes are necessary to the source code of Kripke to run it on AMPI and our implementation of OpenMP. We show weak

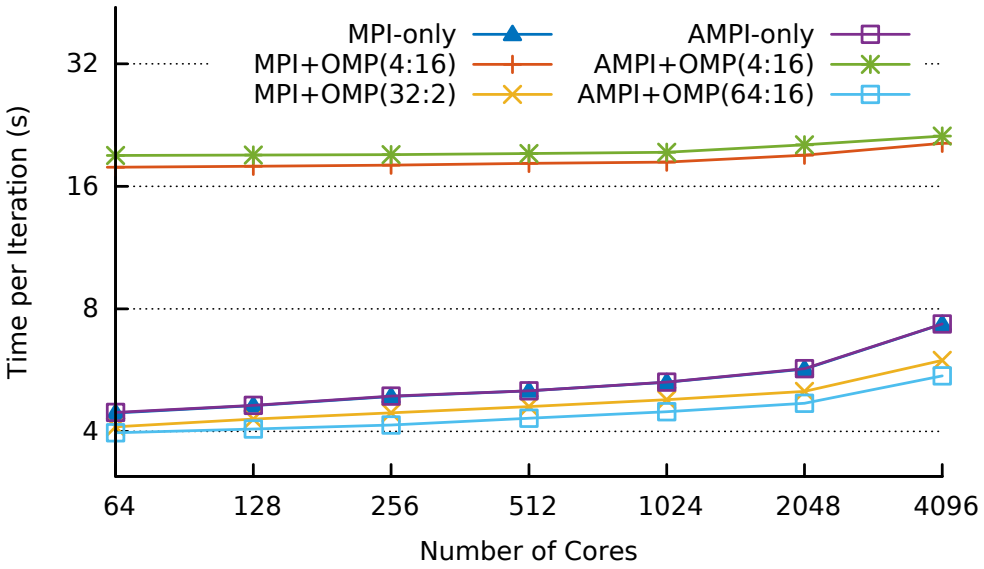



Figure 6.3: Weak scaling Kripke with 4096 spatial zones per core on Theta, the time per iteration is shown for MPI and AMPI with and without OpenMP. Numbers in parentheses indicate how many ranks were used per node and the number of OpenMP threads per rank. Lower is better.

scaling in the number of zones, with the number of groups and directions held constant.

Figure 6.3 shows the time per iteration of Kripke using MPI, MPI+OpenMP, AMPI, and AMPI+OpenMP with two different configurations. The parenthetical in MPI+OMP (4:16) and others identifies how many ranks were launched per node, and how many threads may execute any OpenMP parallel-for loop at a time. Thus, MPI+OMP (4:16) signifies the use of 4 ranks per node with 16 OpenMP threads per rank, and MPI+OMP (32:2) means 32 ranks were launched per node with 2 OpenMP threads per rank. In addition to MPI-only, AMPI-only, and both with four processes and 16-way threading per node, we show the best performing combination of rank and thread counts for each.

Kripke's parallel sweeps benefit from the finer-grained pipeline parallelism that decomposing into more MPI ranks offers. On the other hand, the computational kernels benefit from OpenMP threading. Since sweep dependencies translate to idle times within a node while each wavefront passes through the domain, within-node parallelism can be also be used to balance the load across the idle threads at a given time. Persistence-based load balancing does not help Kripke's performance, since across iterations the load is balanced. The combination of 64 ranks and up to 16-way threading per rank performs 11% better than the next best combination. Essentially, the AMPI+OpenMP (64:16) case gives the runtime the freedom to schedule transient OpenMP work across all available cores on a node while

still decomposing the sweep pipeline into small pipeline stages. These results show the benefits of our unified runtime approach for applications that have transient load imbalances within iterations, which are not the target of AMPI's usual persistence-based dynamic load balancing.

In conclusion, we have demonstrated the need for scheduler integration when running with PE-level virtualization and shown the effectiveness of our integrated OpenMP runtime scheduling. Running in this mode, users can rely on their MPI decomposition to ensure good locality of reference on each thread, while AMPI optimizes the communication within the shared address space and OpenMP is essentially used to expose parallelizable work to within-node work stealing. Taking into account the number of idle PEs in the node and the history of successful steals by other PEs, we minimized the runtime overhead of stealable task creation. An additional benefit is the ability to call into AMPI+OpenMP library code from an AMPI-only application without incurring scheduler interference. For future work, we would like to pursue making the decision of PE- versus node-level overdecomposition configuration–how many PEs should schedule AMPI ranks compared to how many should only be used for OpenMP work?–dynamically rather than statically.

## 6.3 AMPI ON GPUS

As GPUs are becoming more and more common in high performance computing as accelerators, programming models are adapting. OpenMP and OpenCL are both adding features targeted at efficient GPU execution. The MPI Forum is exploring different messaging paradigms that can map efficiently to GPUs. Two issues that arise when running AMPI with CUDA or another programming model for GPUs are 1) supporting efficient communication between devices, which have their own memory separate from the host memory, and 2) efficiently scheduling kernel invocations without blocking the calling thread. In the former case, we wish to support in-place communication to and from GPU memory, and in the latter we wish to avoid blocking the scheduler on a PE when invoking a GPU kernel from the host. We pursued efficient communication in AMPI between GPUs by extending our zero copy communication interfaces, as explained in Chapter 3, and also extended Charm++'s Hybrid API in order to make it usable from AMPI for asynchronous kernel scheduling, launch, and completion detection.

```
// Sender object's method
void Sender::foo() {
        // Send a message to the receiver object
        // to execute the 'bar' entry method
        receiver.bar(my_val1, my_val2);
}

// Receiver object's entry method,
// executed once the sender's message
// is picked up by the scheduler
void Receiver::bar(int val1, double val2) {
        // val1 and val2 are available
        ...
}
```

Figure 6.4: Message-driven execution in Charm++.

### 6.3.1 GPU-aware Communication

Similar to the work in Chapter 3, where we developed new in-place communication support in Charm++, AMPI's underlying runtime system, here we have extended those APIs for communication of buffers in GPU device memory. We do so building on the UCX communication library and its support for GPU communication via its tagged API. UCX is supported as a machine layer in Charm++, positioned at the lowest level of the software stack directly interfacing the interconnect. As AMPI is built on top of the Charm++ runtime system, all host-side communication travels through the Charm++ core and Converse layers where layer-specific headers are added or extracted, with actual communication primitives executed by the machine layer.

The main idea of enabling GPU-aware communication in the Charm++ family of parallel programming models is to retain this route to send metadata and host-side data, while separately supplying GPU data to the UCX machine layer. The metadata is necessitated by the message-driven execution model in Charm++, as shown in Figure 6.4. The sender object provides the data it wants to send to the entry method invocation, but the receiver does not post an explicit receive function. Instead, the sender's message arrives in the message queue of the PE that currently owns the receiver object. When the message is picked up by the scheduler, the receiver object and target entry method are resolved using the metadata contained in the message. Any host-resident data destined for the receiving chare is unpacked from the message and delivered to the receiver's entry method.

With our GPU-aware communication scheme, the sender object's GPU buffers are not included as part of the message. Only metadata containing information about the GPU data transfer initiated by the sender and sender's data on host memory are contained in the

message. Source GPU buffers are directly provided to the UCX machine layer to be sent, and a receive for the incoming GPU data is posted once the host-side message arrives on the receiver. A noticeable limitation of this approach is the delay in posting the receive caused by the need to wait for the host-side message containing the metadata. We are currently working on an improved mechanism where explicit receives can be posted in advance.

Originally contributed by Mellanox, the UCX machine layer in Charm++ is designed to handle low-level communication using the UCP tagged API, providing a portable implementation over all the networking hardware supported by UCX. To support GPU-aware communication, we extend the UCX machine layer to provide an interface for sending and receiving GPU data with the UCP tagged API. We adopt a tag generation scheme specific to GPU-GPU transfers to separate this path from the existing host-side messaging.

