Technical Report
Number: NU-CS-2024-01

March, 2024

On Transparent Optimizations for Communication in Highly Parallel Systems

Michael Wilkins

Abstract

To leverage the omnipresent hardware parallelism in modern systems, applications must efficiently communicate across parallel tasks, e.g., to share data or control execution flow. The longstanding mechanisms for shared memory and distributed memory, i.e., coherence and message passing, remain the dominant choices to implement communication. I argue that these stalwart constructs can be *transparently* optimized, improving performance without exposing developers to the growing complexity of modern hardware that employ both shared and distributed memory. Then, I explore the ultimate ambition: a unified transparent communication abstraction across all memory types.

In shared memory multiprocessors, communication is performed implicitly. Cache coherence maintains the abstraction of a single shared memory among hardware threads, so the application does not have to explicitly move data between them. However, coherence protocols incur an increasing overhead in modern hardware due to their conservative, reactive policies. I designed a new coherence protocol called WARDen to exploit the novel WARD property, which indicates large regions of memory that do not require fine-grained coherence. By transparently disabling the coherence protocol when it is unneeded, WARDen maintains the abstraction of shared memory and improves application performance by an average of 1.46x.

In distributed memory machines, communication between memory domains is performed explicitly by the application. To specify the necessary communication, collective operations are the predominant primitive because they allow programmers to elegantly specify large-scale communication patterns in a single function call. The Message Passing Interface (MPI) is the de facto standard for collectives in high-performance distributed memory systems like supercomputers. MPI libraries typically contain 3-4 implementations (i.e., algorithms) for each collective pattern.

Despite their utility, collectives suffer performance degradation due to poor algorithm selection in the underlying MPI library. I created a series of autotuners named FACT and ACCLAiM that use machine learning (ML) to tractably find the optimal collective algorithms for large-scale applications. The autotuners are sometimes limited when all the available algorithms fail to properly leverage the underlying hardware. To address this issue, I developed a set of more flexible algorithms that can better map to complex, modern networks and increase the potency of autotuning. Combining these efforts on Frontier (the world's fastest supercomputer at time of writing), I achieve speedups of over 4x compared to the proprietary vendor MPI library.

Lastly, I explored my vision for a higher-level programming model that abstracts away communication altogether. I ported the popular NAS Parallel Benchmark Suite to an FMPL (Functional, Memory-managed, Parallel Language). I found that FMPLs have the potential to drastically improve transparency because the program does not need to be aware of communication at all. However, FMPLs are currently limited to shared memory machines. I built a prototype that extends an FMPL to distributed memory, charting the course to FMPLs in high-performance computing.

Across these research thrusts, I developed novel optimizations for communication in high performance applications. Together, they show how existing communication abstractions, i.e., shared memory and message passing, can be transparently optimized, maintaining or even improving the level of abstraction exposed to the developer.

Keywords

Programming Models, Communication, High-Performance Computing, Cache Coherence, Message Passing Interface, Autotuning, Collective Operations

NORTHWESTERN UNIVERSITY

On Transparent Optimizations for Communication in Highly Parallel Systems

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Engineering

By

Michael Wilkins

EVANSTON, ILLINOIS

March 2024

ABSTRACT

On Transparent Optimizations for Communication in Highly Parallel Systems

Michael Wilkins

To leverage the omnipresent hardware parallelism in modern systems, applications must efficiently communicate across parallel tasks, e.g., to share data or control execution flow. The long-standing mechanisms for shared memory and distributed memory, i.e., coherence and message passing, remain the dominant choices to implement communication. I argue that these stalwart constructs can be *transparently* optimized, improving performance without exposing developers to the growing complexity of modern hardware that employ both shared and distributed memory. Then, I explore the ultimate ambition: a unified transparent communication abstraction across all memory types.

In shared memory multiprocessors, communication is performed implicitly. Cache coherence maintains the abstraction of a single shared memory among hardware threads, so the application does not have to explicitly move data between them. However, coherence protocols incur an increasing overhead in modern hardware due to their conservative, reactive policies. I designed a new coherence protocol called WARDen to exploit the novel WARD property, which indicates large regions of memory that do not require fine-grained coherence. By transparently disabling the coherence protocol when it is unneeded, WARDen maintains the abstraction of shared memory and improves application performance by an average of 1.46x.

In distributed memory machines, communication between memory domains is performed explicitly by the application. To specify the necessary communication, collective operations are the predominant primitive because they allow programmers to elegantly specify large-scale communication patterns in a single function call. The Message Passing Interface (MPI) is the de-facto standard for collectives in high-performance distributed memory systems like supercomputers. MPI libraries typically contain 3-4 implementations (i.e., algorithms) for each collective pattern.

Despite their utility, collectives suffer performance degradation due to poor algorithm selection in the underlying MPI library. I created a series of autotuners named FACT and ACCLAiM that use machine learning (ML) to tractably find the optimal collective algorithms for large-scale applications. The autotuners are sometimes limited when all the available algorithms fail to properly leverage the underlying hardware. To address this issue, I developed a set of more flexible algorithms that can better map to complex, modern networks and increase the potency of autotuning. Combining these efforts on Frontier (the world's fastest supercomputer at time of writing), I achieve speedups of over 4x compared to the proprietary vendor MPI library.

Lastly, I explored my vision for a higher-level programming model that abstracts away communication altogether. I ported the popular NAS Parallel Benchmark Suite to an FMPL (Functional, Memory-managed, Parallel Language). I found that FMPLs have the potential to drastically improve transparency because the program does not need to be aware of communication at all. However, FMPLs are currently limited to shared memory machines. I built a prototype that extends an FMPL to distributed memory, charting the course to FMPLs in high-performance computing.

Across these research thrusts, I developed novel optimizations for communication in high performance applications. Together, they show how existing communication abstractions, i.e., shared memory and message passing, can be transparently optimized, maintaining or even improving the level of abstraction exposed to the developer.

ACKNOWLEDGEMENTS

As I near the end of my Ph.D. journey, I am filled with gratitude for those who guided, assisted, and supported me throughout my studies.

I would first like to thank my advisors, Peter Dinda and Nikos Hardavellas. From our initial conversations pitching their labs, I knew I had found the right home to complete my studies. Peter and Nikos were an incredible tandem because they mentored me in different ways, yet always seemed to be on the same page. I will never forget how Peter empowered me to pursue research across a wide range of topics from architecture to HPC, somehow maintaining the expertise to assist me through all of it. Nikos' support was invaluable, particularly challenging me to find the novelty in my work and communicate it effectively, with the proper use of hyphens of course. As a team, Peter and Nikos struck the perfect balance to keep me on track while allowing me to shape my own experience. I am incredibly grateful for the opportunity to work with them.

I would also like to thank my advisors and committee members from Argonne National Laboratory, my second (virtual) home over the last 3 years. Min Si and Pavan Balaji gave me the opportunity to join the laboratory, offering an internship and then a long-term position while ensuring my success despite the start of the pandemic. Additionally, I thank them for the opportunity to join Meta in the summer of 2023 for a singular internship experience. Yanfei Guo and Rajeev Thakur were incredibly gracious to guide me after my initial advisors left for Meta. They expertly helped me complete a fruitful research path and have been critical in my search for my next step.

I also thank my final committee member, Alok Choudhary, for supporting my thesis and providing invaluable feedback.

There are numerous colleagues that deserve praise here, including Sam Westrick, Vijay Kandiah, Alex Bernat, Brian Suchy, Enrico Deiana, Simone Campanoni, Umut Acar, Brian Homering, and Tommy McMichen. I am proud to share my contributions with such an amazing list of kind collaborators. Additionally, I am honored to have supported the journeys of other amazing researchers, including Hanming Wang, Peizhi Liu, Garrett Weil, Liam Patterson, Sean Rhee, Colin Schilf, Yankai Jiang, Yanzhou Wang, Bangyen Pham, and Luke Arnold.

My family's support was critical to my success throughout my education. I thank my mother, Kimberley Wilkins, for always being there for me. She is the sounding board I trust to guide me in the right direction and encourage me at every bump in the road. To my father, Donald Wilkins, who sparked my passion for technology and showed me how to chase my passions. Of course, I would be remiss without mentioning my favorite sibling, Marissa Wilkins. Our friendly competition growing up inspired me to maximize my potential, and her support actually made it possible.

While she cannot read this (she is a dog), my little buddy Hazel deserves a mention nonetheless. She has been perhaps the most constant presence by my side throughout the many hours preparing this work, and I was kept sane via her affection.

Lastly, I would like to thank Abby Carrell for her unconditional love and support. My journey has become our journey, and I am eternally grateful to have her by my side.

THESIS STATEMENT

Communication is a foundational aspect of high-performance parallel programming for applications big and small, but it is a growing bottleneck on modern hardware. I claim that the existing abstractions of communication (e.g., coherence, message passing) can be transparently optimized to address the performance challenge. Furthermore, I hypothesize that a new, higher-level programming model make communication transparent altogether while maintaining high performance.

Thesis Committee

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Application-level parallelism is a critical component of high-performance computer programs. Since the end of Dennard Scaling [68] nearly 20 years ago, the sequential processing capability of computer hardware has plateaued [194]. Applications must now directly leverage hardware parallelism to achieve high performance. Examples are wide-sweeping, from general workloads running on multiprocessors to high-performance computing (HPC) codes using exascale supercomputers.

For all of these use cases, communication between concurrent tasks is a foundational aspect of parallel programming. Application tasks regularly communicate to share data, synchronize, load balance, resolve dependencies, etc. To perform this communication, applications leverage well-understood constructs like shared memory and/or message passing (e.g., MPI [92]). In recent years, new programming models (see Section 6.3 for examples) have arisen to help users better utilize modern hardware, but their adoption remains limited.

I argue that the existing constructs can be *transparently* optimized, maintaining their familiar programming model while improving performance on modern computer systems. Also, I present my own vision for the future of high-performance parallel programming, where a higher-level programming model shields the programmer from communication altogether while maintaining high performance.

This work is organized into three thrusts: 1.) shared memory communication 2.) distributed memory communication, and 3.) new programming model. Below, I include primers for each.

1.1 Communication in Shared Memory

With the slowdown of single core performance improvements, major hardware manufacturers now sell processors with multiple physical cores. This trend continues to grow; for example, the latest AMD EPYC processors include up to 128 cores [3]. Over the last 6 years, this chip line-up has increased its Instructions Per Cycle (IPC) by only 68%, while the maximum number of cores has increased by 4x. Furthermore, many computer systems support more than one processor chip, further increasing the core count. When executing across a single multiprocessor, applications utilize the abstraction of "shared memory" to communicate efficiently.

Shared memory is the illusion that all concurrent tasks are accessing a single memory. This abstraction is incredibly useful because threads can freely access the program's entire working set without an explicit mechanism or programmer effort. The illusion of shared memory is primarily maintained in hardware by cache coherence. In reality, separate cores in multiprocessors have their own memory hierarchies (e.g., caches) and may maintain copies of the same memory location. Cache coherence is the implementation of a policy called a "cache coherence protocol" that ensures all copies of the same memory location are kept consistent.

While it provides a powerful abstraction to applications, cache coherence is a major bottleneck in multiprocessors. Coherence can amount to 50% or more of on-chip interconnect traffic [74, 39, 65, 198]. Future systems, which are projected to expand through higher core counts, multi-socket systems [103, 38, 210, 156, 108, 125, 9, 118, 67] or disaggregation [41, 140, 153, 113, 76, 174, 134, 131] will further increase the cost of coherence.

There have been numerous variations and improvements proposed [203], but cache coherence protocols remain a bottleneck in modern systems because they treat each access equally and reactively. Intuitively, however, not all memory accesses made by an application must be treated the same. A simple example is a read-only data structure. The cache coherence protocol does not have to worry about the data being modified (because it is read-only), and could therefore turn off the protocol to improve performance. However, memory usage restrictions do not propagate to the hardware, so application-driven coherence deactivation optimizations are currently impossible for conventional architectures.

To better bridge the gap between applications and cache coherence, many designs have sought to remove/deactivate coherence, such as hardware-supported, compiler-directed (HSCD) cache coherence [59], OS-driven coherence deactivation [62], and software cache coherence [129, 214, 11, 167, 213, 55, 58, 64, 164]. However, these prior works require the programmer to mark their application's memory behavior using pragmas or recover this information through run-time inspector-executor methods and compiler analyses. Pragmas place additional burden on the programmer, which violates the primary tenet of my thesis. Others recoup the usage information using compiler analysis, but these methods are woefully limited. For example, some analyses treat entire arrays as single variables and fail to detect false sharing [55, 58], or limit array subscripts to loop iterators [64]. Again more approaches rely on software for triggering coherence actions and hardware for selective self-invalidations, but they incur high overhead in lock-intensive programs [11] or are restricted to unity loop iterators in affine loops without conditionals [164]. Lastly, hardware-directed approaches [65, 198] avoid some coherence costs by piggy-backing the virtual-to-physical address translation, but they cannot work in the presence of common complexities like false sharing. In summary, there remains a need for a performant and transparent co-designed cache coherence solution.

My cache coherence protocol, WARDen, seeks to address this outstanding challenge. The basis for WARDen is a new memory property called WARD, which stands for **W**rite-After-Write **A**pathy and **R**ead-After-Write **D**ependence freedom. WARD refers to Write-After-Write (WAW)

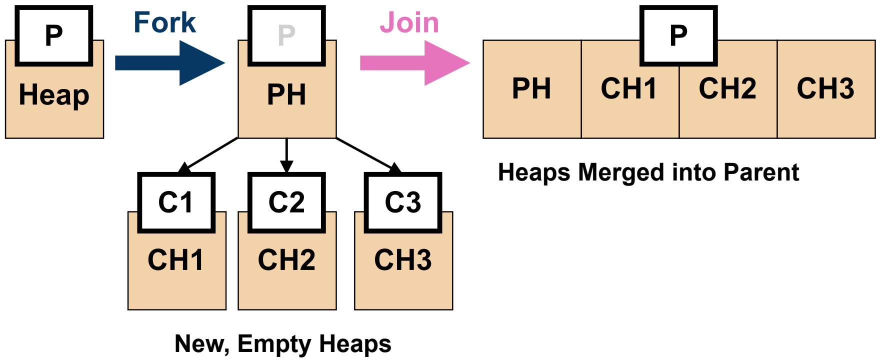


Figure 1.1: **Memory Heap Hierarchy of a Fork-Join Parallel Program.** Memory heaps are organized identically to the fork-join pattern of the parallel tasks. "Leaf Heaps" are the memory heaps at the lowest level of the hierarchy (e.g., PH is initially a leaf heap, then CH1-3 are leaf heaps). Tasks may only access their own heap and those of their ancestors in the hierarchy.

and Read-After-Write (RAW) memory hazards, which along with Write-After-Read (WAR) form the three categories of memory (or data) hazards that can be encountered in a parallel programs. These hazards occur when two parallel tasks access the same memory location. For example, one task may attempt to read a value written by another task, which constitutes a RAW hazard. It is the cache coherence protocol's primary responsibility to correctly service memory hazards. Intuitively, the WARD property restricts the allowable memory hazards to make the cache coherence protocol superfluous, meaning WARDen can safely disable it. A formal definition of the WARD property is included in Chapter 2.

If WARDen were to mimic the previous work, it would either a.) require to programmer to manually signal where the WARD property occurs (called "WARD memory regions" or "WARD regions" for short) or b.) attempt to derive it using some limited analysis. Instead, I co-designed WARDen directly with the *programming language* to overcome these challenges.

WARDen is co-designed with the High-Level Parallel Language (HLPL) family of programming languages. HLPLs are memory-managed languages that make parallel programming simpler and safer. For my thesis research, I focused on HLPLs that implement the fork-join programming idiom popularized in lower-level contexts such as Cilk [29, 87] and OpenMP [165]. Example HLPLs include Multicore Ocaml [206] and various dialects of Parallel ML [85, 231, 7, 205].

HLPLs are growing in popularity across academia, industry, and education. Academia and industry are actively developing and improving these languages [7, 231, 206, 207]. Also, HLPLs are part of the curricula of top undergraduate programs. For example, Carnegie Mellon University uses HLPLs to teach the fundamentals of data structures and parallel algorithms. Due to their growing popularity and attractive properties, I selected HLPLs as the target for my new protocol.

HLPLs use memory in a disciplined manner to ensure some degree of determinism, efficiency, and scalability [132, 133, 27, 97, 10, 28, 26, 175, 138, 98, 208, 85, 182, 95, 231, 7]. Through WARDen, I show that this disciplined use of memory can be exploited to improve performance of the underlying communication. The HLPL I specifically target (a derivative of Parallel ML defined by the MPL compiler [231]) organizes memory into a hierarchy that matches the logical hierarchy of the parallel tasks. An example hierarchy and how it changes throughout the fork-join life cycle is shown in Figure 1.1.

I observed that the leaf nodes in the memory hierarchy, which represent the local working sets of the active tasks, all have the WARD property *by construction*, meaning WARDen can leverage it without programmer effort or any analysis. WARDen receives the memory locations of leaf heaps from the language runtime, and uses a novel hardware protocol called MESI-W to dynamically disable coherence for these regions. On a set of benchmarks and applications written in Parallel ML, WARDen achieves a 1.46x average speedup that increases with scale and memory disaggregation. Overall, the WARDen protocol showcases the possibility to transparently upgrade existing communication mechanisms.

1.2 Communication in Distributed Memory

Many parallel applications must span multiple distributed memory domains to fully utilize the hardware parallelism of a system. Supercomputers are a prominent example. In 2022, Oak Ridge National Laboratory's Frontier became the first exascale system at the top of the TOP500 list. Frontier's 9408 distributed nodes achieve 1.6 exaflops peak performance. To communicate across such a vast system, applications predominantly rely on the Message Passing Interface (MPI).

MPI is the de facto standard for communication in distributed high-performance computing [92]. The primary abstraction provided by MPI is the idea of "message passing". To communicate between two distributed processes, the program calls MPI_Send on one process and MPI_Recv on the other. The MPI library interfaces with the network to transfer the message from the send buffer to the receive buffer. Point-to-point messaging is an iconic programming model, but it quickly becomes unwieldy at large scale. For example, to communicate a single message from one node to all the other nodes on Frontier, an application would need 1000s of point-to-point calls.

To better express large-scale communication, MPI includes "collective" primitives. Collectives allow the programmer to express the communication pattern for all processes with a single call. The one-to-all example from the previous paragraph can be implemented using a single $MPI\_Broadcast$ collective. Collectives are the most popular primitive in MPI, and they are the focus of my optimization efforts.

While popular, MPI collectives are a significant bottleneck. A characterization of production HPC systems in 2018 found that MPI applications spend 50% or more of their execution time on MPI rather than actual computation [60]. 25–33% of the MPI overhead in this study is contributed by collectives. This percentage is expected to grow on new systems. A profile of the Exascale

Computing Project's (ECP) Proxy Application Suite 4.0 found the expected workload of exascale systems, such as Frontier, expend 40% or more of their overall runtime on collectives alone [211]. Clearly, collectives are an important target for optimization.

Collectives' higher level of abstraction allows for optimization of the communication pattern's implementation. In response, MPI libraries sport a set of algorithms for each collective. Selecting a sub-optimal algorithm can significantly reduce performance (35–40% [109]), but selecting a good algorithm is not easy.

Generally speaking, the performance of each algorithm is influenced by many factors, from software (e.g., message size) to hardware (e.g., network latency). Dynamic factors (e.g., network congestion, connectivity of the participating nodes) vary frequently, making it impossible to make accurate static selections. To address the challenge of collective algorithm selection, autotuning is a natural solution.

Multiple methods have been proposed to autotune collective algorithm selection. Examples include analytical models [225, 77, 177, 178, 142] and exhaustive benchmarking tools [47]. Analytical models have failed to gain adoption because they are difficult to implement, maintain, and expand for new algorithms. Production tools such as Intel's MPITune and OPTO [47] use exhaustive benchmarking. Benchmarks must be rerun frequently to account for dynamic factors, making this brute-force strategy impractical for large systems. In practice, it can be used only infrequently to tune individual scenarios.

I instead explore machine learning (ML)-based autotuning approaches. ML can improve upon exhaustive approaches by learning to predict scenarios that have not been benchmarked [111, 109], lessening the benchmarking overhead. ML also has an inherent advantage over analytical models because it can learn patterns in the data caused by factors that are difficult to model analytically, such as real-time and/or machine-specific influences.

While ML has the potential to improve collective algorithm selection, there remain many challenges. Most prominently, existing ML autotuners rely on random collection to build their training dataset. This approach was sufficient for the previous work [109] because it only scaled to a small number of nodes (up to 48). I estimate collecting this data for a larger system of 512 nodes (only a fraction of the size of Frontier) would take approximately 75,000 core hours, which is over 6 days of machine time. This data collection must be repeated frequently to capture dynamic factors like network congestion, resulting in *multiple* 6 day blocks of machine downtime.

My thesis research features a sequence of ML autotuner designs named FACT and ACCLAiM (both are acronyms that are expanded in Chapter 3). Among many advancements, FACT and ACCLAiM primarily seek to reduce the training data collection time. For the previous example, ACCLAiM reduces the estimated 6 days of data collection time to 4–6 minutes per collective or 4–10 minutes per application.

During my time spent on autotuning existing collective algorithms, I observed that the available algorithms are outdated due to advancements in networking hardware found on exascale supercomputers. As a result, ACCLAiM frequently encounters scenarios where there is not an algorithm available that improves performance, limiting its effectiveness. To overcome this issue, I created a set of “generalized” (i.e., variable-radix) algorithms targeted at exascale-class systems.

Previous works have developed generalized collective algorithms which expose the radix of the algorithm as an optimizable parameter. These efforts show significant performance benefits by making the algorithms more flexible, so they can better match new networks. However, the previous works are specialized for specific scenarios (e.g., a specific network topology [195], small message size allreduce [192], or intranode broadcast [193]), and the broader efficacy of the strategy is unknown. As a result, generalized (i.e., variable-radix) algorithms appear very sparsely in current implementations of MPI, sacrificing performance on modern HPC systems. In the context of my autotuning efforts, the lack of new algorithms limits the upside of my work.

I identified hardware features shared among the upcoming exascale supercomputers and algorithm generalizations that could better leverage them: binomial, recursive doubling, and ring [193, 192, 101]. Then, I designed 10 new system-agnostic generalized algorithms based on these generalizations. Combined with optimized algorithm selection, my new algorithms achieve up to a 4.6x speedup over the *proprietary, vendor implementation* of MPI on Frontier. This result closes the loop on my work to transparently optimize distributed memory communication.

1.3 A Higher-Level Parallel Programming Model

After investigating optimizations for shared and distributed memory, I sought to make communication transparent across both domains. Therefore, I explored higher-level parallel programming models for high-performance applications. There are many recent programming models (see Section 6.3)which propose new communication semantics to better match modern hardware, such as partitioned global data structures, asynchronous tasks, etc.

However, advancements in individual fields show another path is possible, where higher-level programming models can make communication completely transparent *and* achieve high performance. A prominent example is AI, where PyTorch [170] and TensorFlow [1] have enabled an explosion of new research in the field. Similarly, PETSc is the world's most widely used parallelism solution for scientific applications involving partial differential equations. These libraries are successful because they present a familiar, consistent programming model while transparently communicating across distributed hardware.

Inspired by this trend, I explored a more general, high-level programming model for high-performance applications. To do so, I expanded upon a Functional, Memory-Managed, Parallel Language (FMPL) to unify and transparentize communication across shared and distributed memory. FMPLs are, as the name implies, memory-managed, functional programming languages that support nested parallelism. Example FMPLs include NESL [25], parallel Haskell [99], and Parallel ML [94, 230]. FMPLs make parallel programming easier by transparently communicating between parallel threads to manage memory and execution flow. In addition, their functional semantics strictly control data mutation, helping users avoid the painful race conditions that are notorious in parallel programming [161, 147, 2, 35, 33, 34]. FMPLs support parallelism through high-level constructs (e.g., fork-join, parallel-for loops, parallel recursion, and nested data parallelism) that are popularized by tools like OpenMP [63]. All together, FMPLs enable transparent parallel programming with a familiar interface.

FMPLs are growing in popularity in many communities. In academia, FMPLs are actively being developed and improved by programming language researchers [230, 8]. One specific example, the ML language family, has been integral to programming language research for multiple decades [145]. Industry has also adopted these languages. The most prominent example is the trading company Jane Street, who contribute research and use FMPLs in production code [206, 207]. On the other hand, the FMPLs are unheard of in high-performance areas like HPC.

Performance-driven developers have shown a willingness to consider new parallel programming models and languages both in the past (e.g., High Performance Fortran [141], Coarray Fortran [162]) and present (e.g., SYCL [187], Chapel [51], modern Fortan [185], etc.). Meanwhile, there is a nascent trend among the high-performance community pushing towards modern languages such as Python [242] to avoid the growing complexity of modern systems, such as communication.

On paper, FMPLs have the potential to be the best of both worlds. Their functional semantics and strict data/memory control enable powerful optimizations that aim to match the performance of traditional, lower-level languages [230, 229, 238]. Additionally, FMPLs allow expression of the parallelism available in the algorithm without direct reference to the underlying communication, increasing productivity and portability. Therefore, FMPLs match my vision for a high-performance, transparent programming model.

I first set out to study the intersection between high-performance applications and FMPLs to better understand the benefits and drawbacks of FMPLs for my use case. I implemented the most well-known and widely studied benchmark suite for HPC, the NAS Parallel Benchmarks (NPB), in a state-of-the-art FMPL, Parallel ML. I documented my experience with FMPL development and compare the resulting benchmarks with the predominant existing C+OpenMP NPB implementation [160]. I found that FMPL implementations were 1.02–5.76× slower than the OpenMP versions, depending on the benchmark. Surprising, the FMPL implementations achieved competitive performance on a benchmark with regular parallelism (slowdown of 1.02× in the EP benchmark), indicating that for massively parallel HPC applications, FMPLs can be performance-competitive with lower-level implementations. However, they are currently limited to shared memory.

Armed with this promising result, I have begun to expand Parallel ML to support distributed memory parallelism, so it can truly support HPC applications. A distributed Parallel ML language could be massively beneficial to HPC developers, making it possible to write programs in an easy-to-use, high-level language and run them on distributed HPC systems *without modification*.

This project is highly ambitious and far exceeds the scope of my thesis, but I describe my current progress and the key challenges I have identified. The overarching goal of this work is to create a blueprint for future research on this exciting idea.

CHAPTER 2
WARDEN: OPTIMIZING SHARED MEMORY COMMUNICATION

I began my journey to transparently optimize communication by focusing on multiprocessors, where communication occurs through shared memory. I developed WARDen: a new cache coherence protocol co-designed with a high-level parallel language (HLPL). WARDen maintains the illusion of shared memory while actually disabling cache coherence for many regions of memory where it is unnecessary. An overview of the WARDen project is shown in Figure 2.1. In this section, I describe HLPLs and cache coherence in more detail. Then, I discuss the WARD property and my novel cache coherence protocol (WARDen) that leverages it. I conclude the section with the performance results from our evaluation of WARDen, which show how WARDen achieves an average speedup of 1.46x.

2.1 Disentanglement in High-Level Parallel Languages

WARDen is designed for HLPLs with nested fork-join parallelism, which relieve the programmer from manually managing parallelism. They instead use high-level constructs such as parallel-for loops. This approach relies on a thread scheduler (e.g., work stealing) to create and schedule parallel threads. It enables fine-grained parallelism by forking and joining many light-weight threads.

2.1.1 Spawn Trees

During execution, a fork-join program consists of a dynamic tree of light-weight threads called the *spawn tree*. Initially, there is a single root thread. At any moment, any leaf (i.e., a thread with

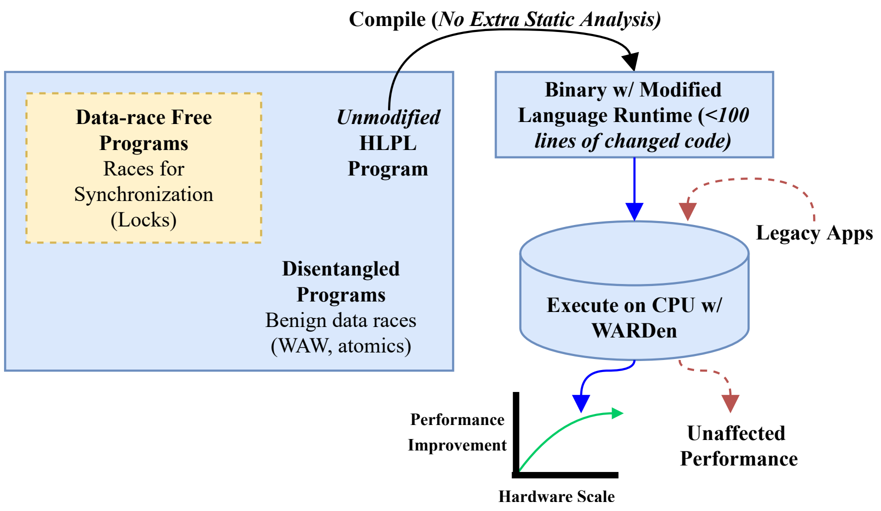

Compile (No Extra Static Analysis)
Data-race Free Programs
Races for Synchronization (Locks)
Unmodified HLPL Program
Disentangled Programs
Benign data races (WAW, atomics)
Binary w/ Modified Language Runtime (<100 lines of changed code)
Legacy Apps
Execute on CPU w/ WARDen
Performance Improvement
Hardware Scale
Unaffected Performance

Figure 2.1: **WARDen Overview.** WARDen transparently optimizes a broad set of shared-memory applications while maintaining performance for legacy applications. Acceleration increases with hardware scale.

no children) may *fork*, which suspends the thread and spawns two or more new child threads to run in parallel. When all children of a thread have completed execution, they may *join*, which removes the completed children from the tree and resumes the parent as a leaf. In this way, all internal (non-leaf) threads are suspended, and all computation is performed by leaf threads. Two threads are *concurrent* if neither is an ancestor of the other. That is, concurrent threads are siblings, cousins, etc.

2.1.2 Disentanglement

To avoid race conditions, fork-join parallel programs use memory in a disciplined manner. There are a number of approaches to this end, including race-detection [81, 86, 54, 148, 183, 184, 224, 23, 83], type and effect systems [32, 121], programming techniques for determinism [132, 133, 27], as well as determinism-by-default with purely functional programming [97, 10, 28, 26, 175, 138, 98, 208, 85, 182, 95, 231, 7].

Recent work identifies a property called *disentanglement*, which means that concurrent threads are oblivious to each other's allocations [182, 95, 231, 7, 228]. Disentanglement has broad applicability; it emerges naturally from fork-join parallelism [182], can be guaranteed by race-freedom [231], and is more general than pure functional programming. Disentanglement allows communication between concurrent threads only through memory allocated by common ancestors.

2.1.3 MPL and the Heap Hierarchy

Disentanglement can be exploited for improved efficiency and scalability, especially for automatic memory management and parallel garbage collection, as demonstrated by the MPL ("maple") compiler [155, 231, 7] for Parallel ML. To automate memory management, MPL organizes memory into a dynamic tree of heaps called the *heap hierarchy*, which mirrors the spawn tree. Maintenance of the heap hierarchy is illustrated in Figure 1.1. When a thread is spawned, it receives its own fresh heap in which it allocates all data. When a thread completes, its data is returned to the parent by merging its heap into the parent's.

2.1.4 Disentanglement Definition

Within the heap hierarchy, disentanglement is the property that threads only "use" data in their root-to-leaf path of heaps. A thread uses some data if the thread is holding a pointer to that data. By maintaining the heap hierarchy, the language runtime automatically ensures that the disentanglement property holds for the generated code [231, 228]. A more formal definition of disentanglement is as follows: A fork-join parallel program is **disentangled** if each thread only holds pointers (in its stack or registers) to data in either its own heap or the heap of an ancestor.

Disentanglement can also be defined in terms of the source language (Parallel ML), and the compiler (MPL) then ensures this property holds of the generated code [231].

2.1.5 Usefulness of Disentangled Programs

I now compare disentanglement with the classic property of data race freedom (DRF). Previous work leveraged DRF to improve cache coherence [32, 57, 191, 190]. However, disentanglement is a more general/useful property for developers.

Disentanglement is more general than data race-freedom because it allows “benign” data races. One example that leverages data races is a prime sieve, where multiple threads race to mark numbers as composite; this constitutes a write-write race, but this race is “benign” because all threads are writing the same value. Next, consider a calculation that does not decompose into equal-size parallel tasks on the target hardware (e.g., if an application seeks to calculate an odd number of values using vector units). In this case, the programmer may opt for a small amount of redundant computation, which will create a write-after-write race.

A final example is a parallel breadth-first search of a graph where the search uses inexact criteria (i.e., more than one vertex may meet the search criteria). The threads race to write an acceptable vertex to a shared memory location allocated by the ancestor who initiated the search. It does not matter which thread “wins” the race because they are all writing back values which meet the search criteria.This example highlights that “benign” data races may include write-after-write races with different values.

These examples contain data races, but are nevertheless disentangled. If desired, these algorithms *could* be made entirely data-race-free and deterministic by adding an explicit deduplication step. However, this would introduce additional memory pressure and decrease efficiency because the deduplication results would need to be written to memory. Instead, disentanglement trades a small amount of non-determinism for fewer allocations and improved performance.

2.2 Disabling Cache Coherence for Disentangled Programs

2.2.1 Data Hazards

Disabling coherence exploits the absence or irrelevance of data hazards between regions of concurrent computation. Data hazards exist when two hardware threads interact with a shared memory address and at least one thread writes to the address. When this occurs, the hardware must order the reads/writes to achieve correctness with respect to the consistency model.[1] Consequently, execution pauses, and overall progress is slowed. Of the three varieties of data hazards (Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW)), WARD reasons about RAW and WAW. True WAR hazards are prevented by WARDen's reconciliation process (§2.6.2).

2.2.1.1 Read-after-Write(RAW)

A read-after-write data hazard occurs when one hardware thread attempts to read an address written by another thread. If a RAW hazard is not executed in the correct order, the reading thread may receive stale data, resulting in incorrect program execution. To ensure correctness, the cache coherence protocol handles RAW hazards according to the memory consistency model by halting execution on the reading thread until the written data becomes available to it. Event 1 in Figure 2.2 shows an example, where thread j reads $value$, which is written by thread i. In the absence of the coherence protocol, thread j would read the value 0 instead of the value 1.

[1]I assume Total Store Order (TSO) [157].

2.2.1.2 Write-after-Write(WAW)

A Write-After-Write (WAW) data hazard occurs when two hardware threads write to the same memory address in a specific order. If the order between writes is not enforced, then the memory address may contain the wrong value, leading to incorrect program behavior. To ensure correctness, the coherence protocol orders the writes according to the memory consistency model, by stalling out-of-order writes as needed. Event 2 in Figure 2.2 shows an example WAW hazard, where thread j writes to $value$ after thread i. In the absence of the coherence protocol, $value$ would equal 1 at the end of execution when it should equal to 2.

2.2.2 Drawbacks of Cache Coherence

Cache coherence traditionally suffers from inefficiencies. One pitfall is false sharing, which persists as a challenge despite being studied for more than 25 years [223]. Chabbi et al. [48] developed a false sharing detection tool in 2018 that found false sharing even in benchmarks from the PARSEC suite. These findings show how difficult eliminating false sharing from programs can be.

Cache coherence protocols also suffer slowdowns due to *true* sharing. Current protocols address all true sharing events equally and reactively. However, some true sharing conflicts, such as benign WAW races found in disentangled programs, do not require fine-grained coherence. In addition, proactively flushing private caches can significantly improve performance (8–28%) [82]. WARDen improves both false sharing and true sharing scenarios to minimize the drawbacks of cache coherence for disentangled programs.

2.3 Opportunity for Optimization

HLPLs' disciplined memory management enables new advancements to overcome the bottlenecks of cache coherence. Previous works addressed some drawbacks of cache coherence for some programs using a property called data-race-freedom (DRF) [32, 57, 191, 190]. DRF prohibits concurrent accesses to *data* values (i.e., not synchronization primitives) if at least one access is a write. This property is more restrictive than disentanglement because it disallows benign WAW dependencies.

DRF must be enforced on an individual basis for each program. This burden typically falls on the programmer through manual annotation [32, 57, 191]. Requiring the programmer to carefully annotate the memory behavior of a high-level program defeats the purpose of a memory-managed language, and it clearly violates the "transparent optimization" focus of this thesis.

Disentanglement, on the other hand, is embedded in the language definition. The language runtime ensures that all programs exhibit the property without programmer involvement. By targeting a broader and language-enforced property, WARDen provides a completely transparent improvement for programmers using HLPLs.

2.4 WARD Property

I now formally define the WARD property and provide examples of how the definition applies to programs, both generally and in a high-level fork-join language.

2.4.1 WARD Definition

I define the WARD property for a memory location M. M displays the WARD property when two conditions hold for all hardware threads. For any two hardware threads i and j:

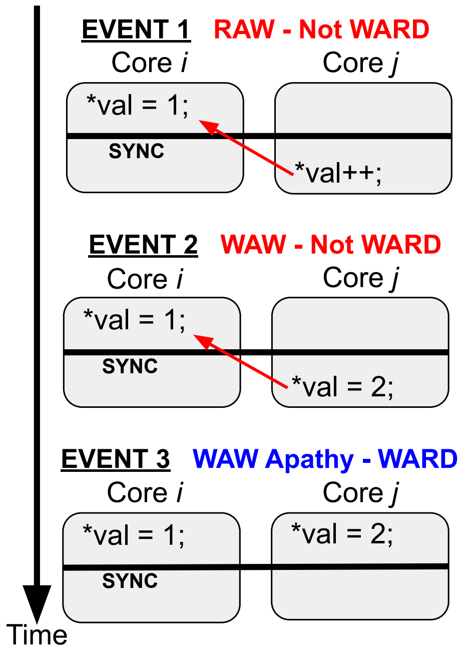


Figure 2.2: **Examples of Non-WARD (Events 1, 2) and WARD regions (Event 3).**

1. There exist no execution orders which include RAW dependencies between *i* and *j* at *M*.

2. Any possible WAW dependencies betweeen *i* and *j* at *M* may be resolved in any order.

If these conditions are true for all possible combinations of *i* and *j*, then *M* has the WARD property.

Because the WARD property may exist for a specific set of memory locations and/or for a limited amount of time, we refer to the WARD property in terms of regions. A WARD region *r* is constrained in memory space and time such that:

$$r = \big(\{M\}, (t_s, t_e)\big) \tag{2.1}$$

During the time interval from t_s to t_e, the set of addresses $\{M\}$ have the WARD property.

2.4.2 General WARD Example

WARD can be understood by observing the example in Figure 2.2. Note that the ”sync” line in the figure refers to natural synchronization points such as barriers and fork-join points from the fork-join model. We see that each event includes two cores (analogous to the hardware threads in the WARD definition), which both operate on the same variable.

In Event 1, hardware thread *i* writes to the shared variable *val*. After synchronization, hardware thread *j* reads and subsequently writes to *val*. When this situation occurs, a RAW dependency exists at *val*. Therefore, the WARD property does not exist by condition 1 of the definition.

In Event 2, *i* writes to *val*, then *j* writes to *val*. After synchronization, hardware thread *j* writes a new value to *val*. When this situation occurs, a WAW dependency exists at *val*. The program requires hardware thread *j*’s final value to persist via the memory fence. The WAW is not apathetic, so the WARD property does not exist by condition 2.

In Event 3, *i* and *j* again both write to *val*. We observe that are no RAW dependencies between *i* and *j* at *val* because *val* is never read during the event. On the other hand, there is a WAW dependency. However, the program does not provide explicit ordering, so it is safe to resolve the WAW dependency in either order and maintain correctness. Event 3 meets both conditions of the WARD definition, and thus *val* holds the WARD property for the duration of Event 3.

In see the WARD property in action, Figure 2.3 provides an example of how it applies to high-level parallel languages. The pseudocode is for a prime sieve computation that is a simplified version of the *primes* benchmark included later in WARDen’s performance evaluation. It defines a function `prime_sieve_upto` which outputs an array of booleans, `flags`, to mark which integers up to N are prime. For large enough N, the function first computes all primes up to $\lfloor\sqrt{N}\rfloor$ via a recursive call. Then, for each integer p less than $\lfloor\sqrt{N}\rfloor$, if p is prime, it marks every multiple of p as composite. When this process completes, all flags will be correct up to N.

```
// compute all primes less−or−equal to N
// output: array of flags, p is prime if flags[p] is true
def prime_sieve_upto($N$):
  bool flags[] = // size $N+1$, all initially true
  flags[0] = false
  if $N \geq 1$:
    flags[1] = false
  if $N \geq 2$:
    bool sqrtflags[] = prime_sieve_upto($\lfloor\sqrt{N}\rfloor$)
    parallelfor $p$ in range($0, \lfloor\sqrt{N}\rfloor$):
      if sqrtflags[$p$]:
        // $p$ is prime, mark multiples as not prime
        parallelfor $m$ in range($2, \lfloor N/p\rfloor$):
        flags[$p \star m$] = false
  return flags            flags is a WARD region

```

Figure 2.3: **WARD example: Prime Sieve.**
Throughout execution, all instances of `flags` are WARD regions.

