Fall 12-1-2018

Adaptive Parallelism for Coupled, Multithreaded Message-Passing Programs

Samuel K. Gutiérrez

Samuel Keith Gutiérrez

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

Professor Dorian C. Arnold, Chair

Professor Patrick G. Bridges

Professor Darko Stefanovic

Professor Alexander S. Aiken

Patrick S. McCormick

Adaptive Parallelism for Coupled, Multithreaded Message-Passing Programs

by

Samuel Keith Gutiérrez

B.S., Computer Science, New Mexico Highlands University, 2006
M.S., Computer Science, University of New Mexico, 2009

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December 2018

Dedication

To my beloved family

"A Dios rogando y con el martillo dando."

Unknown

Acknowledgments

Words cannot adequately express my feelings of gratitude for the people that made this possible. Please know that this accomplishment is as much yours as it is mine. We made it—persevered. With all my being, thank you.

I was incredibly fortunate to have a dedicated academic advisor who was demanding, yet patient—making me a better writer and researcher along the way. Dorian, thank you for your friendship and mentorship throughout these years. I owe a thank-you to the members of my committee, Alex Aiken, Dorian Arnold, Patrick Bridges, Patrick McCormick, and Darko Stefanovic, for their helpful comments. To the members of the Scalable Systems Lab at UNM, thank you for your honest feedback and encouragement, but more importantly your fellowship. Special thanks to David DeBonis, Matthew Dosanjh, Noah Evans, Kurt Ferreira, Aaron Gonzales, Taylor Groves, Nathan Hjelm, Dewan Ibtesham, Edgar León, Scott Levy, Whit Schonbein, and Patrick Widener.

Many supportive and understanding individuals undoubtedly enabled my going back to school while also working full-time at LANL. To my management, both past and present, John Cerutti, Linn Collins, David Daniel, Ed Dendy, Gary Grider, Jeff Johnson, Christoph Junghans, Mike Lang, Patrick McCormick, David Montoya, Randal Rheinheimer, and Galen Shipman, thank you for allowing my research at work to align nicely with my academic endeavors. To my colleagues, thank you for being my extended academic family. Special thanks to Randy Baker, Zach Baker, Ben Bergen, Bob Bird, Jon Dahl, Kei Davis, Daniel Holladay, Erika Maestas, Patrick McCormick, Scott Pakin, Robert Robey, Ben Santos, Justin Tripp, and Joe Zerr.

To all my friends, thank you for reminding me that the world is full of fun, exciting people. To my uncles, aunts, and cousins, thank you for letting a curly haired kid be a part of such a loving and supportive group. I thank especially my uncle Frank, aunt Isabel, Krystal, and Brittany. To Max, Beth, Christina, Gerald, and Max, thank you for accepting me so completely into your family and letting me be a part of your lives.

I was blessed to have an upbringing full of love, encouragement, support, happiness, hard work, and discipline. Mom, thank you. You worked so very hard, often taking on multiple jobs so that I could have and do things that I found interesting. You always put me first and made sure your *hito* was loved and cared for no matter what. Grandma and grandpo, thank you for raising me as one of your own. Some of my most cherished childhood memories are of us at the ranch.

To Jess—my love, my rock, my friend and confidant, the mother of our son. Thank you for being there, through thick and thin. I know I'm not the easiest person to get along with when I'm stressed, so thank you for your steadfast love and support. It is because of your commitment that I saw this thing through—we did it. To Andres—my son, my pride and joy. Thank you for bringing us so much happiness. I look forward to our adventures together.

Sponsorship

Work supported by the Advanced Simulation and Computing program of the U.S. Department of Energy's NNSA and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Los Alamos National Laboratory is managed and operated by Los Alamos National Security, LLC (LANS), under contract number DE-AC52–06NA25396 for the Department of Energy's National Nuclear Security Administration (NNSA).

Adaptive Parallelism for Coupled, Multithreaded Message-Passing Programs

by

Samuel Keith Gutiérrez

B.S., Computer Science, New Mexico Highlands University, 2006
M.S., Computer Science, University of New Mexico, 2009

Ph.D., Computer Science, University of New Mexico, 2018

Abstract

Hybrid parallel programming models that combine message passing (MP) and shared-memory multithreading (MT) are becoming more popular, especially with applications requiring higher degrees of parallelism and scalability. Consequently, *coupled* parallel programs, those built via the integration of independently developed and optimized software libraries linked into a single application, increasingly comprise message-passing libraries with differing preferred degrees of threading, resulting in *thread-level heterogeneity*. Retroactively matching threading levels between independently developed and maintained libraries is difficult, and the challenge is exacerbated because contemporary middleware services provide only static scheduling policies over entire program executions, necessitating suboptimal, *over-subscribed* or *under-subscribed*, configurations. In coupled applications, a poorly configured component can lead to overall poor application performance, suboptimal resource utilization, and increased time-to-solution. So it is critical that each library executes

in a manner consistent with its design and tuning for a particular system architecture and workload. Therefore, there is a need for techniques that address dynamic, conflicting configurations in coupled multithreaded message-passing (MT-MP) programs. Our thesis is that we can achieve significant performance improvements over static under-subscribed approaches through reconfigurable execution environments that consider compute phase parallelization strategies along with both hardware and software characteristics.

In this work, we present new ways to structure, execute, and analyze coupled MT-MP programs. Our study begins with an examination of contemporary approaches used to accommodate thread-level heterogeneity in coupled MT-MP programs. Here we identify potential inefficiencies in how these programs are structured and executed in the high-performance computing domain. We then present and evaluate a novel approach for accommodating thread-level heterogeneity. Our approach enables full utilization of all available compute resources throughout an application's execution by providing programmable facilities with modest overheads to dynamically reconfigure runtime environments for compute phases with differing threading factors and affinities. Our performance results show that for a majority of the tested scientific workloads our approach and corresponding open-source reference implementation render speedups greater than 50 % over the static under-subscribed baseline.

Motivated by our examination of reconfigurable execution environments and their memory overhead, we also study *the memory attribution problem*: the inability to predict or evaluate during runtime where the available memory is used across the software stack comprising the application, reusable software libraries, and supporting runtime infrastructure. Specifically, dynamic adaptation requires runtime intervention, which by its nature introduces additional runtime and memory overhead. To better understand the latter, we propose and evaluate a new way to quantify component-level memory usage from unmodified binaries dynamically linked to a

message-passing communication library. Our experimental results show that our approach and corresponding implementation accurately measure memory resource usage as a function of time, scale, communication workload, and software or hardware system architecture, clearly distinguishing between application and communication library usage at a per-process level.

Contents

List of Figures

List of Tables

List of Tables

Glossary

ANSI	American National Standards Institute
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
CPU	Central Processing Unit
DRAM	Dynamic Random-Access Memory
ECC	Error-Correcting Code
FLOPS	Floating point operations per second
GPU	Graphics Processing Unit
HPC	High-Performance Computing
HPL	High-Performance LINPACK: a parallel version of LINPACK.
I/O	Input/Output
ISO	International Organization for Standardization
LINPACK	A performance benchmark for computers that analyzes and solves dense systems of linear equations [35].

LSB-0	Least significant bit zero
MP	Message Passing
MPS	Messages Per Second
MT-MP	Multithreaded Message Passing
NIC	Network Interface Controller
Node	A component in a system of networked machines—a computer.
NUMA	Non-Uniform Memory Access
OS	Operating System
PE	Processing Element: a core or hardware thread.
POSIX	Portable Operating System Interface
PPN	Processes Per Node
RMA	Remote Memory Access
Rmax	An HPL performance metric.
Task	A thread of execution associated with either an OS process or thread.
TFLOPS	10^{12} FLOPS
T_i/C_j	Task T_i has affinity to (can be scheduled to execute on) core C_j.
$T_i/C_j\|\|C_k$	Task T_i has affinity to cores C_j and C_k.
TX/RX	Send/Receive
UMA	Uniform Memory Access

Chapter 1

Introduction

Parallel computer simulation has been used as a tool to further scientific understanding for decades, as it provides a way to conduct experiments that would otherwise be too costly, dangerous, or impractical [7, 52, 58, 75, 76, 87]. Consequently, parallel programming systems and applications have evolved to improve their performance and scalability as computer systems have grown to higher degrees of parallelism. This, in turn, has led to a field of study concerning how best to structure, execute, and analyze massively parallel and distributed applications. In this work, we study all three of these aspects in the context of coupled message-passing programs used predominantly in the high-performance computing (HPC) domain.

1.1 Motivation

Hybrid parallel programming models that combine message passing (MP) and shared-memory multithreading (MT) are becoming more popular, especially with applications requiring higher degrees of parallelism and scalability. Consequently, *coupled* parallel programs, those built via the integration of independently developed and

optimized software libraries linked into a single application, increasingly comprise message-passing libraries with differing preferred degrees of threading, resulting in *thread-level heterogeneity*. Retroactively matching threading levels between independently developed and maintained libraries is difficult, and the challenge is exacerbated because contemporary middleware services provide only static scheduling policies over entire program executions, necessitating suboptimal *over-subscribed* or *under-subscribed* configurations. In coupled applications, a poorly configured component can lead to overall poor application performance, suboptimal resource utilization, and increased time-to-solution. So it is critical that each library executes in a manner consistent with its design and tuning for a particular system architecture and workload. Therefore, there is a need for techniques that address dynamic, conflicting configurations in coupled multithreaded message-passing (MT-MP) programs.

1.2 Thesis Statement

Our thesis is that we can achieve significant performance improvements over today's static under-subscribed approach through reconfigurable execution environments that consider compute phase parallelization strategies along with both hardware and software characteristics.

1.3 Contributions and Organization

This dissertation presents new ways to structure, execute, and analyze coupled MT-MP programs. For the remainder of this section, we outline this document's structure, summarizing significant contributions along the way. Please note that a substantial amount of material in this dissertation has been presented or published in other venues:

Samuel K. Gutiérrez, Kei Davis, Dorian C. Arnold, Randal S. Baker, Robert W. Robey, Patrick McCormick, Daniel Holladay, Jon A. Dahl, R. Joe Zerr, Florian Weik, and Christoph Junghans. Accommodating Thread-Level Heterogeneity in Coupled Parallel Applications. In *2017 IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Orlando, Florida, 2017.

Samuel K. Gutiérrez, Dorian C. Arnold, Kei Davis, and Patrick McCormick. On the Memory Attribution Problem: A Solution and Case Study Using MPI. *Journal on Concurrency and Computation: Practice and Experience (ExaMPI Special Issue Paper)*. To Appear.

Chapter 2: Background: We present an overview of core topics in parallel and distributed computation with a focus on concepts and techniques typical to HPC and their application to modeling and simulation. Related work and supplemental background material on specific topics are not presented here but instead located within relevant chapters that follow.

Chapter 3: Accommodating Thread-Level Heterogeneity: In this chapter, we study coupled MT-MP applications with dynamic, phased configuration conflicts. Focusing on applications based on the Message Passing Interface (MPI), we address the practical challenges of thread-level heterogeneity. We present a general methodology and corresponding implementation for dynamically (at runtime) accommodating coupled application configuration conflicts in a way that is composable, hardware topology aware, MPI implementation agnostic, works with a variety of unmodified Pthread-based parallel programming systems, increases overall system resource utilization, reintroduces lost parallelism, and is straightforward to incorporate into existing parallel applications. To the best of our knowledge, this is the first work to satisfy all of these criteria. Significant contributions of this work are summarized as follows:

- We examine contemporary approaches used to accommodate thread-level heterogeneity in coupled MT-MP programs. Here, we identify potential inefficiencies in how these coupled programs are currently structured and executed in the HPC domain.

- We present a novel approach for accommodating thread-level heterogeneity. Our approach enables full utilization of all available compute resources throughout an application's execution by providing programmable facilities to dynamically reconfigure runtime environments for compute phases with differing threading factors and affinities.

- We evaluate our methodology by applying it to three production-quality simulation codes employing a variety of parallelization strategies. Our performance results show that for a majority of the 30 tested scientific workloads our approach and corresponding open-source reference implementation, Quo, render speedups greater than 50 % over the static under-subscribed baseline.

Chapter 4: Addressing The Memory Attribution Problem: We present the design and implementation of memnesia, a novel memory usage profiler for parallel and distributed message-passing applications. Our approach captures component-level memory usage statistics from unmodified binaries dynamically linked to a message-passing communication library. This work is motivated principally by the lack of parallel tools capable of extracting metrics relevant to our study in Chapter 5 concerning Quo-induced memory overhead. Significant contributions of this work are summarized as follows:

- We examine contemporary approaches in memory profiling and discuss their limitations as pertaining to what we call *the memory attribution problem*: the inability to predict or evaluate during runtime where the available memory is used across the software stack comprising the application, reusable software

libraries, and supporting runtime infrastructure needed to enable the application at a given scale, under a given workload, and in a time- and space-sharing scheduled environment.

- We propose an approach for accurate, per-process quantification of memory resource usage over time that is able to distinguish between application and MPI library usage clearly. With this new capability, we show that job size, communication workload, and hardware/software architecture can influence peak runtime memory usage.

- We develop a corresponding open-source profiling library named memnesia for applications using any implementation of the Message Passing Interface. We develop this software with a specific goal in mind: once memory attribution is better understood, applications will potentially be able to improve or maintain their memory utilization as they are developed, maintained, and deployed.

- We evaluate our memory profiler's runtime overhead and behavior using micro-benchmarks. Here, we show that memnesia overheads are most apparent at small-message sizes, where its effect on operational latencies dominates messaging rates. Large-message bandwidth is least affected by the presence of memnesia instrumentation, as increased operational latencies are amortized over the transfer of larger payloads. That is, once a transfer is initiated, memnesia instrumentation has no appreciable effect on transfer rate.

- We discuss memnesia's memory overhead and application perturbation. For the former, we show that total memory overhead is proportional to $2s\sum_{i=0}^{m-1} n_i$, where the size of a single trace record $s = 25\,\mathrm{B}$, m is the total number of processes under memnesia supervision, and n_p is the total number of trace events triggered by process $p \in \{0, 1, 2, \ldots, m-1\}$. For the latter, we show that for a single process the amount of tool-induced application perturbation

is proportional to s times the number of trace records already collected by our event-driven profiler.

Chapter 5: Overhead of Adaptive Parallelism: A Case Study with Quo: Focusing on runtime and memory costs brought on by the use of our dynamic approach, we examine QUO's overhead using proxy and full applications. We show that QUO's overhead is modest, imposing small runtime and memory usage penalties over the static baseline. Our results are summarized as follows:

- We quantify the individual overhead costs for a representative set of QUO operations, showing that runtime operational latencies average ∼2 ms at 16 processes per node (PPN) across 128 nodes.
- We evaluate the overhead of QUO process quiescence by comparing two approaches, namely `MPI_Barrier()` and `QUO_barrier()`. Our results show that `QUO_barrier()` significantly outperforms `MPI_Barrier()` and is close to the ideal case where quiescence is not necessary. In particular, our approach introduces approximately an 8 % overhead, while the naive approach using `MPI_Barrier()` introduces approximately 116 % overhead.
- We quantify data remapping overhead at different scales and input configurations using three scientific applications. Because of our approach's quiescing and later resumption of tasks (i.e., MPI processes), application data remappings across library domains may increase and are dependent on job scale and inter-domain data movement requirements. So we study those overheads as a function of job size and application workload in two distinct regimes. Our results show that in the worst case data remapping consumes approximately 15 % of overall optimized application runtime—on average consuming about 4 % across the 30 workloads tested.

- We study Quo-induced memory overhead, focusing on two sources: Quo runtime state and increased hardware subscription levels often required by our approach. For the former, our experiments show that Quo's memory footprint is influenced primarily by hardware/software architecture, job scale, and process distribution—averaging across the 18 experiments ∼1.4 MB of additional memory per process. For the latter, we show that the cost of maintaining additional MPI processes is hardware-subscription-, platform- and implementation-dependent, averaging across the 18 configurations tested ∼14 MB per process.

Chapter 6: Conclusion: We conclude with a summary of our results and discuss opportunities for future work.

Chapter 2

Background

In this chapter, we present an overview of core topics in parallel and distributed computation with a focus on concepts and techniques typical to HPC and their application to modeling and simulation. We begin with a discussion of parallel computer architectures and programming models, focusing on Flynn's taxonomy and execution models that are of particular interest to this work. We then discuss HPC platforms and describe how they are commonly programmed. Finally, we conclude with a short description of parallel speedup.

2.1 Parallel Computers and Execution Models

A parallel computer comprises a potentially distributed collection of connected components (processors and memories) that work cooperatively to solve a computational problem. A standard way to classify these machines is Flynn's taxonomy, which categorizes computers according to the number of data streams and instruction (or control) streams they have [39]. In total there are four possibilities: SISD, MISD, SIMD, and MIMD.