The first four bits (MSG_BITS) of the 64-bit tag are used to differentiate the message type, where the new UCX_MSG_TAG_DEVICE type is added for inter-GPU communication. The remainder of the tag is split into the source PE index (PE_BITS, 32 by default) and the value of a counter maintained by the source PE (CNT_BITS, 28 by default). This division can be modified by the user to allocate more bits to one side or the other to accommodate different scaling configurations.

The core functionalities of GPU-aware communication in the UCX machine layer are exposed as the following functions:

```
void LrtsSendDevice(int dest_pe, const void*& ptr,
                    size_t size, uint64_t& tag);
void LrtsRecvDevice(DeviceRdmaOp* op,
                    DeviceRecvType type);
```

LrtsSendDevice provides the functionality to send GPU data using the information provided by the calling layer including the destination PE, address of the source GPU buffer, size of the data, and a reference to the 64-bit tag to be set. The tag is generated within this function by incrementing the tag counter of the source PE, and included as metadata by the caller to be sent along with any host-side data. Once the destination UCP endpoint is determined, the source GPU buffer is sent separately with ucp_tag_send_nb using the generated tag.

Once the metadata arrives on the destination PE, the corresponding receive for the incoming GPU data is posted with LrtsRecvDevice. The DeviceRdmaOp struct passed by the calling layer contains metadata necessary to post the receive with ucp_tag_recv_nb, such as the address of the destination GPU buffer, size of the data, and the tag set by the sender. DeviceRecvType denotes which parallel programming model has posted the receive, so that the appropriate handler function can be invoked once the GPU data has been received. The

```
// Charm++ Interface (CI) file
// Exposes chare objects and entry methods
chare MyChare {
        entry MyChare();
        entry void recv(nocopydevice char data[size],
                        size_t size);
};

// C++ source file
// (1) Sender chare
void MyChare::send() {
        peer.recv(CkDeviceBuffer(send_gpu_data), size);
}

// (2) Receiver's post entry method
void MyChare::recv(char*& data, size_t& size) {
        // Set the destination GPU buffer
        // Receive size is optional
        data = recv_gpu_data;
}

// (3) Receiver's regular entry method
void MyChare::recv(char* data, size_t size) {
        // Receive complete, GPU data is available
        ...
}
```

Figure 6.5: GPU-aware communication interface in Charm++.

following sections describe in detail how the different parallel programming models build on the UCX machine layer to perform GPU-aware communication.

Communication in Charm++ occurs between chare objects that may be scheduled on different PEs. It should be noted that multiple parameters can be passed to a single entry method invocation, as in Figure 6.4. We provide an additional attribute in the Charm++ Interface (CI) file, nocopydevice, to annotate parameters on GPU memory. Figure 6.5 illustrates this extension as well as the usage of a CkDeviceBuffer object, which wraps the address of a source GPU buffer and is used by the runtime system to store metadata regarding the GPU-GPU transfer. The structure of CkDeviceBuffer is presented in Figure 6.6.

### Send

An entry method invocation such as peer.recv() in Figure 6.5 executes a generated code block that prepares a message containing data on host memory and sends it to the receiver object. We modify the code generation to send GPU buffers in tandem, using the

```
// Converse layer metadata
struct CmiDeviceBuffer {
        const void* ptr; // Source GPU buffer address
        size_t size;
        uint64_t tag; // Set in the UCX machine layer
        ...
};

// Charm++ core layer metadata
struct CkDeviceBuffer : CmiDeviceBuffer {
        CkCallback cb; // Support Charm++ callbacks
        ...
};
```

Figure 6.6: Metadata object used for GPU communication in Charm++.

CkDeviceBuffer objects provided by the user (one per buffer). These objects hold information necessary for the UCX machine layer to send the GPU buffers with LrtsSendDevice. The tags set by the machine layer are stored in the CkDeviceBuffer objects, which are packed with host-side data as well as other metadata needed by the Converse and Charm++ core layers. This packed message is sent separately, also using the UCX machine layer. Figure 6.7 illustrates this process.

### Receive

To receive the incoming GPU data directly into the user's destination buffers and avoid extra copies, we provide a mechanism for the user to specify the addresses of the destination GPU buffers by extending the Zero Copy API of Charm++ that we discussed in Chapter 3. The user can provide this information to the runtime system in the *post entry method* of the receiver object, which is executed by the runtime system before the actual target entry method, i.e., *regular entry method*. As can be seen in Figure 6.5, the post entry method has a similar function signature as the regular entry method, with parameters passed as references so that they can be set by the user.

When the message containing host-side data and metadata (including CkDeviceBuffer objects) arrives, the post entry method of the receiver chare is first executed. Using information about destination GPU buffers provided by the user in the post entry method and source GPU buffers in the `CkDeviceBuffer` objects, the receiver instructs the UCX machine layer to post receives for the incoming GPU data with LrtsRecvDevice. Once all the GPU buffers have arrived, the regular entry method is invoked, completing the communication.

Communication between AMPI ranks occurs through an exchange of AMPI messages

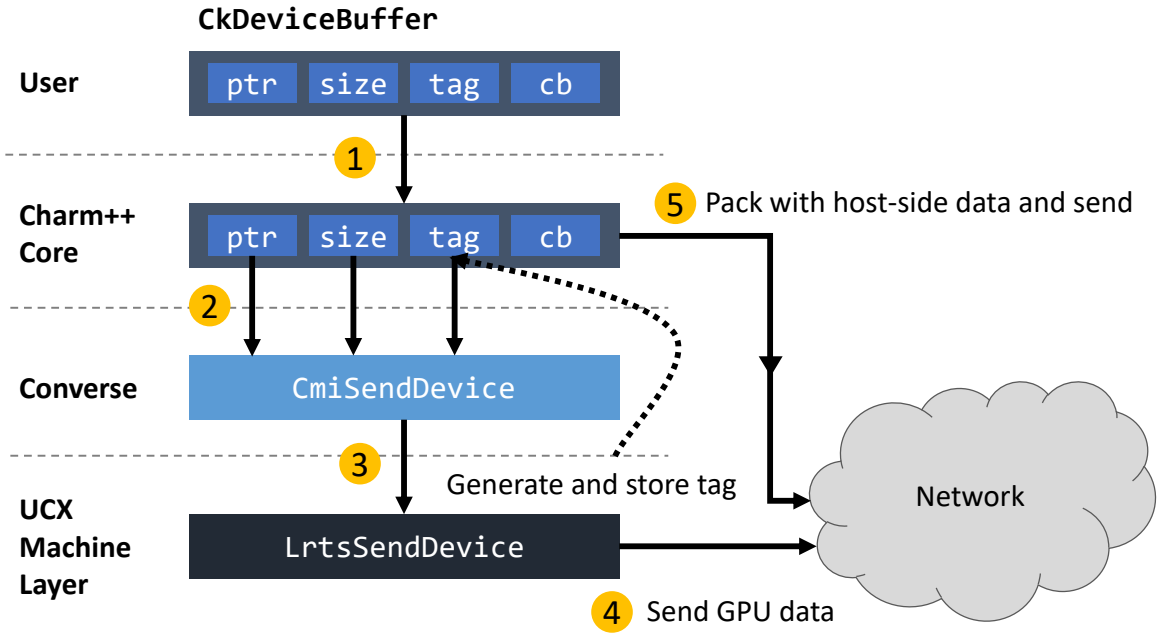



Figure 6.7: Sender-side logic of GPU-aware communication in Charm++.

between the respective chare objects. An AMPI message adds AMPI-specific data such as the MPI communicator and user-provided tag to a Charm++ message, and we modify how it is created to support GPU-aware communication with the CkDeviceBuffer metadata object. This change is transparent to the user, and GPU buffers can be directly provided to AMPI communication primitives such as MPI_Send and MPI_Recv like any CUDA-aware MPI implementation.