In this example, the flags array is a WARD region. The only races on the flags array are write-write races at indices which are multiples of two or more primes, but the same value (the boolean `false`) is always written at each location. Therefore, the writes may be resolved in any order. Consequently, the flags array satisfies both conditions of the WARD definition.

2.5 Transparently Detecting WARD Regions

I now describe how WARD regions can be automatically detected in HLPL programs without programmer annotation or compiler analysis. These results are specifically relevant for MPL [155, 231, 7], a compiler for the Parallel ML language that I focused on for this project.

2.5.1 Conservative Scheduling

For the purposes of connecting logical parallelism (i.e., the fork-join structure) and actual parallelism on the hardware, I assume that the scheduler does not over-parallelize. Specifically, if two instructions occur concurrently on two hardware threads during execution, then the instructions are logically in parallel according to the fork-join structure of the program. All user-level thread schedulers that I am aware of, including those used in languages and systems such as Cilk, OpenMP, and MPL, follow this assumption. The standard retirement process in an out-of-order superscaler core then assures that effects become visible as the scheduler intends.

2.5.2 Disentanglement ⇒ WARD

Through the lens of disentanglement and the heap hierarchy (§2.1.2), I observe that at each leaf heap, all data is entirely local to one thread. Therefore, *all leaf heaps are WARD regions in disentangled programs*. This is a critical point because at every moment throughout execution, approximately *half* of all heaps are leaves, even as the hierarchy grows and shrinks due to forks and joins. (Leaf heaps may become internal due a fork, but then later become a leaf again due to a join.)

Note that internal heaps *may also* be or contain WARD regions, but I do not leverage them. From disentanglement alone, I cannot ensure that the data at internal heaps is WARD. Disentanglement allows for communication through ancestor (i.e., internal) heaps, which may violate the WARD property.

Despite this conservative assumption, my approach still encompasses many memory accesses, including significant benign WAW races. Allocations occur at leaf heaps, so I ensure that all newly-allocated data occurs in regions that are WARD in disentangled programs. For programs that utilize considerable immutable data, WARD also covers all memory writes which initialize new immutable objects. Specifically regarding WAWs, significant races can occur from the language runtime interacting with application memory.

2.5.3 Disentanglement by Construction in MPL

The MPL compiler directly produces x64 executables (no JIT) linked with a runtime system. To ensure disentanglement by construction, MPL offers a standard library consisting of a number of datatypes, including sequences, sets, dictionaries, etc. The library code is implemented under-the-hood via efficient data structures and algorithms, utilizing in-place updates where crucial for efficiency. This approach allows programmers to write efficient parallel algorithms without needing to reason about either race conditions or disentanglement. The resulting programs are disentangled *by construction*.

2.5.3.1 Automatically Exploiting WARD with MPL

In MPL, the runtime system performs task scheduling and automatic memory management. It already exploits disentanglement for improved parallel memory management (especially parallel GC). Again, during execution, all leaf heaps are WARD regions. To take advantage of these WARD regions afforded by disentanglement, I modified two parts of MPL's runtime system: the memory manager and scheduler.

MPL's memory manager implements heaps as linked lists of pages, where allocations are performed within each page via bump-allocation. When a page is exhausted, a fresh page is allocated and the current heap is extended. These pages are always allocated by leaf threads; therefore, whenever a new page is allocated to extend a leaf heap, the page is marked as a WARD region. Pages are later un-marked by the scheduler (which is a standard work-stealing scheduler [30]) at forks, which cause leaf heaps to become internal. At each fork in the program, the scheduler pushes one or more new child tasks onto its work queue and begins working on one of the children with a fresh heap. The scheduler unmarks WARD pages of the current heap before each fork.

2.5.4 Software Engineering Effort

The modifications to MPL are invisible to the application programmer. Also, the implementation effort is minor because much of the logic necessary already exists in the memory management of the language run-time. The total implementation changes to MPL involved adding less than 100 lines of new code.

2.6 WARDen Cache Coherence Protocol

Next, I describe the WARDen cache coherence protocol. WARDen augments a standard MESI (Modified, Exclusive, Shared, Invalid) [157] directory-based protocol with a WARD state W. Figure 2.4 shows the updated directory controller FSA. Cache blocks enter the WARD state when their addresses are contained within WARD regions. When a WARD region is removed, the hardware reconciliation process returns the associated cache blocks to the MESI states. In this dissertation, I discuss coherence messages as defined by Nagarajan, et al. [157].

2.6.1 The WARD Coherence State

To begin, the directory tracks all active WARD regions. WARD regions are therefore defined globally (i.e., WARD regions cannot exist for some cores but not others). WARD region members transition to the new WARD state when they enter the directory or upon the first sharing event (i.e., a request from a second core). The directory effectively disables coherence for cache blocks in the WARD state. In practice, this means that read and write requests are fulfilled without downgrading or invalidating the same block from other caches.

Shared caches (i.e., caches that are used by multiple cores, such as the last-level cache (LLC)) must keep track of the active WARD regions. Using this information, they silently handle any

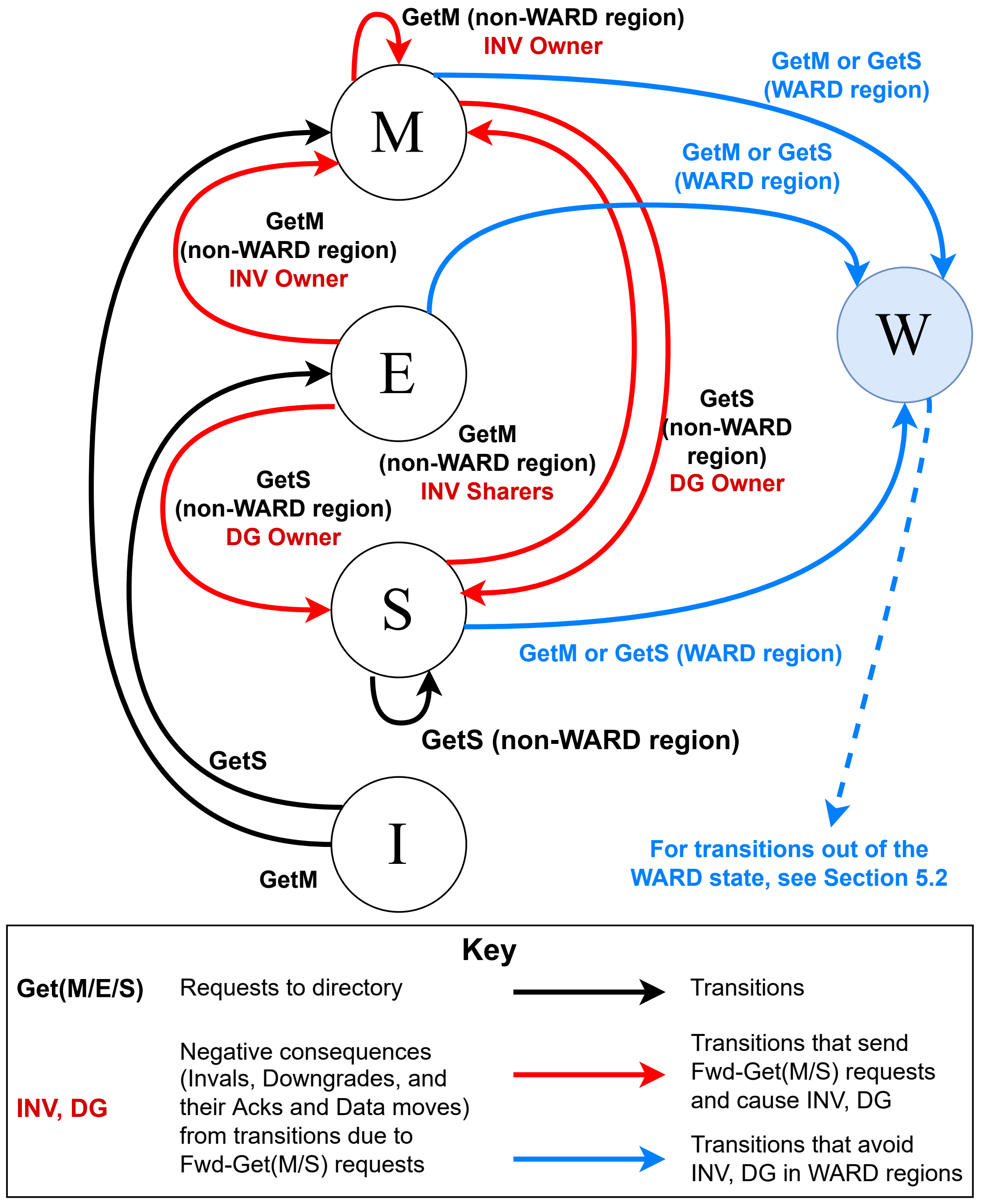

GetM (non-WARD region)
INV Owner
GetM or GetS (WARD region)
GetM or GetS (WARD region)
M
GetM (non-WARD region)
INV Owner
E
W
GetS (non-WARD region)
DG Owner
GetM (non-WARD region)
INV Sharers
GetS (non-WARD region)
DG Owner
S
GetM or GetS (WARD region)
GetS (non-WARD region)
GetS
I
GetM
For transitions out of the WARD state, see Section 5.2
Key
Get(M/E/S)
Requests to directory
Transitions
INV, DG
Negative consequences (Invals, Downgrades, and their Acks and Data moves) from transitions due to Fwd-Get(M/S) requests
Transitions that send Fwd-Get(M/S) requests and cause INV, DG
Transitions that avoid INV, DG in WARD regions

Figure 2.4: **Simplified WARDen Directory FSA.** Put(S/E/M/M+data) transitions/transient states not shown. The WARD state (blue) prevents costly downgrades/invalidations (red) during WARD regions.

upstream requests from private caches. Similar to the directory, shared caches furnish values in the WARD state without inhibiting the use of other copies. For example, consider a cache hierarchy with private L1 caches and a shared L2. If the L1 cache of core A (L1-A) requests write (or read) access to a cache block currently owned by the L1 of core B (L1-B), the request would flow through L2. If the block is not a WARD region member, L2 would invalidate (or downgrade) the block in L1-B. If the block is in a WARD region, L2 would immediately approve L1-A's request without bothering L1-B.

When a shared cache receives a GetS (read request) for a WARD block, it returns an exclusive copy to the requester and hence avoids subsequent upgrade requests even if the same block is concurrently accessed by other cores (e.g., due to false sharing). Thus, WARDen pretends as if the block is private to each core and avoids unnecessary data movement. As a result, private caches can operate as previously defined by the standard cache coherence protocol, and their behavior need not be modified. Leaving private caches unaltered avoids complexity and allows individual cores to operate as quickly as possible. From their perspective, all cache blocks in WARD regions are unused by others.

2.6.2 Reconciliation

When a WARD region is removed, reconciliation merges concurrent updates to cache lines and brings the system to a coherent state. During reconciliation, all WARD cache blocks are placed in the proper MESI state across all cores.

Next, I describe the reconciliation process using three categories: no sharing, false sharing, and true sharing. Note that the directory tracks which cores are sharing each block, and mergers are facilitated by sectored caches (§2.7).

No Sharing: If only one core is holding a block, it has no sharing. Blocks with no sharing can be instantly converted to the Exclusive state. Program correctness is guaranteed in this case because coherence is fundamentally unnecessary for cache blocks that are not shared.

False Sharing: If a block has multiple sharers that modify distinct sectors, it has false sharing. Blocks with false sharing are merged and set to the Shared state. These blocks were shared by multiple cores, but the individual memory locations within them were not. Therefore, each location was written by at most one hardware thread, and that thread's local value is the most up-to-date. To reconcile these blocks, the protocol sets the global value of each location to the local value from the hardware thread that wrote it. Memory locations that were never written are already consistent.

True Sharing: If a block has multiple sharers that modify the same sector, it has true sharing. True sharing occurs when there is a data hazard between separate cores. WARD property guarantees that no RAW hazards will occur, so this form of true sharing is irrelevant. Also, by the WARD property, WAW hazards can be resolved in any order. Thus, reconciliation can merge these lines by convenience (e.g., pick the value processed last by the directory).

Note the reconciliation for false and true sharing use the same mechanism. Their distinction is only for understanding.

2.6.3 Addressing Drawbacks of Cache Coherence

In §2.2.2, I described inefficiencies in modern coherence protocols. First is false sharing, which WARDen addresses through the W state. While a cache block maintains state W, loads/stores to the same cache line by other hardware threads are ignored, so false sharing does not lead to coherence traffic.

WARDen presents two improvements for true sharing scenarios. As with false sharing, the W state avoids coherence traffic. Benign WAW hazards are handled by reconciliation, resulting in a one-time cost, which can be overlapped with computation when eviction occurs before the WARD region ends. Additionally, MPL exposes a less obvious software optimization by unmarking pages at forks. Just before forking, a thread writes information required by the new child thread to execute the forked function (e.g., function pointer, input arguments, etc.) into its heap. When the scheduler unmarks these pages, the corresponding cache lines are effectively flushed from the private caches to a shared cache via reconciliation. The hardware-based reconciliation delay is overlapped with the software-based thread creation delay. Then, the newly-created thread immediately accesses these cache lines faster because it is avoiding downgrades to the previous owner's private caches.

All of these optimizations improve application performance by avoiding invalidations (false sharing, unordered WAWs) and downgrades (false sharing, proactive evictions).

2.7 WARDen System

The WARDen system is my proposed implementation conjoining the WARDen protocol with an HLPL implementation. In this section, I describe the necessary mechanisms to implement the system, and how I simulate it using the Sniper architectural simulator [43].

2.7.1 Proposed Hardware Implementation

Implementing the WARDen system requires "Add/Remove Region" instructions, sectored caches, reconciliation logic, and WARD region storage. WARDen coherence protocol and Parallel ML runtime communicate using two new instructions. The runtime uses these instructions to signal the hardware when a WARD region begins/ends. The addition of two new instructions will have minimal impact.

Sectored caches are necessary for reconciliation to track which memory locations within a cache block are mutated. Sectored caches include additional bits that to track writes at a granularity smaller than the cache block size.

In this proposed implementation, WARDen requires byte sectoring to match the smallest granularity in software. This approach adds one bit for every eight data bits. Caches already include substantial metadata including tag bits, coherence state bits, sharer bitmasks in the LLC, and the overhead of SECDED codes for error detection/correction. Using CACTI 7.0 [19], I estimated that byte sectoring on 64-byte cache blocks adds a cache area overhead of 7.9%. I believe our average speedup (1.46x) is far greater than could be gained using the added area less cleverly (e.g., increasing cache size).

The proposed reconciliation logic is as follows. All WARD cache blocks are flushed (written back as needed and invalidated) from the private caches, and the LLC and directory process requests in the order they arrive. Any sector of a flushed cache block with the write flag set is written back to the shared cache. For the no-sharing and false-sharing cases, no two local copies of a cache block will have the write flag set for the same sector. For the true sharing case, the final value of each sector is taken from whichever cache block is processed last by the LLC; the WARD property guarantees the correctness of this random process.

In practice, WARD regions usually persist long enough to trivialize the reconciliation delay. During evaluation, our prototype reconciled only one block per 50,000 cycles. Therefore, the WARD property is better considered through the lens of coarse-grained data parallelism where regions are larger in time. Also, optimizations outside the scope of this work, such as reconciling blocks in parallel using a multi-banked directory design, could reduce the penalty. For all these reasons, reconciliation can implemented with reasonable overhead.[2]

Lastly, all directories and shared caches must include storage to track active WARD regions. Regions can be stored with 2 pointers (16 bytes) that indicate their beginning and end. To enable efficient lookups, the simulator models the storage as fully associative caches implemented using

[2]Due to this minimal overhead, the simulator estimates the reconciliation overhead cost by performing a cache flush.

CAM-like structures.

To perform a lookup, the system uses the CAM's per-bit equality comparator to determine the most significant bit that differs from between the region boundary and the address. Then, it checks the value of the differing bit. If the address bit is 1, the address is greater than the region boundary. It follows that if the address bit is 0, the address is less than the region boundary. To pass the check, an address must be greater than the lower bound and less than the upper bound.

This logic will require slightly more area than a standard CAM, but it is substantially simpler than the more complicated TCAM since it is not comparing the results across non-paired entries. If an address is somehow found in more than one region, it is clearly safe to just mark it as WARD.

Again using CACTI, supporting 1024 simultaneous regions would require less than 0.05% additional area. This design allows WARD regions to exist for unlimited periods of time, only bounded by the software's directives.

Overall, these results indicate that WARDen is feasible.

2.7.2 Interoperability

WARDen and its proposed implementation are designed to maintain interoperability with pre-existing/non-HLPL programs and space/time-shared multiprocessor systems. Non-WARDen programs will not signal WARD regions to the hardware, and thus all their coherence traffic will be handled according to the standard MESI protocol. When multiple WARDen programs are running simultaneously, each will signal their own WARD regions and remain oblivious to each other's regions. HLPL programs use a single process with many threads, avoiding any interprocess memory regions.

Our WARDen system is also capable handling more complex system interactions. To illustrate, consider context switching. If a WARDen program is context-switched out, reconciliation is

```
/* Ran on two separate cores (myself and partner) */
while (iterations--) {
        while (buf != partnerID) ;
        buf = myID; }

```

Figure 2.5: **True Sharing Microbenchmark Kernel.**

Table 2.1: **Validation of Sniper Model (Latencies in Cycles).**

Scenario	Real HW Latency	Simulated Latency
Same core	8.738	11.21
Diff. core, same socket	479.68	286.01
Diff. core, diff. socket	1163.23	1213.59

initiated. Applying reconciliation will not noticeably increase the context switch delay. Reconciliation is handled by the directory, meaning a processor can initiate reconciliation and immediately continue execution on the different workload unless it needs to access the memory region being reconciled, which should rarely/never happen. Reconciliation is also completed per-cache-block, so if the new workload accesses a few of the same addresses (e.g., a small false sharing event), it does not have to wait for the whole WARD region to be reconciled. I cannot cover all possible system interactions here, but in general, reconciliation enables safe and efficient interoperability.

2.7.3 Simulated Prototype

To project the performance improvement and energy savings of the WARDen system, I implemented it within the Sniper multicore simulator [42, 43]. I employed the latest version of Sniper [43], which uses an interval simulation technique capable of modeling fine-grained coherence events. I show that Sniper correctly models the latencies of data movement using a true sharing microbenchmark. The kernel of this microbenchmark is in Figure 2.5.

I ran the microbenchmark on a test system with two Intel Xeon Gold 6126 processors in sepa-

Table 2.2: **Simulated System Specifications.**

Parameter	Value	Parameter	Value
L1 Size	32 KB	L1/L2 Associativity	8
L2 Size	256 KB	L3 Associativity	20
L3 Size (per core)	2.5 MB	L1/L2/L3 latencies	6-16-71 cycles
Cache Block Size	64 B	Frequency	3.3 GHz
Cores per Socket	12	Intersocket latencies	vary, see text

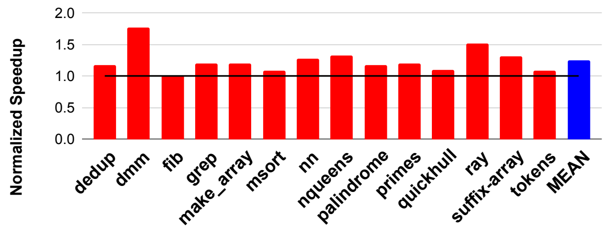
rate sockets. Each processor has an L1-L2-L3 (latency in cycles: 6-16-71) cache hierarchy, where L1 and L2 are private and L3 is shared. I configured Sniper identically to match this system. I evaluated the true sharing microbenchmark in three scenarios, in which the competing hardware threads are on (1) the same core, (2) different cores but the same socket, and (3) different cores and different sockets. I measured the cycles per iteration over 100 million iterations of the benchmark.

I ran each scenario 10 times and report the measured averages for real and simulated hardware in Table 2.1. Both the real hardware and simulated latencies align closely. While the simulator latencies are not identical to that of real hardware, they allow for accurate relative comparisons.

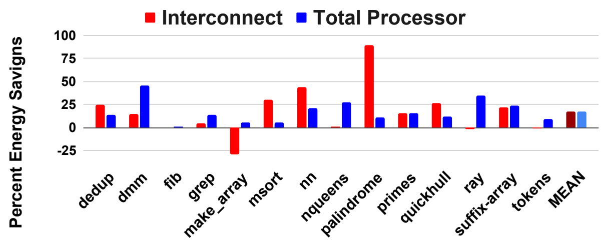
Next, I proceed to evaluate WARDen on larger scale benchmarks.

2.8 Evaluation

I compared the performance and energy of standard MPL binaries with the MESI cache coherence protocol versus the WARDen protocol using Sniper and measured energy consumption using the McPAT [139] power model included with Sniper. Table 2.2 shows the configuration of the machine and model. For my evaluation, I studied one and two socket versions and likely future hardware: many-socket and disaggregated systems. As a whole, these results show how WARDen can transparently improve application performance on a wide variety of modern systems.



(a) Performance (Speedup)



(b) Energy

Figure 2.6: **WARDen Single Socket Results.**

2.8.1 Evaluation Methodology

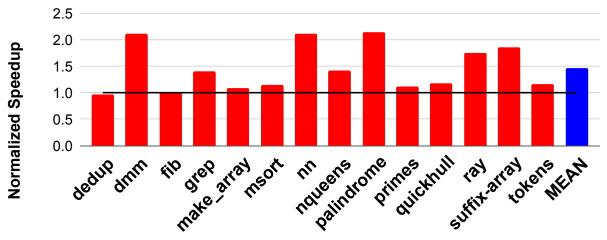
I evaluated the combination of MPL and WARDen using the PBBS benchmark suite [204]. PBBS is a well-cited suite that includes benchmarks from many problem domains, including graph analysis, numerical algorithms, text/image/audio processing, computational geometry, and computer graphics. These benchmarks have been ported to Parallel ML and compiled using MPL, ensuring disentanglement. Many of these benchmarks are ported from state-of-the-art C/C++ benchmarks in the PBBS benchmark suite [204], and all are highly optimized.

I tuned the benchmark input sizes so each executes without simulation in .1-.5 seconds, resulting in execution times on the simulator between .5-4 hours. I chose these times to feasibly explore multiple configurations.

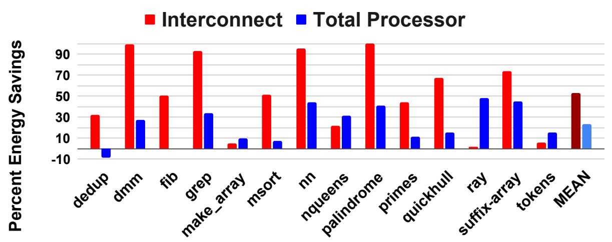
2.8.2 Modern Hardware

2.8.2.1 Single socket

First is a single socket version of the configuration shown in Table 2.2. As shown in Figure 2.6(a), WARDen produces speedups of 1–1.8x, with a mean speedup of 1.24x. Figure 2.6(b) shows total processor and interconnect energy gains. There is more variation in these, but the averages are 17.4% and 17.3%, respectively. Total processor energy decreases with WARDen in large part be-



(a) Performance (Speedup)



(b) Energy

Figure 2.7: **WARDen Single Socket Results.**

cause execution time decreases. Network energy decreases due to the smaller number of coherence messages and data transfers. The performance/power overhead of tracking/reconciling WARD regions negatively impacts the results for benchmarks which benefit minimally from WARDen (e.g., make_array).

2.8.2.2 Dual socket

Next, I show how WARDen scales to a dual socket system with two of the single-socket processors. Figure 2.7(a) shows that WARDen produces speedups of 1–2.1x with a mean of 1.46x. These speedups are higher than those from the single socket case, suggesting that the benefits of WARDen scale with machine size. There is also more separation between benchmarks that benefit from WARDen (e.g., palindrome) and those who do not (e.g., dedup).

As shown in Figure 2.7(b), energy savings increase on the dual socket system compared to the single socket system. For this case, interconnect energy savings, with a mean of 52.9%, outpace the total energy reduction, with a mean of 23.1%, and in some cases, they are the sole driving factor in the cumulative decrease. This result is expected because coherence messages are now passed between sockets, therefore consuming far more energy in the network. WARDen is able to eliminate many of these messages. Across the less-accelerated benchmarks, we do not see a negative effect greater than a 5% power increase.

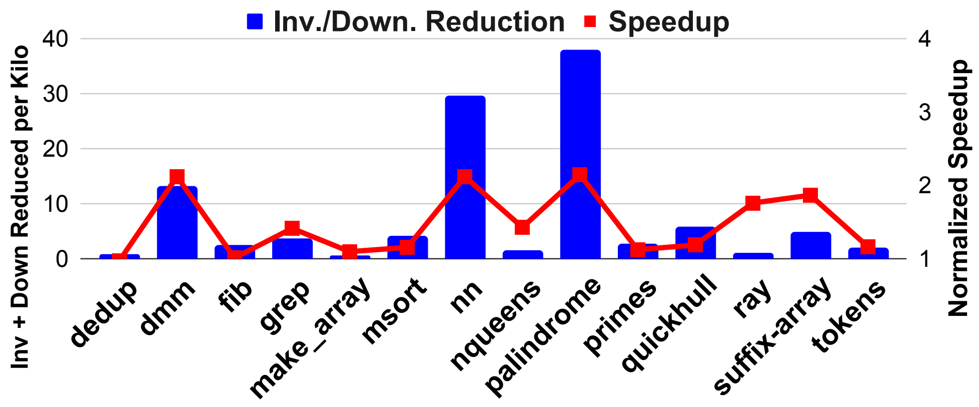


Figure 2.8: **Dual socket speedup w/ Reduction in Invalidations & Downgrades.**

2.8.2.3 Analysis

To show that the improvements to cache coherence generate the observed speedups, I studied the dual socket results in greater detail. For all the benchmarks except *tokens*, WARDen recognizes the vast majority (90%+) of memory accesses as occurring in a WARD region. There is no correlation with the performance results because many accesses are to private variables for which there will be no coherence traffic anyway. To understand the effect of WARDen, I instead focused on the memory accesses that would incur costly downgrades or invalidations with a standard coherence protocol.

Figure 2.8 compares the dual-socket speedups with the reduction in invalidations and downgrades, counting the number of invalidations/downgrades avoided per 1000 instructions executed. Note that invalidations and downgrades are counted per cache, so a single execution may cause many invalidations or downgrades throughout the cache hierarchy. Figure 2.8 shows a positive correlation between reducing costly memory events and speedup. For many benchmarks, WARDen avoids invalidations and downgrades, which in turn accelerates performance. Conversely, benchmarks with small reductions of these events show little speedup.

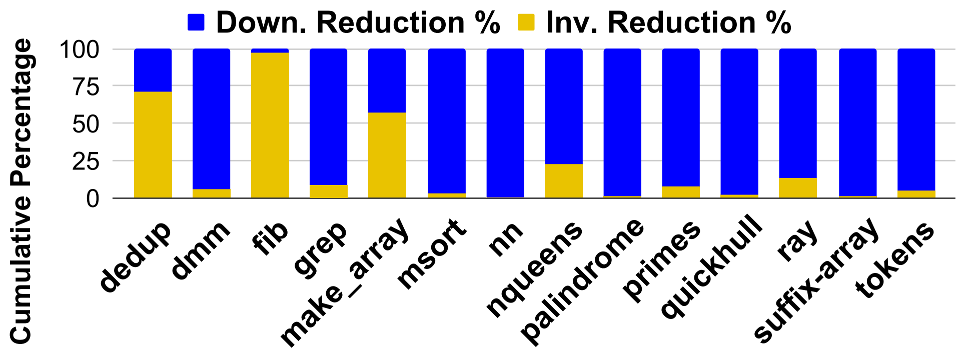


Figure 2.9: **Percent of Reduction by Invalidations and Downgrades.**

A few measurements in Figure 2.8 appear anomalous. Three benchmarks (*nqueens*, *ray*, and *suffix_array*) show significant speedups, yet have relatively small reductions in coherence events. Meanwhile, *fib*, *msort*, *primes*, and *quickhull* underperform given what might be expected given their relatively large reductions in coherence events.

To dive deeper, Figure 2.9 shows the breakdown of invalidations and downgrades by percentage. Downgrades are generally more important than invalidations for application performance because they affect load operations. Loads are *blocking* operations that pause dependent computation. In contrast, invalidations are caused by store operations, which are injected into the processor's store buffer and commit without waiting for their completion in the memory hierarchy. Unless the store buffer is full, which is relatively rare, store latency (and hence invalidation latency) does not impact execution. Figure 2.9 shows that *Nqueens*, *ray*, and *suffix_ray* mostly avoid downgrades (77.7%, 86.4%, and 98.3%, respectively). Contrast these results with a more subtle outlier, *fib*. *Fib* experiences a significant reduction in negative coherence events but does not see any appreciable speedup. This likely occurs because *fib* has the lowest percentage of downgrades out of all benchmarks (2.65%).

Now, I explain the results for *msort*, *primes*, and *quickhull*. All three mostly avoid downgrades, so I would expect more substantial speedups. However, the performance of these benchmarks is

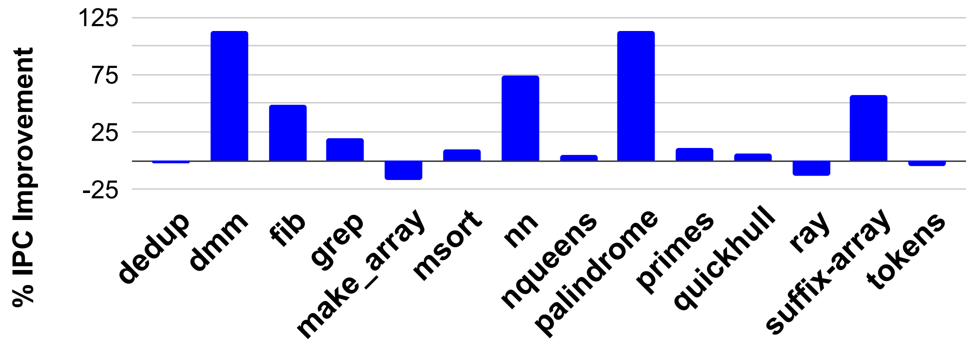
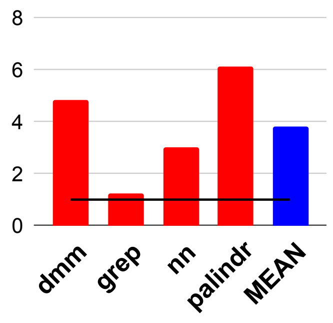


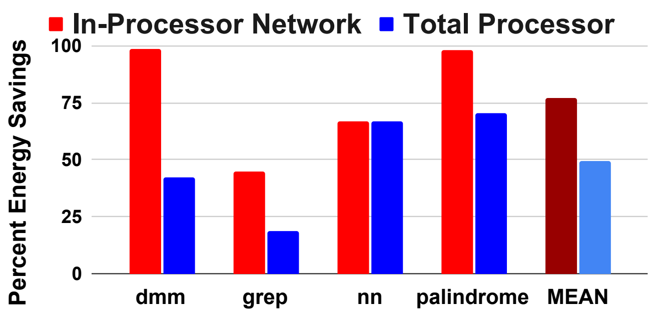
Figure 2.10: **Percentage IPC improvement.**

not bound by coherence events. To understand these benchmarks, I plot the percent IPC improvement generated by WARDen in Figure 2.10. IPC helps us to understand the application's ability to continue executing instructions despite coherence delays. Benchmarks with low IPC improvements do not take advantage of the faster memory accesses provided by WARDen, indicating their speedup should be lower. Surely enough, all of *msort*, *primes*, and *quickhull* show minimal IPC improvement from avoiding downgrades and invalidations.

Perhaps the most interesting measurement in Figure 2.10 is *ray*, which shows an IPC reduction despite its large speedup. I believe this IPC result indicates an improvement to synchronization delays. The PBBS benchmark suite uses busy-waiting synchronization primitives implemented using *compare_and_swap* atomics. Busy waiting involves executing many cheap read/write instructions. WARDen's improvements help individual threads reach synchronization points more quickly and evenly, eliminating fast waiting instructions, and thus lowering the IPC despite improving application performance. *ray* enjoys a 49.5% reduction in load instructions executed that justifies this claim. This reduction of instructions executed also further supports the reported speedup.



(a) Speedup



(b) Energy

Figure 2.11: **Performance and Energy Improvements on Disaggregated Hardware.**

2.8.3 Future Machines

2.8.3.1 Many Sockets

In the future, systems with many sockets will become more commonplace. Programs written in HLPLs are a great candidate to run on such machines because they allow expression of algorithmic parallelism, making readily available the higher levels of parallelism such machines require. In this case, interconnect latencies (like those in Table 2.1) will continue to rise. The advantages of WARDen will become even more prevalent in such a situation. To better understand why, consider another likely future hardware scheme: disaggregation.

2.8.3.2 Disaggregated

I modeled a 2 node, disaggregated system, with a remote access time of 1 μs. This time is conservative, outstripping the performance of state of the art systems [41, 93]. For this study, I included the most promising benchmarks from the previous section.

Figure 2.11 shows that benchmark speedup improves dramatically, to about a mean of 3.8$\times$, on the disaggregated system. As seen in Figure 2.11, network energy savings improves to a mean of $\sim$77.1% and processor energy savings improves to a mean of $\sim$49.5%. This result makes sense because the disaggregated system has a L3 miss penalty more than 3$\times$ greater than the standard dual socket system. Coherence downgrades and flushes are therefore more costly, which in turn makes WARDen more valuable.

A subtle result of this experiment is that disaggregation appears to widen the performance-improvement gap between benchmarks that drastically benefit from WARDen and those with more middling results. *Grep* was the weakest performer of the selected benchmarks in the experiments simulating standard hardware, and its speedup did not grow nearly as much as the others in the disaggregated case.

disaggregated systems, which have far less consensus on how to implement shared memory, are a great subject for further study.

2.9 WARD in Low-level Parallel Programs

While WARDen was developed in the context of high-level parallel languages, it may also be useful more generally because the WARD property occurs in lower-level programs as well. I contributed to a tool named CARMOT that analyzes the hot loops of a program that has been lowered to LLVM [66]. I applied CARMOT to popular low-level benchmark suites, including NAS 3, SPEC '17 and PARSEC 3.0. Our tool *conservatively* determined that 96% of the variables in the hot loops of these benchmarks displayed the WARD property, suggesting low-level codes could also benefit from WARDen. Transparently recognizing WARD regions in these programs is an exciting avenue for future work.

2.10 Conclusion

Lower-level parallel languages have been implicitly and explicitly co-designed with hardware to varying degrees for decades. HLPLs present the opportunity to explore new co-design opportunities. WARDen shows that it is possible to *transparently* optimize shared memory communication (via cache coherence) for HLPLs. WARDen was published at the CGO'23 conference [238]. For interested readers, the artifact for this work is available online [237].

CHAPTER 3

AUTOTUNING COLLECTIVE ALGORITHM SELECTION

My work to accelerate shared-memory systems (WARDen) proved it is possible to transparently improve communication for high-performance applications. However, many programs require far more parallelism than is possible in a single shared memory domain. These applications run on massive distributed memory systems like supercomputers. For these applications, I focused on optimizing the Message Passing Interface (MPI), which is the de facto standard for communication in high-performance distributed applications. Specifically, I studied the collective operations (i.e., collectives) in MPI, which are its most popular primitive.

3.1 Collective Algorithm Selection

3.1.1 Importance

Due to their popularity, collectives represent more than a quarter of overall execution time of applications on supercomputers [60], so improving collective performance will, in turn, improve application performance. However, doing so is a significant challenge. Unlike point-to-point communication, collective's higher-level specification conceals the actual communications occurring, thus creating uncertainty in their performance. MPI implementations include multiple implementations (i.e., algorithms) for the same collective. The best algorithm for a given collective is highly dependent on the situation, and using the wrong algorithm can significantly hinder performance.

Table 3.1: **Collective Performance Variables.**

Type of Variable	**Examples**
Programmatic	Message Size Number of Processes (N) Processes per Node (PPN)
Non-Programmatic	Library Versions Node Topology CPU Performance Network Bandwidth/Latency/Congestion

3.1.2 Challenge

Collective algorithm selection is challenging due to the quantity and dynamicity of influential factors. I separate these factors into two categories shown in Table 3.1. First are programmatic variables that are manually set in software. The programmatic variables are message size, number of processes (N), and processes per node (PPN). The other category is non-programmatic variables, which are innate to the hardware and software outside the target application. All of these factors in both categories must be weighed when selecting the optimal algorithm for a given collective; the wrong choice can result in a slowdown by a factor of two or greater.

To illustrate the complexity of algorithm selection, I present two algorithms for *MPI_Bcast* in Figure 3.1. Figure 3.1a is a serial broadcast: root node 1 sends the message to nodes 2, 3, and 4 in order. Figure 3.1b shows a binomial tree implementation where child nodes forward the message in parallel. In this case, node 2 can send the message to node 4 while node 1 sends the message to node 3. It may seem like the tree algorithm is always the superior choice. However, for very small message sizes, node 1 may be able to push all of its messages into the network before node 2 even receives the information. In this case, the serial implementation will perform faster. In other cases, yet more algorithms must be considered. For large message sizes, an algorithm that performs a scatter followed by an allgather has been shown to be superior [216].

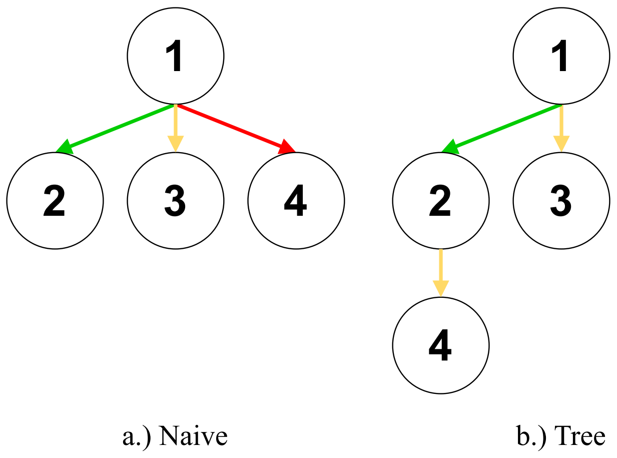


Figure 3.1: **Example MPI_Bcast Algorithms.** The Tree algorithm overlaps the 1→3 communication with the 2→4 communication. The correct algorithm to use depends on a multitude of factors.

It is clear that picking the optimal algorithm requires an inordinate amount of expert knowledge. To alleviate this burden from the application developer, a transparent solution is necessary.

3.1.3 Existing Transparent Approaches

There are many proposed methods to transparently perform collective algorithm selection. To select the correct algorithms, the most popular open source implementations of the MPI standard—Open MPI [91], MVAPICH [168], and MPICH [154]—use heuristics. I experimented with MPICH because it serves as the basis of many popular production MPI libraries. The most prominent example is Cray MPI, which is the primary MPI implementation on the supercomputers studied in this work, including Oak Ridge National Laboratory's *Frontier*, the most powerful supercomputer in the world. In common with the other implementations, MPICH's heuristics are outdated and inaccurate. In 2020, Hunold et al. [109] found that optimized selections can accelerate collectives by **35–40%** compared with the default heuristic approach.

Autotuners improve upon the existing heuristics by using either analytical calculations or results from empirical evidence (i.e., benchmarking the candidate algorithms and picking the best). Much of this work has focused on analytical models that can project algorithm performance [225, 77, 142]. These models show varying performance results, and they have failed to gain widespread adoption because they are difficult to implement, maintain, and expand for new algorithms. Other tools such as Intel's MPITune and Open MPI's OPTO [47] use exhaustive benchmarking to find the best algorithms. This strategy maximizes accuracy, but it requires so much data (and thus machine time) that it can only be deployed to tune individual scenarios on large scale systems.

Machine learning (ML) promises the best of worlds. Hunold et al. showed that a random forest ML model can improve upon exhaustive approaches by learning to predict scenarios that have not been benchmarked [111], lessening the benchmarking overhead. They also have an inherent advantage over analytical models because it can learn patterns in the data caused by factors that are difficult to model analytically, such as real-time and/or machine-specific influences.