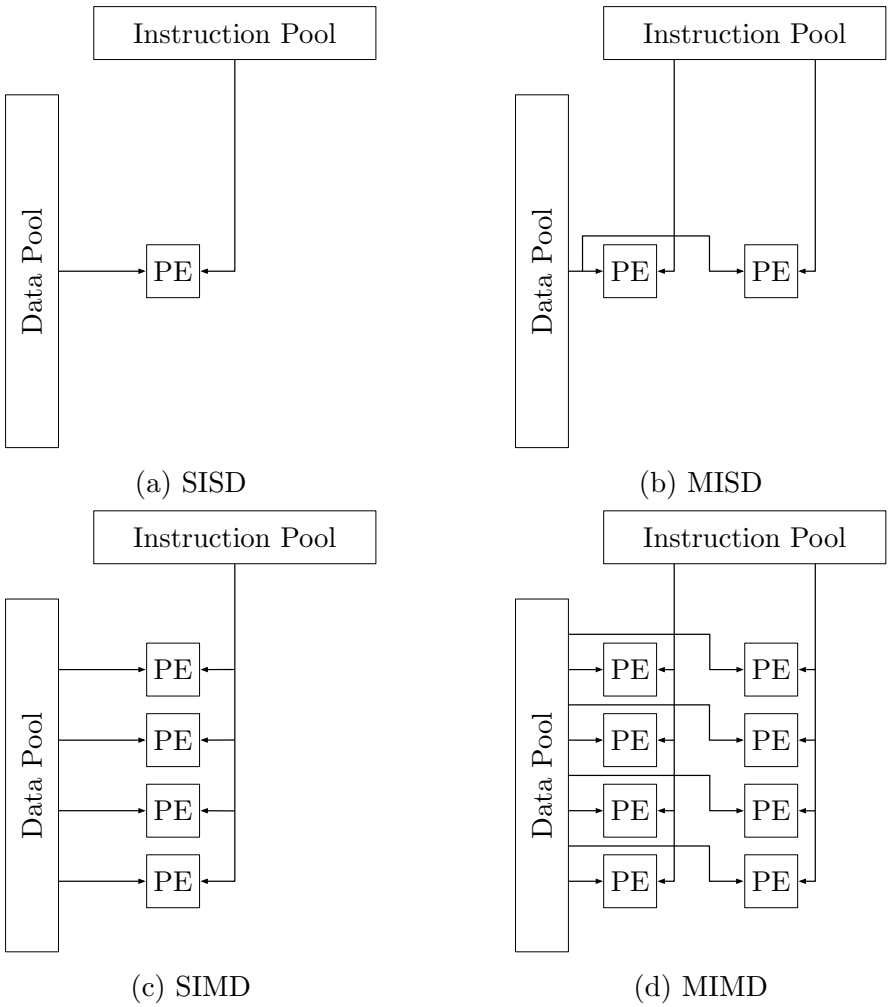


Figure 2.1: Flynn's computer taxonomy.

Single Instruction Stream, Single Data Stream (SISD): A sequential computer architecture exploiting neither instruction stream nor data stream parallelism as shown in Figure 2.1a.

Multiple Instruction Streams, Single Data Stream (MISD): An uncommon parallel computer architecture exploiting instruction stream parallelism on a single stream of data as shown in Figure 2.1b.

Single Instruction Stream, Multiple Data Streams (SIMD): A parallel com-

puter architecture exploiting data stream parallelism, where a single operation, such as a multiply, is applied to multiple data simultaneously as shown in Figure 2.1c.

Multiple Instruction Streams, Multiple Data Streams (MIMD): A parallel computer exploiting both instruction stream and data stream parallelism wherein multiple autonomous processors execute different operations on different data as shown in Figure 2.1d. The MIMD classification can be divided into two parallel execution models: SPMD and MPMD.

Single Program, Multiple Data Streams (SPMD): The most common parallelization strategy used in the high-performance computing domain. Developed by Darema et al. [34], SPMD is characterized by a set of cooperating tasks executing the same program while operating on multiple pieces of data.

Multiple Programs, Multiple Data Streams (MPMD): A parallel execution model that extends the SPMD model to multiple programs.

2.2 High-Performance Computing Platforms

HPC platforms are built for executing parallel numerical calculations of modeled systems. These parallel computer simulations are used across a broad range of scientific disciplines because they provide a way to conduct experiments that would otherwise be too costly, dangerous, or impractical. As a consequence, the compute capability of high-performance computing systems has grown exponentially over the last two decades (Figure 2.2) to keep pace with increasingly ambitious goals such as modeling complex physical phenomena through coupled multi-physics simulation [42].

Large-scale parallel computer simulations require an enormous amount of parallelism and memory capacity, so they must execute on parallel computers based on a distributed memory architecture in which compute nodes with local processors

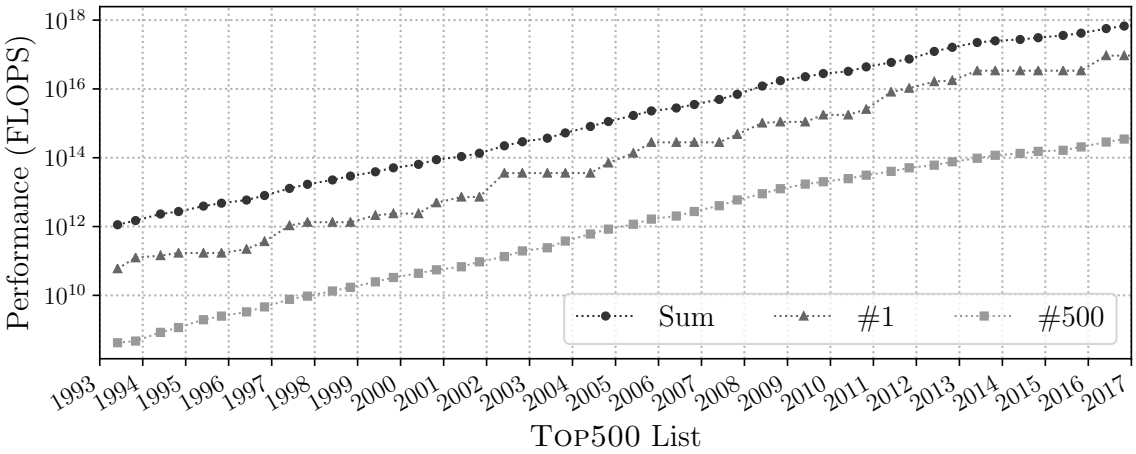


Figure 2.2: Performance development of HPC systems as recorded by the Top500.

and memories are networked to create a larger system—a *supercomputer*. Below we present architectural details of the top 10 supercomputers according to the November 2017 Top500 list (Table 2.1) and show the scales at which they operate (Table 2.2).

System	Node Architecture	Network
TaihuLight	Sunway SW26010 260C	Sunway
Tianhe-2	Xeon E5–2692v2 12C, Xeon Phi 31S1P	Express-2 Fat-Tree [74]
Piz Daint	Xeon E5–2690v3 12C, Tesla P100	Aries Dragonfly [8]
Gyoukou	Xeon D-1571 16C, PEZY-SC2	Infiniband EDR
Titan	Opteron 6274 16C, Tesla K20x	Gemini 3D Torus [9]
Sequoia	IBM Power BQC 16C	BG/Q 5D Torus [29]
Trinity	Xeon E5–2698v3 32C, Xeon Phi 7250 68C	Aries Dragonfly
Cori	Xeon E5–2698v3 32C, Xeon Phi 7250 68C	Aries Dragonfly
Oakforest	Xeon Phi 7250 68C	Omni-Path Fat-Tree [72]
K Computer	SPARC64 VIIIfx 8C	Tofu 6D Torus [6]

Table 2.1: Node and network architectures of contemporary HPC platforms.

System	Node Count	Core Count	Memory Capacity	Rmax
TaihuLight	40,960	10,649,600	1,311 TB	93,015 TFLOPS
Tianhe-2	16,000	3,120,000	1,375 TB	33,863 TFLOPS
Piz Daint	5,320	361,760	438 TB	19,590 TFLOPS
Gyoukou	1,250[1]	19,860,000	576 TB	19,136 TFLOPS
Titan	18,688	560,640	710 TB	17,590 TFLOPS
Sequoia	98,304	1,572,864	1,536 TB	17,173 TFLOPS
Trinity	19,392	979,968	2,163 TB	14,137 TFLOPS
Cori	12,076	735,200	1,469 TB	14,015 TFLOPS
Oakforest	8,208	558,144	919 TB	13,555 TFLOPS
K Computer	88,128	705,024	1,410 TB	10,510 TFLOPS

[1]A node is defined here as eight SC2 chips connected to a single Xeon D-1571.

Table 2.2: The scales at which contemporary supercomputers operate.

2.3 Parallel Programming Environments

Most parallel and distributed scientific applications (or software libraries) are programmed using general-purpose languages parallelized via optimizing compiler techniques (e.g., automatic vectorization), language features or extensions (e.g., parallel loop constructs), or runtime/middleware system services (e.g., collective inter-process communication). Supercomputers offer a hierarchy of exploitable concurrency, so parallelism in scientific programs is achieved by combining approaches that best suit each level of an architecture-defined hierarchy. An example is the use of message-passing SPMD for coarse-grained parallelism in which distributed tasks also exploit loop- and instruction-level parallelism from within a node. Below we describe popular approaches used in contemporary scientific software engineering.

2.3.1 Message Passing

In the context of parallel programming models, message passing is a way of structuring cooperation between a collection of tasks executing concurrently on a parallel computer. In this model, tasks cooperate by sending messages to one another; so the sharing of data by other means, for example, shared memory, is prohibited. Data may be exchanged between cooperating tasks using *synchronous* or *asynchronous* messages, or a combination thereof. The former requires synchronization between a sender and receiver such that the receiver is *ready* for receipt of a message before the sender initiates transmission, whereas asynchronous messaging is less restrictive in that a message may be sent to a task before it is ready (or able) to receive data. The exchange of data between a single sender and a single receiver is called *point-to-point communication*. The generalization of this concept, which allows for the transfer of data between multiple senders and receivers, is called *collective communication*. For more information about message passing models, consult the seminal works by Hewitt et al. [54], Baker and Hewitt [16], Hoare [55], Valiant [99], and Milner [79].

Now that we have a general understanding of the message passing model let us now focus on a particular message passing specification: MPI, the Message Passing Interface [78]. MPI is a portable application programming interface (API) specification for point-to-point communication with extensions to the canonical message passing model that includes collective communication, remote memory access (RMA) operations, dynamic process creation, and parallel file input/output (I/O). Function calls defined by the standard's C and Fortran language bindings express these operations, though other language bindings exist outside the standard's purview. For over twenty years, MPI has served as the de facto message passing standard for parallel and distributed scientific applications. Thus, a tremendous amount of software infrastructure has been designed and built around its specification, which has undergone two major revisions since its official debut in 1994.

2.3.2 Shared-Memory Multithreading

In the shared-memory model, a collection of tasks share a common address space, and any data that are not explicitly designated as *task-local*, accessible only to a single task, are shared. Tasks are allowed to read and write shared data structures asynchronously, so multithreaded programs are carefully structured to avoid *race conditions*, undesirable non-determinism that can affect program correctness. Because scientific applications require large amounts of memory, shared-memory parallelization strategies cannot be used without a distributed-memory component. Consequently, hybrid approaches combining shared-memory multithreading and message passing are becoming commonplace in scientific software (details in Chapter 3).

2.4 Parallel Speedup

Speedup is a standard metric used to assess the scalability of parallel programs. For a fixed problem size, x, speedup is defined as follows:

$$S(p, x) = \frac{T(1, x)}{T(p, x)}\,, \tag{2.1}$$

where $T(1, x)$ is the time taken by an optimized sequential program to perform a given computation on a single processor and $T(p, x)$ is the time taken to perform the same calculation in parallel using p processors. Measuring speedups in this way quantifies a parallel program's *strong scaling* characteristics for a given workload at increasing processor counts. By contrast, *weak scaling* measures speedups as a function of both problem size and processor count and assumes a fixed problem size per processor, so speedup in this regime is defined as

$$S(p, x \cdot p) = \frac{T(1, x)}{T(p, x \cdot p)}\,. \tag{2.2}$$

Demonstrating strong scaling requires solving a fixed problem size faster as p increases, while demonstrating weak scaling requires solving increasing larger problems within a fixed amount of time.

2.5 Summary

In summary, supercomputers can be structured and programmed in different ways. Some are built entirely from commodity components, a classic example is a Beowulf cluster [20], while others opt for custom hardware altogether—the approach used for TaihuLight. More commonly, though, a graded approach is taken where both custom and commodity technologies are integrated into a single distributed memory machine. As an example let us consider the architectural features of the computers listed in Table 2.1. Here we will notice that the majority (9/10) are built using commodity technologies interconnected by specialized high-speed networks. With this diversity, achieving program and performance portability is challenging, so parallelism is achieved and expressed in different ways.

Chapter 3

Accommodating Thread-Level Heterogeneity

Parallel and distributed software such as multi-physics applications play crucial roles in science and engineering. Because of their interdisciplinary nature, these applications are often coupled, that is, built via the integration (or coupling) of independently developed and optimized software libraries linked into a single application. As previously described, in such coupled applications, a poorly performing library can lead to overall poor application performance, suboptimal resource utilization, and increased time-to-solution, so it is critical that each library executes in a manner consistent with its design and tuning for a particular system architecture and workload. Generally, each library (*input/compute phase pair*) has its optimal *runtime configuration*, for example, number and placement of processes or threads. In coupled applications, effective configuration parameters are determined (most often heuristically, manually, and offline) for all performance-critical computational phases.

Configuration conflicts arise when an optimal configuration for one phase is suboptimal for another, and there are a variety of approaches for resolving configuration

conflicts. At one extreme lie applications written to parallel and distributed programming systems such as Legion [18] and Charm++ [62], which by design resolve such conflicts at runtime. At the other extreme lie MT-MP applications that use message passing for inter- and intra-node parallelism and multithreading for additional intra-node parallelism, where the common approach is to allocate resources to satisfy the most demanding compute phase. The library with the highest degree of threading per process has one processing element per thread, and libraries with fewer threads per process run under-subscribed, using only a fraction of the available compute resources when running.

In this chapter, we study coupled MT-MP applications with dynamic, phased configuration conflicts. Focusing on applications based on the Message Passing Interface, we address the practical challenges of thread-level heterogeneity, where a coupled application comprises MPI libraries requiring different degrees of thread-level parallelism. We present a general methodology and corresponding implementation for dynamically (at runtime) accommodating coupled program configuration conflicts in a way that is composable, hardware topology aware, MPI implementation agnostic[1], works with a variety of unmodified Pthread-based parallel programming systems, increases overall system resource utilization, reintroduces lost parallelism, and is straightforward to incorporate into existing applications. To the best of our knowledge, this is the first work to satisfy all of these criteria. Finally, we evaluate our methodology by applying it to three production-quality simulation programs that employ a variety of parallelization strategies. Our results show that significant performance improvements are achievable when used in environments positioned to make effective use of the additional levels of parallelism our approach enables.

[1]So long as the underlying representation of an MPI process is a system process. This is true for most MPI implementations with one notable excepion: MPC-MPI [81, 82].

3.1 Coupled Applications and Their Challenges

As previously described, parallel applications are often built by coupling independently developed and optimized software libraries. For example, coupled physics applications are often implemented in a fashion where each physics library, in turn, updates common application state data. Such scientific libraries tend to have their preferred data discretization scheme, for example, unstructured meshes, regular meshes, or particles, so they manage their distributed state and parallelization strategies with little or no coordination across library boundaries. More generally, libraries interact by exchanging data through APIs that remap data from one library domain to another, for example, from a field defined on a computational mesh to a system of linear equations, or from one mesh to another as illustrated in Figure 3.1. Quite often, such data structure remappings suggest complementary remappings of tasks to hardware. Inter-library interactions can take place many times during the lifespan of an application. Furthermore, at a given program point these interactions may change during a simulation to accommodate new requirements, for example, particular physics appropriate for a specific spatial scale and resolution.

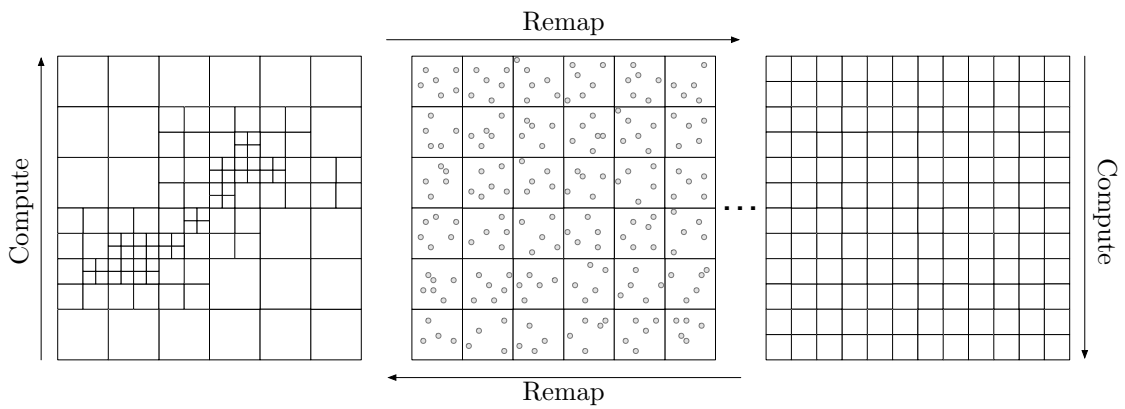


Figure 3.1: Notional illustration of computational phases interleaved with data structure remapping phases across library domains.

3.1.1 Parallelism

Parallel scientific application executions exploit data parallelism, where many instances of the same computation execute in parallel on different data and on different computational resources. In the canonical MP model, message passing is used for both inter- and intra-node parallelism (other than SIMD vectorization). For MPI applications this is called *MPI-everywhere*. In this model, computational resources are usually *fully subscribed*, that is, the program's set of single-threaded processes is in one-to-one correspondence with processing elements (i.e., cores or hardware threads) and parallelism is realized via either SPMD or MPMD schemes. Alternatively, a scientific application can employ a hybrid model using multithreaded message passing (MT-MP) for inter- and intra-node parallelism. For MPI applications, MT-MP is an instance of the more general *MPI+X* model in which applications employ additional on-node parallelization strategies. This approach is increasingly popular as core (or hardware thread) counts increase in shared-memory nodes, and because of the flexibility and performance potential of a hierarchical approach [26, 53, 73].