### Send

The user application can send GPU data by invoking a MPI send call with parameters including the address of the source buffer, number of elements and their datatype, destination rank, tag, and MPI communicator. The chare object that manages the destination rank is first determined, and the source buffer's address is checked to see if it is located on GPU memory. A software cache containing addresses known to be on the GPU is maintained on each PE to optimize this process. Figure 6.8 illustrates the mechanism that is executed when the source buffer is found to be on the GPU, where a CkDeviceBuffer object is first created in the AMPI runtime to store the information provided by the user. A Charm++ callback object is also created and stored as metadata, which is used by AMPI to notify the sender rank when the communication is complete. The source GPU buffer is sent in an

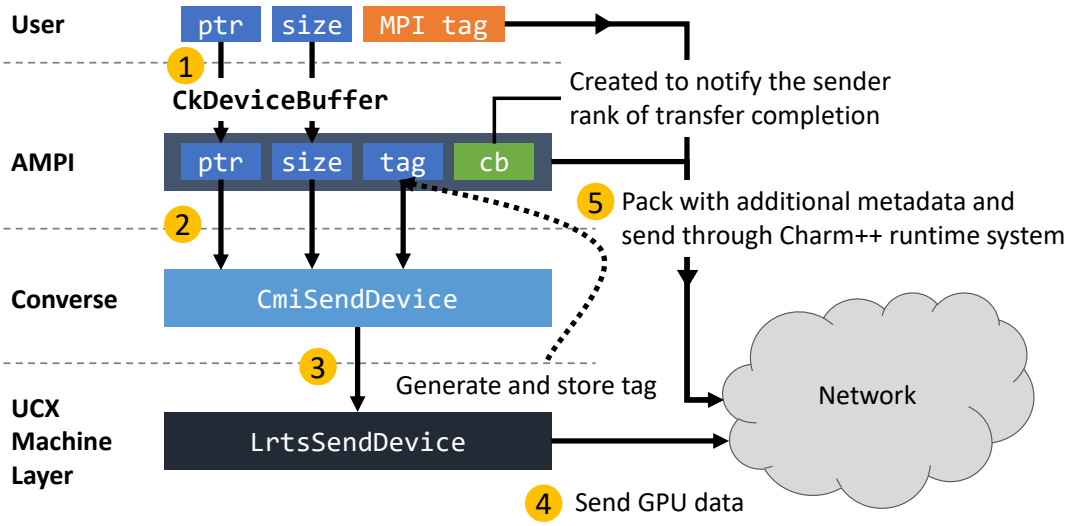



Figure 6.8: Sender-side logic of GPU-aware communication in AMPI.

identical manner as Charm++ through the UCX machine layer with LrtsSendDevice. The tag that is needed by the receiver rank to post a receive for the incoming GPU data is also generated and stored inside the CkDeviceBuffer object. Note that this tag is separate from the MPI tag provided by the user, which is used to match the host-side send and receive.

### Receive

Because there are explicit receive calls in the MPI model in contrast to Charm++, there are two possible scenarios regarding the host-side message that contains metadata: the message arrives before the receive is posted, and vice versa. If the message arrives first, it is stored in an unexpected message queue, which is searched for a match when the receive is posted later. If the receive is posted first, it is stored in a request queue to be matched when the message arrives. The receive for the incoming GPU data is posted after this match of the host-side message, with LrtsRecvDevice in the UCX machine layer. Another Charm++ callback is created for the purpose of notifying the destination rank, which is invoked by the machine layer when the GPU data arrives.

We used the Summit supercomputer at Oak Ridge National Laboratory to measure performance of our communication optimizations. Each Summit IBM AC922 node contains two IBM Power9 CPUs and six NVIDIA Tesla V100 GPUs. Each CPU is connected to three GPUs, which are interconnected via NVLink with a theoretical peak bandwidth of 50 GB/s.

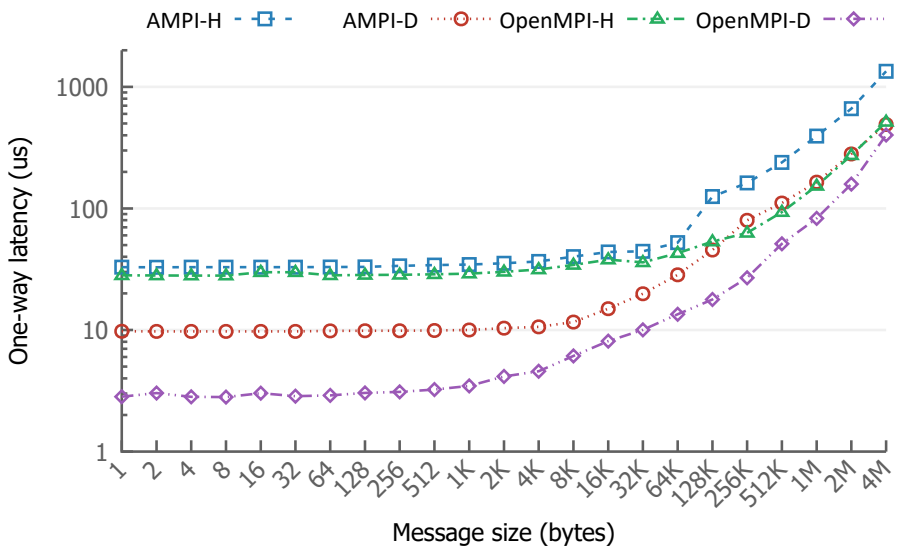



Figure 6.9: OSU MPI Latency benchmark results for internode device messages. We compare host-staging (H) and device (D) communication protocols. Lower is better.

For a GPU to communicate with another GPU connected to the other CPU, data needs to travel through the X-Bus that connects the CPUs with a bandwidth of 64 GB/s. The network interconnect is based on Mellanox Enhanced Data Rate (EDR) Infiniband, providing up to 12.5 GB/s of bandwidth. For reference, the performance of OpenMPI is provided along with our results.

To evaluate the performance of point-to-point communication primitives involving GPU memory, we use the OSU micro-benchmark suite. Performance results are presented with both axes in log-scale, comparing the GPU-aware version of the benchmark (suffixed with D) against the host-staging version (suffixed with H).

Figure 6.9 shows the OSU latency benchmark results. The observed improvement in latency increases with message size when using GPU-aware communication, as the host-staging mechanism suffers significant slowdowns caused by host memory copies in the runtime system. We also measure point-to-point communication bandwidth, shown in Figure 6.10. Here again we see dramatic improvements over host-staging. AMPI achieves up to 10 GB/s bandwidth out of a theoretical limit of 12.5 GB/s. Although the performance of AMPI improves substantially with GPU-aware communication, it does not quite match the performance of CUDA-aware OpenMPI. Further investigation is needed to isolate the precise causes of this overhead, though we have verified that the UCX transfer itself is competitive with OpenMPI's performance and that AMPI is not adding any significant overhead to Charm++. This means the overhead is in Charm++, which could have multiple remaining inefficiencies: message packing and unpacking, additional host-side message that contains metadata,

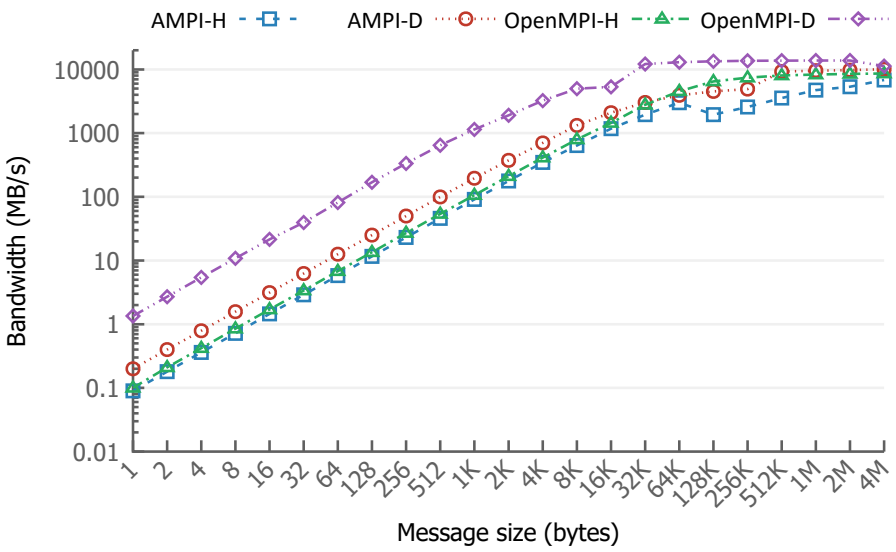



Figure 6.10: OSU MPI Bandwidth benchmark results for internode device messages. We compare host-staging (H) and device (D) communication protocols. Higher is better.

