TECHNIQUES FOR COMMUNICATION OPTIMIZATION OF PARALLEL PROGRAMS IN AN ADAPTIVE RUNTIME SYSTEM

BY

MICHAEL P. ROBSON

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 at Urbana-Champaign, 2020

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kalé, Chair
Professor Josep Torellas
Associate Professor Craig Zilles
Professor Thomas Quinn, University of Washington

ABSTRACT

With the current continuation of Moore's law and the presumed end of improved single core performance, high performance computing (HPC) has turned to increased on-node parallelism in order to address ever growing challenges and numbers of transistors. While this has resulted in a continued increase in overall computing performance, supercomputer networks have lagged far behind in their development and are now oftentimes the singular bottleneck in achieving performance and scalability in modern HPC applications. New machines are consistently built with 'deeper' nodes that improve the single node compute performance, as measured by the achievable floating point operations per second (FLOPs), relative to earlier generations with a corresponding increase in network bandwidth or sufficient decrease in latency. This unequal increase has previously partially been addressed by partitioning duties between runtimes at the shared memory node level, e.g. OpenMP, and distributed memory communication level, e.g. MPI, to create a model known as MPI+X. In this work, we present an alternative approach to improving the performance of modern HPC applications utilizing current generation supercomputer networks. We focus on the combination of several of the benefits of the Charm++ programming model, namely overdecompsition, with OpenMP and the ability to 'spread' work across several cores. This allows applications to smoothly inject messages onto the network, constantly overlapping their communication requirements with their compute phases, our overall focus for this work. We further describe a complementary suite of techniques to fully utilize modern supercomputers and balance FLOPs and communication. We extend these techniques through micro-benchmark studies and integration into the production scale Charm++ runtime. We also turn our attention from internode communication optimization to apply these same techniques to intranode communication between various hardware devices, i.e. CPUs and graphics processing units, as well. We also discuss many of the tradeoffs of these approaches and attempt to quantify their general effect. While embodied in the Charm++ runtime system, these ideas are applicable to a wide swath of communication bound applications, a class of programs that we expect to only grow over time with the continuing trend of increased differential between node and network performance.

To my wife, Halie, without whom this would not have been possible. To my parents, for their love and support. To my gaming group, friends, and all other loved ones who helped me along the way.

ACKNOWLEDGMENTS

The author would like to thank Kavitha Chandrasekar, Jaemin Choi, and Halie Rando as well as the following sources of support. This research used resources of the Oak Ridge Leadership Computing Facility, which is a Department of Energy (DOE) Office of Science User Facility supported under Contract DE-AC05-00OR22725. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. This work used the Extreme Science and Engineering Discovery Environment (XSEDE) Stampede2 and Bridges at the Texas Advanced Computing Center and Pittsburgh Supercomputing Center through allocation TG-ASC050039N. We gratefully acknowledge the support of NVIDIA Corporation with the donation of GPUs used for this research. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

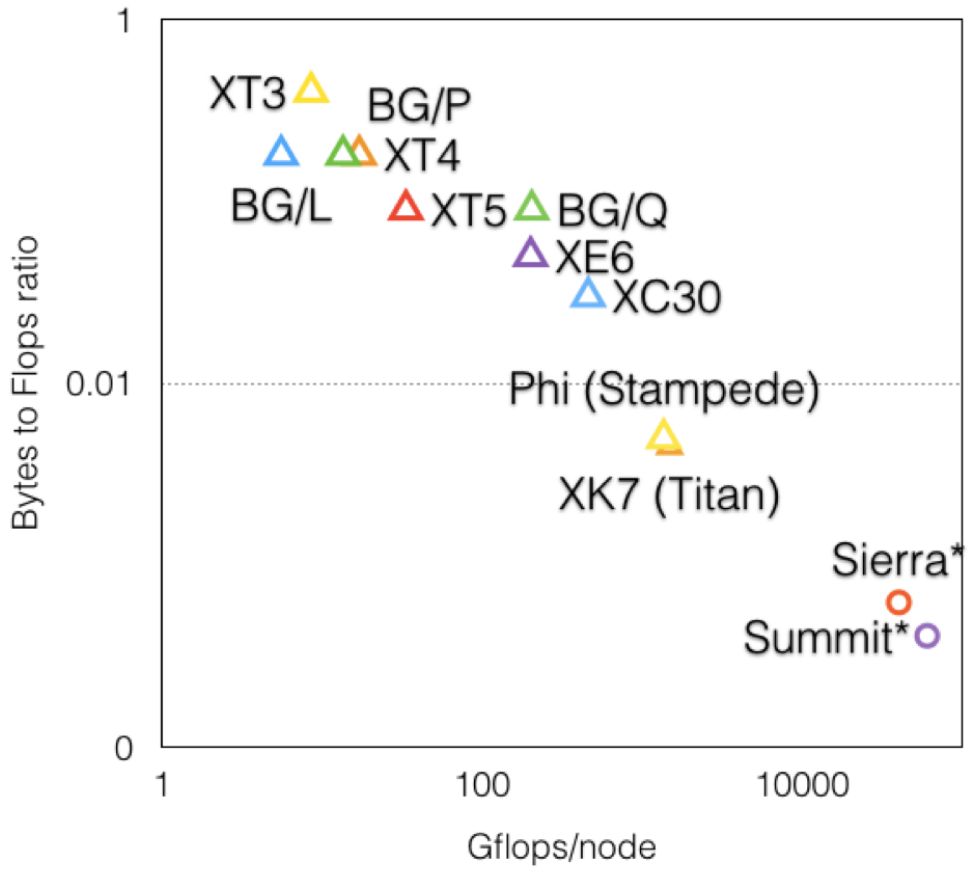
Since the 1920s, there has been significant interest in engineering computing machines that can solve complex problems. In the 1960s, Seymour Cray identified that silicon transitors that used the planar process could, with refrigeration to cool them down, run computations extremely fast. He built the CDC 6600 which ran computations ten times faster than existing computers and was thus deemed a "supercomputer". Thus, since its inception, the field of supercomputing has revolved around the goal of generating the computational power needed to solve increasingly complex problems.

Over time, however, society's conceptualization of complex problems has changed, in part because the field of supercomputing has evolved. Success was originally framed in terms of floating point operations per second (FLOPS). In 1964, Cray's CDC 6600 was able to sustain 500 kiloflops per processor while addressing standard mathematical problems. A decade later, machines could sustain computation on the megaflop scale, such as the CDC STAR-100 in 1974, and then a decade after that, on the gigaflop scale, such as M-13 in 1985. In the 1990s, the Top500 list began to track advances in supercomputing speed by recording the peak speed (Rmax) on record worldwide.

The trends in the rate of change of computing speed was described in 1965 by Gordon Moore and is known as Moore's Law. Moore's Law states that the number of transistors on a microchip will double every 18 months to two years [1]. This trend has borne out not only in computing, but also in other fields. In line with the early emphasis on transistor speed as the primary influence on computing power, the field of supercomputing has tended to emphasis the construction of 'deeper' nodes, which are machines that have more computational power relative to their memory and bandwidth capacities [2, 3]. Other technological advances have also worked to increase computation speed. For example, within the past two decades, graphics processing units (GPUs) were introduced for general-purpose computing and have shown a rapid increase in popularity. As of this writing, GPUs compromise upwards of 35% of FLOPS in the list of the 500 fastest supercomputers, driven primarily by the ten fastest machines on the planet [4]. Partially as a result of this trend computational power is no longer the major issue constraining performance for many applications.

Early in supercomputing, the emphasis for performance was on standard mathematical operations. Today, however, in the age of big data, there are additional concerns that must be considered beyond how quickly a machine can, for example, multiply two matrices. Instead, for most analyses that seek to utilize high-performance computing, there are significant considerations about data movement and storage, which is dependent on both memory and

Figure 1.1: Trend of diminishing network performance relative to increasing computational performance in leaders of the Top500 list over time.

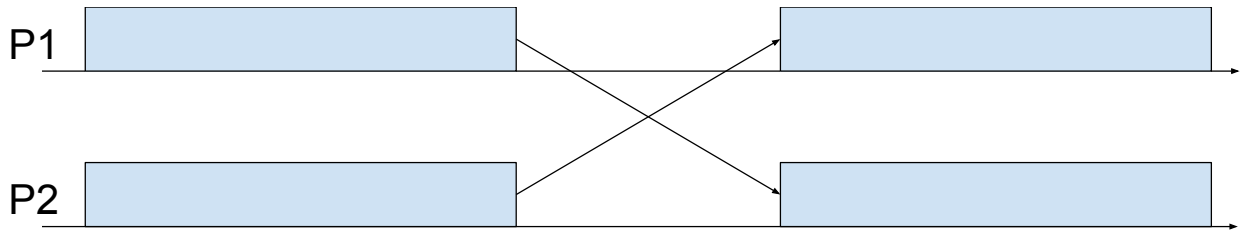


bandwidth. There is thus a significant disconnect between the shape of modern research questions well-suited to high-performance computing analysis, and the tasks supercomputers have been optimized to perform. A typical analysis is likely to involve many steps of moving data either from a CPU (host) to GPU (device), DRAM to CPU, GPU to GPU, node to node, etc., depending on experimental and cluster designs. Due to the myopic focus on computating speed, modern scientists find themselves in a regime where data movement costs are the dominant considerations in terms of both energy and computation time. In this dissertation, we primarily focus on inter-node communication. Figure 1.1 shows how injection bandwidth (i.e. number of bytes each node is able to inject into the network per second) has evolved in relation to the floating point performance of a node. As you go from left to right, from older machines to newer machines, GFLOPs/node increases, whlle the ratio of injection bytes to FLOP/s decreases. Instead of remaining constant, which would be ideal notion of balance, it actually decreases dramatically, by almost a factor of thousand!

This imbalance between computational power (FLOPS) and network capacity, in terms of both latency and bandwidth, is likely only to get worse.

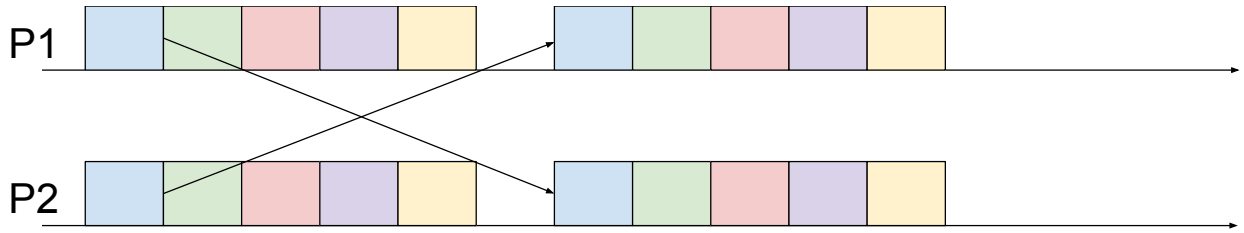
This disconnect introduces trade-offs that influence how users are able to interface with supercomputers and other performant machines. Looking forward, there are a few possible strategies for how machines could be optimized to confront these challenges. One potential solution would be to build machines, i.e. (super)computers, with more inter-node bandwidth. However, our current network technologies are already over engineered and expensive relative to their use. Assuming a fixed budget, improving network performance would therefore likely come at the cost of the compute power of a machine. Given the constraints on engineering machines that better balance processing, memory, and bandwidth, a more realistic question may therefore be how application performance can be improved to better match the current balance of network performance to computing power available. Phrased another way, are supercomputers currently using their networks effectively, and are there ways to design applications that balance the optimization of processing and networking? Today, many opimization efforts still tend to focus on FLOPs utilization. Network utilization is likely to present an increasingly critical challenge. Indeed, current parallel programming approaches typically lead to bursty communication, relying on networks for short durations of the critical path. See Figure 1.2 for an example. The horizontal axis is time. The two processes P1 and P2 alternate between computation and communication phases. The network is heavily utilized and is on the critical path during the communication phase, but is idle during the computation phase. These considerations are relevant not only to computer scientists and engineers looking to the future of supercomputing machinery, but also to all fields that utilize big data and high-performance computing.

Figure 1.2: Example of bulk synchronous approach without overlap.



An alternative possible solution to this problem is to consider overlap of communication and computation as a critical consideration in application development. This will reduce the importance of the network as a critical path component. It will also, as we will show, spread the communication over time, thus allowing for a less bursty, if not continuous, utilization of the network. Overlap would thus allow application developers to optimize performance

Figure 1.3: Example of overdecomposed approach with overlap.



of their applications given existing computing infrastructures and network constraints.

One could accomplish this by re-writing codes using languages that natively promote overlap, such as Charm++. In the Charm++ model, the data and computation are partitioned into a large number of objects, with typically many objects per processor. This is called "overdecomposition". Communication is addressed to objects, rather than processors, thus creating an object-to-object communication graph. Charm++ is able to dynamically balance load by migrating objects across processors. However, such load balancing is not the focus of this dissertation. From our point of view, what is important is the effect overdecomposition has on communication. Overdecomposition automatically creates adaptive overlap of communication and computation without any additional programmer effort. Figure 1.3 demonstrates this approach in action. Since there are multiple objects on a processor, they inject the messages into the network multiple times during a timestep. Even when each object injects only once, at the end of its computation phase, but together they end up spreading the injection of communication. Further, since the runtime system is scheduling objects based on availability of the data they are waiting for, no single objects blocks the processor waiting for communication. In this context, however, the question of how best to utilize existing computing infrastructure while minimizing memory and network considerations can be reframed in terms of (over)decomposition, i.e. to what degree a problem should be overdecomposed? Since the scheduling of an object is based on availability of data (messages), such overlap is *adaptive*, and essentially "free" from a programmer's perspective; however, there is overhead associated with overdecomposition as we will describe later in this section.

MPI is the most popular traditional parallel programming model for distributed memory machines. One can increase the extent of communication-computation overlap in MPI programs by using non-blocking send and receive operations, and moving send operations earlier in the code, and delaying waiting for the receive operations to as late as possible. Although this improves communication performance, notice that a program still injects data into the network only once, as before (retaining the burstiness of communication). One can also

explicitly program the overdecomposition approach in an MPI program, by using multiple blocks of data per processor, and using non-blocking communication and possibly event-driven loops to interleave execution of different blocks. Structured AMR applications are examples of such an approach with MPI. The reasoning and the techniques developed in this dissertation are applicable to such programs as well, although we will focus on Charm++ approach.

An alternative approach could be to rely on some higher level programming language, which, possibly with a compiler-assisted manner, can create programs that solve the problem of spreading communication over time and adaptively overlapping communication and computation. The long-term solution to addressing this problem then would be to re-write the large collection of existing high performance computing (HPC) applications using such a new language or framework. Is such a paradigm shift feasible? An illustration of a successful adoption of this approach is seen in CUDA for GPU programming. CUDA was designed by Nvidia to provide a parallel environment that facilitates the use of NVIDIA's GPUs for general purpose processing. In spite of the skepticism about the user community adopting a new way of writing programs, CUDA has succeeded spectacularly. However, this is arguably a unique case, where the process of adapting existing code to the GPU environment offered obvious computational gains for developers. A wider-scale shift would take a decade or longer at best. We therefore explore the more natural approach of identifying ways to shape the current and future applications in Charm++ (or even MPI as mentioned above) to optimize communication.

Overdecomposition could, by itself, have been an adequate solution for communication optimization, but for a significant and relatively recent challenge. The modern parallel machine consists of nodes that have a large number of cores working with a shared memory. One of the reasons for the imbalance between injection bandwidth and computational capabilities of a node, depicted in Figure 1.1, is indeed the fact that modern "nodes" include multiple processor chips, each with a large number of cores. To understand this challenge, let us visit the issue of overhead of overdecomposition mentioned earlier.

As the number of overdecomposed pieces increase, the overhead of scheduling each one of them also increases. Many common applications in science and engineering (such as stencil computations) involve communicating halos, or boundaries, with neighboring objects. The amount of memory dedicated to storing such halo regions, as well as the copying cost incurred for communicating halos, also adds to the overhead of overdecomposition. Further, in several applications, there is significant algorithmic cost to overdecomposition. For example, domain-decomposition based linear system solvers incur a higher number of iterations to reach convergence as the number of domains is increased. It is therefore important to

limit the degree of overdecomposition as much as possible.

In this context, the so-called "Charm++ everywhere" approach employed in the naive usage of Charm++ creates a large number of objects, which exacerbates the overheads mentioned above. Assuming a conservative ten objects per core, on a forty-core machine, the Charm++ everywhere approach would spawn four hundred objects. While this may sound like a large number of objects, it is in fact a very realistic situation given that the leading US supercomputer Summit contains 42 user-visible cores per node and can manage up to 800 objects (or more) on an 80 core machine. Similarly, Stampede2 at TACC has at least 96 cores not including hyperthreads. The hyperthreads would lead to a two- or four-fold increase again in the number of objects.

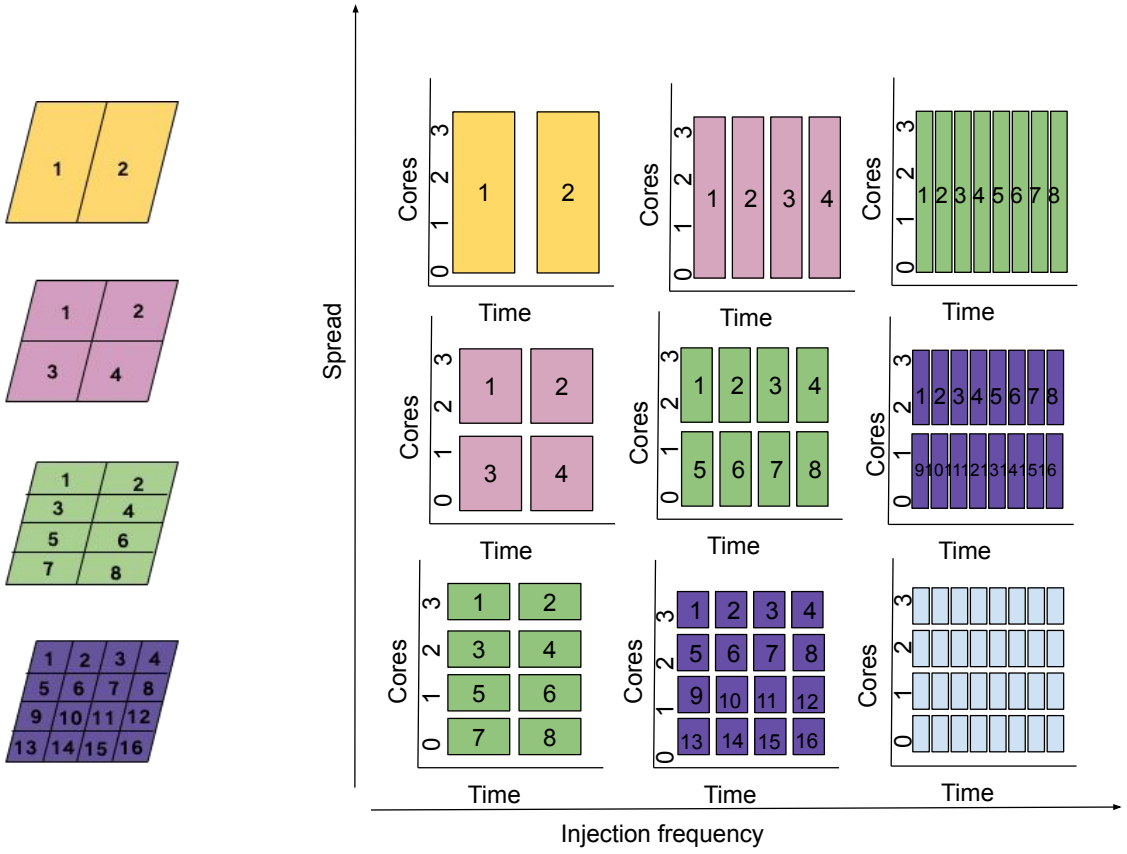
One potential alternative is to aggregate the work in a single per-node object, as in the MPI+X/OpenMP model, and spread the work across all cores using threads. However, thinly spreading this work can create its own problems with respect to OpenMP overheads. This dissertation addresses the challenge of dealing with large number of cores, while using overdecomposition to adequately spread communication injection over time and achieve good communication computation overlap so as to optimize overall program execution time. The basic idea, and the research hypothesis resulting from this, is described below.

1.1 HYPOTHESIS

The current state of supercomputing is therefore such that communication-related issues significantly affect application performance. While some techniques have been developed to allow programmers to modulate overlap, these methods can either be challenging to use (OpenMP) or can present high memory demands. However, there are an array of under-explored software techniques that hold significant potential for mitigating these issues and even improving the performance of large parallel applications that are currently limited by network demands. The main goal of this work is to develop techniques that will allow application users and developers to develop frameworks that perform efficient analyses using large datasets on existing supercomputing infrastructures, without requiring users to invest significant time in manually tuning application communication, overburdening machines' memory, or rewriting large code bases.

The main hypothesis explored in this dissertation is that using carefully controlled overdecomposition combined with strategies for spreading the work of objects to cores can increase performance on existing computing infrastructure without the need to rewrite large codebases associated with parallel applications. Our (primary) proposed alternative is to reduce the total number of objects per node while simultaneously increasing the number of cores a

Figure 1.4: Different spreading configurations, showing both data decomposition and core assignment.



single object can execute on at a given time (slice). We term this process 'spreading' and demonstrate how it can be accomplished through shared memory libraries (e.g. OpenMP) used in conjunction with pure Charm++. This approach allows us to 'shape' the application execution and adapt to the configurations of current and future machines. Figure 1.4 presents a pictorial description of this idea, demonstrating both problem domain decomposition of a two-dimensional grid on the left and several possible approaches for 'spreading' the execution in time and space of the various grids across a representative single node. This approach utilizes bandwidth spread over time, both in terms of network bandwidth used for cross nodes communication as well as within node data transfer bandwidth, as well as good efficiency for the compute portion of the application's execution, thus leading to improved overall application performance

1.2 BOUNDING FACTORS

In this work, I explore a variety of communication-oriented problems. As discussed previously, computational performance has and continues to improve faster than communication performance. While computational performance has long been the focus of supercomputing, here I prioritize communication to identify novel opportunities to improve performance. I focus on a suite of applications, benchmarks, and mini-applications whose performance is limited by their communication patterns, e.g. communication-bound as opposed to computation-bound applications. For this subset of problems, advances in hardware generally do not translate to performance gains. These communication-bound codes usually fall into one or several sub-categories based on the network property that currently impedes their performant execution. These main sub-classifications include applications whose performance is slowed due to high network latency, e.g. is latency bound, and this effect is often due to communication on the critical path. Another class, bandwidth-bound applications, are slowed by a lack of available bandwidth to send information during crucial periods of the application iteration. A third category, contention-bound applications, arise not due to any particular parameter of the network but rather its general configuration contributing to congestion and thus overall application slowdown [5].

It is important to bear in mind that, although applications can generally be classified into these three categories, the categories exist only when considering a specific set of machines. A theoretical future machine with perfect ratios of latency, bandwidth, FLOPS, memory, etc. would see none of these issues. Unfortunately, when clusters are built for general purpose use, no single application is totally suited to their configuration. Thus, indentification of communication-related limitations to performance hold continuing importance.

The techniques described above serve to alleviate some of the issues of each of these subtypes of problems as well as communication-bound problems generally. They are outlined in the next section and detailed further as they are employed.

1.3 OUTLINE

We first illustrate this idea in the context of multicore nodes and inter-node bandwidth. We also present initial work on the exploration of the analogous idea of overlap of CPU and GPU execution and the importance of maximizing the use of internal bandwidth between these two devices. In the rest of this work, I discuss the following chapters in order. Chapter 2 continues with a description of the specific software tools and techniques used in this work. We first review a single technique, spreading, in depth in Chapter 3 and demonstrate the

effectiveness of our techniques in general. Chapter 4 details focused study of the communication techniques, namely spreading and its potential extensions (e.g. aggregation and simulation), as well as exploring other techniques, e.g. staggering, and prioritization. We then discuss some of our earlier work in overlaping communication and computation on a nodes with accelerators in Chapter 5. Finally, Chapter 6 describes a proposed large-scale study of current modern performance HPC applications, analyzes and classifies them. Chapter 6 futher describes and formalizes a well-known common sense theorem about the limits of performance improvement due to communication optimizations We also examine the nonlinear effects and impacts of overheads in runtimes and communication layers that helps support overturning the folk theorem. Chapter 7 concludes this analysis with a description of further extensions of this work, as well as general observations applicable to a wide swath of communication-intensive parallel applications.

CHAPTER 2: BACKGROUND

2.1 PARALLEL PROGRAMMING SYSTEMS

This work relies on a variety of programming systems and software. They are described in more detail below.

2.1.1 OpenMP

Throughout this work, I utilize OpenMP [6] to automate shared memory programming. In this use case, I place a simple set of 'pragmas' around the hot loop in the primary computational kernel. This allows for the control of the 'width' of the parallelism, as well as other factors including scheduling and collapse decisions. I leverage OpenMP instead of other shared memory models due to its overwhelming ubiquity, good support, ease of use, and non-interference with serial code.

2.1.2 MPI

For some of the distributed memory experiments, I take advantage of the Message Passing Interface (MPI) [7]. MPI was originally developed to standardize various communication layers sending messages in across nodes in machines with single core 'nodes'. Here, it is used in conjunction with OpenMP to provide two levels of differing parallelism, shared and distributed. MPI is also used to facilitate both levels simultaneously, but recent work has shown success in combining both models [8, 9].

2.1.3 Task-based Models

Alternatives to traditional MPI or MPI+X (e.g. MPI+OpenMP models), task-based programming models, including Charm++, are gaining popularity in the HPC community. Their emerging popularity stems from a variety of differences with MPI that lead to the relative strengths detailed below.

First of all, in these tasking models, the programmer is responsible for making logical divisions of work (and potentially data) instead of focusing on the direct physical mapping present in MPI. This allows the programmer to write code that matches their problem structure. Additionally, this flexibility provides several advantages. For one, the programmer is able to easily scale their code to different problem and machine sizes without altering

their fundamental algorithms and can alter grain size [10] to improve performance. This performance improvement can come in a variety of ways, including through improved load balancing characteristics and responsiveness, a key area of study in this work. Finally, task-based models allow the programmer to focus their efforts on writing serial code within tasks and then orchestrate those tasks in a broader framework by defining their dependencies on both data and other tasks; in contrast, MPI requires the programmer to intermix parallel and serial elements. Charm++, an adaptive runtime system, embodies these key principles of task-based models in a proven community framework for parallel programming.

2.2 TRACER

Another key aspect of this work is the examination of current and projected supercomputer network configurations and their possible effects on application performance. We model these supercomputer networks via TraceR [11] and its constituent components ROSS [12] and CODES [13]. The TraceR framework allows us to compare current and future runs of the same traced application on varying machine configurations. This comparison is accomplished by tracing the original application, in our case using the BigSim layer in Charm++[14] and providing those traces to TraceR in addition to machine configuration parameters. TraceR support both BigSim (Charm++) and DUMP (MPI) trace formats, which I plan to explore in the application case study, see Chapter 6.1. TraceR then simulates the performance of this application using a packet-level network simulator and simulated execution times. I installed TraceR and its dependencies on both local and remote machines via the Spack package manager [15], which ensures consistent versions across machines for both generation and simulation and allows us to run simulations at various scales. TraceR itself can be run in parallel, drastically decreasing the time required to simulate the application's execution and also allowing for larger scale experiments.

2.3 PROJECTIONS

Charm++ provides its own performance visualization tool, Projections. Users can automtically instrument Charm++ codes by enabling the correct flags at compile and run time, see the Charm++ runtime guide for detailed instructions. This does contribute to a slight performance overhead, and is not turned on during production runs. Once collected, Projections traces are used for post-mortem performance analysis and visualization of application behvaior and characteristics. These automatically generated logs can be sup-

plemented by user tagged events, called user events, which are then displayed on rendered execution timelines along the normally traced portions.

CHAPTER 3: FLEXIBLE HIERARCHICAL EXECUTION OF PARALLEL TASK LOOPS

In this work, we demonstrate the effectiveness of combining the techniques of overdecomposition and work-sharing to improve application performance. Our key insight is that tuning these two parameters in combination optimizes performance by facilitating the trade off of communication overlap and overhead. We explore this new space of potential optimization by varying both the problem size decomposition (grain size) and number of cores assigned to execute a particular task (spreading). Utilizing these two variables in concert, we can shape the execution timeline of applications in order to more smoothly inject messages on the network, improve cache performance, and decrease the overall execution time of an application. As single-node performance continues to outpace network bandwidth, ensuring smooth and continuous injection of messages into the network will continue to be of crucial importance. Our preliminary results demonstrate a greater than two-fold improvement in performance over a naive OpenMP-only baseline and a thirty percent speedup over the previously best performing implementation of the same code. The contributions of this work include the examination of the interaction of these two parameters and their potential to increase application performance.

3.1 INTRODUCTION

One of the persistent trends in modern supercomputer architectures is a significant, even dramatic, increase in computational power of an individual node without a corresponding proportionate increase in across-node communication bandwidth. One can see this trend in the decrease in the number of nodes in supercomputers combined with a substantial increase in total flop count, e.g. from 100,000 nodes of BlueGene/P, to 25,000 nodes of Blue Waters (and Titan), to 4,000+ nodes on Summit. This trend is often described by the phrase "fat nodes", and it puts increasing pressure on the network. This is further exacerbated by a common compute-communicate-compute pattern followed by many applications. Since the compute finishes fast, the long communication latencies start impacting performance negatively.

Overdecomposition has emerged as a useful technique for automatically overlapping communication and computation. Explored and popularized by systems such as Charm++ and Adaptive MPI, overdecomposition is a technique that involves partitioning the data and computation into a large number of chunks. During each iteration, or time-step, each processor schedules and executes many such chunks one after the other, depending on the availability

of data. To do this, such systems are driven by the availability of messages (and are also called task based runtimes for this reason). No individual chunk can hold the processor hostage and block it for the data it wants to receive. Instead, a user-space scheduler allows those chunks to proceed that have the data that they need. This method automatically overlaps communication with computation, and allows effective use of the network by spreading communication injection over the duration of the time-step.

However, the narrative gets more complicated due to another factor: modern nodes have tens of cores and multiple accelerators. This parallelism combined with complex memory hierarchies creates new challenges that the basic overdecomposition technique may not be able to handle. In particular, if we overdecompose with respect to cores, we make too tiny a set of chunks, leading to high overhead both in terms of memory (e.g. ghost regions) and scheduling time. Should we instead spread the work of a single chunk to all the cores? Or to a subset of them? This is the question that we address in this chapter.