While MPI+X is gaining popularity, it is not ubiquitous. Restructuring large, mature code bases to exploit new parallel programming systems effectively is challenging and generally requires a significant amount of effort that is often unjustifiable because of cost or priority. Furthermore, it is not uncommon that an MPI-everywhere version of a scientific code performs as well as or better than its MPI+X instantiation [15, 69], which discourages speculative hybridizing of existing programs. Finally, while an MPI+X library may be written such that its runtime configuration is settable within some range at startup, the particular runtime parameters that give the best performance may depend on static or dynamic variables such as input and problem scaling factors. For all of these reasons, coupled scientific codes will for the foreseeable future continue to be built from libraries that use a mix of non-uniform runtime configurations as illustrated in Figure 3.2. A runtime configuration may

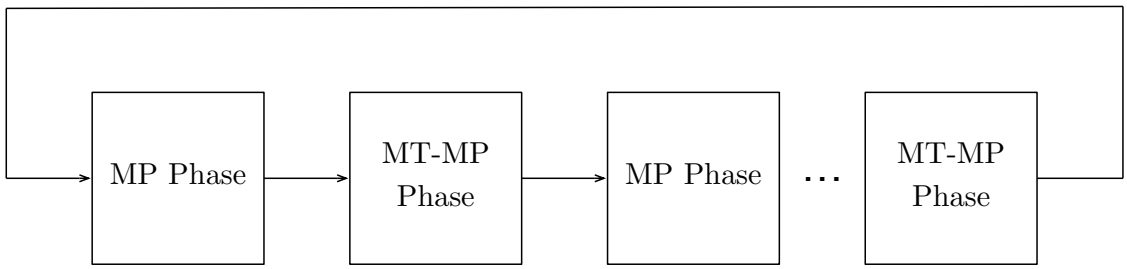


Figure 3.2: Interleaved phases of a coupled thread-heterogeneous message-passing application with non-uniform runtime configuration requirements.

include the total number of processes to use for SPMD or MPMD parallelism, a process threading degree for shared-memory multithreading, and a mapping of tasks (processes and threads) to compute resources, for example, PEs and memories.

3.1.2 Conflicting Configuration Requirements

For decades coupled applications had relatively uniform library configuration requirements because they were built from single-threaded message-passing libraries, so static configurations set at application launch were sufficient. Today, however, configuration conflicts are common in coupled applications because they comprise independently developed and maintained scientific libraries that have been written or ported to hybrid MT-MP programming models.

Static Configurations

In today's static computing environments, dynamically accommodating inter-library configuration conflicts is difficult. While it is well understood that binding tasks to hardware resources can improve the performance of an MPI application [25, 43], parallel application launchers such as orterun [27], srun [103], aprun [66], and Hydra [71] allow only static allocations and static binding capabilities: launch-time configura-

tions persist for the entire parallel application's execution. Most single-threaded applications are launched by binding a single PE dedicatedly to each process. This approach mitigates ill effects of task migration in multiprocessor architectures, for example, cache invalidation that occurs when a thread moves from one PE to another.

Given a static configuration for coupled MT-MP applications with conflicting configurations, the two primary configuration options are *over-subscription* and *under-subscription*. In over-subscribed configurations, all allocated resources are always in use, that is, the number of PEs equals the number of threads in the computational phase with the lowest degree of threading per process. In phases that require more threads, resources are over-subscribed with multiple threads per PE. Figure 3.3 illustrates the evolution of an over-subscribed MPI+X configuration where hardware utilization $u(t)$ remains constant at 100 %. In this example, MPI-everywhere phases fully subscribe hardware resources (phases P0-P2, P5-P6), while multithreaded regions over-subscribe them (phases P3-P4). In practice, over-subscription is generally avoided because increased resource contention in the threaded regions tends to affect overall application performance and scalability negatively [101].

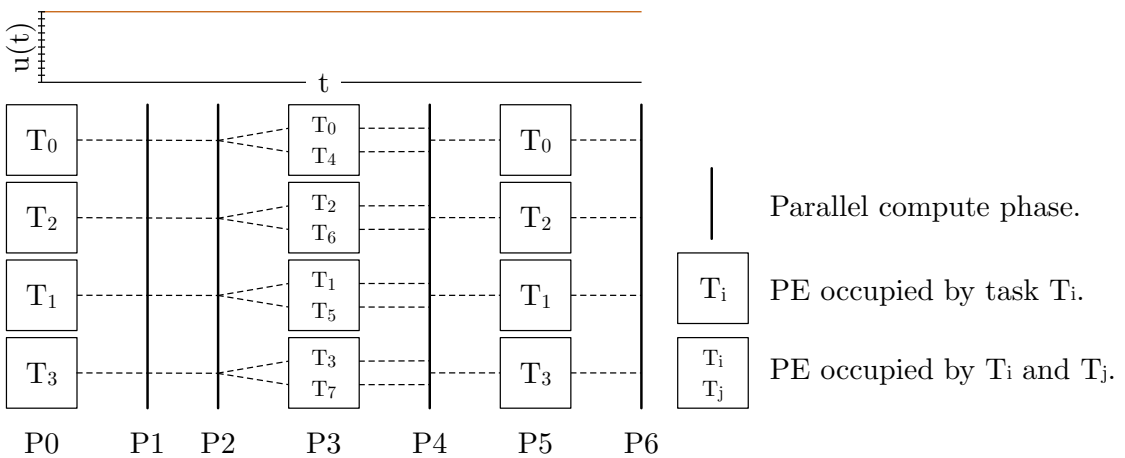


Figure 3.3: Compute resource utilization $u(t)$ by tasks (processes and threads) over time of a static over-subscribed MPI+X configuration.

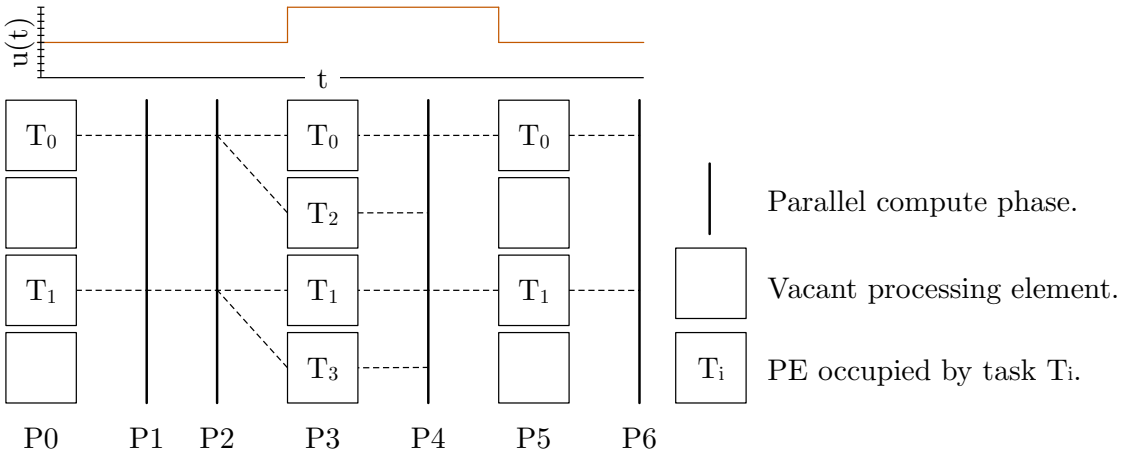


Figure 3.4: Compute resource utilization $u(t)$ by tasks (processes and threads) over time of a static under-subscribed MPI+X configuration.

The standard approach for accommodating thread-level heterogeneity in coupled MPI applications is to statically (at application launch time) under-subscribe compute resources such that the computational phase with the highest degree of threading per MPI process has one PE per software thread. As a consequence, phases with fewer threads per process use only a fraction of the available compute resources, thus leading to poor system resource utilization. Figure 3.4 illustrates the evolution of compute hardware resource utilization over time for a typical MPI+X configuration. Over time, hardware utilization fluctuates between 50 % and 100 % as the application moves in and out of regions with differing degrees of multithreading.

Lost Parallelism Through Resource Under-Subscription

Given an application that strong-scales perfectly (the theoretical upper bound), we can calculate the theoretical slowdown of static under-subscription approaches using Amdahl's law [11],

$$S = \left(\sum_{i=1}^{m} \frac{p_i}{s_i} \right)^{-1}, \tag{3.1}$$

where n is the total number of available processors; m is the total number of phases; t_i is the optimal threading degree for a phase i; $t_{\max} = \max(t_1, \ldots, t_m)$; $u_i = t_i / t_{\max}$ is a phase's processor under-subscription factor; and $s_i = n \cdot u_i$ is the speedup factor for a given phase. Consider two serial phases L1 and L2 whose percentages of execution time are equal: $p_1 = 0.5$ and $p_2 = 0.5$. Assuming L1 runs optimally with an MPI-everywhere parallelization strategy and L2 optimally with an MPI+X strategy, Figure 3.5 plots the theoretical speedups of three under-subscribed runtime configurations where L1's threading degree is fixed at $t_1 = 1$ and L2's varies. We compare those to an idealized configuration (Ideal) where each phase of the parallel computation is exposed to all available PEs. This simple model illustrates the potential losses in parallelism resulting from today's static under-subscription approach.

In summary, coupled scientific applications based on the MT-MP model can comprise libraries with conflicting configuration requirements. For such applications, today's static computational environments necessitate suboptimal over-subscribed or under-subscribed configurations. Therefore, there is a need for techniques that address dynamic, conflicting configurations in coupled MT-MP applications.

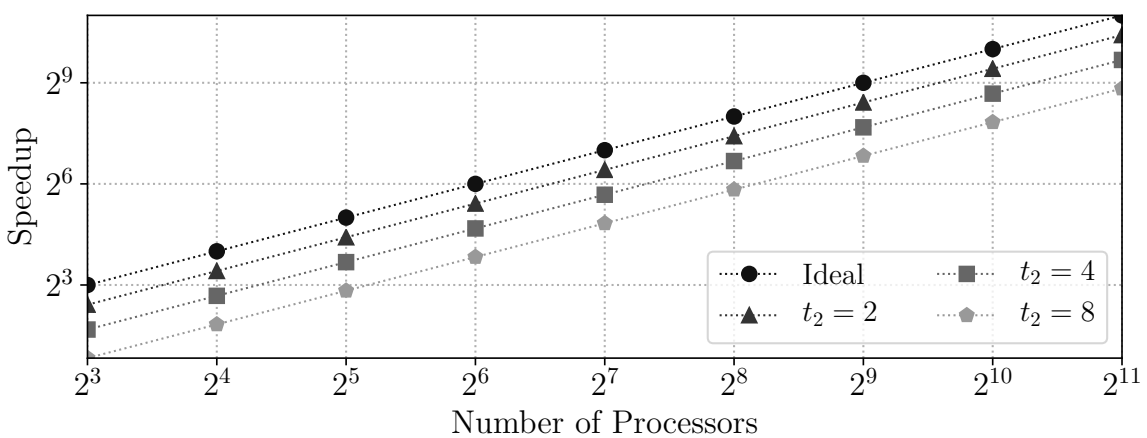


Figure 3.5: Log-log plot of modeled speedups showing the potential losses in parallelism resulting from under-subscription as a function of scale.

3.2 Background and Related Work

In this section, we begin with a discussion of portable hardware locality and abstract hardware representation, focusing exclusively on techniques and software systems used in this work. We then discuss related topics in multiprocessor scheduling, affinity scheduling, and runtime configuration conflict resolution.

3.2.1 Portable Hardware Locality

Contemporary HPC node architectures are complex and diverse, demanding careful consideration of their processor and memory configurations. To effectively guide the dynamic (runtime) mapping of application-specific software (logical) affinities to hardware resources, one must be able to obtain both the underlying platform's resource information and the application's current usage of those resources. To that end, we use hwloc [25, 44], an open-source software library that provides services to gather such information at runtime from the most popular and widely used operating systems in HPC. Operating system and hardware portability are achieved by the library's use of Standard C (ANSI/ISO C), a common, widely supported C standard, and a plugin architecture that allows for back-end component specialization.

3.2.2 Abstract Hardware Representation

hwloc represents hardware information as a k-ary tree of typed hardware objects that carry additional information through type-appropriate attributes, for example, cache level and size. The topology formed by these objects represents memory and processor relationships existing in hardware, including cache, NUMA, and other general hardware affinities such as those existing when processing elements are packaged together within a socket. Further, abstract hardware topologies need not be *full* (for

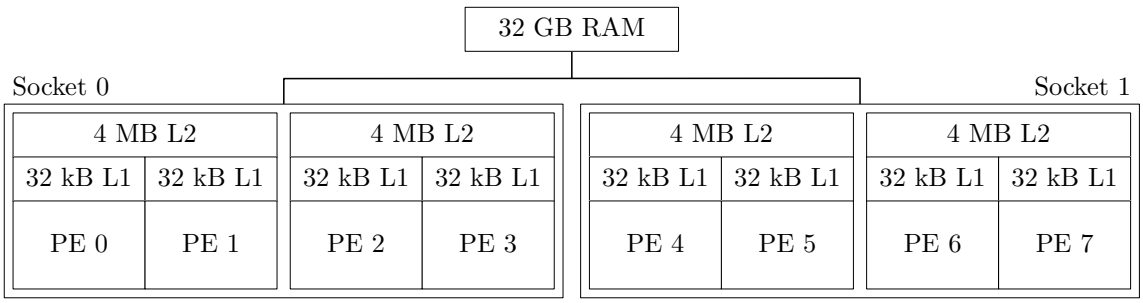


Figure 3.6: Schematic of a machine with two quad-core sockets.

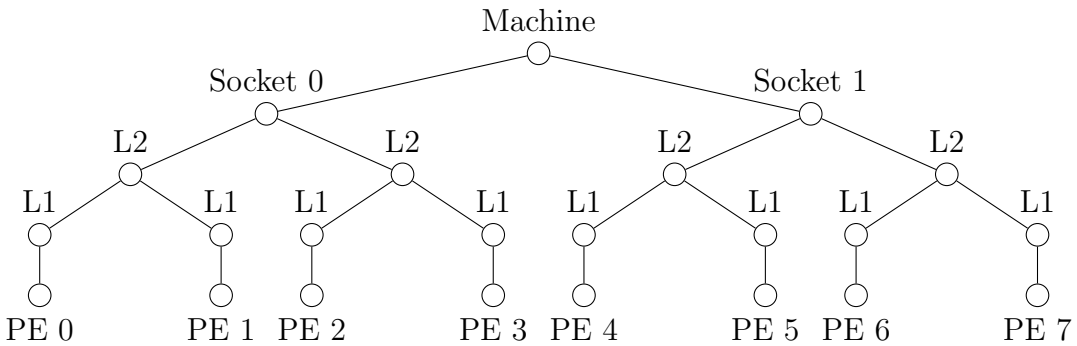


Figure 3.7: Hardware topology of the machine diagrammed in Figure 3.6.

all levels each node has exactly 0 or k children), *symmetric* (being a mirror image of itself), or *homogeneous* (built from exactly one type of memory or processor), and so can represent complex hardware configurations. The binary tree shown in Figure 3.7 depicts the compute node architecture diagrammed in Figure 3.6, namely a uniform memory access (UMA) machine built from two quad-core sockets, where each core has a dedicated level one cache (L1) and a shared level two (L2).

3.2.3 Multiprocessor Scheduling

Multiprocessor Scheduling (MSP) is a well-known optimization problem that can be stated as follows: given q processors and n tasks of varying lengths, find

a *schedule*—an assignment of n tasks to q processors—with minimal total length. Because of the problem's considerable importance and due to its computational intractability (MSP is known to be NP-hard [19, 41]), many efficient heuristics have been proposed and studied [12, 31, 57, 67, 77, 91, 100]. Here, we focus on a few that align with our primary interests.

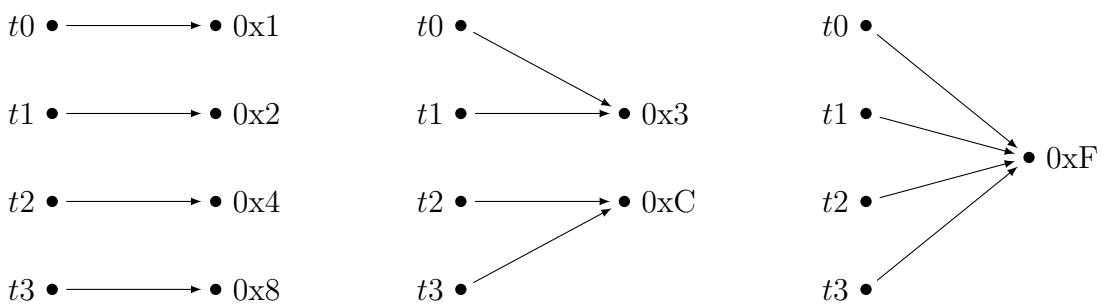
Affinity Scheduling

Markatos and LeBlanc studied *affinity scheduling* in their work concerning loop scheduling algorithms for shared-memory multiprocessors [77]. Their results show that parallel workloads on such systems can benefit from scheduling policies that exploit data locality in caches and local memories during parallel loop iteration assignment to processors. Their work builds on that of Squillante and Lazowska, which shows that locality-indifferent scheduling increases the *cache-reload transient penalty*—the cache misses caused by bringing a task's working set from main memory into cache as it is reinitiated after being suspended temporarily [96]—thus negatively affecting both individual task and overall system performance [91].

An affinity schedule is given by a potentially dynamic specification that exploits application-specific features (e.g., adjacency graphs, parallelization strategies) to improve upon application-oblivious schedules. The problem can be stated as follows: given q processors and a set of cooperating tasks T, find a set of task scheduling directives M and a binary relation $R : T \mapsto M$ that minimizes T's makespan. Most often the derivation of M and R involves offline manual experimentation, but automated techniques have been proposed and studied in the literature [5, 21, 22, 23, 60, 61, 88, 89]. A task's *CPU affinity mask* determines the set of processors $\hat{P}$ on which it is eligible to run and is defined as follows: given q linearly ordered processors $P = \{p_0, p_1, p_i, \ldots, p_{q-1}\}$, let m be a least significant bit zero (LSB-0) string defined by $m_i \leftarrow 1$ if use of p_i is permitted and $m_i \leftarrow 0$ otherwise,

so $\hat{P} = \{p_i \mid 0 \leq i < n, m_i = 1\}$. To better understand what we mean, consider the following scenarios given four processors $P = \{p0, p1, p2, p3\}$ and as many cooperating tasks $T = \{t0, t1, t2, t3\}$. Relation R_1, diagrammed in Figure 3.8a, represents a configuration commonly used for single-threaded data-parallel workloads: a processor is dedicated to a single thread of execution to minimize task migration costs. If we let $T_l = \{0, 1\}$ and $T_u = \{2, 3\}$ be two multithreaded processes taken from T, then Figure 3.8b represents a typical configuration used to influence the schedule of multithreaded data-parallel workloads: each multithreaded process is given a set of dedicated processing elements so that threaded regions can execute in parallel while maintaining some memory locality—an especially important point on NUMA architectures. Finally, the relation shown in Figure 3.8c represents a completely permissive scheduling policy. That is, each task is runnable (can be scheduled) on any available processor. In practical terms this means that the operating system's scheduler can assign tasks to resources in any way it sees fit, most likely using completely fair scheduling.

$$M_1 = \{0001_2\ (0\text{x}1), 0010_2\ (0\text{x}2), 0100_2\ (0\text{x}4), 1000_2\ (0\text{x}8)\}$$
$$M_2 = \{0011_2\ (0\text{x}03), 1100_2\ (0\text{xC})\},$$
$$M_3 = \{1111_2\ (0\text{xF})\}$$



(a) $R_1 : T \mapsto M_1$ (b) $R_2 : T \mapsto M_2$ (c) $R_3 : T \mapsto M_3$

Figure 3.8: Example task to affinity mask relations.