3.1.4 Limitations of ML Approaches

While ML has the potential to improve collective algorithm selection, the state-of-the-art ML system [109] is not ready for production use. To begin, it is difficult for a prospective user to compare ML autotuners and other existing approaches because there is no framework to quantify performance differences. Also, the state-of-the-art ML autotuner still requires an intractable amount of benchmarking data for training. The evaluation in [109] uses an ad-hoc method to generate training and test sets with a small number of nodes (up to 48). The proposed method is a strong proof of concept, but it does not generalize to encompass the performance of an entire system. The standard approach would collect data for the entire feature space, then randomly split the data into training and test sets. This process seems straightforward, but collecting this data for a larger system is intractable. For example, on 512 nodes on Argonne National Laboratory's Theta supercomputer, data collection alone would take approximately 75,000 core hours, which is over 6 days of machine time. The cost of data collection is so large dominates the machine learning inferencing and prediction costs, rendering them both negligible, and invalidating ML autotuners as a practical solution.

In my thesis research, I initially set out to address these challenges: 1.) design new statistics to properly quantify ML autotuner performance, and 2.) make them practical for large-scale systems.

3.2 Quantifying Performance

The researchers in [109] evaluate their work by benchmarking every algorithm for their test data points. They use this data to simulate the performance of selections by the MPI library's default method, their ML autotuner, and an oracle that always selects the best algorithm. They show sometimes significant (1.3-1.5x) speedup over the default selections. However, they do not comprehensively compare to the oracle. The omission of this comparison makes it difficult to understand the autotuner's actual performance.

Average speedup over the default (i.e., Average Speedup) does not paint the whole picture with respect to applications. For example, programs use only a small fraction of the feature space. Average speedup could cover the weaknesses of a high-variation autotuner, which makes some very good selections and some very bad selections. If an unlucky application only uses scenarios with bad selections, they could actually suffer from the autotuner.

To better quantify autotuner performance, I designed the set of metrics in Table 3.2.

3.2.1 Metrics

Each metric encapsulates an important dimension of autotuner performance. First is R^2 Score, a generic value that represents ML model fitness. An R^2 Score close to one means the individual

Table 3.2: **New Autotuner Performance Metrics.**

Metric	Definition	Range of Values
R^2 Score	General statistic describing how well the model fits the data	[0,1] (closer to 1 is better)
Average Selected Algorithm Slowdown (Average Slowdown)	Slowdown of the selected algorithm compared to the optimal/oracle algorithm averaged across all feature sets	[1,∞] (closer to 1 is better)
Classification Accuracy	Proportion of feature sets where the ML model accurately predicts the fastest algorithm	[0,1] (closer to 1 is better)
Significant Mistake Proportion	Proportion of feature sets where the predicted algorithm is more than 10% slower than the fastest algorithm	[0,1] (closer to 0 is better)

regression models capture the trends in the dataset. This metric is useful because it indicates how the model may perform on untested scenarios or more difficult areas of the feature space.

The most commonly used metric is Average Selected Algorithm Slowdown, or Algorithm Slowdown for short. Average Slowdown represents the expected inefficiency of the autotuner's selections compared to optimal. This metric is most useful when making a comparison between autotuners because it represents the performance of selected collective algorithms across the entire feature space.

The next metric is Classification Accuracy, which represents the chance that a selection will be the optimal algorithm. This metric is useful to measure model performance in the presence of outliers. For example, a model may perform very poorly for edge cases that go unused in applications. This model would have a poor Average Slowdown, but the Classification Accuracy would more fairly represent its high performance.

Lastly, I include Significant Mistake Proportion to represent the chance that a selected algorithm will be significantly ($> 10\%$) slower than the optimal selection. Significant Mistake Proportion is a valuable statistic to decide whether an autotuner is "good enough" for users because it shows the chance a selected algorithm will perform noticeably worse than it should. Because applications only use a small fraction of the feature space, a small Significant Mistake Proportion guarantees that they will see near-optimal performance from the autotuner. It is important to note that 10% is an arbitrary value and can easily be changed.

These metrics provide a much better understanding of autotuner performance than Average Speedup. I completely omit Average Speedup because default selection is much better on some hardware systems than others. Attempting to compare autotuners from separate sources based on Average Speedup can be actively misleading because it depends more on the difference in default performance. Instead, my metrics should be applied separately to an autotuner and the default and then compared.

3.2.2 Previous Work Re-Evaluation

To illustrate the usefulness of my metric set, I re-implemented the existing work in [109]. I exhaustively benchmarked the standard collectives for all power of two feature values up to 64 nodes, 32 process per node, and 1 MB messages. I selected 64 nodes because it is the closest power of two greater than the 48 nodes used for evaluation in [109]. Then, I trained the ML autotuner using randomly selected training data. The test set includes all of the collected data points. I used the *RandomForestRegressor* from the *scikit-learn* package [172] as the ML model. Because I benchmarked every algorithm in MPICH for every feature set, I knew which algorithm is optimal in every scenario. I then calculated the new metrics using the autotuner's selections and the optimal selections. For comparison, I also simulated the selections MPICH would make by default and found an Average Slowdown of 1.3. This means that there is headroom to improve on the MPICH selections and speed up collectives by 23.1%.

I performed these experiments on the Bebop cluster at Argonne National Laboratory. I used

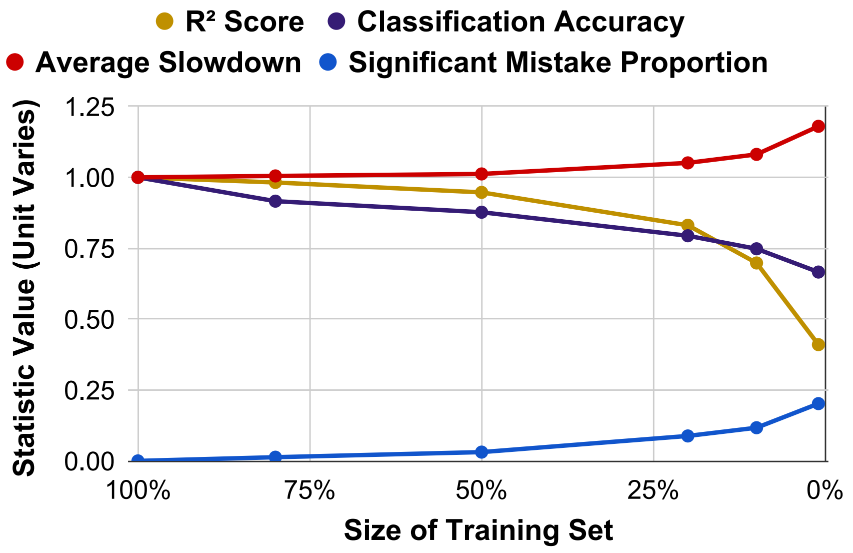


Figure 3.2: **Previous State-of-the-Art Autotuner Performance.** Shown as a function of training set constriction.

a 64-node subset of the 664 standard nodes. Each node contains an Intel Xeon-E5-2694v4 with 36 cores (I used up to 32 cores) and 128GB of DDR4 memory. The results of this evaluation are summarized in Figure 3.2. I repeated the experiment with smaller training sets to understand how the ML autotuner performs when I train it with more realistic amounts of data. The left side of the graph confirms the results from the previous work: the ML autotuner performs exceptionally well with copious training data. With an Average Slowdown near 1 and a Significant Mistake Proportion near 0, applications would enjoy near optimal collective performance. However, the picture becomes increasingly tainted as I move to the right in the table. The results stay stable for 50%, steadily deteriorate at 20/10%, and completely collapse at 1%. 10% is the upper limit of what may be feasible on a larger scale system (1.82 machine hours). With 10% of the feature space for training, an Average Slowdown of 1.10 is substantially sub-optimal, and a 0.12 Significant Mistake Proportions means many applications would see noticeable performance deterioration.

I conclude that when trained with a realistically sized training set, the state of the art provides little practical value. These results showcase the need to reduce the amount of data required to train an accurate ML autotuner.

3.3 Minimizing Data Collection w/ the FACT Approach

To reduce the necessary training data to create a useful ML autotuner, I created the FACT methodology. FACT is an acronym of its four steps: **F**eature Scaling, **A**ctive Learning, **Converge**, **T**une Hyperparameters. FACT builds upon the existing state of the art [109] (i.e., training a random forest model using microbenchmark results) with several advancement that together reduce the training data needed for a useful model by 6.88x. Below, I describe each component.

3.3.1 Feature Scaling

Data preprocessing is a ubiquitous step in machine learning applications. Preprocessing allows the developer to use domain knowledge to help expose data patterns to the ML models, greatly improving model accuracy. More specifically, the developer processes the data to eliminate anomalies that may mislead the learner. One of the most common preprocessing steps is feature scaling. Feature scaling is vital for collective autotuners because some regression models typically treat larger feature values as more important by construction. In algorithm selection, message size has a much bigger range than number of nodes or processes per node, but the model should treat them all equally.

To reduce our feature ranges equitably, I apply log_2 scaler to the values. I then add one to avoid feature values of zero, which are also known to confuse some models. Through this feature scaling approach, value ranges become much more similar. Table 3.3 summarizes the ranges before and after scaling.

Scaling the input values has a straightforward solution, but the output values present a more

Table 3.3: **FACT Feature Ranges.**

	Range	Scaled Range
N	[1,512]	[1,10]
PPN	[1,32]	[1,6]
MSG SIZE	[1, 1MB]	[1,21]

complicated case. The range of output values is quite large, from a few microseconds for small inputs to a full second or more for large inputs. In this scenario, a regression model may treat outputs for small feature values as essentially the same. However, our metrics normalize the output values for each feature set to the optimal algorithm. A difference of a couple of microseconds may be a significant slowdown/speedup for small feature values, and the model must maintain this information.

The most common approaches for scaling are standardization and normalization. In short, standardization assumes the data fits a normal distribution and re-scales it to a mean of 0 and a standard deviation of 1. Normalization is a uniform scaling technique that compresses the data into the range [0,1] without affecting the shape of its distribution. Another approach to consider is copying the metrics and scaling each output to the value of the fastest algorithm. I refer to this technique as *algorithm scaling*. Finally, an additional log_{10} scaler to algorithm scaling could further improve performance by matching the scaling pattern already applied to the inputs.

To evaluate these options, I repeated the experiments from Figure 3.2 with input scaling and each of the output scaling options applied. The results are shown in Figure 3.3. To summarize, my custom solution applying log_{10} with algorithm scaling outperforms the competitors. To understand why, consider the pitfalls of each option. Standardization performs poorly because it assumes that the data fits a normal distribution, which is not true for this data set. Normalization fails because the input scaling complicates the input-output relationship by artificially introducing an exponential component, and normalization passes this complexity on to the ML model. Given the

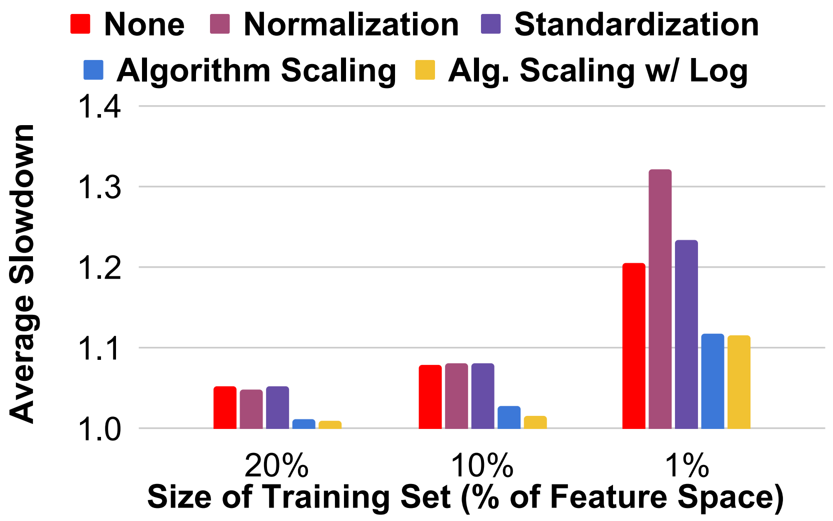


Figure 3.3: **Preprocessing Techniques Comparison.** Algorithm Scaling w/ Log produces the best (lowest) Average Slowdown every training set size.

weaknesses of the other preprocessing schemes, the custom solution using algorithm scaling with an additional log_{10} scaler is the clear winner. It performs well because it scales the output to make relative differences in performance more important, and it also eliminates the exponential relation introduced by the input scaling technique.

By combining the custom feature scaling techniques, model performance significantly improves. Figure 3.4 compares the new preprocessing to the original solution, which did not preprocess the data; expectedly, all metrics improve. With a feasible amount of training data (1% of the feature set), Average Slowdown decreases by 35.7%. Meanwhile, Significant Mistake Proportion decreases by 14%. Interestingly, Classification Accuracy only sees a relatively modest uplift (4.3%). Considering the trends across statistics, preprocessing improves the model not by making more correct selections, but by making smaller mistakes when it does miss-select.

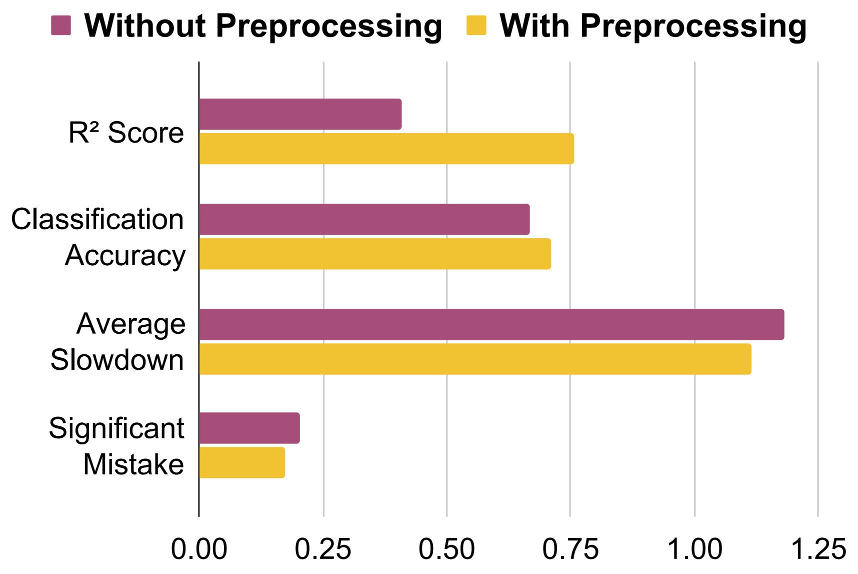


Figure 3.4: **Benefit of Custom Preprocessing.** Preprocessing creates significant improvement across all metrics.

3.3.2 Active Learning

Active learning is a type of machine learning where the learner interactively queries the data set [199]. It is an iterative process that begins with a set of unlabeled data. In each iteration, the learner chooses data points to be labeled by an oracle and adds them to its training set. Then the process repeats, and the model selects new points. Training continues until the model accuracy converges or a time limit is reached. Active learning is typically used for ML applications where data labeling is a challenging or time consuming task. Typical use cases include situations that require human intervention (e.g., text/speech/image recognition), but the same description applies to algorithm selection.

For point selection, I used the most common strategy: uncertainty sampling [136]. This algorithm queries data points that it is most uncertain how to label. However, the random forest model does not report prediction uncertainty. To estimate uncertainty, I fit a Gaussian process surrogate model [17] to report features values at which it is most uncertain.

To use active learning to train a collective autotuner, I specified an unlabeled data space using the input features and their scaled ranges. During each iteration, I looked up the execution times of the chosen points using the exhaustive benchmark results to simulate data collection. After each iteration, I tested model accuracy to check if I had met the convergence criteria. To approximate performance indistinguishable from the oracle, I used a convergence criteria of Average Slowdown below 1.03 and Significant Mistake Proportion below .05.

3.3.3 Hyperparameter Tuning

Hyperparameter tuning is a process where the parameters of the learning model (hyperparameters) are optimized. All of the most common learners have many hyperparameters, and using the optimal values can greatly improve model accuracy. For example, hyperparameters of the random forest model include the number of decision trees and the max depth of the trees.

In theory, hyperparameter tuning should be performed every time a regression model is trained, so as to maximize accuracy. However, similar to data set collection, searching the hyperparameter space is time consuming. To perform hyperparameter tuning during the active learning iterations before testing for convergence, data collection would have to pause while the model retrains with many combinations of hyperparameter values. In my experience, collecting more data improved model accuracy far quicker than hyperparameter tuning. For this reason, I include hyperparameter tuning as a postprocessing step in the FACT methodology. This way, it can be performed offline or on a single node, not expending the data collection resources. Active learning instead uses fixed hyperparameter values that are derived from previous offline tunings.

It is best to think of hyperparameter tuning as an optional step to squeeze the last bit of accuracy

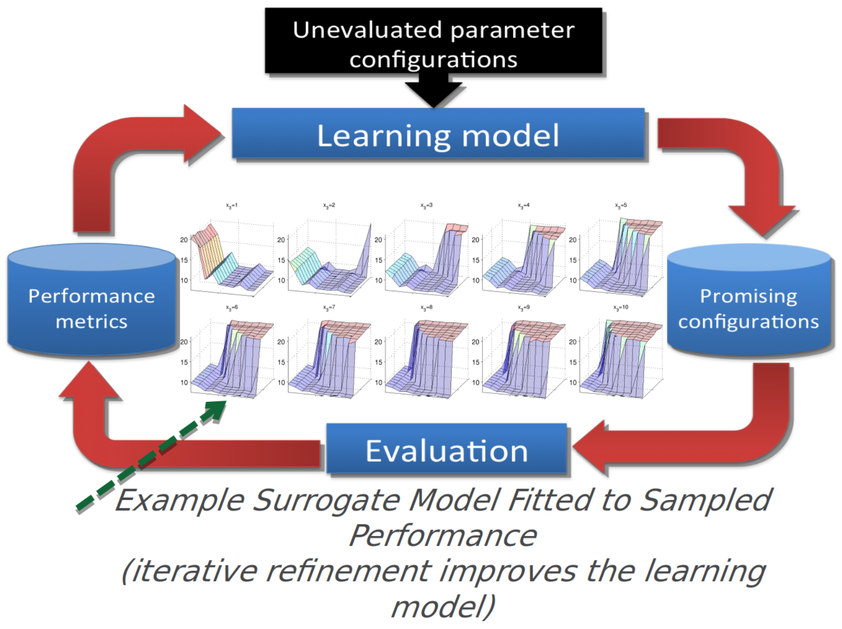

Unevaluated parameter configurations
Learning model
Performance metrics
Promising configurations
Evaluation
Example Surrogate Model Fitted to Sampled Performance
(iterative refinement improves the learning model)

Figure 3.5: **DeepHyper System Diagram**. [18]

out of the model after data collection. This step is most valuable when the user only has a fixed amount of time to collect training data. In this case, the active learning process may not converge, resulting in inaccurate models. Hyperparameter tuning can bridge the gap and produce a model with converged-level accuracy. It is through this lens I later evaluate hyperparameter tuning.

3.3.4 Implementation

I created an initial prototype of the FACT methodology that integrates the main three ideas from this section.

The prototype performs data collection using the Ohio State University (OSU) microbenchmark suite [166]. The OSU benchmarks are a widely accepted suite for benchmarking MPI collectives. Once the data is collected, it is preprocessed and used to train the *RandomForestRegressor* from the *scikit-learn* Python package [172].

For active learning, the prototype employs a special instance of the DeepHyper tool [16, 18]. DeepHyper is primarily an automatic hyperparameter tuning tool. It works by iterating through hyperparameter configurations, training an underlying (surrogate) model. The surrogate model maps the hyperparameter configurations to a result statistic defined by the user (e.g., classification accuracy or R^2 score). DeepHyper balances two modes: *exploration* and *exploitation*. During the exploration phase, it queries the surrogate for the hyperparameter values with the most uncertainty. It then trains the target model with those values, tests its performance based on the user-defined metric, and uses the results to retrain the surrogate model. The "exploitation" phase then uses the surrogate model to predict which hyperparameter values will maximize the performance of the target model. Figure 3.5 illustrates DeepHyper's inner workings.

Exploration phase of DeepHyper is very similar to the active learning process. The prototype uses DeepHyper's framework for active learning as follows:

- I set my autotuner's random forest model as the target model (DeepHyper's surrogate model is a Gaussian process).
- I set DeepHyper's β value to ∞, which forces DeepHyper to always stay in exploration mode.
- I define DeepHyper's performance criteria as the execution time of the collective algorithm.
- I specify the input features (N, PPN, message size) as the hyperparameters for DeepHyper to optimize.
- To train the target model, I instruct DeepHyper to run the selected microbenchmark and report the execution time.

By manipulating DeepHyper, I implemented a custom active learning method with much less development effort than an ad-hoc approach.

For hyperparameter tuning, the prototype applies DeepHyper for its intended purpose. It tunes the following hyperparameters for the random forest model: number of trees, split criterion, max tree depth, and minimum number of samples to split a node.

3.4 FACT Evaluation

I now compare FACT's performance against the existing work and showcase its potential benefits at large-scale. By reducing the data collection time, especially on larger scale machines, FACT makes ML-based collective algorithm autotuner more feasible on exascale systems.

In the FACT prototype, data collection and machine learning techniques are separate. To evaluate it, I simulated the iterative process of collecting data, looking up the benchmark results from the previously collected exhaustive results.

3.4.1 Existing Work Comparison

For comparison, I recreated Figure 3.2 using the FACT prototype. Again, I used a 64 node subset of the Argonne Bebop cluster, each containing an Intel Xeon-E5-2694v4 with 36 cores (I used up to 32 cores) and 128GB of DDR4 memory. The result is shown in Figure 3.6.

Given the exact same exhaustive dataset, the FACT-based approach greatly improves performance with smaller training sets. For training sets between 50% and 1% of the feature space, Average Slowdown decreased by 33-68% compared the previous state of the art. Classification Accuracy (3-13%) and Significant Mistake Proportion (32-55%) also saw substantial improvements.

By improving autotuner performance with small training sets, FACT minimizes data collection

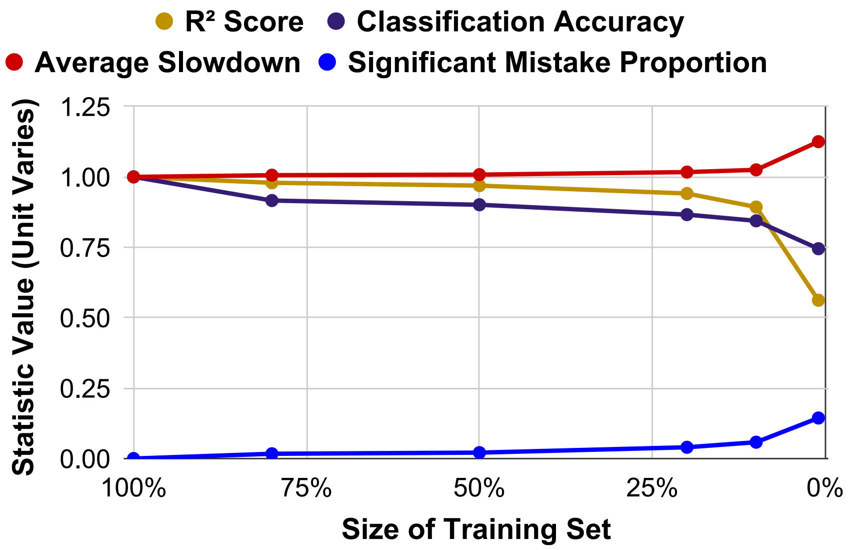


Figure 3.6:
FACT Prototype Performance.**FACT Prototype Performance.** A FACT-based approach greatly improves autotuner performance with smaller training sets.

time. I again apply "near-optimal" convergence criteria: Average Slowdown below 1.03 and Significant Mistake Proportion below .05 The FACT prototype reaches the criteria with a training set of roughly 10% of the feature space, while the existing work requires roughly 50% of the feature space. It takes 2.58 machine hours to collect the 50%, randomly selected training set. The 10%, active learning training set takes .375 machine hours to collect. The 6.88x reduction is more than expected based on the set size decrease (5x).

Active learning improves data collection time even more than the training set size suggests because it is naturally biased towards smaller feature values. Smaller feature sets have more variation and symbiotically take less time to collect. It follows that the surrogate model assigns greater uncertainty values in this range, therefore choosing these points instead of larger feature values. This effect is even greater with the smallest training sets. When both methods use 1% of the feature space, the active learner collects its data in 7.2x less time (.05 hours compared to .36 hours). Full

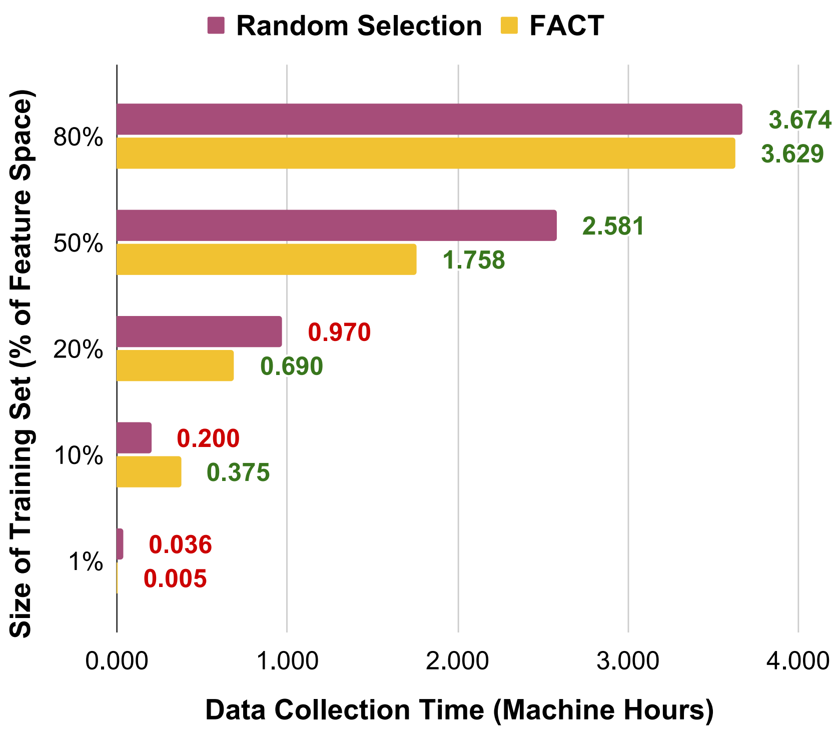


Figure 3.7: **FACT vs. State of the Art: 64 Node Data Collection Time for All Collectives.** Training sets with green labels result in ML models that meet the convergence criteria, while those with red labels do not. On average, FACT reduced data collection time by an average of 1.14x for training sets of the same size.

results for data collection time are shown in Figure 3.7.

3.4.2 Towards Exascale Systems

To understand how the FACT prototype scales with machine size, I also applied it to a larger scale test system. I used a 512-node subset of the 4,392 node Theta supercomputer at Argonne. Each node is comprised of an Intel Xeon Phi 7230 with 64 cores (I again used up to 32 cores) and 192 GB of DDR4 memory. At larger scale, it is impractical to collect exhaustive data for all collectives as in Section 3.2. I instead collected data for one of the most popular collectives: *MPI_Bcast* [60].

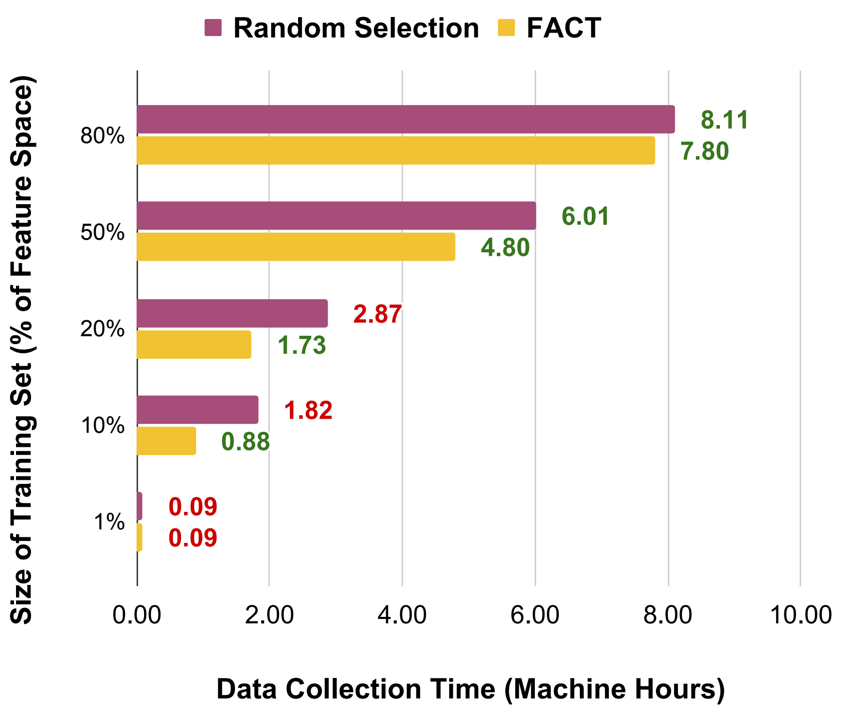


Figure 3.8: **512 Node Data Collection Time for *MPI_Bcast*.** Training sets with green labels result in ML models that meet the convergence criteria, while those with red labels do not. On average, FACT reduced data collection time by an average of 1.31x for training sets of the same size.

Using large scale *MPI_Bcast* data, I plot the data collection time for each training set size in Figure 3.8. Note that this figure only shows the data collection time for *MPI_Bcast*, while Figure 3.7 shows the data collection time for all standard, non-blocking collectives. I used *MPI_Bcast*'s increase in data collection time from 64 to 512 nodes times cumulative results up to 64 nodes on Theta to calculate the machine/core hour estimates stated in the introduction.

On the 512 node production scale machine, the benefits of FACT are amplified. Assuming the convergence criteria are met with the same amounts of data as the previous experiment, FACT requires 6.83x less training time (6.01 hours to .88 hours). Even on larger systems, FACT continues to greatly decrease data collection time.

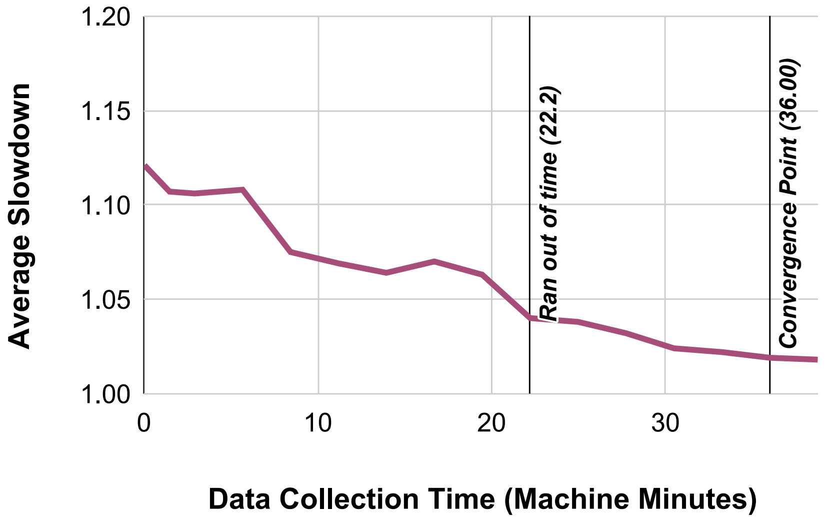


Figure 3.9: **Example Convergence Graph.** The user runs out of data collection time when they are only halfway to convergence.

3.4.3 Hyperparameter Tuning

To understand the benefits of hyperparameter tuning, again consider the scenario where the user does not have enough machine time to generate a converged model. For this experiment, the convergence criteria is when the model has an average slowdown less than 1.02.

I represent the convergence in Figure 3.9, which shows how Average Slowdown decreases as the active learner adds points to the training set. This data is generated from *MPI_Reduce_Scatter* from the 64 node testcase. I chose *MPI_Reduce_Scatter* because it is the slowest collective operations to converge, making it the most likely to require hyperparameter tuning. The first line indicates where the user ran out of data collection time (22.2 machine minutes), and the second line represents the model convergence point if they had been able to continue training (36 machine minutes).

I ran DeepHyper's hyperparameter tuning routine to automatically generate a better random

Table 3.4: **Hyperparameter Tuning Iterations**

Iteration	Average Slowdown
1	1.04
2	1.057
...	...
20	1.019

forest configuration for the training set that minimizes average slowdown. The results of the most important iterations are shown in Table 3.4. The tuning process produces an ML model that reduces the average slowdown by 2x, getting under the convergence target in less than two-thirds the data collection time.

Note that for this experiment, DeepHyper ran for around 15 minutes to find performant hyperparameter values. Attempting to introduce this process into active learning would increase the overall time by 15 minutes times the number of training points collected. In the scenario with limited collection time, attempting to run hyperparameter tuning during active learning would inhibit so much data collection that the model would never converge, regardless of the hyperparameter values.

3.5 Conclusions from FACT

The FACT approach can generate an ML-based autotuner with performance equal to the state of the art while reducing the training data collection time, making ML-based autotuners more feasible on exascale supercomputers. Overall, FACT's 6.88x expected reduction in data collection time reduces my original estimate for a large-scale autotuner on Theta from 6 days of machine time to 1 day. This result is highly promising, but it clear that there a ways to go.

3.6 Remaining Challenges

The FACT approach is a significant step toward practical ML autotuners for large-scale, production supercomputers. However, there are still many remaining challenges that were revealed during the evaluation process.

3.6.1 Training Point Selection

Despite its focus on the challenge, FACT's primary bottleneck remains training data collection. To understand why, re-consider FACT's use of DeepHyper [18]. During the iterative training process, FACT queries DeepHyper to select new training points to collect. DeepHyper selects the next point based on its surrogate model. Then, DeepHyper benchmarks the point and reports the result. FACT uses the data to train its own ML model, which is eventually used to make algorithm selections.

FACT implements this indirect approach because its own ML model, comprising random forest regressors, does not have a straightforward way to select points to benchmark. However, by relying on DeepHyper, FACT's training point selections are specific to the target environment but not to the FACT ML model, resulting in suboptimal selections. In addition, FACT must run an entire additional application (DeepHyper).

3.6.2 Non-Power-of-Two Points

FACT assumes that all feature values are power-of-two (P2). In practice, these values do not always meet this assumption. Incorporating non-power-of-two (non-P2) values greatly increases the search space for training, requiring many more training points and ballooning the overall collection time.

In simulation, non-P2 values can easily be avoided; but in production, any feature values may

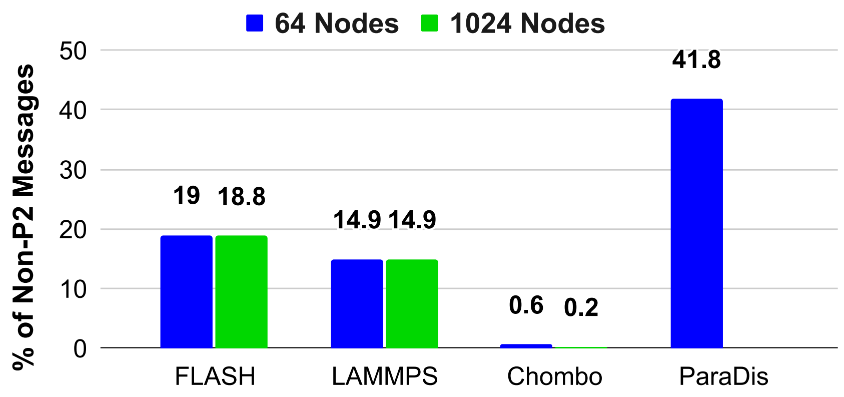


Figure 3.10: **Percentage of Non-Power-of-Two Message Sizes in HPC Applications.** 15.7% of collective calls use non-power-of-two message sizes, so we must consider their performance. 1024-node trace data is unavailable for ParaDis.

be used by applications. The "number of nodes" feature value is frequently non-P2 because non-P2 job sizes are common on Theta. The job scheduler may prioritize non-P2 node counts to maximize utilization. On the other hand, "processes per node" will rarely be non-P2 for Theta, which has 64 hardware threads per node. For systems with non-P2 hardware threads, training can use fractions of their thread count, ignoring P2 versus non-P2.

The last feature value, "message size," is less clear-cut. Messages use datatypes such as *char* and *int*, which have P2 bytes. However, an application may send a non-P2 count of a datatype, making the overall message size non-P2. To study the behavior of applications, I profiled traces from Lawrence Livermore National Laboratory [227]. Figure 3.10 shows that 15.7% of message sizes across four applications are non-P2. For each application, the percentage is nearly the same for both small- and large-scale jobs (1,024-node trace data is unavailable for *ParaDis*). Because a significant portion of application collective calls use non-P2 message sizes, ML autotuner must perform well for these values.

To understand its performance for non-P2 nodes and messages sizes, I reevaluated FACT using non-P2 test datasets for *MPI_Bcast*. I chose *MPI_Bcast* because two of its algorithms ($binomial$,

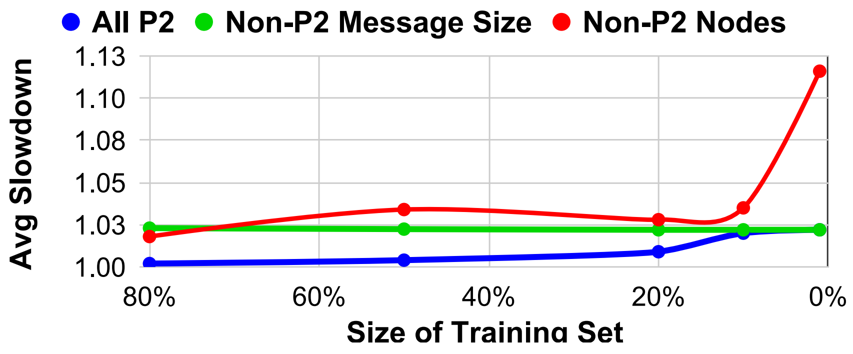


Figure 3.11: **FACT Non-Power-of-Two Performance (*MPI_Bcast*).** The FACT ML model (trained only with power-of-two values) fails to learn the performance trends in non-power-of-two message sizes.

$scatter\_recursive\_doubling\_allgather$) favor P2 feature values, while the third ($scatter\_ring\_allgather$) does not, making it the most interesting collective to study here.

I collected three new datasets on 64 nodes on Bebop like the FACT evaluation. One dataset uses all P2 values just like FACT's original evaluation. The other two datasets include only randomly selected non-P2 numbers of nodes and message sizes, respectively. I trained the FACT prototype (which uses only P2 points for training data) and tested it separately on all three new test sets.

The results are shown in Figure 3.11. The FACT methodology produces an ML model with significantly inferior performance for non-P2 test points. For the "All P2" test set, FACT with plentiful training data at 80% performs almost optimally, then slowly deteriorates to the right. "Non-P2 Nodes" has the correct shape, while its *average slowdown* is always higher than the "All P2" results. This trend appears because of the higher performance variability of MPI_Bcast algorithms for non-P2 node counts. The ML model is learning the performance patterns at the same rate as the "All P2" set; it just gets punished more severely for incorrect selections.

"Non-P2 Message Size" shows a significant *average slowdown* across the entire graph. The model does not optimize performance for this test set, even with large ($>80\%$) amounts of training

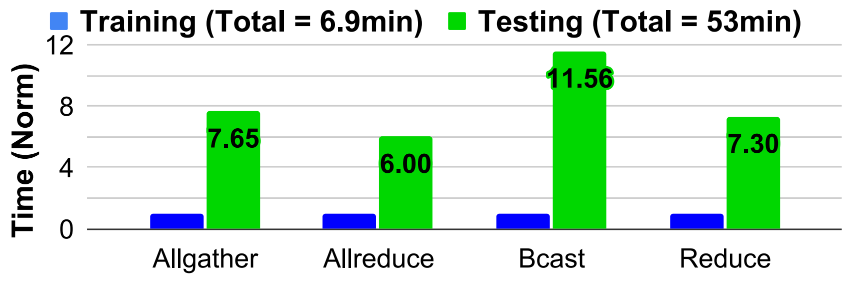


Figure 3.12: **Training Set and Test Set Data Collection Time for Simulated Experiments.** Data collection for the test set drastically outpaces ACCLAiM's training data collection time.

data. The reason is that the model fails to learn the trends in the data. Therefore, it is best to focus on non-P2 message sizes, where the model shows the least ability to learn the performance trends.

3.6.3 Model Testing

In each iteration of active learning, the ML model is measured to check whether the overall performance has "converged." For convergence testing, FACT uses metrics such as Average Slowdown to measure the model's prediction quality. To calculate Average Slowdown, FACT needs additional points to test, aptly called the "test set." Previous simulated experiments ignore the test data collection time. In production, test data points are benchmarked like training points, but the results must be kept separate. Test points consume critical data collection time but cannot be used to train or improve the model.

FACT reduced the required *training* set size to $\sim$1% of the feature space. However, the *testing* set size needs to cover 20% of the feature space for machine learning methods to work correctly. Consequently, the time to collect test data dwarfs the time to collect training data, drastically inflating the overall data collection time. In Figure 3.12, I plot the data collection time of a 20% test set compared with the training data collection time. I normalize the values to the training data collection time for each collective. Each collective requires 6–11x more time to collect test data than training data. FACT's 1-day training time estimate completely ignores test data collection, so this component has the potential to consume a week or more of machine time in production, which is far too large.