Charm++ callback invocations, and the fact that the receiver rank cannot post a receive until the metadata message is received. There are also a couple of heap memory allocations that are used to retain metadata for the UCX machine layer. We plan to further analyze and optimize the code to get AMPI's performance as close to OpenMPI and the hardware limits as possible.

### 6.3.2 Asynchronous GPU Scheduling

The second issue we have identified with AMPI usage on GPU accelerated systems is that users may perform blocking kernel invocations from the host over the device. This will block not only the calling rank, but the scheduler thread as well by default, preventing AMPI from making progress on other ranks and overlapping the kernel's execution with useful work on the host. Instead, we would like to make the interface for invocation and completion detection of kernels non-blocking, so that the runtime can proceed with scheduling on the host while kernels execute on the device. Our scheduler can then poll for completion of a kernel or multiple kernels and notify the application.

In order to support asynchronous completion of CUDA kernels (we limit our implementation to NVIDIA GPUs), we provide applications the ability to execute kernels in a non-blocking manner, with the runtime managing its asynchronous completion. In the Charm++ interface, users specify completion callbacks to be invoked when the kernel finishes execution, while in AMPI they can do the same with either registered functions or an MPI_Request

```
// In a .cu file

__global__ void kernel() {
  // Do something ...
}

void launchKernel(cudaStream_t stream) {
  dim3 grid_dim(1, 1);
  dim3 block_dim(16, 16);

  kernel<<<grid_dim, block_dim, 0, stream>>>();
}


// In a separate C/C++ file:
  ...
  MPI_Request req;
  cudaStream_t stream;

  launchKernel(stream);
  AMPI_GPU_Iinvoke(stream, &req);
  MPI_Wait(&req, MPI_STATUS_IGNORE);
  ...
```

Figure 6.11: AMPI application usage of our nonblocking kernel completion API.

based interface that we provide, in which the user can test or wait on completion of the request object after invoking the kernel. Underneath, we track completion of the kernel(s) on the CUDA stream, and invoke a Charm++ callback afterwards, which either invokes the user's registered function or marks the request object as complete. This allows the PE scheduler to continue running while kernels are being executed on the GPU asynchronously.

We evaluate both weak and strong scaling performance of Jacobi-3D using up to 256 nodes (1,536 GPUs) of Summit, comparing the time per iteration of the host-staging and GPU-aware communication mechanisms. Both are using the asynchronous kernel completion detection support in AMPI. Jacobi-3D is weak scaled with a base domain size of $1{,}536^3$ double values and each dimension doubled in x, y, z order. Strong scaling experiments executed on eight to 256 nodes maintain the domain size of $3{,}072^3$ doubles.

Figures 6.12 and 6.13 show the weak and strong scaling performance, respectively. We see that GPU-aware communication outperforms host-staging by up to 2x, and that with these optimizations AMPI is now competitive with OpenMPI in most cases. At small node-counts with weak scaling, and at large node counts when strong scaling we see AMPI outperform OpenMPI, while OpenMPI performs better for other sizes.

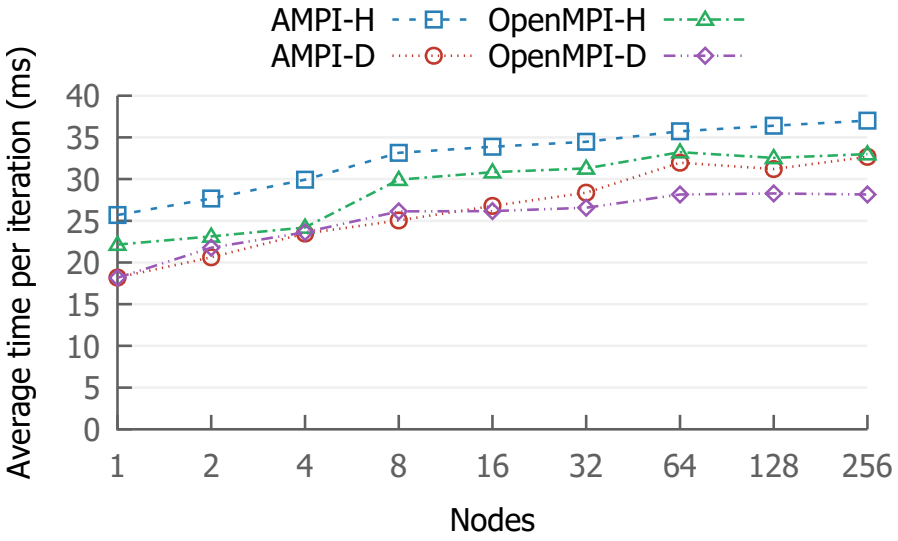



Figure 6.12: Weak scaling performance for Jacobi-3D of AMPI compared to OpenMPI.

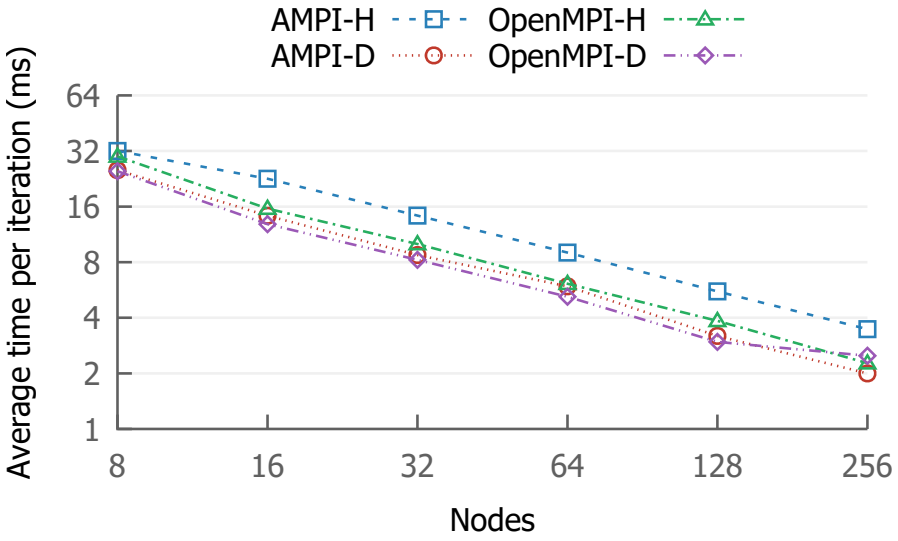



Figure 6.13: Strong scaling performance for Jacobi-3D of AMPI compared to OpenMPI.

### 6.3.3 Conclusion

We have tried identified and addressed two impediments to using AMPI on GPU accelerated systems. The first hurdle was in handling communication of device-resident buffers without staging copies of the buffer through the host's memory. We accomplished this using RDMA capabilities provided by the UCX networking library. We improved the point-to-point latency and bandwidth significantly, but work remains to bring the overheads down to be competitive with other GPU-aware MPI libraries. Further, we have not explored collective communication routines for device-resident buffers. Secondly, we identified the need for asynchronous launch and completion of GPU kernels. We developed support for Charm++'s HAPI interface, which uses Charm++ callbacks to signal completion asynchronously, wrapping them in MPI request objects. This work allows AMPI applications to run more efficiently on GPUs, but much research remains in this area.