3.2.4 Current Approaches in HPC

Affinity scheduling has proven useful for improving HPC workload performance [25, 43, 60, 90], so much that, in one form or another, it is regularly used in practice. Today, the majority of parallel applications use static affinity (or *binding*) policies characterized by their use of a single, persistent configuration set during parallel application startup. Most single-threaded applications are launched by binding a single processing element dedicatedly to each process to mitigate the ill effects of task migration, for example, cache invalidation occurring when a task moves from one processing element to another. In contrast, multithreaded applications tend to favor less restrictive process binding policies that allow for maximal hardware-level parallelism during the execution of threaded regions, meaning in practical terms that a CPU affinity mask is chosen so a process's threads are eligible for execution on an appropriate number of processing elements.

Hybridizing MPI applications has been studied extensively [30, 36, 84, 85]. These works suggest that choosing between MPI-everywhere and MPI+OpenMP is a non-trivial task involving careful consideration regarding, but not limited to, algorithmic choices in the application and the characteristics of the target architecture. These works evaluate MPI+OpenMP schemes that use a static under-subscription approach, whereas we present a general methodology to dynamically accommodate a broader set of Pthread-based MPI+X schemes that additionally consider both data and hardware localities at runtime.

The study of dynamic process and memory binding methodologies that consider application, data, and hardware localities is not new. Broquedis et al. present and evaluate hwloc by incorporating it into MPI and OpenMP runtimes to dynamically guide task affinities at runtime [25]. While similar in many respects, our work differs from theirs in that we present a general methodology for programmatically resolving

configuration conflicts in dynamic, phased Pthread-based MPI+X programs—a use case not considered in their work.

For HPC applications there are a variety of published approaches for efficiently resolving runtime configuration conflicts that arise in thread-heterogeneous environments. Carribault et al. present a unified runtime for both distributed- and shared-memory MPI+X codes [81, 82]. Unlike other MPI implementations, theirs implements MPI processes as user-level threads (instead of processes), so their scheduler can efficiently accommodate both single- and multi-threaded regions during the execution of an MPI+X application. In contrast, our approach is MPI implementation agnostic and exposes an API to programmatically influence task placement and scheduling at runtime. Huang et al. present another MPI implementation that uses processor virtualization to facilitate application adaptation, including thread-level heterogeneity [59]. Their approach, however, requires the use of their runtime and modified versions of others, for example, GNU OpenMP, whereas ours works with unmodified MPI and OpenMP runtimes. Other parallel and distributed programming systems such as Legion [18] and Charm++ [62] are designed to dynamically resolve runtime configuration conflicts, but once again require that applications be rewritten to their respective paradigms.

3.3 Adaptive Parallelism for Coupled MPI+X

Next we present a general, composable runtime approach for programmatically accommodating library configuration conflicts that arise in dynamic, coupled, thread-heterogeneous MPI+X applications. Our design is influenced by requirements for generality, composability, efficiency, and pragmatism in the face of production HPC software realities, that is, easily fitting into large, established code bases that are still be under active development.

3.3.1 Adaptive Execution Environments with Quo

Quo (as in *"status quo"*) is both a model and a corresponding implementation that facilitates the dynamically varying requirements of computational phases in coupled MT-MP applications. Specifically, Quo allows an application to dynamically query and reconfigure its process bindings. While the model is general, the current implementation focuses on Pthread-based MPI+X applications [49]. Fundamentally, Quo uses hwloc [25] and MPI, interfacing with those libraries and the application as shown in Figure 3.9. The hwloc library is used to gather system hardware topology information and to control process binding policy changes during the target application's lifetime. MPI is used primarily for exchanging setup information during Quo *context* (QC) setup, which is discussed in a later section.

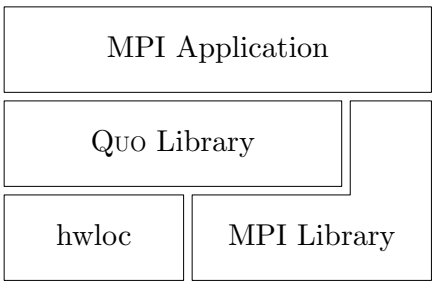


Figure 3.9: Quo architecture diagram.

The portable, production-quality, open-source runtime library is written in C, but also provides C++ and Fortran language bindings. The Quo API operates on QC pointers. This design allows for the creation and manipulation of multiple QCs within a single application that are either encapsulated within a library or passed from one library to another—a key for composability. The remainder of this section presents the principle concepts and mechanisms that underlie our design and approach.

3.3.2 Quo Contexts

Quo contexts, which encapsulate state data gathered and manipulated by Quo, are created via a collective call to `QUO_create()` in which all members of the initializing communicator must participate. QCs may be created at any time after the underlying MPI library has been initialized and remain valid until freed via `QUO_free()`, which must occur before the MPI library has been *finalized*. Quo can maintain multiple independent, coexisting QCs within a single application.

3.3.3 Hardware/Software Environment Queries

As previously described in Section 3.2.1, contemporary HPC node architectures are complex and diverse, demanding careful consideration of their resource (PE and memory) configurations. To effectively guide the dynamic (runtime) mapping of application-specific software (logical) affinities to hardware resources, one must be able to obtain both the underlying platform's resource information and the application's current usage of those resources. In this regard, Quo's approach is straightforward: its API provides thin convenience wrappers around commonly-used hwloc hardware query routines for hardware information. Relevant hardware information includes memory localities relative to PEs in non-uniform memory access (NUMA) architectures and hierarchical hardware relationships (e.g., determining how many cores are contained in a particular socket).

Process affinity state queries provide another mechanism to influence runtime software-to-hardware mapping decisions based on the hardware affinities of cooperating processes within a compute node. For example, on a per-node basis, one can query for the set of MPI processes with affinity to a particular hardware resource. For these queries, Quo uses a combination of hwloc and MPI services. For a given QC, Quo uses MPI to share a cached mapping of MPI processes to process IDs,

and hwloc is used to query the affinities of the relevant processes. We note that to effectively map tasks to PEs, both intra-process (first party) and inter-process (third party) affinity state queries are necessary.

3.3.4 Programmable Dynamic Process Affinities

Quo allows arbitrary process binding policies to be enacted and reverted during the execution of an MPI+X application. Ideally, binding instantiations and reversions will coincide with the entries and exits of the application's different computational phases. Accordingly, Quo exposes a straightforward, stack semantics through `QUO_bind_push()` and `QUO_bind_pop()`. For example, a new process binding policy can be instantiated before entering a computational phase via `QUO_bind_push()` and then reverted at the end of that phase via `QUO_bind_pop()`. This semantics allows a user to conveniently and dynamically stack an arbitrary number of binding policies that map to the potentially stacked composition of coupled components in a Quo-enabled MPI+X application (Listing 3.1 and Figure 3.10).

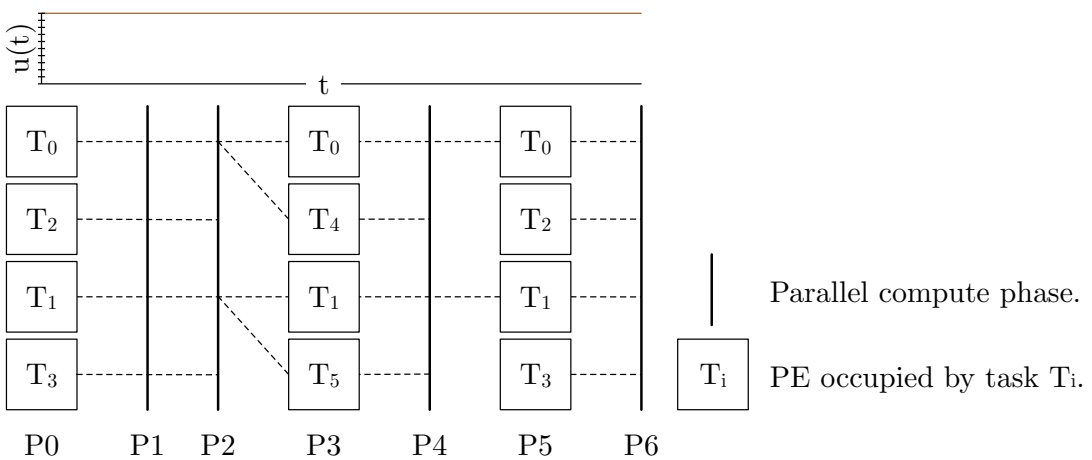


Figure 3.10: Compute resource utilization by tasks over time $u(t)$ for a Quo-enabled MPI+X configuration. Consult Table 3.1 for a description of the phases.

	Phase Description
Phase P0	Four single-threaded processes $P = \{T_0, T_2, T_1, T_3\}$ are launched onto cores $R = \{C_0, C_1, C_2, C_3\}$, where each process $T_{0,\ldots n-1} \in P$ has affinity to the core on which it was launched: T_0/C_0, T_2/C_1, T_1/C_2, T_3/C_3. Process state data $S = \{M_0, M_2, M_1, M_3\}$ is initialized for each process in P.
Phase P1	Processes in P execute in parallel during first compute phase, fully subscribing the compute resources in R.
Phase P2	Processes in P map data from their domain X (resident in S) to the callee's domain Y, $\mathbb{M} : X_m \rightarrow Y_n$, where $m = \lvert P \rvert$ and $n = \lvert L \rvert$. Then two processes in P, $Q = \{T_2, T_3\}$, are quiesced while the remaining processes $L = P - Q$ *push* a new binding policy such that their hardware affinities expand to cover two cores each: $T_0/C_0 \lvert\lvert C_1$, $T_1/C_2 \lvert\lvert C_3$.
Phase P3	Two new threads $\hat{P} = \{T_4, T_5\}$ are spawned by their respective parents in L onto cores C_1, C_3: cores once occupied by MPI processes in Q. State in M_0 is shared between T_0 and T_4, while M_1 in a similar fashion is shared between T_1 and T_5.
Phase P4	Processes and threads residing in $L \cup \hat{P}$ execute in parallel during this compute phase, fully subscribing the compute resources in R. The threaded compute phase completes and the spawned threads in $\hat{P}$ die or are suspended by the threading library's runtime. Processes in L revert to their previous binding policies by *popping* them off their respective affinity stacks.
Phase P5	Processes in Q resume execution on the computational resources they relinquished in Phase P2.
Phase P6	Processes in P map data from domain Y (resident in $\hat{S} = \{M_0, M_1\}$) back to the caller's domain X (residing over state in S), $\mathbb{M} : Y_n \rightarrow X_m$, where $n = \lvert L \rvert$ and $m = \lvert P \rvert$. That is, results are disseminated via explicit message passing from n processes in L to m processes in P.

Table 3.1: Explanation of Quo-enabled MPI+X phases in Figure 3.10.

QUO offers two variants of `QUO_bind_push()`. The first pushes a hardware affinity policy specifically outlined by the caller. This variant unconditionally, without regard to the caller's current hardware affinities, changes the calling process's affinity mask to encompass the PEs dominated by the provided policy. QUO also offers a more sophisticated version of this call that first queries the caller's current hardware affinities to choose the *closest* target resource that dominates the caller's current hardware affinities in hwloc's hardware object tree. If, for example, the caller currently has an affinity to a core in socket 3, then a call to the latter variant with a target resource of `SOCKET` will automatically expand the caller's affinity mask to encompass all PEs in socket 3. The rationale for this functionality is to maintain data locality (i.e., memory affinity) while moving in and out of process binding policies, in this case keeping data resident within one NUMA region across library calls.

Intra- and inter-process affinity state queries are used to guide dynamic binding policy choices and are often used in concert with `QUO_bind_push()`. For added convenience, QUO offers an automatic task distribution capability through `QUO_auto_distrib()`. This routine automates the two-step query and bind process at the cost of generality. Specifically, this routine allows callers the ability to distribute tasks across a specified resource with minimal effort. For example, one can specify that a maximum of two tasks be assigned to each socket on the target compute resource, and this routine will do so by choosing at most two tasks that are enclosed within (i.e., have an affinity to) each respective socket. When there exists a subset of cooperating processes not *bound* to a particular hardware resource, QUO favors bound processes, avoiding *unbound* processes even if the distribution criteria were not completely satisfied with bound processes. This helps maintain data locality when moving in and out of process binding policies, easing programmer burden. With these primitives, applications can dynamically create policies tailored specifically to their current needs based on the underlying hardware characteristics and the current process binding policies of other participating processes within a node.

3.3.5 Data Dependencies

Before the number of active MPI processes can be safely increased or decreased, data must be exchanged among node-local processes to satisfy all inter-process data dependencies. Typically, this occurs via node-local gathers and scatters before and after QUO-enabled regions as described in Listing 3.1 (P2 and P6) and shown in Figure 3.11. As is typical for message passing models, inter-process data dependencies are managed explicitly and programmatically. Once dependencies are satisfied, QUO can enact arbitrary task reconfigurations.

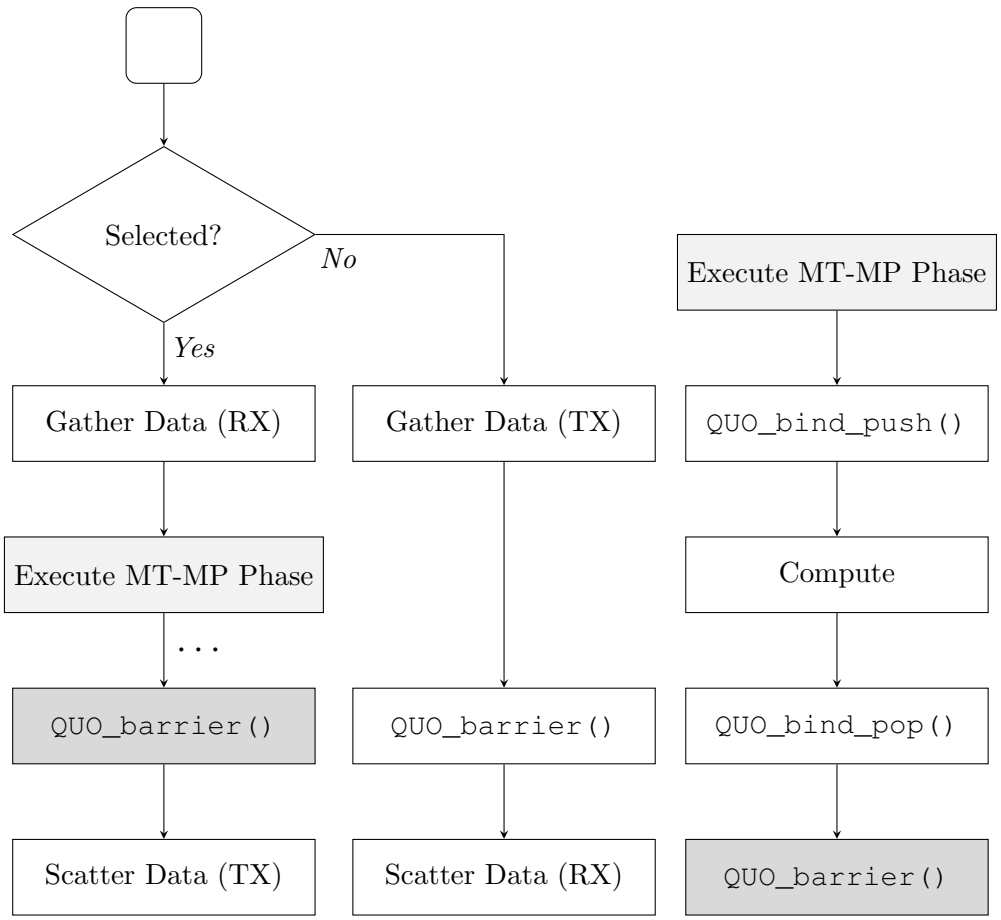


Figure 3.11: Control flow of a QUO-enabled application.

3.3.6 Parallel Task Distribution

As discussed in Section 3.3.4, QUO offers an automatic task distribution capability through `QUO_auto_distrib()`. Recall that this function aids in mapping tasks to arbitrary node-local resources such that hardware utilization and data locality for a given task set are maximized. `QUO_auto_distrib`'s algorithm is as follows.