3.6.4 Data Collection

FACT, like the previous works, collects all data points sequentially. This process is important because HPC networks may route packets indirectly to increase the effective bandwidth between nodes. This policy can create unexpected network congestion between communications, which must be avoided in order to accurately measure collective performance. While safe, a sequential collection strategy is very inefficient, particularly for larger machines.

3.7 ACCLAiM: A Production-Capable Autotuner

To address the significant challenges from the FACT prototype, I developed a new autotuner named ACCLAiM (**A**dvancing **C**ollective **C**ommunication (**L**) **A**utotuning using **M**achine **L**earning). Figure 3.13 shows how ACCLAiM's approach fundamentally differs from all previous work on the topic. Instead of just using simulation results to theoretically show improvements, ACCLAiM seeks to build upon FACT to achieve a production-capable ML autotuner for collective algorithm selection.

Broadly, ACCLAiM further reduces training time by minimizing the number of training points collected and maximizing hardware utilization. ACCLAiM is the first ML autotuner prototype that is practical to run on large-scale production supercomputers.

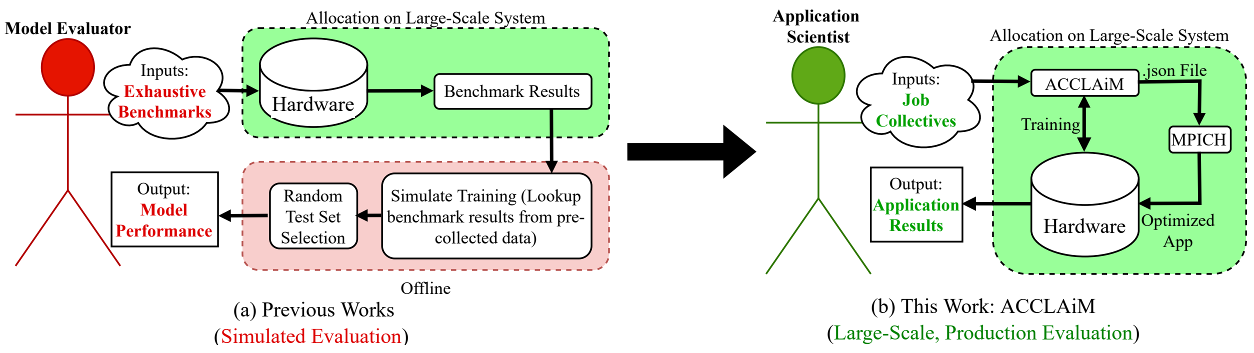

Model Evaluator
Inputs: Exhaustive Benchmarks
Allocation on Large-Scale System
Hardware
Benchmark Results
Output: Model Performance
Random Test Set Selection
Simulate Training (Lookup benchmark results from pre-collected data)
Offline
(a) Previous Works
(Simulated Evaluation)
Application Scientist
Inputs: Job Collectives
Allocation on Large-Scale System
ACCLAiM
.json File
Training
MPICH
Output: Application Results
Hardware
Optimized App
(b) This Work: ACCLAiM
(Large-Scale, Production Evaluation)

Figure 3.13: **ACCLAiM User Model Comparison.** Previous works use offline simulation to show the promise of ML collective autotuners. ACCLAiM aims to make these autotuners practical for applications on large-scale production supercomputers.

3.8 ACCLAiM Improvements

This section describes ACCLAiM's improvements that address each of remaining challenges from FACT.

3.8.1 Training Point Selection Improvements

As described in Section 3.6.1, the FACT approach generates training point selections using a second, separate ML model. To simplify the process, ACCLAiM uses jackknife variance calculations to derive training points from a single ML model.

The jackknife technique calculates summary values through resampling [71]. Consider an example where the objective is to find the variance of n values. The values are $p = (p_1, p_2, ..., p_n)$, where p_i is the ith value. Let x_p equal the mean of p. A jackknife works by creating *jackknife samples*. For a jackknife with n values, there are n jackknife samples. The i^{th} jackknife sample equals the mean of p with p_i removed. By removing single values, the jackknife generates n unique samples. Let x be the means of the jackknife samples, where $x = (x_1, x_2, ..., x_n)$ and x_i is the mean of p with p_i removed. Then the variance (σ^2) is calculated as

$$\sigma^2 = \frac{\sum_{i=1}^{n} (x_p - x_i)^2}{n-1}$$

ACCLAiM uses the jackknife technique on the random forest model to calculate the variance of potential training points. Applying the jackknife technique to a random forest regressor was first proposed by Wager et al. [226]. Random forest is an *ensemble* machine learning model, meaning its predictions are the average of an ensemble of individual ML models. Random forests consist of decision trees. ACCLAiM applies the jackknife technique as follows:

1. Let $n =$ the number of decision trees in the random forest.

2. Let $p =$ the set of predictions from the decision trees. p_i is the prediction from the i^{th} decision tree in the forest.

3. Let $x_p =$ the mean of p.

4. Calculate x_i by removing each p_i one a time.

5. Input x_p and x_i into the jackknife variance equation.

ACCLAiM repeats this process for every possible training point. Then, it selects the point with highest variance as the next training point. By selecting points with high variance, ACCLAiM provides its ML model with data that fills gaps in its understanding, minimizing the number of points required to train a well-formed model. ACCLAiM only considers P2 feature values only when using jackknife to limit the number of calculations.

3.8.2 Non-Power-of-Two Point Improvements

To address non-P2 points from Section 3.6.2, ACCLAiM incorporates an extra step in the training point selection. Every fifth training point, instead of choosing the exact point with the highest variance, it instead selects a point with a random non-P2 message size where the selected message size is the closest P2 value. For example, if the highest variance point has a message size equal to 8, ACCLAiM would select a new message size between 6 and 12 that is not 8. The non-P2 frequency was experimentally determined that it best balances P2 and non-P2 performance (Section 3.10.2). By incorporating non-P2 variants, the ML model learns to accurately predict non-P2 points without additional data collection time.

3.8.3 Model Testing Improvements

As described in Section 3.6.3, previous approaches like FACT calculate metrics using a costly set of test points. ACCLAiM needs to estimate model performance *without a test set*, which means it cannot calculate any traditional performance metrics. It instead needs a quantity that is measurable during training with minimal overhead and correlates with model performance. To achieve this, ACCLAiM reapplies the jackknife technique, summing the variance of every point.

To test whether this variance measure correlates with model performance, I performed a simulated experiment tracking both FACT's standard convergence metric (Average Slowdown) and variance. Figure 3.14 shows that variance correlates with Average Slowdown. Both metrics trend downward over the same time interval. At around 400 seconds, the spike in Average Slowdown is matched with a spike in variance. This event indicates that variance is capable of mimicking fine-grain changes in performance. With these observations in hand, I designed ACCLAiM with cumulative variance as a proxy for Average Slowdown and therefore its convergence criterion. I provide further evaluation of this technique in Section 3.10.3.

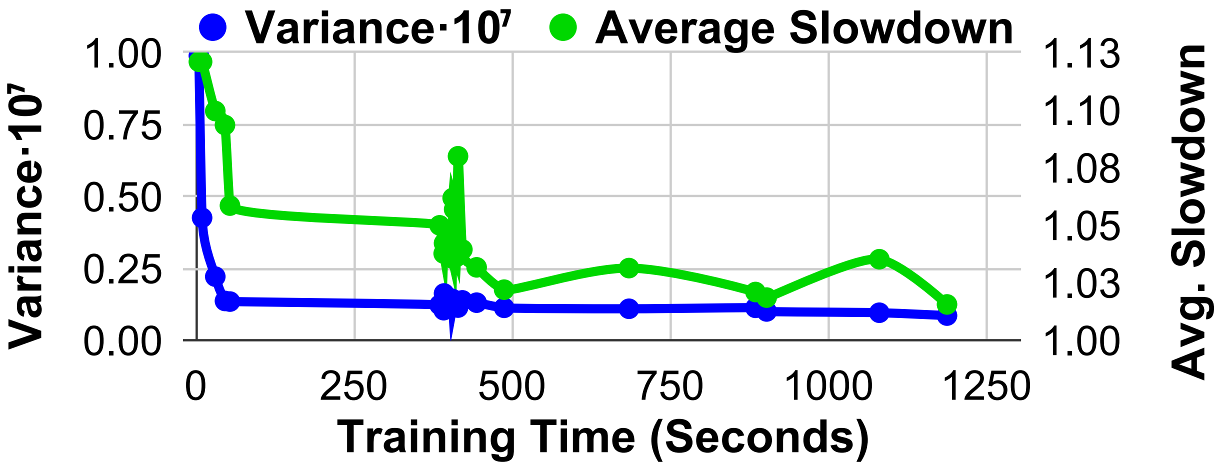


Figure 3.14: **Variance and Average Slowdown as a Function of Training Time.** Average Slowdown converges as variance converges.

3.8.4 Data Collection Improvements

As described in Section 3.6.4, previous approaches collect training data points *sequentially* to avoid potential network congestion. To enable *parallel* data collection, ACCLAiM leverages the network topology.

Figure 3.15 shows a simplified version of the Dragonfly topology, which is a popular topology for modern supercomputers, including Theta and Frontier. Nodes are numbered sequentially within a rack and across racks. This Dragonfly example has three layers[1]. The first layer connects nodes within a rack. Layer two pairs every two racks. The third layer connects the rack pairs. Network congestion will occur if benchmarks share a single instance of a layer. For example, two benchmarks cannot run on nodes in a single rack. Similarly, if a run is using nodes on both rack 0 and rack 1, another benchmark cannot use any remaining nodes on those racks.

ACCLAiM uses a greedy algorithm to run many benchmarks in parallel. Instead of selecting a single training point, ACCLAiM generates a list of potential training points sorted by variance.

[1]Theta uses a 3-layer Dragonfly, while Frontier's Dragonfly has only 2, skipping the "rack-pair" layer.

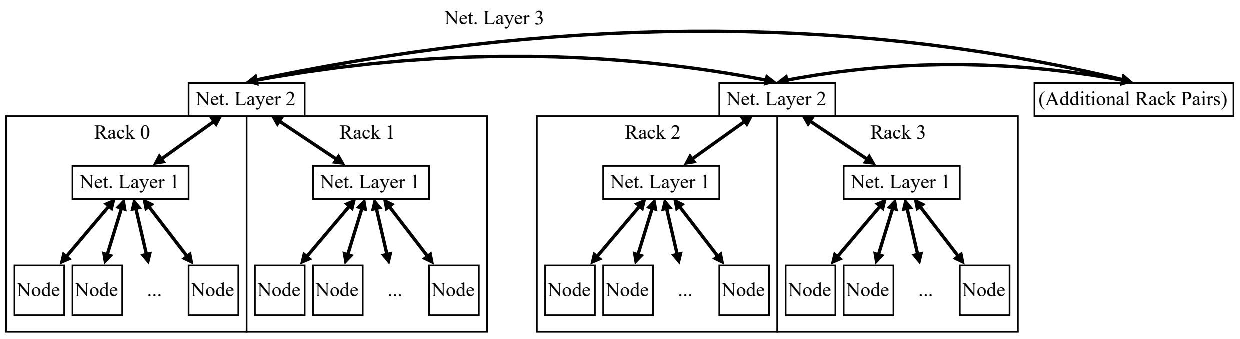


Figure 3.15: **Simplified Dragonfly Topology.** This topology is a simplified version of the design found on many supercomputers, such as Argonnne's Theta and Oak Ridge's Frontier. Jobs must be scheduled to minimize network congestion in the first two layers.

Then, it generates a "schedule," assigning benchmarks to run on disparate nodes in parallel. The algorithm works as follows:

1. Select the highest-variance uncollected point p, which requires n nodes.
2. Attempt to schedule p on the next n "unused" sequential nodes.
3. If n can fit in the eligible nodes, schedule p on the next n "unused" nodes, mark those nodes and any remaining nodes in the same racks as "used," and repeat.
4. If n cannot fit, exit and run all scheduled benchmarks in parallel.

This algorithm avoids network congestion by disallowing different benchmarks to run in the same rack and scheduling on sequential nodes. By disallowing shared racks, this algorithm prevents congestion on the first network layer. Scheduling on sequential nodes prevents congestion on the second network layer because multi-rack benchmark runs cannot simultaneously schedule across the same racks. If a run needs nodes on another rack, it must first fill its original rack, preventing a second run from also being scheduled across those racks.

If two runs schedule across multiple rack pairs and the first run ends within the same pair as the second run begins, the third network layer may see slight congestion. However, because the third network layer is implemented using direct, high-bandwidth connections, incidental congestion is relatively low. Also, when designing for an active production environment, congestion in the third layer is already expected from other applications. Third-layer congestion is mitigated by measuring each collected point multiple times.

3.9 ACCLAiM Implementation

Here I describe how a user interacts with ACCLAiM and its implementation details. To fully account for dynamic non-programmatic variables, ACCLAiM uses a unique "allocation-time" training approach, meaning that it trains once a job is scheduled onto a partition but prior to application execution. Because ACCLAiM retrains for every job, training consumes execution time during a job's allocation. Figure 3.13(b) shows how ACCLAiM is designed to be used in a production environment.

3.9.1 User Input

The user submits a job to the HPC system through ACCLAiM. The only additional information required is a list of collectives predominantly used in the application. This collective list should be common knowledge for highly optimized applications. If not, users can provide a conservative list including every collective that *might* require autotuning or use a tool such as Intel's APS [114].

3.9.2 Training

When a job is scheduled in the production environment, instead of immediately running the application, ACCLAiM trains its ML model for the specified collectives. ACCLAiM adopts the basic training framework from FACT. To collect training points, ACCLAiM again uses the Ohio State University microbenchmark suite [166]. For its random forest model, it again uses the *RandomForestRegressor* from the *scikit-learn* Python package [172]. ACCLAiM uses a single random forest model per collective and enumerates "algorithm" as an additional feature.

3.9.3 Configuration File Generation

Once the models have converged, ACCLAiM generates an edited version of the default algorithm selection `.json` file with its models' selections. The `.json` file is a list of logic rules that indicate which algorithm to select. An example rule is `if(`$message\_size \leq 32$`) {algorithm = binomial}`. The rule set must be "complete," meaning that every possible input must resolve to a selection. The rules also must be pruned such that no two consecutive rules resolve to the same prediction. This step minimizes the selection delay during execution.

To create a list of rules, ACCLAiM collects its ML model's algorithm selections for every P2 point. Naively, it could iterate through the selections and create a rule every time the selection value changes. However, this method abandons the model's non-P2 point selections. Instead, it iterates through the ML model's algorithm selections and detects when the selection changes. Below, I describe the logic applied during this process. Point A is the last point with the old algorithm selection. Point C is the first point with the new algorithm selection. Point B is the non-P2 point halfway between A and C. I refer to the selected algorithm at each as "ALG-(Point)" (e.g., ALG-A is the selection at A).

ACCLAiM generates rules based on the change in selected algorithm, as shown in Figure 3.16. It creates three rules: all values below A (inclusive) use ALG-A, all values between A and C (exclusive) use ALG-B, and all values above C (inclusive) use ALG-C. These rules enable unique selections for non-P2 points between A and C. Then, to optimize the rule set and minimize

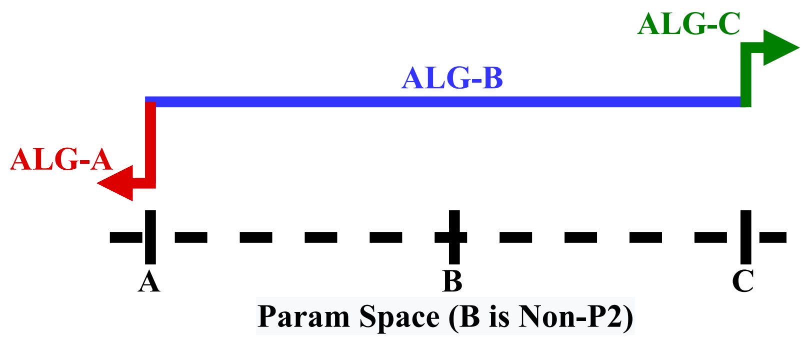


Figure 3.16: **ACCLAiM's Rule Creation Logic.** ACCLAiM generates new rules in the configuration file to communicate the ML model's selections to the MPICH library. Each color represents a different rule.

selection delay, ACCLAiM prunes these rules. If ALG-A = ALG-B, it merges the first two rules into a single rule for all values less than C. If ALG-B = ALG-C, it merges the last two rules into a single rule for all values greater than A.

3.9.4 Application Execution

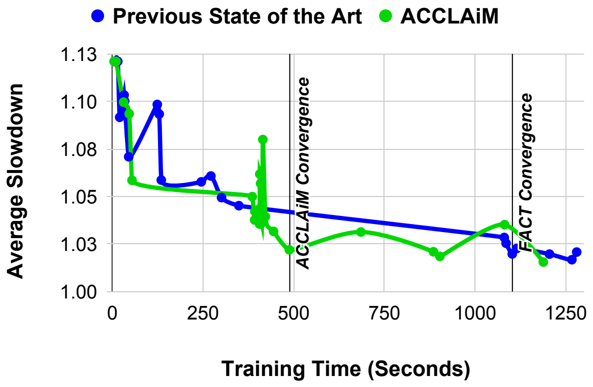
ACCLAiM directs MPICH to the new `.json` file using an environment variable. Then, the application proceeds and outputs its results, completing the job.

No previous work combined ML autotuning with job execution. ACCLAiM, on the other hand, completes the entire process while remaining transparent to the user.

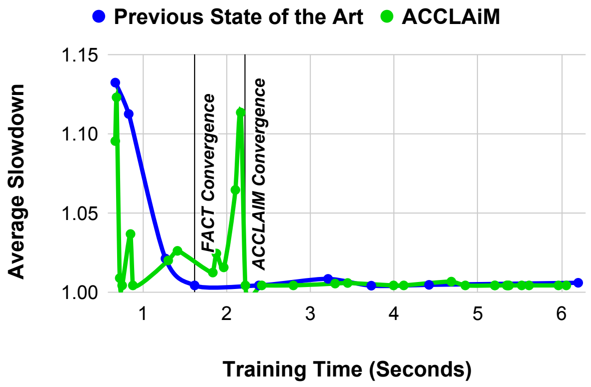
3.10 ACCLAiM Evaluation

To understand the impact of ACCLAiM's improvements, I first compared with FACT. For all comparative results, I used simulated experiments like those used to evaluate FACT originally. Then, I studied ACCLAiM's capabilities on *Theta*.

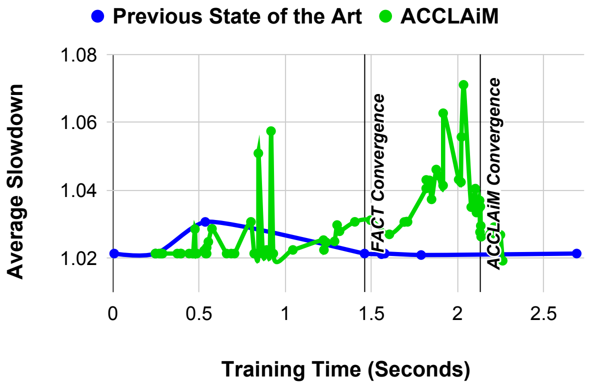
In both environments, I studied the performance of the four most popular collectives from



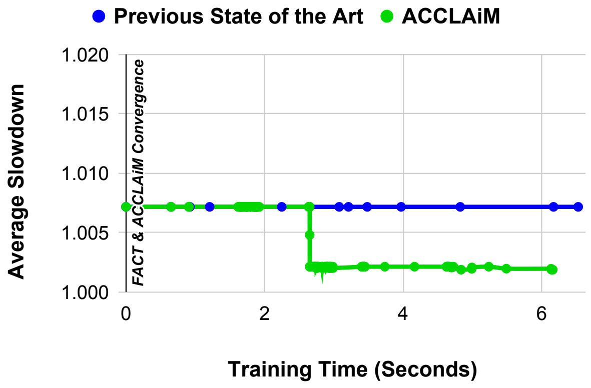
(a) *MPI_Allgather*



(b) *MPI_Allreduce*



(c) *MPI_Bcast*



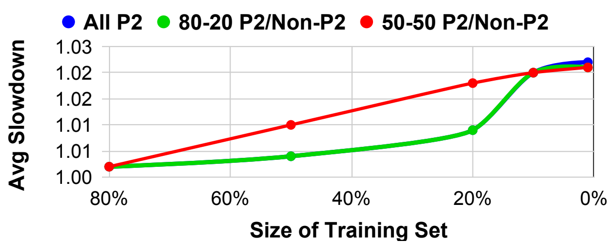
(d) *MPI_Reduce*

Figure 3.17: **ACCLAiM Training Point Selection Comparison.** Cumulatively, ACCLAiM converges in 2.25x less time than FACT.

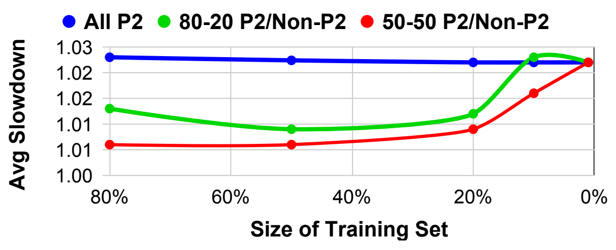
Chunduri et al. [60]: allgather, allreduce, bcast, and reduce. Experiments that do not explicitly separate the four collectives show aggregate values.

3.10.1 Training Point Selection: Up to 2.3x Faster

I began with the reduction in training point collection time created by our jackknife-based training point selection methodology. I compared ACCLAiM's selection approach vs. FACT's previous state of the art. Figure 3.17 shows Average Slowdown vs. training time graphs for each collective.

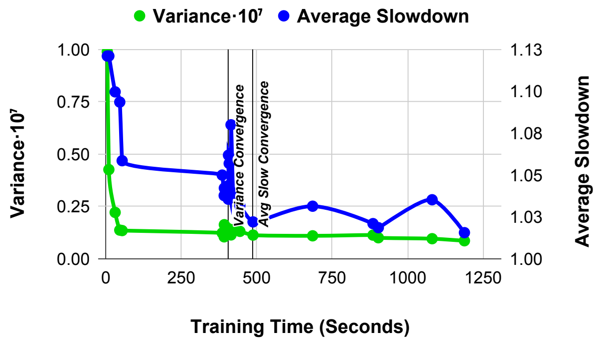


(a) All P2 Test Dataset

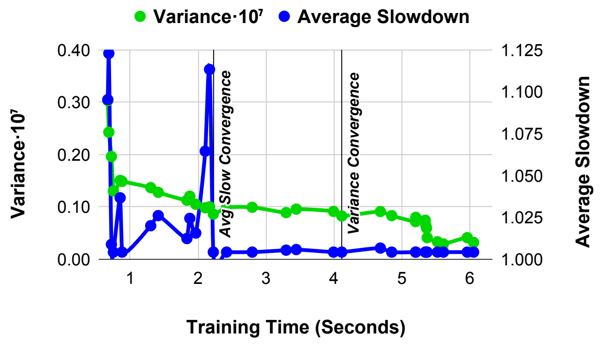


(b) Non-P2 Message Size Test Dataset

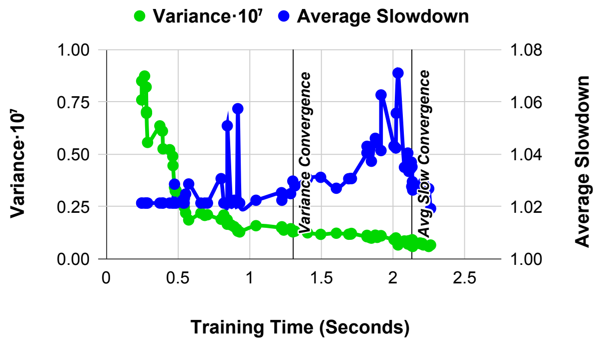
Figure 3.18: **ACCLAiM P2 Training Point Incorporation for *MPI_Bcast*.** ACCLAiM's 80-20 split maintains P2 performance while dramatically improving non-P2 performance.



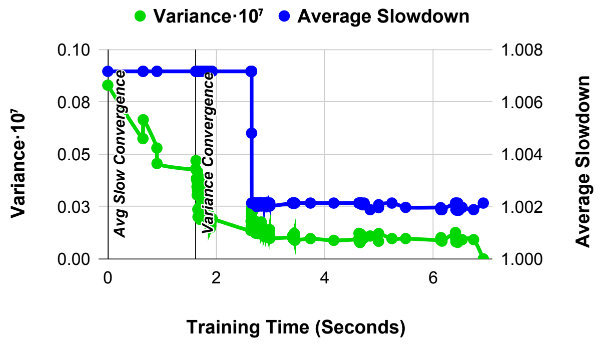
(a) *MPI_Allgather*



(b) *MPI_Allreduce*



(c) *MPI_Bcast*



(d) *MPI_Reduce*

Figure 3.19: **ACCLAiM Convergence Speedup & Accuracy** Using cumulative variance as a proxy for *average slowdown*, ACCLAiM detects convergence 1.19x faster while avoiding a potential 6–11x slowdown caused by test set data collection.

The x-axis represents training data collection time by summing the benchmark execution times. Both training methodologies want to decrease their Average Slowdown as quickly as possible. The graoh is marked when each training methodology reaches the convergence criterion, which is the standard Average Slowdown< 1.03.

ACCLAiM converges for all four collectives in up to 2.3x less time than FACT. *MPI_Allgather* is the most expensive collective to tune and also ACCLAiM's biggest win at 2.3x. FACT performs slightly better for *MPI_Allreduce* and *MPI_Bcast*, by 1.37x and 1.46x, respectively. Both methodologies converge almost instantly for *MPI_Reduce*. Overall, ACCLAiM's model-specific selections create a significant reduction in training data collection time. The reported speedups do not account for the additional acceleration from simplifying the point selection process by eliminating DeepHyper. In practice, I expect even greater speedups.

For both Figure 3.17 and Figure 3.19 (which I explore in Section 3.10.3), a few visual oddities are important to understand. First, all lines do not begin 0 on the x-axis. To evaluate model performance, each autotuner requires at least one point for training. If the first training point takes a significant amount of time to collect, a "gap" may appear on the left side of the graph. This behavior is normal and expected; it just indicates that the first training point was expensive to collect. Additionally, large flat sections appear in some graphs. Note that these graphs are discrete, since there cannot be partially collected data points. The lines between points are included to indicate trends. If a point takes a long time to collect and does not greatly affect the model's understanding, a flat section appears on the graph. However, these sections do not represent potential convergence, just expensive point collection.

3.10.2 Non-Power-of-Two Points: Now Modeled

Next I evaluated the effects of incorporating non-P2 points in our sampling process. I re-evaluated the “All P2” and “Non-P2 Message Size” test datasets from Figure 3.11 incorporating various amounts of non-P2 training data. Again, the figures constrict the amount of training data given to the model from right to left on the graph. The amount of training data is a percentage of the possible training points.

The goal is to maintain an Average Slowdown as close to 1.0 as possible with minimal training data. I studied training sets with all P2 data, a 50-50 selection split, and ACCLAiM’s 80-20 data selection split. Each training point includes the same total number of training points. Selecting 50% of the non-P2 points means that I removed half of the P2 points.

From Figure 3.18(b), a 50-50 split maximizes non-P2 performance. However, it sacrifices P2 performance, as shown in Figure 3.18(a). ACCLAiM’s 80-20 split preserves P2 performance while significantly improving non-P2 performance. By selecting every fifth point as non-P2, ACCLAiM maintains the “Goldilocks” performance balance.

3.10.3 Model Testing: Avoid 6–11x Test Set Collection Slowdown

Here I compared cumulative variance as a convergence criterion vs. Average Slowdown. The variance convergence criterion is that four consecutive training iterations must have a difference in variance less than 10^{-9}. I selected this criterion from prior tuning experience while developing ACCLAiM. It works well for both simulation and the production system (discussed in the next section).

In Figure 3.19 I graph the cumulative variance values on the left vertical axis and Average Slowdown from Figure 3.17 on the right vertical axis. The graph is marked when both metrics meet their respective convergence criterion. An ideal variance convergence would occur at the same time as the Average Slowdown convergence because the goal is to precisely model Average Slowdown. In these experiments, I accepted variance convergences close to the Average Slowdown convergence point if both points produce ML models of nearly equal performance.

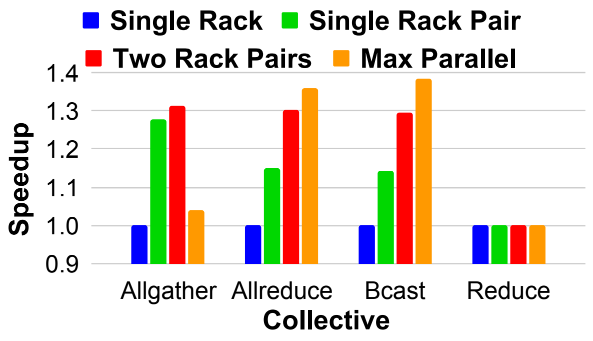
For all collectives, Figure 3.19 shows that the variance convergence criterion consistently produces trained models with low Average Slowdown. *MPI_Allreduce* and *MPI_Reduce* have variance convergence points after the original *average slowdown* points, adding an extra 1.007x to the cumulative training time for all collectives. However, the overall training time is actually reduced by 1.19x because of the other two collectives.

For *MPI_Allgather* and *MPI_Bcast*, the variance convergence point is before the average slowdown point. In both cases, the model-tested variance has an *average slowdown* of 1.04. While this value is above the *average slowdown* convergence criterion, this slight uncertainty is well worth the trade-off of eliminating the testing set. Overall, ACCLAiM avoids the potential 6–11x slowdown from Figure 3.12 without sacrificing convergence accuracy.

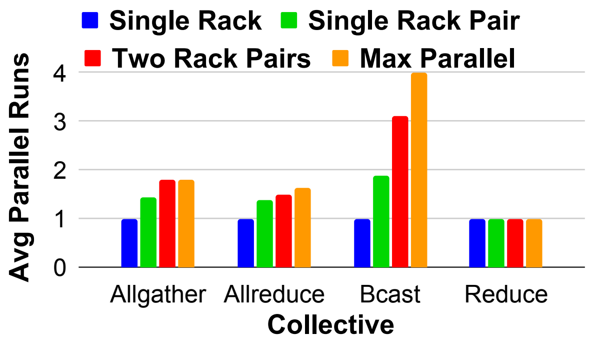
3.10.4 Data Collection: 1.4x Faster Using Parallelism

To evaluate our parallel data collection strategy, I simulated scheduling the simulation dataset across four different theoretical topologies: all 64 nodes on a single rack, 32 nodes each on two racks in a pair, 16 nodes each on four racks in two pairs, and single nodes on different racks all from separate pairs (1-0-1-0...). The "separate pairs" topology represents the maximum parallelism potential, so I henceforth refer to it as "Max Parallel." These topologies represent a range of situations from no/minimal parallelism (Single Rack, Single Rack Pair) to "Max Parallel." The results are shown in Figure 3.20.

Data collection is accelerated by up to 1.4x by running 1–4 benchmarks simultaneously. Even for topologies with modest parallelism opportunities, there are significant speedups ($\sim$1.3x).



(a) Speedup



(b) Average number of benchmarks ran in parallel

Figure 3.20: **ACCLAiM's Parallel Data Collection.** ACCLAiM achieves a 1–1.4x speedup by running 1–4 benchmarks in parallel.

An interesting data point occurs in Figure 3.20(a) for the "Max Parallel" topology for $MPI\_Allgather$. Here, the parallelization speedup decreases compared with the other topologies. During scheduling, "Max Parallel" enables a low-latency benchmark to run in parallel with a high-latency, succeeding benchmark. The high-latency benchmark is now unable to be parallelized with the next benchmark, which also has a high latency. Running the two high-latency benchmarks sequentially more than eliminates the original advantage. The other topologies disallow the first parallelism opportunity, which coincidentally enables the second. This situation highlights the sub-optimal nature of greedy algorithms.

3.10.5 Production Practicality: Benefits Typical Jobs on Theta

After evaluating ACCLAiM's contributions individually, I applied at large-scale on Theta. For these experiments, ACCLAiM selects algorithms up to 128 nodes, 16 processes per node, and 1 MB message size.

The training time is shown in Figure 3.21. In the larger-scale, production environment, AC-

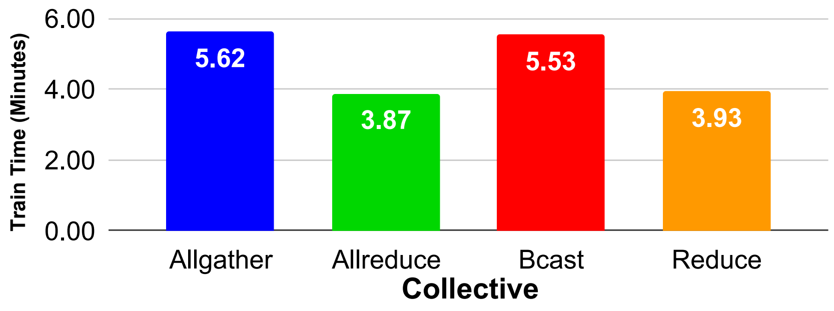


Figure 3.21: **ACCLAiM Training Time on Theta (128 Nodes).**

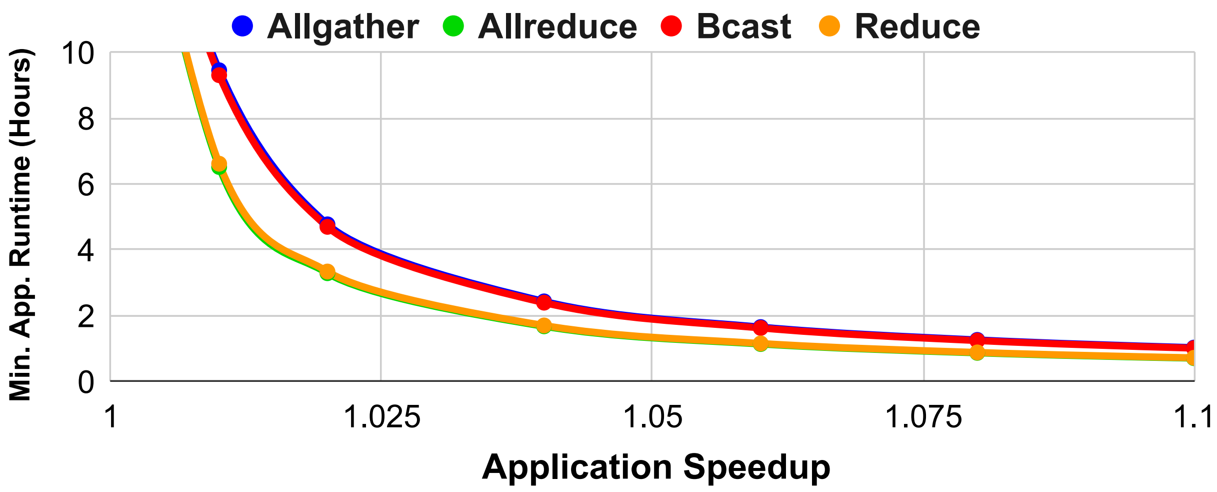


Figure 3.22: **Minimum Application Runtime for Overall Acceleration from ACCLAiM.** In many cases, applications must run for only a few hours to recover ACCLAiM's training time.

CLAiM converges in a matter of minutes, compared with the many hours estimated for FACT. I cannot make a direct performance comparison because FACT is unable to function in this "real" (not simulated) setting.

Finally, I consider ACCLAiM's impact on application performance. Collective performance is critical to the performance of many HPC applications [225, 60]. Previous works have provided examples of how autotuning collective algorithm selections can improve application performance [176, 142].

To assess the broader applicability for ACCLAiM, Figure 3.22 shows the minimum application runtimes required to gain an overall speedup when using ACCLAiM. I present a range of application speedups, which vary depending on the quality of the default selections and the percentage of time the application spends on collective calls. Applications that run for more than a few hours and are slowed even slightly by the default algorithm selections are great candidates for ACCLAiM. For example, an application with a 1.01x speedup from improved algorithm selections only has to run for 6.4–9.5 hours, which is well within the common duration for jobs on Theta. By showing that ACCLAiM can accelerate applications with moderate runtimes, I demonstrate ACCLAiM's practicality for large-scale production systems.

3.11 Conclusion

Overall, FACT and ACCLAiM make collective algorithm selection autotuning feasible on large-scale supercomputers. Together, they show how it is possible to *transparently* accelerate distributed memory communication. FACT was published at the ExaMPI'21 workshop [234], and ACCLAiM was published at the CLUSTER'22 conference [233]. Next, I describe my efforts to create new algorithms that increase the potency of autotuning.

CHAPTER 4
GENERALIZED COLLECTIVE ALGORITHMS

FACT and ACCLAiM can provide a significant, transparent improvement for MPI collectives on modern hardware. However, they are ultimately restricted by the algorithms they have at their disposal. The open-source implementations of the MPI standard including MPICH [154] (the focus of my work) and Open MPI [91] implement a limited set of algorithms per collective.

For example, consider "MPI_Allreduce". "MPI_Allreduce" is arguably the most important collective operation; it makes up the largest percentage of workloads [60, 211] and is the linchpin for ubiquitous applications like distributed data-parallel AI. For all messages and P2 process counts, MPICH uses a single algorithm (recursive doubling) to implement "MPI_Allreduce". In this case, it is impossible for algorithm selection to improve performance because there is not another option.

To increase collective algorithm tuning potential, I created an extensive set of "generalized" (i.e., variable-radix) algorithms. Generalized algorithms support additional parameters that change the algorithm's behavior. For example, the "binary tree" algorithm (like in Figure 3.1b.)) can be generalized to "k-nary tree". Beyond algorithm selection, "k" parameter value can be tuned to further improve performance.

Previous works have developed generalized collective algorithms, but the limited scope of their initial designs limit their usefulness today. For example, they are limited to a specific network topology [195], small message size allreduce [192], or intranode broadcast [193]. As a result, generalized (i.e., variable-radix) algorithms appear very sparsely in current implementations of MPI, sacrificing performance on modern HPC systems.

My generalized algorithms, on the other hand, are designed to broadly leverage the features

Table 4.1: **10 New Generalized Collective Algorithms.**

Base Kernel	Generalized Kernel	Collective Operations
Binomial	K-nomial	*MPI_Reduce*, *MPI_Bcast*, *MPI_Allgather*, *MPI_Allreduce*
Recursive Doubling	Recursive Multiplying	*MPI_Bcast*, *MPI_Allgather*, *MPI_Allreduce*
Ring	K-Ring	*MPI_Bcast*, *MPI_Allgather*, *MPI_Allreduce*

of the latest generation of supercomputers. I recognized that "exascale era" systems, including Frontier, Argonne's upcoming Aurora supercomputer, and Lawrence Livermore's upcoming El Capitan supercomputer (the three announced exascale supercomputer in the United States), among others, share multiple critical network features. My algorithms match these features, maximizing performance.

In this chapter, I describe the commonplace network features of exascale era supercomputers, then describe my algorithms. To create my algorithms, I designed generalizations of three major communication patterns (i.e., kernels): binomial (§4.2), recursive doubling (§4.3), and ring (§4.4) inspired by my identified generalizations [193, 192, 101]. Applying the kernels to individual collective operations, I implemented 10 algorithms for the four most common collective operations (see Table 4.1). For each generalized algorithm, I created analytic models to develop intuition regarding the optimal radix values. Then, I integrated the new algorithms into MPICH and experimented on Frontier and Polaris [180] (a pre-exascale system at Argonne). Compared to the non-generalized versions, my generalized algorithms improve performance by 1–2.5x. Combined with optimized algorithm selection, the advantage balloons to up to 4.6x.

4.1 Commonalities of Exascale Networks

Exascale supercomputers combine multi-CPU/GPU nodes with high-bandwidth, high-performance networks. Frontier and the next expected exascale supercomputers (e.g., Aurora, El Capitan) share multiple features that impact collective performance. Here I list the relevant features and pinpoint algorithm generalization techniques to leverage them.

4.1.1 Network Topology

Similar to the Theta system from ACCLAiM's evaluation, exascale networks use the dragonfly topology [128] Frontier, Aurora, and El Capitan specifically leverage a two-layer topology design that was introduced relatively recently. Its fully connected groups and hierarchical design minimizes the latency between nodes at large scale. One advantage of dragonfly is that it uses high radix virtual switches and global adaptive routing to ensure that there exists a shortest path between any two nodes. Therefore, topology-aware generalized algorithms for traditional HPC interconnects (e.g., torus, hypercube) that use non-minimal routing [195] will not be effective. Instead, I designed minimal communication algorithms that leverage other hardware features (discussed below).