We will focus on nodes with multiple CPU cores for this chapter. However, we believe that our techniques can be extended to the accelerators as well. A multicore node presents a more controllable environment for our user-space scheduling techniques for now. Accelerators with their built-in runtime schedulers present additional challenges that we plan to address in future work. Further, some of the upcoming large-scale architectures do not employ accelerators, where this work will be directly applicable.

The approach we propose and advocate for here involves flexibly combining overdecomposition with "spreading" the work of an overdecomposed chunk onto a subset of cores.

By combining the power of both task granularity and flexible resource execution, we can achieve a variety of new goals including: creating a new optimization space, mapping applications to run on a variety of hierarchical hardware platforms, improving performance via work set size reduction and the steady injection of messages on an otherwise overtaxed interconnect network. In this work, Charm++ provides overdecomposition parameter space and OpenMP allows us to spread tasks amongst various threads and cores. We tested our approach on a variety of platforms and configurations in order to validate its performance and utility. One caveat emerged via this testing: as with most forms of parallelism, the application must have adequate work to justify the overhead of parallelism. Under these ideal conditions, we achieve a four-fold speedup over a naive, but commonly used, baseline in the best-case and a 30% speedup over the current state of the art. These results hold for up to 256 nodes of Summit under weak scaling.

In the next section, we review the supporting technologies. We continue in Section 3.3 to further describe our approach. Section 3.4 describes our evaluation methodology and their implications. We then conduct a survey of similar and related approaches in the next section

before highlighting future directions and conclusion in Sections 3.6 and 3.7.

3.2 BACKGROUND

We rely on several existing technologies to implement our hierarchical approach to the flexible execution of parallel tasks. These include parallel programming libraries such as OpenMP and Charm++, discussed below, as well as supporting tools for analysis and simulation.

3.2.1 OpenMP

OpenMP [6] is used generally (and here) to divide potentially parallel regions of code into tasks that can be then executed on OpenMP-spawned worker threads. The OpenMP library provides many useful utilities, including scheduling and data privatization directives, as well as commands to limit the number of executing threads. In this work, we utilize OpenMP to create and execute the thread-level parallelism of our main compute kernel. This is what we refer to as the spreading factor, which is described in more detail in Section 3.3.1.

3.2.2 Charm++

Charm++ [16, 17] is a parallel programming framework for both distributed and shared memory machines. It divides a problem space into work and data units, named chares, and then schedules their execution on the system resources as the data becomes available through asynchronous messages. Charm++ provides many tools to assist program analysis and simulation, namely Projections, discussed below.

3.2.3 Overdecomposition

One of the central tenets of programming in Charm++ is the idea of overdecomposition, i.e. decomposing your problem domain into more tasks than the number of hardware cores or threads[18]. In a typical problem, the user thinks of each core as a separate stream of execution responsible for a subset of the data that depends on both the problem and machine size. Using Charm++, a programmer is free to decompomse their problem domain into as many logical units as makes sense given the structure of their data and the overhead associated with separate logical units. We refer to the size of this work and data unit (i.e.

chare) as the grain size of the application, which is now a user-, and potentially runtime-, configurable parameter.

3.3 METHODS

Traditional MPI-focused models typically partition problems across a set of ranks on a single compute node. While this provides the programmer some flexibility in determining an appropriate grain size and decomposition for the problem, it often forces them to program directly to the hardware. Overdecomposition represents a conceptual pivot from the hardware-centric programming model towards an emphasis on creating logical work units, freeing the programmer to divide their work in the most logical manner.

In this work, we propose a possible extension of this approach. This single axis of optimization, grain size, can be projected onto two dimensions with the addition of a second parameter, the spreading factor. By writing programs that are expressed using their logical work granularity and then flexibly running those tasks on one or more cooperating threads, we can shape the execution timeline of an application to improve performance. This improvement is achieved specifically by spreading the injection of messages on to the network smoothly over time as opposed to concentrating them at the end of a time step. As we show in Section 3.4, this approach can have the added benefit of improving cache performance through smaller work set sizes for each core.

3.3.1 Spreading

For our initial implementation, we leveraged the capabilities of the OpenMP runtime to generate work chunks from our tasks. In order to control our process-to-thread ratio, we run Charm++ in non-SMP mode, which spawns a single process per scheduler. We map these schedulers to separate cores that are spaced by the number of threads that we plan to generate. During the execution of each run, we pass the number of threads per process to spawn as a command line parameter to the application. This allows us to test different process-to-thread ratios without recompiling the code and within a single batch job execution. We then set the number of OpenMP threads to execute programmatically using `omp_set_num_threads(int)`. While this is an upper bound, we have found that given correct mapping of processes and threads to cores (below) we always generate the requested number of threads per scheduler. In general, we choose our process and thread numbers such that: $num\_procs * num\_thds = num\_cores$, where num_cores is the total number of hardware cores visible to the executing program, optionally including hyperthreading, num_procs is the

number of processes, i.e. schedulers or PEs, spawned by CHARM++ directly, and *num_thds* corresponds to the number of threads passed as a runtime parameter to the program and set as the number of OpenMP threads to be spawned in the parallel section. As execution of our application progresses from initialization to simulation, we annotate our intensive compute kernels with OpenMP pragmas in order to generate work chunks. In the future, we also plan to integrate this functionality with CHARM++'s loop scheduling library, CkLoop[19]. We also plan to test this with applications that exhibit variability in their per-iteration workloads, both within and across iterations. This approach serves as a hierarchical alternative to Charm SMP mode, which spawns a single process and then a thread per core, such that the total is equal to the total number of hardware cores. Note that a further optimization is possible with additional implementation work: one could have a process covering (say) an entire NUMA domain, but have multiple schedulers within them covering a subset of cores. This will allow read-only sharing and more efficient communication via shared memory, at the cost of more elaborate CPU affinity specifications for OpenMP loops, possibly using teams.

Our key insight is that we can provide more computational resources to a compute kernel to decrease its runtime and consequently inject messages on to the network quicker and more smoothly. This approach does involve overhead in spreading work across multiple threads on separate cores; however, assuming the grain size is sufficiently large, we have shown that this technique can significantly improve performance.

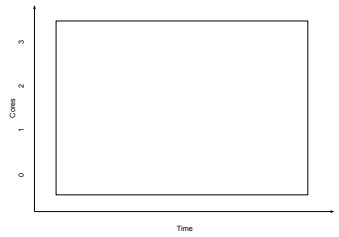
3.3.2 Affinity

In order to ensure our threads spawned via OpenMP do not compete for the same resources (cores) on a node, we utilize a mixture of affinity libraries to pin processes and threads to cores. Inside each process we create a `CPU_SET` for that scheduler's (process) exclusive use and we use Linux's `sched_setaffinity` command to bind any threads spawned by the process to this domain. On Summit, we also rely on the `jsrun` job launcher to correctly allocate and place our processes and threads in coordination with our application. Correctly setting this affinity mask is vitally important to performance and is potentially the reason we see decreased performance when OpenMP regions are mapped across NUMA domains (these results are discussed in Section 3.4).

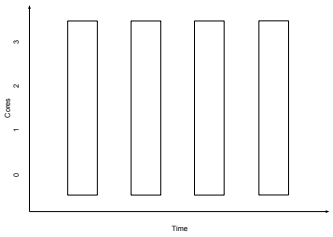
3.3.3 Combined with Overdecomposition

While our approach to spreading work across multiple cores within a process is not unique,

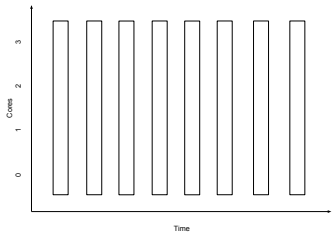
Figure 3.1: Two axes of optimization: overdecomposition (number of objects) and spreading (number of threads per object).



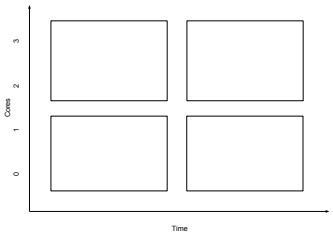
(a) One object with four threads (**OpenMP**)



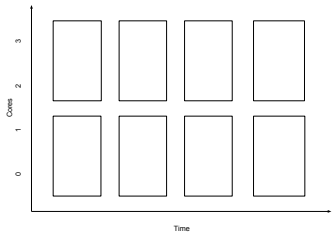
(b) Four objects with four threads each



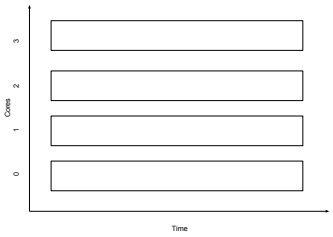
(c) Eight objects with four threads each



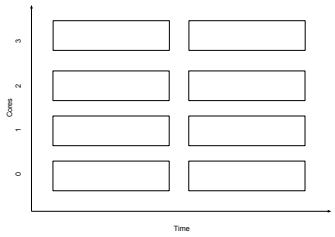
(d) Four objects with two threads each



(e) Eight objects with two threads each



(f) Four objects with one thread each (**MPI**)



(g) Eight objects with one thread each (**Charm++**)

we instead derive our increased utility by combining this technique with the concept of overdecomposition embedded in Charm++. Through this novel combination, we chart a previously unexplored configuration space for parallel applications. The real power comes from the flexibility inherent in shaping execution timelines and message injections into the network. Succinctly, we now have the choice to trade-off extra overhead inherent in distributing shared work amongst multiple processors with increased overlap of communication, i.e. message sends, and computation. Furthermore, our approach is such that a programmer (or the runtime system) can reconfigure the application's behavior at runtime, adapting the execution model over various machine specifications and problem sizes. We now have the

opportunity to selectively spread work amongst various hardware components, e.g. sockets, NUMA domains, etc.

An example of how these two factors interact and how current programming paradigms map on to optimization space is presented in Figure 3.1. Although we draw this figure with only three values on each axis, it extends indefinitely in either direction but we omit those details for clarity. As we move from left to right on the X axis, we increase the total number of objects and proportionally decrease the object size or, in other words, increase overdecomposition. This leads to a smoother and less bursty injection of messages into the network, as typified by Charm++'s approach to overdecomposition, at the cost of increased overhead and complexity. Moving in the opposite direction exhibits the reverse tradeoff, with decreased overhead for bursty communication due to bulk synchrony. Moving from top to bottom along the Y axis represents an increase in the number of threads assigned to execute each object, corresponding to an increase in the spreading factor. This axis represents a similar tradeoff between lower overhead using a single thread and smaller working set sizes and more consistent message injection rates (with correspondingly higher overhead). Our contribution is to begin enumerating this new optimization space, where overdecomposition and spreading interact to form a new Pareto frontier.

3.4 EVALUATION

In order to demonstrate the validity of our approach, we implemented our combination of spreading and overdecomposition in a common parallel application, iterative Jacobi, as described below. We evaluated the performance of our approach in a variety of configurations and on a variety of platforms, primarily focusing on the Summit system.

3.4.1 Machine

We ran our initial experiments on the Bridges system at the Pittsburgh Supercomputing Center and the Stampede2 system at the Texas Advanced Computing Center [20] but did not see any early promising results and thus shifted our entire focus to Summit. A single node of Stampede2's KNL partition, where we ran our experiments, contains a 68-core Intel Xeon Phi 250 Knight's Landing processor. These cores can use one-, two-, or four-way hyperthreading, which we tested in our results below. Each node also contains only 96 GB of DDR4 RAM plus 16 GB of configurable high-speed MCDRAM. While we did not see a difference in performance for various MCDRAM configuration modes, we suspect that the small amount of main memory was a limiting factor in our ability to demonstrate improved

speedups, as it did not provide enough data to offset the overhead of spreading out the work across cores. Previous work [21] has shown a similar non-effect on the performance of a variety of Charm++ applications and benchmarks. This supports our broad hypothesis that our approach will be move valuable as nodes become faster, larger, and 'deeper', i.e. increase their FLOPs/bandwidth ratio.

We performed our experiments primarily on the Summit machine at Oak Ridge National Laboratory. A single node on Summit consits of two 22-core IBM POWER9 CPUS [22] and 512 GB of DDR4 memory. It also contains six NVIDIA Volta V100 GPU and an additional 96 GB of High Bandwidth Memory (HBM), which we did not utilize for these experiments and leave as future work.

3.4.2 Application

Our particular focus is on iterative regular applications that can be efficiently decomposed into fine-grained work units. Therefore, we chose to focus our efforts on a well-studied application, iterative Jacobi. Our particular experiments were run using a two-dimensional decomposition and simulation space, but our methods can easily be extended to other decomposition and simulation dimensions.

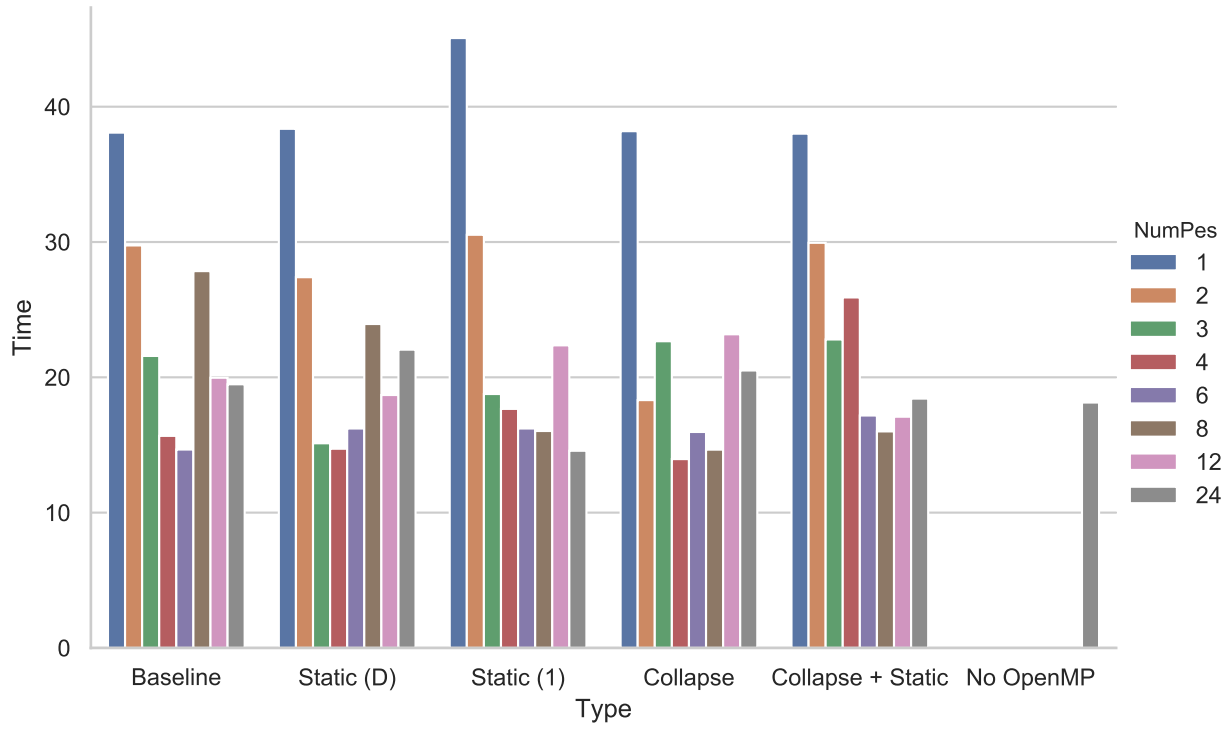
3.4.3 Experiments

Each experiment contains five runs of the same application in various configurations, e.g. with one, two, or more OpenMP threads per process and without OpenMP. These results are presented as an average of these runs, collected in a single job on the same set of nodes and with the same configuration parameters, e.g. affinity. We compiled each code with full optimization, i.e. `-O3`, using IBM XL C/C++ compiler 16.1. When building Charm++, we utilized non-SMP mode to isolate each scheduler in a single process in control of a separate pool of threads. In the future we plan to move to a model where we spawn a single league of threads with one team dedicated to each scheduler [6]. We built Charm++ using the pami[23] layer for communication due to its native bindings, high-performance, and relatively stability and enabling all performance related optimizations, i.e. `--with-production`, including disabling shared libraries [24].

On each machine and node configuration, we performed calculations over a two-dimensional grid, roughly half the size of the node's main memory. We then further decomposed this grid into a variable number of chares. We left approximately 10% of the total memory unallocated as overhead for the runtime and other non-grid variables. The reported time is

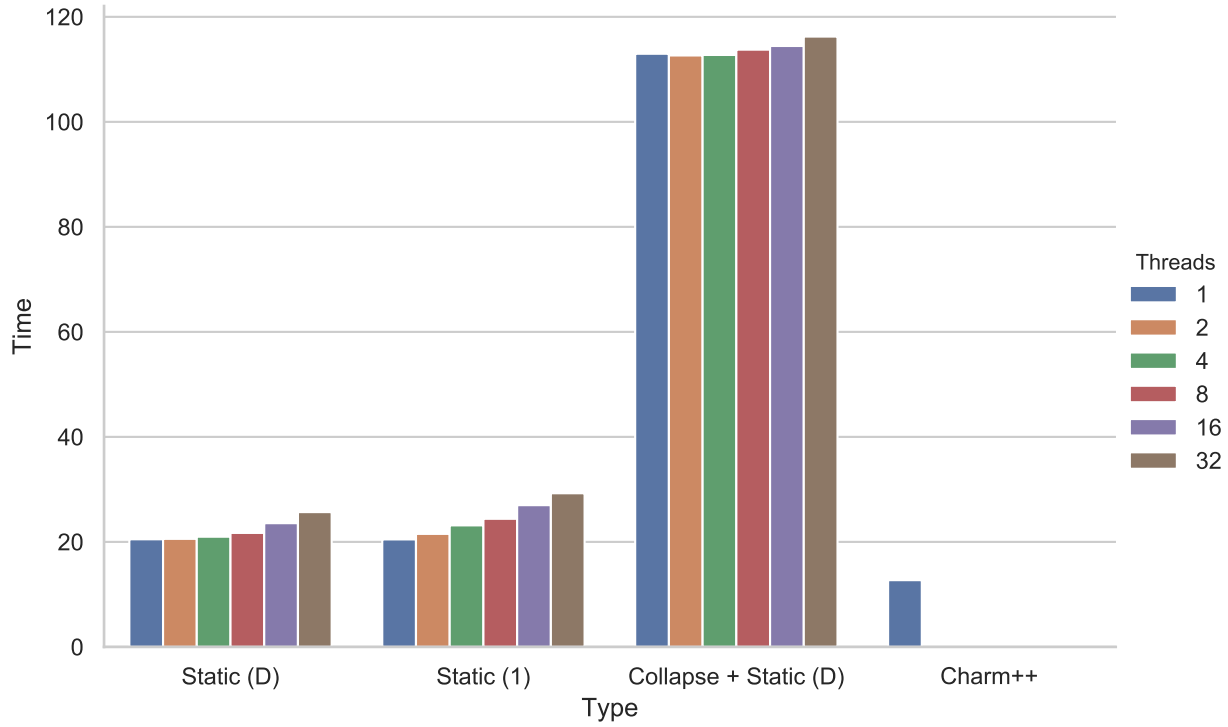
for the total execution of the application's main logic and excludes startup overhead. Timing data was gathered using Charm++'s high-precision timers, which utilize an appropriate high-precision timer based on the compute platform, placed immediately before and after execution of the main compute loop. Our baseline in most experiments, labeled Charm++ represents pure Charm++ code without any OpenMP based loop spreading. We also compare our approach to a naive Charm++ + OpenMP implementation that uses Charm++ for message passing and OpenMP for shared memory parallelization. As mentioned in Section 3.3.1, we always ensure that the total number of processes multiplied by threads is equal to the total number of cores on a node.

Figure 3.2: **Bridges** - A comparative study of the effects of various OpenMP schedules on our flexible execution strategy compared to the pure Charm++ baseline. The number in parenthesis indicates the chunk size, where D stands for default. The bar color is coded to the number of threads launched per process (spreading factor).



We tested our initial implementation on a single node of Bridges with a variety of OpenMP schedules to determine the best initial configuration for further study. For this experiment, we used a grid size of 89424×89424 doubles and a block size of 7452 in both the X and Y dimensions. In Figure 3.2, we can see no clear combination of parameters consistently results in the optimal execution time, but we achieve a speedup of up to 1.3x over our Charm++ baseline (the lone Charm++ entry in this figure).

Figure 3.3: **Stampede2** - comparative study of the effects of various OpenMP schedules on our flexible execution strategy compared to the pure CHARM++ baseline. The number in parenthesis indicates the chunk size, where D stands for default. The bar color is coded to the number of threads launched per process (spreading factor).



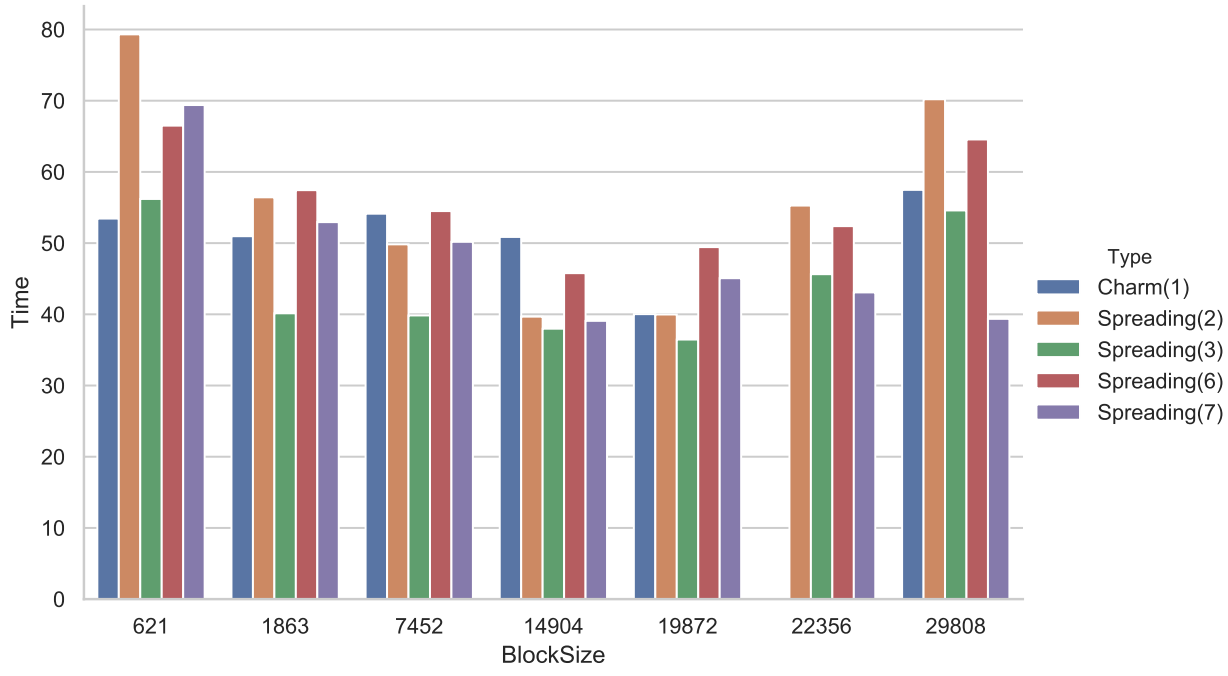
We also tested our approach on a single node of Stampede2 using the same OpenMP schedule parameters as on Bridges. We only utilize 64 of the 68 available cores in all of our experiments, baseline and otherwise, in order to equally divide our grid and reserve a set of cores for OS operations in order to reduce noise. Since Stampede2 is a Knight's Landing chip, we have the ability to allocate its HBM in a variety of configurations. We experimented with both the cache (default) and flat modes and saw little to no variation in performance, thus the results have been omitted. We utilized cache in all of our reported Stampede2 experiments. As you can see in Figure 4.5, we did not see much variation in performance with various schedules, aside from the obvious slowdown when using the `collapse` OpenMP pragma.

After our single node experiments on Bridges and Stampede2, we concluded that the static chunk size scheduling had the most promise. Further results examined only the two different static size options and omit the collapse directive which had a dramatic negative impact on performance. In addition to the impact of OpenMP scheduling on our approach, we also examined the effects of hyperthreading on distributed execution. Increasing the number

of hardware threads available to our application did not improve performance, leading to similar results as Figure 4.5, and they have thus been omitted for space.

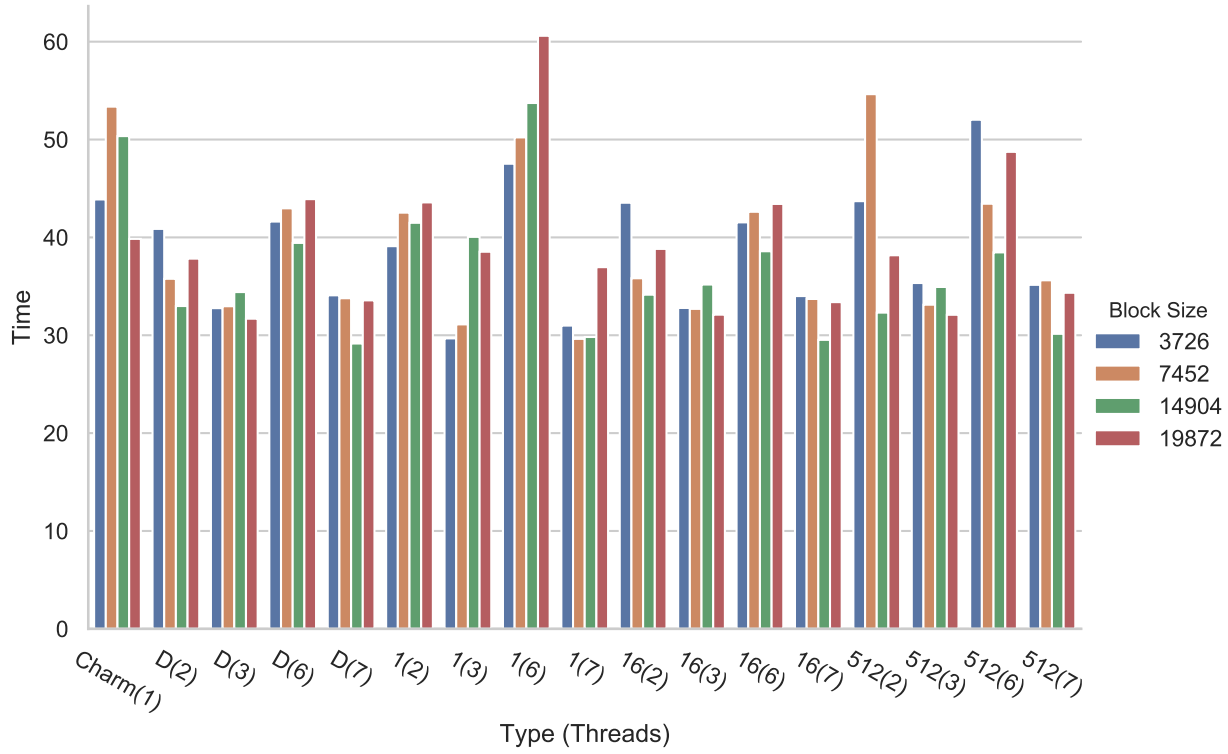
We then turned our attention to running our application on Summit across multiple nodes. Our first experiment tested a variety of work block, i.e. (over) decomposition, sizes for our Charm++ baseline as well as our implementation, labeled Spreading in this and future figures. We also indicate the number of threads we utilize per process in the legend. This experiment was run on four nodes of Summit, with a per-node block size of 178848^2 doubles. We tested a wider number of both block sizes and thread counts for our spreading factor. The results of those experiments are presented in Figure 3.4. We see a speedup of up to 1.46x over our Charm++ baseline at each block size and 1.1x speedup for the best performing configurations of both Charm++ and our implementation overall. Of particular note is the ideal block size for differing spreading factors. In the next set of experiments we include block sizes of 3726, 7452, 14904, and 19872 where possible to ensure each configuration is optimal.

Figure 3.4: **Block Size** - A series of experiments examining the effects of block size and loop schedules have on execution performance on four nodes of Summit. The bar color is coded to the number of threads launched per process (spreading factor).



In addition to examining a variety of block sizes on Summit, we also tested a larger combination of potential chunk size for the static schedule we examined earlier. We also reexamined the collapse pragma but, again, observed worse performance overall with every

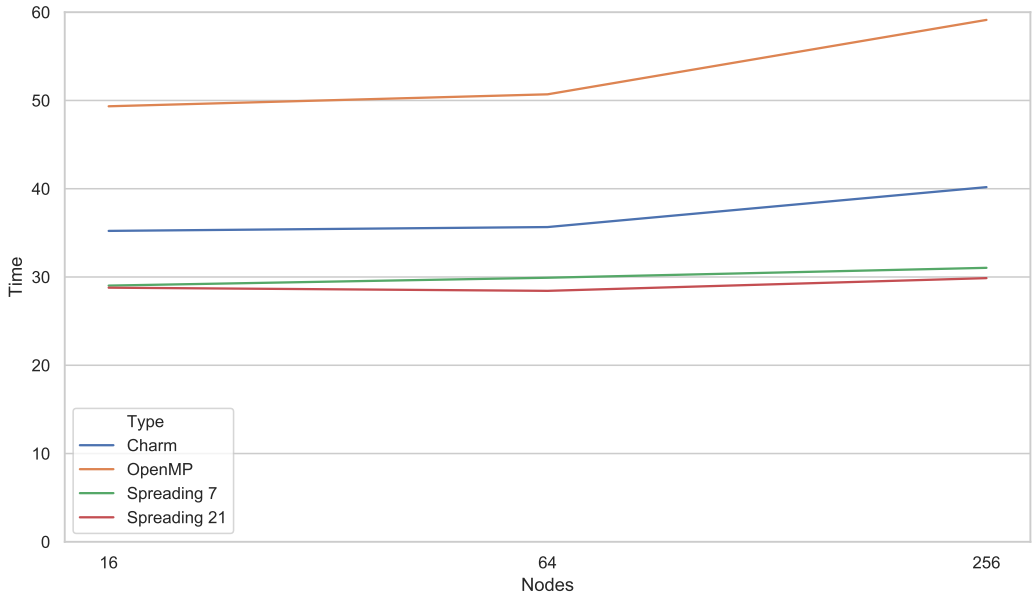
Figure 3.5: **Pragmas** - A series of experiments examining the effects of block size and loop schedules have on execution performance on four nodes of Summit. The bar color is coded to the block size. The number in paraenthesis indicates the number of threads launched per process (spreading factor). The letters and numbers indicate the chunk size, where D stands for default.



combination of static schedules. We tested these new schedules with a wider variety of thread counts and the aforementioned block sizes that achieved the range of best performance. As we can see in Figure 3.5, we had previously overlooked a faster combination for this system, namely a chunk size of 1 and thread count of 7, for a variety of block sizes. Using these configuration we see a speedup of up to 1.8x over the Charm++ baseline at the same block sizes and 1.35x using the best performing configuration for both.