Algorithm 1: Automatic SPMD Task Distribution

Input : A hardware resource type, τ.
The maximum number of allowable tasks per resource, m.
Output: A boolean indicating whether or not the caller was selected.
begin
  let R be the set of all available node-local resources of type τ;
  let T be the set of all tasks that share memory and a QUO context;
  let A be an $|R|$-element collection of totally ordered task sets $(X, \prec)$, where $\prec$ is defined by the sequence of insertions into X;
  let u be the caller's unique node-local task identifier in $\{0, 1, \ldots, |T| - 1\}$;
  for $r \in R$ **do**
    for $t \in T$ **do**
      // If t has affinity to r, add it to r's task set.
      if t/r **then let** $A_r \leftarrow A_r \cup \{t\}$;
  let $I \leftarrow \bigcap_{r \in R} A_r$ *// Calculate task set intersection.*
  // If task hardware affinities overlap completely.
  if $|I| = |T|$ **then**
    // Allow if a distribution slot exists for the caller.
    if $u < m \cdot |R|$ **then return true** ;
  // Prevent if caller shares affinity with other tasks.
  else if $u \in I$ **then return false** ;
  // Either $I = \emptyset$ or the caller was not a member of $I \neq \emptyset$.
  else
    for $a \in A$ **do**
      // Select if in task set with an index less than m.
      if $u \in a$ ***and*** $i(a_u) < m$ **then return true** ;
  return false;

3.3.7 Node-Level Process Quiescence

To make our approach maximally effective, there must be a portable and efficient mechanism for *quiescing* sets of MPI processes to yield their use of compute resources to *make room* for more threads of execution, as detailed in Table 3.1 and illustrated in Figure 3.10. A naive approach might use MPI-provided facilities such as an `MPI_Barrier()` across a sub-communicator containing only processes that may communicate over shared memory, for example, a sub-communicator created by calling `MPI_Comm_split_type()` with `MPI_COMM_TYPE_SHARED` as its split type. While this approach certainly works, as demonstrated in an early implementation of QUO, it introduces prohibitively high overheads and is therefore unusable in practice (an analysis of process-quiescence-induced application overhead is presented in Section 5.1.2). Instead, we employ an efficient, portable approach for compute-node-level process quiescence via `QUO_barrier()`. Its underlying machinery is straightforward:

1. At `QUO_create()` a shared-memory segment is created by one MPI process and then attached to by all processes P that are (a) members of the initializing communicator and (b) capable of communicating over shared memory.

2. A `pthread_barrier_t` is embedded into the shared-memory segment with an attribute that allows all processes with access to the shared-memory segment to operate on it. Finally, its count parameter is set to the number of MPI processes that must call `pthread_barrier_wait()` to complete the barrier, i.e., the number of processes in P.

3.3.8 Policy Management

Policies that influence how logical (software) affinities are mapped to hardware resources may be managed with QUO in a variety of ways. In a *caller-driven* approach,

the caller modifies the callee's runtime environment and assumes responsibility for resource selection (the computational resources to be used by a particular computational phase), MPI process selection (the set of MPI processes that will use the selected resources), and process affinity selection (*pushing* and *popping* binding policies as the target library's computational phases are entered and exited, respectively). A caller-driven approach is appropriate when using off-the-shelf threaded libraries that are difficult or impossible to modify at the source code level. Caller-driven approaches require the caller to be cognizant of the inner workings of the target library to make informed policy decisions. An example caller-driven policy is provided in Listing A.1. In contrast, *callee-driven* policies are encapsulated within called libraries such that the caller may be oblivious to policy decisions made by the libraries it uses, as shown in Listing A.2. Because these policies are directly embedded in the target library and are under developer control, they can be tailored precisely to the library's implementation and runtime requirements.

3.4 Quo Performance and Effectiveness

Our performance evaluation is designed to show performance and scaling characteristics for full applications. We integrate Quo into three production-quality parallel scientific applications using a variety of parallelization strategies. With these, we measure and analyze Quo's costs and benefits, and how these vary with scale. Integrating Quo into different codes demonstrates the generality of the Quo approach.

3.4.1 Experimental Setup

Performance results were gathered from the following systems located at Los Alamos National Laboratory. Data were collected during regular operating hours, so the

systems were servicing other workloads alongside the performance evaluation runs. A summary of architectural details is provided in Table 3.2.

Cielo is a 96-cabinet Cray XE6 system with a 16×12×24 (XYZ) three-dimensional torus topology built from Gemini ASICs that provide two NICs and a 48-port router [9]. Each Gemini connects two 16-core nodes (dual eight-core AMD Opteron Magny-Cours clocked at 2.4 GHz with two NUMA domains per socket), each equipped with 32 GB of memory. This system has 8,944 compute nodes, totaling 143,104 compute cores, and 272 six-core AMD Opteron Istanbul service nodes. Compute nodes run Cray Linux Environment (CLE), a Linux-based operating system.

Wolf is a 616-node system connected by a Qlogic Infiniband (IB) Quad Data Rate (QDR) network in a fat tree topology. Each 16-core compute node (dual eight-core Intel Xeon Sandy Bridge E5–2670 processors clocked at 2.6 GHz with one NUMA domain per socket) has a total of 64 GB of memory. Compute nodes run Clustered High Availability Operating System (CHAOS), a Red Hat Linux derivative [24].

Darwin is a cluster comprising different compute node architectures running CentOS 7, a Red Hat Linux derivative. Our experiments were conducted on 20-core compute nodes (2× 10-core 2.6 GHz Intel Xeon E5–2660 v3 processors) equipped with 128 GB of memory, each connected by 10 Gbit Ethernet.

System	CPU	Sockets Per Node	NUMA Per Socket	Cores Per Socket
Cielo	AMD 6136	2	2	8
Wolf	Intel E5–2670	2	1	8
Darwin	Intel E5–2660	2	1	10

Table 3.2: An overview of compute node architectures used for this study.

3.4.2 Application Results: Evaluating Quo's Effectiveness

Tables 3.3 and 3.4 detail the three Quo-enabled parallel scientific applications using all the supported language bindings (C, C++, and Fortran) and a diversity of parallelization strategies, workloads, and software environments. All application configurations represent real workloads to showcase different application communication and computation characteristics.

We evaluated Quo's effectiveness at increasing resource utilization with comparisons against a baseline (without Quo) that under-subscribes compute nodes such that the computational phase with the highest degree of threading per process ($t_{\max}$)

Identifier	Application Description
2MESH	LANL-X is an application used at Los Alamos National Laboratory comprising two libraries L0 and L1. L0 simulates one type of physics on an adaptive structured mesh and L1 simulates a different physics on a separate, structured mesh. L0 phases are MPI-everywhere and L1 phases are MPI+OpenMP.
RAGE	xRage+inlinlte: xRage is a multi-physics application used in a variety of high-deformation flow problems. xRage solves the Euler equations of conservation of mass, momentum, and energy on an adaptive structured mesh. All of the other physics are coupled through added terms to the conservation of momentum and energy equations. inlinlte solves for atomic populations in cases not in local thermodynamic equilibrium (LTE). xRage phases are parallelized using MPI-everywhere, while inlinlte phases are multithreaded with Kokkos [38].
ESMD	MD+Analysis: ESPResSo [13] is a molecular dynamics (MD) program for coarse-grained soft matter applications. Its analysis routines typically calculate observables (functions of the current system state). MD phases are MPI-everywhere and analysis phases are MPI+OpenMP.

Table 3.3: Application identifiers and descriptions of the applications they represent.

ID	Quo Version (Language Binding)	Compiler	MPI Implementation	System
2MESH	Quo 1.2.4 (Fortran)	Intel 15.0.4	Cray MPICH 7.0.1	Cielo
RAGE	Quo 1.2.9 (C)	Intel 16.0.3	Open MPI 1.6.5	Wolf
ESMD	Quo 1.3-alpha (C++)	GCC 4.9.3	Open MPI 1.10.3	Darwin

Table 3.4: Target applications and their environments used for this study.

has one PE per thread. This baseline represents the previous, long-standing mode for production runs of these applications. Table 3.5 shows the application configurations. For baseline experiments, MPI processes are launched with a static process binding policy set by either aprun (Cray MPICH) or orterun (Open MPI). For example, 2MESH is launched with four MPI processes per node (one process per NUMA domain), each with a NUMA binding policy. In contrast, Quo-enabled experiments fully subscribe resources at startup such that each MPI process is bound to a single core (by the parallel launcher) and MPI+X configuration policies are enacted dynamically using Quo.

Identifier	MPI+X Process Binding Policy	Processes Per Resource	$t_{\max}$
2MESH-W	NUMA	1/NUMA	4
RAGE-W	Machine	1/Machine	16
ESMD-W	Socket	10/Socket	2
2MESH-S	NUMA	1/NUMA	4
RAGE-S	Machine	1/Machine	16
ESMD-S	Socket	5/Socket	4

Table 3.5: Application configurations used in this study.

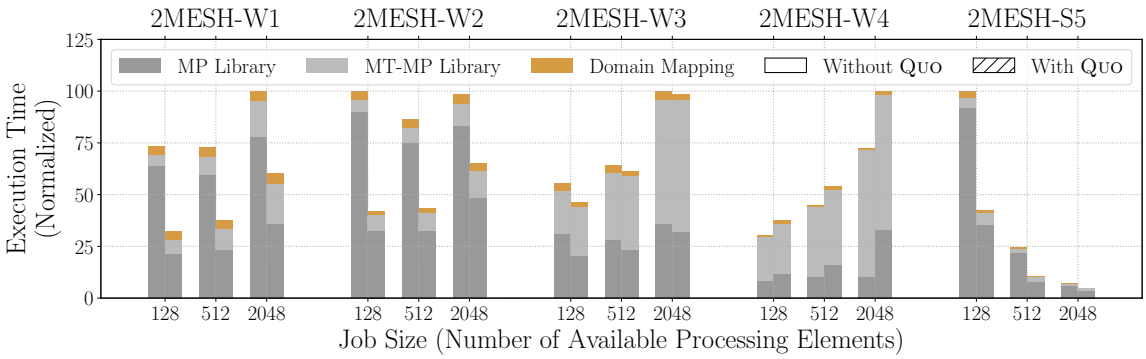
Quo Performance Results

We evaluated the three Quo-enabled applications on three different platforms at scales up to 2,048 PEs (and processes).[2] Figure 3.12 shows all the application performance and scaling results: 30 sets of experiments, ten different application/workload combinations, each executed at three different scales. Quo's effectiveness is determined principally by two criteria: 1. how much of an application's overall runtime is dominated by under-subscribed computational phases and 2. how well these otherwise under-subscribed computational phases strong-scale at full node utilization.

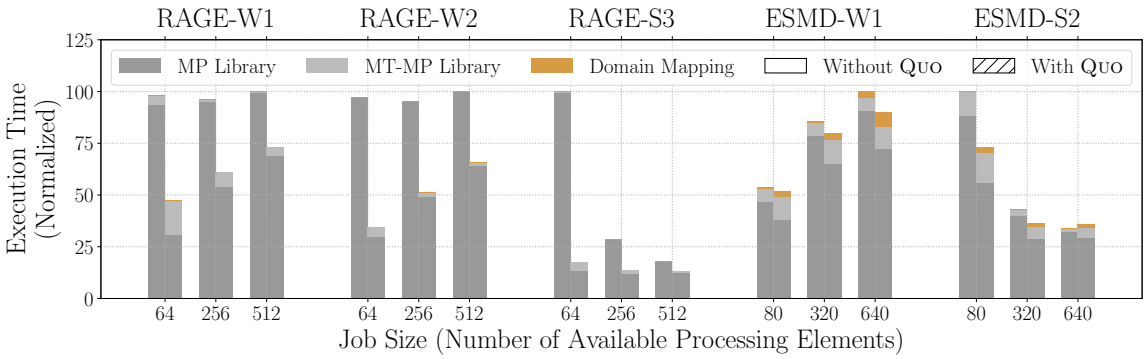
The overall average speedup across all 30 Quo-enabled workloads was ∼70 %. Of these workloads, 26 show an overall speedup when using Quo, with more than half the cases (16) yielding speedups greater than 50 %. RAGE-S3 yields a maximum Quo-enabled speedup of 476 % at 64 PEs, and seven other workload configurations showed a speedup of greater than 100 %. The reason these workloads realize huge benefits when dynamically configured using Quo is that their otherwise under-scribed computational phase (in this case the MPI-everywhere phase) strong-scales well with the given realistic input sets.

Four of the Quo-enabled workloads yield modest speedups (less than 10 %) and four other cases in fact demonstrated slowdowns (ESMD-S2–640, 2MESH-W4–128, 2MESH-W4–512, and 2MESH-W4–512). There are three main reasons for this: 1. as previously mentioned, if the under-subscribed phase does not strong-scale well, Quo's approach will not yield a significant performance boost; 2. Quo can increase the costs of data domain remappings; and 3. in some cases, Quo appears to add some overhead to the multithreaded computational phase. These phenomena can be observed in Figure 3.12a and Figure 3.12b.

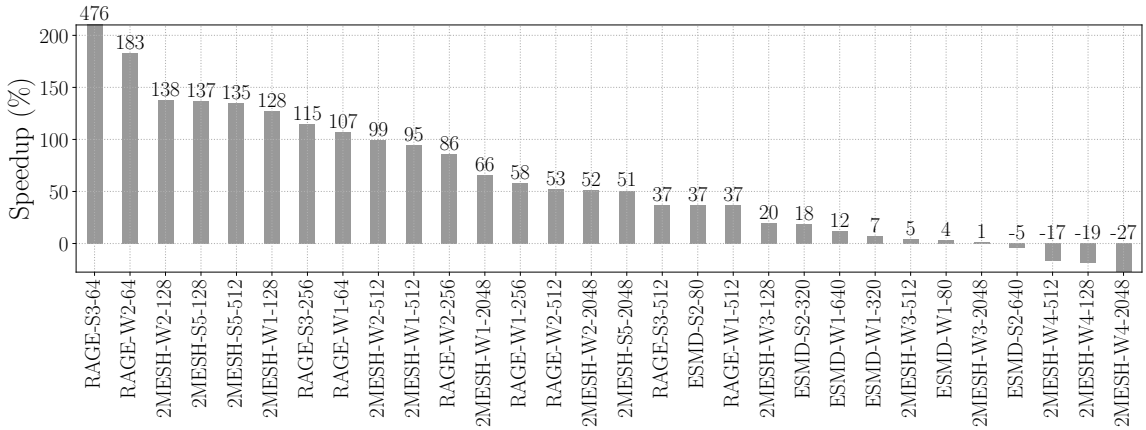
[2]The seemingly strange PE counts (80, 320, 640) in the ESPResSo experiments are from runs on a system with 40 hardware threads per node (2 hardware threads per core).



(a) 2MESH results without and with Quo.



(b) RAGE and ESMD results without and with Quo.



(c) Quo-enabled speedups: Quo versus a standard under-subscribed baseline.

Figure 3.12: Application results without and with Quo.

3.5 Practical Considerations

As previously described, commonly used parallel application launchers only provide for static, whole-application binding policies, or none at all, and each has its syntax for command-line or configuration-file specification of binding policy. Using Quo one does not need to specify binding policies via the job launcher: Quo can completely specify and manage resource bindings efficiently and dynamically. While Quo is simple, efficient, effective, and convenient, it does introduce some practical considerations and complexities:

- Increased Code Complexity: With the quiescing and later resumption of tasks, application data remappings across library domains may increase. Either the library developer or the library user must be prepared to deal with this added programming burden. We posit that in a well-engineered library such complexity is manageable.

- Encapsulating Dynamically Driven Code Regions: When using the Quo approach one must identify and surround computationally-intensive code regions with calls to `QUO_bind_push()` and `QUO_bind_pop()`. Again, in a well-engineered library these modifications should be trivial—ordinarily, such code regions are well-bound by a function call or loop body.

- Determining Minimum Threading Levels: Though not brought by the use of our methodology, determining the minimum required threading level at `MPI_Init_thread()` can be challenging in a dynamic multi-library environment. That is, a threaded library may only execute under certain circumstances that are not necessarily evident at MPI initialization time, for example, at runtime requiring a new physics capability. Blindly initializing with the highest level of thread safety (that is, `MPI_THREAD_MULTIPLE`) is wasteful because of

performance degradation brought by higher degrees of required critical section protection in an MPI library [93].

- Conflicting Runtime Affinity Policies: actions taken by OpenMP runtimes concerning their static affinity policies can be counterproductive in dynamic MPI+OpenMP environments such as the one we have presented. For example, we have encountered instances where setting a `KMP_AFFINITY` policy, a static mechanism used to bind OpenMP worker threads to specific hardware resources, for Quo-enabled applications has degraded their performance in OpenMP-enabled regions by $\sim$30$\times$ on Intel Knights Landing architectures when compared to a comparable configuration not using Quo. While we have not yet determined the cause for this slowdown, manually disabling `KMP_AFFINITY` alleviates the performance degradation.

3.6 Summary

We have presented a novel approach and implementation for accommodating thread-level heterogeneity in coupled MPI applications. Our approach, Quo, enables full utilization of all available compute resources throughout an application's entire execution. Significant performance improvements are achievable when used in environments positioned to make effective use of the additional levels of parallelism our strong-scaling approach enables. Our performance results show that for a majority of the 30 tested workloads, using Quo renders speedups greater than 50 %, and the best case speedup was a resounding 476 %.

Quo's interface is programmable, meaning that it can be used preferentially in cases where it will improve performance and not used otherwise in favor of a conventional static policy. Better yet, a graded approach could be used wherein only that subset of libraries that benefit from strong-scaling are strong-scaled, and to the

optimal degree within the available bounds. This, in turn, implies that the decision to actively use Quo, and the strong-scaling factors used when it is, could be made dynamically, but we have not yet explored this possibility.

One important aspect of analysis remains, namely the precise measurement of Quo runtime and memory overheads. Before delving into this examination, however, we must first address *the memory attribution problem*, as our solution enables analyses pertinent to the study presented in Section 5.2.

Chapter 4

Addressing The Memory Attribution Problem

In HPC, parallel programming systems and applications are evolving to improve performance and energy efficiency, particularly as systems scale to higher degrees of intra- and inter-node parallelism. As the number of processing elements in these systems continues to grow, memory capacity to core ratios tend to remain constant or shrink. The data in Table 4.1 show the ratios of memory capacity to compute core count for the last ten top-ranked systems of the TOP500 [97]. Of these systems, only three have a *memory-to-core* ratio (in GB:core) of at least 2:1—the first was 4:1 in 2002 and the last was 2:1 in 2011—with over half of the remaining seven systems having less than 1 GB of memory per core. This decrease has been pushing many applications toward memory-capacity-bound computing regimes. In these cases, developers will increasingly rely on understanding how the supporting software infrastructure (i.e., operating system (OS), software libraries, and middleware) affects overall application memory efficiency along three major axes: runtime, job size (*scale*), and workload.