4.1.2 Multi-Port Nodes & Message Buffering

In exascale supercomputers, high network bandwidth is necessary to support the computational power of multi-CPU/GPU nodes. To meet this need, exascale networks assign subsets of GPUs to dedicated network links. For example, each node on Frontier includes four 200 Gb/s links (one per 2 GPUs). Furthermore, non-blocking send/receive primitives in software enable the buffering of multiple messages simultaneously beyond the number of physical ports. Message buffering is critical to overlap the message submission latency of smaller messages.

Collective algorithms must employ multi-port functionality and message buffering to fully utilize exascale systems. However, in popular communication patterns such as binomial tree and recursive doubling, each process only communicates with one other process at a time, thus they only buffer a single message to send through a single network port. To solve this challenge, I developed two generalized algorithms, k-nomial and recursive multiplying.

Previous works have presented generalizations for limited circumstances, namely intranode broadcast [193] and small message (<4kB) allreduce [192] I developed new generalized algorithms inspired by these strategies to exploit multi-port networks and message buffering. I used the variable radix of these algorithms to elegantly capture the interplay between hardware and software to optimize for exascale systems.

4.1.3 Intranode Links

Applications on multi-GPU node-sytems may assign a separate MPI process to each GPU. For example, applications on Frontier commonly use 8 MPI processes per node. These processes communicate via dedicated higher-bandwidth hardware links (e.g., NVLink, InfinityFabric) that provide higher performance compared to the internode network. However, the predominant communication kernel for large (i.e., bandwidth-bound) message sizes, the ring kernel, does not differentiate between link types, slowing the entire algorithm to match the slower connections.

To address this issue, I designed a generalized "k-ring" kernel. Previous work on a new reduction algorithm showed how a hierarchical strategy can better saturate a heterogeneous network structure [101]. Combining this idea with the ring kernel, "k-ring" better utilizes the high-bandwidth intranode links found on exascale systems.

In summary, by analysing the major factors for algorithm performance, I identified three promising generalizations (k-nomial, recursive multiplying, and k-ring), which all expose their radix as a tunable parameter. I proceed to explain each generalized kernel and use analytical models to predict how changing the parameter values will affect performance.

4.2 Binomial Tree and K-nomial Tree Algorithms

In each algorithm section (§4.2, §4.3, and §4.4), I begin by describing the original communication kernel, then describe my generalized version. The first communication kernel is the binomial tree algorithm. Binomial is typically the optimal choice for small message sizes (<16KB), where the limiting factor for performance is point-to-point latency. Binomial tree minimizes the effect of latency by overlapping communications.

4.2.1 The Binomial Tree Algorithm

A binomial tree is a recursive tree structure where the sub-tree at each non-leaf node is the same as the sub-tree at that node's first child. Figure 4.1 shows an example binomial tree communication for *MPI_Gather* on 8 processes. The identical sub-trees are highlighted using red and blue, and the first "round" of overlapped communications is highlighted in green.

4.2.2 Binomial Tree Algorithm Cost

I defined cost model of the binomial algorithm for the number of processes p. I used the common (α, β) model [104]. In this model, the execution time of a point-to-point communication is $\tau = \alpha + \beta * n$. α (latency) represents the startup cost, β (bandwidth) is the per-byte cost, and n is message size (bytes). Intuitively, α determines performance for small messages, while $\beta * n$ controls larger messages. Collectives are composed of point-to-point messages between p processes, so I modeled them by scaling the equation by (p). For example, a naive broadcast, where the root sends to every process sequentially, is thus $\tau = p(\alpha + \beta * n)$. For the reduction collectives, I also included γ (per-byte computation cost). Collective algorithms overlap point-to-point communications to reduce the impact of α or β. Note that I used the same symbols/model in all analyses.

The costs of binomial tree algorithms for the simpler collectives are shown in (4.1). Note that I include *MPI_Gather* as a prerequisite for *MPI_Allgather*.

$$T(n,p) = \begin{cases} \log_2(p)\alpha + n\log_2(p)\beta & \text{Bcast} \\ \\ \log_2(p)\alpha + n\log_2(p)\beta & \\ +n\log_2(p)\gamma & \text{Reduce} \\ \\ \log_2(p)\alpha + n\frac{p-1}{p}\beta & \text{Gather} \end{cases} \tag{4.1}$$

Allgather and allreduce are implemented using a gather or reduce followed by a bcast, as shown in (4.2).

$$T(n,p) = \begin{cases} \log_2(p)\alpha + n(\log_2(p) + \frac{p-1}{p})\beta & \text{Allgather} \\ \\ \log_2(p)\alpha + n(\log_2(p) + \frac{p-1}{p})\beta & \\ +n\log_2(p)\gamma & \text{Allreduce} \end{cases} \tag{4.2}$$

In these models, the recursive tree structure causes the latency overhead α to scale *logarithmically* with the number of processes, *p*. Hence, the algorithm performs well for latency-bound, small message operations.

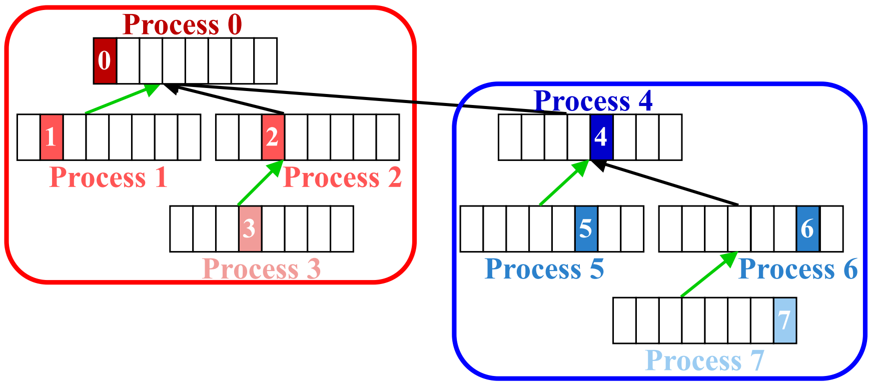


Figure 4.1: **Binomial Tree Algorithm for *MPI_Gather*.** The recursive structure allows all sub-trees to be processed in parallel.

4.2.3 The K-nomial Tree Algorithm

The goal of the k-nomial generalization is to further reduce the latency penalty for small message collectives. Thanks to overlapping communications, the latency cost of a basic binomial tree is the latency of a point-to-point communication times the depth of the tree. Therefore, decreasing the depth of the tree can reduce the overall communication latency.

Binomial trees have an assumed radix of 2, which I generalized to create the k-nomial tree algorithm. Figure 4.2 shows a trinomial tree $(k = 3)$, still with 8 processes. In a full k-nomial tree, the sub-trees rooted at a non-leaf node is identical to the sub-tree at that node's first $(k - 1)$ children. Notice how in Figure 4.1, the 8^{th} node increases the depth of the tree. However, Figure 4.2 shows how a trinomial tree can hold up 9 nodes without increasing the depth.

Increasing the radix of k-nomial tree flattens the structure by overlapping communications within a given level of the tree. In Figure 4.2, the messages from processes 1 and 2 to 0 (and 4+5 to 3) are executed simultaneously as highlighted by the green arrows. To overlap these messages, k-nomial leverages multi-port/message buffering (§4.1.2), so that a single endpoint can send/receive multiple messages simultaneously.

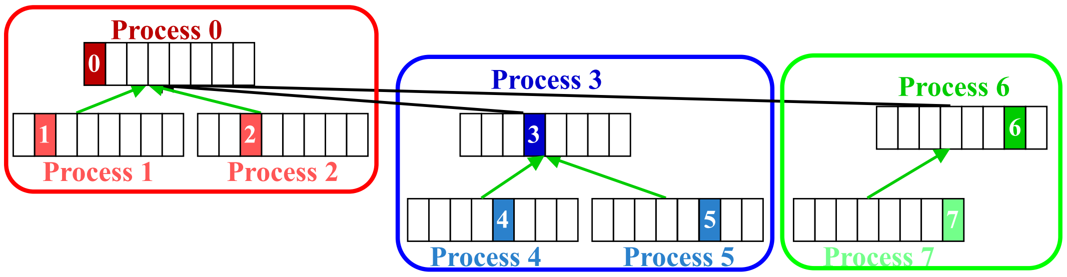


Figure 4.2: **Trinomial Tree Algorithm (k=3) for *MPI_Gather*.** Decreased tree depth increases the parallelism per subtree.

4.2.4 K-nomial Tree Algorithm Cost

K-nomial tree algorithms change constants in the cost model, so tuning the parameter value controls the impact of each term. The costs of the k-nomial tree algorithms are shown in (4.3).

$$T(n,p,k) = \begin{cases} \log_k(p)\alpha + (k-1)n\log_k(p)\beta & \text{Bcast} \\ \\ \log_k(p)\alpha + (k-1)n\log_k(p)\beta & \\ +(k-1)n\log_k(p)\gamma & \text{Reduce} \\ \\ \log_k(p)\alpha & \\ +(k-1)n(\log_k(p) + \frac{p-1}{p})\beta & \text{Allgather} \\ \\ \log_k(p)\alpha & \\ +(k-1)n(\log_k(p) + \frac{p-1}{p})\beta & \\ +(k-1)n\log_k(p)\gamma & \text{Allreduce} \end{cases} \tag{4.3}$$

Larger *k* values decrease the effect of latency (α) and increase the effect of the bandwidth (β). For very small message sizes, bandwidth is a non-factor, so increasing *k* should improve performance. Smaller *k* values should perform better for larger messages when bandwidth is the limiting factor.

These models assume multiport and message buffering enable perfect overlapping of messages with shared endpoints. The optimal *k* value and the performance gain from the k-nomial algorithm are dependent on this assumption. An ideal overlapping would result in an optimal *k* value for very small messages at or near p. However, it is possible that the physical number of network ports caps the number of overlapping communications per endpoint, lowering the optimal *k*.

4.3 Recursive Doubling and Recursive Multiplying Algorithms

I now consider the recursive doubling algorithm, which performs best for small-to-medium message sizes (1B-512kB). For these sizes, latency is again the dominant bottleneck. In current MPI implementations, recursive doubling is commonly used for small-to-medium message sizes because it minimizes the number of sequential communication rounds.

4.3.1 The Recursive Doubling Algorithm

Recursive doubling is a pairwise exchange algorithm where during every round, each process is assigned a peer with which to exchange information. As an example, consider Figure 4.3, which depicts a recursive doubling allgather with 4 processes. In total, there are two communication rounds. In the first round, processes exchange data with peers that are $2^0 = 1$ apart. Peers are formed from groups of size $2^0 = 1$ between odd and even-numbered groups. In the second round, processes in odd and even groups of size $2^1 = 2$ exchange their accumulated data with peers that are $2^1 = 2$ apart. The amount of data exchanged doubles every round.

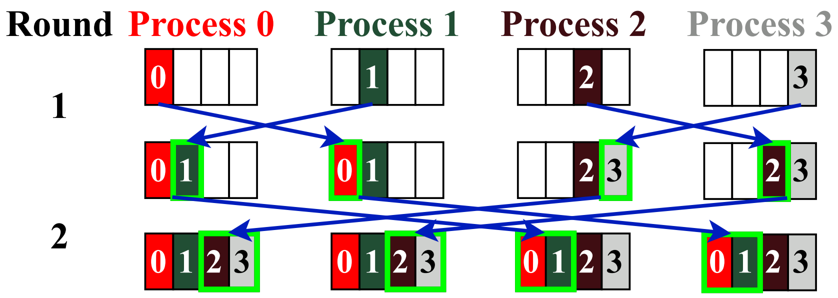


Figure 4.3: **Recursive Doubling for *MPI_Allgather*.**

4.3.2 Recursive Doubling Algorithm Cost

The cost models for the recursive doubling algorithm assuming a power-of-two number of nodes are shown in (4.4).

$$T(n,p) = \begin{cases} \alpha \log_2 p + \beta n \frac{p-1}{p} & \text{Allgather, Bcast} \\ \\ (\log_2 p)\left(\alpha + (\beta + \gamma)\, n\right) & \text{Allreduce} \end{cases} \tag{4.4}$$

The cost of round i is given by (4.5).

$$T_i(n,p) = \begin{cases} \alpha + \beta n \frac{2^{i-1}}{p} & \text{Allgather, Bcast} \\ \\ \alpha + (\beta + \gamma) n & \text{Allreduce} \end{cases} \tag{4.5}$$

Like binomial, recursive doubling scales logarithmically with latency, making it a good choice for smaller messages.

4.3.3 The Recursive Multiplying Algorithm

Recursive doubling, by only doubling the amount of data sent in each round, can induce unnecessary latency with many rounds of small message sizes. My recursive multiplying kernel balances the latency and bandwidth trade-off through the number/size of rounds.

Recursive multiplying introduces parameter k that controls the number of communication partners each round. For each round i, every process exchanges data between $k-1$ other processes spaced a multiple of k^{i-1} apart, with the specific pairings chosen by dividing p processes into k^i groups.

Figure 4.4 shows an example recursive multiplying implementation of allgather with $p = 9$ processes and $k = 3$. Despite the added processes, the allgather still completes in just 2 rounds. Sending more messages per round decreases the number of rounds, improving performance for small-medium messages.

4.3.4 Recursive Multiplying Algorithm Cost

Recursive multiplying's cost model in (4.6) is similar to (4.4) except for the recursive base k.

$$T(n,p,k) = \begin{cases} \alpha \log_k p + \beta n \frac{p-1}{p} & \text{Allgather, Bcast} \\ \\ (\log_k p) & \\ \cdot \left(\alpha + (\beta + \gamma)\,(k-1)n\right) & \text{Allreduce} \end{cases} \tag{4.6}$$

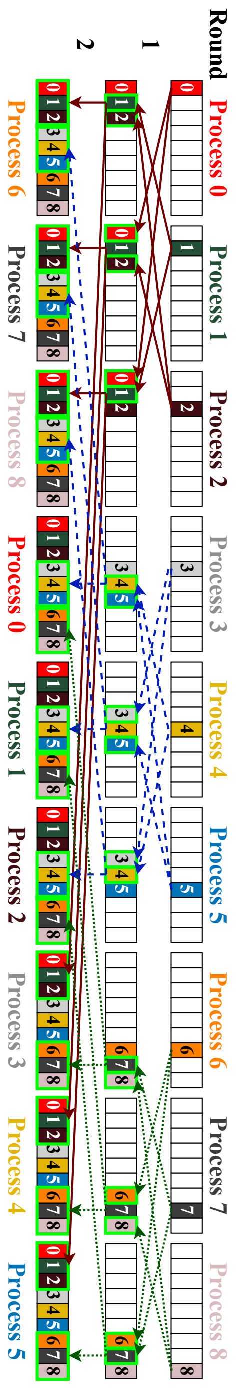


Figure 4.4: **Recursive Multiplying for *MPI_Allgather*.** Each round, processes exchange with two other nodes using a power-of-3 offset.

The parameter k increases the per-round cost of the recursive multiplying algorithm. The cost for the i^{th} round is now (4.7).

$$T_i(n,p,k) = \begin{cases} \alpha + \beta n \frac{(k-1)k^{i-1}}{p} & \text{Allgather, Bcast} \\ \\ \alpha + (\beta + \gamma)(k-1)n & \text{Allreduce} \end{cases} \tag{4.7}$$

The per-round bandwidth and computation costs increase to accommodate multiple messages per round. The validity of this strategy once more depends on the overlapping capabilities of the multiport network and message buffering (§4.1.2).

4.4 Ring and K-Ring Algorithms

Last is the ring kernel. Ring is used for larger messages, where the communication bottleneck shifts to bandwidth. The ring algorithm provides a bandwidth-optimal implementation by using neighbor-only communication.

4.4.1 The Ring Algorithm

In the ring algorithm, processes only communicate with their two neighboring processes. Each round, every process receives new data from its left neighbor and forwards the received data from the previous round to its right neighbor in a ring-like fashion. Figure 4.5 gives an example of the ring algorithm used for the allgather collective with 6 processes.

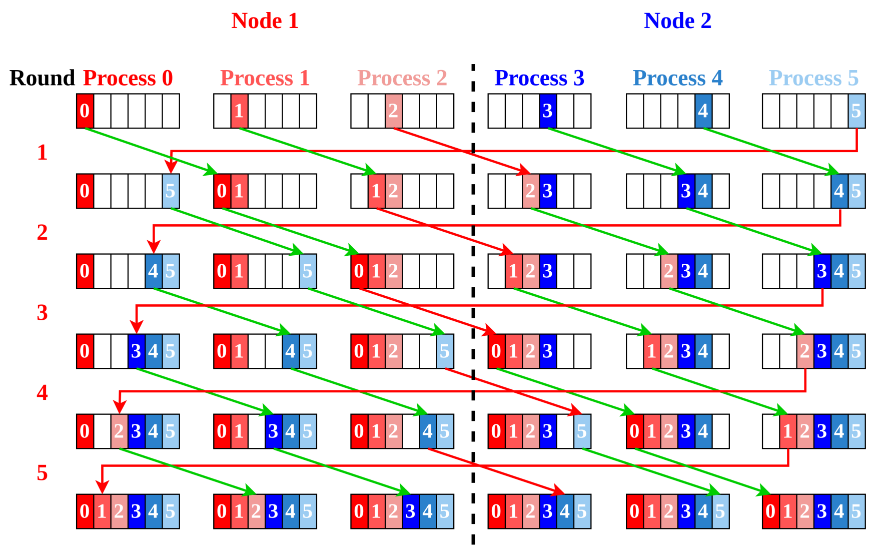


Figure 4.5: **Ring algorithm for *MPI_Allgather*.** In each round, processes forward data to their cyclic adjacent neighbors. Internode communications (in red) slow the entire algorithm.

4.4.2 Ring Algorithm Cost

We define the cost model for the ring algorithm for allgather, allreduce, and part of bcast in (4.8).

$$T(n,p) = (p-1)T_i \tag{4.8}$$

Given p processes, the ring algorithm will have $(p-1)$ rounds of communication, where the single-round cost T_i is (4.9).

$$T_i(n,p) = \begin{cases} \alpha + \beta\frac{n}{p} & \text{Allgather, Bcast} \\ \\ \alpha + \beta\frac{n}{p} + \gamma\frac{n}{p} & \text{Allreduce} \end{cases} \tag{4.9}$$

When compared to the recursive doubling algorithm, ring has a worse latency term (log $\rightarrow$ linear) and equivalent bandwidth term (both linear). Nonetheless, ring is preferred for large messages in practice because the neighbor communication minimizes network hops and congestion, which would limit bandwidth.

Given sufficiently large message sizes, the cost of the ring algorithm reduces roughly to (4.10), which is independent of latency and the number of processes.

$$T(n) = \begin{cases} \beta n & \text{Allgather, Bcast} \\ \\ \beta n + \gamma n & \text{Allreduce} \end{cases} \tag{4.10}$$

4.4.3 The K-Ring Algorithm

The classic ring algorithm is optimized for bandwidth, but it does not consider the extremely high-bandwidth intranode links on modern supercomputers (§4.1.3). When applications use one MPI process per GPU on exascale systems, the more powerful links create a discrepancy in communication cost between processes in the ring algorithm. This heterogeneity is unfavorable for the ring algorithm which has an implicit barrier between rounds, so processes with intranode neighbors are starved for data by the slower internode links.

To illustrate this challenge, reconsider the example in Figure 4.5. The processes in this example are split across two nodes, demarcated by red and blue. Faster, intranode communications are further labeled in green, and slower, internode communications are in red. Because every round includes at least one slower communication, the entire round will perform at that inferior rate.

To reduce the impact this bottleneck, the generalized k-ring algorithm breaks the communication into multiple, smaller "rings." k determines the size of the smaller ring groups; for p total processes, there are $\frac{p}{k}$ groups. Every process has two pairs of left (receive) and right (send) neigh-

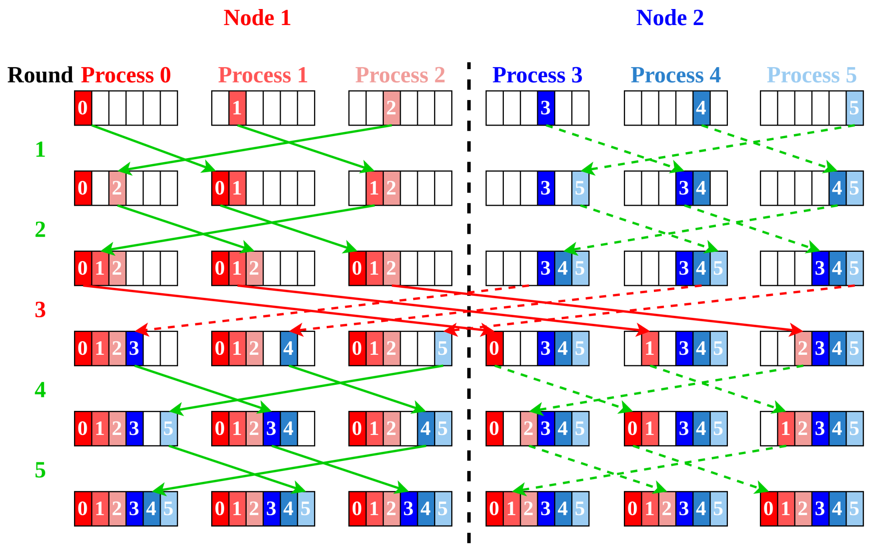


Figure 4.6: **K-Ring Algorithm for *MPI_Allgather*.** Two faster intranode rounds alternate with a slower internode round.

bors, one within its group and one with another group. For the communication pattern, the k-ring algorithm is carried out in a series of alternating intra-group communication rounds and a single inter-group round using the two ring structures.

An example of the k-ring algorithm for *MPI_Allgather* with 6 processes and group size $k = 3$ is shown in Figure 4.6. In the first 2 rounds, processes within groups communicate in rings of size k. Therefore, the third round is inter-group communication with processes passing data to their inter-group neighbors. Following the third round, another two rounds of intra-group communication complete the allgather.

The goal of the k-ring algorithm is to prevent bottleneck links from slowing the entire communication. Within the smaller rings, communication is faster and more consistent because processes are likely to be physically closer (e.g., within the same node) in the system topology. Inter-ring communication rounds are slower, but they are far less frequent (e.g., in Figure 4.6, there are 4 intra-ring rounds to only 1 inter-ring round).

4.4.4 K-Ring Algorithm Cost

With the per-round cost T_i, the k-ring algorithm cost model is split into $g(k-1)$ intra-group and $(g-1)$ inter-group communication rounds, where the number of groups is $g = \frac{p}{k}$. The intra-group and inter-group costs are shown in (4.11).

$$\begin{cases} T_{\text{intra}}(n,p,k) = g(k-1)T_i \\ \\ T_{\text{inter}}(n,p,k) = (g-1)T_i \end{cases} \tag{4.11}$$

Hence, the total cost is (4.12).

$$\begin{aligned} T(n,p,k) &= T_{\text{intra}} + T_{\text{inter}} \\ &= (p-1)T_i \end{aligned} \tag{4.12}$$

The advantage of k-ring is the reduction of data exchanged between groups. In the example shown in Figure 4.6, given each partition of size ϕ, the total inter-group data sent and received by Group 0 is 6ϕ. For the ring algorithm, the total inter-group data sent and received would be 10ϕ. To generalize this idea for p nodes and an intra-ring group size of k, the amount of data sent/received by a group for the k-ring algorithm is (4.13).

$$D_{\text{k-ring}}(n,p,k) = 2n\frac{p-k}{p} \tag{4.13}$$

This formula reduced to the classic ring algorithm (*k=1*) is (4.14).

$$D_{\text{ring}}(n,p) = 2n\frac{p-1}{p} \tag{4.14}$$

Should there be bandwidth bottlenecks between groups of processes, the generalized ring algorithm can be chosen with an appropriate group size k to reduce its effect.

Implementation of the k-ring algorithm for allgather and bcast are identical since bcast uses a "scatter-*allgather*" algorithm. The implementation of allreduce is slightly different as the partitions are offset by 1.

Given the difference between intranode and internode links on exscale systems, the best radix value for k-ring should be the number of processes per node. However, there may be additional bottlenecks between nodes that are farther apart. The k-ring algorithm provides the flexibility to explore the realities of the intranode and internode topologies.

4.5 Performance Evaluation of Generalized Algorithm

I now describe how I integrated the new algorithms into MPICH, detail the experimental methodology to test their performance, and analyze the results.

4.5.1 MPI Library Integration

I implemented each new algorithm in the MPICH source code based on the non-generalized version if it exists. Implementations range from 100–400 lines of code, with *MPI_Bcast* being the longest because of the multi-phase communication for recursive multiplying and k-ring. The largest burden was ensuring correctness for the many corner cases induced by our generalizations (e.g., non-uniform group sizes for the recursive multiplying and k-ring algorithms).

4.5.2 Experimental Methodology

To directly showcase the benefit on my generalized algorithms, I tested them on Frontier, the world's first exascale supercomputer, at Oak Ridge National Laboratory. Frontier contains 9,408 compute nodes, each equipped with one 64-core AMD EPYC 7A53 CPU, four AMD MI250X (eight logical GPUs), and 512 GB of DDR4 memory. The GPUs within each node are connected to each other via Infinity Fabric and the network via 4x200 Gb/s links for a per-node bandwidth of 800 Gb/s.

I compared the algorithms to two baselines using the OSU microbenchmark suite [166]. First, I compared each algorithm against the existing, non-generalized algorithm in MPICH [154], meaning I tested k-nomial against MPICH's binomial, etc. This practice isolates the improvement gained by generalization.

Then, I compared against Cray MPI. For each case, I compared the best performing generalized algorithm and parameter value against Cray MPI's default selections. These experiments highlight the total speedup from generalized algorithm and autotuned selections (e.g., ACCLAiM). Cray MPI is the vendor-supported, state-of-the-art MPI implementation on Frontier. Therefore, these values represent the speedup a user could experience due to the adoption of all my transparent optimizations for distributed memory communication.

I ran all my experiments on Frontier in both 32-node and 128-node configurations. I tested with both common programming models for the machine: 1 MPI process per node (PPN) (1 MPI process per GPU) and 8 MPI PPN (MPI + X). I performed each experiment multiple times to account for runtime variance and selected representative trials for visualization and analysis. Overall, I found the results to be very similar for both 32 nodes and 128 nodes and with 1 MPI process per node and 8 processes per node. I proceed to focus on the 128 node with 1-PPN results and highlight the scenarios where the results diverge.

Beyond the core set of results, I also include experiments using 1024 nodes on Frontier to measure how the performance improvements from generalization scale to leadership-class applications. Due to limited resources and job length, I was only able to test the most promising configurations identified at smaller scale. Finally, I tested how the improvements translate to other exascale hardware by using another system, Polaris. Polaris is a pre-exascale system, but it shares the hardware commonalities from Section 4.1. Polaris contains 560 multi-GPU nodes connected by a dragonfly network, each with one 32-physical-core AMD EPYC Milan 7543P, four NVIDIA A100 GPUs fully connected with 600GB/s NVLink, 512GB of DDR4 memory, and two Slingshot network ports via 64 GB/s PCIe Gen4 for internode communication.

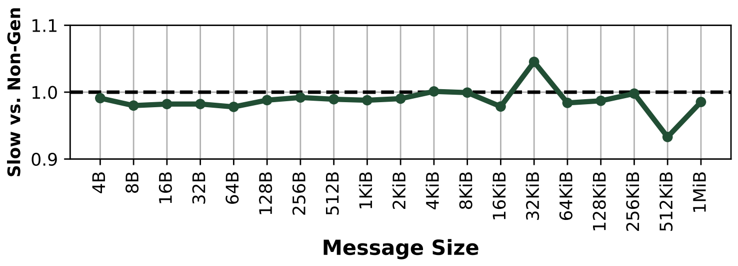
4.5.3 Frontier Results

In this section, I analyze the core results from Frontier. To summarize the experiments, I selected representative operations for each generalized kernel. For k-nomial, I chose *MPI_Reduce* because k-nomial is the only new generalized algorithm for *MPI_Reduce*. For recursive multiplying, I show results for *MPI_Allreduce* because it is the most popular collective for exascale applications [211]. For the k-ring algorithms, I selected *MPI_Bcast* because it is utilizes the *MPI_Allgather* ring algorithm, so I encapsulate both collectives and cover all four operations across these selections.

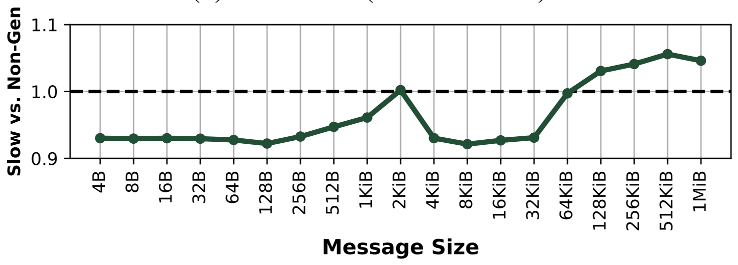
4.5.3.1 Generalized vs. Non-Generalized

In Figure 4.7, I plot the slowdown of the generalized implementation of each algorithm using the default parameter value ($k = 2$, $k = 2$, and $k = 1$) versus the fixed-radix implementation in MPICH. In this configuration, both algorithms are logically identical. These experiments ensure that generalization does not sacrifice performance from MPICH's low-level software tricks (e.g., bitwise operations) that require a fixed radix. In these graphs, a value of 1.0 means that the generalized and non-generalized algorithms have identical performance. Above 1.0 means that the generalized version is slower, and below 1.0 means that the generalized version is faster.

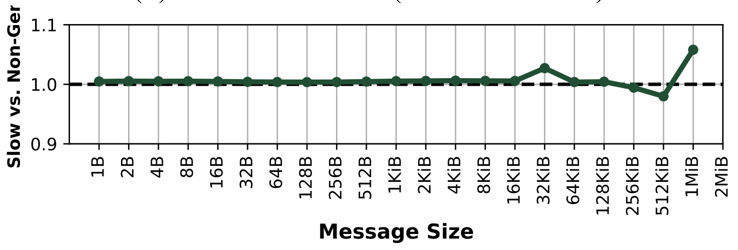
Across all three plots,generalization has a minimal effect on the default radix performance.



(a) K-nomial (MPI_Reduce)

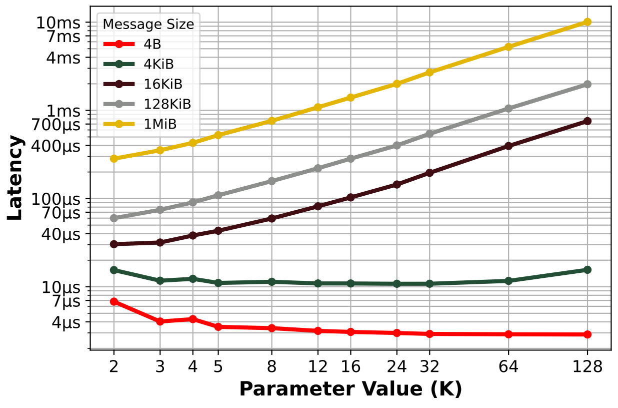


(b) Recursive Mult. (MPI_Allreduce)

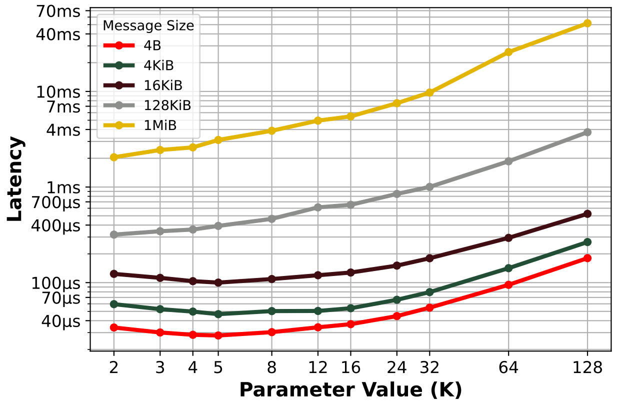


(c) K-ring (MPI_Bcast), 8 PPN

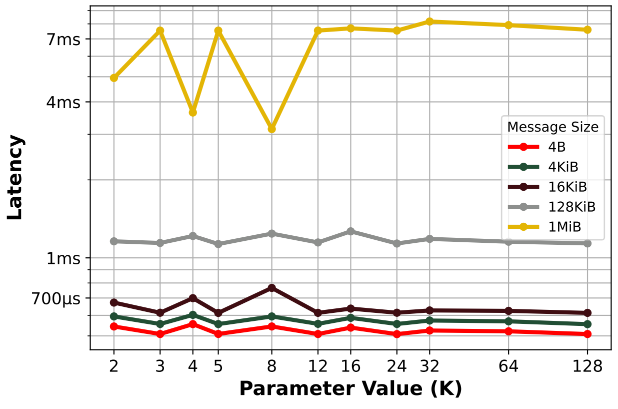
Figure 4.7: **Message Size vs. Slowdown (Lower is Better), 128 Nodes w/ 1 or 8 Process(es) Per Node on Frontier.** Generalization does not result in slowdown.



(a) *K-nomial (MPI_Reduce)*



(b) *Recursive Mult. (MPI_Allreduce)*



(c) *K-ring (MPI_Allgather), 8 PPN*

Figure 4.8: **Parameter Value (K) vs. Latency (Lower is Better), 128 Nodes w/ 1 or 8 Process(es) Per Node on Frontier.** For all algorithms, the parameter value has a significant impact on performance.

Slowdowns do not exceed 1.06x, and the generalized algorithms slightly outperform the fixed-radix version on average, thanks to their careful implementations and noise on the machine. With the implementations proven effective, I used my algorithms with the default radix as a baseline in future figures to further eliminate potential sources of variation.

4.5.3.2 Sensitivity to Parameter Value

Now, I describe how varying the parameter value affects performance. Figure 4.8 plots the parameter values on the x-axis and the latency on the y-axis in microseconds. I plot various message sizes as separate lines on the graph.

I begin with k-nomial in Figure 4.8(a). For the k-nomial algorithm, software message buffering is the dominant performance feature, and the latency/bandwidth trade-off of the message size determines the optimal parameter value. For very small messages (<128 bytes), large parameter values (k=128, i.e., k = the number of processes) greatly outperform small parameter values. As the message size increases, the optimal parameter value decreases. This smooth trend follows the analytical model. Message buffering and multiport functionality overlaps many communications and garner significant speedups, which I later quantify in Figure 4.9.

These results reveal that the network's physical limitation of 4 network ports does not hamper performance. This hardware restriction does not matter in this case because the amount of shared-endpoint communications is relatively small for k-nomial. The one-sided nature of the algorithm means that each thread is only sending *or* receiving *k* messages per round.

Among the other collectives, *MPI_Bcast* had a pattern similar to *MPI_Reduce*. For *MPI_Allgather* and *MPI_Allreduce*, other algorithms outperformed k-nomial for all message sizes, rending their trends irrelevant. This result is logical because the k-nomial kernel maps more naturally to unbalanced collectives where there is a single node sending/receiving the data.

The next algorithm is recursive multiplying in Figure 4.8(b). For the recursive multiplying algorithm, the number of network ports is the dominant performance feature, and the number of ports per node determines the optimal k-value. For all message sizes regardless of magnitude, *k* values at or near 4 (the number of ports per node) are the best-performing choice for *MPI_Allreduce*. This result contradicts expectations based on the analytical models. It occurs because compared to k-nomial, the number of simultaneous messages increases much faster with larger k values. In recursive multiplying, each process sends *and* receives k messages per round, further stressing the multiport network.

MPI_Allgather and *MPI_Bcast* favor *k=4* or a multiple of *k=4*. They do not require receiver-side computation like a reduction, which creates less predictable performance.

The last algorithm is k-ring in Figure 4.8(c), which I tested with 8 processes per node for the 1-MPI-process-per-GPU programming model. For the k-ring algorithm, the intranode links are the dominant performance feature, and the number of processes per node determines the optimal k-value. For larger message sizes, $k = 8$ is the most performant parameter value. When $k = 8$, the "intragroup" and "intergroup" communication steps are effectively "intranode" and "internode" steps, allowing much of the communication to leverage the superior intranode interconnect without implicit synchronization with internode messages.

As part of the *MPI_Bcast* algorithm, *MPI_Allgather* naturally sees similar benefit from k-ring. For *MPI_Allreduce*, the reduce-scatter-allgather algorithm (which can also leverage the *MPI_Allgather* k-ring algorithm) generally outperforms ring for large message allreduces.

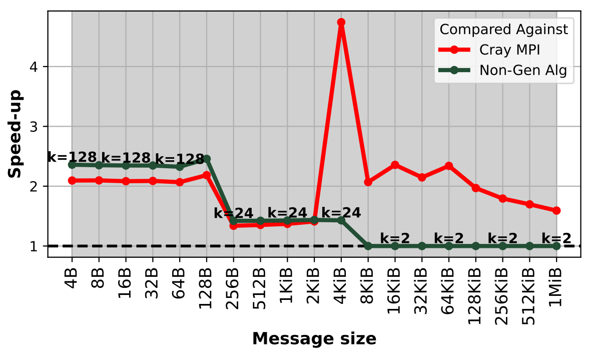
4.5.3.3 Speedup

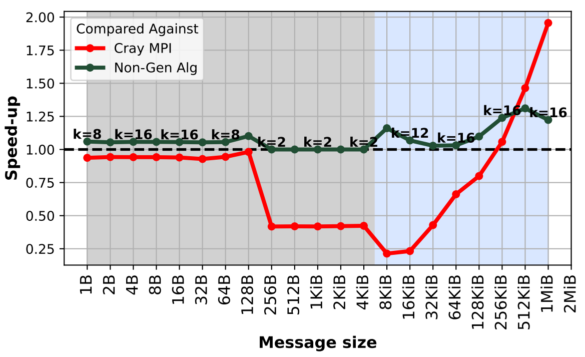
In Figure 4.9, I show the speedup of each collective operation by selecting the optimal algorithm for each message size using the complete results. I denote the algorithm using a color overlay (grey is k-nomial, blue is recursive multiplying). I did not encounter situations where k-ring is optimal over recursive doubling, including additional experiments to study even larger message sizes. The k-ring algorithm gets outperformed because jobs of smaller size are dispersed across the 9000+ nodes in the system, eliminating k-ring's neighbor communication advantage.

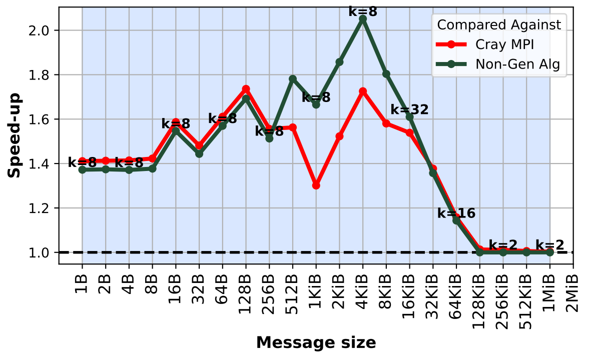
The X-axis is once again the message size, and the Y-axis is the speedup over the two baselines. The dark green line represents the speedup over the default radix of the algorithm to show the speedup from generalization alone. The red line represents the speedup over Cray MPI. The Cray MPI baseline showcases how a current production user stands to benefit from my contributions. Cray MPI occasionally outperforms the best generalized algorithm most likely due to other algorithms, which may be proprietary or hardware-accelerated. In these cases, the user would not actually see a performance degradation; the autotuner would instead select Cray MPI's current behavior.

Now I analyse the results shown in each figure. The first one is *MPI_Reduce* in Figure 4.9(a), for which k-nomial is the only generalized algorithm. As expected from the previous section, the speedup starts out high (over 2.0x) and erodes as the message size increases for the default-parameter-value baseline. Surprisingly, the Cray MPI baseline matches the small-message speedup, meaning that it is also employing the binomial algorithm instead of the more competitive "linear" algorithm. Then, the speedup over Cray MPI soars to over 4.5x, where Cray MPI is likely incorrectly switching algorithms.

The second one is *MPI_Bcast* in Figure 4.9(b), which typically sees little speedup. Message sizes under 256KB have small (1.05x-1.2x) speedups over binomial/recursive doubling and no speedup over Cray MPI. For large messages, recursive multiplying accomplishes its only significant performance improvements (up to nearly 2.0x over Cray MPI) with *k=16*. For *MPI_Bcast*, multiples of four are best for the recursive multiplying parameter value.