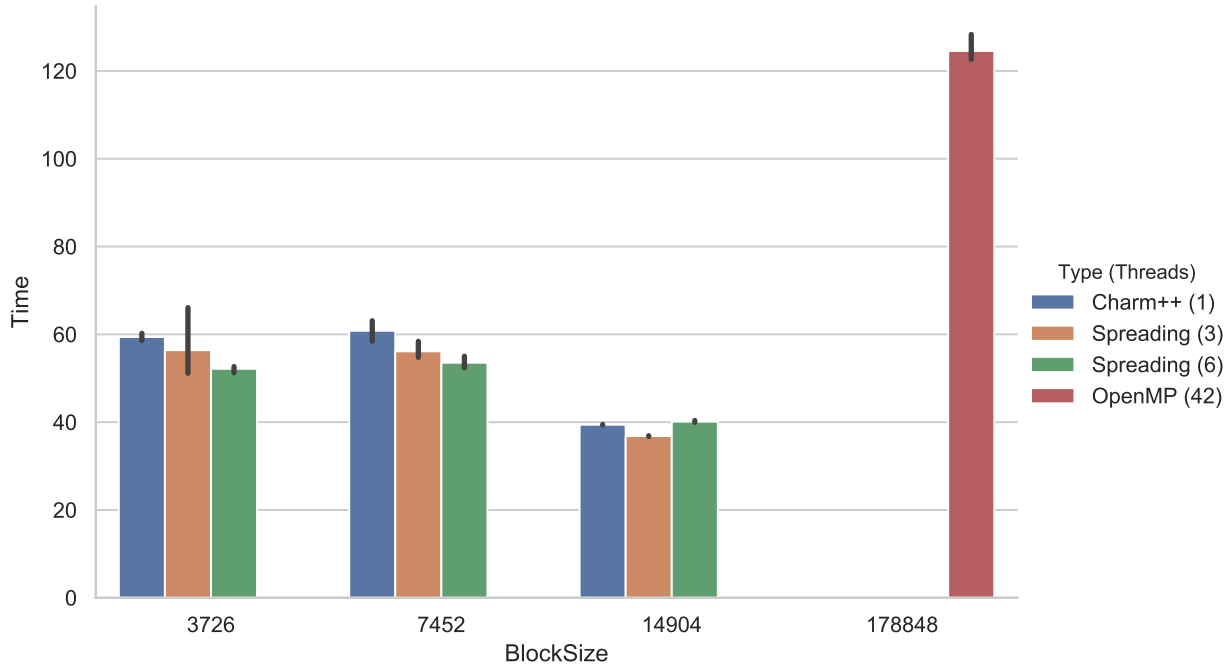
In order to determine how much of our performance gain is due to cache effects, we modified our initial code to remove the send path, and ran the resulting executable on only a single node. This setup allows us to observe effects of caching and other non-communication related performance. In Figure 3.7 we see speedup of 0.98 to 1.14x depending on the block size. Therefore, the rest of our speedup is attributable to network effects. We present further evidence of this below. Additionally, we collected PAPI L1 cache counters for the send and non-send versions as we observed the cache miss ratio go from 11.41% to only 9.73% in the best case (3 threads). [25]

Figure 3.6: Weak scaling results up to 256 nodes of Summit. The color represents the spreading factor (Charm only, OpenMP only, 7, and 21 threads. Same problem size as before, filling the majority of the node's memory and scaling with increased number of nodes.



Finally, to demonstrate the impact our approach has on spreading previously bursty communication smoothly over time, we utilize Charm++'s builtin tracing library, Projections, to gather and visualize performance data [26]. The graphs below show the number of bytes received across the network during the lifetime of the application for our three configurations: naive OpenMP baseline, Charm++ baseline without spreading, and Charm++ with OpenMP spreading. We have matched the X and Y axes scales, although the time values are slightly shifted due to different process startup protocols involved. As we can see from the figures, our OpenMP baseline has a very pronounced burst of communication at the end of each timestep (Figure 3.8). The Charm++ baseline improves upon this implementation by smoothing out the communication over time but still suffers very high peak rates (Figure 3.9). Our implementation (Figure 3.10) shows a very smooth overall use of the network, supporting our original hypothesis that smooth network injection is crucial to performance. In all three cases, we chose the block size and thread count that lead to the optimal performance on four nodes.

Figure 3.7: Single node cache results on one node of Summit. Color corresponds to parallelization and the number of threads (spreading factor) is indicated in parenthesis where appropriate. Same fixed problem size as other experiments.



3.5 RELATED WORK

There have been several projects combining Charm++ and OpenMP. These include Bak et. al's work to integrate OpenMP and Charm++ runtimes primarily for load balance[27]. Our work is complementary as they focus on tight integration for unblanaced workloads whereas we are concerned with rapid execution of balanced loops across cores for improved communication and cache performance.

There has also been some work in describing the importance of spreading an application's message injection over time to ease the impact of network performance. In particular, Preissl et al [28] mention the benefits of spreading communication over time to reduce network effects. However, their approach is focused on the traditional MPI+OpenMP model, or in their case PGAS+OpenMP, combined with one-sided communication and they do not take overdecomposition into account. There has also been a wealth of research in communication-computation overlap, [29, 30, 31, 32] among many others, but we distinguish our approach by focusing on smoothing.

Figure 3.8: OpenMP Baseline - The above results are for four nodes of Summit.

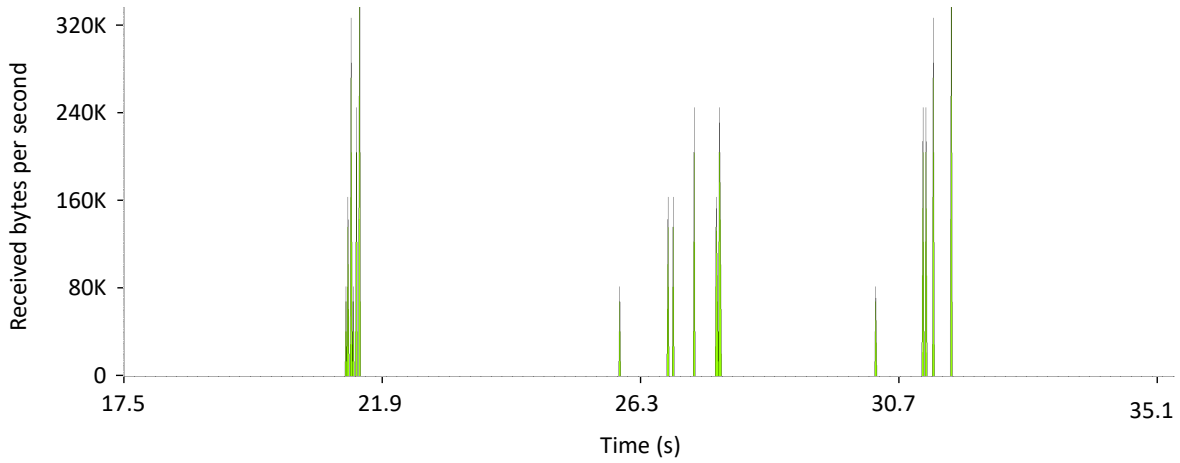
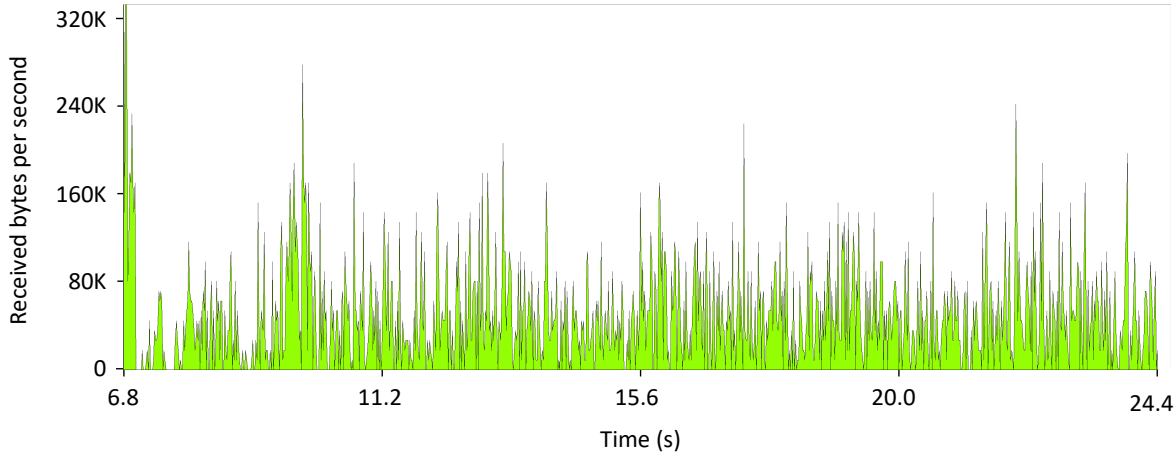


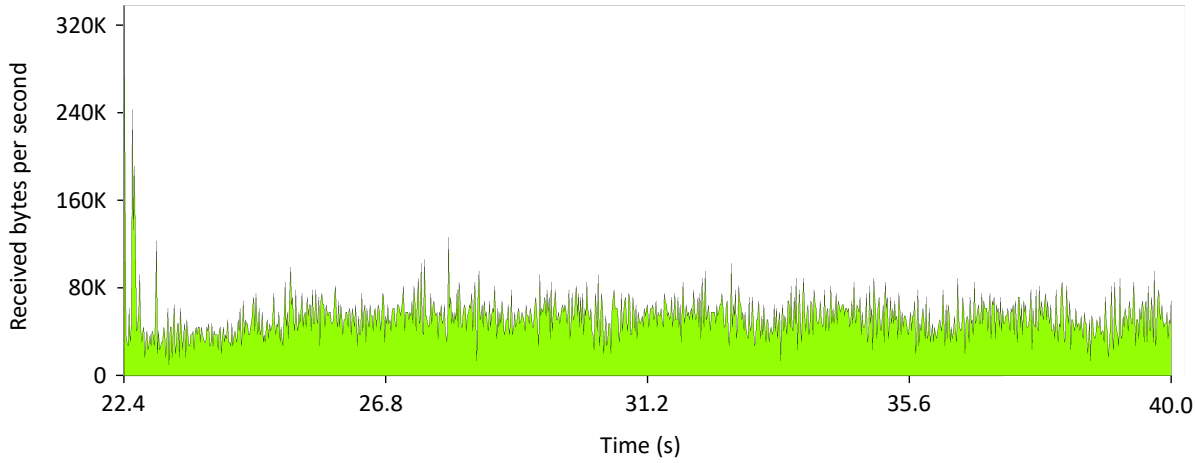
Figure 3.9: Charm Baseline - The above results are for four nodes of Summit.



3.6 FUTURE WORK

While we have begun to lay the groundwork for a flexible execution strategy that combines overdecomposition and loop level work-sharing (i.e. spreading), there are still many interesting opportunities to expand this work. One direction we intend to explore next is expanding our approach to function in coordination with GPU work. There is already an 'impedance mismatch' between the grain size of a singleton chare and this approach may serve as a possible solution by enabling bulkier chares on both the CPU and GPU. Similar approaches have been explored previously [33]. Another interesting area of exploration is tighter integration with the Charm++ runtime. There are already two loop-level work-sharing libraries available: OpenMP[27] and CkLoop[19]. We already utilize OpenMP to parallelize our criti-

Figure 3.10: Spreading - The above results are for four nodes of Summit.



cal loops and are currently exploring how to ensure OpenMP thread teams are spawned and available to execute work during critical compute kernels. As mentioned in 3.3.1, we are also working on implementing a deeper hierarchy that enables multiple schedulers to cooperate inside a single process while still directing disjoint sets of OpenMP threads. Charm++'s task level language, CkLoop, serves a similar approach and merits further investigation as it allows more control and flexibility. In either case, we plan to spawn tasks using one of the two libraries and then dynamically control the spreading factor throughout the execution. In coordination with these activities, we plan to further evaluate our approach with more complex applications. Once our runtime integration is complete this would allow us to easily parallelize any existing Charm++ applications, e.g. NAMD[34], OpenAtom [35], or ChaNGa[36], without explicitly spawning threads and directing loop level parallelism. Concurrently, we can gather performance traces for these applications and project their performance on future network constrained machines using BigSim [14] and TraceR-CODES[37, 38], further validating our hypothesis.

3.7 CONCLUSION

In this work, we have shown that by combining existing high-performance parallel libraries, Charm++ and OpenMP, and leveraging their unique styles of parallelism in a complementary way, we can improve an application's performance over what either approach can do individually. In the future, tigther integration between these two sources of parallelism should make this aporoach more widely applicable and even faster. As it stands now, we are able to achieve, a four-fold speedup over a worst-case baseline and greater than thirty

percent speedup over the current best-case performing implementation for a given grid size, up to a scale of 256 nodes of the Summit supercomputer. This improved performance comes from a variety of sources including better cache utilization and steadier injection of messages onto the network throughout a time step. With the continuted imbalance and increased ratio between computational power (FLOPs) and connectivity (bandwidth) between nodes, this work has and will continue to become more important. Furthermore, this approach 'unlocks'a new area of potential optimization: the tradeoff between grain size and loop-level parallelism. This new degree of freedom enables codes to be better mapped to differing hardware topologies both current and future.

CHAPTER 4: TECHNIQUES FOR IMPROVING APPLICATION COMMUNICATION PERFORMANCE

4.1 INTRODUCTION

This chapter focuses on a common application pattern: stencil computation. Stencil computation divides data into an n-dimensional grid, commonly two, three, or four dimensions corresponding to the physical world, and performs a constant amount of calculations per grid element, depending the type of experiment being conducted and the driving algorithm. Common examples include Jacobi, Gauss-Seidel, and specific applications such as MILC [39], which computes a four-dimensional stencil. These calculations involve data from neighboring grid points, e.g. temperature, which must be communicated between steps or iterations. We refer to these transmitted elements as *ghosts* throughout the text. After receiving the ghost elements and performing the corresponding calculation, each element updates its value in the global grid and informs its neighbors of its updated value, which they in turn use for their own calculation in the following iteration.

4.2 MOTIVATION

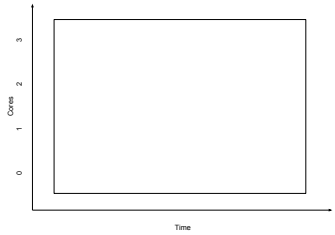
Due to the widespread use of this parallel programming paradigm, optimizing the communication performance of stencil is an important problem. In addition to being an important pattern, stencil computations have several other interesting features for the purposes of the current analysis. Firstly, stencil codes are iterative, regular applications that generate a predictable and fixed amount of work and communication per step. The work and communication can be directly calculated from and manipulated by the input and by parameters such as grain size and the dimensions of decomposition. These parameters can therefore be tuned systematically to study their effects in a controlled way in conjunction with other proposed optimizations. Secondly, stencil codes are well studied applications, which give us good baselines and metrics to compare against in our optimization efforts. Finally, iterative grid-based nearest neighbor calculations, i.e. stencil codes, can range in their overlap of communication and computation from no overlap in the naive case to perfect overlap in the optimized case. This range of communication behaviors, when combined with the predictability and regularity, allows us to assess the effects of our optimization at various levels.

4.3 TECHNIQUES

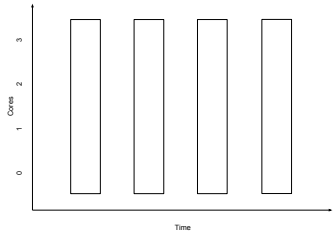
We now describe our three primary techniques to alleviate communication performance issues observed in the stencil application.

4.3.1 Technique 1: Spreading

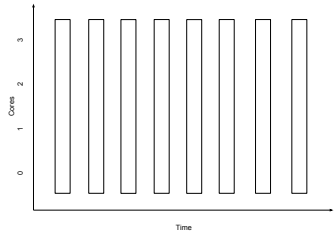
Figure 4.1: Two axes of optimization: overdecomposition (chares) and spreading (OpenMP).



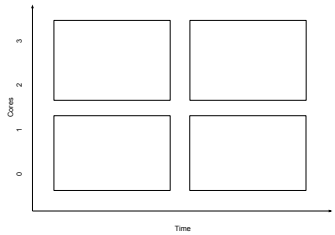
(a) No overdecomposition and 4 x spreading



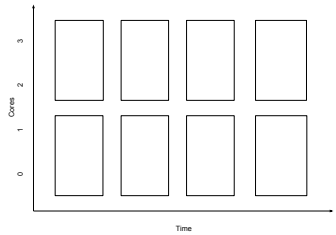
(b) 1 x overdecomposition and 4 x spreading



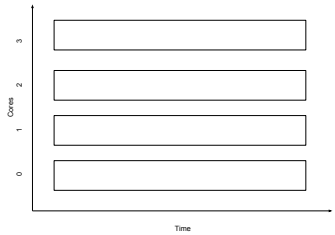
(c) 2 x overdecomposition and 4 x spreading



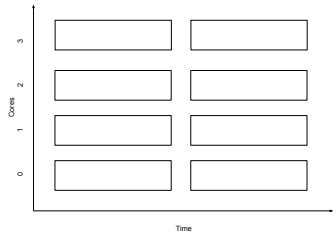
(d) 1 x overdecomposition and 2 x spreading



(e) 2 x overdecomposition and 2 x spreading



(f) 1 x overdecomposition and 1 x spreading



(g) 2 x overdecomposition and 1 x spreading

General Idea Normally in a CHARM++ program, a single entry method is run on a single core. The calculation of individual sub-grids of the stencil computation can be sped up by *spreading* the calculation across several adjacent cores using a shared memory programming

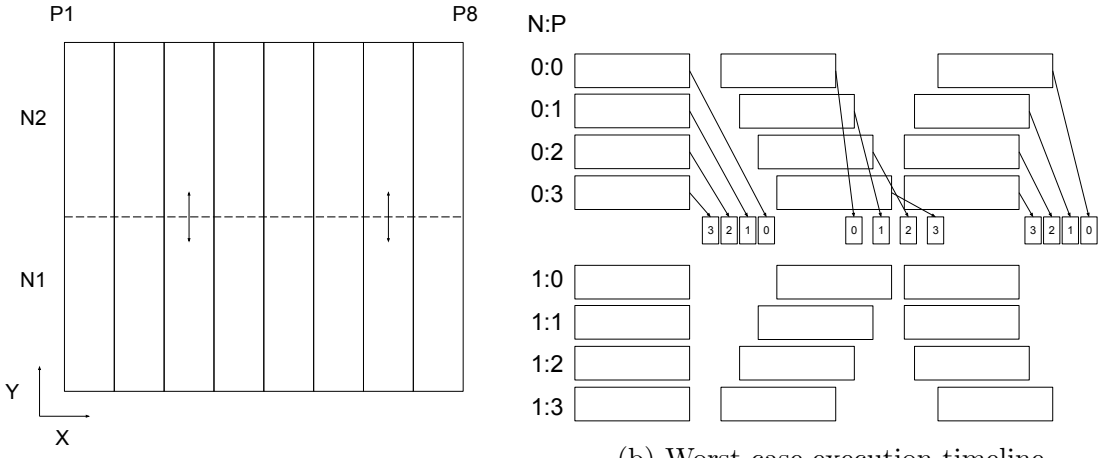
paradigm, such as OpenMP. This enables the calculation to maintain a larger grain size and hide possible overheads as well as prevent the network from being constantly flooded with many small messages.

Referring to Figure 4.1, replicated here from earlier, we can imagine the potential for execution on two axes. One axis, in this case the x axis, represents the amount of overdecomposition present in the problem. The other, i.e. the y axis, represents the amount of *spreading* or usage of OpenMP. The figure illustrates a simple example with four cores and up to two times overdecomposition, but this model can be extended further in both directions.

We explored this technique in detail in Chapter 3 and include it here due to several extensions and continuing work we've performed in later sections, as well as a reference to other techniques.

4.3.2 Technique 2: Staggering (non-sync/timestepped)

Figure 4.2: A possible problem decomposition and mapping (L) and worst-case scenario (R).



(a) Decomposition and Mapping

(b) Worst-case execution timeline

General Idea There are situations where randomized ordering e.g. non-ordering of messages can create significant performance slow downs, especially when compared with the minimal overhead required to synchronize message execution.

Let us now consider a situation in which random queuing would create a significant decrease in performance. Imagine two physical nodes connected by one link. Every rank on node 0 must communicate with its corresponding rank on node 1.

If every rank completes its execute step at the same time, then they will all try to utilize

the same network link to send to their neighbor on the other nodes, and vice versa. This message delivery order is not guaranteed, which means the first rank to send may reach its neighbor last. This random staggering continues throughout each iteration, in the worst case having the first message arrive last and vice versa. This could lead to a slowdown, based on the calculations below.

$$t = t_{comm} + t_{comp} = \alpha * N + \beta * M + t_{comp} \tag{4.1}$$

Where N is the number of other message that arrive before, in the range zero to num workers/chares/PEs/etc and the rest are their normal values, e.g. α is latency, β is bandwidth, M is message size, and t_{comp} is time for computation.

However, this is the worst possible (degenerate/perverse) case. In general, the delay would be $N/2$ on average. With our staggering optimization, the execution time becomes:

$$t = t_{comm} + t_{comp} = \alpha + \beta * M + t_{comp} \tag{4.2}$$

since messages no longer delay each other.

4.3.3 Technique 3: Prioritization

So far, it is clear that Jacobi/stencil applications respond to the receipt of ghost/remote data in a first-come-first-served manner using first-in-first-out (FIFO) queues. However, this presents an obvious problem when we extend our application to run on more than a single node as data is now at various 'distances away'. We therefore must appropriately respond to processing remote data by prioritizing it above more easily reached local data. This includes both the send, receive, and calculation phases as our application can be communication-bound for especially large networks or poor topology configurations, which have much lower performance limits than local sends.

4.4 EXPERIMENTS

To demonstrate the effectiveness of these techniques, we performed a variety of experiments to monitor their improvement in performance. These experiments were run on a variety of supercomputing installations to demonstrate their wide applicability. These machines include: Bridges at the Pittsburgh Supercomputing Center (PSC) [40], Comet at the San Diego Supercomputing Center (SDSC) [41], Blue Waters at the National Center for Supercomputing Applications (NCSA) [42], Stampede2 at the Texas Advanced Computing

Center (TACC) [43], and Summit at the Oak Ridge National Laboratory (ORNL) [44]. The respective configurations of each machine can be found at the linked resources and will be mentioned as appropriate below.

The first set of experiments are direct runs, i.e. non-simulated, of the stencil application, referred to as Jacobi 2D for the type of calculation performed and its decomposition. We perform this experiment both on a single node, where communication occurs but is less constrained by available bandwidth and latency, and across several nodes of a machine. Currently, technique one, e.g. spreading, is applied to this calculation. In the baseline version, a two-dimensional grid is divided amongst various logical workers, or chares, that are responsible for the corresponding calculation and data management and messaging. Those logical workers are then assigned to physical cores by the Charm++ runtime. In each iteration they perform a local calculation and then transmit their updated border regions to their neighbors, which require it for local calculations. This computation is a simple average but can be tweaked to represent more intense calculations, e.g. gravity. In order to 'spread' this work, we parallelize the inner calculation loop of each worker using OpenMP [6]. We executed several experiments to determine the optimal execution parameters, see Figure 4.5. Following this, we now space our workers out amongst the cores based on the number of sub-worker threads they will spawn. For instance, if they will use two threads to perform their inner calculations, then we only spawn half as many processes as normal and assign them to alternating cores, leaving every other spare for a worker thread. We then compare our optimized version against two baselines. One baseline, the 'true' baseline, is the naive, MPI+OpenMP-like implementation that creates a single rank, e.g. MPI process, and divides the work amongst every available core using OpenMP threads. The other baseline, is the optimized Charm++ that uses overdecomposition to overlap communication and computation but does not 'spread' out the work of chares across cores.

Experiments are in progress to determine and extrapolate the performance of stencil applications using these various techniques on future machines and networks. To accomplish this work, we are collecting traces of application runs at scale using Charm++'s BigSim [14] framework. These traces include both execution time for tasks and communication time for messages. Moving forward, we will replay these traces using TraceR [11] to estimate the impact of worsening bandwidth to FLOPs ratios. We are also using these experiments as an opportunity to validate BigSim and TraceR against empirical runs.

4.5 RESULTS AND EVALUATION

The results of these experiments are found in the figures below (Figures 4.3-4.8). Figure

4.3 shows the execution time for the spreading technique described above when applied the Jacobi stencil program and run on a single node of Bridges. Utilizing a variety of OpenMP tuning parameters, e.g. scheduling and loop collapse, revealed the best overall configurtion to be static scheduling using the default chunk size and no loop collapse. This configuration is reused in later experiments. Comparing against the no OpenMP version, which is identical except it is not compiled with the OpenMP flags, exhibited one of benefits of OpenMP: preserving the original serial code. This experiment was run with a variety of OpenMP threads to represent the amount of spreading. With a larger number of threads, the number of processes/PEs is reduced correspondingly such that: $numCores = numPEs * numThreads$. The best performance is found using six threads in the baseline case. This experiment was also repeated across four nodes of Bridges in Figure 4.4, and similar results were observed. In this instance, the best configuration parameters are utilized as determined above, and a limited number of thread configurations were also run based on our above results. Interestingly, twelve threads was observed to be a slightly better number to run in this mode, potentially owing to the increase in cross-node communication and spanning a single CPU in a process. Overall, this approach achieved a 30% speedup over the no OpenMP baseline on a single node and a 33% speedup when run across four nodes, in the best case.

These results suggested this technique could also be tested on a platform with more lighter-weight cores. The results are in Figure 4.5 (single-node) and Figure 4.6 (four nodes) and show minimal to no improvement using these techniques. The cause of this difference is still being investigated, but one possible explanation is the difference in interconnection networks and node types.

Finally, this approach was applied to one of the newest supercomputers available for this analysis, Summit, in an effort to replicate the initial results. Rerunning the best single- (Figure 4.7) and multi-node (Figure 4.8) configurations revealed speedups in line with the experiments on Bridges, i.e. 12% and 20% speedup over the best possible baseline configuration for single-node and four nodes, respectively. Additionally, these configurations were compared with an OpenMP-less baseline and another configuration in which an MPI+OpenMP configuration was simulated by creating a single object holding all the nodes simulated data and parallelizing the local computation through a single OpenMP invocation of 42 threads as opposed to combining it with the overdecomposition based approach found in charm. This experiment is labeled baseline (42) in the data and indicated up to a 3x and 4x speedup over this case in the single and four node configurations, respectively.

These experiments sufficiently demonstrate that spreading intensive calculations over multiple cores can improve an application's overall performance. While there is some inherent

Figure 4.3: Bridges, single node results.

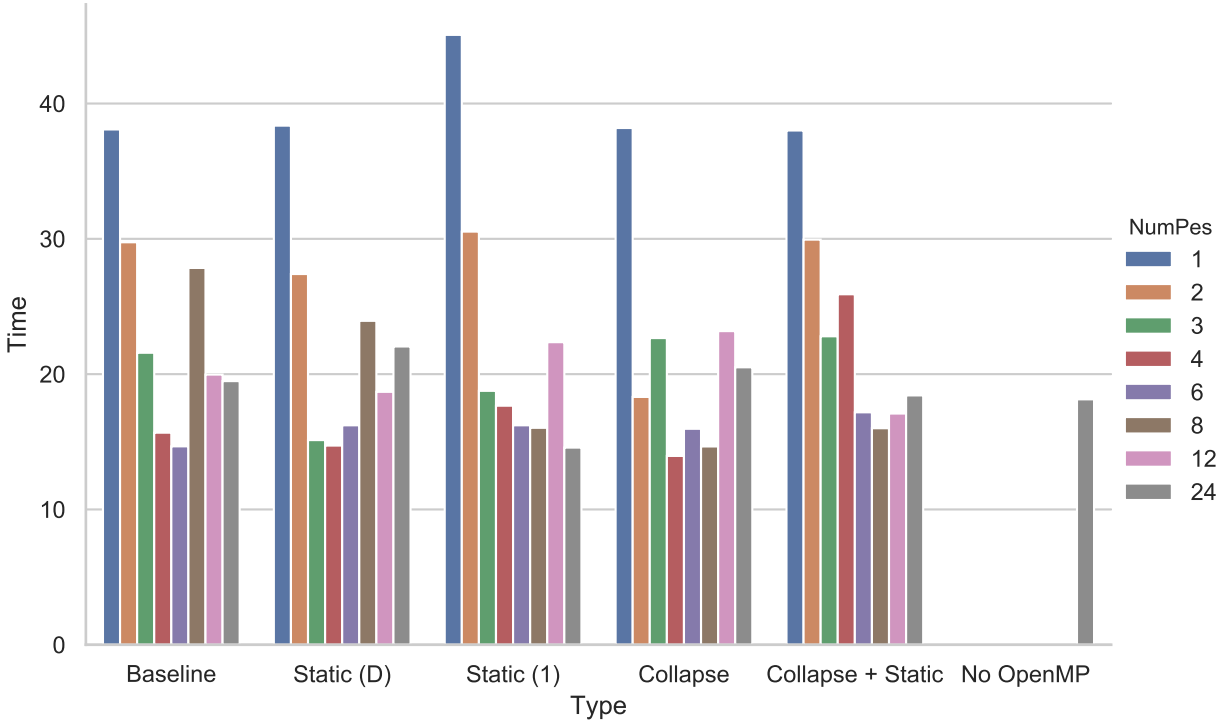


Figure 4.4: Bridges, Distributed (4 nodes) results.