	System	# Cores	Memory	Memory/Core
Jun. 2002	Earth Simulator	5,120	20,480 GB	4.00 GB
Nov. 2004	Blue Gene/L	32,768	8,192 GB	0.25 GB
Jun. 2008	Roadrunner	122,400	106,086 GB	0.87 GB
Nov. 2009	Jaguar	298,592	598,016 GB	2.00 GB
Nov. 2010	Tianhe-IA	186,368	229,376 GB	1.23 GB
Jun. 2011	K Computer	705,024	1,410,048 GB	2.00 GB
Jun. 2012	Sequoia	1,572,864	1,572,864 GB	1.00 GB
Nov. 2012	Titan	560,640	710,144 GB	1.27 GB
Jun. 2013	Tianhe-2	3,120,000	1,024,000 GB	0.33 GB
Jun. 2016	Sunway TaihuLight	10,649,600	1,310,720 GB	0.12 GB

Table 4.1: Hardware statistics of the last 10 number one computer systems according to the TOP500 by earliest date of first-place ranking.

HPC application developers commonly couple the high-level application driver code, software components that drive the use of lower-level parallel programming systems, with supporting software such as a message passing library, resulting in a single executable after linking. Such couplings can make it difficult to accurately attribute an application's memory usage across the full set of software components. For example, we may not be able to accurately answer questions such as: *What is the message passing library's contribution to my application's overall memory footprint?* In general, this *memory attribution problem* arises when an application developer cannot predict or evaluate during runtime where the available memory is used across the software stack comprising the application, software libraries, and supporting runtime architecture needed to enable the application at a given scale, under a given workload, and in a time- and space-sharing scheduled environment.

In summary, improving application memory efficiency is becoming increasingly important in the development, deployment, and upkeep of parallel and distributed

programs, but is complicated by concurrent instances of coupled software components dynamically consuming memory resources over time. At the same time, there is a lack of parallel tools capable of extracting the relevant metrics to solve the memory attribution problem. In this work, we address the memory attribution problem in parallel and distributed message-passing software systems as follows.

- We propose an approach for accurate, per-process quantification of memory resource usage over time that is able to clearly distinguish between application and MPI library usage. Our experimental results show that job size, communication workload, and hardware/software architecture can influence peak runtime memory usage.
- We develop a corresponding open-source profiling library named *memnesia* [48] for applications using any implementation of MPI. We develop this software with a specific goal in mind: once memory attribution is better understood, applications and MPI implementations will potentially be able to improve or maintain their memory utilization as they are developed and maintained.
- We evaluate our profiler's runtime overhead and behavior using both micro-benchmarks and proxy applications, concluding with an analysis of memnesia's memory overhead and perturbation.

To the best of our knowledge this is the first work to both implement and evaluate such an approach for parallel and distributed software systems.

4.1 Background

In the first half of this section, we discuss application memory utilization in the context of parallel message-passing programs, and then go on to discuss techniques

in parallel application analysis, where we summarize well-known taxonomies that categorize tools along four axes. The last half of this section describes key approaches, mechanisms, and system software infrastructure used by our memory usage profiler.

4.1.1 Parallel Application Memory Utilization

Application memory utilization is concerned with application memory usage and often focuses on studying dynamic heap behavior. In this context, an application's *memory footprint*, the minimum memory capacity required to complete its calculation successfully, is the aggregate of the *application driver footprint* and each of the middleware and runtime library footprints. The application driver implements the numerical methods that underlie a particular system model or simulation, while the middleware and runtime services coordinate the execution of parallel (and potentially distributed) process instances. An application driver's footprint is primarily influenced by 1. its underlying numerical methods, 2. how those methods are implemented (e.g., data structures, parallelization strategies), and 3. the size and fidelity of its computational domain. Message passing libraries such as Open MPI [40] and MPICH [46] are examples of message-passing middleware. Like the application drivers they support, they consume memory to maintain their internal state, which is primarily influenced by how they are driven with respect to job size (e.g., the size of `MPI_COMM_WORLD`) and communication workload.

4.1.2 Parallel Application Analysis

Parallel and distributed tools that provide insight into application behavior are important for the development, deployment, and upkeep of parallel programs. Developing such tools is challenging because data collection and analysis is usually distributed across a set of computational resources, requiring that their resulting

outputs be aggregated for further analysis. Tools may be categorized by

- *Functionality* (correctness or performance): Correctness tools aid in identifying application (algorithmic) correctness bugs, whereas performance tools aid in identifying performance bugs.
- *Instrumentation methodology* (dynamic or static): Dynamic tools generally operate on unmodified application binaries and use facilities such as *ptrace* [4] to observe and control the execution of application processes. In contrast, static tools insert instrumentation instructions such as *probes* into applications during preprocessing-, compilation-, or link-time transformations.
- *Measurement methodology* (event tracing, event sampling): Event tracing gathers data by activating a set of instrumentation probes at every event associated with a trace, for example, *function interposing*, whereas sampling-based measurements are typically interrupt-driven and provide only a statistical view of application behavior, e.g., program counter sampling.
- *Interactivity* (online, offline): Online analysis tools are interactive and meant to be influenced at run time by an end user during data collection and analysis phases. Offline analysis tools, in contrast, are generally more static, meaning that the tool is started with and runs alongside an application until termination, then tool data are written, post-processed, and finally analyzed by other programs. This approach, while popular in practice because of its simplicity, tends to scale poorly because of high data storage and analysis costs [86].

4.1.3 Intercepting Application Behavior

Function interposition is a powerful technique used to insert arbitrary code between a caller and its intended callee [33, 92]. For compiled languages this is typically

achieved by *function symbol overloading*, where a duplicate function definition is introduced into an application such that the duplicate entry's symbol is resolved ahead of the intended callee's, with the consequence that its code is executed instead. This technique is well known and widely used to instrument dynamically linked libraries because probes can be introduced into unmodified binaries via the runtime loader, which is typically achieved by using `LD_PRELOAD`.

The MPI profiling interface (PMPI) provides a straightforward and portable mechanism for intercepting all MPI-defined functions [78]. Specifically, the MPI specification requires that libraries provide an alternate entry point, achieved through a *name shift*, which can be used for tooling purposes. Listing 4.1 shows an example of how a tool might intercept application calls to `MPI_Barrier()` using PMPI and ultimately function interposing.

Listing 4.1: MPI profiling interface example.

```
// For MPI_Comm type definition.
#include "mpi.h"

int MPI_Barrier(MPI_Comm comm) {
    // Tool code before barrier.
    ...
    // Execute MPI barrier.
    int rc = PMPI_Barrier(comm);
    // Tool code after barrier.
    ...
    return rc;
}
```

4.1.4 Collecting Process/System Information

The *proc pseudo file system* (*procfs*) offers a convenient interface for obtaining information about and influencing the state of a running OS kernel [3]. *procfs* provides

user-space access to kernel-maintained state by exposing a file-based access semantics to the structure hierarchy it maintains (*directories* and *files*). Obtaining information about current OS state, including that of active processes, is accomplished by opening and parsing *files* located below *procfs*'s mount point (typically `/proc`). In many cases the content of these special files is generated dynamically to provide an updated view of the operating system's state.

In Linux, `/proc/[pid]/smaps` (*smaps*) shows memory consumption for each of the process's mappings [3]. Each *smaps* entry can be thought of as having two pieces: a *header* and a *body*. The header contains address range occupancy, access permission, and (if applicable) backing store information, while the body contains memory map statistics, including resident set size (RSS) and proportional set size (PSS). The RSS represents how much of the mapping is currently *resident* in RAM, including shared pages. In contrast, PSS represents a process's share of the mapping, meaning that, for example, if a process has 100 private pages and additionally shares 100 more with another process, then its PSS is 150 (i.e., $100 + 100/2$). A process's RSS and PSS will change during run time and are both influenced by the amount of process-driven memory pressure exerted on the system.

4.2 Methods in Memory Utilization Analysis

In this section, we begin with an examination of related work in *memory utilization analysis*, describing how contemporary approaches address the previously described memory attribution problem. We then describe our approach and its corresponding open-source implementation, memnesia.

4.2.1 Heap Profiling and Memory Map Analysis

Heap profiling identifies and gathers statistics about call paths containing dynamic memory management calls, for example, `malloc()` and `free()`. Notable heap profilers include Valgrind Massif [80], Google heap profiler (GHP) [45], and memP [28]. GHP and memP work by using `LD_PRELOAD` and function overloading of memory management calls. This approach to heap profiling has limitations: 1. it does not work on statically linked binaries, 2. it does not allow a user to distinguish between memory pages mapped into a process's address space and memory pages that are resident in physical memory, 3. it multiply counts shared memory pages and does not allow a user to determine which cooperating process *owns* the page, and 4. it does not allow a user to distinguish application driver memory usage from runtime/middleware memory usage.

Memory map analysis collects information about a running application by inspecting application-specific entries in *procfs*. This approach is appealing for a variety of reasons. First, it is relatively straightforward to implement, avoiding complications brought by user-level interception of memory management functions (e.g., some memory management calls cannot be traced, for example when `glibc` calls `__mmap()` [102]) or virtualization (e.g., processes run in isolated virtual environments do not adequately emulate OS-specific management schemes regarding shared pages). Second, when compared to heap profiling alone, it can provide a more holistic view into important features that ultimately impact a process's actual memory footprint, namely the size and count of private/shared pages and their occupancy in RAM. Finally, it is language-agnostic and therefore readily applicable to any running process. As an example, smem [95] is a standalone tool capable of generating a variety of whole-system memory usage reports based on the PSS metric. Like memnesia, smem uses memory map analysis of *smaps* for usage reporting but is not a parallel tracing tool.

4.2.2 Middleware Attribution of Memory Usage

As previously described, determining how much memory the message-passing library consumes is challenging and becoming increasingly important in the development, upkeep, and deployment of parallel programs. Current approaches for MPI library memory attribution generally can be categorized as library-specific instrumentation or benchmark-driven library analysis. An example of the former, *craymem*, can be found in Cray's implementation of MPICH [83], where through environmental controls, internal memory monitoring statistics can be accessed via textual output (either to a terminal or a file.). Such library-specific approaches are implementation-dependent and often provide coarse-grained output. For example, the output

```
# MPICH_MEMORY: Max memory allocated by malloc: 711424 bytes by rank 0
# MPICH_MEMORY: Min memory allocated by malloc: 710024 bytes by rank 1
# MPICH_MEMORY: Max memory allocated by mmap: 83072 bytes by rank 0
# MPICH_MEMORY: Min memory allocated by mmap: 83072 bytes by rank 0
# MPICH_MEMORY: Max memory allocated by shmget: 159307008 bytes by rank 0
# MPICH_MEMORY: Min memory allocated by shmget: 0 bytes by rank 1
# [0] Max memory allocated by malloc:711424 bytes
# [0] Max memory allocated by mmap: 83072 bytes
# [0] Max memory allocated by shmget: 159307008 bytes
# [1] Max memory allocated by malloc: 710024 bytes
# [1] Max memory allocated by mmap: 83072 bytes
# [1] Max memory allocated by shmget: 0 bytes
```

does not allow a user to determine when during the program's execution memory usage *high-water marks*—the maximum recorded values—were reached or whether these maxima were transient or sustained for long periods of time.

As an example of the latter, mpimemu [47] provides benchmark-driven memory attribution for MPI implementations. mpimemu is an MPI program with built-in memory map monitoring that works by sampling `/proc/self/status` and `/proc/meminfo`, while also imposing a scalable communication workload on the system. Runtime memory attribution is approximated by calculating usage deltas between samples collected during its execution and those collected before the MPI

library was initialized. This approach works for understanding coarse-grained application and workload features captured in the given benchmarks, but does not provide any insight into how a given MPI library's memory usage is affected when driven by a specific application or set of applications.

4.2.3 Our Approach

We present an event-driven-analysis approach for accurately capturing both application- and message-passing-library-specific memory usage of parallel and distributed message-passing programs. As shown in Table 4.2, our approach overcomes virtually all of the shortcomings of previous approaches. While our approach generalizes to any MP system, our reference C++ implementation, memnesia, relies on OS support for certain *procfs* features (i.e., *smaps*) and C entry points into the MPI library.

	Language Independent	Linkage Agnostic	Residency Aware	Residency Aware (Proportional)	Component Aware	Temporal
GHP	—	—	—	—	—	—
memP	—	—	—	—	—	—
Massif	—	—	—	—	—	✓
smem	✓	✓	✓	✓	—	—
craymem	✓	✓	—	—	—	—
mpimemu	—	✓	✓	—	—	—
memnesia	✓	—	✓	✓	✓	✓

Table 4.2: Tools and their respective attributes.

Application Instrumentation and Data Collection

We implement our application instrumentation as a runtime system compiled into a shared library that is loaded into target application binaries at startup via the runtime loader. In practice this is accomplished via environmental controls: before application startup, `LD_PRELOAD` is set to include the memnesia runtime, then the application is launched as usual. The application drives data collection through its use of MPI. Each entry point into the message passing library becomes an instrumentation point at which we execute tool code between the caller and the intended callee using function interposition. At each instrumentation point, the memnesia runtime places *calipers*—a pair of instrumentation probes—around the target function such that *smaps* data are collected immediately before and after callee execution, as shown in Listing 4.2. Tool data are stored in per-process, in-memory caches maintained by the memnesia runtime through parallel data aggregation. On program completion, memory analysis data are written to disk, as shown in Figure 4.4.

Listing 4.2: Code snippet showing memnesia instrumentation of `MPI_Barrier()`.

```
int MPI_Barrier(MPI_Comm comm) {
    int rc = MPI_ERR_UNKNOWN;
    {
        // Constructor collects /proc/self/smaps sample.
        memnesia_scoped_caliper caliper(MEMNESIA_FUNC);
        // Execute barrier on behalf of the application.
        rc = PMPI_Barrier(comm);
    }
    // caliper's destructor collects another smaps sample.
    return rc;
}
```

memnesia trace data are stored in a straightforward on-disk representation made up of records containing three fields:

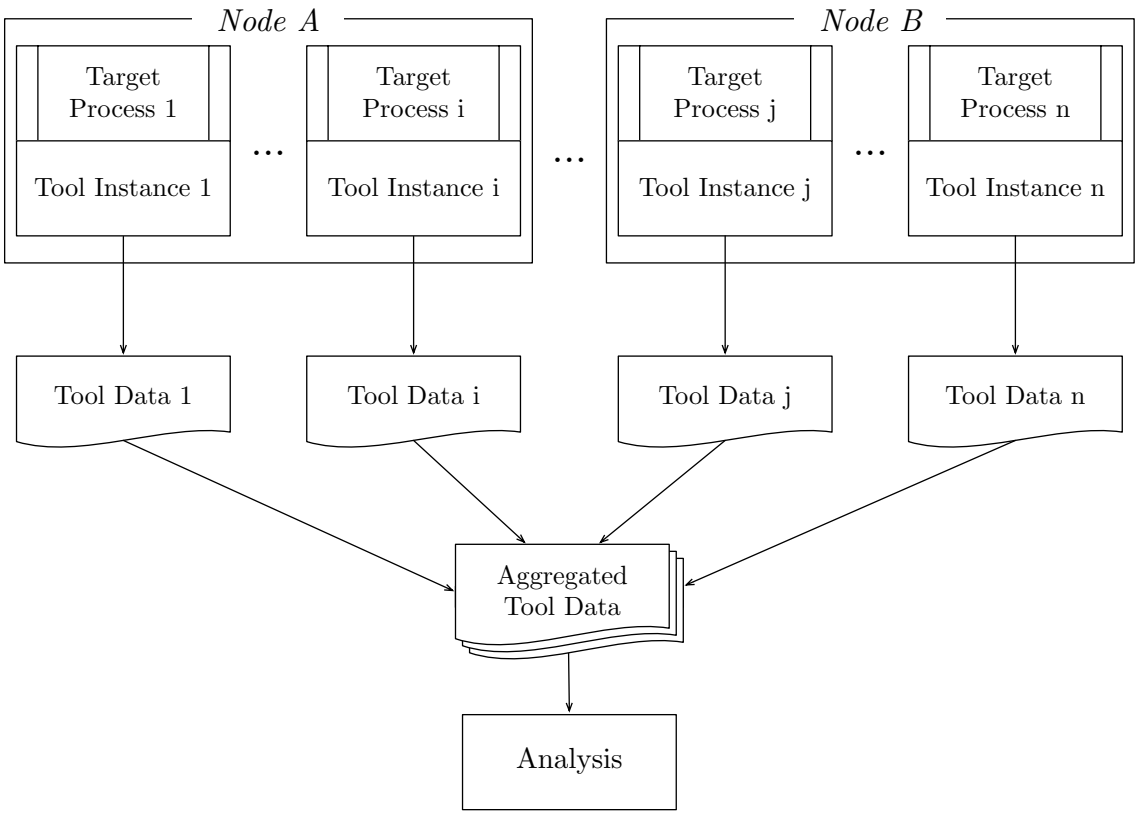


Figure 4.3: A typical offline tool architecture where analysis probes start with the application and remain in place for the entirety of the application's execution. After all analysis data are written, they are then read, aggregated, and finally analyzed by a separate tool.

- *trigger* (8-bit integer): name (ID) of the function triggering data collection.
- *time* (float): collection time relative to when the MP library was initialized.
- *usage* (float): observed memory usage calculated by summing PSS entries in *smaps*, while ignoring those associated with our instrumentation library—an enhancement included to improve the accuracy of our reported statistics.