(a) *MPI_Reduce*


(b) *MPI_Bcast*


(c) *MPI_Allgather*

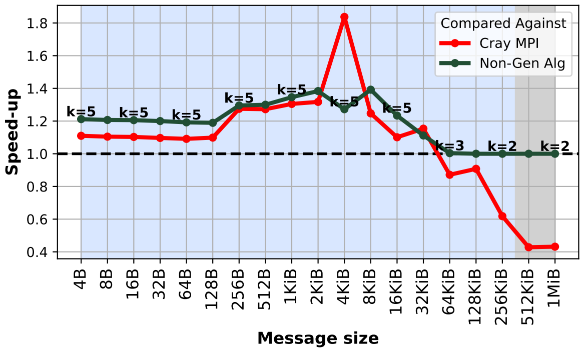

(d) *MPI_Allreduce*

Figure 4.9: **Message Size vs. Speedup (Higher is Better), 128 Nodes w/ 1 Process Per Node on Frontier.** Generalization provides varying speedups over both baselines in most cases.

Third is *MPI_Allgather*, whose speedups are shown in Figure 4.9(c). Nearly all message sizes show significant (1.4x-2.0x) speedups over both baselines. Similar to *MPI_Bcast*, multiples of four are the best parameter value.

The last is *MPI_Allreduce* in Figure 4.9(d). As in Figure 4.8(b), recursive multiplying prefers parameter values near 4, and it generates significant (1.2x-1.8x) speedups. While the optimal parameter value is *k=5*, *k=4* is less than 1% worse on average, meaning the slight win by *k=5* is likely noise. For the largest message sizes in the range, performance improvement tails off as expected from the analytical models.

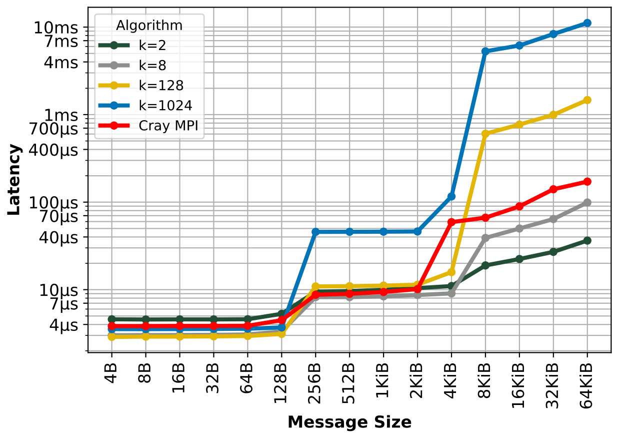
4.5.4 Large-Scale Frontier Results

With performance trends established with smaller node counts, I now show how the performance gains scale up to 1024 nodes. To keep the experiments tractable at this size, I could no longer perform a sweep of all parameter values. Instead, I determined the most promising trends at smaller scale and studied how they translate to larger scale.

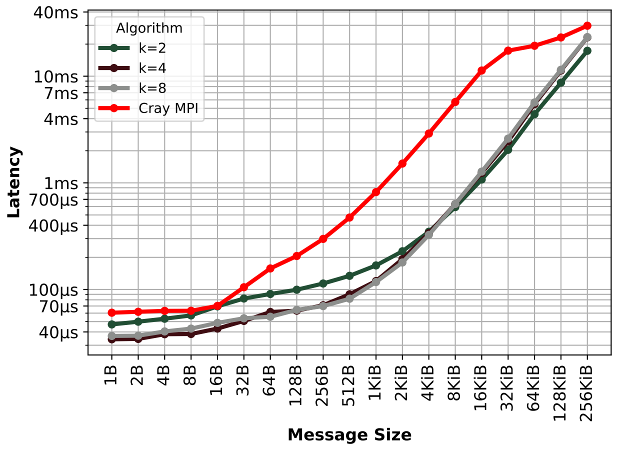
In Figure 4.10, I present three figures representing 1024 node performance for the best k-nomial and recursive multiplying scenarios from smaller scale. In these graphs, I plot the message size on the x-axis again and the latency in microseconds of the various configurations on the y-axis. I include the speedup baselines (Cray MPI and *k=2*) as lines on the graph for easy visual comparison.

In Figure 4.10(a), the performance trends for *MPI_Reduce* k-nomial remain intact; larger parameter values provide significant speedup at smaller message sizes. Interestingly, the parameter value equal to the number of processes (1024) always performs worse than *k=128*. It appears that at large scale, the parameter value may have an upper limit.

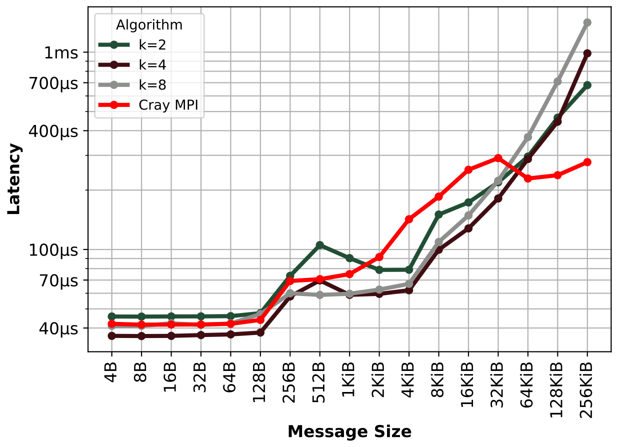
Figures 4.10(b)-(c) show the larger scale performance for *MPI_Allgather* and *MPI_Allreduce*.



(a) *MPI_Reduce (K-nomial)*



(b) *MPI_Allgather (Recursive Mult.)*



(c) *MPI_Allreduce (Recursive Mult.)*

Figure 4.10: **Message Size vs. Latency (Lower is Better), 1024 Nodes w/ 1 Process Per Node for *MPI_Reduce*, *MPI_Allgather*, and *MPI_Allreduce* on Frontier.** The speedups from generalization are maintained at large scale.

These experiments created the most consistent speedups in the smaller-scale tests, and those trends are replicated here. While there is some noise in the *MPI_Allreduce* results (e.g., 512KB performs worse than some larger message sizes), there is consistent speedup from *k=4* and *k=8* until large message sizes.

Overall, the large-scale experiments show how generalized algorithms can provide transparent performance gains for leadership-class use cases.

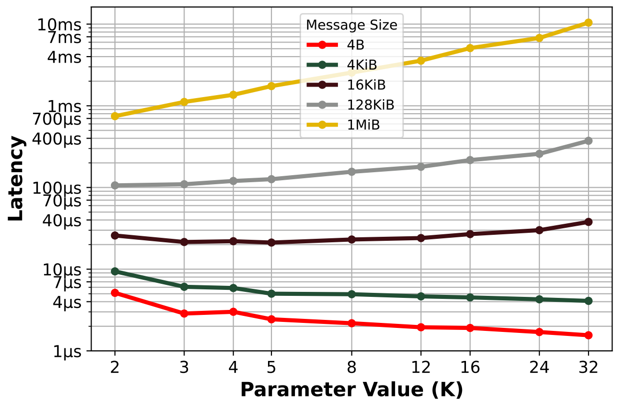
4.5.5 Polaris Comparison

I conclude this evaluation by exploring how generalized algorithms perform on other pre-exascale hardware, specifically Polaris. For these plots, shown in Figure 4.11, I used the same style as Figure 4.8.

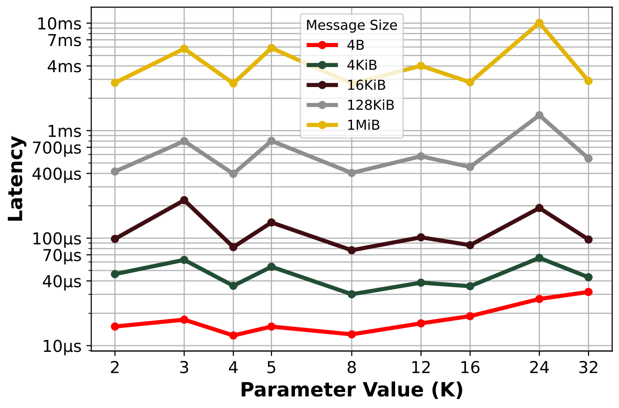
For k-nomial and recursive multiplying (Figures 4.11(a)-(b)), the results match expectations. Just as on Frontier, the optimal k-nomial parameter value for very small messages is close to the number of processes and decreases as message sizes increases. Again matching Frontier, the optimal recursive multiplying parameter value is four or eight, which are the smallest multiples of the two ports per node on Polaris.

For k-ring, however, the parameter value shows minimal affect. Unlike Frontier, Polaris' nodes are fully connected with equal bandwidth between every pair of GPUs. This architecture is less compatible with the k-ring algorithm because many links within a node go underutilized.

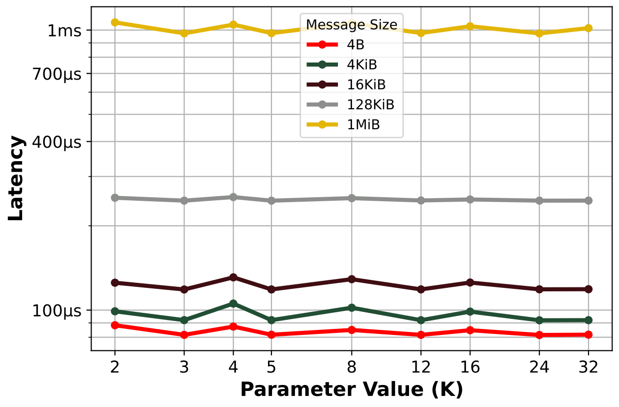
Despite unclear results for k-ring, both k-nomial and recursive multiplying show that the generalized algorithm findings translate from one exascale system to another.



(a) *K-nomial (MPI_Reduce)*



(b) *Recursive Mult. (MPI_Allgather)*



(c) *K-ring (MPI_Bcast), 4 PPN*

Figure 4.11: **Parameter Value (K) vs. Latency (Lower is Better), 32 Nodes w/ 1 or 4 Process(es) Per Node on Polaris.** The trends in how the parameter value affects performance are similar to those observed on Frontier (Figure 4.8).

4.5.6 Evaluation Summary

Overall, my experimental analysis generated many new findings. For k-nomial, I found the software features (message buffering) control performance, and that our analytical models are fairly accurate for optimizing the generalization. For recursive multiplying and k-ring, hardware features (multiport/intranode links) dominate performance, and empirical analysis contradicted the analytical intuition. I showed how these same trends also achieve significant speedups at larger scale and other exascale hardware.

4.5.7 Considerations

To ensure the stability of the results, I re-executed each microbenchmark 4-10x (depending on execution length) within each trial. I included repetition at every level of my experimental methodology (within the microbenchmarks, re-running the microbenchmarks, and running multiple separate jobs on the systems). Still, when re-running experiments to select representative trials and develop our understanding, I encountered significant run-to-run variance, which changed the optimal algorithm selections and parameter values.

These effects are previously documented [110], but they are less well understood, particularly on exascale hardware. Using my manual conclusions alone, I do not claim to have analytically or empirically determined the optimal algorithms/parameters for all cases. Instead, autotuning tools like FACT and ACCLAiM should be used in conjunction with these new algorithms to achieve the maximum performance uplift showcased by our comparison with Cray MPI.

4.6 Conclusion

I used a novel, comprehensive approach to create many new collective algorithms for exascale. By identifying algorithm generalizations that leverage the features of exascale systems, these algorithms provide a wide-sweeping speedup for multiple popular collective operations for two distinct exascale and pre-exascale systems (Frontier and Polaris). This work was published at the CLUSTER'23 conference [235].

As HPC expands to exascale and beyond, new machines will continue to increase in size and complexity. Generalized collective algorithms are a great fit for more complex systems, because they expose easy-to-tune parameters, revealing the best solution for each system. In combination with my autotuning advances, collective operations can be transparently optimized for modern supercomputers.

CHAPTER 5
A NEW PROGRAMMING MODEL FOR HIGH-PERFORMANCE PARALLELISM

The previous chapters describe my multiple transparent optimizations for parallel communication. Throughout these many projects, I consistently wondered whether it is possible to make communication completely transparent, i.e., avoid exposing it the programmer altogether. Seeking the answer, I explored the viability of Functional, Memory-Managed Programming Languages (FMPLs) for high-performance parallel applications. Functional, memory-managed parallel languages (FMPLs) are a recent higher-level approach for shared memory parallelism that, among other advantages, abstract away parallel communication. Despite their rising prevalence in other areas, FMPLs have yet to gain traction for high-performance parallel computing.

Here I describe my work to develop the NAS Parallel Benchmarks (NPB) in an FMPL to understand the usefulness of the programming model in its current state, and the initial results of my ongoing effort to port an FMPL to distributed memory.

5.1 NAS Parallel Benchmarks

I begin by describing the NPB suite and the benchmarks' individual characteristics.

5.1.1 Benchmark History/Relevance

Stemming from NASA's Numerical Analysis Simulation (NAS) Program, the NAS Parallel Benchmarks (NPB) were developed to test supercomputers. The NAS Parallel Benchmarks were designed to mimic the core kernels of computational fluid dynamics programs. They are formally

Table 5.1: **NPB Kernels.**

Abbreviation	Descriptive Name
IS	Integer Sort
EP	Embarrassingly Parallel (Random Number Generation)
CG	Conjugate Gradient
MG	Multi-Grid Solver
FT	3-D Fast Fourier Transform

described as algorithms, meaning any implementation that properly executes the implementation-independent steps is considered valid. This design enables cross-language comparison; two implementations that both adhere to the specified algorithm can be directly compared.

The original NPB 1.0 specification has been cited nearly 4000 times, demonstrating that NPB is a widely used tool for assessing the performance of highly parallel systems [15]. Since NPB 1.0's release, several versions have been developed. I specifically studied the five kernels shown in Table 5.1 from NPB 3.0 [158]. Because FMPLs are currently limited to shared memory parallelism, I compared them with the third-party C+OpenMP implementation [160].

Overall, the NAS Parallel Benchmarks and their implementations are a reliable tool to determine the effectiveness of different high-performance programming techniques, such as FMPLs.

5.1.2 Benchmark Descriptions

The Embarrassingly Parallel (EP) benchmark generates pairs of Gaussian Deviates. The main data structure is a one-dimensional array containing floating point values. EP's main parallelism comes from generating the pairs in parallel, with a parallel reduction to collect results. As its name suggests, nearly all of EP's calculations are parallelizable, meaning it has the most fine-grain regular parallelism in the suite.

The Fourier Transform (FT) benchmark solves a partial differential equation using fast fourier transforms. FT works with three-dimensional arrays that contain complex floating-point numbers. FT's parallelism comes from parallel loops used to iterate over the working arrays. FT's arrays are constant size, so the benchmark exposes a large amount of regular parallelism.

The Conjugate Gradient (CG) benchmark uses the inverse power method to estimate the largest eigenvalue of a sparse matrix. The main data structures are 1-D floating-point arrays. The majority of CG's irregular parallelism comes from parallel iteration over the arrays and parallel reductions.

The Integer Sort (IS) benchmark sorts a pseudo-random set of integers. The main data structures are 1-D arrays of integers. The primary parallelism opportunity is iterating over the integer set in parallel during the sorting algorithm. IS contains a decent amount of parallelism, but its execution pattern is irregular, as the parallel sections are interrupted by sequential, "master" thread-only sections.

The Multi-Grid (MG) benchmark performs several iterations to approximate a solution to a Poisson problem on a 3-D array. The main data structures are 3-D floating-point arrays of various sizes. MG's parallelism comes from parallel loops which iterate through the dimensions and values of the arrays. Due to its complex operations on large arrays, MG is the most difficult kernel to characterize, as is it contains both regular and irregular parallelism.

5.2 Parallel ML: A State-of-the-Art FMPL

I targeted Parallel ML as the FMPL of choice. The following sections describe Parallel ML's advantages over other FMPLs and detail how to write programs using its unique characteristics.

5.2.1 Why Parallel ML

I initially considered 3 modern FMPLs: Data Parallel Haskell (DPH), Multicore OCaml (MOC), and Parallel ML (PML). I eliminated DPH because it is a pure functional language, meaning that all data is immutable. As described in Section 5.1.2, the NPB kernels rely on arrays. In a pure functional setting (no mutations), array manipulations involve copies. This approach causes significant slowdowns because updating an array field is not as simple as updating (i.e., mutating) the proper memory location.

MOC and PML support mutable arrays and are considered generally performant [206, 230, 8]. For this work, both languages are acceptable choices. However, as I later describe, I have continued this work with changes to the language runtime. I chose PML over MOC because its design and underlying runtime are more research-focused, whereas MOC's underpinnings are unnecessarily complex for these purposes. Note that both languages are ML derivatives, meaning it is straightforward to translate these findings to MOC; in this spirit, I refer to FMPLs generally when possible to indicate results that are not language-specific.

Eagle-eyed readers may notice the connection between Parallel ML and MPL, the co-design target from Chapter 2. Generally speaking, FMPLs are a subset of HLPLs, and MPL's derivative of Parallel ML is in fact a FMPL. For my study of the NPB suite, I again compiled the target programs using MPL [230, 8]. However, this project was my first experience writing programs using the language (for WARDen, I relied on pre-existing benchmarks [204]).

Parallel ML is a memory-managed, functional language with support for data mutation (i.e., side effects) and nested parallelism. Parallelism in PML is exposed through a variety of high-level parallelism constructs.

5.2.2 Programming in Parallel ML

Programming in Parallel ML's is fundamentally different compared to traditional high-performance languages like Fortran and C/++. Throughout this section, I refer to line numbers from the example in Figure 5.1.

```
(×* Immutable Data Types ×*)
val i = 4  (* int *)
val a = 0.0 (* real (double) *)

(×* Mutable Data Types ×*)
val pointer = ref 0 (* ref (pointer) to int *)
val _ = pointer := 1 (* updated ref *)
val contents = !pointer (* dereferenced ref *)

(* 4-element array *)
val array1 = Array.array(i, a)
(* Reading first element *)
val first = Array.sub(array1, 0)
(* Updating first element *)
val _ = Array.update(array1, 0, 1.0)

(×* Functions ×*)
(* Basic func: increment array elem *)
fun incrElem(array, index) =
  let
    val new = Array.sub(array, index) + 1
  in
    Array.update(array, index, new)
  end

(* Higher order func: for-loop *)
fun forLoop((i, j), f : int → unit) =
  if i ≥j then ()
  else (
    f(i);
    forLoop((i+1, j), f))

(* Using HOF: incrementing array *)
val len = Array.length(array1)
val _ = forLoop((0, len), fn i ⇒
        incrElem(array1, i))

(* Parallelizing array increment *)
val G = 1
val _ = ForkJoin.parfor G (0, len) (fn i ⇒
        incrElem(array1, i))
```

Figure 5.1: **Basic Parallel ML Programming Examples**

5.2.3 Immutability by Default

By default, nearly all data in Parallel ML are immutable. Immutable data are values that cannot be changed/updated. In Parallel ML, programs declare immutable values as shown starting at line 1. Primitive types like *int* and *real* (i.e., a double in C/C++) are immutable by default. Simple tasks in other languages, such as incrementing an *int*, cannot be performed in Parallel ML. Instead, the runtime will store the result of the summation in a new location, even if the programmer uses the same name.

On the other hand, mutable values can be updated. Starting on line 7, I showcase the two forms of mutable data in Parallel ML: *refs* and *arrays*. On line 20, the program updates the value at the first index in an array. Note that the return value is stored in "_", meaning the function does not have a useful return value. Instead, the main result is the function's "side-effect": updating the array value.

5.2.4 Functions

As the term "*functional* language" suggests, functions play an out-sized role in Parallel ML compared to lower-level languages. Functions are treated as first-class citizens alongside other types like int and $real$. In practice, this idea means that functions can be stored in variables and used as input arguments and return values. A simple function that increments an array element is shown on line 23.

incrElem takes advantage of the common "let-in-end" structure. This structure enables scoped variables, meaning values like *new* are only valid between "in" and "end".

To fully understand the power of functions in Parallel ML, consider the function shown on line 31. *forLoop* implements a for-loop. By default, Parallel ML does not include an implementation of a sequential for-loop. Instead, programmers use recursion, as shown in the for-loop example.

The *forLoop* function takes another function, *f*, as an input argument. The closest analogue to this behavior in C is passing a function pointer. Because it includes another function, *forLoop* is considered a "higher order function". The program proceeds to use *forLoop* to increment all of *array1* on line 38.

5.2.5 Parallelism

To perform array increments in parallel, a programmer simply swaps out the sequential for-loop for the builtin parallel for-loop, *ForkJoin.parfor*, as shown on line 43. *ForkJoin.parfor* dynamically executes the loop in parallel using work stealing [30]. It only takes one additional parameter: grain size (G). Grain size allows the user to easily control the granularity of parallelism by changing the size of computational units in the application. Grain size has a significant impact on performance.

Parallelization in FMPLs is simpler than OpenMP because of the dynamic parallelization and memory management scheme performed by the language runtime. In FMPLs, there is no user-level concept of threads. Instead, the runtime uses work stealing to distribute parallel computation across the multiprocessor. Work stealing is a dynamic scheduling technique where runtime threads will "steal" tasks from other threads and execute them in parallel. By ensuring that every thread stays busy, work stealing can provide efficiency beyond what is possible using a traditional, static scheduler.

During work stealing, the runtime is responsible for managing memory across the parallel threads. Therefore, important (and challenging) manual optimizations in OpenMP, such as declaring variables as thread-private or shared, are automated by the FMPL runtime. In other words, all parallel communication (e.g., shared variables) is transparent to the programmer.

Parallel ML also recently added support for select high-level parallel operations on data collections [229].

5.3 Fitness of FMPLs for HPC Applications

While I did not perform a formal user study, I evaluated the fitness of FMPLs for high performance applications through my experience porting NPB to Parallel ML. I consider the advantages and disadvantages of the FMPL programming model and compare the performance with the C+OpenMP implementations of NPB.

5.3.1 Advantages of FMPLs for HPC

I found that FMPLs provide many advantages that make them appealing for HPC applications.

5.3.1.1 Functional Programming

The first and most apparent feature of FMPLs is functional programming. The primary advantage of this approach is high-level legibility and abstraction. Every expression within an FMPL is a function with inputs and outputs, which means that the flow and purpose of a new piece of code can be easily read and understood. High-level functional languages also include rich semantics for user-defined datatypes. For example, in FT, I created custom datatypes and functions to elegantly manipulate 3-D arrays of complex numbers.

5.3.1.2 Automatic Memory Management

Another major advantage of high-level functional languages is automatic memory management. All allocations and deallocations are handled by the language runtime. Automatic memory management is particularly attractive for HPC applications, as HPC memory systems are becoming increasingly complex (e.g., NUMA architectures, heterogeneous accelerators, etc).

5.3.1.3 High-Level Parallel Constructs

Parallel ML's support for nested parallelism includes simpler constructs such as *parallel for* loops popularized by OpenMP and more complex parallel data operations such as reduce, map, and filter [230, 229]. In my NPB implementations, I used these operations to expose the same parallelism opportunities as the C+OpenMP code.

The main advantage of these constructs is their elimination of communication-related tasks in OpenMP like declaring shared vs. private variables, which can be difficult for developers to implement in complex scenarios. I also had a positive experience with some of the more complex constructs. Particularly, I frequently used the data-parallel reduction to efficiently parallelize regions of EP, CG, and MG.

5.3.1.4 Foreign Function Interface

Foreign Function Interface (FFI) support is critical to the development process when porting existing code to an FMPL. The FFI in PML allows full bidirectional interoperability, meaning PML programs can directly call C functions and vice versa. When porting code, developers can use the FFI to avoid rewriting non-performance-critical code (e.g., random number generation and data initialization). Additionally, porting can be done incrementally, entirely using the existing code to start and replacing it one function at a time. This process maintains the full functionality of the program while iteratively expanding the scope of the PML section. In my experience, I used the FFI in both these ways to accelerate development. I was able to routinely test the code instead of rewriting the entire benchmark from scratch. Note that using the FFI creates the opportunity for mixed language programs, which introduces concerns about maintainability.

5.3.1.5 File I/O

File I/O is a prevalent and important component of modern high-performance applications, Loading and parsing data from many-gigabyte files into PML is surprisingly feasible. During an early iteration of the benchmarks, I used File I/O to import input data generated by the C+OpenMP benchmarks. Overall, I found FMPLs elegantly handle file I/O, although it was not needed in the final implementation for NPB.

5.3.1.6 Memory Protection/Exceptions

A convenient feature of FMPLs is their robust memory protection scheme. NPB, like most high-performance parallel applications, involves complex iterations over large memory regions. When developing these applications, it is easy to make mistakes. In C, errors like out-of-bounds array accesses may go completely undiagnosed until runtime, where they may induce strange behavior or segmentation faults.

Parallel ML solves this issue by protecting data structures and creating detailed exceptions. For example, an out-of-bounds array access halts the program and produces an ”Array: Subscript” error. These exceptions point developers to their mistakes, which greatly improved my debugging experience.

While memory protection is beneficial during the development process, it comes at a performance cost. During execution, the language runtime must perform a bounds check for every array access.

5.3.2 Disadvantages of FMPLs for HPC

Despite their many advantages, FMPLs also introduce drawbacks that are important to consider.

```
(** C Version of MG Benchmark **)
if (Class == "A" || Class == "S" || Class =="W") {
  c[0] =  -3.0/8.0;
  c[1] =  1.0/32.0; ...}
else {
  c[0] =  -3.0/17.0;
  c[1] =  1.0/33.0; ...}

(** PML Version of MG Benchmark **)
fun smallClass(class: string) =
class = "A" orelse class = "S" orelse class = "W";
if smallClass(CLASS) then
  let
    val _ = Array.update(c, 0, ~3.0/8.0)
    val _ = Array.update(c, 1, 1.0/32.0) ...
  in() end
else
  let
    val _ = Array.update(c, 0, ~3.0/17.0)
    val _ = Array.update(c, 1, 1.0/33.0) ...
  in() end;

```

Figure 5.2: **Porting an *if* Statement from C to PML in the MG benchmark.** PML's functional semantics result in more verbose *if* statements.

5.3.2.1 Functional Programming

While the functional programming paradigm is useful for HPC applications in many ways, it can be unintuitive for programmers, particularly HPC developers who have years of experience with low-level, imperative programming. I also experienced this learning curve. In this vein, certain familiar concepts are more difficult to use. For example, *if* statements are treated as functional expressions, meaning they must resolve to a single value (e.g., the return value of a function call). To illustrate the complexity induced by functional *if* statements, consider the *if* statement in each version of the MG Benchmark shown in Figure 5.2. Using this *if* statement, the program decides between two sets of initial conditions according to class size. Each portion of the *if* statement performs array updates. Bringing this behavior into PML, there is no *if* statement that directly performs the array updates because each update has its own (null) return value. Instead, a "let-in-end" structure assigns the return value of each operation to a dummy value, then performs an empty function call for the *if* statement's return value. Behavior like this example is an obvious headache for developers more familiar with imperative languages.

Another issue caused by the functional paradigm is verbosity. For example, C has special syntax to elegantly index arrays, but Parallel ML uses yet more function calls.

5.3.2.2 High-Level Parallel Constructs

While Parallel ML's parallel constructs simplifies some programming tasks, the high-level abstractions makes some complex parallel control flows difficult to implement.

To illustrate this challenge, consider the IS benchmark. The C+OpenMP implementation of IS is essentially one large parallel section operating on a few thread-private arrays. Figure 5.3 shows an example parallel computation from the eventual implementation of IS in PML. In C+OpenMP, thread private copies prevent a race condition when *prev_buf*'s elements are incremented at various locations according to *key_buff2*.

In PML, no notion of threads is exposed to the programmer, so no version of thread-private copies can be declared. To circumvent this issue, I implemented the kernel using a single shared array. To prevent race conditions on the shared array, I had to manually develop concurrency control. Currently, PML only provides a basic compare-and-swap operation for concurrency control, so I had to create a custom *lock* type and *lock*/*unlock* methods. Developing locks was ultimately straightforward, but this task is far lower-level than the promised programming model of FMPLs.

```
fun for() =
let val l = lock_init()
in ForkJoin.parfor G (0, n) (fn i =>
  let
    val ind = Array.sub(key_buff2, i)
    val cur = Array.sub(prv_buff1, ind)
  in(
    lock(l);
    Array.update(prv_buff1, ind, cur+1);
    unlock(l))
  end)
end

```

Figure 5.3: **A Parallel Kernel from IS.** In the C implementation, *prv_buff1* is thread-private. In PML, thread-private values persist across iterations, so data locks protect a single shared array.

5.3.2.3 Compilation Time

Another pain point with FMPLs is the compilation time. NPB are small applications (<1000 lines of code), meaning the C+OpenMP implementations compile near instantly with any commonplace compiler (GCC, Clang, etc.). PML, on the other hand, can take more than a minute to compile smaller files. This issue is a known challenge for Parallel ML compilers [150], which perform full-program analyses that quickly grow in complexity relative to the program size. These full-program analyses greatly decrease the execution time for the compiled binaries, so they are very important to the compilation process.

During the debugging process, compile time became a significant overhead as I repeatedly tweaked, re-compiled, and tested the applications. For larger-scale HPC applications, compile time when using FMPLs will also grow commensurately to code size, becoming a greater challenge.

The codebases of real-world scientific applications are many magnitudes larger than NPB, so the current compilation time would be intractable for development.

5.3.2.4 Floating-Point Correctness

Low-level languages like C include assumptions regarding how to compute exact floating point values. These assumptions may not be shared by FMPLs, potentially causing errors.

During the NPB development process, I struggled to generate precise floating point values to match the C+OpenMP implementation. Initially, I implemented a replica of the C random number generator, which is used to create pseudorandom input values for multiple benchmarks. I found that the FMPL version of the function produced slightly erroneous values. For example, the C+OpenMP implementation would produce a value of .8081127688**77** while PML would generate .8081127688**80** for the exact same input. These discrepancies caused the otherwise-correct FMPL implementations to produce invalid outputs. In the end, I avoided this issue by using the FFI to directly call the C random number generator.

The root cause is that the C+OpenMP implementation allows programmers to manipulate variables in a way that does not directly translate to FMPLs. Specifically, the C version casts 64-bit double precision values directly to 32-bit integers, which incurs rounding. ALL IEEE compliant platforms provide multiple rounding modes that can be set using rounding control bits. The C+OpenMP and PML implementations are using different rounding modes, leading to different results. Low-level computational minutiae translates poorly to FMPLs, so developers must pay special attention to maintain correctness in their programs. It is important to note that beyond random number generation, there were no other issues with floating point correctness.

5.3.2.5 Debugging Support

Even more so than lower-level parallel languages, there is a lack of source-level debugging tools for FMPLs. Given their cutting-edge nature, none of the FMPLs we considered for this work have debuggers.

Instead, I embedded source-level debug information through print statements, which can be toggled as needed. This approach to debugging is very similar to the one used by the C+OpenMP implementation, and this issue is a symptom of the much broader need for debugging support for parallel programmers.

5.3.3 Performance Evaluation

The performance trade-off for FMPLs' higher-level abstraction is significant. To quantitatively evaluate the fitness of FMPLs for HPC, I compare the performance of the Parallel ML implementations of the NPB kernels to the existing C+OpenMP implementations.

5.3.3.1 Experimental Methodology

For these experiments, I used a single-node, four socket machine with Intel Xeon Gold 6238L CPUs with 384GB of DDR4 memory. Each processor contains 22 physical cores, each with 2 hyperthreads, resulting in 176 logical cores. I collected each measurement 5 times. Overall, there was minimal run-to-run performance variation.

To compile the C+OpenMP NPB implementations, I used GCC version 9.4.0 with the *O3* and *mavx512* optimization flags. MPL only supports source-to-source compilation to C. I therefore compiled Parallel ML to C and then to binaries using MPL and GCC with the exact same flags as the C+OpenMP version.

In Figure 5.4, I present graphs for each NPB kernel showing how performance changes over the three configurable performance factors: input size (i.e., class or class size), thread count, and grain size. For each benchmark (IS, EP, CG, MG, and FT), I performed a sweep over the three factors and measured execution time for both the PML and C+OpenMP implementations.

Each graph varies one parameter along the x-axis, while showing the relative performance dif-

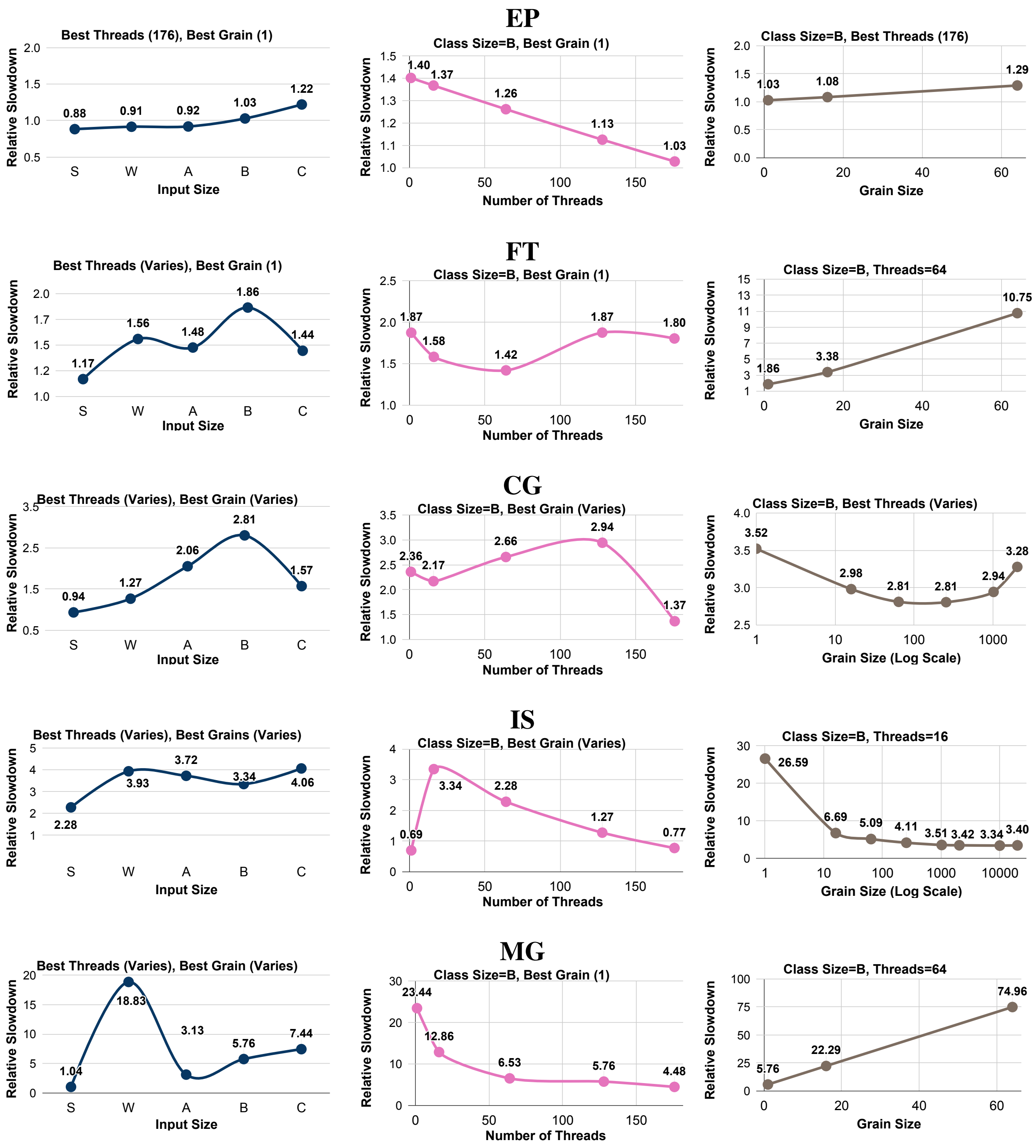


Figure 5.4: **Relative Slowdowns for PML vs. C+OpenMP Implementations of NPB.** Subfigures show results for varying input sizes, numbers of threads, and grain sizes. To choose the fixed parameters for each graph, I selected the largest class size that I fully evaluated (B) and the best-performing number of threads/grain size. These decisions are made per-benchmark, so the graphs for different benchmarks will feature different grain sizes and thread counts. Details regarding the problem sizes and their associated parameter values can be found online [181].

ference between PML and C+OpenMP on the y-axis. I represent the performance difference as the relative slowdown of PML compared to C+OpenMP (i.e., how many times slower the PML implementation is when compared to C+OpenMP). For these experiments, input sizes ranged between S-C, thread counts were up to 176, and grain sizes were up to 64 or 20000 if the benchmark performance improved with greater grain sizes. Regarding grain size, I varied the grain size in PML, but I left the equivalent OpenMP parameter (chunk size) as its default value because the C+OpenMP implementation of NPB does not vary chunk size. When manipulating other variables, I selected the best-performing number of threads/grain size and class size B.

The figures are ordered by relative performance, best to worst. In general, there was a correlation between the amount of parallelism available in the benchmark and the performance of the PML implementation.

5.3.3.2 EP

EP is where MPL's best-performing benchmark. Across input sizes, thread counts, and grain sizes, I measured relative slowdowns at most 25–30% compared to the C+OpenMP implementation. Typically, the performance difference is negligible, and the PML version is actually faster for small input sizes.

EP, as indicated by its full name, "Embarrassingly Parallel", contains abundant regular parallelism. The results here show that PML can reach performance parity with C+OpenMP code when supplied with significant amounts of fine-grained parallelism.

This result is promising and surprising. FMPL's biggest performance advantage over OpenMP was thought to be the dynamic load balancing provided by work stealing. When there is sufficient parallelism to satiate OpenMP's static scheduler, inefficiencies like array bounds checks are expected to slow the FMPL version. However, the FMPL achieves performance parity in this case.

5.3.3.3 FT

The PML implementation of FT maintains relative slowdowns of less than 2×. FT's many array operations expose sufficient regular parallelism opportunities, but the individual tasks are larger than in EP because only the outermost loop of three when mutating the 3-D arrays can be parallelized.

In FT, larger tasks result in a more significant slowdown compared to OpenMP. This result points towards inefficiencies induced by the FMPL's memory management, as larger tasks on multi-dimensional arrays stress this component.

5.3.3.4 CG

The PML implementation of CG produces relative slowdowns of between 2–3× for large input sizes. This benchmark contains many fine-grained parallelism opportunities, but they are irregular, with parallel reductions interspersed among some parallel loops. The small task size is highlighted by the varying grain size plot (brown line), where for the first time, the PML benchmark performs better with larger grain sizes.

This result was the most disappointing of the benchmarks. FMPL's load balancing should provide an inherent advantage over OpenMP for CG's irregular parallelism, but there is still a significant slowdown.

5.3.3.5 IS

IS generally hovers between 3.5–4× slower than the C+OpenMP implementation. IS again contains bountiful irregular parallelism opportunities, but as highlighted in Section 5.3.2.2, it has a complex parallel workflow. In IS, the parallel sections are separated by sequential critical sections only executed by the "master" thread. These sequential sections cause a "join" operation in the

FMPL implementation, while the OpenMP version avoids any slowdown through a large "omp parallel" region and "master" pragmas. As a result, the FMPL IS implementation is one of the worst performers.

5.3.3.6 MG

MG is the worst performing FMPL benchmark. PML MG tends to be 4–8$\times$ slower than the same C configuration, with far more variance than the other benchmarks.

MG's poor performance has multiple causes. The benchmark contains irregular parallelism and many heavyweight tasks, which as shown in the previous benchmarks, cause performance issues for the FMPL. Its core computations require multiple array accesses (i.e., stencil calculations). Frequently, new array values are calculated using four or more older values from multiple arrays. Each of these individual array accesses incur a performance penalty compared to C+OpenMP to support higher-level language features, (e.g., bounds checks to enable protection from Section 5.3.1.6). These slow calculations decrease the parallelism of the benchmark, because like FT, only the outermost loop is parallelizable. For all these reasons, FMPL MG struggles the most compared to C+OpenMP.

5.3.4 Discussion

With respect to their programmability, FMPLs' advantages outweigh their disadvantages. Although functional semantics are sometimes verbose, they quickly become familiar due to the strict and consistent rule set. Then, higher-order functions can quickly scale an implementation. Additionally, by using the FFI, applications can be ported incrementally, and bugs are easier to identify.

Perhaps the most interesting way to highlight FMPL's advantages is what the programmer does *not* have to do. Thanks to the language abstractions, they do not have to worry about threads, mem-

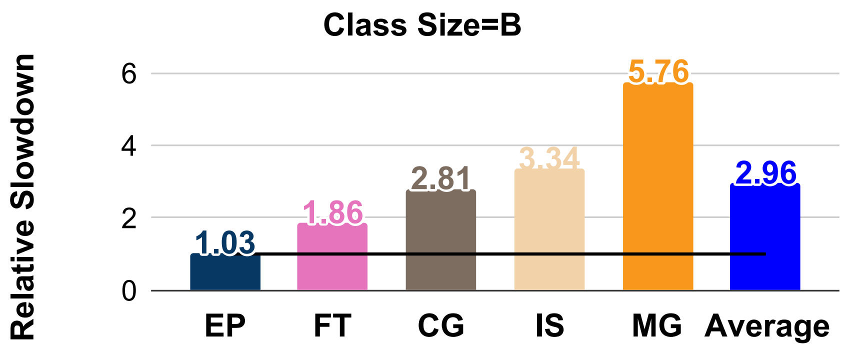


Figure 5.5: **Slowdowns for PML vs. C Implementations of NPB.** The bars represent the relative slowdown for each benchmark where both the C and PML implementations use their most performant parameters (e.g., thread count) for Class Size B. The average across all benchmarks is 2.96×.

ory management, or communication, which makes for a straightforward programming experience.