## 6.4 CHARM++ INTEROPERATION

Charm++ is one of the most widely used task-based parallel programming models which are receiving much interest as the HPC community moves into the exascale era. Charm++ of course underlies AMPI's implementation and we have discussed many changes to it throughout this thesis, but here we focus on interoperating Charm++ and MPI code using AMPI as the MPI implementation. This brings three potential advantages: one, it allows matching the overdecomposed objects in a Charm++ program with MPI ranks, without the need for users to explicitly map data from chares to processes. That mapping requires communication and synchronization if Charm++ objects are migrating dynamically around the system. Second, it allows composing MPI and Charm++ so that MPI ranks can be co-scheduled along with Charm++ chares and their entry method invocations. This opens the door to new workflows in which MPI and Charm++ overlap their execution while sharing PEs. Third, it avoids the need for building Charm++ on top of MPI as its communication substrate. Charm++ usually performs best when built directly on top of the lowest level networking API supported on a given system, such as Cray uGNI or IBM PAMI. Because Charm++'s MPI layer is based on two-sided MPI point-to-point communication and because Charm++ communication is essentially all unexpected from the point of view of MPI, and because MPI imposes extra semantics such as non-overtaking messages between each pair of communicating ranks, Charm++'s MPI layer implementation often performs worse than native layers. Charm++ and MPI interoperation currently requires MPI as the networking layer. AMPI-Charm++ interoperation frees Charm++ users from the need to run atop the MPI

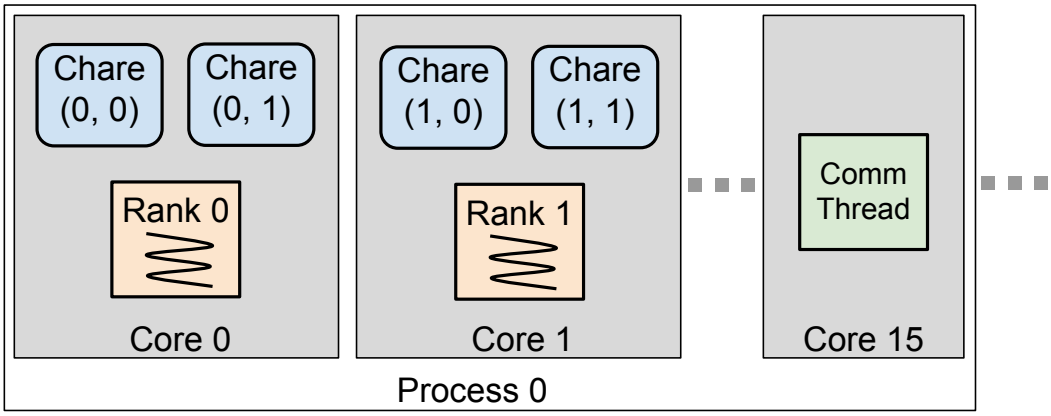



Figure 6.14: AMPI-Charm++ interoperability enables cooperative scheduling of chare array element entry methods and threaded AMPI virtual ranks. Here a 2D chare array interoperates with AMPI.

layer.

We support AMPI-Charm++ interoperation by essentially exposing AMPI's underlying chare array to Charm++ users. We modified AMPI's startup procedure to enable passing a custom chare array map into it so that users can map AMPI ranks arbitrarily, most likely to match their chare array decomposition and mapping. Next we provide a method for users to inject messages into AMPI from Charm++ code, so that they can communicate between the two, rather than simply accessing shared state between co-located chares and ranks. We expose AMPI's message object class, *AmpiMsg*, to applications including our interoperability header and support sending messages from chares to AMPI ranks by allowing them to inject a message into MPI in two modes: one, the sender chare has to send its message through a PE-local AMPI rank, masquerading as it; or two, the sent message will have a special AMPI_CHARM_SOURCE sender field that the receiver can match. These messages are unsequenced and so can overtake each other. Currently, we only support point-to-point messages and broadcasts, but we could potentially support MPI intercommunicator-like collective semantics for other collective operations whose semantics have equivalents in Charm++ such as reduce and allreduce. In order to communicate from AMPI back to Charm++, users can serialize Charm++ callback objects and send them as parts of messages to AMPI ranks, which the rank can then invoke anytime after deserializing it from an MPI_BYTE typed message buffer. Currently, this requires that the MPI code is C++ in order to be able to invoke Charm++ entry methods and use Charm++'s Pack-UnPack (PUP) serialization framework.

For future work, we would like to improve the usability of this interoperable library by supporting more communication routines between Charm++ and AMPI and by supporting

C and Fortran interfaces. We would like to prove AMPI-Charm++ interoperability in a full-scale Charm++ application. Quinoa, an adaptive computational fluid dynamics code developed on Charm++ at Los Alamos National Laboratory, is one such candidate [83]. Quinoa currently requires building on top of Charm++'s MPI layer and must aggregate data from overdecomposed migratable chares back to the node-level before passing off execution to Zoltan for partitioning and parallel HDF5 for file output. AMPI interoperability can address both of those restrictions as well as supporting interleaved execution of AMPI ranks and Charm++ chares.

## 6.5 CONCLUSION

Interoperability is critical to high performance computing application developers. As hardware systems become more heterogeneous and increasingly parallel the need for accelerator and shared memory programming models is growing. Throughout this thesis we have explored how AMPI's virtualization approach can be more efficient on these systems by becoming aware of the hierarchical nature of current machines. For example, we optimized communication separately for intra-PE transfers from transfers across PEs in the same address space. But, there remain strong practical arguments for different types of applications and users to adopt a more hierarchical hybrid programming model. In this chapter, we have pursued interoperation of AMPI with OpenMP, CUDA, and Charm++, though the techniques described apply to other programming models like them. For OpenMP and shared memory models, we developed an integrated runtime for AMPI+OpenMP applications that benefit from PE-level virtualization but have transient load imbalances within a timestep that are otherwise difficult for AMPI's persistence-based load balancing to address. For using GPUs, we integrated support in the runtime for efficient RDMA communication between GPUs and for asynchronous scheduling of work on them. Finally, we explored interoperation with Charm++ as an alternative to Charm++-MPI interoperability.

For future work, we would like to explore more dynamic configurations of OpenMP integration, where the runtime could alter which PEs host AMPI ranks and which PEs are scheduled by OpenMP. Currently, the configuration is static, but the optimal choice could be different for various phases of an application or for different libraries. For GPUs, much work remains to optimize AMPI and to efficiently schedule fine-grained virtualized work units on GPUs. Collective communication of buffers residing on device memory without falling back to point-to-point communication and prioritized scheduling of GPU communication handling over computation could both improve our support. Lastly, we would like to explore use cases for Charm++ interoperation with AMPI. The Quinoa application mentioned

previously has motivated our proof-of-concept implementation, but this requires changes to any MPI libraries used as well as the application code.

# CHAPTER 7: CONCLUSION

This thesis has sought to identify and address the various challenges associated with virtualization of MPI ranks as migratable user-level threads. We first considered the limitations of existing virtualization techniques in order to motivate the development of PIEglobals, our novel runtime privatization method that achieves an unprecedented combination of automation, portability, runtime performance, and support for user-level rank migration. Next, we identified memory usage as a limiting factor for AMPI applications and minimized the persistent per-rank memory footprint and transient communication-related memory overheads, enabling applications to run more efficiently and with larger problem sizes. We then examined point-to-point communication with respect to latency, bandwidth, and asynchrony. We exploited the shared address space between ranks on a node for not only faster memory copies but also increased concurrency and asynchrony, optimizing for both the full MPI semantics and a relaxed set that offers higher performance for applications that can tolerate those semantics. Next, we considered collective communication performance in terms of virtualization and migration overhead. We sought to limit the effects of virtualization through optimizations within the shared address space, and identified non-commutative allreduce operations as a particularly interesting case for optimization, motivating the development of rank placement-adaptive algorithms. Finally, we considered interoperation with other programming models such as OpenMP, CUDA, and Charm++. We developed new interfaces and integrated runtime scheduling support in order to incorporate those models efficiently into AMPI's virtualized execution model.