From those data, component-level metrics can be obtained readily. Total application memory usage $m(t)$ (that of the application driver and MPI library) at time t is equal to the *smaps* usage u reported at that point, i.e., $m(t) = u$. MPI library

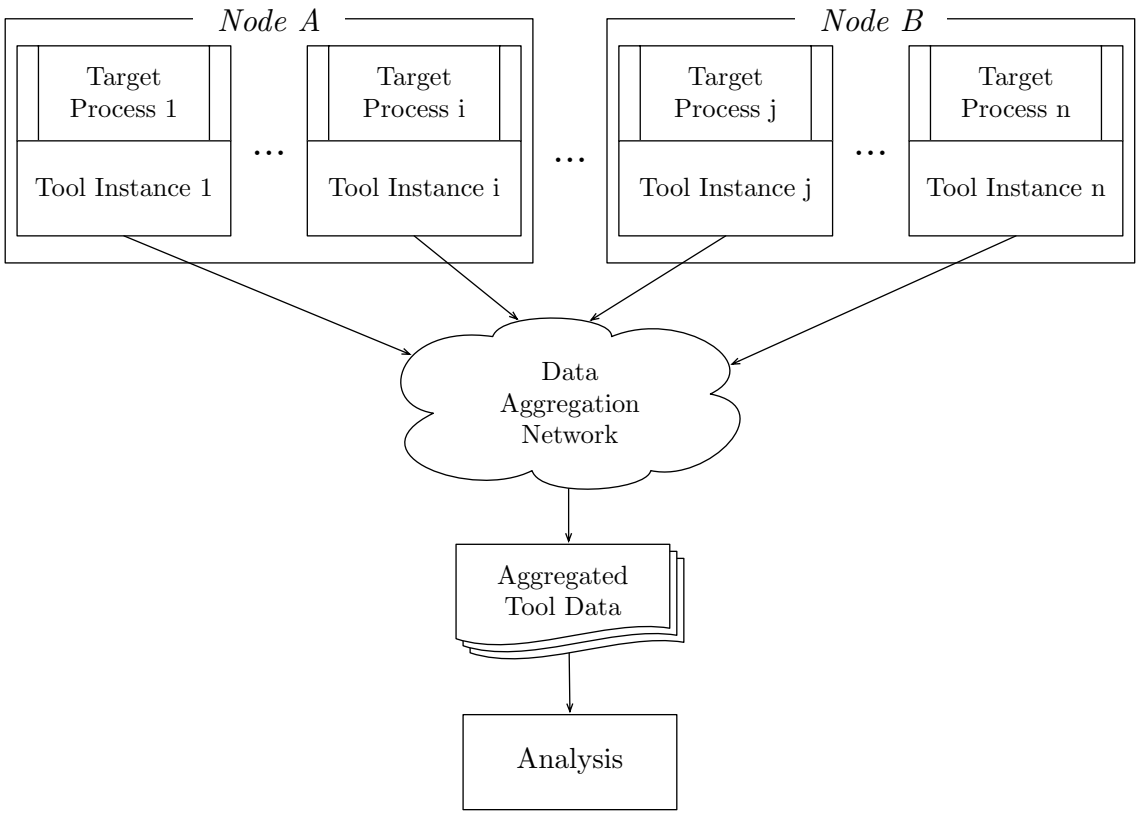


Figure 4.4: The tool architecture we adopted, which bears many similarities to its counterpart shown in Figure 4.3. The key difference is that tool data aggregation is parallelized using the job's resources with MPI.

usage $\hat{m}(t)$ at time t is determined by summing all preceding usage deltas (Equation 4.1 and Figure 4.5)—the intuition is that there is a causal relationship between MPI library calls and any observed usage deltas (positive or negative), since the MPI library was the only software component executing between data collection points. With these values, an application driver's memory can trivially be calculated as the difference between total memory usage and MPI library memory usage.

In summary, our approach overcomes virtually all the shortcomings of previous methods, though our current reference implementation has limitations: 1. memnesia requires OS support for certain *procfs* features and C entry points into the MPI

library, accessed through dynamic linkage, and 2. PSS reporting for applications that use *hugepages* [1, 2] is not currently supported.

$$\hat{m}(t_j) = \sum_{i=0}^{i<\lfloor j+1/2 \rfloor} \Delta_i,\ \Delta_i = u_{2i+1} - u_{2i} \tag{4.1}$$

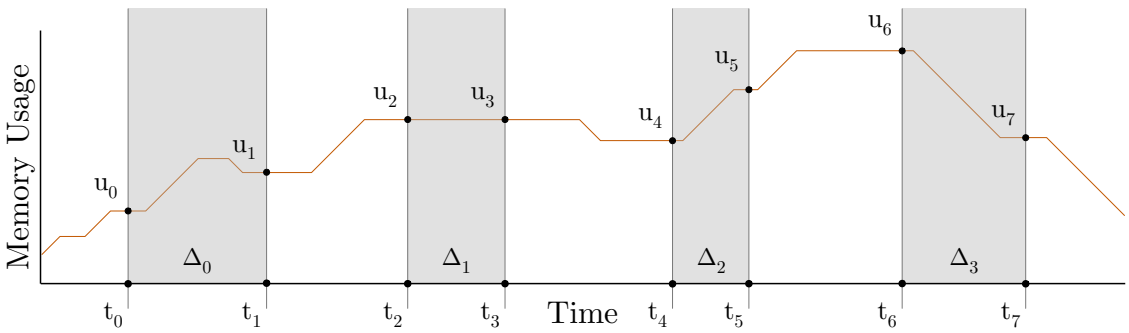


Figure 4.5: Single-process memory usage and data collection points.

4.3 Micro-Benchmarks and Proxy Applications

Computational benchmarks are commonly used to assess and compare the performance of various workloads on differing software (e.g., library, OS) and hardware (e.g., processor, network) configurations. Individual benchmarks may be designed to exhibit a minimal set of behaviors to enable precise characterization of specific hardware or software mechanisms—so-called *micro-benchmarks*. At the other end of the spectrum, the net performance characteristics of whole real-world applications on a range of inputs may be the atomic units of observation. While the latter can calibrate expectations for the applications tested, such benchmarking may be expensive in terms of resources consumed and time to result and may produce results not generalizable to other applications.

As a middle ground, *proxy applications* have become established as useful tools. The proxy is intended to be a software construction that is somehow representative of a larger application (or some components of a larger application) in terms of algorithmic structure (perhaps for the purpose of rapid prototyping) or computational and communication behavior (for benchmarking). In an effort to generalize beyond specific applications, in the context of HPC the notion of a "*computation dwarf*" has been developed. In the general sense, a dwarf is not a specific program but a closely related set of algorithmic methods with closely related computational and communication behaviors. Widely known lists of dwarfs include Colella's list of seven [32] and Asanovic et al.'s extension of that list to thirteen [14], both of which are concerned with numerically intensive scientific computing. These have inspired other such lists, for example for symbolic computing [63], but we are concerned with the former.

In the general sense the dwarfs are concepts, not computer codes, but for practical experimentation concrete programs are needed. For this work, we consider the first two of Colella's dwarfs, structured grids and unstructured grids, and their reification as the well-known *proxy applications* LULESH [65] and Kripke [70], respectively. Proxy applications serve as proxies—representatives—of full applications, again in terms of some specified properties such as problem type or runtime behavior.

4.3.1 Application Drivers: Proxy Applications

In this section, we describe the proxy applications used in our study.

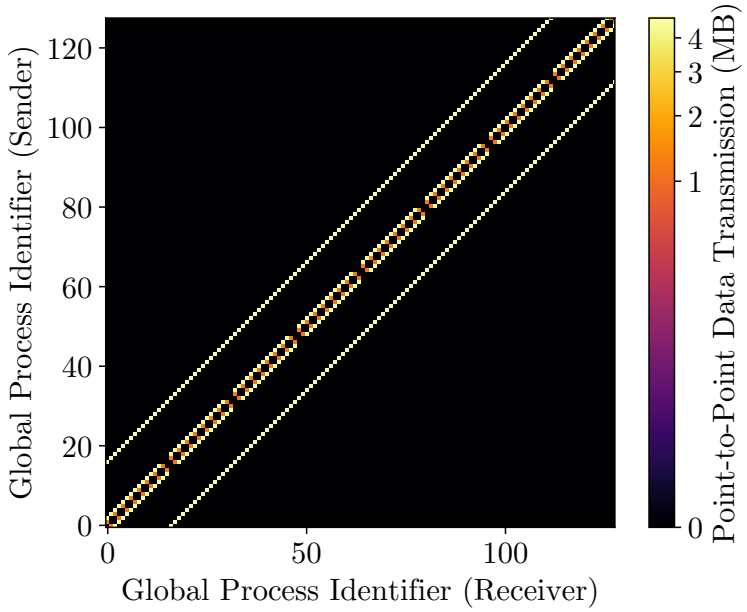
Trivial and Alltoall: Micro-benchmarks that are meant to represent extreme ends of the *in-band* (i.e., application-driven) communication spectrum. Our *Trivial* benchmark calls `MPI_Init()` and then immediately calls `MPI_Finalize()`, thereby representing the most trivial of all MPI applications: one with no communication. We study the trivial case to understand an MPI library's minimum required memory

footprint for parallel application lash-up. *Alltoall*, by contrast, is meant to represent applications that impose the most stressful (from the MPI library's perspective) communication workload: an all-to-all communication pattern where data are exchanged between every pair of processes. In particular, this program executes `MPI_Alltoall()` over `MPI_COMM_WORLD` in a loop, alternating between per-process message sizes of 2 kB and 4 MB. For each iteration of the loop, new communication buffers are allocated before the all-to-all data exchange and then freed after its completion. We use this workload to study the memory efficiency of runtime metadata structures associated with memory registration and connection management.

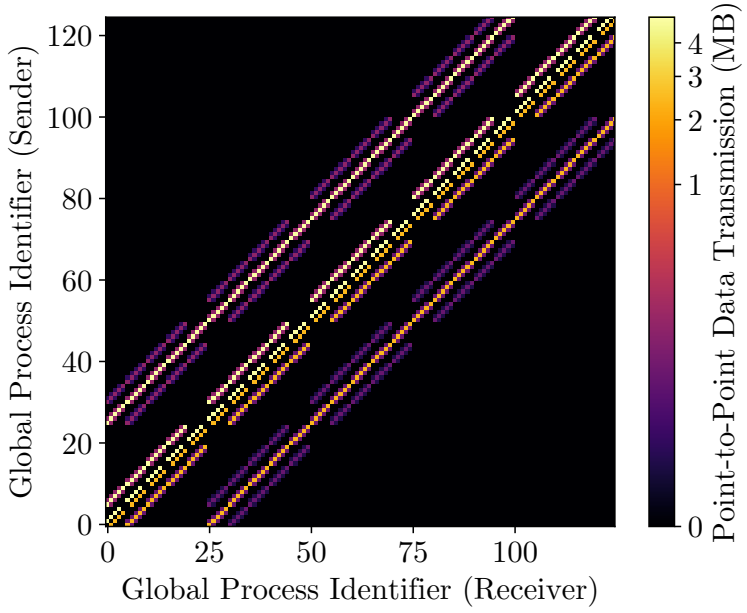
Multiple Bandwidth/Message Rate: The OSU multiple bandwidth/message rate test measures aggregate uni-directional bandwidth between multiple pairs of processes using MPI [94]. The purpose of this micro-benchmark is to quantify achieved bandwidth and message rates between a configurable number of processes concurrently imposing a communication workload on a system.

Structured Grids: Kripke: Kripke is a proxy application developed at Lawrence Livermore National Laboratory, designed to be a proxy for a fully functional discrete-ordinates (Sn) 3D deterministic particle transport code [70]. It is widely regarded as an exemplar of Colella's structured-grid computational dwarf. Figure 4.6a shows the point-to-point communication structure formed by this application.

Unstructured Grids: LULESH: LULESH is a proxy hydrodynamics code that is widely regarded as an exemplar of Colella's second computational dwarf, unstructured grids [65]. It is in fact a family of implementations specialized to various programming models and expressed in multiple programming languages. Our work uses the C++ MPI port [64]. Figure 4.6b shows the point-to-point communication structure formed by this proxy application.



(a) Kripke



(b) LULESH

Figure 4.6: Point-to-point communication structure formed by each proxy application used in this study. Colors are mapped to data transfer totals between MPI processes (send/receive pairs) using point-to-point communication operations.

4.4 Results

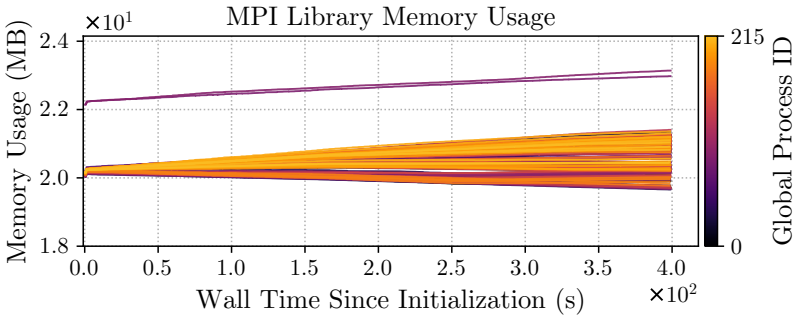
In this section, we present and discuss our results gathered using our profiling and analysis infrastructure. We first discuss our tool's capabilities and the resulting insight into how memory is allocated as a function of run time, scale, and workload. Further, we show how memnesia is able to capture features particular to a workload instance, namely those related to data structure management, message protocol, and communication pattern, all at a per-process and per-software-component level—a capability that is not readily available today through other means.

4.4.1 Experimental Setup

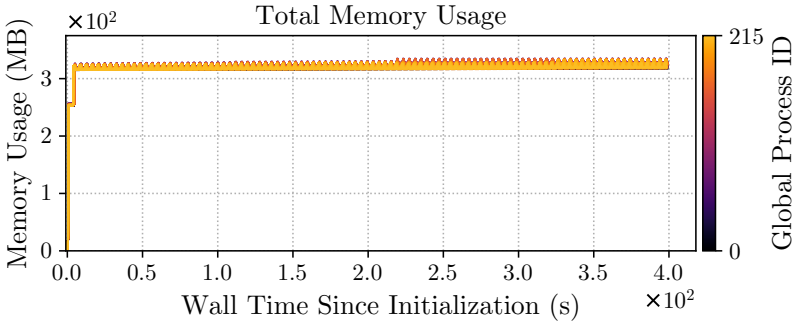
Performance results were gathered from the Trinitite and Snow systems located at Los Alamos National Laboratory, detailed in Table 4.3. Data were collected during regular operating hours, so the systems were servicing other workloads alongside, but in isolation from, ours. For each study in this section, experiments were executed in succession on a single set of dedicated hardware resources. Our experiments used weak scaling such that each process was given a fixed problem size.

4.4.2 Memory Usage Timelines

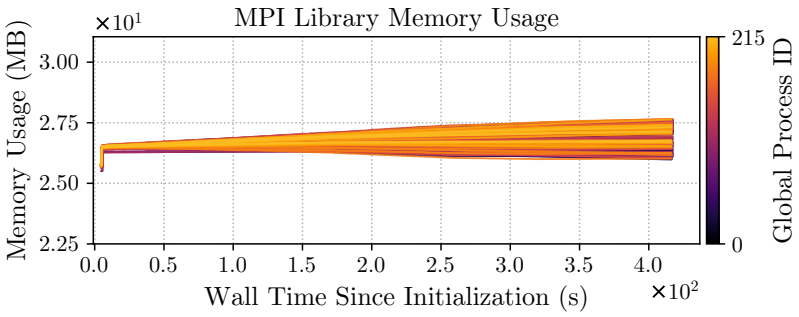
Figures 4.7 and 4.8 show per-process memory usage for two proxy applications running atop different MPI implementations on Snow. Results shown are from 100-cycle runs of LULESH (96^3 elements per process) and 50-cycle runs of Kripke (16^3 zones per process)—small-scale configurations meant to showcase our tool's analysis and reporting capabilities. The left column shows the evolution of MPI library memory usage (in isolation from the application driver's) over time and highlights how different communication substrates and workloads, shown in Figure 4.6, influence



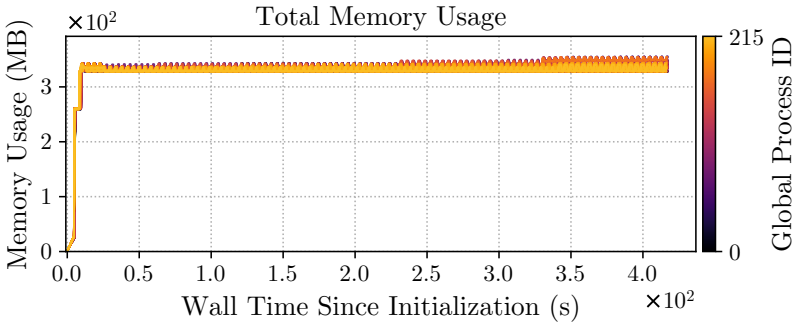
(a) LULESH (Open MPI)



(b) LULESH (Open MPI)

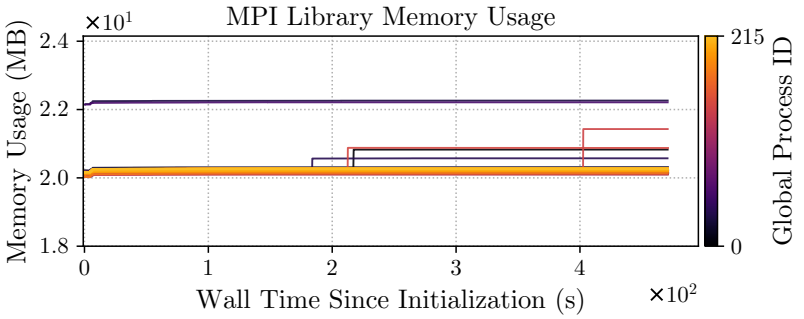


(c) LULESH (MVAPICH2)

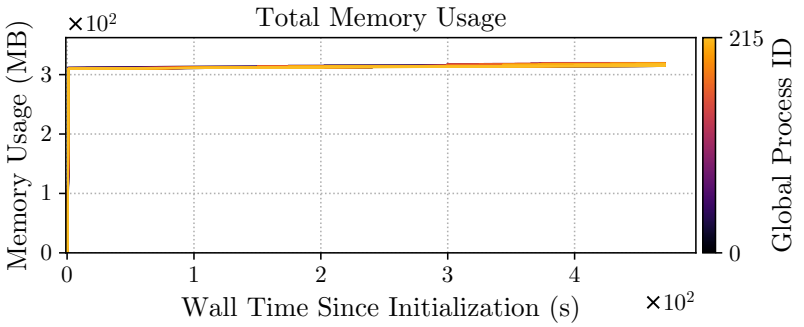


(d) LULESH (MVAPICH2)

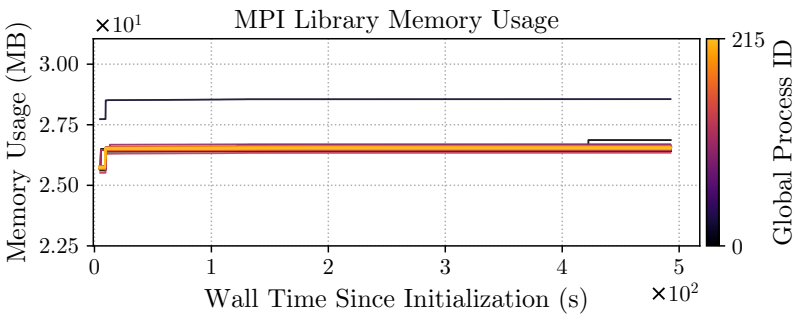
Figure 4.7: Tool output showing per-process memory usage over time for LULESH. Colors are mapped to a process's `MPI_COMM_WORLD` rank.