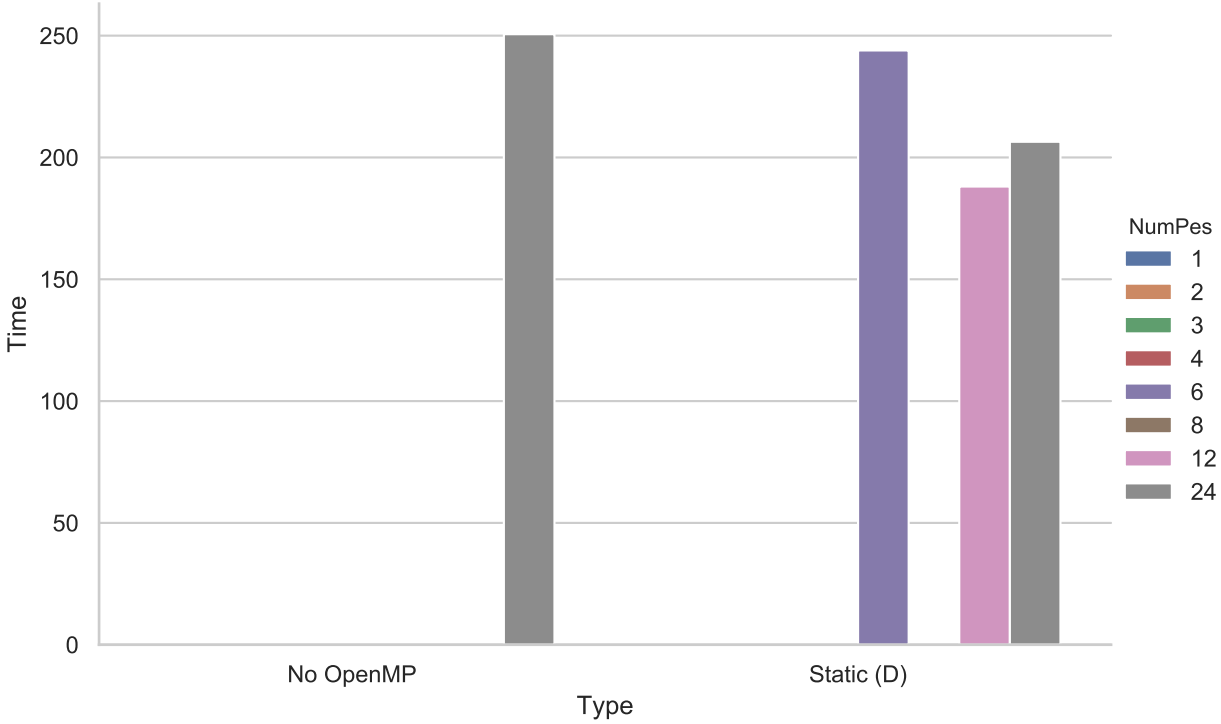


Figure 4.5: Stampede2, single node results.

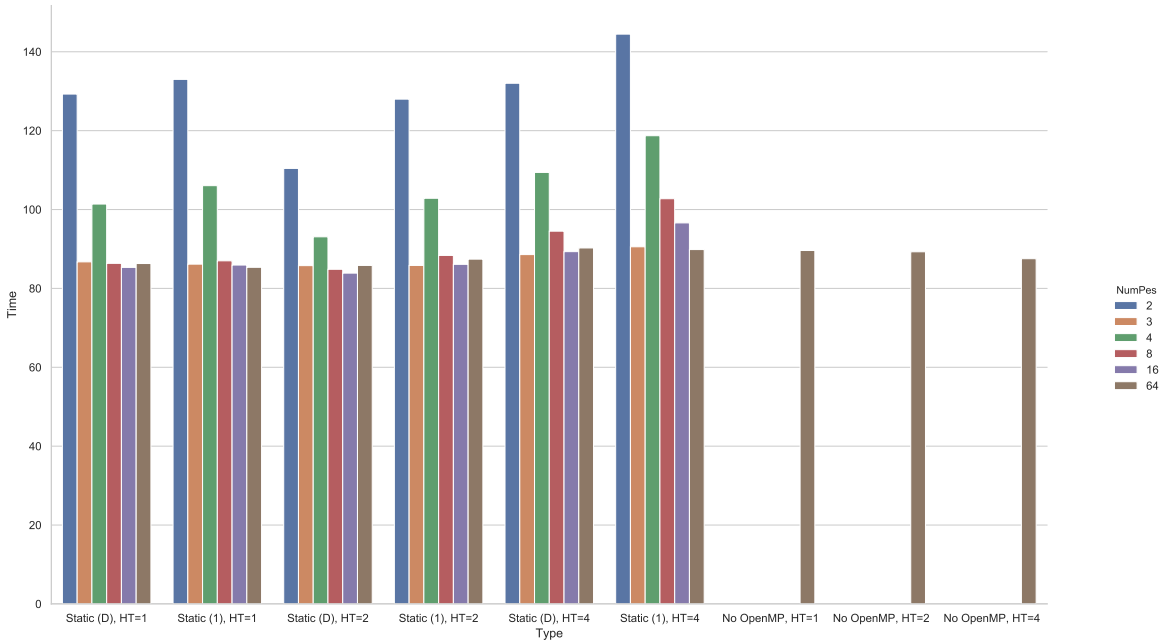


Figure 4.6: Stampede2, Distributed results (4 nodes).

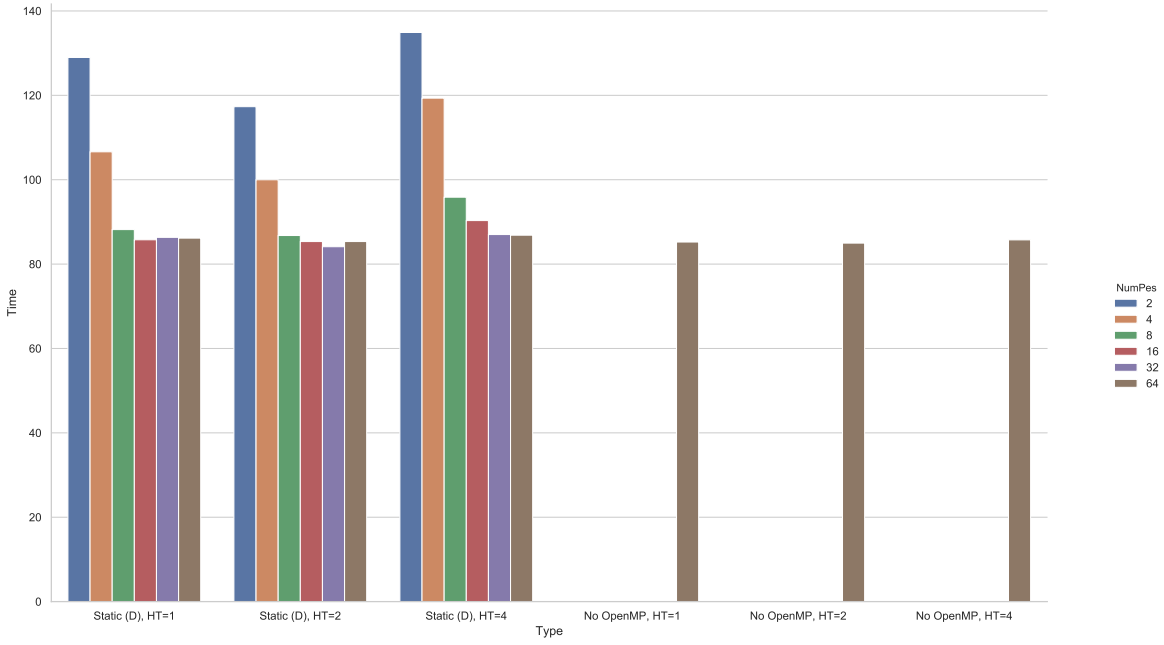


Figure 4.7: Best of 5 Runs for each configuration. 178k x 178k grid, with 7k x 7k chares. 10 iterations. Single node of Summit

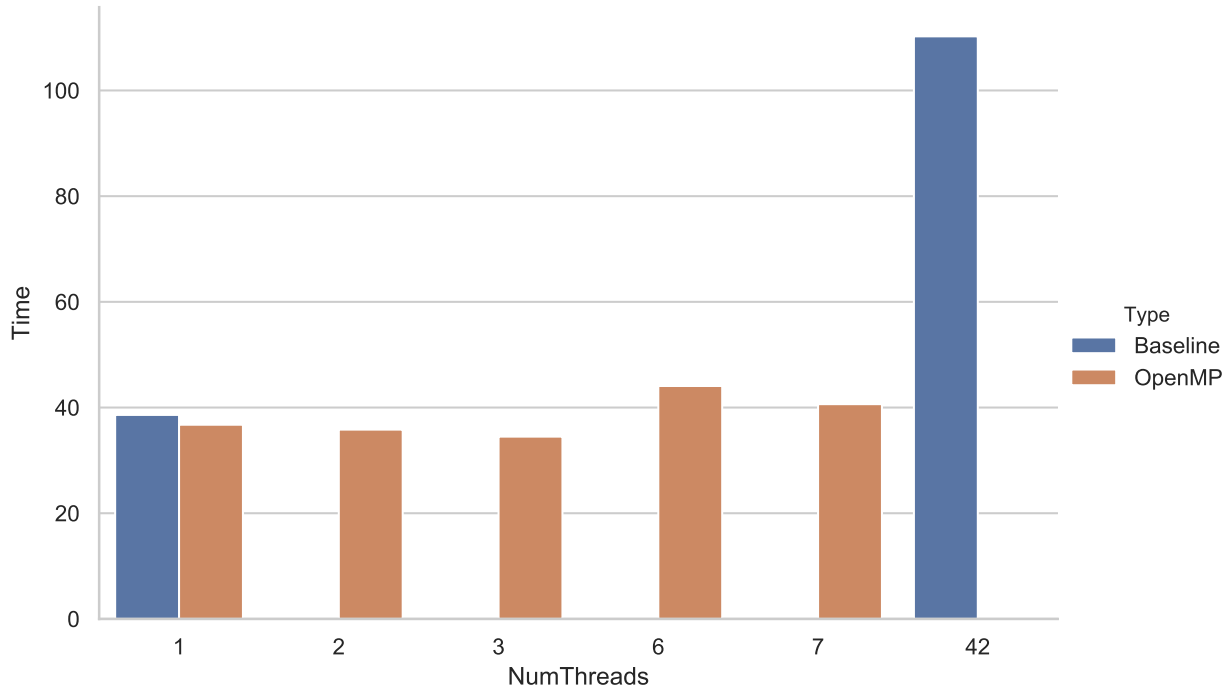
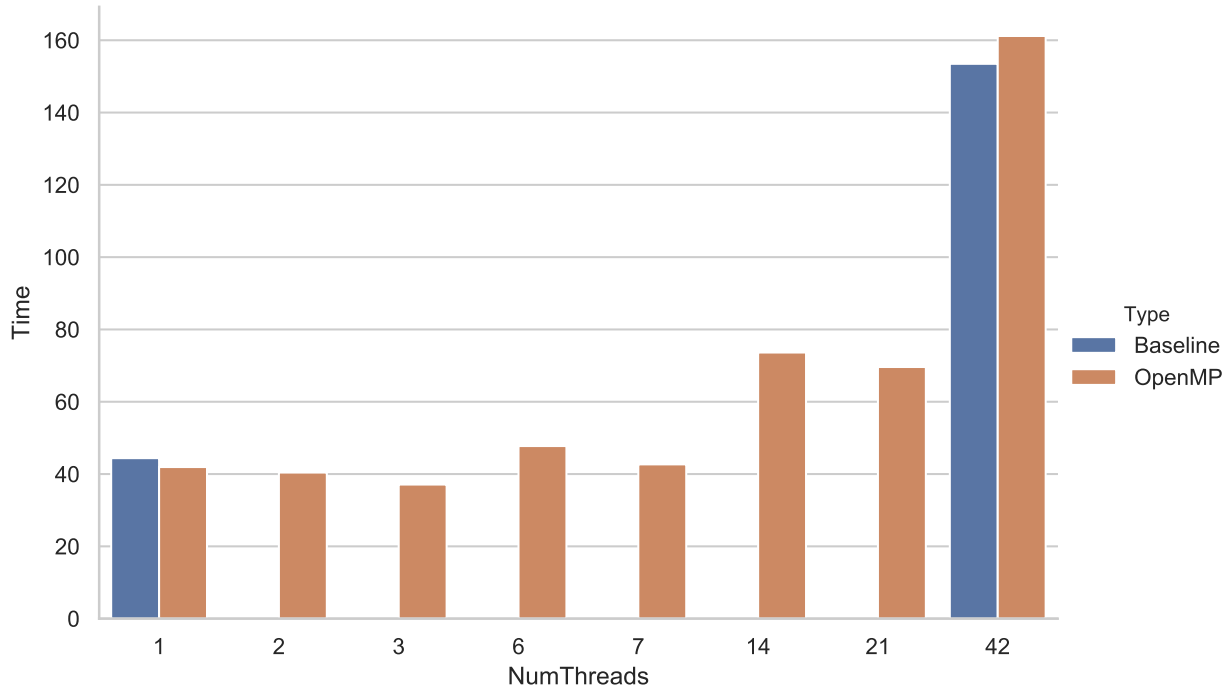


Figure 4.8: Best of 5 runs for each configuration. 178k x 178k grid with 7k x 7k chares. 10 iterations. 4 Nodes of Summit.



overhead with thread creation costs and coordinating data and operations across several cores, these analyses indicate that this extra work can be beneficial to an application's execution. This improvement is due mainly to the improved communication performance of the application, spreading message sends more smoothly over an iteration. Efforts underway to apply this technique to a variety of additional application types and platforms.

4.6 EXTENSIONS

4.6.1 Simulation

In addition to experiments on current supercomputers, we have also simulated the execution of various techniques on machines representative of future trends. To do this, we utilize the TraceR-CODES [37, 38] framework to simulate the execution of our chosen applications on both future compute nodes and networks. In order to make this as accurate as possible, we first capture the execution of our applications, with and without our optimizations, using Charm++'s built-in tracing framework, BigSim [14]. This execution on real machines, generates BigSim logs, or trace files, which can be ingested by TraceR and used to both replay and predict future performance. TraceR allows us to model certain network and machine parameters, adjusting the expected execution time of all or specific functions to simulate offloading to an accelerator, the continutal general increase of node performance, etc. We can independently control the machine network configuration's parameters, including topology, latency, and bandwidth. We thense use these studies to demonstrate the potential future impact and utlity of our approach.

Simulation Stack TraceR is an application developed by Lawrenece Livermore National Laboratory in coordination with the Parallel Programming Laboratory at the University of Illinois. It is built on the discrete event simulation (DES) application, CODES, which is in turn built upon the the Rensselaer's Optimistic Simulation System (ROSS). [12]

Experiments We collected our initial set of BigSim logs using the Summit supercomputer at Oak Ridge National Laboratory (ORNL), running on both a single node and four nodes for our distributed trials, using the same experimental setup as our spreading experiments, i.e. the OpenMP and Charm++ integration with various process to thread ratios simulating a fixed size grid occupying 90% of the node's main memory running for a fixed number of iterations.

Interference We are also currently exploring the potential benefits of our techniques, namely spreading, when used in an environment that exhbits network congestion. [45] We suspect that the benefits may also be apparent in situations where interference from other jobs is causing communication delays. We are actively exploring this hypothesis using the same simulation traces and stack as mentioned previously, e.g. BigSim + TraceR-CODES.

4.6.2 Aggregation

As an extension of this work we turned to the idea of message aggregation to add another possible axis of optimization, as well as to potentially gain back (or improve) performance, by bringing back regularity, decreasing the number of messages, and increasing their size in order to move from a latency constrained regime to a focus on bandwidth utilization. To do this, we leverage the Topological Routing and Aggergation Mesh (TRAM) library [46]. TRAM, aggregates message sends in the Charm++ runtime system.

Our initial experiments were conducted on Summit at ORNL under same conditions as before. We have compiled a draft version of our results in Figure 4.9 below.

4.6.3 OpenMP Integration

Finally, we are also working to integrate the spreading style techniques used throughout into the Charm++ runtime system. This approach is similar to both work with OpenMP and CkLoop, and would allow all Charm++ applications to execute loops automatically in parallel with OpenMP annotations, as opposed to requiring user side intervention for spreading. To accomplish this goal, we leverage the `teams` pragma, originally designed for execution on accelerators, to generate separate leagues of teams that can execute varying pieces of parallel codes. Each of these leagues is controlled by a master thread, much like a single parallel region, and these initial teams and controllers are generated by the runtime at initialization. At this point a specified number of threads are masters, each running a separate league of teams. This is the same concept as before, but implemented using lighter weight threads for isolation of parallel work tasks instead of separate processes as before. Additionally, this approach enables disparate leagues to share work when they are unbalanced. Below, in Figures 4.10-4.12 are the preliminary results from adopting this approach, which shows similar performance characteristics and improvements relative to our intial user based approach, but that can be automatically applied to any Charm++ application that contains OpenMP loops.

Figure 4.9: Preliminary aggregation results, four nodes of Summit.

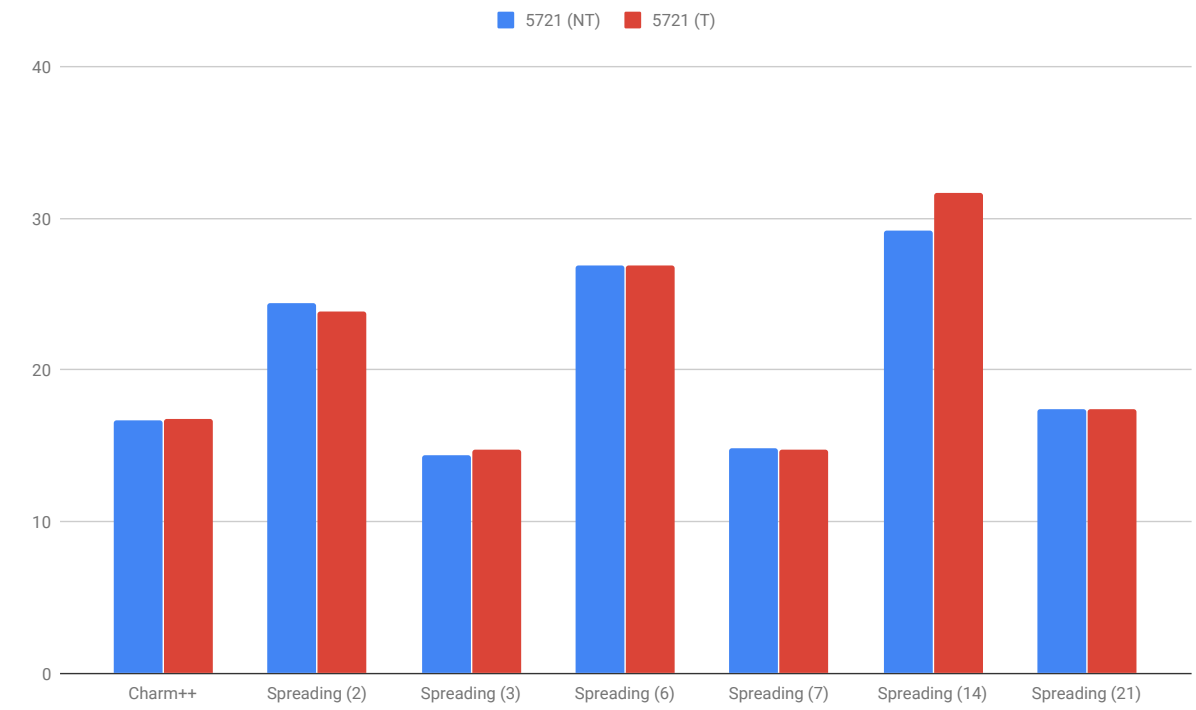


Figure 4.10: **Bridges** - single-node integrated OpenMP runs for SMP and Non-SMP builds.

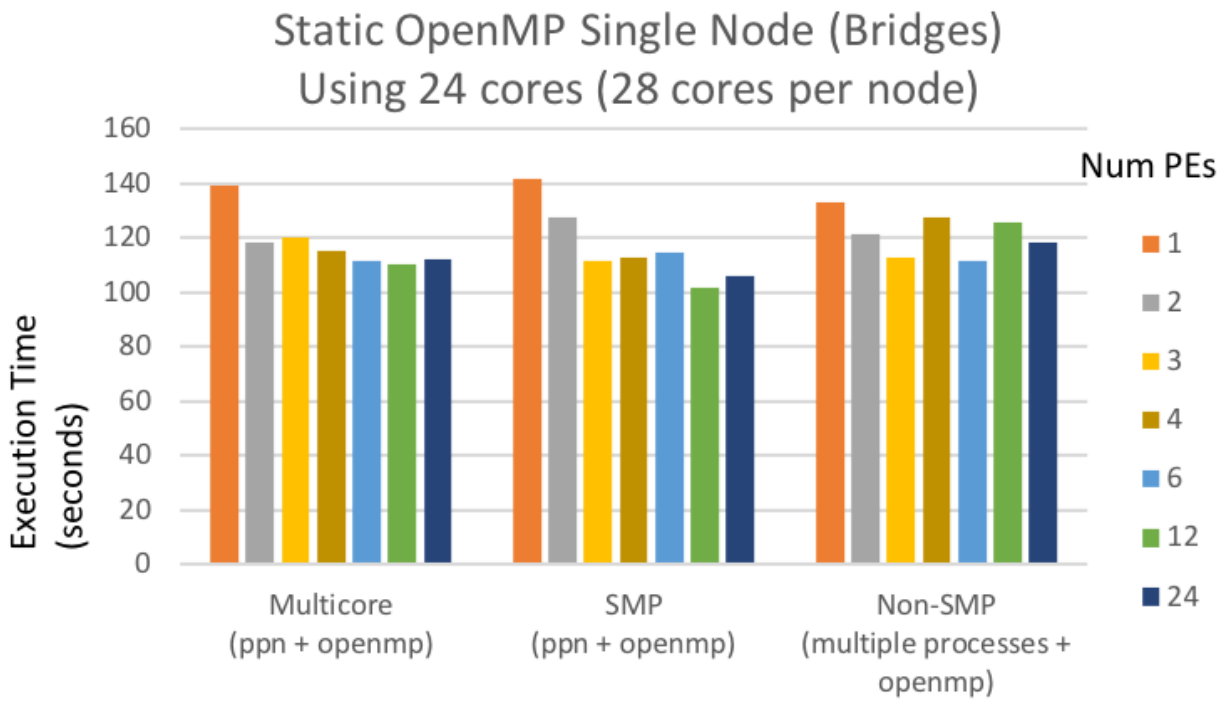


Figure 4.11: **Stampede2** - Skylake 2-node run integrated OpenMP.

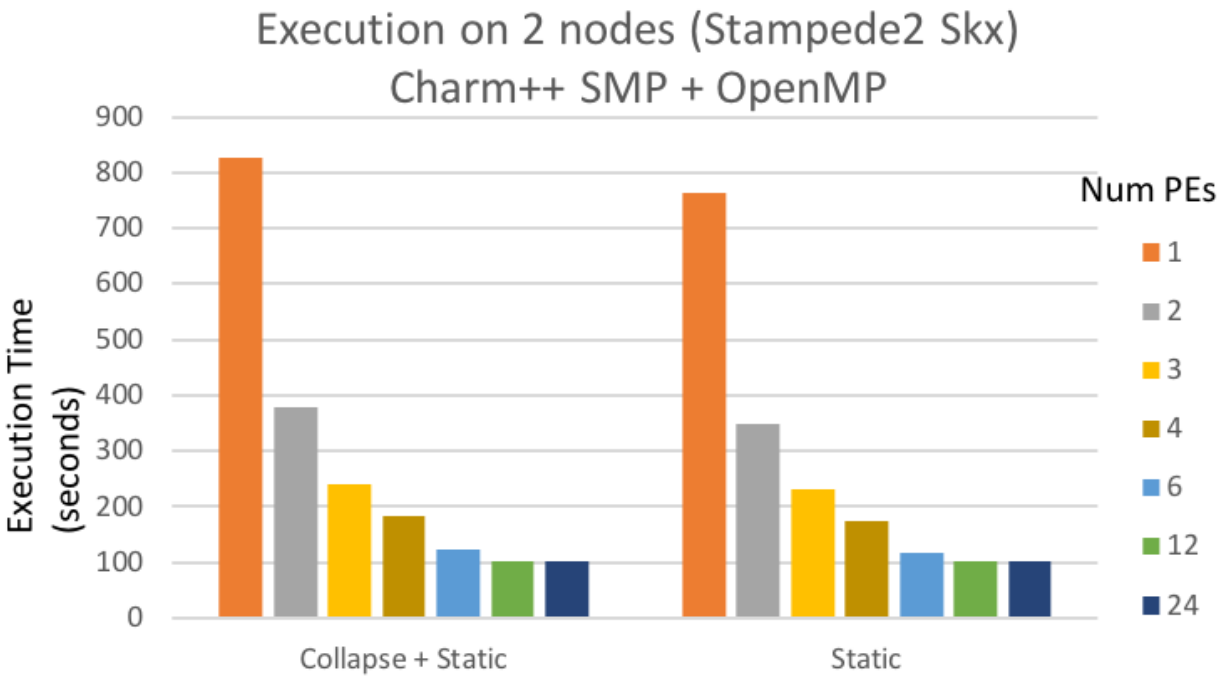
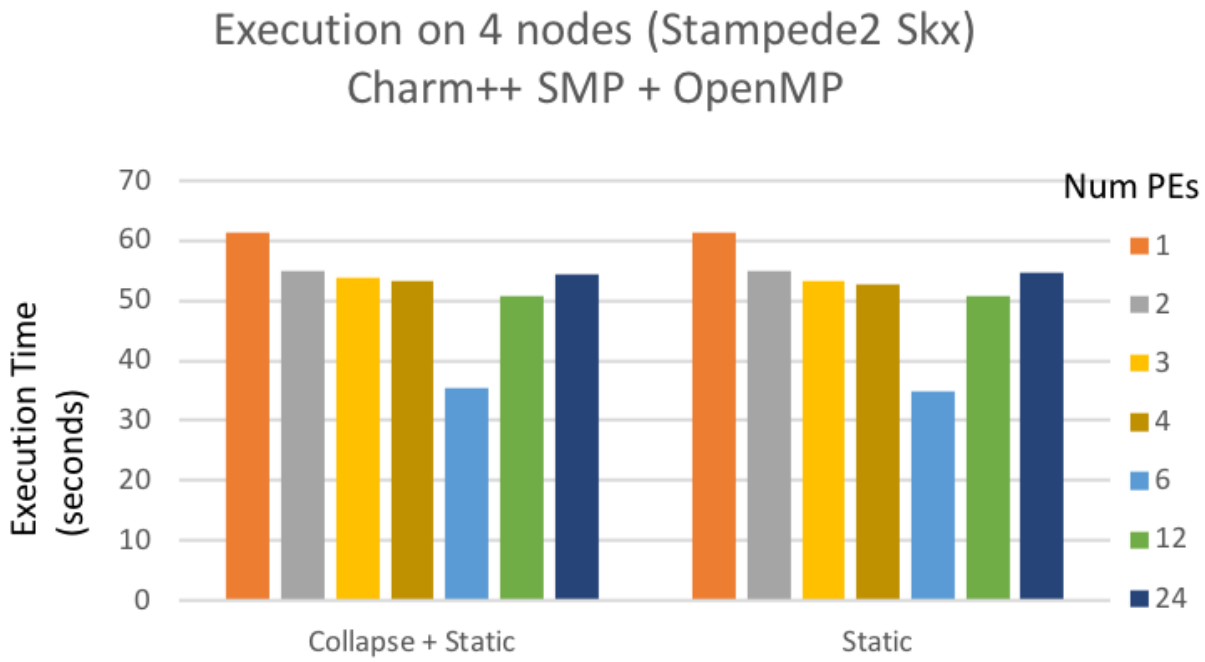


Figure 4.12: **Stampede2** - Skylake 4-node run integrated OpenMP.



CHAPTER 5: RUNTIME COORDINATED HETEROGENEOUS TASKS IN CHARM++

Effective utilization of the increasingly heterogeneous hardware in modern supercomputers is a significant challenge. Many applications have seen performance gains by using GPUs, but many implementations leave CPUs sitting idle.

In this chapter, we describe a runtime managed system for coordinating heterogeneous execution. This system manages data transfers to and from GPU devices and schedules work across the computational resources of the system. The programmer need only tag methods and parameters to enable heterogeneous execution.

Using this system, we observe improvements in programmer productivity and application performance. For selected benchmarks, when using heterogeneous execution we observe speedups of up to 3.09x relative to using only the host cores or only the device.

5.1 INTRODUCTION

Many current supercomputers derive a majority of their compute power from accelerator devices. Nvidia GPUs and Intel Xeon Phis have already seen widespread adoption in many Top 500 machines. As the march to exascale continues, several new machines will derive a sizable portion of their overall FLOPS from GPUs. These include both the Summit system at ORNL and the Sierra system at LLNL. However, programming models and systems have been slow to adapt to this changing environment. In this chapter, we examine an extension to the Charm++ parallel programming library that enables coordinated execution of heterogeneous tasks. We focus on compute kernels developed for Nvidia GPUs using CUDA. Our framework automatically generates tasks from user-annotated functions that can be executed on either the host or device. This strategy ensures full utilization of available hardware and reduces computation time. In this chapter we examine the heterogeneous performance of two mini applications, `stencil2d` and `md`.

5.2 BACKGROUND AND RELATED WORK

Charm++ [47] is a task based, asynchronous parallel programming framework with an adaptive runtime system (RTS). In Charm++ programs, data is decomposed into logical units (chares) which are then mapped to hardware resources (PEs). Chares communicate and exchange data via messages that invoke asynchronous methods. The parallel structure and methods of Charm++ programs are described in a charm interface file, which is parsed

by the charm translator `charmxi` to generate code for the runtime. In this chapter, we modify `charmxi` to generate both host and CUDA versions of the entry methods tagged for execution on different devices. It can be extended to generate code for any hardware platform, but these two targets are sufficient for our tests. We also augment the Charm++ runtime, adding the capability to schedule heterogeneous work across the host and device based on a provided heuristic.

Graphical processing units (GPUs) are becoming prevalent in the HPC community, as is evident from their number over time in the Top 500. Originally intended as special purpose accelerators for graphics applications, they are now user programmable and often referred to as the "device" (as opposed to the CPU or "host" cores) due to their supplementary use in a system. A variety of languages and tools for GPU programming exist ([48], [49], etc.), but GPUs remain more difficult to program for than traditional host cores. Unlike CPUs, GPUs are made up of hundreds of lightweight cores grouped together into streaming multiprocessors (SMs). These SMs share critical resources, such as registers and shared memory. Collections of threads, called warps, are launched on these SMs and execute in lockstep. This unique design can lead to strong performance for some highly parallel applications, e.g. graphics, but can be hampered by its strict SIMD nature (for instance when encountering branch divergence in code). Data movement is also a concern since the GPU cannot directly access host memory. Therefore, data must be copied to the device before being used, which often limits performance due to the latency and bandwidth constraints associated with transferring data across the PCIe bus.