Along the way we have tried to summarize the main takeaways and to identify opportunities for further research, but we restate them here for reference:

- **Chapter 2: Automatic Process Virtualization**

  We surveyed existing methods for privatization of global state and identified shortcomings in terms of portability and support for dynamic rank migration. That motivated our development of PIEglobals, a fully automatic runtime process virtualization technique. PIEglobals supports high levels of overdecomposition and rank migration while being more portable than PIPglobals and all TLS-based methods. Using it, we have been able to more quickly get applications running with load balancing than ever before. Its main drawbacks are memory overhead and the difficulty of debugging virtualized programs due to the duplication and relocation of code segments. For future work we intend to prove PIEglobals on more production applications and to minimize its memory usage where possible. Integrating support for debugging PIEglobals pro-

grams into a debugger tool would also greatly improve the user experience of debugging PIEglobals failures.

- **Chapter 3: Memory Usage Optimizations**

  We identified different sources of memory overhead coming from AMPI's rank virtualization, classifying it in terms ownership, mutability, and usage. We then sought to deal with transient sources of memory overhead through memory pooling and the development of in-place communication and migration capabilities. This necessitated developing support for new communication primitives and interfaces in AMPI's underlying tasking runtime. This limited the peak memory usage during both communication phases and load balancing. We also hoisted the storage of immutable internal MPI objects such as MPI_Groups and MPI_Datatypes to the node-level rather than being stored per-rank. This reduced the memory footprint associated with rank virtualization. Overall, this work allowed us to run at unprecedented scale and for larger problem sizes than were possible previously. For future work, we would like to make load balancing strategies explicitly aware of memory usage and to stage migrations over time in order to minimize spikes in peak memory usage during load rebalancing phases. The use of MPI shared memory windows in AMPI applications could also be beneficial for minimizing per-rank memory footprint.

- **Chapter 4: Point-to-Point Communication Optimizations**

  We made use of the shared address space between ranks on each node in order to optimize for communication locality and to enable more asynchronous communication. Shared address space is not only portable across all systems but provides for fast and predictable memory copies and memory sharing. Taking advantage of it inside our runtime made possible fine-grained concurrency and various optimizations for asynchrony. We pursued two paths toward higher performing point-to-point communication: 1) within the MPI semantics, we optimized AMPI with prioritized scheduling, asynchronous completion of requests, and communication protocol-aware thread resumption policies, and 2) relaxing the usual MPI semantics for non-overtaking messages and wildcard receives, and building on related work in the context of MPI+X, we adapted concurrent message matching support for AMPI's endpoints model, combining that with a novel locality-aware message matching scheme. Overall, we showed that for applications that can tolerate the relaxed semantics our optimized runtime offers the best performance, particularly when running with more virtual ranks than cores. For future work, we would like to make the relaxed semantics optimizations

applicable at the scope of communicators, and to apply this work to more applications. For example, this work could potentially be applied to applications that use wildcards as long as the sender and receiver both know that the message will be received as such. Our concurrent message matching support could also be absorbed into Charm++'s zero copy Post API implementation, and we could potentially push the concurrency of our multiple endpoints further down the Charm++ stack to its networking layer where multiple endpoints could drive the network instead of a single dedicated communication thread per process..

- **Chapter 5: Collective Communication Optimizations**

  We analyzed the performance implications of rank virtualization and rank migration on collective communication routines. We identified virtualization's impact on commonly used collective routines such as broadcast and allreduce, and migration's effects on non-commutative reduction operations in particular. We sought to limit the overheads through virtualization and shared address space aware algorithms, and by utilizing in-place communication for large messages. This also necessitated various semantic changes and additions to Charm++'s programming model. Then, we developed novel adaptive algorithms for non-commutative allreduce operations over migrating ranks which may be disordered in their mapping across the cores and nodes of the system. For future work, we plan on investigating other collective communication routines for optimization in the context of migratable virtualized ranks, such as applying our placement-adaptive non-commutative allreduce algorithms to allgather. We also would like to explore use cases for non-commutative reduction operations in applications that can benefit from dynamic load balancing, and to develop load balancing strategies that take into account the use of non-commutative operators when migrating ranks in order to maintain rank-continuity on each node.

- **Chapter 6: Interoperability with other Parallel Programming Models**

  While we have contrasted our endpoints model of thread-based virtual MPI ranks against MPI-only and MPI+X throughout, we also acknowledge the utility of interoperation for various usage models. We explored incorporating OpenMP parallelism into AMPI's execution model by integrating its runtime scheduling into ours. This achieves effective load balancing for transient imbalances within each timestep of a simulation, and also allows nicer composition of AMPI-everywhere codes with AMPI+OpenMP libraries. We also sought to integrate support for efficient GPU point-to-point communication and asynchronous CUDA kernel invocation, and for Charm++ interoperation.

For future work, we would like to make the choice of AMPI overdecomposition versus OpenMP parallelism dynamically adaptable by the runtime, such that the choice of configuration can be tuned at runtime. We would also like to identify use cases for AMPI-Charm++ interoperation and to pursue collective communication and alternative messaging semantics for GPUs.

Throughout our work we have striven to make AMPI more applicable to all kinds of existing MPI codes whether they require privatization of mutable global state, run with large memory sizes, or use other programming models such as OpenMP; we have tried to lower the costs of virtualization and migration through per-rank memory footprint reductions, communication locality optimizations, and algorithmic adaptations to collective communication routines; and we have attempted to make use of the shared address space among virtualized ranks for efficient resource utilization within each node, whether that be for more asynchronous communication, lower peak memory usage, or for efficient shared memory parallelism. Throughout we have demonstrated the need for careful attention to the semantics of different parallel programming models–be it MPI, Charm++, OpenMP, or CUDA–since they can have major impacts on performance in ways that are subtle or non-obvious to application users.

Broadly, this thesis has sought to define and address the challenges surrounding virtualization of an existing bulk-synchronous parallel programming model. Our existing model has been MPI, since it is widely used in high performance numerical computing, but the insights and techniques are applicable to virtualization of other parallel programming models and to parallel runtime systems generally. We can imagine virtualizing many other parallel programming models, for example, ones as disparate as Chapel [19] and Coarray Fortran [84]. Chapel's concept of locales has traditionally been tied to hardware units such as a core, socket, or node, but these could be virtualized, tasks for each locale could be co-scheduled, and virtual locales made migratable across address spaces for dynamic load balancing purposes. The same idea could be applied to Coarray Fortran images, which are traditionally implemented as processes. So while our holistic approach has resulted in an implementation of our ideas in AMPI and its underlying task-based runtime system, we hope that our work not only finds use by MPI application developers but also that our ideas spur interest from the parallel programming models community in alternative implementations of existing models used by legacy codes and in new ways of incorporating our runtime techniques into parallel programming models.

## REFERENCES


[1] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "Toward efficient support for multithreaded mpi communication," in *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2008. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87475-1_20 pp. 120–129.

[2] A. Amer, H. Lu, Y. Wei, P. Balaji, and S. Matsuoka, "Mpi+threads: Runtime contention and remedies," *SIGPLAN Not.*, vol. 50, no. 8, pp. 239–248, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2858788.2688522

[3] A. Amer, H. Lu, P. Balaji, M. Chabbi, Y. Wei, J. Hammond, and S. Matsuoka, "Lock contention management in multithreaded mpi," *ACM Trans. Parallel Comput.*, vol. 5, no. 3, pp. 12:1–12:21, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3275443

[4] H.-V. Dang, S. Seo, A. Amer, and P. Balaji, "Advanced thread synchronization for multithreaded mpi implementations," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGrid '17. Piscataway, NJ, USA: IEEE Press, 2017. [Online]. Available: https://doi.org/10.1109/CCGRID.2017.65 pp. 314–324.

[5] H.-V. Dang, M. Snir, and W. Gropp, "Towards millions of communicating threads," in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI 2016. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2966884.2966914 pp. 1–14.