However, using an FMPL to develop HPC applications can incur a significant performance loss. The magnitude of this slowdown shifted drastically from one application to another depending the amount and structure of parallelism in each benchmark.

Figure 5.5 shows the best-performing thread count and grain sizes for the PML and C implementations of each benchmark for class B. Table 5.2 shows the thread count and grain sizes used to provide the best performance for each benchmark. Figure 5.5 shows a massive difference in relative slowdown between the best-performing PML implementation (EP=1.02×) and the worst (MG=5.76×). The average relative slowdown across all benchmarks is 2.96×.

In their current state, FMPLs are not truly useful to existing HPC developers for a few reasons. Performance is the most obvious concern. Promising results for lightweight, regular parallelism (e.g., EP) show that FMPLs have potential to close this gap with further research, but the current state (average slowdown of nearly 3× is obviously untenable for performance-conscious developers.

Table 5.2: **Optimal Thread Count and Grain Size for Class=B**

Benchmark	Language	Thread Count	Grain Size
EP	C	176	N/A (Default)
EP	PML	176	1
FT	C	128	N/A (Default)
FT	PML	128	1
CG	C	128	N/A (Default)
CG	PML	128	256
IS	C	16	N/A (Default)
IS	PML	16	10000
MG	C	128	N/A (Default)
MG	PML	128	1

More fundamentally, FMPLs lack a groundbreaking feature to out-compete existing shared memory programming models. Interfaces like OpenMP already provide a programmer-friendly, directive-based approach to shared memory parallelism. Seasoned developers who are familiar with OpenMP or an equivalent are unlikely to see benefit from switching to an FMPL.

That said, the future of FMPLs is not all doom and gloom. For example, FMPLs could help more users write highly parallel programs. Data science and deep learning are currently hot topics, and developers in these fields already use high-level languages like Python. Efforts are already underway to provide parallel programming models better suited for these communities [242, 21]. FMPLs could provide a familiar programming model and substantially better parallel performance for these applications.

Lastly, it is important to note that these languages are still nascent and under active research, which may create new features and better performance. FMPL language properties, such as disentanglement, enable promising new optimizations, like WARDen, to overcome the performance gap. In the next section, I detail my effort to extend the Parallel ML runtime by adding *automatic* distributed memory parallelism.

5.4 DMPL: Completely Transparent Communication

FMPLs' most obvious shortcoming in the context of this thesis is their limitation to shared memory parallelism. In shared memory, cache coherence already provides a highly transparent abstraction for programmers. By contrast, distributed memory programs must still explicitly communicate between processes. To address this challenge and merge these domains, I set out to develop DMPL: a distributed FMPL that maintains the original programming model. The idea is to provide a consistent high-level programming experience regardless of the underlying hardware. Users can develop a single program that will run on both a multiprocessor and a supercomputer. Throughout, communication between the many layers of hardware parallelism are managed transparently.

As my final project, DMPL's ambition extends far beyond this thesis. Here I describe my initial intuition regarding how to port an FMPL to distributed memory, my progress so far, and the key outstanding challenges.

5.4.1 MPL Refresher

The MPL compiler's runtime for Parallel ML manages memory in a strict manner that makes it especially suitable for high-performance, distributed computation. Specifically, MPL enforces the "disentanglement" memory property [230], as described in Section 2.1.2. From a systems perspective, disentanglement ensures that concurrent threads remain oblivious to each other's allocations.

MPL automatically ensures disentanglement by enforcing a strict memory heap hierarchy. When a thread forks and creates parallel child threads (e.g., to begin executing a parallel-for loop), each child thread allocates a new, separate heap. Child threads can access their parent's heap, which they can use freely for communication. Because parent threads are suspended while their children execute, they are not concurrent threads, and accesses to the parent heap by the child

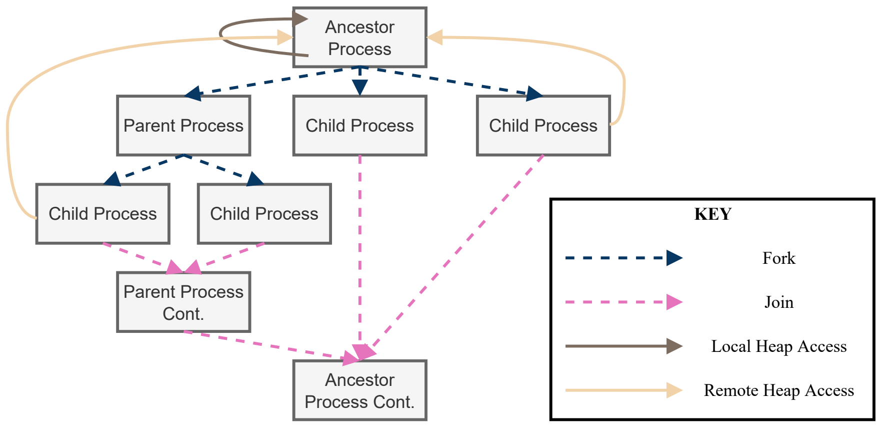


Figure 5.6: **Example Execution Path of a Disentangled Parallel Program.** Memory accesses may occur in the local heap or specific remote heaps depending on forks and joins.

threads do not violate disentanglement. On the other hand, child threads remain unaware of each other's personal heaps. This policy ensures that no child threads can access their sibling's heap concurrently, maintaining disentanglement. The heap management surrounding forks and joins is illustrated in Figure 1.1.

5.4.2 Distributed FMPL Execution

I observe that FMPL programs can be automatically executed across distributed memory systems, where parallel tasks are distributed across separate processes. Importantly, disentangled parallel programs create a specific execution model with two distinct types of memory accesses and four forms of communications overall. An example execution is shown in Figure 5.6. In this example, assume each thread is now a process distributed across separate memory domains. Then, there are four forms of communication that encompass the fork/join execution model and memory accesses therein.

- **Forks** (navy): When a process forks, it must spawn new "child" processes and tell them what computation to complete.

- **Joins** (pink): When all of the child processes complete their computation, they must communicate their results (i.e., return values, relevant portions of their local heap) to the "parent" and exit.

- **Local Heap Access** (brown): During computation, any process may access the local heap that they allocated.

- **Remote Heap Access** (tan): During computation, a child process may access a remote ancestor heap.

Each type can be mapped onto existing patterns in lower-level communication libraries:

- **Forks** (navy): Forks can be performed by distributing work to parallel processes using collective communication (e.g., broadcast the function to run and scatter the input values).

- **Joins** (pink): Joins can also be performed using collective communication (e.g., gather or reduce the results).

- **Local Heap Access** (brown): Local accesses can occur without any communication.

- **Remote Heap Access** (tan): Remote accesses can occur through one-sided operations. Disallowing copies of shared data effectively maintains the processor consistency model expected by fork/join programs.

5.5 DMPL Programming Model

Building upon these ideas, I am in the initial stage of developing DMPL. To simplify the development of parallel programs, DMPL provides a single, consistent programming model for shared and distributed memory. Sequential programming remains identical to existing FMPLs, including functional semantics, immutable data, etc.

Also similar to FMPLs, programmers implicitly declare parallelism through high-level constructs such as parallel for loops and reductions. However, for correctness and optimizations discussed in the following sections, programmer-specified data accesses to any heap within parallel constructs must maintain the WARD property from Section 2.4. Note that this requirement is stronger than in WARDen, which only recognizes leaf heaps maintain WARD by default. However, it does not significantly restrict application behavior due to the broadness of the WARD property. Note that dependencies can exist implicitly, such as in reduction operations.

Recall that WARD's two requirements are (generally): 1.) no read-after-write dependencies, and 2.) write-after-write dependencies can resolve in any order. Assuming that the programmer intuits that concurrent tasks have no inherent order and utilizes the higher-level parallel constructs, they will follow these restrictions naturally. WARD essentially only prevents races that would be non-deterministic in lower-level programming.

In its current state, DMPL relies on the programmer to manually ensure the WARD property. In the future, it is likely that low-cost tooling assist/automate this process. For example, existing tools to detect whether a program is disentangled [229] can help programmers identify and correct entanglement issues.

5.5.1 Functionality Requirements

To automatically execute disentangled programs across distributed memory systems, DMPL needs to match the parallelism functionality of shared-memory FMPLs. The necessary functions are:

1. Task Distribution
2. Mutable Data Updates
3. Write-after-Write Race Resolution
4. Reduction Operations

These functions encompass the execution model from the previous section. “Task Distribution” is the implementation of forks, where new tasks are sent to be executed in parallel.

Next, consider memory accesses. Local accesses do not require additional functionality for DMPL. For remote accesses, recall that FMPLs classify memory as immutable or mutable. Immutable data is read-only, so it can be freely copied and read locally by all processes. On the other hand, DMPL disallows copies of mutable data to maintain coherence and consistency. Therefore,“Mutable Data Updates” is the ability to update these values across memory domains. When multiple data updates occur close together, DMPL must avoid data hazards. Write-after-write races are the only data hazard allowed by the programming model, hence “Write-after-Write Race Resolution”. Lastly, “Reduction Operations” are data updates that combine together using an arithmetic operation. This set of memory functionality encompasses DMPL’s programming model and is sufficient to run significant applications like NPB.

The last part of the execution model are join operations. At these points, data can be merged using the previously described functionality. Then, function clean-up (e.g., freeing memory) can be performed locally as in MPL.

To test whether a DMPL implementation correctly implements these operations, I define a set of test cases, acting as litmus tests that cover each case:

1. **Task Distribution:** Distribute tasks that print the process ID where they are executed.
2. **Update Mutable Data:** Populate an array with random integers in parallel.
3. **Write-after-Write Race:** Search a random array in parallel for some key that may have duplicates.
4. **Reduction Operation:** Sum an array of random integers in parallel.

By successfully completing these test cases, I show that the DMPL prototype is able to cover a significant set of high-performance parallel programs, including the NPB benchmarks. The DMPL prototype provides the required functionality through a set of mechanisms, detailed in the following sections for each required operation.

5.5.2 Mechanisms for Task Distribution

DMPL's scheduler has been modified to support dynamic task distribution. An example of MPL's original scheduler is shown in Figure 5.7.

As mentioned previously, MPL's runtime uses work stealing to dynamically distribute tasks across hardware threads. Work stealing is a ubiquitous scheduling policy because it provides reasonable efficiency guarantees [31]. However, work stealing relies on shared memory to quickly check peer threads for work and perform stealing. In distributed memory, checking other processes is far more expensive because the process must traverse the network. Instead, DMPL uses "work sharing" to dynamically distribute tasks, meaning that a process will proactively push a portion of the tasks to another process. Within each process, the original work stealing strategy then further disperses the computation.

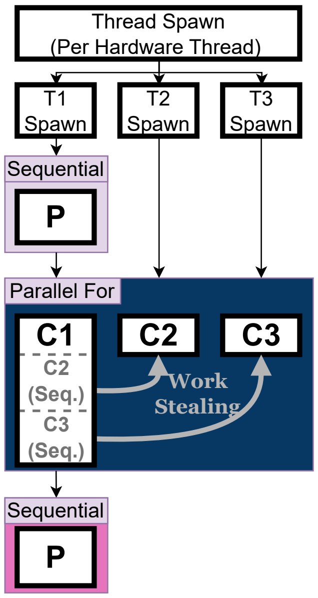


Figure 5.7: **Work Stealing Example.** Tasks spawn on each hardware thread. When threads run out of work (i.e., starve), they steal from another random thread.

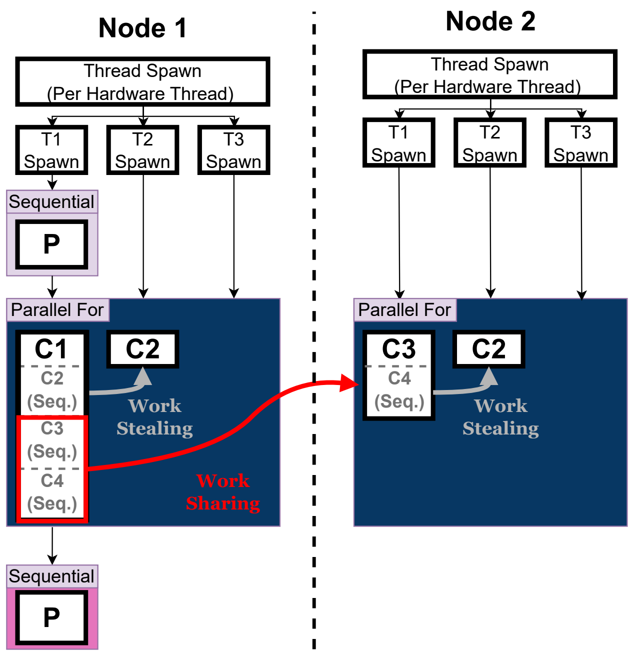


Figure 5.8: **Distributed Scheduler.** Thread 1 on Node 1 shares tasks across memory domains, then work stealing takes place locally.

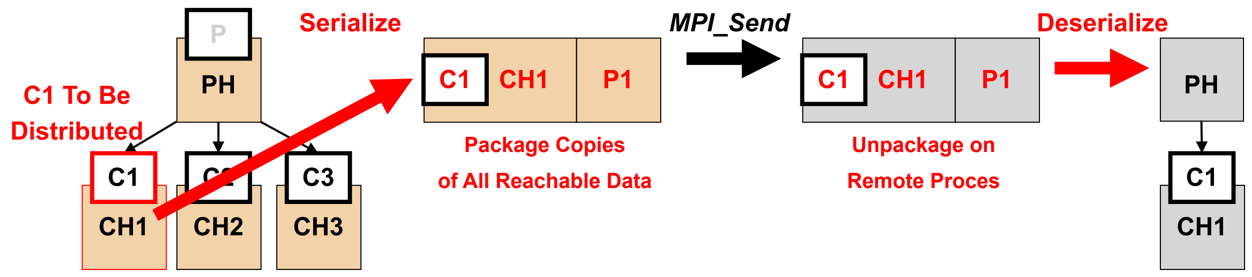


Figure 5.9: **Serializing a Task for Distribution.** The language runtime recursively copies all reachable memory heaps into one contiguous buffer and sends it to the remote process, which unpacks the buffer and executes the task.

An example of DMPL's complete scheduler is shown in Figure 5.8. DMPL uses an additional grain size parameter (G_{dist}) to decide how much work to distribute.

When the scheduler selects a task to distribute, it serializes the task into a contiguous memory buffer and sends it to the remote process using MPI. Upon receipt, the remote process reconstructs the heap and executes the task. The process is visualized in Figure 5.9.

Serialization is a complex process where the language runtime has to package all of the data the task may use. Note that is similar to a closure but simplified thanks to disentanglement; DPLM only must serialize data in ancestor heaps. First, the runtime recursively traverses the memory hierarchy and sums the size of each heap to determine the buffer size. Then, it allocates the buffer and copies the heaps.

During the copy, the runtime must take extra care to not break pointers. Pointers are converted to point to the copy of their target in the buffer. Then, the runtime sends the buffer to the remote process using MPI. The runtime on the remote process deserializes the buffer and executes the task. The base address of the buffer in the original address space is included as metadata during the *MPI_Send*, so the remote process can fix the pointers in its memory. I implemented this functionality in my DMPL prototype, so it is able to correctly execute the first, "hello-world" test case.

5.5.3 Mechanisms for Mutable Data Races

Copying data during serialization ensures correctness for all immutable data in DMPL because once an immutable value is declared, it may be copied and used freely. However, mutable types like refs and arrays may be updated by distributed processes. In this case, the copies may become inconsistent, breaking the correctness of the program. To address this issue, DMPL converts mutable data structures to a dynamic MPI Window when they are first copied for distribution and include the window tag alongside the pointer in the buffer. Then, any time a process updates its copy of the data structure, it also uses a one-sided RMA operation to update the original copy. When reading from the data structure, the process just uses its local copy; by the WARD property, it will never attempt to read an updated value from another process.

Because one-sided operations are non-blocking and each process otherwise only interacts with its local copy, processes operate with the full efficiency of shared memory. Fences are automatically inserted at each join to ensure all updates from a parallel section are completed by every process participating in the join before the program proceeds. The fences also resolve any write-after-write races that appear. Again by the WARD property, DMPL can safely allow these races to resolve in random order through RMA.

The functionality described here is also implemented in my DMPL prototype, and it successfully completed test cases 2 (populate a random array) and 3 (array search).

The last piece of DMPL's current programming model is the parallel reduction. This function is built on top of the mutable data updates and atomic operations. A reference to the final reduction variable is passed as an additional parameter in the buffer. When the remote task completes, it atomically updates the variable with its result using the reduction operation. My DMPL prototype supports parallel reduction, completing the test cases.

5.6 Remaining Challenge: Memory Management

At time of writing, my DMPL prototype supports the complete programming model described in this dissertation, and it is able to execute the microbenchmarks described in the previous section. Developing DMPL has highlighted the need for smarter distributed memory management.

For example, despite the disentanglement optimization, the buffers remain quite large in the DMPL prototype. It is clear that DMPL is conservatively serializing unneeded data, and this area remains a subject for further study.

There are other memory management complexities to consider. First, multiple layers of parallelism will result in the same values being copied repeatedly. Particularly for applications with large data structures like FT and MG, this inefficiency will create a massive bottleneck. In future work, I plan to improve this process by *merging* new data with the existing heap at the remote process.

A series of parallel constructs frequently reuse the same mutable data structures. For example, FT performs multiple sets of parallel computations on the same array. In the current DMPL prototype, all changes to the array are recorded at an ancestor using RMA, then the entire array is re-copied to the other processes when the following construct is distributed. I plan to optimize this bottleneck by maintaining heap data after a parallel task ends. When a new task begins, it will mark the old data structures as "dirty" and lazily retrieve the new values as needed using one-sided communication back to the ancestor.

The current implementation may also run into memory capacity issues. If many processes are sharing a single ancestor data structure, the data structure size will be constrained by the memory available to the ancestor process and not scale with the number of children. To overcome this challenge, I plan to consider many strategies, including automatic sharding for large data structures.

5.7 Conclusion

FMPLs provide an exciting new alternative for HPC applications. My exploration in the context of shared memory parallelism found significant opportunities to improve upon existing distributed memory techniques. This work was published at the HIPS'23 workshop [236].

Work on a new distributed FMPL, DMPL, is ongoing, and it has the potential to benefit HPC developers. The new runtime makes it possible to run some FMPL programs on distributed systems *without modification*. The language runtime utilizes both the existing shared-memory back-end and the new distributed back-end to transparently communicate and execute applications across multiple layers of parallel hardware.

There are countless long-term benefits, including for load balancing, dynamic scaling (e.g., growing if a larger allocation becomes available on a supercomputer or shrinking to make room for an on-demand job) and fault tolerance (e.g., buffers can be cached and re-distributed following single node failures), etc. Additionally, further co-designed optimizations like WARDen could unlock performance beyond what is possible with existing programming models.

Considering these major benefits, distributed FMPLs are an exciting new frontier for programming model research, and I hope my thesis research spurs further efforts in this direction.

CHAPTER 6
RELATED WORK

6.1 Shared Memory Communication

WARDen improves shared memory communication by selectively disabling cache coherence according to language properties. There is substantial prior work in coherence-related optimizations. Removing/deactivating coherence has been proposed before in hardware-supported, compiler-directed (HSCD) cache coherence [59], OS-driven coherence deactivation [62], and software cache coherence [129, 214, 11, 167, 213, 55, 58, 64, 164]. WARDen is distinct because it drives coherence deactivation from the properties of HLPLs, which can provide the information to control coherence by construction. In contrast, these prior works give this task to the programmer (pragmas) or recover this information through run-time inspector-executor methods and compiler analyses. These analyses treat entire arrays as single variables and fail to detect false sharing [55, 58], or limit array subscripts to loop iterators [64]. Others rely on software for triggering coherence actions and hardware for selective self-invalidations but incur high overhead in lock-intensive programs [11] or are restricted to unity loop iterators in affine loops without conditionals [164].

Other works improve cache coherence for discplined memory programs. DeNovo [57, 212] simplifies hardware cache coherence by banning “wild shared-memory behaviors”, but it uses a restrictive programming model that requires user-annotated code. VIPS-M [191] avoids directory accesses and invalidations given data-race-free guarantees from software. This approach is limited compared to WARDen because it only supports DRF programs. In addition, it does not support legacy applications, meaning all programs must enforce DRF. SPEL expands VIPS by implementing a dual cache coherence protocol, allowing for legacy, non-DRF applications [190]. SPEL relies on static compiler analysis to identify DRF code regions, which limits its scope compared to WARDen. The static analysis will fail on any disentangled, non-DRF regions. Also, WARDen avoids any compile-time overhead/analysis. Jimborean et al [119] recognize DRF regions in programs manually parallelized with *pthreads* using static compiler analysis and target the SPEL dual cache coherence protocol. The performance of this approach is limited by the conservativeness of modern alias/pointer-analysis; it is unable to detect up to 50% of potential extended DRF regions. In contrast, WARDen avoids the limitations of compile-time analysis by targeting the disentanglement property of HLPLs. For all these DRF-based works, note that disentanglement encompasses DRF. Therefore, these works could be tweaked to target WARDen, allowing WARDen to support some non-HLPL programs.

Alternative cache designs and protocols [186, 46, 129, 107, 73, 126] and transactional memory [102, 201, 100] also relax hardware coherence according to higher-level directives. However, these schemes only support regions that are limited in time and/or number of addresses. They also require programmer intervention through memory fence annotations and/or transaction boundaries.

Earlier software distributed shared memory work aims to provide coherence in software through middleware [137, 45, 5, 130, 209] or a hypervisor [52], but they are built on the kernel's paging system and share its limitations (expensive faults, large granularity, etc). Shasta [196] supports a fine-grain shared address space through rewriting the application binary to intercept loads and stores and perform in-line checks for sharing, but these checks cause performance degradation. Blizzard [197] offers similar functionality with hardware assistance but also with sizeable overheads. Hierarchical private/shared classification [189] uses hierarchical sharing status, but complicates page management.

6.2 Distributed Memory Communication

I created a series of projects for optimizing distributed collective communication, namely algorithm selection autotuning (FACT/ACCLAiM) and new algorithms (generalized algorithms). Collective optimization dates back more than 20 years [216, 90].

6.2.1 Collective Algorithm Selection Autotuning

My autotuning work builds upon the ideas presented by Hunold et al.[111, 109], who present the idea of using machine learning to autotune collective algorithms. They prove that basic machine learning models without hyperparameter tuning can accurately select collective algorithms. The work in [109] goes further, building an autotuner prototype using basic ML models without hyperparameter tuning. Their paper includes an ad hoc testing methodology that shows an ML autotuner works well on allocations up to 48 nodes on larger supercomputers. FACT and ACCLAiM include many advancements to scale the idea for exascale-era supercomputers.

Many others have proposed methodologies for selecting collective algorithms besides machine learning. The popular approach is analytical models [225, 77, 178, 177, 142, 163]. The most recent of these proposals is by Luo et al. [142]. They create submodules that represent lower-level portions of a collective task, some of which can be mapped to specific hardware components. Analytical models suffer from other minor concerns, but their ultimate downfall is development cost. Compared with the effort required to maintain handcrafted models and analyze new algorithms, ML provides a black-box solution with automatic expandability.

Another approach is exhaustive benchmarking. Chaarawi et al.'s OPTO tool tunes individual scenarios in Open MPI using a complete search [47]. Tools such as OPTO, however, require far too much data collection time to compete with ML models.

Faraj et al. proposed STAR-MPI, an “online” autotuner that builds a statistical model during program execution and dynamically selects MPI parameters [79]. In general, online approaches are rare because of their runtime overhead; decision space exploration directly slows the application performance. Performance modeling/guideline approaches are simpler and also use runtime information to make selections dynamically [112, 202]. However, these tools are restrained by the models/heuristics that guide them, similar to the existing solutions in production MPI libraries.

Machine learning is becoming a prominent optimization tool across HPC. Pellegrini et al. optimized other MPI runtime parameters using ML [173]. Isaila et al. built an ML model to tune I/O tasks [117]. Mohammed et al. used ML to predict failures in a virtualized system/application [151]. Zhang et al. scheduled HPC batch jobs with an ML model [240].

6.2.2 Collective Algorithms

Many past works have optimized collective algorithms. Seminal work on collective algorithms by Thakur et al. [215, 216] laid the foundations for the standard set of collective implementations in MPICH [154]. Other important works include Bruck’s algorithm [37], the n-way dissemination barrier by Hoefler et al. [105], and the ring-based all-reduce algorithm designed by Patarasuk et al. [171].

More similar to my work, others have also created generalized collective algorithms for specific scenarios. Ruefencht et al. proposed a generalization of the recursive doubling algorithm for *MPI_Allreduce* for small message sizes [192]. Ruhela et al. first utilized the k-nomial algorithm to optimize intranode *MPI_Bcast* [193]. *MPI_Allreduce* k-nomial appears in Intel MPI, but its use case is unexplained. Hasanov et al. created a hierarchical structure across reduction algorithms [101]. Recently, Fan et al. generalized the Bruck’s algorithm [37] by developing the padded Bruck and two-phase Bruck algorithms for nonuniform all-to-all communication [78]. My work is inspired by and goes beyond these previous efforts by showing how a single generalized kernel can optimize for multiple collectives on multiple systems.

To further improve collective performance on new and emerging hardware, many works focus on the development of new, topology-aware collective algorithms. Bienz et al. designed a locality-aware Bruck allgather [24]. Gong et al. proposed a set of network-aware algorithms for MPI bcast, reduce, gather, and scatter on cloud platforms [89]. Most recently, Feng et al. simulated collective algorithms specially designed for the standard dragonfly topology [80]. My approach is more general because I do not incorporate the topology directly into the algorithms; instead, I designed system-agnostic algorithms that consider both the topology and other features (e.g., multi-port, etc.), and the variable parameter fits the algorithm to the specific system.

The rising popularity of machine learning has motivated GPU-centric research. Cai and Liu et al. proposed the Synthesized Collective Communication Library (SCCL) to synthesize optimal algorithms on specific GPU topologies [40]. Leveraging new, collective-specific hardware, Haghi et al. offloaded collective operations to in-switch hardware accelerators with two additional modules: a Collective Control Module and a Reduction Unit [96]. Awan et al. pipelined bcast operations during deep learning workloads on GPU clusters [13]. Also for distributed deep learning, Cho et al. [56] decomposed allreduce operations into parallel reduce-scatter and allgather operations. These network/application-specific designs are restricted to the contexts for which they were created. By contrast, my algorithms are transparent to both applications and hardware, meaning they can impact a wider variety of systems and users.

6.3 High-Performance Programming Models

I ported the NAS Parallel Benchmarks to an FMPL (Parallel ML) and am actively developing a new programming model for distributed parallelism, DMPL.

6.3.1 NPB

The NAS Parallel Benchmarks have frequently been used to evaluate the effectiveness of emerging languages for HPC applications. Examples include UPC [72, 120] and Chapel [51]. In addition, NPB is commonly used to compare different HPC programming models [200].

6.3.2 FMPLs

There is a long history of functional languages for distributed memory, such as the MPI backend for NESL [25]. NESL is an ML-like language, and it was one of the original functional languages to support nested data parallelism and automatic vectorization. Other functional languages to support distribution include Erlang [6] and Cloud Haskell [75]. However, both of these languages expose distribution to the programmer through message passing.

Parallel ML is one of many recently developed FMPLs, which maintain a higher-level abstraction compared to the older works via the fork-join programming model. From the ML family of languages alone, other relatively new FMPLs include Multicore OCaml [206, 207] and the Manticore project [84], which has its own dialect of Parallel ML.

Multicore OCaml is an industry-led, shared-memory parallel version of OCaml. In this work, I chose Parallel ML using the MPL compiler because of the planned development of DMPL, which benefits from MPL's research-oriented design. A natural future step is bringing any MPL enhancements to Multicore OCaml.

The Manticore project defines its own dialect of Parallel ML, which includes implicitly threaded data structures such as parallel arrays. I focus on MPL's PML over Manticore because MPL's high-level parallelism constructs are more similar to parallelism APIs currently used in HPC (e.g., OpenMP).

Beyond the ML family, another recent FMPL is Distributed Parallel Haskell [49]. DPH focuses on pure functional programming, which is far less suitable for high-performance applications because it does not support mutable arrays.

6.3.3 Other New Programming Models

A wide variety of new programming models have been created for high-performance parallel programming. Broadly speaking, they can be separated into multiple categories based on the hardware systems they target and/or the type of abstraction they provide to the programmer. A similar summary, which goes into even more depth, can be found at [152].

6.3.4 Shared Memory

Various programming models only target shared-memory systems. Cilk [29, 115, 188] is a task-based model invented at MIT and later supported by Intel. Cilk Plus is a library which extends C/C++ with a non-blocking *spawn* function and a *sync* function. The tasks are structured into a directed acyclic graph (DAG), which effectively implements the fork/join model. TBB [116] is a C++ threading library created by Intel. It allows programmers offload lighter-weight tasks to a pool of OS threads, avoiding expensive OS thread creation/termination. OpenMP [63] is the predominant programming model for shared memory parallelism. It was originally designed to implement the fork/join model in C, C++, and Fortran through parallel for loops, and now also supports asynchronous tasks. OpenMP is the most common shared-memory programming model in HPC, and it is the most popular “X” in “MPI+X”. OmpSs [69, 218] extends OpenMP with more-advanced task concepts from the StarSs programming model [179]. OmpSs also supports executing tasks on heterogeneous hardware. Qthreads [232, 221] is a threading library designed to support massive numbers of user-level threads.

Overall, there are multiple programming models and libraries for shared memory systems that enable lightweight parallel execution. Many, such as Cilk and OpenMP, implement a fork/join programming model that is very similar to the programming model in FMPLs. However, the obvious limitation here is all of these libraries require the underlying shared memory abstraction in hardware.

Some works [20, 143, 144] have attempted to bring fork/join programming in the form of OpenMP to distributed-memory systems. However, these designs rely on lower-level software to implement distributed shared memory (DSM), essentially passing the buck for managing distributed memory down the system stack. DSM is a long-standing research topic with known performance issues. Therefore, it is unlikely these many programming models will reach beyond shared memory any time soon. The remaining categories all support distributed execution.

6.3.5 Fortran Variants

Fortran is perhaps the most longstanding programming language for high-performance applications, dating back to the 1950's [14]. Since then, there have been proposed updates to improve Fortran for modern high-performance applications. Prominent examples include Coarray Fortran [162] and High Performance Fortran (HPF) [141]. Coarray Fortran extended F95 with SPMD *coarrays*, which were later adopted into Fortran 2008. HPF extended F90 with innovative features like parallelism directives, but it largely fizzled [127]. Various flavors of Fortran (new and old) still remain popular in high-performance computing. However, new developments are largely focused on new programming paradigms, as described below.

6.3.6 Partitioned Global Address Space (PGAS)

There are multiple runtimes and languages that support a PGAS abstraction. In a global address space, multiple parallel threads may freely and directly access memory across distributed regions.

The UPC/++ [44, 241] languages are the classical examples of the PGAS abstraction in action. However, UPC and UPC++ are very low-level extension of C and C++, respectively, requiring programmers to explicitly state all communication between the threads, synchronization, the necessary memory consistency model, etc. Chapel [50, 217] is a separate PGAS language that provides a higher-level abstraction compared to UPC/++. It relies on the same underlying communication library (GASNet) as UPC++. X10 [239, 53] is PGAS language built on Java. It includes a few unique abstractions (e.g., data can be stored private memory called "places") and executes on top of the Java Virtual Machine (JVM) runtime system.

Despite years of advancement and new, higher-level languages, the PGAS abstraction remains a fairly niche choice for HPC programming. It is difficult to suggest why HPC programmers seem reluctant to adopt this new programming model. By contrast, DMPL maintains the fork/join programming model HPC programmers are already familiar with through more popular tools, such as OpenMP.

6.3.7 Task Parallel

Task parallel programming models the fastest-growing category in HPC today. The basic idea is that the programmer specifies parallel "tasks" and data dependencies between them. Then, the language/runtime system can automatically execute tasks in parallel across distributed resources.

Charm++ [124] is one the first task-parallel implementations, dating back to the 1990's, and is still relevant work. Charm++ extends C++ to include "charms", which are asynchronous tasks that are distributed to execution resources. HPX [123] also extends C++, but instead targets the ParalleX [122] execution model, which enables asynchronous task-parallel execution. HPX improves on Charm++ by using an *active* global address space (AGAS), meaning data can migrate en masse to better fit the dynamic execution model. Legion [22, 135] is a lower-level task parallel model that requires programmers to explicitly map data into memory partitions. Legion enables fine-grained control of execution across heterogeneous distributed systems for maximum performance. OCR [146] is more recent task-based runtime specifically designed as an alternative to “MPI + X” on exascale systems. Uintah [149, 88, 222] is a set of task libraries specifically designed for large-scale simulation, such as those common in HPC. PaRSEC [36, 169] requires users to specify the dataflow between dependent code segments, which in-turn guides the runtime system to better distribute tasks onto the processing resources.

Task-based programming models are another example of a powerful abstraction that is logically different than longstanding approaches like MPI. Individual applications tend to be biased towards one or the other depending on the structure and regularity of the available parallelism.

6.3.8 Heterogeneous Systems

In this thesis, I focus on addressing the challenge of programming high-performance systems, which are quickly increasing in scale. In addition, these machines are also increasing in heterogeneity. Non-CPU accelerators, such GPUs and FPGAs, are becoming more commonplace in supercomputers. To combat the separate challenge of programming for these devices, multiple heterogeneous programming models have been developed.

OpenCL [220] is the most prominent example. It presents a generic programming interface for functions, which can then be automatically offloaded to a heterogeneous accelerator. The OpenCL runtime manages the translation from one hardware type to another. OpenACC [219] is a heterogeneous programming model specifically designed for CPU+GPU systems. This tight integration allows OpenACC to present a simpler programming interface similar to OpenMP. StarPU [12] is a task-parallel programming model targeted towards CPU+GPU systems. CUDA [61], developed by NVIDIA, is an API for programming NVIDIA GPUs for general purpose applications, including scientific programs. The ROCm interface [4] is AMD's analogue to CUDA, and they also include a tool to convert CUDA code to HIP, the AMD API. Intel's preferred heterogenous programming model is SYCL [187], a C++ library inspired by OpenCL which relies heavily on template functions. Kokkos [70] is another C++ library designed to enable performance portability across CPU and accelerators, such as GPUs. Kokkos programmers operate on data as multi-dimensional arrays, then the runtime the data in memory according to the target architecture. Last is RAJA, another C++ compatibility layer that utilizes almost exclusively template functions [106].

These programming models take cues from both fork/join and task parallel models to tackle heterogeneous hardware. Such systems remain outside the scope of this dissertation. Future work on DMPL could target heterogeneous hardware.

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

Communication is a critical component of parallel programming, but traditional communication mechanisms lag behind the growth of modern hardware. My dissertation focuses on multiple novel advancements to improve their performance for newer systems like manycore multiprocessors and large-scale distributed memory machines like exascale supercomputers *without* disrupting the programming model. For shared memory, I created WARDen, a novel cache coherence protocol, and for distributed memory, I developed multiple advancements for collective communication. Examples include for autotuning (FACT and ACCLAiM) and new collective algorithms (generalized algorithms).

Additionally, I explored a new programming model for high-performance parallelism. I took an exciting new field of parallel languages called FMPLs and examined their usefulness within shared memory. Then, I built DMPL, an expanded FMPL runtime that encompasses both shared and distributed memory.

Going forward, these works serve as proof of the potential for transparent optimizations for programming models old and new. Below, I list the primary contributions on this dissertation.

7.1 Summary of Contributions

Argument The foremost contribution of this work is the argument that longstanding communication abstractions (e.g., shared memory, message passing) can be *transparently* optimized for modern computer systems, and future programming models can abstract away communication altogether. I provided evidence for the validity of this claim through the following projects.

WARDen I designed WARDen, a novel cache coherence protocol for high-level parallel languages. WARDen shows that it is uniquely possible to transparently reign in the cost of cache coherence for HLPLs.

FACT I devised FACT, a methodology for ML-based autotuners for collective algorithm selection. FACT shows that ML autotuners can be trained more feasibly at large scale using a clever training methodology.

ACCLAiM I am the lead and sole developer of ACCLAiM, the world's first ML collective autotuner to be trainable on a large-scale supercomputer.

Algorithms I designed 10 new "generalized" MPI collective algorithms that include a new tunable parameter to better leverage modern systems.

NAS-MPL I ported and maintain NAS-MPL, an open-source implementation of the NAS Parallel Benchmarks in the derivative of Parallel ML defined by the MPL compiler. NAS-MPL is freely available online [159].

DMPL I created DMPL, an extension of the MPL compiler and runtime system that supports dynamic distributed computation without modifying the existing programming model.

7.2 Other Contributions

During my Ph.D. journey, I also had the opportunity to contribute to other research projects. Most notably, I helped create CARMOT (**C**ompiler **a**nd **R**untime **M**emory **O**bservation **T**ool) [66]. CARMOT helps parallel programmers perform Program State Element Characterization (PSEC), which is the analysis of program elements like variables and memory regions to understand how they interact across parallel threads. CARMOT provides source-code level assistance to the programmer for popular programming models. For example, it can recommend the correct OpenMP pragmas to use, or how to classify a C++ smart pointer object. For benchmarks across the PARSEC and NPB suites, CARMOT's OpenMP recommendations matched or exceeded the performance of hand-crafted optimizations.

Most relevant to my thesis work, the need for CARMOT in the first place shows how difficult it can be for parallel programmers to use new abstractions for parallel programming. It emphasizes both the importance of updating existing models that programmers are already comfortable with, and developing higher-level models that make it easier to write correct, performant code.

7.3 Future Work

There are multiple potential directions for further research that stem from this dissertation, most notably the convergence of my ML autotuners and generalized collective algorithms. In my experiments in Chapter 4, I combined these efforts by manually selecting the optimal algorithm parameters using analytic models and empirical evidence. In the future, I envision an autotuner that selects both the proper algorithm *and* its parameters. However, algorithm parameter autotuning drastically increases the autotuner's search space, which risks reaggravating the data collection challenges FACT and ACCLAiM so greatly sought to overcome.

To tune collective parameters without more training data, I propose "hybrid" tuning. FACT and ACCLAiM leverage "allocation-time" tuning, a novel approach where the tuner trains its ML model once a job is scheduled onto the supercomputer but prior to application execution. In hybrid tuning, training also continues "online" while the application is running.

As discussed in Chapter 6, online tuning has been explored in previous work. However, these approaches suffer from significant performance degradation due to search space exploration. When an online tuner wants to understand how an algorithm performs, it must replace the one the application is using. If the new algorithm is slower than the previous one, the algorithm can lose significant performance. It is difficult to infer how one algorithm will perform based on others (i.e., Algorithm 'A' performing better than Algorithm 'B' is unlikely to inform the autotuner about Algorithm 'C'), so these errors must be encountered frequently to eventually find the optimal choice.

Generalized algorithms inherently solve the online tuning selection issue. Their parameters offer fine-grained, transitive tuning, meaning the autotuner can incrementally alter the algorithm's behavior (e.g., increasing/decreasing the parameter value slowly), and make inferences about untested values (e.g., if k=4 performs better than k=2, then k=3 is likely better than 2 but worse than 4).

To implement the hybrid autotuner, I propose tuning algorithm selection at allocation time and parameters online. The allocation-time tuning methodology (e.g., ACCLAiM) will remain unaltered. Then, a new online component will track the most common collective operations. When an operation reaches a certain frequency threshold, the autotuner will apply a simple multiplicative increase/subtractive decrease strategy[1] to the next few invocations of the collective to quickly locate the optimal parameter value.

The other most exciting direction for further research is DMPL. DMPL remains an ongoing project, where the basic functionality exists, but it has not been extensively evaluated. In the future, this project will involve a thorough performance evaluation of the DMPL prototype for both microbenchmarks and real applications (the implemented functionality is sufficient to execute my NPB implementations). This evaluation will identify the key performance bottlenecks in the design. As I described in Section 5.6, there any many avenues to transparently improve DMPL's

[1]For familiar readers, this strategy is the converse of TCP's well-known congestion control mechanism.

performance, extending the impact of this dissertation.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: large-scale machine learning on heterogeneous systems. Software available from tensorflow.org. (2015). `https://www.tensorflow.org/`.

[2] Sarita Adve. 2010. Data races are evil with no exceptions: technical perspective. *Communications of the ACM*, 53, 11, 84–84.