A similar approach to using runtimes in heterogeneous environments can be found in the StarPU programming library[50]. They also schedule tasks, called codelets, and automate data transfer dynamically across different hardware targets. However, StarPU does not have a mechanism to automatically generate kernels for different platforms as our work does. We distinguish ourselves from other task based run times such as OmpSs[51] by offering more generality, not requiring entire programs to be explicitly constructed as a DAG. Similar work has also been carried out in the context of OpenCL[52], [53] with great success, but we can extend our work to multiple nodes.

The authors of [54] propose a solution that divides work into fine grain tasks and enqueues them in a single location. This potentially allows for heterogeneous execution and dynamic load balancing. However, they use a work stealing approach with a persistent device kernel, instead of a central manager, and they do not show results for mixed CPU-GPU execution as presented in this chapter. The Legion programming model [55] can also execute in heterogeneous environments using similar techniques to our approach.

5.3 METHODOLOGY

Our execution model builds upon the earlier work of GPU Manager [56], which handles the delegation and execution of CUDA kernels in the context of the asynchronous message-driven runtime of Charm++ . This allows us to focus our work on higher-level concerns, such as code generation and dynamic target selection in our framework.

5.3.1 Charm++ GPU Manager

The GPU Manager operates by registering GPU kernels to be managed with the runtime system. By having the runtime asynchronously invoke kernels when data is available on the device, we automate the overlap of data movement and execution as seen in Figure 5.1. Due to inherent asynchrony of Charm++ , it is important to ensure that blocking operations, such as `cudaHostMalloc`, are handled by the system and do not block in user code. GPU Manager also automates some tedious CUDA-related tasks, namely copying data to and from the device before and after kernel execution.

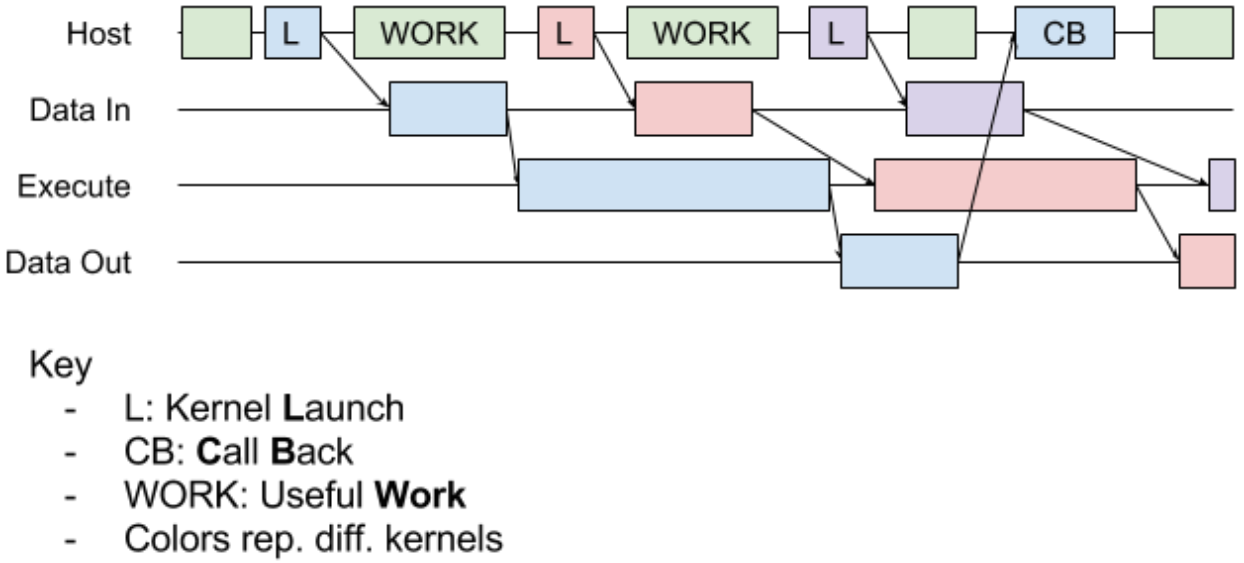
When using GPU Manager directly, the user must write an explicit CUDA kernel and denote buffers which need to be moved to and from the device. The programmer must also register a callback with the runtime, which is called when the kernel is finished and data has been copied back to the host. This step is necessary since the call to GPU Manager returns once the runtime has copied the CUDA buffers; it does not block until the kernel has finished. GPU manager coordinates data movement and kernel invocations through a FIFO queue. When a PE goes idle and enough time has passed, the runtime invokes a progress function to issue new requests to the GPU. At this time, GPU Manager attempts to offload data for a new kernel, launch a kernel with complete data on the device, and move data for the completed kernel back to the host. Finally, when the data for the completed kernel is fully copied back, GPU Manager invokes the user supplied callback to continue execution.

5.3.2 Accel Framework

The Accel Framework[57], or accel, extends GPU Manager by automatically generating CUDA kernels from host code and dynamically deciding where entry methods should be executed.

accel alleviates many of the programmer productivity problems associated with using GPUs effectively in parallel applications by virtue of its automatic kernel generation. This generation occurs only for entry methods annotated with the `accel` keyword. To improve

Figure 5.1: GPU Manager



performance, additional tags can be applied to methods, such as `splittable`, which allows methods to be split into several independent tasks, which can more fully utilize the many processors on a GPU. Inside `splittable` methods, `splitIndex` and `numSplits` variables are defined, analogous to the `threadIdx` and `blockDim` variables in CUDA. This differs from CUDA in that the code can be targeted to a variety of platforms. A full listing of other annotations can be found in [57].

ACCEL has a variety of strategies to determine where to execute particular entry methods. The strategy is passed to as a runtime argument. Example strategies include `+accelHostOnly`, `+accelDeviceOnly`, `+accelPercentDevice`, which specify a static division of work between the computing resources. In this chapter, we manually sweep through different static divisions to observe the performance behavior of the various configurations. However, there are several available automated methods to find the best split, such as greedy strategies and hill climbing. Further description is outside the scope of this chapter and is detailed in [57].

In order to maximize GPU utilization and avoid serialization, ACCEL tries to batch multiple device method calls into a single kernel launch. This batching occurs when a specified count is reached or a certain amount of time elapses. The `triggered` keyword informs the runtime system that the accelerated entry method (AEM) will be invoked on every chare and that all chares will invoke said entry method before any chare invokes it a second time. Programmers can also specify the number of threads to be used per block in a kernel launch instead of having the runtime automatically determine one.

It is beneficial for the RTS to minimize data movement and overlap it with computation

when possible. Data movement is automatically overlapped with computation as described in Section 5.3.1. Method parameters are automatically copied to the device, but are not copied back since the CHARM++ model dictates that entry method parameters have no lifetime beyond the entry method. However, object data used in an `accel` annotated method must be marked as `readonly`, `writeonly`, or `readwrite` to indicate whether it should be copied in, out, or both. Additional annotations such as `shared` and `persistent` allow the user to control the lifetime of the data on the device. With these annotations, `charmxi` automatically generates code to move data to and from the device. The implObj variable seen in the code is required due to the lack of a proper CHARM++ compiler since we require a handle to the chare object and its data.

The last token in Listing 5.1 specifies a callback invoked when the `accel` entry method is finished executing. It is used in the same way as in GPU Manager but is listed here instead of as an input or member variable due to parsing constraints. The callback is used to send messages to invoke other methods since CHARM++ messages cannot be sent from accelerator devices.

Listing 5.1: Accelerated Entry Method Annotations

```
entry [triggered splittable(NUM_ROWS) accel] void doCalculation() [
 readonly:float matrix[DATA_BUFFER_SIZE]
     <implobj->matrix>,
 writeonly:float matrixTmp[DATA_BUFFER_SIZE]
     <implobj->matrixTmp>
] { ... } doCalculation_post;
```

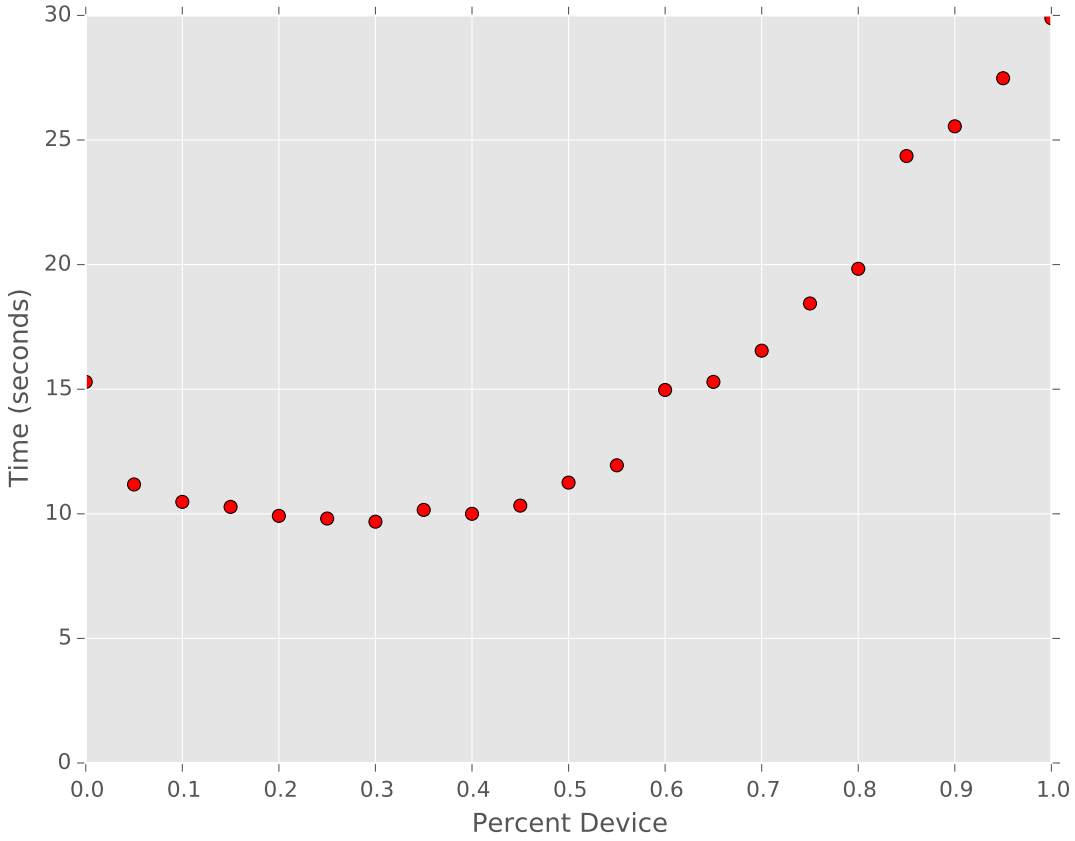
5.4 RESULTS

We analyze performance for varying distributions of work between the host and device for two different applications, `stencil2d`, which implements a two dimensional stencil, and the more complex `md`, which simulates electrostatic molecular dynamics. In both applications, the main compute methods have been annotated with `accel` and other tuning parameters. Our tests vary the percentage of work allocated to the device from 0% to 100% in increments of 5%. Theoretically, hybrid computation will improve performance, since more hardware can be used, but data transfer and batching costs create performance impediments.

The experimental results were gathered on the Stampede supercomputer. In particular, we used the visualization nodes of the system, which each feature an NVIDIA K20 GPU and two Intel Xeon E5-2680 processors. All runs were performed on a single node of the system

with 16 CHARM++ processing elements, matching the 16 cores in the node. We measured elapsed time from the start of the calculation to the end of the last error calculation for both applications. This does not include startup or other fixed costs.

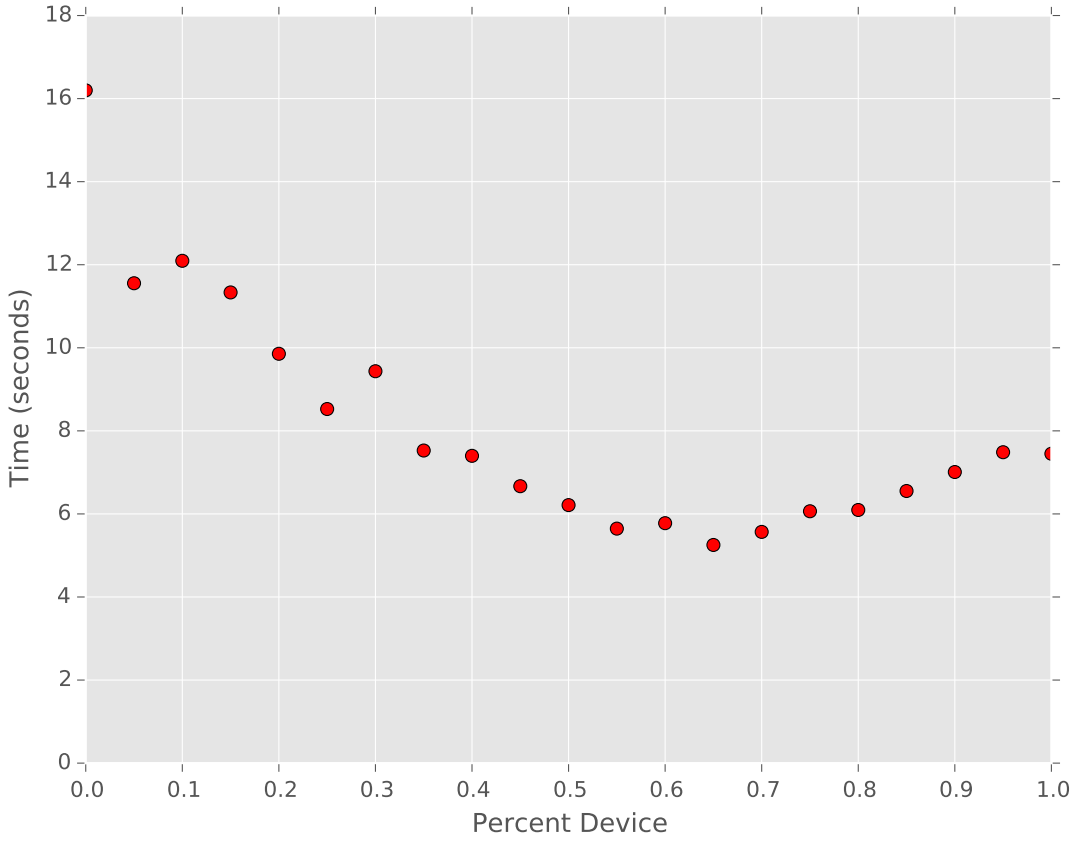
Figure 5.2: Timing for `stencil2d`



5.4.1 Stencil 2D

`stencil2d` performs a single-precision weighted five point stencil. Given results use a 6144x6096 2D array decomposed into 24 tiles per dimension, a 254x254 section per element. For work performed on the GPU, the algorithm performs approximately 1.25 single-precision FLOP per transferred byte (10 FLOP/(1 float in + 1 float out)). As shown in Figure 5.2, this low FLOP/byte ratio causes the host only case to beat the device only case. Optimal performance occurs in the 30% device case.

Figure 5.3: Timing for `md`



5.4.2 Molecular Dynamics

`md` executes much faster in the device only case than in the host only case. Given results use a 5x5x5 3D array with 256 molecules per array element, a total of 32k molecules. The FLOP per byte ratio for `md` is higher than that of `stencil2d` since each particle has a relatively complex interaction with every other particle in the simulation, requiring distance, electrostatic force, position, and acceleration calculation and normalization. Optimal performance occurs in the 65% device case.

5.4.3 Analysis

For both selected applications, we observe an increase in performance when using both the host and the device as compared to using only the host or only the device.

There are some clear discontinuities in the performance of the chosen applications; see

the jumps at 60% and 85% device in Figure 5.2. These are likely a performance artifact of the batching used in the Accel Framework. Since the GPU is a throughput-oriented device, launching an additional batch takes much longer than adding some work to an existing batch. This behavior is not seen for smaller allocations of work to the device because the host was spending more time on the work than the device, so it was the dominant term.

The timing data follows a "bathtub plot", so termed because it is low in the middle and high on both sides. When performance follows this pattern, the goal is to set the parameters such that execution happens in the "floor" region. As shown in Table 5.1, the best configurations achieve speedups of between 1.46x and 3.09x relative to host only and device only configurations.

	Best Split	Host Only	Device Only
`stencil2d`	30% device	1.58x	3.09x
`md`	65% device	3.02x	1.46x

Table 5.1: Speedup of Best Configuration Relative to Host/Device Only

5.4.4 Caveats

All applications do not benefit from a heterogeneous execution system. Even applications that are amenable to heterogeneous execution may not see benefit in all configurations. The most significant reason for this is data movement. Just as HPC applications can slow down when run on two nodes versus one node due to the effects of adding network communication, using a GPU can degrade performance unless the application amortizes the costs of data movement. Additionally, not all algorithms are well suited to run on the GPU. In particular, programs that make heavy use of branching, that cannot expose enough parallelism to fully utilize the GPU, or that are composed of a variety of disparate tasks do not perform well on GPU hardware.

However, large HPC applications often feature a variety of different kinds of work, so it is likely that some portion will improve when executed on heterogeneously.

5.5 FUTURE WORK

This work is a small survey of the initial implementation of Charm++ support for heterogeneous compute environments. Our current solution to generating CUDA kernels,

copying the entry method body directly into the kernel body with a few extensions, can be vastly improved. Using `splittable` allows performance to be greatly improved, but does not allow the user to make platform specific optimizations. One potential extension is to allow the user to explicitly provide optimized kernels for different platforms, as other runtimes, such as OmpSs, allow. We could also extend GPU manager to observe utilization and launch multiple kernels.

ACCEL currently provides mechanisms for the programmer to control data movement to and from the device (or in the case of persistence, residence). However, there is more work to be done here. For instance, adding support for GPU Direct, which would allow GPUs to directly communicate with each other, and reducing the number of copies of data made inside CHARM++ when transferring to and from the device. We also plan on taking into account data location and movement cost when making scheduling decisions to further minimize data movement. We anticipate that NVLink will partially alleviate some of these problems.

Finally, we plan on applying this work to automatically load balance heterogeneous applications at large scale. By extending the CHARM++ load balancing framework to support heterogeneous load balancing, we will balance work both across nodes and across the hardware resources in a node. This would enable CHARM++ programs to adapt to arbitrary hardware platforms of arbitrary size with minimal code changes.

5.6 CONCLUSIONS

The use of accelerators in HPC has grown in recent years and will likely continue for the foreseeable future. While many HPC applications make use of accelerators to improve their performance, it remains difficult to fully utilize all hardware resources available on a machine.

In this chapter, we describe a runtime for managing execution in heterogeneous environments. In contrast to common ways of using accelerators, in this scheme, the system handles data allocation, transfers, scheduling, and coordination across the heterogeneous hardware. This system requires minimal additional work from developers and is not tied to any specific accelerator platform.

We demonstrate the efficacy of this system for GPU execution using two CHARM++ applications. We achieve up to a 3.09x speedup relative to running the main computation solely on the host or device.

CHAPTER 6: FUTURE WORK

6.1 APPLICATION CASE STUDY

We now turn our attention to a large application case study. Whereas in previous chapters we have demonstrated the validity of our techniques using a combination of mini-apps and simulations, we will now employ these techniques in the context of a broader parallel application.

Table 6.1 describes potential candidate applications for this work, including a brief description of their scientific domain, basic parallel programming pattern, production status, and parallel programming framework, including whether or not they use GPUs.

Our plan is to profile these applications using their default benchmarks, e.g. Water 256 and MOF for OpenAtom, on a variety of platforms, as we did when exploring our techniques in Chapter 4. This includes scaling runs on production level machines as well as simulated runs using BigSim and TraceR (or another tracing software, e.g. DUMPI [60] where appropriate. From these experiments, we will review profiling data using the software's profiling tools, e.g. Projections [61] for Charm++and AMPI. After this stage we will document the potential and observed problems and choose one or more applications which we believe our techniques are applicable to improving their performance. We will then implement our techniques describe in previous chapters, individually and together, and re-run these scaling experiments observing their new performance.

Due to their complex nature, heavy reliance on communication, and variety of both parallel patterns and programming frameworks in addition to the broad applicability of our approaches, we expect several of these applications will benefit from our optimizations, es-

Application	Prd	Domain	Pattern	Parallelization	GPU	Cite
OpenAtom	Yes	Quantum chemistry		Charm++	No	[35]
ChaNGa	Yes	Astrophysics	Oct-tree	Charm++	Yes	[36]
Barnes-Hutt	No	Astrophysics	Oct-tree	Charm++	No	[58]
MILC	Yes		4D stencil	MPI	No	[39]
PlasCommCM	Yes	Combustion		AMPI		
Lulesh	No	Particle physics	Stencil	AMPI and Charm++		[59]

Table 6.1: Candidate applications for further study, including their features. Prd is short for production and lists whether an application is used for production science runs. Pattern describes the general parallel pattern used in the application with parallelization listing what technology is used to code this pattern. GPU is whether an application uses GPUs.

pecially in preparation for future machines with networks that are constrained relative to their node level FLOPS.

In the future, we aim to develop a suite of tests suitable for applications to run to determine automatically whether and what potential techniques their parallel code would likely benefit from.

6.1.1 MILC

For our final evaluation of these approaches, we choose to utilize the MILC application for several reasons. MILC, which stands for MIMD Lattice Collaboration, is a lattice code developed, as it name implies, via a large scale collaboration between many physicists. It is also a stencil code, which spreading has shown good results for previously. Additionally, we already have an AMPI version we can utilize, including for BigSim trace generation which can then be used for TraceR simulation in turn. Finally, MILC is created and maintained by an external group, ensuring our techniques work with a broader swath of applications.

6.2 RE-EXAMINING THE FOLK THEOREM ABOUT COMMUNICATION COSTS

Communication Computation Overlap is the amount of time as a fraction of the total execution time that the application spends simultaneously sending messages and performing useful computation. Obviously, in the ideal scenario an application would be near 1.00 (or 100%) overlap between these two phases, constantly sending messages that perfectly overlap their corresponding computational portions. However, we know this is a near impossibility yet it remains our goal.

General Description When thinking about the room for optimizing communication and computation overlap there is a folk theorem, or rule of thumb, that "states" that the maximum speedup possible is 2x.

Formalization The reason for this 2x factor is clear if we imagine both the best and worst case scenarios for communication computation overlap in an application.

In the worst case, or a 0% overlap, the application is either performing communication related work (and doing nothing else productive) or performing a computation (and not sending or receiving data or doing other productive work). Since both the communication stage and compute stage are entirely on the critical path the total time to execute the application is: $t = t_{comm} + t_{comp}$.

On the opposite end, complete overlap (or 100%) would mean that the application is constantly sending and computing data. We can calculate this run time with the following

formula: $t = min(t_{comm}, t_{comp})$.

Therefore, the speedup due to overlap is: $(t_{comm} + t_{comp})/max(t_{comm}, t_{comp})$. The highest possible value for this expression is attained when $t_{comm} = t_{comp}$, leading to a speedup of 2.

Bulk synchronous models such as MPI, when used naively, are infamous for the former behavior. That is, when written naively, MPI applications tend to exhibit long phases of only computation followed by an intensive communication phase which communicates the required information for the next computation but which must wait on the communication to complete before continuing.

We detail our reasons we hypothesize this 2x barrier is not always valid is, in addition to issues we explored in the previous chapter, in the following sections. Succinctly, they are:

- overhead
- non-linear effects
- responsiveness/remote data/priority/critical path

6.2.1 Overhead

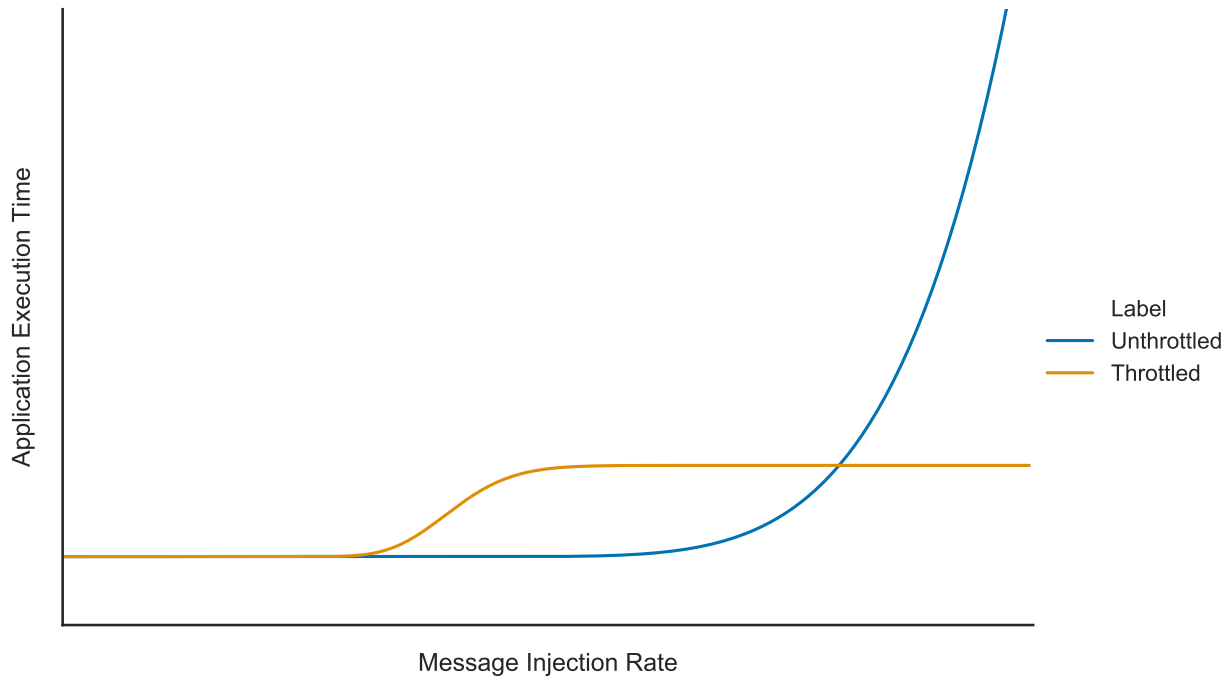
One major component not capture in our original simplistic model equation, $t = t_{comm} + t_{comp}$ is the overhead of various system components. These range from the CPU to network interface card (NIC) and possibly include any device on the send/receive path of a message [62]. We examine the details of the overhead from message collisions, termed contention, in Section 6.2.2 below. We can imagine several scenarios in which overhead would significantly impact the performance of a parallel application. One specific scenario would be in the case where processors are 'overloaded' with work, that is they are constantly computing useful results and unable to inject messages on the network efficiently due to these competing demands. One possible solution to this situation is reserving a single thread on each node to handle communications (both in and out bound) while the rest continue calculations. We can see that this leads to improved performance yet can lead to its own problems where the single communication thread is overwhelmed with messages. One possible solution would be to enable a flexible number of threads for communication while reserving the rest of the threads on a node for computation. One can also construct similar arguments for handling communication with GPUs as well. Regardless, we can capture the effect of this, and other, form of overhead in our model by adding an additional term, $t_{overhead}$ that expresses the time not spent directly computing or communicating. Thus our updated model would be $t = t_{comm} + t_{comp} + t_{overhead}$. In this work, we plan to investigate several causes of overhead, including communication thread bottlenecks, in both the CPU and NIC co-processor, and we pose several potential solutions including a flexible number of communication threads

as well as utilizing message aggregation, through the TRAM framework [46] to improve performance. We plan to conduct these studies using both actual and simulated runs on Summit and various other systems.

6.2.2 Network Contention

Another potential source of overhead captured by our $t_{overhead}$ term, comes from contention amongst various nodes trying to send messages across the same interconnection links. This form of overhead deserves special attention as we can model it separately, due to its potentially nonlinear nature and different possible solutions. Similar to our approach with generalized overhead from both the hardware and software (runtime) above, we plan to demonstrate the effectiveness of our suggested approach via both real and simulated runs to highlight the potential importance for future machines that will likely have reduced network resources. In this case, our solution is to provide less resources to the application, as opposed to more. By this, we mean throttling the amount of messages sent per unit time in an application via software/runtime controls. If we look at Figure 6.1, we can see a potential

Figure 6.1: Hypothetical non-linear effects on execution times



scenario in which a linear increase in message injection rates lead to a non-linear slowdown in network, and thus application, performance. In our hypothetical scenario, the blue line

represents the normal application performance which follows an exponential decrease in application execution time as we saturate the network. The yellow line represents our idealized solution, which slows application performance even for smaller message injection rates but which in turn smooths out the network performance and thus the overall application execution time. Somewhat paradoxically, by throttling our application and ensuring it does not exceed a congestion threshold we hypothesize that we can improve performance. We aim to demonstrate these results in a real application, e.g. OpenAtom (see Chapter 6.1), and simulated experiments. Additionally, we plan to work on automating this detection and software throttling to ensure all applications can benefit from it.

6.2.3 Responsiveness

Our final area of concern in regards to possible application overhead is that of responsiveness. Responsiveness in the case refers to the ability, or inability, of the application to process and execute incoming messages on a timely basis. We can imagine many potential scenarios when this might be a problem, but one obvious case is mis-prioritizing local work not on an application's critical path over remote work that is. In fact, in the worst case where the application no longer overlaps communication and computation i.e. waits until the end of a step to send messages, we would expect our model equation to expand to $t = n * t_{comp} + m * t_{comm}$. We've seen one potential solution to this problem, a set of dedicated communication threads. Other possible techniques that can help to mitigate this problem include overdecomposition and correct prioritization. We plan to create a synthetic application which exhibits this exact scenario in order to further study its possible effects on application execution. From there, we will test several possible remedies described above and then generalize them to larger scale runtimes and applications. In this section, we primarily focus on CPU responsiveness but other components of the machine, namely NICs and links, can also suffer from this problem. Solutions to these problems as similar to those described above.