[6] R. Zambre, A. Chandramowliswharan, and P. Balaji, "How i learned to stop worrying about user-visible endpoints and love MPI," in *Proceedings of the 34th ACM International Conference on Supercomputing*. ACM, jun 2020. [Online]. Available: https://doi.org/10.1145%2F3392717.3392773

[7] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," in *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, ser. PDP '09. Washington, DC, USA: IEEE Computer Society, 2009. [Online]. Available: http://dx.doi.org/10.1109/PDP.2009.43 pp. 427–436.

[8] M. Diener, S. White, L. V. Kale, M. Campbell, D. J. Bodony, and J. B. Freund, "Improving the memory access locality of hybrid mpi applications," in *Proceedings of the 24th European MPI Users' Group Meeting*, ser. EuroMPI '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3127024.3127038 pp. 11:1–11:10.

[9] S. Moreaud, B. Goglin, R. Namyst, and D. Goodell, "Optimizing mpi communication within large multicore nodes with kernel assistance," in *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, April 2010, pp. 1–7.

[10] L. Chai, A. Hartono, and D. K. Panda, "Designing high performance and scalable mpi intra-node communication support for clusters," in *2006 IEEE International Conference on Cluster Computing*, Sept 2006, pp. 1–10.

[11] J. Vienne, "Benefits of cross memory attach for mpi libraries on hpc clusters," in *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, ser. XSEDE '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2616498.2616532 pp. 33:1–33:6.

[12] H.-W. Jin and D. K. Panda, "Limic: Support for high-performance mpi intra-node communication on linux cluster," in *Proceedings of the 2005 International Conference on Parallel Processing*, ser. ICPP '05. Washington, DC, USA: IEEE Computer Society, 2005. [Online]. Available: https://doi.org/10.1109/ICPP.2005.48 pp. 184–191.

[13] S. Chakraborty, H. Subramoni, and D. K. Panda, "Contention-aware kernel-assisted mpi collectives for multi-/many-core systems," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 13–24.

[14] C. Huang, O. Lawlor, and L. V. Kalé, "Adaptive MPI," in *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, College Station, Texas, October 2003, pp. 306–322.

[15] M. Perache, P. Carribault, and H. Jourdren, "MPC-MPI: An MPI implementation reducing the overall memory consumption," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI 2009)*, ser. Lecture Notes in Computer Science, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2009, vol. 5759, pp. 94–103. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03770-2_16

[16] H. Kamal and A. Wagner, "FG-MPI: Fine-Grain MPI for multicore and clusters," in *The 11th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDESC)*. IEEE, Apr. 2010.

[17] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur, *Enabling MPI interoperability through flexible communication endpoints*. Association for Computing Machinery, 2013, pp. 13–18.

[18] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel Programming with Migratable Objects: Charm++ in Practice," ser. SC, 2014.

[19] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *Int. J. High Perform. Comput. Appl.*, vol. 21, pp. 291–312, August 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1286120.1286123

[20] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014, p. 6.

[21] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: expressing locality and independence with logical regions," in *Proceedings of the international conference on high performance computing, networking, storage and analysis*. IEEE Computer Society Press, 2012, p. 66.

[22] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé, "Performance Evaluation of Adaptive MPI," in *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[23] E. R. Rodrigues, P. O. A. Navaux, J. Panetta, C. L. Mendes, and L. V. Kale, "Optimizing an MPI Weather Forecasting Model via Processor Virtualization," in *Proceedings of International Conference on High Performance Computing (HiPC)*, 2010.

[24] S. Bak, H. Menon, S. White, M. Diener, and L. V. Kalé, "Multi-level load balancing with an integrated runtime approach," in *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018*, 2018. [Online]. Available: https://doi.org/10.1109/CCGRID.2018.00018 pp. 31–40.

[25] N. Jain, A. Bhatele, S. White, T. Gamblin, and L. V. Kale, "Evaluating hpc networks via simulation of parallel workloads," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16 (to appear), 2016.

[26] G. Zheng, G. Kakulapati, and L. V. Kalé, "Bigsim: A parallel simulator for performance prediction of extremely large parallel machines," in *18th International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, New Mexico, April 2004, p. 78.

[27] J. Ang, A. A. Chien, S. D. Hammond, A. Hoisie, I. Karlin, S. Pakin, J. Shalf, and J. Vetter, "Reimagining codesign for advanced scientific computing: Report for the ascr workshop on reimagining codesign," 10 2021. [Online]. Available: https://www.osti.gov/biblio/1822199

[28] N. Jain, A. Bhatele, J.-S. Yeom, M. F. Adams, F. Miniati, C. Mei, and L. V. Kale, "Charm++ & MPI: Combining the best of both worlds," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (to appear)*, ser. IPDPS '15. IEEE Computer Society, May 2015, lLNL-CONF-663041.

[29] M. Pérache, H. Jourdren, and R. Namyst, "MPC: A unified parallel runtime for clusters of NUMA machines," in *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '08. Berlin, Heidelberg: Springer-Verlag, 2008. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85451-7_9 p. 78–88.

[30] M. P. I. Forum, *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*. High-Performance Computing Center Stuttgart, University of Stuttgart, 2015. [Online]. Available: https://books.google.com/books?id=Fbv7jwEACAAJ

[31] S. Negara, G. Zheng, K.-C. Pan, N. Negara, R. E. Johnson, L. V. Kalé, and P. M. Ricker, "Automatic mpi to ampi program transformation using photran," in *Proceedings of the 2010 Conference on Parallel Processing*, ser. Euro-Par 2010. Berlin, Heidelberg: Springer-Verlag, 2010, p. 531–539.

[32] 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)*, no. 10-14, Ischia/Naples/Italy, August 2010.

[33] G. Zheng, S. Negara, C. L. Mendes, E. R. Rodrigues, and L. V. Kale, "Automatic handling of global variables for multi-threaded mpi programs," in *Proceedings of the 16th International Conference on Parallel and Distributed Systems (ICPADS) 2011*, no. 11-23, December 2011.

[34] S. White and L. V. Kale, "Optimizing point-to-point communication between adaptive mpi endpoints in shared memory," *Concurrency and Computation: Practice and Experience*, pp. n/a–n/a, 2018. [Online]. Available: http://dx.doi.org/10.1002/cpe.4467

[35] J.-B. Besnard, J. Adam, S. Shende, M. Pérache, P. Carribault, J. Jaeger, and A. D. Maloney, "Introducing task-containers as an alternative to runtime-stacking," in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI 2016. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2966884.2966910 p. 51–63.

[36] M. Tchiboukdjian, P. Carribault, and M. Pérache, "Hierarchical local storage: Exploiting flexible user-data sharing between mpi tasks," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 366–377.

[37] A. Hori, M. Si, B. Gerofi, M. Takagi, J. Dayal, P. Balaji, and Y. Ishikawa, "Process-in-process: Techniques for practical address-space sharing," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3208040.3208045 pp. 131–143.

[38] S. Browne, C. Deane, G. Ho, and P. Mucci, "Papi: A portable interface to hardware performance counters," 1999.

[39] K. Roberts, J. Dietrich, D. Wirasaet, W. Pringle, and J. Westerink, "Dynamic load balancing for predictions of storm surge and coastal flooding," vol. 140, no. 105045, 2021.

[40] Y. Xia, J. Goral, H. Huang, I. Miskovic, P. Meakin, and M. Deo, "Many-body dissipative particle dynamics modeling of fluid flow in fine-grained nanoporous shales," *Physics of Fluids*, vol. 29, no. 5, p. 056601, 2017. [Online]. Available: https://doi.org/10.1063/1.4981136

[41] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff, "Mpi on a million processors," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 20–30.

[42] D. Goodell, W. Gropp, X. Zhao, and R. Thakur, "Scalable memory use in mpi: A case study with mpich2," in *Recent Advances in the Message Passing Interface*, Y. Cotronis, A. Danalis, D. S. Nikolopoulos, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 140–149.

[43] S. Kumar, P. Heidelberger, and C. Stunkel, "Space performance tradeoffs in compressing mpi group data structures," in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI 2016. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2966884.2966911 pp. 32–40.