[3] AMD. 2023. 4th gen amd epyc processor architecture. (2023). `https://www.amd.com/en/campaigns/epyc-9004-architecture`.

[4] 2023. Amd rocm platform. (2023). `https://www.amd.com/en/products/software/rocm.html`.

[5] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, Honghui Lu, R. Rajamony, Weimin Yu, and W. Zwaenepoel. 1996. Treadmarks: shared memory computing on networks of workstations. *Computer*, 29, 2, 18–28.

[6] Joe Armstrong. 2010. Erlang. *Communications of the ACM*, 53, 9, 68–75.

[7] Jatin Arora, Sam Westrick, and Umut A. Acar. 2021. Provably space efficient parallel functional programming. In *Proceedings of the 48th Annual ACM Symposium on Principles of Programming Languages (POPL)"*.

[8] Jatin Arora, Sam Westrick, and Umut A Acar. 2021. Provably space-efficient parallel functional programming. *Proceedings of the ACM on Programming Languages*, 5, POPL, 1–33.

[9] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. Mcm-gpu: multi-chip-

module gpus for continued performance scalability. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 320–332.

[10] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. 1989. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11, 4, (October 1989), 598–632.

[11] Thomas J. Ashby, Pedro Díaz, and Marcelo Cintra. 2011. Software-based cache coherence with hardware-assisted selective self-invalidations using bloom filters. *IEEE Transactions on Computers*, 60, 4, 472–483.

[12] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23, 2, 187–198.

[13] Ammar Ahmad Awan, Karthik Vadambacheri Manian, Ching-Hsiang Chu, Hari Subramoni, and Dhabaleswar K Panda. 2019. Optimized large-message broadcast for deep learning workloads: MPI, MPI+ NCCL, or NCCL2? *parallel computing*, 85, 141–152.

[14] John Backus. 1978. The history of fortran i, ii, and iii. *ACM Sigplan Notices*, 13, 8, 165–180.

[15] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The nas parallel benchmarks summary and preliminary results. In *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. IEEE, 158–165.

[16] Prasanna Balaprakash, Romain Egele, Misha Salim, Stefan Wild, Venkatram Vishwanath, Fangfang Xia, Tom Brettin, and Rick Stevens. 2019. Scalable reinforcement-learning-based neural architecture search for cancer deep learning research. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–33.

[17] Prasanna Balaprakash, Robert B. Gramacy, and Stefan M. Wild. 2013. Active-learning-based surrogate models for empirical performance tuning. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, 1–8.

[18] Prasanna Balaprakash, Michael Salim, Thomas D Uram, Venkat Vishwanath, and Stefan M Wild. 2018. DeepHyper: asynchronous hyperparameter search for deep neural networks. In

2018 IEEE 25th international conference on high performance computing (HiPC). IEEE, 42–51.

[19] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. Cacti 7: new tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14, 2, 1–25.

[20] Ayon Basumallik, Seung-Jai Min, and Rudolf Eigenmann. 2007. Programming distributed memory sytems using openmp. In *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1–8.

[21] Michael Bauer, Wonchan Lee, Manolis Papadakis, Marcin Zalewski, and Michael Garland. 2021. Supercomputing in python with legate. *Computing in Science & Engineering*, 23, 4, 73–79.

[22] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[23] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. 2004. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *16th Annual ACM Symposium on Parallel Algorithms and Architectures*, 133–144.

[24] Amanda Bienz, Shreeman Gautam, and Amun Kharel. 2022. A locality-aware Bruck all-gather. In *Proceedings of the 29th European MPI Users' Group Meeting*, 18–26.

[25] Guy E Blelloch. 1992. *NESL: a nested data parallel language*. Carnegie Mellon Univ.

[26] Guy E. Blelloch. 1996. Programming parallel algorithms. *Commun. ACM*, 39, 3, 85–97.

[27] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In (PPoPP '12). New Orleans, Louisiana, USA, 181–192.

[28] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. 1994. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.*, 21, 1, 4–14.

[29] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: an efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37, 1, 55 –69.

[30] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46, 5, (September 1999), 720–748.

[31] Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46, 5, 720–748.

[32] Robert L Bocchino Jr, Vikram S Adve, Danny Dig, Sarita V Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, 97–116.

[33] Robert L Bocchino Jr, Stephen Heumann, Nima Honarmand, Sarita V Adve, Vikram S Adve, Adam Welc, and Tatiana Shpeisman. 2011. Safe nondeterminism in a deterministic-by-default parallel language. *ACM SIGPLAN Notices*, 46, 1, 535–548.

[34] Hans-J Boehm. 2011. How to miscompile programs with" benign" data races. In *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar 11)*.

[35] Michael D Bond, Katherine E Coons, and Kathryn S McKinley. 2010. Pacer: proportional detection of data races. *ACM Sigplan Notices*, 45, 6, 255–268.

[36] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. 2013. Parsec: exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15, 6, 36–45.

[37] J. Bruck, Ching-Tien Ho, S. Kipnis, E. Upfal, and D. Weathersby. 1997. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8, 11, 1143–1156.

[38] Bull. 2021. Bull bullion s16 server. http://www.scaleupservers.com/Bullion-S16-Server.asp. (2021).

[39] Paul Caheny, Lluc Alvarez, Said Derradji, Mateo Valero, Miquel Moretó, and Marc Casas. 2018. Reducing cache coherence traffic with a numa-aware runtime approach. *IEEE Transactions on Parallel and Distributed Systems*, 29, 5, 1174–1187.

[40] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. 2021. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP '21). Association for Computing Machinery, Virtual Event, Republic of Korea, 62–75. ISBN: 9781450382946. https://doi.org/10.1145/3437801.3441620.

[41] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS 2021). Association for Computing Machinery, Virtual, USA, 79–92. ISBN: 9781450383172. https://doi-org.turing.library.northwestern.edu/10.1145/3445814.3446713.

[42] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC).* (November 2011).

[43] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO).*

[44] William W Carlson, Jesse M Draper, David E Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. 1999. Introduction to UPC and language specification. Technical report. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences.

[45] John B. Carter, John K. Bennett, and Willy Zwaenepoel. 1991. Implementation and performance of munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (SOSP '91). Pacific Grove, California, USA, 152–164. ISBN: 0897914473. https://doi-org.turing.library.northwestern.edu/10.1145/121132.121159.

[46] Nicholas P. Carter, Aditya Agrawal, Shekhar Borkar, Romain Cledat, Howard David, Dave Dunning, Joshua Fryman, Ivan Ganev, Roger A. Golliver, Rob Knauerhase, Richard Lethin, Benoit Meister, Asit K. Mishra, Wilfred R. Pinfold, Justin Teller, Josep Torrellas, Nicolas Vasilache, Ganesh Venkatesh, and Jianping Xu. 2013. Runnemede: an architecture for ubiquitous high-performance computing. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 198–209.

[47] Mohamad Chaarawi, Jeffrey M Squyres, Edgar Gabriel, and Saber Feki. 2008. A tool for optimizing runtime parameters of Open MPI. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 210–217.

[48] Milind Chabbi, Shasha Wen, and Xu Liu. 2018. Featherlight on-the-fly false-sharing detection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 152–167.

[49] Manuel MT Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. 2007. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, 10–18.

[50] Bradford L Chamberlain, David Callahan, and Hans P Zima. 2007. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21, 3, 291–312.

[51] Bradford L Chamberlain, Steve Deitz, Mary Beth Hribar, and Wayne Wong. 2015. Chapel. *Programming Models for Parallel Computing*, 129–159.

[52] Matthew Chapman and Gernot Heiser. 2009. Vnuma: a virtual shared-memory multiprocessor. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference* (USENIX'09). USENIX Association, San Diego, California, 2.

[53] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40, 10, 519–538.

[54] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. 1998. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures* (SPAA '98).

[55] H. Cheong and A.V. Veidenbaum. 1988. A cache coherence scheme with fast selective invalidation. In *[1988] The 15th Annual International Symposium on Computer Architecture. Conference Proceedings*, 299–307.

[56] Minsik Cho, Ulrich Finkler, David Kung, and Hillery Hunter. [n. d.] Blueconnect: decomposing all-reduce for deep learning on heterogeneous network hierarchy. *Proceedings of Machine Learning and Systems*, 1, 241–251.

[57] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V Adve, Vikram S Adve, Nicholas P Carter, and Ching-Tsun Chou. 2011. Denovo: rethinking the memory hierarchy for disciplined parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 155–166.

[58] L. Choi and Pen-Chung Yew. 1994. A compiler-directed cache coherence scheme with improved intertask locality. In *Supercomputing '94:Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, 773–782.

[59] Lynn Choi and Pen-Chung Yew. 1996. Compiler and hardware support for cache coherence in large-scale multiprocessors: design considerations and performance study. In *Proceedings of the 23rd annual international symposium on Computer architecture*.

[60] Sudheer Chunduri, Scott Parker, Pavan Balaji, Kevin Harms, and Kalyan Kumaran. 2018. Characterization of MPI usage on a production supercomputer. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 386–400.

[61] NVIDIA Corporation. [n. d.] CUDA 2.0 reference manual. ().

[62] Blas Cuesta, Alberto Ros, María E Gómez, Antonio Robles, and José Duato. 2011. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 93–103.

[63] Leonardo Dagum and Ramesh Menon. 1998. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5, 1, 46–55.

[64] Ervan Darnell, John M. Mellor-Crummey, and Ken Kennedy. 1992. Automatic software cache coherence through vectorization. In *Proceedings of the 6th International Conference on Supercomputing* (ICS '92). Washington, D. C., USA, 129–138. ISBN: 0897914856. https://doi-org.turing.library.northwestern.edu/10.1145/143369.143398.

[65] Abhishek Das, Matt Schuchhardt, Nikos Hardavellas, Gokhan Memik, and Alok Choudhary. 2012. Dynamic directories: a mechanism for reducing on-chip interconnect power in multicores. In *Proceedings of the Conference on Design, Automation Test in Europe*. Dresden, Germany, 479–484.

[66] Enrico Armenio Deiana, Brian Suchy, Michael Wilkins, Brian Homerding, Tommy McMichen, Katarzyna Dunajewski, Peter Dinda, Nikos Hardavellas, and Simone Campanoni. 2023. Program state element characterization. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization* (CGO 2023). Association for Computing Machinery, Montréal, QC, Canada, 199–211. ISBN: 9798400701016. `https://doi.org/10.1145/3579990.3580011`.

[67] Yigit Demir, Yan Pan, Seukwoo Song, Nikos Hardavellas, John Kim, and Gokhan Memik. 2014. Galaxy: a high-performance energy-efficient multi-chip architecture using photonic interconnects. In *Proceedings of the 28th ACM International Conference on Supercomputing* (ICS'14). Munich, Germany.

[68] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. 1974. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of solid-state circuits*, 9, 5, 256–268.

[69] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters*, 21, 02, 173–193.

[70] H Carter Edwards, Christian R Trott, and Daniel Sunderland. 2014. Kokkos: enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing*, 74, 12, 3202–3216.

[71] Bradley Efron and Charles Stein. 1981. The jackknife estimate of variance. *The Annals of Statistics*, 586–596.

[72] Tarek El-Ghazawi and François Cantonnet. 2002. Upc performance and potential: a npb experimental study. In *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE, 17–17.

[73] Marco Elver and Vijay Nagarajan. 2014. Tso-cc: consistency directed cache coherence for tso. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 165–176.

[74] Natalie Enright Jerger, Li-Shiuan Peh, and Mikko Lipasti. 2008. Virtual tree coherence: leveraging regions and in-network multicast trees for scalable cache coherence. In *International Symposium on Microarchitecture*, 35–46.

[75] Jeff Epstein, Andrew P Black, and Simon Peyton-Jones. 2011. Towards haskell in the cloud. In *Proceedings of the 4th ACM symposium on Haskell*, 118–129.

[76] Ericsson. 2017. Time for memory disaggregation? https://www.ericsson.com/en/blog/2017/5/time-for-memory-disaggregation. (2017).

[77] Graham E Fagg, Jelena Pjesivac-Grbovic, George Bosilca, Thara Angskun, J Dongarra, and Emmanuel Jeannot. 2006. Flexible collective communication tuning architecture applied to Open MPI. In *Euro PVM/MPI*.

[78] Ke Fan, Thomas Gilray, Valerio Pascucci, Xuan Huang, Kristopher Micinski, and Sidharth Kumar. 2022. Optimizing the Bruck algorithm for non-uniform all-to-all communication. In (HPDC '22). Association for Computing Machinery, Minneapolis, MN, USA, 172–184. ISBN: 9781450391993. `https://doi.org/10.1145/3502181.3531468`.

[79] Ahmad Faraj, Xin Yuan, and David Lowenthal. 2006. STAR-MPI: self tuned adaptive routines for MPI collective operations. In (January 2006), 199–208.

[80] Guangnan Feng, Dezun Dong, and Yutong Lu. 2022. Optimized MPI collective algorithms for dragonfly topology. In *Proceedings of the 36th ACM International Conference on Supercomputing*, 1–11.

[81] Mingdong Feng and Charles E. Leiserson. 1999. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems*, 32, 3, 301–326.

[82] Sevin Fide and Stephen Jenks. 2008. Proactive use of shared l3 caches to enhance cache communications in multi-core processors. *IEEE Computer Architecture Letters*, 7, 2, 57–60.

[83] Jeremy T. Fineman. 2005. *Provably Good Race Detection That Runs in Parallel*. Master's thesis. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA.

[84] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2010. Implicitly threaded parallelism in manticore. *Journal of functional programming*, 20, 5-6, 537–576.

[85] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2011. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming*, 20, 5-6, 1–40.

[86] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. 2009. Reducers and other Cilk++ hyperobjects. In *21st Annual ACM Symposium on Parallelism in Algorithms and Architectures*, 79–90.

[87] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 212–223.

[88] J Davison de St Germain, John McCorquodale, Steven G Parker, and Christopher R Johnson. 2000. Uintah: a massively parallel problem solving environment. In *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*. IEEE, 33–41.

[89] Yifan Gong, Bingsheng He, and Jianlong Zhong. 2015. Network performance aware MPI collective communication operations in the cloud. *IEEE Transactions on Parallel and Distributed Systems*, 26, 11, 3079–3089.

[90] Sergei Gorlatch, Christoph Wedler, and Christian Lengauer. 1999. Optimization rules for programming with collective operations. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999*. IEEE, 492–499.

[91] Richard L Graham, Galen M Shipman, Brian W Barrett, Ralph H Castain, George Bosilca, and Andrew Lumsdaine. 2006. Open MPI: a high-performance, heterogeneous MPI. In *2006 IEEE International Conference on Cluster Computing*. IEEE, 1–9.

[92] William Gropp. 2002. MPICH2: a new start for MPI implementations. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 7–7.

[93] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with infiniswap. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 649–667.

[94] Adrien Guatto, Sam Westrick, Ram Raghunathan, Umut Acar, and Matthew Fluet. 2018. Hierarchical memory management for mutable state. *ACM SIGPLAN Notices*, 53, 1, 81–93.

[95] Adrien Guatto, Sam Westrick, Ram Raghunathan, Umut A. Acar, and Matthew Fluet. 2018. Hierarchical memory management for mutable state. In *Proceedings of the 23rd ACM SIG-*

PLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018, 81–93.

[96] Pouya Haghi, Anqi Guo, Qingqing Xiong, Rushi Patel, Chen Yang, Tong Geng, Justin T Broaddus, Ryan Marshall, Anthony Skjellum, and Martin C Herbordt. 2020. FPGAs in the network and novel communicator support accelerate MPI collectives. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–10.

[97] Robert H. Halstead Jr. 1984. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming* (LFP '84). ACM, Austin, Texas, United States, 9–17.

[98] Kevin Hammond. 2011. Why parallel functional programming matters: panel statement. In *Reliable Software Technologies - Ada-Europe 2011 - 16th Ada-Europe International Conference on Reliable Software Technologies, Edinburgh, UK, June 20-24, 2011. Proceedings*, 201–205.

[99] Kevin Hammond. 2011. Why parallel functional programming matters: panel statement. In *International Conference on Reliable Software Technologies*. Springer, 201–205.

[100] Tim Harris, James Larus, and Ravi Rajwar. 2010. Transactional memory. *Synthesis Lectures on Computer Architecture*, 5, 1, 1–263.

[101] Khalid Hasanov and Alexey Lastovetsky. 2017. Hierarchical redesign of classic MPI reduction algorithms. *The Journal of Supercomputing*, 73, 2, 713–725.

[102] Maurice Herlihy and J Eliot B Moss. 1993. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, 289–300.

[103] Hewlett Packard Enterprise. 2021. HPE integrity mc990 x server. (2021). `https://www.hpe.com/psnow/doc/PSN1008798952USEN.pdf`.

[104] Roger W Hockney. 1994. The communication challenge for MPP: intel Paragon and Meiko CS-2. *Parallel computing*, 20, 3, 389–398.

[105] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. 2006. Fast barrier synchronization for InfiniBand. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, 7 pp.–.

[106] Richard D Hornung and Jeffrey A Keasler. 2014. The raja portability layer: overview and status.

[107] Derek R. Hower. 2012. *Acoherent Shared Memory*. PhD thesis. USA. ISBN: 9781267539397. AAI3522117.

[108] C.C. Hu, M.F. Chen, W.C. Chiou, and Doug C.H. Yu. 2019. 3d multi-chip integration with system on integrated chips (soic™). In *2019 Symposium on VLSI Technology*, T20–T21.

[109] Sascha Hunold, Abhinav Bhatele, George Bosilca, and Peter Knees. 2020. Predicting MPI collective communication performance using machine learning. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 259–269.

[110] Sascha Hunold and Alexandra Carpen-Amarie. 2016. Reproducible MPI benchmarking is still not as easy as you think. *IEEE Transactions on Parallel and Distributed Systems*, 27, 12, 3617–3630.

[111] Sascha Hunold and Alexandra Carpen-Amarie. 2018. Algorithm selection of MPI collectives using machine learning techniques. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 45–50.

[112] Sascha Hunold and Alexandra Carpen-Amarie. 2018. Autotuning MPI collectives using performance guidelines. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, 64–74.

[113] IBM. 2018. Advancing cloud with memory disaggregation. (2018). `https://www.ibm.com/blogs/research/2018/01/advancing-cloud-memory-disaggregation/`.

[114] Intel. [n. d.] Intel application performance snapshot (aps). (). `https://software.intel.com/sites/products/snapshots/applicationsnapshot`.

[115] [n. d.] Intel cilk plus. (). `https://www.cilkplus.org/`.

[116] [n. d.] Intel tbb. (). `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html`.

[117] Florin Isaila, Prasanna Balaprakash, Stefan M Wild, Dries Kimpe, Rob Latham, Rob Ross, and Paul Hovland. 2015. Collective I/O tuning using analytical and machine learning models. In *2015 IEEE International Conference on Cluster Computing*. IEEE, 128–137.

[118] Subramanian S. Iyer. 2016. Heterogeneous integration for performance and scaling. *IEEE Transactions on Components, Packaging and Manufacturing Technology*, 6, 7, 973–982.

[119] Alexandra Jimborean, Jonatan Waern, Per Ekemark, Stefanos Kaxiras, and Alberto Ros. 2017. Automatic detection of extended data-race-free regions. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 14–26.

[120] Haoqiang Jin, Robert Hood, and Piyush Mehrotra. 2009. A practical study of upc using the nas parallel benchmarks. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, 1–7.

[121] Robert L. Bocchino Jr., Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. 2011. Safe nondeterminism in a deterministic-by-default parallel language. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. Thomas Ball and Mooly Sagiv, editors. ACM, 535–548.

[122] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. 2009. Parallex an advanced parallel execution model for scaling-impaired applications. In *2009 International Conference on Parallel Processing Workshops*. IEEE, 394–401.

[123] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. Hpx: a task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, 1–11.

[124] Laxmikant V Kale and Sanjeev Krishnan. 1993. Charm++ a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, 91–108.

[125] Ajaykumar Kannan, Natalie Enright Jerger, and Gabriel Loh. 2015. Enabling interposer-based disintegration of multi-core processors. In *Proceedings of the International Symposium on Microarchitecture*.

[126] Pete Keleher, Alan L Cox, and Willy Zwaenepoel. 1992. Lazy release consistency for software distributed shared memory. *ACM SIGARCH Computer Architecture News*, 20, 2, 13–21.

[127] Ken Kennedy, Charles Koelbel, and Hans Zima. 2007. The rise and fall of high performance fortran: an historical object lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 7–1.

[128] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. 2008. Technology-driven, highly-scalable dragonfly topology. *ACM SIGARCH Computer Architecture News*, 36, 3, 77–88.

[129] Wooil Kim, Sanket Tavarageri, P. Sadayappan, and Josep Torrellas. 2016. Architecting and programming a hardware-incoherent multiprocessor cache hierarchy. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 555–565.

[130] Leonidas Kontothanassis, Galen Hunt, Robert Stets, Nikolaos Hardavellas, Michał Cierniak, Srinivasan Parthasarathy, Wagner Meira Jr., Sandhya Dwarkadas, and Michael Scott. 1997. Vm-based shared memory on low-latency, remote-memory-access networks. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (ISCA'97). Denver, Colorado, United States, 157–169. ISBN: 0-89791-901-7. http://doi.acm.org.turing.library.northwestern.edu/10.1145/264107.264163.

[131] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S. Milojicic, and Gustavo Alonso. 2021. Farview: disaggregated memory with operator off-loading for database engines. *CoRR*, abs/2106.07102. arXiv: 2106.07102. https://arxiv.org/abs/2106.07102.

[132] Lindsey Kuper and Ryan R Newton. 2013. Lvars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*. ACM, 71–84.

[133] Lindsey Kuper, Aaron Todd, Sam Tobin-Hochstadt, and Ryan R. Newton. 2014. Taming the parallel effect zoo: extensible deterministic parallelism with lvish. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '14). Edinburgh, United Kingdom, 2–14. ISBN: 978-1-4503-2784-8.

[134] Youngeun Kwon and Minsoo Rhu. 2019. A disaggregated memory system for deep learning. *IEEE Micro*, 39, 5, 82–90.

[135] [n. d.] Legion programming lanugage. (). https://legion.stanford.edu/.

[136] David D Lewis and William A Gale. 1994. A sequential algorithm for training text classifiers. In *SIGIR'94*. Springer, 3–12.

[137] Kai Li and Paul Hudak. 1989. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7, 4, (November 1989), 321–359. `https://doi-org.turing.library.northwestern.edu/10.1145/75104.75105`.

[138] Peng Li, Simon Marlow, Simon L. Peyton Jones, and Andrew P. Tolmach. 2007. Lightweight concurrency primitives for GHC. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, 107–118.

[139] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*.

[140] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (ISCA '09). Austin, TX, USA, 267–278. ISBN: 9781605585260. `https://doi.org/10.1145/1555754.1555789`.

[141] David B Loveman. 1993. High performance fortran. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1, 1, 25–42.

[142] Xi Luo, Wei Wu, George Bosilca, Yu Pei, Qinglei Cao, Thananon Patinyasakdikul, Dong Zhong, and Jack Dongarra. 2020. HAN: a hierarchical autotuned collective communication framework. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 23–34.

[143] Robert Lyerly, Sang-Hoon Kim, and Binoy Ravindran. 2019. Libmpnode: an openmp runtime for parallel processing across incoherent domains. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, 81–90.

[144] Robert Lyerly, Changwoo Min, Christopher J Rossbach, and Binoy Ravindran. 2020. An openmp runtime for transparent work sharing across cache-incoherent heterogeneous nodes. In *Proceedings of the 21st International Middleware Conference*, 415–429.

[145] David MacQueen, Robert Harper, and John Reppy. 2020. The history of standard ml. *Proceedings of the ACM on Programming Languages*, 4, HOPL, 1–100.

[146] Timothy G Mattson, Romain Cledat, Vincent Cavé, Vivek Sarkar, Zoran Budimlić, Sanjay Chatterjee, Josh Fryman, Ivan Ganev, Robin Knauerhase, Min Lee, et al. 2016. The open community runtime: a runtime system for extreme scale computing. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[147] John Mellor-Crummey. 1991. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. IEEE, 24–33.

[148] John Mellor-Crummey. 1991. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing'91*, 24–33.

[149] Qingyu Meng, Alan Humphrey, and Martin Berzins. 2012. The uintah framework: a unified heterogeneous task scheduling and runtime system. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 2441–2448.

[150] [n. d.] Mlton. (). `www.mlton.org`.

[151] Bashir Mohammed, Irfan Awan, Hassan Ugail, and Muhammad Younas. 2019. Failure prediction using machine learning in a virtualised HPC system and application. *Cluster Computing*, 22, 2, 471–485.

[152] Mohammad Alaul Haque Monil. 2021. Dynamic adaptation techniques and opportunities to improve hpc runtimes.

[153] Moor Insights and Strategy. 2013. Intel's disaggregated server rack. (2013). `https://moorinsightsstrategy.com/wp-content/uploads/2013/08/Intels-Disagggregated-Server-Rack-by-Moor-Insights-Strategy.pdf`.

[154] [n. d.] MPICH. (). `https://www.mpich.org`.

[155] [n. d.] MPL compiler. `https://github.com/mpllang/mpl`. ().

[156] Samuel Naffziger, Noah Beck, Thomas Burd, Kevin Lepak, Gabriel H. Loh, Mahesh Subramony, and Sean White. 2021. Pioneering chiplet technology and design for the amd epyc™ and ryzen™ processor families : industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 57–70.

[157] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2020. *A Primer on Memory Consistency and Cache Coherence: Second Edition*.

[158] 2022. Nas parallel benchmarks. (2022). `https://www.nas.nasa.gov/software/npb.html`.

[159] [n. d.] Nas parallel benchmarks - parallel ml implementation. (). `https://github.com/mjwilkins18/nas-mpl`.

[160] [n. d.] Nas parallel benchmarks 3.0 - unofficial openmp c version. (). `https://github.com/benchmark-subsetting/NPB3.0-omp-C`.

[161] Robert Netzer and Barton P Miller. 1989. Detecting data races in parallel program executions. Technical report. University of Wisconsin-Madison Department of Computer Sciences.

[162] Robert W Numrich and John Reid. 1998. Co-array fortran for parallel programming. In *ACM Sigplan Fortran Forum* number 2. Volume 17. ACM New York, NY, USA, 1–31.

[163] Emin Nuriyev and Alexey Lastovetsky. 2020. Accurate runtime selection of optimal MPI collective algorithms using analytical performance modelling. *arXiv preprint arXiv:2004.11062*.

[164] M. F. P. O'Boyle, R. W. Ford, and E. A. Stohr. 2003. Towards general and exact distributed invalidation. *J. Parallel Distrib. Comput.*, 63, 11, (November 2003), 1123–1137. `https://doi.org/10.1016/j.jpdc.2003.07.007`.

[165] 2018. *OpenMP Application Programming Interface, Version 5.0.* Accessed in July 2018. (November 2018).

[166] [n. d.] OSU micro-benchmarks. (). `https://mvapich.cse.ohio-state.edu/benchmarks/`.

[167] Susan Owicki and Anant Agarwal. 1989. Evaluating the performance of software cache coherence. *ACM SIGARCH Computer Architecture News*, 17, 2, 230–242.

[168] Dhabaleswar K Panda, Karen Tomko, Karl Schulz, and Amitava Majumdar. 2013. The MVAPICH project: evolution and sustainability of an open source production quality MPI library for HPC. In *Workshop on Sustainable Software for Science: Practice and Experiences, held in conjunction with Int'l Conference on Supercomputing (WSSPE)*.

[169] [n. d.] Parallel runtime scheduling and execution controller. (). `https://icl.utk.edu/parsec/index.html`.

[170] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: an imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[171] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69, 2, 117–124.

[172] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: machine learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.

[173] Simone Pellegrini, Jie Wang, Thomas Fahringer, and Hans Moritsch. 2009. Optimizing MPI runtime parameter settings by using machine learning. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 196–206.

[174] Ivy Peng, Roger Pearce, and Maya Gokhale. 2020. On the memory underutilization: exploring disaggregated memory on hpc systems. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 183–190.

[175] Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. 2008. Harnessing the multicores: nested data parallelism in Haskell. In *FSTTCS*, 383–414.

[176] Jelena Pjesivac-Grbovic. 2007. Towards automatic and adaptive optimizations of MPI collective operations.

[177] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E Fagg, Edgar Gabriel, and Jack J Dongarra. 2007. Performance analysis of MPI collective operations. *Cluster Computing*, 10, 2, 127–143.

[178] Jelena Pješivac-Grbović, George Bosilca, Graham E Fagg, Thara Angskun, and Jack J Dongarra. 2007. MPI collective algorithm selection and quadtree encoding. *Parallel Computing*, 33, 9, 613–623.

[179] Judit Planas, Rosa M Badia, Eduard Ayguadé, and Jesus Labarta. 2009. Hierarchical task-based programming with starss. *The International Journal of High Performance Computing Applications*, 23, 3, 284–299.

[180] 2023. Polaris user guide. (2023). `https://docs.alcf.anl.gov/polaris/getting-started/`.

[181] [n. d.] Problem sizes and parameters in nas parallel benchmarks. (). `https://www.nas.nasa.gov/software/npb_problem_sizes.html`.

[182] Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. 2016. Hierarchical memory management for parallel programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (ICFP 2016). ACM, Nara, Japan, 392–406.

[183] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2010. Efficient data race detection for async-finish parallelism. In *Runtime Verification*. Lecture Notes in Computer Science. Volume 6418. Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors. Springer Berlin / Heidelberg, 368–383. ISBN: 978-3-642-16611-2.

[184] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '12), 531–542.

[185] John Reid. 2018. The new features of fortran 2018. In *ACM SIGPLAN fortran forum* number 1. Volume 37. ACM New York, NY, USA, 5–43.

[186] Yuxin Ren, Gabriel Parmer, and Dejan Milojicic. 2020. Ch'i: scaling microkernel capabilities in cache-incoherent systems. In *2020 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, 12–21.

[187] Ruyman Reyes and Victor Lomüller. 2016. Sycl: single-source c++ accelerator programming. In *Parallel Computing: On the Road to Exascale*. IOS Press, 673–682.

[188] Arch D Robison. 2013. Composable parallel patterns with intel cilk plus. *Computing in Science & Engineering*, 15, 02, 66–71.

[189] Alberto Ros, Mahdad Davari, and Stefanos Kaxiras. 2015. Hierarchical private/shared classification: the key to simple and efficient coherence for clustered cache hierarchies. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 186–197.

[190] Alberto Ros and Alexandra Jimborean. 2015. A dual-consistency cache coherence protocol. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1119–1128.

[191] Alberto Ros and Stefanos Kaxiras. 2012. Complexity-effective multicore coherence. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 241–251.

[192] Martin Ruefenacht, Mark Bull, and Stephen Booth. 2017. Generalisation of recursive doubling for allreduce: now with simulation. *Parallel Computing*, 69, 24–44. `https://www.sciencedirect.com/science/article/pii/S0167819117301199`.

[193] Amit Ruhela, Bharath Ramesh, Sourav Chakraborty, Hari Subramoni, Jahanzeb Hashmi, and Dhabaleswar Panda. 2019. Leveraging network-level parallelism with multiple process-endpoints for mpi broadcast. In *2019 IEEE/ACM Third Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM)*. IEEE, 34–41.

[194] Karl Rupp. 2018. 42 years of microprocessor trend data. (2018). `https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/`.

[195] Paul Sack and William Gropp. 2012. Faster topology-aware collective algorithms through non-minimal communication. *ACM SIGPLAN Notices*, 47, 8, 45–54.

[196] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. 1996. Shasta: a low overhead, software-only approach for supporting fine-grain shared memory. *SIGPLAN Not.*, 31, 9, (September 1996), 174–185. `https://doi.org/10.1145/248209.237179`.

[197] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. 1994. Fine-grain access control for distributed shared memory. In (ASPLOS VI). San Jose, California, USA, 297–306. ISBN: 0897916603. `https://doi-org.turing.library.northwestern.edu/10.1145/195473.195575`.

[198] Matthew Schuchhardt, Abhishek Das, Nikos Hardavellas, Gokhan Memik, and Alok Choudhary. 2013. The impact of dynamic directories on multicore interconnects. *IEEE Computer*, 46, 10, 32–39.

[199] Burr Settles. 2009. Active learning literature survey. Technical Report, University of Wisconsin-Madison, Department of Computer Sciences. (2009).

[200] Hongzhang Shan, Filip Blagojević, Seung-Jai Min, Paul Hargrove, Haoqiang Jin, Karl Fuerlinger, Alice Koniges, and Nicholas J Wright. 2010. A programming model performance study using the nas parallel benchmarks. *Scientific Programming*, 18, 3-4, 153–167.

[201] Nir Shavit and Dan Touitou. 1997. Software transactional memory. *Distributed Computing*, 10, 2, 99–116.

[202] Sergei Shudler, Yannick Berens, Alexandru Calotoiu, Torsten Hoefler, Alexandre Strube, and Felix Wolf. 2019. Engineering algorithms for scalability through continuous validation of performance expectations. *IEEE Transactions on Parallel and Distributed Systems*, 30, 8, 1768–1785.

[203] Hanan Shukur, Subhi Zeebaree, Rizgar Zebari, Omar Ahmed, Lailan Haji, and Dildar Abdulqader. 2020. Cache coherence protocols in distributed systems. *Journal of Applied Science and Technology Trends*, 1, 3, 92–97.

[204] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement: the problem based benchmark suite. In (SPAA '12). Pittsburgh, Pennsylvania, USA, 68–70. ISBN: 9781450312134. `https://doi.org/10.1145/2312005.2312018`.

[205] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting parallelism onto ocaml. *Proc. ACM Program. Lang.*, 4, ICFP, 113:1–113:30.

[206] KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting parallelism onto ocaml. *arXiv preprint arXiv:2004.11663*.

[207] KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto ocaml. In *Proceedings of the 42nd ACM*

SIGPLAN International Conference on Programming Language Design and Implementation, 206–221.

[208] Daniel Spoonhower. 2009. *Scheduling Deterministic Parallel Programs*. PhD thesis. Carnegie Mellon University, (May 2009). https://www.cs.cmu.edu/~rwh/theses/spoonhower.pdf.

[209] Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srinivasan Parthasarathy, and Michael Scott. 1997. Cashmere-2l: software coherent shared memory on a clustered remote-write network. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (SOSP '97). Association for Computing Machinery, Saint Malo, France, 170–183. ISBN: 0897919165. https://doi-org.turing.library.northwestern.edu/10.1145/268998.266675.

[210] Rabin Sugumar, Mehul Shah, and Ricardo Ramirez. 2021. Marvell thunderx3: next-generation arm-based server processor. *IEEE Micro*, 41, 2, 15–21.

[211] Nawrin Sultana, Martin Rüfenacht, Anthony Skjellum, Purushotham Bangalore, Ignacio Laguna, and Kathryn Mohror. 2021. Understanding the use of message passing interface in exascale proxy applications. *Concurrency and Computation: Practice and Experience*, 33, 14, e5901.

[212] Hyojin Sung, Rakesh Komuravelli, and Sarita V. Adve. 2013. Denovond: efficient hardware support for disciplined non-determinism. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '13). Houston, Texas, USA, 13–26. ISBN: 9781450318709. https://doi-org.turing.library.northwestern.edu/10.1145/2451116.2451119.

[213] Igor Tartalja and Veljko Milutinovic. 1996. The cache coherence problem in shared memory multiprocessors: software solutions. In *XVI International Symposium on Nuclear Electronics and VI International School on Automation and Computing in Nuclear Physics and Astrophysics*, 131.

[214] Sanket Tavarageri, Wooil Kim, Josep Torrellas, and P Sadayappan. 2016. Compiler support for software cache coherence. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE, 341–350.

[215] Rajeev Thakur and William D. Gropp. 2003. Improving the performance of collective operations in MPICH. In *Recent Advances in Parallel Virtual Machine and Message Passing*

Interface. Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors. Springer Berlin Heidelberg, 257–267. ISBN: 978-3-540-39924-7.

[216] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Spring 2005. Optimization of collective communication operations in MPICH. *International Journal of High-Performance Computing Applications*, 19, 1, 49–66.

[217] [n. d.] The chapel parallel programming language. (). `https://chapel-lang.org/`.

[218] [n. d.] The ompss programming model. (). `https://pm.bsc.es/ompss`.

[219] [n. d.] The openacc application programming interface. (). `https://www.cs.uoregon.edu/Reports/AREA-202103-Monil.pdf`.

[220] [n. d.] The opencl specification. (). `https://www.khronos.org/registry/OpenCL/specs/2.2/html/OpenCL_API.html`.

[221] [n. d.] The qthreads library. (). `https://www.sandia.gov/qthreads/`.

[222] [n. d.] The uintah software. (). `http://www.uintah.utah.edu/`.

[223] Josep Torrellas, HS Lam, and John L. Hennessy. 1994. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43, 6, 651–663.

[224] Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. 2016. Provably good and practically efficient parallel race detection for fork-join programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, 83–94.

[225] Sathish S Vadhiyar, Graham E Fagg, and Jack Dongarra. 2000. Automatically tuned collective communications. In *SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE.

[226] Stefan Wager, Trevor Hastie, and Bradley Efron. 2014. Confidence intervals for random forests: the jackknife and the infinitesimal jackknife. *The Journal of Machine Learning Research*, 15, 1, 1625–1651.

[227] Chen Wang, Marc Snir, and Kathryn Mohror. 2020. High performance computing application I/O traces. in Lawrence Livermore National Laboratory (LLNL) Open Data Initiative. (2020). `http://library.ucsd.edu/dc/object/bb95276921`.

[228] Sam Westrick, Jatin Arora, and Umut A. Acar. 2022. Entanglement detection with near-zero cost. *Proc. ACM Program. Lang.*, 6, ICFP, 679–710. `https://doi.org/10.1145/3547646`.

[229] Sam Westrick, Mike Rainey, Daniel Anderson, and Guy E Blelloch. 2022. Parallel block-delayed sequences. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 61–75.

[230] Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A Acar. 2019. Disentanglement in nested-parallel programs. *Proceedings of the ACM on Programming Languages*, 4, POPL, 1–32.

[231] Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. 2020. Disentanglement in nested-parallel programs. In *Proceedings of the 47th Annual ACM Symposium on Principles of Programming Languages (POPL)”*.

[232] Kyle B Wheeler, Richard C Murphy, and Douglas Thain. 2008. Qthreads: an api for programming with millions of lightweight threads. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–8.

[233] Michael Wilkins, Yanfei Guo, Rajeev Thakur, Peter Dinda, and Nikos Hardavellas. 2022. ACCLAiM: advancing the practicality of MPI collective communication autotuning using machine learning. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 161–171.

[234] Michael Wilkins, Yanfei Guo, Rajeev Thakur, Nikos Hardavellas, Peter Dinda, and Min Si. 2021. A FACT-based approach: making machine learning collective autotuning feasible on exascale systems. In *2021 Workshop on Exascale MPI (ExaMPI)*. IEEE, 36–45.

[235] Michael Wilkins, Hanming Wang, Peizhi Liu, Bangyen Pham, Yanfei Guo, Rajeev Thakur, Peter Dinda, and Nikos Hardavellas. 2023. Generalized collective algorithms for the exascale era. In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, 60–71.

[236] Michael Wilkins, Garrett Weil, Luke Arnold, Nikos Hardavellas, and Peter Dinda. 2023. Evaluating functional memory-managed parallel languages for hpc using the nas parallel benchmarks. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 413–422.

[237] Michael Wilkins, Sam Westrick, Vijay Kandiah, Alex Bernat, Brian Suchy, Enrico Armenio Deiana, Simone Campanoni, Umut Acar, Peter Dinda, and Nikos Hardavellas. 2022. Artifact for ”WARDen: Specializing Cache Coherence for High-Level Parallel Languages”. (November 2022). `https://doi.org/10.5281/zenodo.7374334`.

[238] Michael Wilkins, Sam Westrick, Vijay Kandiah, Alex Bernat, Brian Suchy, Enrico Armenio Deiana, Simone Campanoni, Umut A. Acar, Peter Dinda, and Nikos Hardavells. 2023. Warden: specializing cache coherence for high-level parallel languages. In *IEEE/ACM International Symposium on Code Generation and Optimization*, To Appear.

[239] [n. d.] X10: performance and productivity at scale. (). `http://x10-lang.org/`.

[240] Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao, and Bing Xie. 2020. RLScheduler: an automated HPC batch job scheduler using reinforcement learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC20)*. IEEE Press.

[241] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. 2014. Upc++: a pgas extension for c++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 1105–1114.

[242] Alexandros Nikolaos Ziogas, Timo Schneider, Tal Ben-Nun, Alexandru Calotoiu, Tiziano De Matteis, Johannes de Fine Licht, Luca Lavarini, and Torsten Hoefler. 2021. Productivity, portability, performance: data-centric python. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–13.