6.3 ADAPTIVE HIGH-PERFORMANCE COMPUTING SYSTEM DESIGN FOR NEXT GENERATION SCALABLE WORKLOADS

6.3.1 Introduction

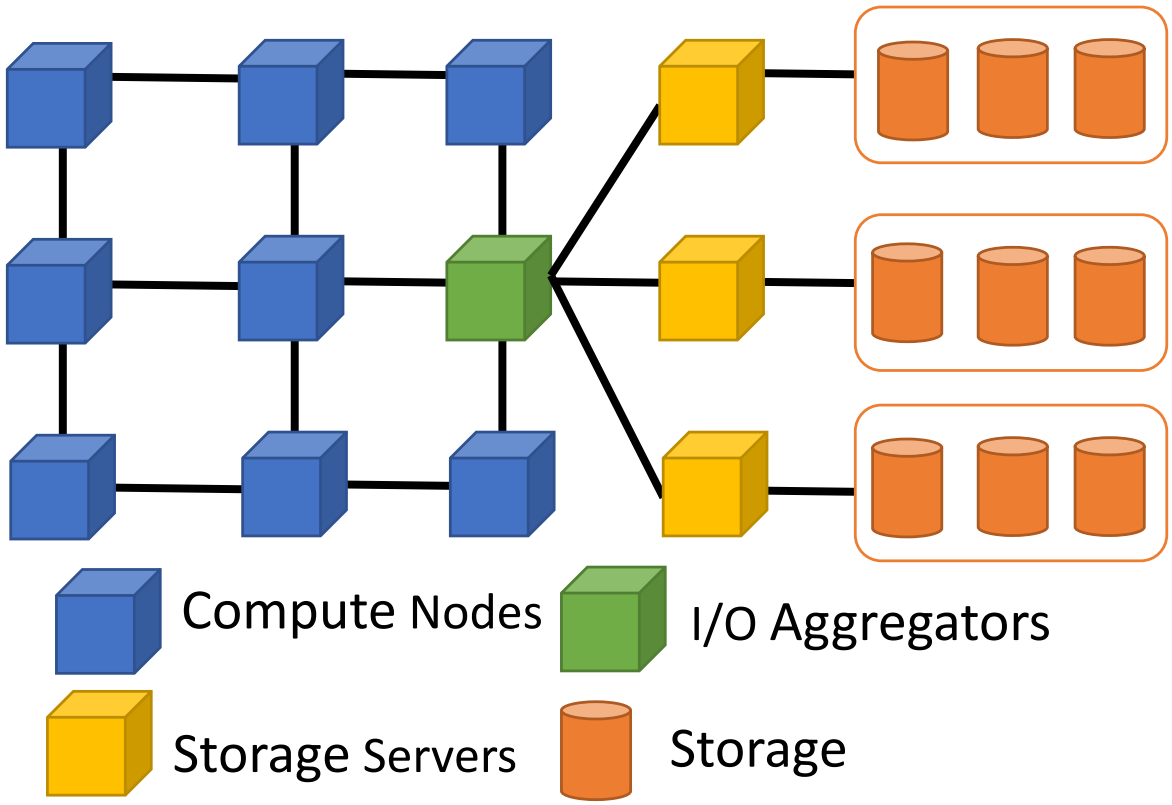
Large-scale high-performance computing (HPC) systems are fundamental to scientific advancement across all fields of science and engineering. As new HPC systems come online,

their hardware and software design targets a few high-priority applications or application classes to provide them with the best performance. Primarily, these applications are large-scale simulations of physical phenomena evolving over time — e.g., climate, molecular dynamics, astrophysics — that execute in the "bulk-synchronous" model [63, 64, 65]. Applications with this execution model have roughly fixed amounts of computation, communication, and I/O throughout their execution. However, as the HPC market converges with the Cloud Computing market, new applications that do not conform to the "bulk-synchronous" model are run on HPC systems. Thus, the design of the system can not be too specific to the target applications because it must also support a larger set of non-priority applications. These new applications vary between latency critical quality-of-service (QoS) applications, machine learning, and data analytics applications with irregular execution patterns. Future systems, if targeted at a few high-priority applications, will not be well designed for a diverse set of applications that may run on it over its lifetime. The goal of this work is to develop the Adaptive Runtime for Compute and I/O (ARC-IO), a unified scheduler and I/O subsystem written using the active message programming model that enables HPC systems to dynamically adapt the node's configuration and role — compute vs I/O — to the current workload characteristics. ARC-IOenables improved scalability and performance for diverse applications by mitigating I/O bottlenecks.

Figure 6.2 shows a generic overview of a current HPC system with a static distribution of resource types — e.g. compute nodes, I/O nodes. The compute nodes in the system are heterogeneous comprising CPUs accelerated with GPUs. However, other accelerators such as FPGAs, TPUs, SmartNICs have been shown to accelerate HPC and Cloud applications [66, 67, 68]. Currently, users must decide which accelerator(s) to use and how to partition work between them. Abstracting the accelerator selection to a dynamic runtime system has the potential to improve performance of applications. We plan to construct an offload system that will use machine learning and runtime performance monitoring to intelligently schedule offload tasks to accelerators.

As applications scale to leverage more nodes, the system's I/O performance has been shown to be limiting factor in performance [69, 70] and system reliability [71]. As system size grows, DRAM sizes are increasing faster than I/O bandwidth which exacerbates the issue. In times of high I/O demand, the I/O subsystem can become backlogged with requests leading to bad performance and performance variability [72, 73]. We propose ARC-IOto unify the scheduler and I/O subsystem to dynamically reconfigure free compute nodes to function as I/O nodes enabling better handling/caching of incoming I/O requests. Moreover, our I/O subsystem is implemented using the active message model to enable improved flexibility and scalability Figure 6.3 shows this adaption to the system from Figure 6.2 with more I/O

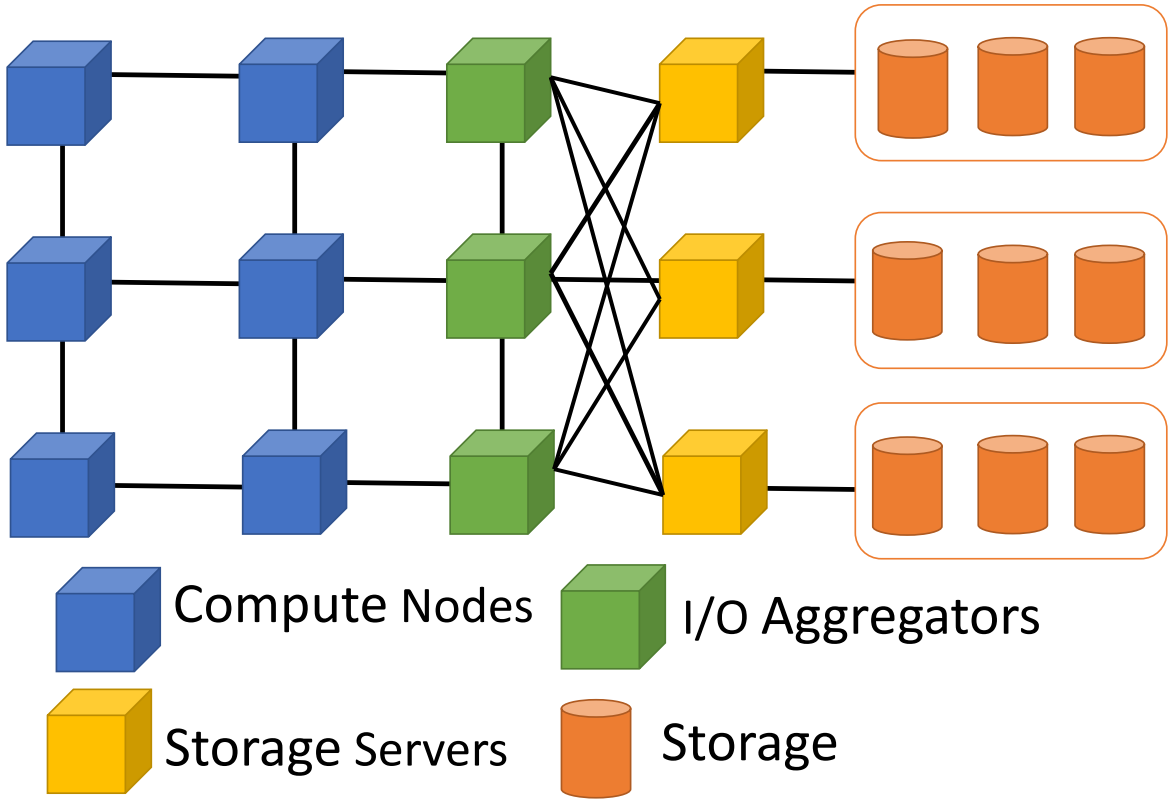
Figure 6.2: Architecture of a current HPC system.



nodes to handle incoming requests. The proposed I/O subsystem leverages heterogeneous accelerators to efficiently encode/decode and compress/decompress data moving in and out of the file system. ARC-IOdynamically load balances by assigning I/O requests to under utilized nodes. Thus, application's I/O performance will scale commensurately with the application even when many applications compete for I/O.

Due to the ever expanding workflows on future HPC systems, effort must be taken to ensure the *trust* and *integrity* in the data. Input data for applications may come from sensitivity sources — e.g. sensor devices, private data, proprietary, medical data — requires trust that the data is correct and has not been corrupted by malicious agents. The use of lossy data compression to accelerate data movement and reduce storage sizes perturbs data and possibly results. Moreover, providing a ledger of what operations took place provides transparency and reproducible. We propose to extend the idea of provenance to ARC-IOfor ensuring reproducibility of workloads and also as a means of producing deterministic configurations of workloads for detecting anomalous system events. We will employ lossy

Figure 6.3: Architecture of a HPC system adapted using ARC-IO.



compression to mitigate the cost of large data movements.

The development of ARC-IOis broken down into five interconnected research efforts: (1) improving performance via dynamic analysis and measurement of applications; (2) extending the scheduler to hook into the I/O subsystem; (3) employing provenance to ensure the integrity of data and improve reproducible execution; (4) leveraging reconfigurable hardware and ML to improve task acceleration; and (5) investigating lossy and lossless data compression to improve transfer bandwidth.

We evaluate ARC-IOusing traditional HPC bulk-synchronous applications and benchmarks along with data analytics and machine learning applications (see Section 6.3.4).

This work addresses high-performance computing (HPC) systems' inability to adapt their configuration. Adapting HPC systems to better conform to the current workload enables higher performance on a diverse set of HPC and Cloud applications. Better management of the I/O subsystem will reduce performance variability when multiple applications compete for the file system. This work offers the following intellectual merit: (1) constructs ARC-IO,

a unified scheduler and I/O subsystem that dynamically scales the number of compute and I/O nodes based upon the current workload and load balances the I/O tasks; (2) implements the I/O subsystem using active message to enable dynamic flexibility and scalability; (3) develops end-to-end real-time provenance in HPC for security and reproducibility; (4) enables real-time adaption and reconfigurable node architectures using performance models; and (5) establishes a connection between lossy compression error and application data fidelity.

6.3.2 Focus Areas

The research activities of this work address the following focus areas (FA):

1. **Computer Architecture:** To account for a variety of applications on next generation HPC systems, we envision nodes in the system to be over provisioned with various types of accelerators (e.g. FPGAs, GPUs, TPUs, SmartNICs). We propose to use real-time performance measurements to guide the selection of accelerator choice at runtime using performance models.
2. **High-Performance Computing:** We plan to explore a diverse set of HPC applications that have some of the following characteristics: real-time constraints, compute vs I/O bound, coupled, various phases. ARC-IOallows the system to dynamically configure the ratio of compute and I/O nodes and their architecture to yield improved I/O performance for the applications.
3. **Systems:** We investigate sensor networks that transmit data using lossy compression to HPC facilities for real-time analysis. In doing so, we explore correctness and accuracy of the workloads due to lossy compression and 3rd party data modifications.
4. **Security and Privacy:** We provide a mechanism to establish data trustworthiness in ARC-IO by using data provenance. Data provenance provides a comprehensive history of system interactions over time. This enables reproducibility of system workloads and a means of detecting anomalous events which could have grave consequences on critical system applications.
5. **Theory and Algorithms:** We establish a formal relationship between lossy compression error and correctness of data analytics and ML; removing costly trial-and-error error-bound selection.

6.3.3 Targeted Systems

ARC-IOtargets large-scale systems where each node has various accelerator types. The role and architectural configuration of the nodes will change based upon the perceived system workload needs determined by ARC-IO. In addition, we will investigate sensor networks that stream data into our proposed system. To test our ideas, we will leverage the Clemson's Palmetto Cluster, the C2M2 Intelligent Transportation System test bed, and Cloud Lab.

6.3.4 Targeted Applications

HPC Applications and Benchmarks Traditional HPC applications come from many scientific and engineering domains. They tend to center around dense or sparse linear algebra and n-body algorithms. When evaluating ARC-IOfor improvements in I/O scalability and application performance, we will leverage traditional HPC benchmarks such as LINPACK [74], HPCG [75], Graph500 [76], and IO500 [77] to gauge a single performance metric such as computational performance or I/O performance. In addition, we will leverage proxy applications from the U.S. Department of Energy's Exascale computing project [78]. To serve as interference when stressing the I/O subsystem, we will use IOR [79] to read/write varying amounts of data to the parallel file system.

Data Analytics and Machine Learning Prior work by Smith develops high-performance data analysis pipelines for various domains. These pipelines run on a variety of environments, including HPC and cloud, and involve big data processing with machine learning. They are representative of non-traditional workloads that ARC-IOtargets.

KINC: Knowledge Independent Network Construction (KINC) is a bioinformatics pipeline for constructing gene co-expression networks (GCNs) [80]. A GCN is a graph in which nodes are genes and edges exist between genes which are highly correlated according to some similarity measure such as Pearson or Spearman correlation. Unlike other GCN construction tools, KINC identifies clusters within each gene pair before computing the similarity scores. KINC employs a master/worker distributed approach to divide the work among many processes, where workers are accelerated with GPUs. The master assigns work dynamically such that each worker receives work blocks as fast as it can process them. Alternatively, KINC can use a chunk-and-merge approach in which many independent tasks are submitted with a static work distribution, each task produces a "chunk file" of the intermediate results, and a final merge process reads all of the chunk files and produces the final output files. This approach does not require MPI and may be advantageous on shared systems because

it consists of many small jobs instead of one large job; however it may suffer from workload imbalance since it does not use a dynamic work distribution.

Gene Orcale: Gene Oracle (GO) is a pipeline for identifying biologically significant biomarkers, or "candidate genes", from a high-dimensional gene expression dataset [81]. GO uses machine learning models to evaluate the "classification potential" of user-defined gene sets, and for those sets which perform the best it uses machine learning to select the most salient genes within each set. GO can use many classifiers — e.g., linear models, SVMs, random forests, neural networks.

TSPG: Transcriptome State Perturbation Generator (TSPG) is a deep learning pipeline for identifying genes with transitions between biological conditions [82]. TSPG uses a generative-adversarial network (GAN) to generate realistic gene expression perturbations between a source and target condition, such as normal kidney tissue to cancerous kidney tissue. The genes most strongly perturbed in the transition are used in downstream analyses. TSPG consists of several steps, including training a target model, training the GAN, and generating and visualizing the final perturbations. While a single TSPG model is trained with a fixed set of genes and a fixed target class, multiple TSPG models can be trained in parallel to cover different combinations of gene sets and target classes.

STRIDE: STRucture IDEntification (STRIDE) is a deep learning application developed to classify different types of molecular structures [83]. STRIDE adapts the PointNet architecture to identify local structure in molecular simulations without any feature engineering. Thus, STRIDE takes the 3D coordinates from a molecular dynamics simulation and predicts the class of each coordinate, making it a powerful tool for generating segmentations of large molecular simulations.

Intelligent River Intelligent River (NSF CNS-1126344) develops a river and water-resource sensor network on the entire 312 mile long Savannah River basin, and provides real-time access to essential environmental and hydrological parameters — e.g., water temperature, flow rate, turbidity, oxygen levels, pollutants — at appropriate temporal and spatial scale for managing resources. The data is critically needed to improve water resources management as demand increases for drinking water, hydroelectric power, recreation and industrial production. Battery-operated sensors are inserted into buoy systems anchored to the river floor and process the data and transmit it to Clemson's Palmetto Cluster for analysis and visualization. Challenges in this system include trust in the quality and accuracy of the data and transmission from remote sensors.

In 2018, Hurricane Florence caused $24 billion dollars in total damages and produced historic and extensive flooding in the Pee Dee River watershed resulting in 129 water res-

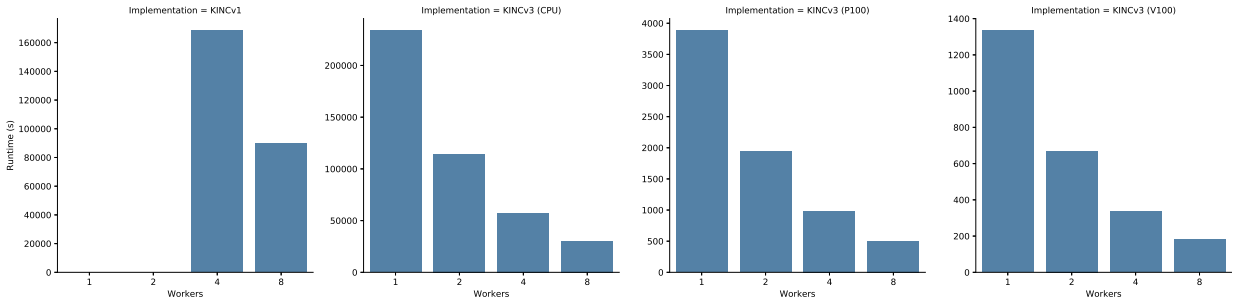
cues, over 1,000 assisted evacuations, displacement of nearly 8,000 people, and 233 roads closures [84]. During the planning phase, we will establish a connection with Dr. Christopher Post at Clemson and discuss the computational needs of expanding Intelligent River to the coastal waterways in the Pee Dee River watershed. Scaling the sensor network from a single river to multiple river systems and the coastal regions of South Carolina will result in massive amounts of real-time data. Thus, efficient data transmission and processing is critical in times of severe weather. ARC-IOwill enable fast and efficient processing of this data. The long-term goal of the collaboration with Dr. Post is to explore the use of the expanded sensor network's data to parameterize real-time watershed models to predict flooding during severe weather events. More accurate watershed modeling can enable faster and more accurate information for evacuation orders and routes.

Pedestrian Detection On average, 69,000 pedestrians are injured each year on US roadways and account for 15% of all US road fatalities in 2016 representing a 22% increase from 2007 [85]. To improve pedestrian safety, video and image data are sent from edge devices and cameras on roadways to the Cloud where it is used as input for machine learning models to monitor traffic flow and to detect pedestrians [86]. After analysis, the results are propagated back through edge devices to connected vehicles to inform and alert the driver. As more connected devices — e.g., cameras, sensors, vehicles — come into service, they compete for a fixed amount of existing bandwidth. This leads to a decrease in pedestrian safety as messages are delayed or the Cloud resource becomes overloaded. Reducing bandwidth requirements for Intelligent Transportation Devices using compression allows more connected devices to transmit data to a processing facility simultaneously. Moreover, ensuring QoS when the data reaches the facility ensures data is processed within the real-time constraints. Thus, enabling improvements in pedestrian safety by making decisions using more data sources and avoiding delays in analyzing and propagating the results to the connected devices.

6.3.5 Notions of Scale

This work considers scalability of applications in two contexts: (1) Traditional scalablility performance as the problem is weak or strong scaled. ARC-IOseeks to improve application I/O performance which will reduce runtime and I/O bottlenecks leading to better parallel efficiency when run on larger numbers of nodes. In addition, we will explore scalability of the I/O subsystem to the maximum number of concurrent readers and writers before I/O bandwidth drops below an minimum quality threshold. (2) Scalability of sensor applications that leverage Cloud and HPC systems for real-time analysis of data. For this, we consider

Figure 6.4: Strong scaling results for KINCv1, KINCv3 CPU, and KINCv3 GPU implementations. Each worker is a node. Note the y-axis scale is different for each subplot in order to emphasize the scalability of KINCv3.



the ability to increase the number of sensor devices in deployment without violating real-time QoS contracts.

6.3.6 Preliminary Scalability

KINC, as well as the other data analytics applications listed above, are *high-throughput* applications and as a result are highly scalable as shown in Figure 6.4. These applications operate on many independent data items which allow them to achieve extremely high parallelism on large-scale systems, as long as input data is sufficiently large. There is little to no synchronization between workers, the only communication overhead consists of the distribution and collection of work. Therefore we expect to be able to achieve an equally high level of scalability with these applications on the proposed system.

6.3.7 Background and Related Work

Runtime Systems Charm++ is a message driven parallel programming framework powered by an adaptive runtime system [16, 17]. It is utilized by several production scientific applications including NAMD [87], ChaNGa [88], and OpenAtom [89], among others. In addition to a C++ library interface, it can virtualize MPI applications via the Adaptive MPI (AMPI) layer [90], interoperate with MPI applications [91], provides a Python interface [92], and integrate with shared memory libraries such as OpenMP [93], Raja [94], and Kokkos [95].

Improving I/O Over the last two decades, a considerable amount of research has focused on I/O for high performance computing systems. I/O is a serious issue in large part due to

almost exclusive focus on processor and memory architecture in the 80's and 90's. Thus a "catch-up" period of I/O research was initiated by a series of meetings leading to funding programs in the area [96, 97, 98]. Among other things, these programs led to the development of a variety of file systems designed for parallel computing [99, 100, 101, 102, 103, 104] and programming environments [105]. These, in turn, provided the basis for the growth in BigData applications.

The ParalleX programming model [106], based on active messages [107], threads, and local synchronization has been shown to improve scaling an large scale parallel computation [108] was integrated with the OrangeFS parallel file system [109] and applied to the functional genomics application area [110, 111, 112]. This work forms the basis of the proposed project to more completely adapt the I/O subsystem.

Performance Modeling and Reconfigurability As heterogeneous HPC systems begin to incorporate different accelerators, programmers are faced with the task of choosing what type of accelerator to use to parallelize their algorithm. Further, many programmers want to know the benefit of implementing a parallel algorithm weighed against the time it takes to develop. Prior work developed an application-to-architecture taxonomy [113, 114, 115] to determine the best performing accelerator for different applications classes using machine learning, but is limited to a few select accelerators. Other works [116, 117, 118], predict the runtimes of parallel algorithms by leveraging different machine learning models. This provides programmers with performance predictions to more efficiently utilize available resources on heterogenous clusters.

Security and Provenance in HPC In computing, provenance is a comprehensive history of a data object from the point of generation to its current state. Data provenance is used to provide transparency of data objects; ensuring that data emanating from software systems has not been tampered with or corrupted during processing. Prior work [119, 120, 121, 122, 123, 124, 125] builds automatic provenance-aware systems for Linux and embedded system environments. HiFi [120] is a provenance framework for the linux kernel that consists of three components: provenance collector, provenance log, and provenance handler. The collector component records provenance information such as processes, IPC mechanisms, network and kernel activities in the OS kernel layer through provenance hooks placed in the kernel. ProvIoT [119] focuses on memory constrained embedded systems and uses dynamic instrumentation of application code at runtime and a graph database for further processing. PASS [124] intercepts system call information at the kernel level and stores this information in a kernel level database. LPS [122] is a lightweight provenance system for HPC environ-

ments. It consists of three components: a tracer, aggregate, and a builder component. The tracer collects provenance using kernel instrumentation in each node's OS. The aggregator component retrieves data from all of the respective nodes. The builder combines provenance collected and provides the appropriate data dependencies between all of the aggregated information.

All of the provenance systems mentioned above primarily focus on provenance collection of system calls in the OS. Our provenance framework focuses on interactions across heterogeneous devices in ARC-IOwhile generating provenance at the application and systems level.

Data Compression To address data movement and storage bottlenecks, data compression is an effective tool to reduce data size [126, 127, 128, 129, 130, 131, 132]. Two main forms of data compression exist: lossy and lossless. Lossless compression — e.g., gzip [133], fpzip [134], zstd [135] — compresses and decompresses with no loss in data fidelity, but are not well suited for floating-point data; achieving a compression ratio of 1–4× [136]. Lossy compression — e.g., SZ [127], ZFP [137] — enables larger compression ratios but at the expense of a user controlled loss in data fidelity. Key to the success of lossy compression algorithms is the quantification and understanding on the impact of compression on applications [138, 139, 140, 128, 141, 142].

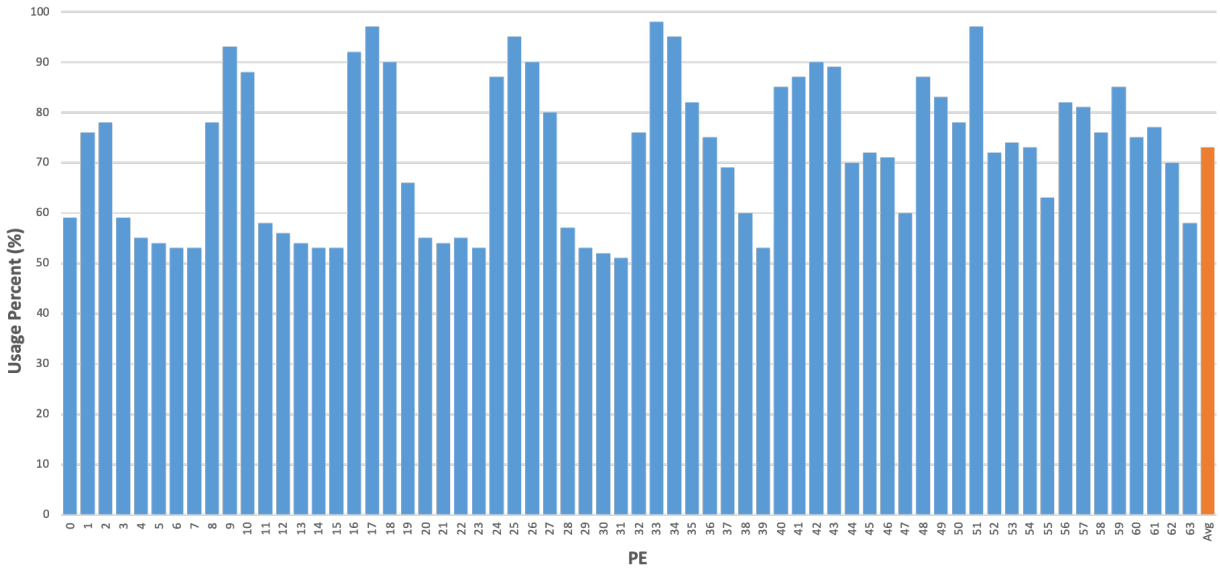
6.3.8 Research Task 1: System Monitoring and Control

While heterogeneous hardware offers an opportunity to improve application performance, leveraging it effectively is very difficult for individual application programmers. Our proposed solution, is to have the system handle this problem, interfacing with the hardware and orchestrating its allocation and supervision. This includes working with remote sensors, clouds, reconfigurable hardware, and accelerators.

This task prototypes the core functionality of using a dynamic runtime system. The planning work proceeds in three stages: measurement, adaptation, and integration. Thus, ARC-IOis able to dynamically monitor both an application's performance and the system's status and automatically assign resources to the program as it executes.

We use the Charm runtime system as a base layer prototype because it already includes measurement and adaptation. We plan to extend this functionality to work on a broader range of systems and applications and serve as a job coordinator [143] instead of a single runtime system for one application. Charm has been shown to run multiple jobs in the cloud effectively [10].

Figure 6.5: Before Adaptation (LB)

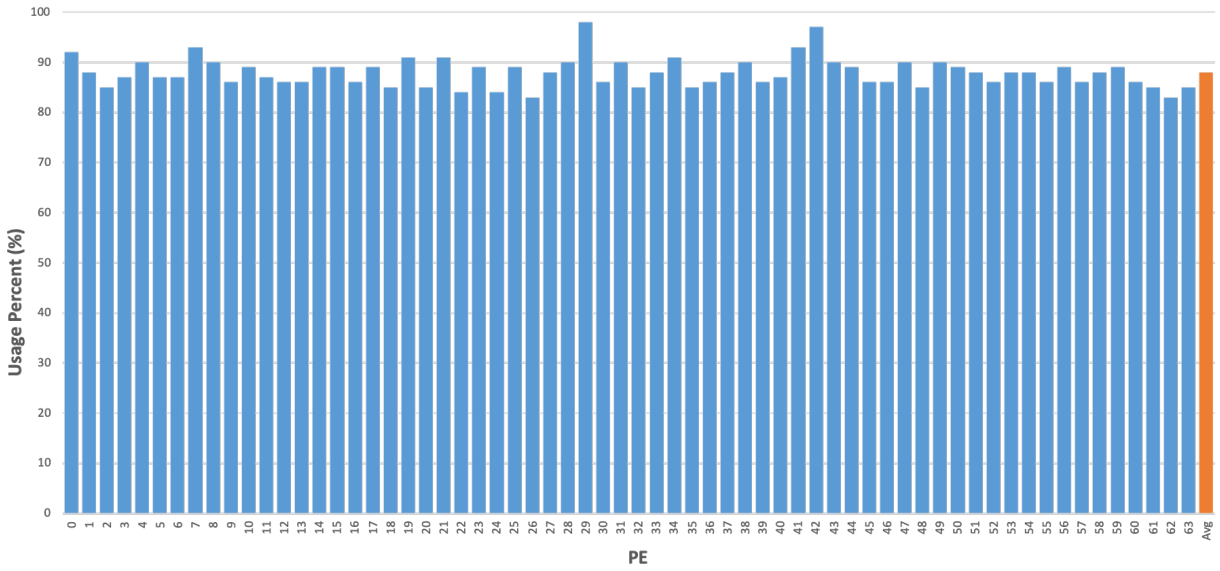


Measurement We will use PAPI [144] to investigate metrics which include cache misses, page faults, network counters, power consumption, etc. using a modified version of the Charm++ runtime system. Next, we determine the suitability of these metrics to determine system and application behavior. After collecting one (or more) metrics through a synthetic application, we categorize our test applications based on different behavior through a variety of phases in their application life times so that the system can have different optimization targets and thresholds.

Adaptation We integrate system controls in to the ARC-IOprototype. This has been successfully demonstrated in the past for targets such as load balance (Figs. 6.5, 6.6) and temperature and energy control for a single application [145]. We extend this work and bring these approaches to bear on other reconfigurable portions of the system — e.g. disabling cache lanes, reallocating I/O nodes. Our prototype serves as a test bed for new integrations and thus we are able to modify various system parameters — e.g, setting a power cap for GPUs executing application code, that are normally unavailable.

Integration We connect the measurement and adaptivity pieces in ARC-IOand create a single cohesive system prototype. We must ensure that the system collects not only the aforementioned metrics online for use by the allocation engine but also integrate this work with the other tasks, including our I/O, provenance, reconfigurability, and compression systems. We will design the system from inception with this idea in mind, and continue throughout

Figure 6.6: After Adaptation (LB)



the project. For instance, we need ensure we can collect application trace behavior for our provenance work in real time. However, at this stage we focus on the online feedback loop between observation and allocation and its integration with the broader components.