[44] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, "MPI+MPI: a new hybrid approach to parallel programming with MPI plus shared memory," *Computing*, vol. 95, no. 12, pp. 1121–1136, 2013.

[45] N. Bhat, S. White, and L. Kale, "Enabling support for zero copy semantics in an Asynchronous Task-based Programming Model," ser. AMTE, Euro-Par, 2021.

[46] M. Pérache, P. Carribault, and H. Jourdren, "Mpc-mpi: An mpi implementation reducing the overall memory consumption," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 94–103.

[47] A. Friedley, G. Bronevetsky, T. Hoefler, and A. Lumsdaine, "Hybrid MPI: efficient message passing for multi-core systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 18.

[48] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[49] W. Gropp and R. Thakur, "Thread-safety in an mpi implementation: Requirements and analysis," *Parallel Comput.*, vol. 33, no. 9, p. 595–604, sep 2007. [Online]. Available: https://doi.org/10.1016/j.parco.2007.07.002

[50] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "Fine-grained multithreading support for hybrid threaded mpi programming," *Int. J. High Perform. Comput. Appl.*, vol. 24, no. 1, p. 49–57, feb 2010. [Online]. Available: https://doi.org/10.1177/1094342009360206

[51] A. Amer, H. Lu, Y. Wei, P. Balaji, and S. Matsuoka, "Mpi+threads: Runtime contention and remedies," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2688500.2688522 p. 239–248.

[52] D. Goodell, P. Balaji, D. Buntinas, G. Dozsa, W. Gropp, S. Kumar, B. R. d. Supinski, and R. Thakur, "Minimizing mpi resource contention in multithreaded multicore environments," in *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, ser. CLUSTER '10. USA: IEEE Computer Society, 2010. [Online]. Available: https://doi.org/10.1109/CLUSTER.2010.11 p. 1–8.

[53] T. Hoefler and A. Lumsdaine, "Message progression in parallel computing - to thread or not to thread?" in *2008 IEEE International Conference on Cluster Computing*, 2008, pp. 213–222.

[54] M. Si and P. Balaji, "Process-based asynchronous progress model for mpi point-to-point communication," in *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Dec 2017, pp. 206–214.

[55] R. Brightwell, R. Riesen, and K. D. Underwood, "Analyzing the impact of overlap, offload, and independent progress for message passing interface applications," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 2, p. 103–117, may 2005. [Online]. Available: https://doi.org/10.1177/1094342005054257

[56] A. Ruhela, H. Subramoni, S. Chakraborty, M. Bayatpour, P. Kousha, and D. K. Panda, "Efficient asynchronous communication progress for mpi without dedicated resources," in *Proceedings of the 25th European MPI Users' Group Meeting*, ser. EuroMPI'18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3236367.3236376 pp. 14:1–14:11.

[57] H. P. J. Pritchard, D. Roweth, D. Henseler, and P. Cassella, "Leveraging the cray linux environment core specialization feature to realize mpi asynchronous progress on cray xe systems," 2012.

[58] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, Spring 2005.

[59] R. L. Graham and G. Shipman, "Mpi support for multi-core architectures: Optimized shared memory collectives," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, A. Lastovetsky, T. Kechadi, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 130–140.

[60] P. Sack and W. Gropp, "Faster topology-aware collective algorithms through non-minimal communication," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2145816.2145823 p. 45–54.

[61] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. Panda, "Efficient inter-node mpi communication using gpudirect rdma for infiniband clusters with nvidia gpus," 10 2013, pp. 80–89.

[62] P. Patarasuk and X. Yuan, "Bandwidth efficient all-reduce operation on tree topologies," 04 2007, pp. 1 – 8.

[63] K. Kandalla, H. Subramoni, A. Vishnu, and D. Panda, "Designing topology-aware collective communication algorithms for large scale infiniband clusters: Case studies with scatter and gather," 05 2010, pp. 1 – 8.

[64] J. Zhang, J. Zhai, W. Chen, and W. Zheng, "Process mapping for mpi collective communications," in *Euro-Par 2009 Parallel Processing*, H. Sips, D. Epema, and H.-X. Lin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 81–92.

[65] T. Ma, T. Herault, G. Bosilca, and J. J. Dongarra, "Process distance-aware adaptive mpi collective communications," in *2011 IEEE International Conference on Cluster Computing*, 2011, pp. 196–204.

[66] A. Bhatele, T. Gamblin, S. H. Langer, P.-T. Bremer, E. W. Draeger, B. Hamann, K. E. Isaacs, A. G. Landge, J. A. Levine, V. Pascucci, M. Schulz, and C. H. Still, "Mapping applications with collectives over sub-communicators on torus networks," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Washington, DC, USA: IEEE Computer Society Press, 2012.

[67] M. Ruefenacht, M. Bull, and S. Booth, "Generalisation of recursive doubling for allreduce," in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI 2016. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2966884.2966913 p. 23–31.

[68] NERSC, "Cori," 2022. [Online]. Available: https://www.nersc.gov/systems/cori/

[69] J. L. Träff, "An improved algorithm for (non-commutative) reduce-scatter with an application," in *Proceedings of the 12th European PVM/MPI Users' Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. PVM/MPI'05. Berlin, Heidelberg: Springer-Verlag, 2005. [Online]. Available: https://doi.org/10.1007/11557265_20 p. 129–137.

[70] Message Passing Interface Forum, "MPI: A Message Passing Interface," in *Proceedings of Supercomputing '93*. IEEE Computer Society Press, 1993. [Online]. Available: citeseer.ist.psu.edu/forum93mpi.html pp. 878–883.

[71] N. Hjelm, "Linux cross-memory attach," 2016. [Online]. Available: https://github.com/hjelmn/

[72] J. Lifflander and P. P. Pébay, "Darma/vt fy20 mid-year status report," 4 2020. [Online]. Available: https://www.osti.gov/biblio/1615717

[73] A. Faraj, X. Yuan, and D. Lowenthal, "Star-mpi: self tuned adaptive routines for mpi collective operations," in *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*. New York, NY, USA: ACM, 2006, pp. 199–208.

[74] F. Petrini, D. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q," in *ACM/IEEE SC2003*, Phoenix, Arizona, Nov. 10–16, 2003.

[75] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[76] V. Kale, A. Bhatele, and W. D. Gropp, "Weighted locality sensitive scheduling for mitigating noise on multicore clusters," in *18th annual IEEE International Conference on High Performance Computing (HiPC 2011)*, December 2011.

[77] V. Kale, A. Randles, and W. D. Gropp, "Locality-optimized mixed static/dynamic scheduling for improving load balancing on SMPs," in *Proceedings of the 21st European MPI Users' Group Meeting*. ACM, 2014, p. 115.

[78] S. Donfack, L. Grigori, W. D. Gropp, and V. Kale, "Hybrid static/dynamic scheduling for already optimized dense matrix factorization," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 496–507.

[79] S. Bak, H. Menon, S. White, M. Diener, and L. Kale, "Integrating openmp into the charm++ programming model," in *Proceedings of the Third International Workshop on Extreme Scale Programming Models and Middleware*, ser. ESPM2'17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3152041.3152085 pp. 4:1–4:7.

[80] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," in *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, Santa Barbara, California, July 1995, mIT. pp. 207–216.

[81] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2005, pp. 21–28.

[82] A. J. Kunen, T. S. Bailey, and P. N. Brown, "KRIPKE - a massively parallel transport mini-app," Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep., 2015.

[83] J. Bakosi, R. Bird, F. Gonzalez, C. Junghans, W. Li, H. Luo, A. Pandare, and J. Waltz, "Asynchronous distributed-memory task-parallel algorithm for compressible flows on unstructured 3d eulerian grids," *Advances in Engineering Software*, vol. 160, p. 102962, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0965997820310085

[84] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, p. 1–31, aug 1998. [Online]. Available: https://doi.org/10.1145/289918.289920