Successful completion of this task will yield the following outcomes: (1) realtime measurement of at least one non-load metric integrated into the ARC-IO prototype; (2) online adaptation of a system parameter (e.g. DVFS) during the lifetime of a single application; and (3) interfacing this system with at least one new application (e.g. KINC) that can leverage these features for improved performance.

Successful completion of this task enables ARC-IOto dynamically adapt based on phases of a single application. We plan to scale this system up to work with multiple workloads. During the planning phase we focus on a single metric and system characteristic — e.g., number of network bytes sent and node power cap via DVFS. Going forward, we will collect a variety of metrics to determine application behavior and modify various system parameters to meet our application execution demands and requirements. Our eventual aim is to have ARC-IObe a fully dynamic, introspective, and responsive system, capable of scheduling a wide range of general purpose computing jobs with a wide range of resource requirements.

6.3.9 Research Task 2: Adapting the I/O subsystem

Over the past few decades there have been numerous studies that characterize the I/O behavior of applications as well as research developments leading to different I/O architectures

meant to accommodate one or more application classes [146]. More recently the Darshan project [147] seeks to provide a practical tool for characterizing the I/O style of applications. Darshan has been used to isolate different I/O behaviors in applications on various architectures — e.g., local storage, point shared storage, distributed shared storage, burst buffer, intermediate storage and SSDs.

In addition, various parameters can have performance effects for different application behaviors including: caches, transfer block size, degree of I/O concurrency, data placement, etc. Application behavior is described by a number of classes including checkpointing, large-scale distributed, sequential access to contiguous file space, larg-escale distributed, sequential access to strided file space, independent sequential access, database file access and small, unaligned, random order access.

Additionally, other factors — e.g., data re-reading, blocking, buffering — have a big impact on how well the I/O subsystem can perform. Note that this characterization includes a diverse set of application types such as classic HPC applications, data analytics, and even high-throughput systems where many otherwise independent applications contribute to a common solution.

{The goal of this task is to develop the ability of a system to utilize performance metrics from an application to manipulate the I/O system to improve I/O performance for a range of application types. We propose to accomplish this by implementing an I/O interface and file system utilizing the features of Charm++ including active messages, threading, and global object addressing. These components will draw heavily from our experience with PVFS [99] and OrangeFS [148] parallel file system projects but will be significantly improved through the use of Charm++ technology.

During the planning phase, we propose to utilize Darshan to create characterizations of the applications (Section 6.3.4). Next, we evaluate different I/O architectures based on this characterization for the applications. These tests would utilize available I/O systems hardware and software including the OrangeFS file system developed by Ligon in circumstances where we need access to internals of the code.

During the planning phase, Task 2 will interact with the other task groups on overlapping issues. Task 2 incorporates the performance monitoring from Task 1 to establish metrics that could be gathered to drive adaptation of the I/O subsystem. Ligon and Robson will interact on developing active components of the I/O subsystem that utilize Charm++ features such as active messages to instantiate the components within the processes of the job. With Task 3, we plan to provide I/O information that indicates how data was managed and selected adaptations used during the execution of programs to annotate produced data sets. This mechanism allows the I/O subsystem to learn from one run of an application to the next

what adaptations are useful. We plan to aggregate this information based on application class to create heuristics to adapt the system. With Task 4, we plan to integrate the I/O subsystem with the system scheduler to allow jobs to request specific I/O resources and configurations when the job is launched. This is critical for minimizing contention for these resources and to communicate to the I/O subsystem the expected usage patterns. Also with this task we plan to consider specific hardware components that may be important for I/O including CPU cores, GPUs, or more likely FPGA structures specific to I/O. With Task 5, we plan to integrate lossy compression to mitigate the cost of expensive data movements. Error bounds will be incorporated from the user or extracted from the data's provenance.

The proposed goal for Task 2 in the full scale project is to design an integrated programming environment that includes adaptive high performance I/O and scales to exascale or better. We envision this environment is based on Charm++ due to its familiar programmer interface and highly dynamic nature. Unlike current environments that tend to have a static number of processes that communicate, Charm++ allows processes and threads to be created and destroyed as needed during the lifetime of a job. It utilizes a global address space to locate objects anywhere in the system, and active messages to create threads that invoke methods of those objects. This model of computation [149, 150, 110] also suggests local synchronization such as futures in order to minimize the use of global constructs such as barriers.

Until now, Charm++ has been used to implement applications. In this project, we intend to explore developing system services (I/O) that are tightly integrated with the application. We have conducted similar studies in past projects such as [109, 110] where we integrated OrangeFS, a parallel file system, with HPX [108], a programming environment similar to Charm++. The results of these projects demonstrated that there is a lot of promise in developing I/O for these environments. This project will take the next logical step, as alluded to in the final reports of these previous efforts, and construct the entire I/O stack using the programming environment thus enabling the I/O complete flexibility.

The proposed I/O system would be an object-based data store, which is to say that all data and metadata stored in it would be represented by objects stored in the underlying file system. Each object would have an object ID used to access it. Objects would store metadata (such as inodes) file data, directory data, etc. Programmer interfaces allow the collection of objects to be viewed as files. Basic features of the proposed I/O system include decoupled data and metadata, distributed metadata, dynamically instantiated data transfer threads and synchronization objects. A key feature is a composable I/O stack including hardware accelerators, burst buffers, local caches and selectable consistency semantics. Finally, other important features include integrated monitoring, capability based access control

and runtime adaptable management of dynamic features.

A major aspect of the design includes built-in counters and timers for gathering real-time performance data. Many of these are designed in to OrangeFS so this can be used as a guide both for implementation and for selecting important metrics. The other important aspect would be building components around policies so that alternative implementations can be selected at runtime. These policies would be selected by a policy manager where different component implementations and parameters are linked to policies which are themselves selected by mapping from performance metrics. This set would be ideal for using a machine learning system. We intend to explore this possibility during the latter portion of the project.

The project aims to work with both hardware and software aspects of the I/O stack. For hardware, we intend to work with Smith to develop hardware accelerators for I/O on FPGAs and potentially on GPUs. FPGA accelerator would tend towards a hardware engine for transferring data between storage and network interfaces largely bypassing software. Other approaches would work to merge multiple incoming data streams to storage, for use when compute nodes are synchronously writing to a shared file using MPI-IO and MPI File Views. Other hardware approaches would utilize available nodes to act as a burst buffer between the compute and I/O nodes. This can be in main memory, or NVRAM. Finally, using local memory as a cache can have big performance gains for some applications.

In software, there are also several options — e.g., the number of data transfer threads to use, the size of transfer buffers, use of data compression, choosing synchronization. The Charm++ environment provides active messages which are used to start threads on remote nodes. Using this for the I/O subsystem, data transfer threads are easily started on compute nodes (clients) and I/O nodes (servers). This allows the I/O to more fully utilize network throughput for better performance. Depending on the details of the I/O request, threads and compression are used to merge data from multiple sources, or to improve performance. Synchronization is required if multiple application tasks are reading or writing the same data, but may not be needed if each bit of data is only ever read or written by a single task.

Thus, the goal of this portion of the project is to be able to tune I/O for the best possible performance based on the behavior on the application characteristics and the system's workload. The primary objective is to identify metrics that lead us to tune the subsystem in a particular way so as to achieve the best performance. Then we can architect the subsystem to allow such tuning.

Successful completion of this task will yield the following outcomes: (1) A collection of metrics that indicate the ideal configuration of the I/O subsystem; (2) A mapping of these items to specific modification of the configuration that will improve performance; (3) A prototype demonstrating the gathering of metrics and changing of the I/O configuration,

and the resulting performance; and (4) A prototype of a new I/O subsystem based on an integrated and distributed model, unlike any currently available.

6.3.10 Research Task 3: Provenance in HPC

This task considers security implications which ARC-IOpresents. Security is a major issue in every modern system and one that must be addressed to ensure effective performance. One major area of focus is data trustworthiness [151]. Data produced from distributed streaming applications can be prone to anomalies since they are produced at a fast rate and sometimes at a large scale. There is a need to ensure the correctness of data propagating in this system. Provenance is a solution to ensuring data integrity. Data provenance is a comprehensive history of events that has occurred on a data object from inception to its current state. Represented as a directed acyclic graph, provenance has applications in digital forensics, system auditing, ensuring reproducibility in scientific experiments and anomaly detection. This allows us to model data workflow while further understanding data dependencies that might exist in our systems.

The work in this planning phase will focus on developing a functional prototype of the provenance-based system which provides an end-to-end provenance capture of all of the working components of our HPC system. This task is broken down into the following:

Integrate data provenance into the HPC framework. Provenance systems have been proposed for memory constrained embedded systems [119] and linux environments [124, 121, 120], but do not provide a cohesive infrastructure for HPC environment. We propose to develop an automatic lightweight end-to-end provenance mechanism which allows dynamic instrumentation of our HPC framework using system and application layer trace events. Fine-grained provenance information is generated from trace data derived from the instrumentation of the operating system kernel and application code at the application level. We map this information to a provenance graph using a provenance-trace model that represents relationships between entities contained in the system using w3c prov, a standard for representing provenance data.

Identify appropriate trace for provenance capture. Deciding what trace data is appropriate for provenance generation is an important part of ensuring the efficiency of our approach. From user-defined trace to automated instrumentation, we will evaluate the following scenarios: (1)The outcome of an application-defined trace collection versus an automated trace collection approach; (2) The use of application level instrumentation versus

system-level instrumentation; and (3) Continuous provenance generation versus provenance summarization. Provenance generates an overwhelming amount of data which is sometimes larger than the data it describes. We will explore the trade-offs of continuous provenance generation over using approach such as pruning, compression, or graph-based summarization.

Develop a visualization framework using provenance data. Visualization is used to uncover data insights by discovering interesting workflow patterns in the data by making complex data dependencies in graphs easily understandable. We plan to develop a visualization tool to view the system interactions that exist in the framework. ARC-IOleverages this tool to understand causal dependencies. The proposed system will be robust, supporting subset and fine-grained views of the data.

Proposed Work The goal of this task is to integrate provenance into the HPC system. As such, provenance data generated is leveraged to provide anomaly detection of advanced persistent threats such as zero-day attacks. Most security systems of these attacks utilize signatures of known malware or techniques that do not properly correlate the long time system behavior which are mostly prone to evasion. We also extend the applications of our provenance environment to ensuring a deterministic execution of workloads while our system executes under normal running conditions. This provides reproducibility and also serves as a means of verifying the correctness. Reproducibility can be achieved by examining the provenance graphs generated to observe the deterministic execution configuration of ARC-IOworkloads. This information can be used to generate recurring configurations of the system workloads.

6.3.11 Research Task 4: Performance Modeling and Reconfigurability

As we move toward large-scale heterogenous HPC systems with many different accelerators, choosing the appropriate accelerator for a given application is a non-trivial task. Choosing the optimal accelerator and developing the algorithm to perform optimally requires understanding of the underlying architecture for each type of hardware (CPU, FPGA, GPU, TPU, etc). One purpose of this task is to assist with ARC-IOnode assignment through the use of performance models and algorithm agnostic hardware prediction. Further work aims to assist users in programming for the proposed system and determine the optimal methods of developing applications for use in ARC-IO.

Prior work by Smith focuses on performance modeling of applications on different hardware [114, 115]. The purpose of this work was to use a random forest classifier to predict

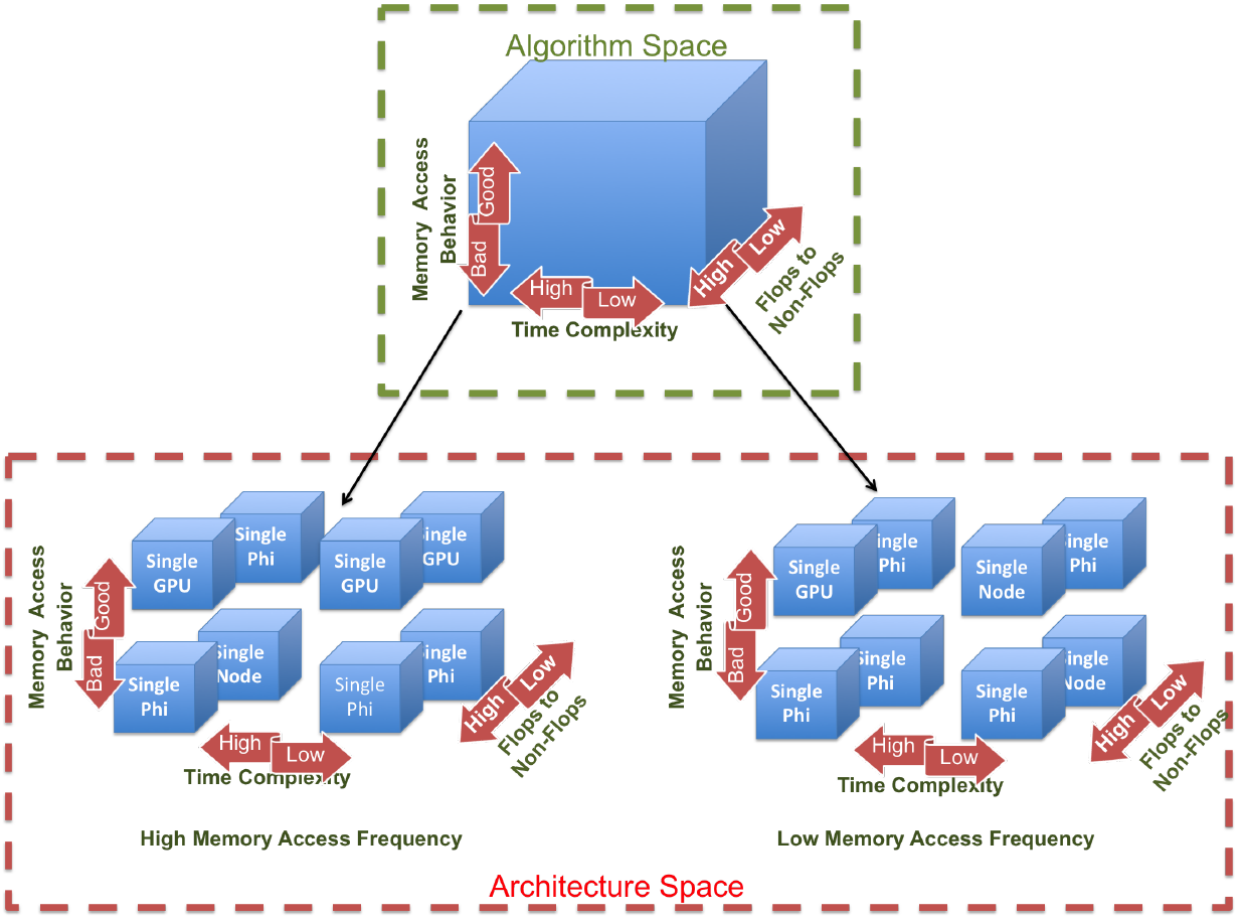


Figure 6.7: Application to Architecture mapping *Tesseract*

the highest performing hardware given application specific characteristics (memory access frequency and behavior, time complexity, and FLOPs to non-FLOP ratio). The developed taxonomy, named *Tesseract*, works as a four-dimensional application to architecture mapping shown in Figure 6.7.

This taxonomy focused on single node CPU, GPU, and Intel Xeon Phi Coprocessors. However, FPGAs are becoming more prominent in HPC systems as accelerators due to their low power usage and reconfigurability. This task aims to extend *Tesseract* to include FPGAs and provide a more robust classification framework for accelerator prediction. The OpenDwarfs benchmark suite [152] will be used to develop the improved taxonomy as it provides multiple different applications designed to represent classes of programs with differing compute and communication characteristics. Completing this provides ARC-IOwith a framework for selecting the best available node when provisioning a running application.

Much of the focus of accelerators in HPC is on the compute performance and efficiency when attempting to improve an algorithm. However, these devices are also used to assist

with multiple I/O tasks, such as when compressing and decompressing data. Task 5 focuses on the use of lossy and lossless compression in applications and ARC-IO, Task 4 focuses on profiling the performance of these algorithms on different hardware. waveSZ [153] and GhostSZ [154] provide FPGA implementations of the SZ [127] using Vivado HLS. The authors of SZ provide both OpenCL and CUDA implementations of the algorithm for use with GPUs. Xilinx offers a GZip library [155] for FPGAs and has been tested and implemented on Amazon Web Services servers for lossless compression. Furthermore, extensive work has been completed toward implementing lossless compressors on GPUs as shown by [156, 157, 158, 159, 160]. This work will perform a more exhaustive search of available accelerated compression algorithms to determine the feasibility of usage within ARC-IO.

Completion of the planning phase provides a performance modeling framework that predicts the best performing hardware given an application's static, pre-runtime characteristics. To extend this work to be applicable in ARC-IO, a modeling framework that assesses the current needs and characteristics of a running algorithm would better predict the appropriate accelerator to allocate to a given task. As ARC-IOis runtime adaptive, a finer grained model for performance prediction utilizing runtime metrics of an application allows *Tesseract* to adapt with the current I/O and compute workload. *Tesseract* initially assumes nothing about the application and learns its performance characteristics overtime. This research will use the performance and I/O metrics from Research Tasks 1 and 2, respectively, to develop *Tesseract* into a dynamic framework that provides application agnostic hardware predictions for ARC-IO.

Further research for this task considers a mixed qualitative and quantitative analysis of the work required to take an existing application and run it on ARC-IO. During development of the adaptive system and testing the applications in Section 6.3.4, we will consider how to achieve the best performance for each application. This requires a classification of the tested algorithms as they apply to ARC-IOin order to develop a taxonomy for future applications running on the system. This research will run the native algorithms on the proposed system and compare them to implementations that are optimized for the adaptive environment. During this development, we plan to perform a study of the best practices for ARC-IOand develop a guide demonstrating how to achieve optimal performance for the different algorithm classifications. We also plan to determine user-level feedback, in the form of pragmas or configuration files, during development that will allow the programmer to inform ARC-IOand the underlying models of application specific insights to improve performance.

Successful completion of this task will yield the following outcomes: (1) Extension of the performance modeling work, *Tesseract*, to include FPGA predictions; (2) A thorough

analysis and selection of accelerated compression algorithms for heterogeneous hardware.

6.3.12 Research Task 5: Compressed Data Transfer

In large-scale systems, the rate of computation is orders-of-magnitude more than the rate of data transfer. This is true for transfers inside and outside of the HPC system. This large disparity has resulted in data movement being a key bottleneck on current systems and will become exacerbated on future systems [161, 162]. ARC-IOseeks to improve I/O scalabilty and performance of applications, but relies on adapting nodes. If all nodes are currently allocated, ARC-IOwill leverage data compression to reduce the volume of data transferred. Moreover, getting data into an HPC system can be expensive. We plan to explore the use of lossy and lossless data compression to improve inter and intra-system transfer bandwidth.

To enable ARC-IOto take advantage of multiple lossy and lossless compression algorithms we will leverage libPressio developed by Calhoun's group [163]. We will extend libPressio to incorporate accelerated versions of lossy compression algorithms such as SZ [153, 164] and ZFP [165, 166].

Data analytics and machine learning applications are two growing classes of applications run on large HPC systems. These applications involve ingesting and processing massive amounts of data. To accelerate these applications, ingesting compressed data would place less burden on the file system. We plan to expand of PI Calhoun's prior work that investigates the feasibility of using lossy data compression in the context of pedestrian detection [142] to further improve compression performance by leveraging different compression algorithms and multiple error tolerances. When using multiple tolerances we will scale the tolerance based on the output of the pedestrian detection model. Thus, if we detect pedestrians we will reduce the compression error in the data to ensure they are tracked correctly. Otherwise, the compression level will increase mitigating the impact on bandwidth use. Over the course of this study, we will explore generalizing this technique to other streaming applications such as Intelligent River. Finally, we will investigate the use of lossy compressed input data for the other data analytics applications in Section 6.3.4 to establish bounds on what are viable tolerances.

Based on the work during the preliminary work, this task will focus on the following: (1) seek to establish a formal link between lossy compression error and data analytics enabling the *a priori* selection of error bounds; and (2) create a framework that will convert any lossy compression algorithm into a fixed-ratio compressor subject to multiple user defined metrics.

Before lossy compress data can be used inside of a workflow, quantifying and understanding the impact of compression error is required. The current state-of-the-practice relies on

ad-hoc trial-and-error testing with a human-in-the-loop to determine if the workflow's results are acceptable [167, 168, 138, 139, 141]. Analyzing the algorithm and how compression error impacts, propagates, or diminishes helps to eliminate the ad-hoc trail-and-error necessity [128, 169, 170]. We will quantify how lossy compression error impacts different layer types — e.g., convolution, pooling — in Deep Learning applications to construct an accuracy model parameterized by types and frequency of layers, their ordering, and compression error bound. The output of this model yields a confidence level for the model's output. Finally, we will explore the impact of model training on lossy compressed data on inference accuracy and training time.

Most compression algorithms seek to achieve the highest compression ratios. Thus, two uncompressed files of the same size can compress to different sizes even when the same compressor and configuration are used. This complicates users ability estimate storage costs. Fixed-rate compressors compress each value to the same fixed-sized code word regardless of how often it is used resulting in small compression ratios [171] and in the case of lossy compression algorithms high amounts of error [172]. To address this issue, our prior work, FRaZ [172], presents a first step approach to construction of a fixed-ratio compressor. FRaZ solves a one dimensional optimization problem for the compression error bound that will yield a target compression ratio within a margin of error. Results show that FRaZ yields better accuracy than fixed-rate methods at the same compression ratio. During the full work, we plan to investigate FRaZ's ability to efficiently solve multidimensional optimization problems. This enables FRaZ to compress to a fixed-ratio while maintains a problem specific accuracy constraints — e.g. preservation of mean and first derivative when using an absolute error bound. Thus, we can ensure that the data meets the application's accuracy requirements and storage requirements.

Successful completion of this task will yield the following outcomes: (1) an accuracy model that accounts for lossy compressed input data to prescribe a confidence in a ML model's result; and (2) a fixed-ratio lossy compression that is able to compress subject to multiple user defined constraints.

6.3.13 Evaluation

Research Task 6.1 We evaluate the success of the ARC-IO prototype system based on the observed speedups of at least one focus applications, KINC, GO, TSPG, and STRIDE, as well as on the performance of at least one preexisting Charm++ application, e.g. NAMD [87], ChaNGa [88], and OpenAtom [89]. We evaluate the overhead of gathering several potential measurement metrics in real-time and the suitability of these various metrics for

capturing application and system behavior.

Completion of this task enables us to select a minimal number of best performing and lowest overhead measurements and use them to improve application execution times.

Research Task 6.2 Task 2 is evaluated by determining a few application metrics from Task 1 that indicate some change in the I/O subsystem results in a performance benefit. Darshan [147] is used to look for such metrics and the size of I/O requests, the sequential nature of such requests, and whether such requests are contiguous. I/O system configurations such as turning caches on and off, adjusting the number of I/O threads, etc. as previously discussed, can be used for manipulating I/O performance. We will use the performance data to establish rules for when adaption in the I/O subsystem is needed.

Completion of this task provides data we would use to evaluate this task in the context of the long term project. During that project we would use these results to evaluate adaptation rules both for smaller programs, and the full applications discussed in Section 6.3.4. These will form our primary evaluation.

Research Task 6.3 When evaluating the integration of end-to-end provenance in HPC applications and ARC-IOwe concern ourselves with two key metrics: runtime and storage costs. We will compute these metrics for both the system and application level integration and compute the slowdown due to provenance and the increase in the memory footprint. Finally, we will visualize the log files and verify the accuracy.

Completion of this task enables us to properly evaluate the feasibility of our approach in providing an efficient and lightweight provenance system with little overhead.

Research Task 6.4 Successful extension of *Tesseract* will be evaluated by the accuracy of hardware prediction for multiple algorithms when compared to measured runtimes on different accelerators. The survey of accelerated compressors will be evaluated by the ability to replicate the reference papers' results when implementing the algorithms in an HPC context. As some of these applications may require modification to work with existing applications, speedup and similar compression ratios will verify feasibility of ARC-IO.

Completion of this task enables us to select the best available hardware for both compute and I/O tasks in ARC-IOto assure the system has deterministic and efficient node provisioning. It also allows us to determine which accelerated compression algorithms are appropriate for use.

Research Task 6.5 We evaluate the effectiveness of the libPressio extension based on our ability to integrate all the target accelerated compression algorithms. We will evaluate the integration of lossy compressed input data into the data analytics workflows by the ability to achieve the correct answer within some fixed, problem dependent, margin of error. Thus, for each of our test applications we will have a heuristic to select compression error tolerances. When utilizing multiple tolerances for the pedestrian use case, we will determine success by our ability to maintain or improve detection accuracy. In addition, we will quantify the bandwidth reduction and compare to the single compression tolerance case.

Completion of this task enables us to transparently leverage multiple compression algorithms that target various hardware in Tasks 2–5. Establishing a heuristic for selecting compression error tolerances sets us up to establish formal relations between application characteristics and compression error propagation, accumulation, and removal in the full project.

6.3.14 Broader Impacts

Broader Impacts of this work involve creation of ARC-IOthat enables HPC systems to adapt their configuration to remove I/O issues and improve scalability for a diverse collection of HPC applications. Thus, the throughput of their workloads is increased, facilitating the scientific contributions in their respective areas. Furthermore, establishing end-to-end provenance provides a ledger for reproducibility of system workloads and a means of providing a deterministic configurations of workloads for detecting anomalous system events. Furthermore, adding in provenance in HPC allows trust when composing systems software and applications from many 3rd party modules because dependencies and operations are strictly tracked and can be inspected for accuracy and fraud detection.

CHAPTER 7: CONCLUSION

One of the greateset challenges facing modern supercomputing is the lag of improvements in networking and memory relative to gains in computational efficiency. We have formulated several ways to deal with the increasing complexity of machines, as more processing power is packed on a single chip or node, with decreasing returns on single core performance and a proliferation of multicore chips along with other specialized accelerator technologies. This is accompanied by the current trend of dwindling network performance, relative to increases in single node performance, often via novel accelerators or other technologies. In this work, we have broadly surveyed the communication performance of large parallel applications and developed or expanded upon various techniques that can be utilized to improve their performance. These techniques include:

- spreading
- prioritization of remote work
- staggering
- overhead reduction
- contention reduction
- increased responsiveness

These techniques build on functionalities available in parallel-programming libraries such as OpenMP, MPI, and Charm++ in order to generate new degrees of freedom that can be leveraged to impact communication behavior and application performance.

So far, we have studied some of these techniques in the form of mini-applications e.g. stencil and other micro-benchmarks, and we plan to continue this work by looking at larger production scientific applications mentioned in Chapter 6.1. We have also seen the importance of BigSim + TraceR simulations in predicting the future performance of applications on the next generation of machines and how our suggestions will help the current wave of applications adapt to a regime in which compute is ever more plentiful and communication performance is even more unbalanced.

We've seen performance improvements on the order of 4x relative to the true baseline and twenty percent faster than the best known technique. These results suggest that spreading communication over time, instead of concentrating it at the end of an iteration or periodically throughout an iteration, can improve the performance of large parallel applications. We plan to study this improvement further, analyze, and quantify it.

We have also exmained the broader applicability of these techniques, focusing in Chapter 5 on internode as well as intranode communication between the CPU and GPU. We applied the

general idea of seamlessly overlapping communiation and computation phases by leveraging native overlap and overdecomposition in Charm++ runtime that allows us to constantly inject small messages onto the network, as well as the connection between devices on the same physical node.

Modern systems require a variety of (sometimes non-obvious) tradeoffs, both in their construction but especially in their use by applications desiring to achieve the highest possible performance. We have tested our approahces on a variety of Top500 systems, including Summit, and observered good scalability. Using current networks efficiently requires tight integration and management of memory movement (or data movement in general), caching described by a complicated performance model. We have begun to embody these techniques in a production scale adaptive runtime system (Charm++), where we have demonstrated the power of combining mutiple methods, e.g. overdecomposition and spreading, to create a new optimization space and improve application performance and ability to adapt to the requirements of new systems, networks, etc. However, this new degree of freedom is a double-edged sword as this is yet another parameter that must be tuned by the end user to achieve ideal performance. In the future, work in integrating auto-tuning frameworks into adaptive runtime systems, combined with sensible default settings, will allow programmers to achieve this higher levels of performance without exhaustive benchmarking. This idea demonstrates the utlimate utility of a system like Charm++, where efforts spent in optimizing the runtime system can be leveraged across all applications automatically. Keeping in mind programmer productivity, our ultimate goal is to annotate code without re-writing, allowing to inter-operate and transparantely utilize runtime system features. This work exemplifies a broader trend in HPC of combining approaches to leverage their respective strengths, e.g. MPI+X. In this work we also witness the power of utilizing (parallel) programming frameworks with notions of asynchrony builtin. However, this approach is one piece of larger optimization problem, and derives most of its power from integrating with and leveraging other techniques. This syngery unlocks further optimization opportunities, and so on. In general, this further supports the importance of a multi-pronged research approach as modern problems require a variety of often times competing solutions to tackle the some of the world's most pressing challenges.

It is our assertion that these techniques are broadly applicable. We are confident that further study, as mentioned in Chapter 4 and 6.2, will demonstrate this assumption. Another advantage, aside from their broad applicability, is their ease of use. OpenMP is a wide-spread, well-adopted, and broadly-supported framework that easily facilitates improved performance in a wide range of applications. Finally, we intend to demonstrate the effectiveness of these and other techniques in other programming systems, namely MPI, potentially through

Adaptive MPI (AMPI). Other contributions so far include the formalization of the folk theorem of maximal performance improvement.

While generally applicable to communication bound applications, our techniques will do little to help alleviate the problems experienced by compute bound applications. However, as discussed in the introduction, it is communication and not computation that is becoming more expensive over time and therefore we expect more applications will move from one category to the other and therefore benefit from our work, especially with future generations of supercomputers and other machines.

Moving further into the future, smart runtimes, even extending to the system level and tying deeply into the Operating System, will start to become a requirement. These smart layers will utilize the unique view the runtime system has of an application's lifetime to manage all resources, not just compute units (CPU, GPU, FPGA) and communication (local and remote), but also IO resources, tracing, etc. This includes a new regime of applications with different requirements that current systems are not engineered to handle.

It is our hope that these ideas, even apart from the programming systems they are embodied in, will inform and motivate the broader community about issues related to communication performance optimization and encourage the adoption of runtimes which support these techniques, ultimately leading to improved (or total) communication computation overlap (or balance) and increased parallel application performance and from there to more impactful scientific discoveries.

CHAPTER 8: REFERENCES

[1] G. E. Moore et al., "Cramming more components onto integrated circuits," 1965.

[2] S. Plimpton, "Molecular dynamics: Looking ahead to exascale," May 2019.

[3] C. Smith, "Load balancing on many-core and accelerated systems," 2019.

[4] "Top500 supercomputing sites," http://top500.org, 2013.

[5] N. Jain, "Optimization of communication intensive applications on HPC networks," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2016.

[6] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computational Science & Engineering*, vol. 5, no. 1, January-March 1998.

[7] "MPI: A Message Passing Interface Standard," in *MPI Forum*, http://www.mpi-forum.org/.

[8] R. Thakur, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, and J. L. Träff, "Mpi at exascale," *Procceedings of SciDAC*, vol. 2, pp. 14–35, 2010.

[9] M. Si, A. J. Peña, P. Balaji, M. Takagi, and Y. Ishikawa, "Mt-mpi: multithreaded mpi for many-core environments," in *Proceedings of the 28th ACM international conference on Supercomputing*. ACM, 2014, pp. 125–134.

[10] A. Gupta, O. Sarood, L. V. Kale, and D. Milojicic, "Improving hpc application performance in cloud through dynamic load balancing," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 2013, pp. 402–409.

[11] 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.

[12] C. D. Carothers, D. Bauer, and S. Pearce, "ROSS: A high-performance, low-memory, modular Time Warp system," *Journal of Parallel and Distributed Computing*, vol. 62, no. 11, pp. 1648–1669, 2002.

[13] Argonne National Laboratory and Rensselaer Polytechnic Institute, "CODES Discrete-event Simulation Framework," October 2019. [Online]. Available: https://github.com/codes-org/codes

[14] 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.

[15] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The spack package manager: bringing order to hpc software chaos," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12.

[16] L. Kale, B. Acun, S. Bak, A. Becker, M. Bhandarkar, N. Bhat, A. Bhatele, E. Bohm, C. Bordage, R. Brunner, R. Buch, S. Chakravorty, K. Chandrasekar, J. Choi, M. Denardo, J. DeSouza, M. Diener, H. Dokania, I. Dooley, W. Fenton, J. Galvez, F. Gioachin, A. Gupta, G. Gupta, M. Gupta, A. Gursoy, V. Harsh, F. Hu, C. Huang, N. Jagathesan, N. Jain, P. Jetley, P. Jindal, R. Kanakagiri, G. Koenig, S. Krishnan, S. Kumar, D. Kunzman, M. Lang, A. Langer, O. Lawlor, C. Wai Lee, J. Lifflander, K. Mahesh, C. Mendes, H. Menon, C. Mei, E. Meneses, E. Mikida, P. Miller, R. Mokos, V. Narayanan, X. Ni, K. Nomura, S. Paranjpye, P. Ramachandran, B. Ramkumar, E. Ramos, M. Robson, N. Saboo, V. Saletore, O. Sarood, K. Senthil, N. Shah, W. Shu, A. B. Sinha, Y. Sun, Z. Sura, E. Totoni, K. Varadarajan, R. Venkataraman, J. Wang, L. Wesolowski, S. White, T. Wilmarth, J. Wright, J. Yelon, and G. Zheng, "The Charm++ Parallel Programming System," Aug 2019. [Online]. Available: https://charm.cs.illinois.edu

[17] L. V. Kale and G. Zheng, "Chapter 1: The Charm++ Programming Model," in *Parallel Science and Engineering Applications: The Charm++ Approach*, 1st ed., L. V. Kale and A. Bhatele, Eds. Boca Raton, FL, USA: CRC Press, Inc., 2013, ch. 1, pp. 1–16.

[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] V. Kale, H. Menon, and K. Senthil, "Adaptive loop scheduling with charm++ to improve performance of scientific applications."

[20] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkins-Diehr, "Xsede: Accelerating scientific discovery," *Computing in Science & Engineering*, vol. 16, no. 5, pp. 62–74, Sept.-Oct. 2014. [Online]. Available: doi.ieeecomputersociety.org/10.1109/MCSE.2014.80

[21] K. Chandrasekar and L. V. Kale, "Optimizing molecular dynamics and stencil mini-applications for intel's knights landing," 2016. [Online]. Available: https://charm.cs.illinois.edu/media/16-16

[22] S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke, "Ibm power9 processor architecture," *IEEE Micro*, vol. 37, no. 2, pp. 40–51, 2017.

[23] S. Kumar, A. R. Mamidala, D. A. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger et al., "Pami: A parallel active message interface for the blue gene/q supercomputer," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 763–773.

[24] J. C. Phillips, "What you should know about namd and charm++ but were hoping to ignore," in *Proceedings of the Practice and Experience on Advanced Research Computing*, 2018, pp. 1–6.

[25] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[26] L. Kalé and A. Sinha, "Projections : A scalable performance tool," in *Parallel Systems Fair, International Parallel Processing Sympos ium*, Apr. 1993, pp. 108–114.

[27] 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.

[28] R. Preissl, J. Shalf, N. Wichmann, S. Ethier, B. Long, and A. Koniges, "Multithreaded global address space communication techniques for gyrokinetic fusion applications on ultra-scale platforms," in *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2011, pp. 1–11.

[29] P.-Y. Calland, J. Dongarra, and Y. Robert, "Tiling on systems with communication/computation overlap," *Concurrency: Practice and Experience*, vol. 11, no. 3, pp. 139–153, 1999.

[30] V. Subotic, J. C. Sancho, J. Labarta, and M. Valero, "A simulation framework to automatically analyze the communication-computation overlap in scientific applications," in *2010 IEEE International Conference on Cluster Computing*. IEEE, 2010, pp. 275–283.

[31] T. Hoefler, J. M. Squyres, W. Rehm, and A. Lumsdaine, "A case for non-blocking collective operations," in *International Symposium on Parallel and Distributed Processing and Applications*. Springer, 2006, pp. 155–164.

[32] E. Castillo, N. Jain, M. Casas, M. Moreto, M. Schulz, R. Beivide, M. Valero, and A. Bhatele, "Optimizing computation-communication overlap in asynchronous task-based programs," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 380–391.

[33] L. V. Kale, D. M. Kunzman, and L. Wesolowski, "Accelerator Support in the Charm++ Parallel Programming Model," in *Scientific Computing with Multicore and Accelerators*, J. Kurzak, D. A. Bader, and J. Dongarra, Eds. CRC Press, Taylor & Francis Group, 2011, pp. 393–412.

[34] J. A. Board, L. V. Kalé, K. Schulten, R. Skeel, and T. Schlick, "Modeling biomolecules: Larger scales, longer durations," *IEEE Computational Science and Engineering*, vol. 1, no. 4, 1994.

[35] N. Jain, E. Bohm, E. Mikida, S. Mandal, M. Kim, P. Jindal, Q. Li, S. Ismail-Beigi, G. Martyna, and L. Kale, "Openatom: Scalable ab-initio molecular dynamics with diverse capabilities," in *International Supercomputing Conference*, ser. ISC HPC '16 (to appear), 2016.

[36] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn, "Massively parallel cosmological simulations with ChaNGa," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[37] B. Acun, N. Jain, A. Bhatele, M. Mubarak, C. D. Carothers, and L. V. Kale, "Preliminary evaluation of a parallel trace replay tool for hpc network simulations," in *Workshop on Parallel and Distributed Agent-Based Simulations*, ser. PADABS, EURO-PAR, Aug. 2015.

[38] J. Cope, N. Liu, S. Lang, P. Carns, C. Carothers, and R. Ross, "Codes: Enabling co-design of multilayer exascale storage architectures," in *Proceedings of the Workshop on Emerging Supercomputing Technologies*, 2011.

[39] M. Collaboration, "MIMD Lattice Computation (MILC) Collaboration Home Page," http://www.physics.indiana.edu/~sg/milc.html.

[40] Pittsburgh Supercomputing Center, "Bridges." [Online]. Available: https://www.psc.edu/bridges

[41] San Diego Supercomputer Center, "Comet." [Online]. Available: https://www.sdsc.edu/support/user_guides/comet.html

[42] National Center for Supercomputing Applications, "Blue Waters project," http://www.ncsa.illinois.edu/BlueWaters/.

[43] Texas Advanced Computing Center, "Stampede 2." [Online]. Available: https://portal.tacc.utexas.edu/user-guides/stampede2

[44] Oak Ridge National Laboratory, "Summit." [Online]. Available: https://www.olcf.ornl.gov/for-users/system-user-guides/summit/

[45] A. Bhatele, N. Jain, Y. Livnat, V. Pascucci, and P.-T. Bremer, "Analyzing network health and congestion in dragonfly-based supercomputers," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 93–102.

[46] L. Wesolowski, R. Venkataraman, A. Gupta, J.-S. Yeom, K. Bisset, Y. Sun, P. Jetley, T. R. Quinn, and L. V. Kale, "TRAM: Optimizing Fine-grained Communication with Topological Routing and Aggregation of Messages," in *Proceedings of the International Conference on Parallel Processing*, ser. ICPP '14, Minneapolis, MN, September 2014.

[47] L. V. Kale and S. Krishnan, "Charm++: a portable concurrent object oriented system based on c++," in *ACM Sigplan Notices*, vol. 28, no. 10. ACM, 1993, pp. 91–108.

[48] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 1-3, pp. 66–73, 2010.

[49] D. Kirk et al., "Nvidia cuda software and gpu parallel computing architecture."

[50] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," in *European Conference on Parallel Processing*. Springer, 2009, pp. 863–874.

[51] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[52] P. Pandit and R. Govindarajan, "Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2544137.2544163 pp. 273:273–273:283.

[53] M. Boyer, K. Skadron, S. Che, and N. Jayasena, "Load balancing in a changing world: Dealing with heterogeneity and performance variability," in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2482767.2482794 pp. 21:1–21:10.

[54] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao, "Dynamic load balancing on single-and multi-gpu systems," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–12.

[55] 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.

[56] L. Wesolowski, "An application programming interface for general purpose graphics processing units in an asynchronous runtime system," M.S. thesis, Dept. of Computer Science, University of Illinois, 2008, http://charm.cs.uiuc.edu/papers/LukaszMSThesis08.shtml.

[57] D. Kunzman, "Runtime support for object-based message-driven parallel applications on heterogeneous clusters," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2012, http://charm.cs.uiuc.edu/media/12-45/.

[58] A. Grama, V. Kumar, and A. Sameh, "Scalable parallel formulations of the barnes-hut method for n-body simulations," in *Proceedings of Supercomputing '94*, 1994, pp. 439–448.

[59] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, August 2013.

[60] Sandia National Laboratory, "SST DUMPI trace library," October 2019. [Online]. Available: https://github.com/sstsimulator/sst-dumpi

[61] L. V. Kale, G. Zheng, C. W. Lee, and S. Kumar, "Scaling applications to massively parallel machines using projections performance analysis tool," in *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, vol. 22, no. 3, February 2006, pp. 347–358.

[62] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, "Efficient inter-node mpi communication using gpudirect rdma for infiniband clusters with nvidia gpus," in *2013 42nd International Conference on Parallel Processing*. IEEE, 2013, pp. 80–89.

[63] A. Gerbessiotis and L. Valiant, "Direct bulk-synchronous parallel algorithms," *Journal of Parallel and Distributed Computing*, vol. 22, no. 2, pp. 251 – 267, 1994. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731584710859

[64] R. H. Bisseling and W. F. McColl, "Scientific computing on bulk synchronous parallel architectures," in *IFIP Congress*, 1994.

[65] T. Cheatham, A. Fahmy, D. Stefanescu, and L. Valiant, *Bulk Synchronous Parallel Computing — A Paradigm for Transportable Software*. Boston, MA: Springer US, 1996, pp. 61–76. [Online]. Available: https://doi.org/10.1007/978-1-4615-4123-3_4

[66] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang, "Enabling fpgas in the cloud," https://dl.acm.org/doi/abs/10.1145/2597917.2597929, 2014. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/2597917.2597929

[67] F. Huot, Y.-F. Chen, R. Clapp, C. Boneti, and J. Anderson, "High-resolution imaging on tpus," *arXiv:1912.08063 [physics]*, Dec 2019, arXiv: 1912.08063. [Online]. Available: http://arxiv.org/abs/1912.08063

[68] S. Choi, M. Shahbaz, B. Prabhakar, and M. Rosenblum, "λ-nic: Interactive serverless compute on programmable smartnics," *arXiv:1909.11958 [cs]*, Sep 2019, arXiv: 1909.11958. [Online]. Available: http://arxiv.org/abs/1909.11958

[69] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," 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. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.18 pp. 1–11.

[70] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao, "A multiplatform study of I/O behavior on petascale supercomputers," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2749246.2749269 pp. 33–44.

[71] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of blue waters," in *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '14. Washington, DC, USA: IEEE Computer Society, 2014. [Online]. Available: http://dx.doi.org/10.1109/DSN.2014.62 pp. 610–621.

[72] J. Ousterhout and F. Douglis, "Beating the i/o bottleneck: A case for log-structured file systems," *SIGOPS Oper. Syst. Rev.*, vol. 23, no. 1, p. 11–28, Jan. 1989. [Online]. Available: https://doi.org/10.1145/65762.65765

[73] R. K. Jain, "Scheduling data transfers in parallel computers and communications systems," Ph.D. dissertation, University of Texas at Austin, 1992.

[74] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart, *LINPACK users' guide*. SIAM, 1979.

[75] H. Benchmark, "Hpcg benchmark," 2016, http://www.hpcg-benchmark.org/. Accessed: 4-5-2020. [Online]. Available: http://www.hpcg-benchmark.org/

[76] Graph500: http://www.graph500.org/.

[77] IO500: https://www.vi4io.org/io500/start.

[78] "Ecp proxy apps suite," https://proxyapps.exascaleproject.org/ecp-proxy-apps-suite/, accessed: 4-15-2019.

[79] IOR - Parallel filesystem I/O benchmark: https://github.com/hpc/ior.

[80] B. T. Shealy, J. J. R. Burns, M. C. Smith, F. Alex Feltus, and S. P. Ficklin, "Gpu implementation of pairwise gaussian mixture models for multi-modal gene co-expression networks," *IEEE Access*, vol. 7, pp. 160 845–160 857, 2019.

[81] C. A. Targonski, C. A. Shearer, B. T. Shealy, M. C. Smith, and F. A. Feltus, "Uncovering biomarker genes with enriched classification potential from Hallmark gene sets," *Scientific Reports*, vol. 9, no. 1, p. 9747, 2019. [Online]. Available: https://doi.org/10.1038/s41598-019-46059-1

[82] C. Targonski, "X-MAP: Deep learning applications for the natural sciences," M.S. thesis, Clemson University, May 2019.

[83] R. S. DeFever, C. Targonski, S. W. Hall, M. C. Smith, and S. Sarupria, "A generalized deep learning approach for local structure identification in molecular simulations," *Chem. Sci.*, vol. 10, pp. 7503–7515, 2019. [Online]. Available: http://dx.doi.org/10.1039/C9SC02097G

[84] M. Griffin, M. Malsick, H. Mizzell, and L. Moore, "Historic rainfall and record-breaking flooding from hurricane florence in the pee dee watershed," *The Journal of South Carolina Water Resources*, pp. 28–35, 01 2020.

[85] "Pedestrians: 2016 data," National Center for Statistics and Analysis, National Highway Traffic Safety Administration, Tech. Rep. DOT HS 812 493, 03 2018.

[86] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, pp. 637–646, 2016.

[87] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten, "Scalable molecular dynamics with namd," *Journal of computational chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005.

[88] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. Quinn, "Massively parallel cosmological simulations with changa," in *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 2008, pp. 1–12.

[89] N. Jain, E. Bohm, E. Mikida, S. Mandal, M. Kim, P. Jindal, Q. Li, S. Ismail-Beigi, G. J. Martyna, and L. V. Kale, "Openatom: Scalable ab-initio molecular dynamics with diverse capabilities," in *International Conference on High Performance Computing*. Springer, 2016, pp. 139–158.

[90] 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.

[91] 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.

[92] J. J. Galvez, K. Senthil, and L. Kale, "Charmpy: A python parallel programming model," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 423–433.

[93] 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.

[94] R. D. Hornung and J. A. Keasler, "The raja portability layer: overview and status," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2014.

[95] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[96] J. Bent, "Two possible paths to exascale." [Online]. Available: http://institute.lanl.gov/hec-fsio/conferences/2010/presentations/day1/Bent-HECFSIO-2010-SwimLanes.pdf

[97] M. Bancroft, J. Bent, E. Felix, G. Grider, J. Nunez, S. Poole, R. Ross, E. Salmon, and L. Ward, "High end computing interagency working group (heciwg) sponsored file systems and i/o workshop hec fsio 2009." [Online]. Available: http://institute.lanl.gov/hec-fsio/docs/HEC-FSIO-FY09-Workshop-Document.pdf

[98] M. Bancroft, J. Bent, E. Felix, G. Grider, J. Nunez, S. Poole, R. Ross, E. Salmon, and L. Ward, "High end computing interagency working group (heciwg) sponsored file systems and i/o 2009 roadmaps." [Online]. Available: http://institute.lanl.gov/hec-fsio/docs/HEC-FSIO-FY09-Gaps_RoadMap.pdf

[99] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for Linux clusters," in *Proceedings of the 4th annual Linux showcase and conference*. MIT Press, 2000, pp. 391–430.

[100] W. B. L. Ligon and R. B. Ross, "Server-side scheduling in cluster parallel i/o systems," in *Parallel I/O for Cluster Computing*, C. Cèrin and H. J. editors, Eds. Kogan Page Science, September 2003, pp. 157–177.

[101] S. Ghemawat, H. Gobioff, and S. Leung, "The Google file system," in *19th ACM Symposium on Operating Systems Principles*, 2003.

[102] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 307–320.

[103] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies*, 2010.

[104] P. J. Braam, "The Lustre storage architecture," Cluster File Systems, Inc., 2003.

[105] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI'04)*. USENIX Association, 2004.

[106] G. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu, "ParalleX: A study of a new parallel computation model," in *IEEE International Parallel and Distributed Processing Symposium, 2007*, march 2007, pp. 1 –6.

[107] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active messages: A mechanism for integrated communication and computation," in *19th International Symposium on Computer Architecture*, Gold Coast, Australia, 1992, p. 256–266.

[108] H. H. Kaiser, M. Brodowicz, and T. Sterling, "Parallex: An advanced parallel execution model for scaling-impaired applications," in *Parallel Processing Workshops, International Conference on Parallel Processing*, 2009.

[109] S. Yang, M. Brodowicz, H. Kaiser, and W. Ligon, "PXFS: A persistent storage model for extreme scale," in *Proceedings of the Workshop on Computing, Networking, and Communications, ICNC'14*, 2014.

[110] N. Mills, A. Feltus, and W. B. Ligon, "Maximizing the performance of scientific data transfer by optimizing the interface between parallel file systems and advanced research networks," *Future Generation Computer Systems*, vol. 79, pp. 190–198, 2018. [Online]. Available: https://doi.org/10.1016/j.future.2017.04.030

[111] N. Mills, E. M. Bensman, W. L. Poehlman, L. W. B., and F. A. Feltus, "Moving just enough deep sequencing data to get the job done," *Bioinformatics and Biology Insights*, 2019. [Online]. Available: https://doi.org/10.1177/1177932219856359

[112] B. D. N. M. William Poehlman, Mats Rynge and F. Feltus, "Osg-kinc: High-throughput gene co-expression network construction using the open science grid," in *IEEE BIBM 2017 Proceedings*, 2017, pp. 1827–1831.

[113] K. Sapra, "Framework for lifecycle enrichment of HPC applications on exascale heterogeneous architecture," in *SC Doctoral Showcase*, Nov. 2015.

[114] K. Sapra, "Framework for lifecycle enrichment of hpc applications towards exascale heterogeneous architectures," Ph.D. dissertation, Clemson University, Dec. 2018.

[115] A. Joshi, "A performance focused, development friendly and model aided parallelization strategy for scientific applications," M.S. thesis, Clemson University, Dec. 2016.

[116] P. Malakar, P. Balaprakash, V. Vishwanath, V. Morozov, and K. Kumaran, "Benchmarking machine learning methods for performance modeling of scientific applications," in *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2018, pp. 33–44.

[117] J. D. Stevens and A. Klöckner, "A mechanism for balancing accuracy and scope in cross-machine black-box gpu performance modeling," 2019.

[118] V. K. Pallipuram, M. C. Smith, N. Raut, and X. Ren, "A regression-based performance prediction framework for synchronous iterative algorithms on general purpose graphical processing unit clusters," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 2, pp. 532–560, 2014. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3017

[119] E. Nwafor, A. Campbell, D. Hill, and G. Bloom, "Towards a provenance collection framework for internet of things devices," in *2017 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computed, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, 2017, pp. 1–6.

[120] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-fi: Collecting high-fidelity whole-system provenance," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2420950.2420989 p. 259–268.

[121] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/bates pp. 319–334.

[122] D. Dai, Y. Chen, P. Carns, J. Jenkins, and R. Ross, "Lightweight Provenance Service for High-Performance Computing," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Portland, OR: IEEE, Sep. 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8091224/ pp. 117–129.

[123] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Fear and Logging in the Internet of Things," in *Proceedings 2018 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2018. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_01A-2_Wang_paper.pdf

[124] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, "Provenance-aware storage systems," in *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ser. ATEC '06. USA: USENIX Association, 2006, p. 4.

[125] A. Gehani and D. Tariq, "Spade: Support for provenance auditing in distributed environments," in *Proceedings of the 13th International Middleware Conference*, ser. Middleware '12. Berlin, Heidelberg: Springer-Verlag, 2012, p. 101–120.

[126] F. Cappello, S. Di, S. Li, X. Liang, A. M. Gok, D. Tao, C. H. Yoon, X.-C. Wu, Y. Alexeev, and F. T. Chong, "Use cases of lossy compression for floating-point data in scientific data sets," *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1201–1220, 2019.

[127] S. Di and F. Cappello, "Fast error-bounded lossy hpc data compression with sz," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 730–739.

[128] J. Calhoun, F. Cappello, L. N. Olson, M. Snir, and W. D. Gropp, "Exploring the feasibility of lossy compression for pde simulations," *The International Journal of High Performance Computing Applications*, vol. 33, no. 2, pp. 397–410, 2019. [Online]. Available: https://doi.org/10.1177/1094342018762036

[129] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis, "The case for compressed caching in virtual memory systems," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '99. Berkeley, CA, USA: USENIX Association, 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268708.1268716 pp. 8–8.

[130] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. R. de Supinski, and R. Eigenmann, "McrEngine: a scalable checkpointing system using data-aware aggregation and compression," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389020 pp. 17:1–17:11.

[131] L. Fischer, S. Götschel, and M. Weiser, "Lossy data compression reduces communication time in hybrid time-parallel integrators," *Computing and Visualization in Science*, vol. 19, no. 1, pp. 19–30, Jun 2018. [Online]. Available: https://doi.org/10.1007/s00791-018-0293-2

[132] R. Filgueira, D. E. Singh, J. Carretero, A. Calderón, and F. García, "Adaptive-compi: Enhancing mpi-based applications' performance and scalability by using adaptive compression," *The International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 93–114, 2011. [Online]. Available: https://doi.org/10.1177/1094342010373486

[133] P. Deutsch, "Gzip file format specification version 4.3," United States, Tech. Rep., 1996.

[134] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 12, no. 5, pp. 1245–1250, 2006. [Online]. Available: http://dx.doi.org/10.1109/tvcg.2006.143

[135] Y. Collet and M. Kucherawy, "Zstandard Compression and the application/zstd Media Type," RFC 8478, Oct. 2018. [Online]. Available: https://rfc-editor.org/rfc/rfc8478.txt

[136] S. W. Son, Z. Chen, W. Hendrix, A. Agrawal, W. keng Liao, and A. Choudhary, "Data compression for the exascale computing era - survey," *Supercomputing frontiers and innovations*, vol. 1, no. 2, 2014. [Online]. Available: http://superfri.org/superfri/article/view/13

[137] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, Dec 2014.

[138] X. Ni, T. Islam, K. Mohror, A. Moody, and L. V. Kale, "Lossy compression for checkpointing: Fallible or feasible?" in *Poster Session of the 2014 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Washington, DC, USA: IEEE Computer Society, 2014.

[139] D. Laney, S. Langer, C. Weber, P. Lindstrom, and A. Wegener, "Assessing the effects of data compression in simulations using physically motivated metrics," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503283 pp. 76:1–76:12.

[140] N. Sasaki, K. Sato, T. Endo, and S. Matsuoka, "Exploration of lossy compression for application-level checkpoint/restart," in *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '15. Washington, DC, USA: IEEE Computer Society, 2015. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2015.67 pp. 914–922.

[141] J. Nardi, N. Feldman, A. Poppick, A. Baker, and D. Hammerling, "Statistical analysis of compressed climate data," NCAR, Tech. Rep., 2018.

[142] M. Rahman, M. Islam, J. Calhoun, and M. Chowdhury, "Real-time pedestrian detection approach with an efficient data communication bandwidth strategy," *Transportation Research Record*, vol. 0, no. 0, p. 0361198119843255, 0. [Online]. Available: https://doi.org/10.1177/0361198119843255

[143] D. Ellsworth, T. Patki, S. Perarnau, S. Seo, A. Amer, J. Zounmevo, R. Gupta, K. Yoshii, H. Hoffman, A. Malony et al., "Systemwide power management with argo," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2016, pp. 1118–1121.

[144] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A Portable Programming Interface for Performance Evaluation on Modern Processors," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 3, pp. 189–204, 2000.

[145] H. Menon, B. Acun, S. G. De Gonzalo, O. Sarood, and L. Kalé, "Thermal aware automated load balancing for hpc applications," in *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*. IEEE, 2013, pp. 1–8.

[146] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *ACM Transactions on Storage*, vol. 7, no. 8, pp. 1–26.

[147] P. H. Carns, R. Latham, R. B. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale I/O workloads." in *In Proceedings of the Workshop on Interfaces and Architectures for Scientific Data Storage*. IEEE Computer Society, 2009, pp. 1–10.

[148] "The orangefs project." [Online]. Available: http://www.orangefs.org

[149] L. Kalé and S. Krishnan, "Charm++ : A portable concurrent object oriented system based on C++," in *Proceedings of the Conference on Object Oriented Programmi ng Systems, Languages and Applications*, September 1993.

[150] H. Kaiser, M. Brodowicz, and T. Sterling, "Parallex an advanced parallel execution model for scaling-impaired applications," in *2009 International Conference on Parallel Processing Workshops*, 2009, pp. 394–401.

[151] F. Cappello, E. Constantinescu, P. Hovland, T. Peterka, C. Phillips, M. Snir, and S. Wild, "Improving the trust in results of numerical simulations and scientific data analytics," Argonne National Lab.(ANL), Argonne, IL (United States), Tech. Rep., 2015.

[152] "Opendwarfs benchamrking suite," https://github.com/vtsynergy/OpenDwarfs, accessed: April 1, 2020.

[153] J. Tian, S. Di, C. Zhang, X. Liang, S. Jin, D. Cheng, D. Tao, and F. Cappello, "Wavesz: A hardware-algorithm co-design of efficient lossy compression for scientific data," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi-org.libproxy.clemson.edu/10.1145/3332466.3374525 p. 74–88.

[154] Q. Xiong, R. Patel, C. Yang, T. Geng, A. Skjellum, and M. C. Herbordt, "Ghostsz: A transparent fpga-accelerated lossy compression framework," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2019, pp. 258–266.

[155] "Xilinx gzip library," https://github.com/Xilinx/Applications/tree/master/GZip, accessed: April 1, 2020.

[156] "Gpu-accelerated lossless compression survey," https://github.com/dingwentao/GPU-lossless-compression, accessed: April 1, 2020.

[157] R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens, "Parallel lossless data compression on the gpu," in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–9.

[158] "Parallel implementation of bzip2 using cuda," https://github.com/bzip2-cuda/bzip2-cuda, accessed: April 1, 2020.

[159] A. Deshpande and P. J. Narayanan, "Fast burrows wheeler compression using all-cores," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015, pp. 628–636.

[160] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman, and K. A. Ross, "Massively-parallel lossless data decompression," in *2016 45th International Conference on Parallel Processing (ICPP)*, 2016, pp. 242–247.

[161] S. Ashby, P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, and P. Messina, "The opportunities and challenges of exascale computing," *Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee*, pp. 1–77, 2010.

[162] Paul Messina, "The u.s. d.o.e. exascale computing project - goals and challenges," NIST, Tech. Rep., 2017.

[163] Libpressio: https://github.com/robertu94/libpressio.

[164] SZ: https://github.com/disheng222/SZ.

[165] CuZFP. 2019. https://github.com/LLNL/zfp/tree/develop/src/cuda_zfp. Online.

[166] G. Sun and S. Jun, "Zfp-v: Hardware-optimized lossy floating point compression," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 117–125.

[167] T. Reza, J. Calhoun, K. Keipert, S. Di, and F. Cappello, "Analyzing the performance and accuracy of lossy checkpointing on sub-iteration of nwchem," in *2019 IEEE/ACM 5th International Workshop on Data Analysis and Reduction for Big Scientific Data (DRBSD-5)*, 2019, pp. 23–27.

[168] C. B. McKnight, A. L. Poulos, M. R. Bender, J. C. Calhoun, and F. A. Feltus, "Exploring lossy compression of gene expression matrices," in *2019 IEEE/ACM 5th International Workshop on Data Analysis and Reduction for Big Scientific Data (DRBSD-5)*, 2019, pp. 28–34.

[169] J. Diffenderfer, A. Fox, J. Hittinger, G. Sanders, and P. Lindstrom, "Error analysis of zfp compression for floating-point data," *SIAM Journal on Scientific Computing*, 02 2019.

[170] P. Lindstrom, "Error distributions of lossy floating-point compressors," *Joint Statistical Meetings 2017*, pp. 2574–2589, October 2017.

[171] S. Han and T. Fingscheidt, "Variable-length versus fixed-length coding: On trade-offs for soft-decision decoding," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2014, pp. 4269–4273.

[172] R. Underwood, J. Calhoun, S. Di, and F. Cappello, "Fraz: A generic high-fidelity fixed-ratio lossy compression framework for scientific data," in *2020 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2020, New Orleans, USA, May 18-22, 2020*. IEEE, 2020.