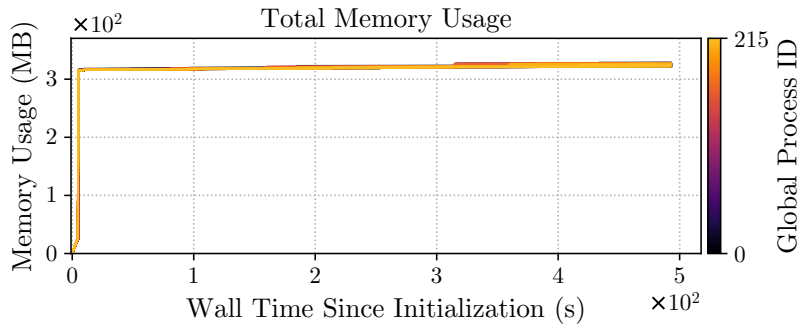
(a) Kripke (Open MPI).



(b) Kripke (Open MPI)



(c) Kripke (MVAPICH2)



(d) Kripke (MVAPICH2)

Figure 4.8: Tool output showing per-process memory usage over time for Kripke. Colors are mapped to a process's `MPI_COMM_WORLD` rank.

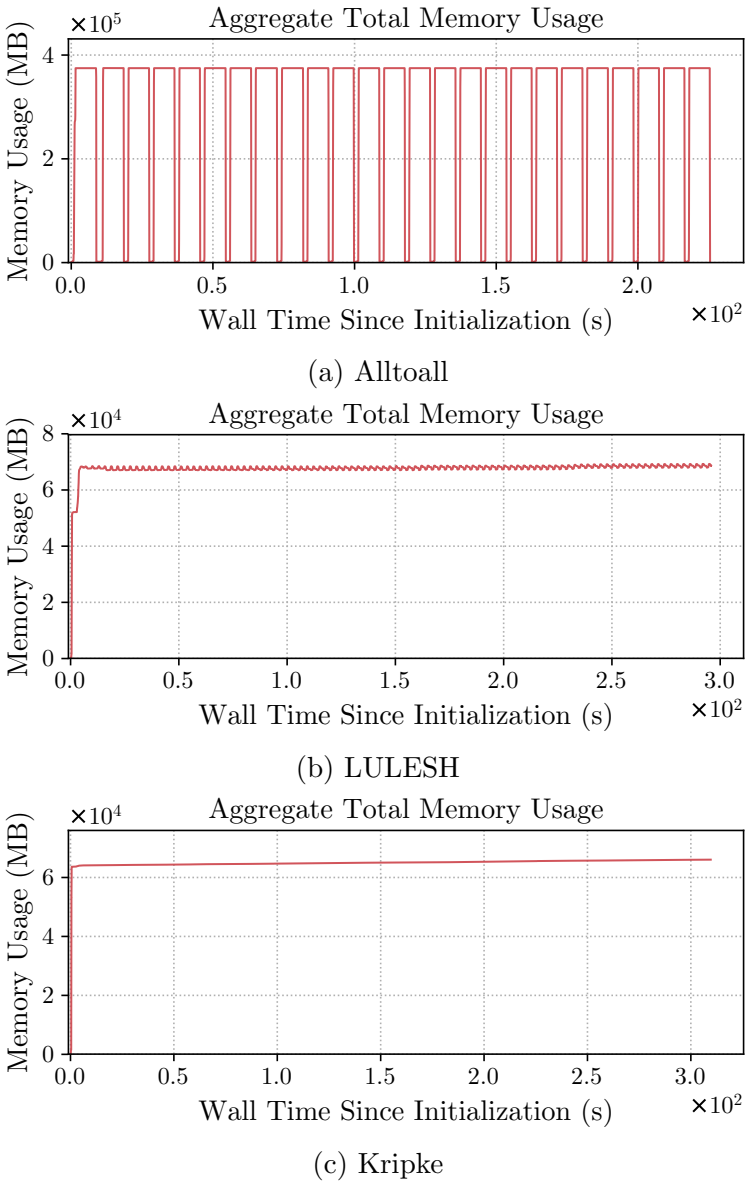


Figure 4.9: memnesia timelines showing aggregate total (i.e., MPI library and application) memory usage over time from 216-process (six-node) runs on Trinitite.

usage features. Similarly, the right column shows total memory usage (i.e., application and MPI library). Here we can see that both applications share a similar memory usage pattern: simulation state data dominate overall usage and remain relatively constant throughout their execution. Figure 4.9 shows aggregate (i.e.,

	Trinitite	Snow
Model	Cray XC40	Penguin Computing
# Nodes	100	368
OS	Cray Linux Environment	TOSS/CHAOS [24] (Linux)
CPU	2× 16-core Intel E5–2698 v3	2× 18-core Intel E5–2695 v4
RAM	128 GB	128 GB
Network	Aries NICs [10, 68]	Intel OmniPath
MPI	Cray MPICH 7.6.2	Open MPI 1.10.5, MVAPICH2 2.2
Compiler	Intel 17.0.4	GCC 5.3.0

Table 4.3: An overview of hardware and software used for this study.

summed) memory usage reports for three workloads run at 216 processes on Trinitite, where for LULESH and Kripke we use the same per-process configurations as before. Notice that our tool can capture application-specific data structure management features, for example, the regular oscillatory behavior exhibited in our Alltoall benchmark. We omit aggregate memory usage plots at other processor counts because the weak-scaled simulation state data dominates overall memory usage, so additional plots would look similar to the ones provided, only differing by some process scaling factor.

4.4.3 Peak Memory Usage

In this section, we study how job scale, workload, and runtime implementation influence per-process peak memory usage for MPI libraries and, whenever possible, compare results gathered from Cray MPICH's internal usage monitoring (craymem) to results reported by memnesia using two different *smaps* metrics: RSS and PSS. Unless otherwise noted, memnesia's reporting is based on proportional set size. We compare RSS and PSS metrics to highlight the differences between the two because

of memory page sharing. RSS tends to be a more pessimistic, and oftentimes inaccurate, metric because memory usage attributed to shared pages is counted multiple times; the multiplier in our case is the number of MPI processes sharing pages on a node. Table 4.4 shows peak memory usage averaged over MPI processes for four different workloads; the reported error for each entry represents the standard deviation across the peak memory usage reported for each process. Since craymem reports three memory usage components (`malloc()`, `mmap()`, and `shmget()`) without providing a corresponding time component (i.e., when they occurred relative to one another), we simply sum those values for reporting.

Our results show that job size, communication workload, and hardware/software architecture influence peak runtime memory usage, as indicated by craymem (cray) and memnesia (rss and pss) reporting (Table 4.4 and Table 4.5). Memory usage spikes observed at 64 processes are caused by crossing a compute node boundary—our experiments run a maximum of 32 processes per node. Of the workloads, Trivial achieves the lowest peak usage, while the proxy applications tend to yield the highest. The large standard deviations observed in the craymem data are due to large variations in per-node `shmget()` usage reporting: a single process reports non-zero usage, while the rest report zero. While valid, attributing shared-memory usage in this way is inaccurate for cooperative parallel workloads.

4.4.4 Tool-Induced Application Overhead

To quantify tool-induced overhead, we study how two performance metrics commonly used to assess message-passing systems, message rate and bandwidth, are affected while under memnesia supervision. This is accomplished by running the previously described multiple bandwidth/message rate micro-benchmark in both the presence and absence of memnesia instrumentation, where the latter serves as our performance baseline. Data were collected on Snow using Open MPI 1.10.5 over three different

	Number of MPI Processes					
	1	8	27	64	125	216
	Trivial					
cray	4.4 ± 0.0	4.9 ± 12.6	6.2 ± 29.4	19.1 ± 33.4	19.1 ± 32.7	23.4 ± 36.9
rss	2.2 ± 0.0	2.4 ± 0.0	2.6 ± 0.0	6.6 ± 0.2	6.7 ± 0.3	6.8 ± 0.4
pss	2.2 ± 0.0	1.0 ± 0.0	1.0 ± 0.0	4.6 ± 0.2	4.7 ± 0.3	4.8 ± 0.4
	Alltoall					
cray	4.4 ± 0.0	4.9 ± 12.6	6.4 ± 29.2	20.2 ± 33.3	20.2 ± 32.5	24.6 ± 36.7
rss	2.2 ± 0.0	3.0 ± 0.3	4.4 ± 0.0	12.7 ± 0.4	11.6 ± 0.5	11.2 ± 0.5
pss	2.2 ± 0.0	1.1 ± 0.1	2.7 ± 0.0	5.9 ± 0.2	6.0 ± 0.3	5.9 ± 0.4
	LULESH					
cray	4.4 ± 0.0	5.9 ± 12.6	7.3 ± 29.2	20.2 ± 33.3	20.2 ± 32.5	24.5 ± 36.7
rss	2.4 ± 0.0	5.0 ± 0.2	6.5 ± 0.8	10.4 ± 1.2	10.8 ± 0.8	11.1 ± 0.7
pss	2.4 ± 0.0	2.0 ± 0.1	2.3 ± 0.1	6.3 ± 0.7	6.9 ± 0.1	6.9 ± 0.1
	Kripke					
cray	4.4 ± 0.0	5.9 ± 12.6	7.2 ± 29.2	20.1 ± 33.3	20.1 ± 32.5	24.4 ± 36.7
rss	2.4 ± 0.0	3.7 ± 0.1	4.3 ± 0.2	8.0 ± 0.2	7.9 ± 0.3	8.0 ± 0.4
pss	2.3 ± 0.0	1.9 ± 0.0	2.1 ± 0.1	5.6 ± 0.2	5.7 ± 0.3	5.8 ± 0.4

Table 4.4: Average reported peak memory consumption (in MB) on Trinitite.

job sizes, plotted in Figure 4.10. We then conclude with an analysis of memnesia's memory overhead and application perturbation.

Effects on Message Rate and Bandwidth

Across the board, memnesia overheads are most apparent at small message sizes, where its effect on operational latencies dominates messaging rates in messages per second (MPS). Notice that our performance baselines have typical messaging rate curves where small message transfers yield the highest rates (3.6×10^6 , 6.0×10^7 , and 1.2×10^8 MPS for 1 B payloads at 1, 18, and 36 TX/RX pairs, respectively)—decreasing steadily from there as message size increases. In contrast, with memnesia

	Number of MPI Processes					
	1	8	27	64	125	216
	Trivial					
sn-ompi	7.6 ± 0.0	6.5 ± 0.0	6.4 ± 0.0	19.8 ± 0.0	19.9 ± 0.0	20.1 ± 0.0
sn-mv2	7.9 ± 0.0	10.9 ± 0.1	11.0 ± 0.1	24.5 ± 0.0	25.0 ± 0.2	25.7 ± 0.0
tt-mpich	2.2 ± 0.0	1.0 ± 0.0	1.0 ± 0.0	4.6 ± 0.2	4.7 ± 0.3	4.8 ± 0.4
	Alltoall					
sn-ompi	7.6 ± 0.0	6.6 ± 0.0	6.5 ± 0.4	19.8 ± 0.0	19.9 ± 0.0	20.2 ± 0.2
sn-mv2	8.0 ± 0.0	11.0 ± 0.1	11.0 ± 0.1	24.6 ± 0.3	25.0 ± 0.1	25.8 ± 0.0
tt-mpich	2.2 ± 0.0	1.1 ± 0.1	2.7 ± 0.0	5.9 ± 0.2	6.0 ± 0.3	5.9 ± 0.4
	LULESH					
sn-ompi	7.6 ± 0.0	7.1 ± 0.5	7.3 ± 0.7	20.6 ± 0.8	20.6 ± 0.6	20.8 ± 0.4
sn-mv2	8.0 ± 0.0	12.3 ± 0.4	12.4 ± 0.7	26.0 ± 0.7	26.3 ± 0.6	27.0 ± 0.4
tt-mpich	2.4 ± 0.0	2.0 ± 0.1	2.3 ± 0.1	6.3 ± 0.7	6.9 ± 0.1	6.9 ± 0.1
	Kripke					
sn-ompi	7.6 ± 0.0	6.6 ± 0.0	6.5 ± 0.0	19.8 ± 0.0	20.0 ± 0.3	20.3 ± 0.2
sn-mv2	8.0 ± 0.0	11.8 ± 0.1	11.8 ± 0.0	25.3 ± 0.0	25.8 ± 0.1	26.6 ± 0.1
tt-mpich	2.3 ± 0.0	1.9 ± 0.0	2.1 ± 0.1	5.6 ± 0.2	5.7 ± 0.3	5.8 ± 0.4

Table 4.5: Average peak memory consumption (in MB) on Trinitite (tt) and Snow (sn) as reported by memnesia.

supervision message rates appear to be capped and remain constant irrespective of message payload size, yielding message rates of approximately 1.0×10^2 , 1.4×10^3 , and 2.4×10^3 MPS across all payload sizes. This is caused by the collection of two *smaps* samples for each call into the MPI library, thereby increasing latency and therefore negatively affecting message rate.

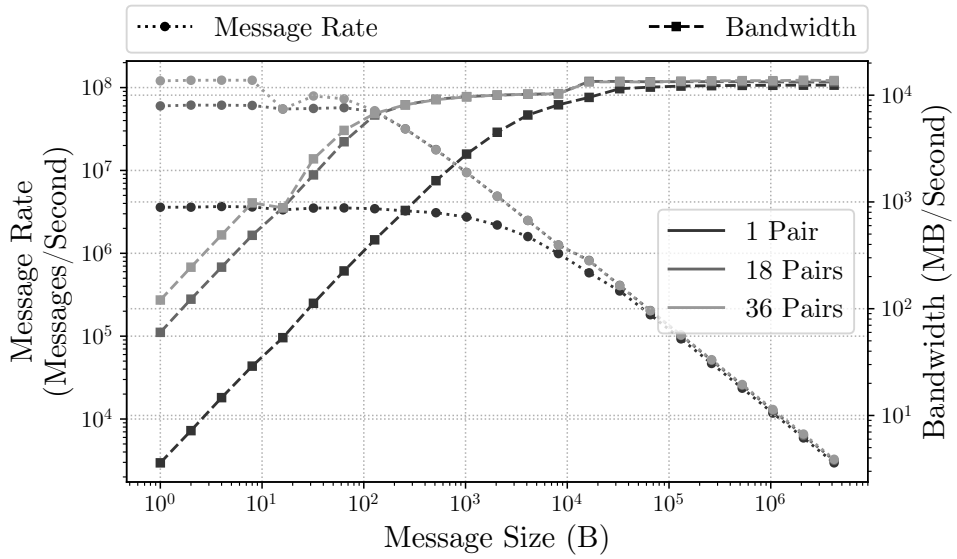
Large-message bandwidth is least affected by the presence of memnesia instrumentation because increased operational latencies are amortized over the transfer of larger payloads. That is, once a transfer is initiated, memnesia instrumentation has no appreciable effect on transfer rate. This micro-benchmark represents a worst-case scenario; still, memnesia can be useful in practice even though relative differences

shown here are large. Scientific applications tend not to be rate bound by small messages, which is the metric that is most severely degraded by the use of memnesia.

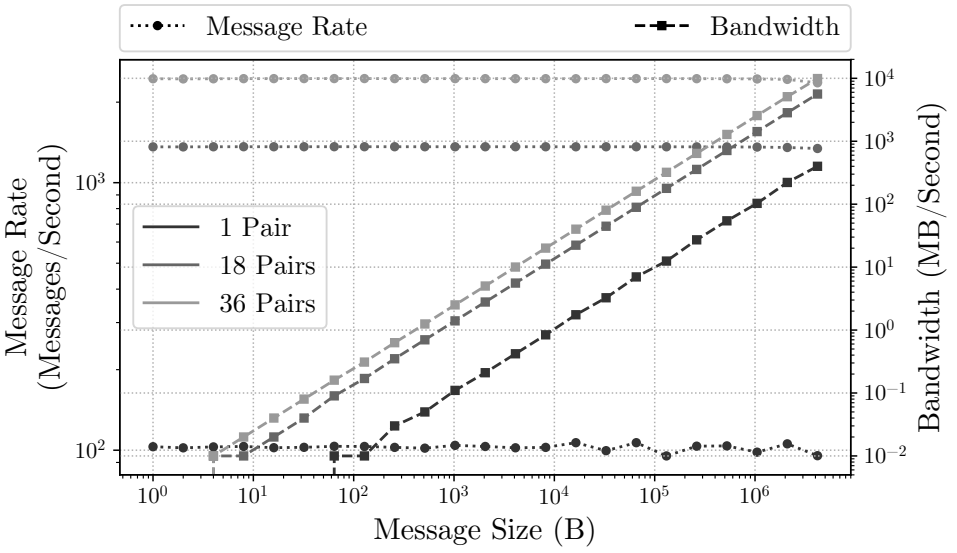
Memory Overhead and Perturbation

As argued in Section 4.2, gathering accurate component-level memory usage statistics is difficult. Because our memory profiler is loaded into the target application binary at startup, it becomes an additional application component and is therefore subject to the memory attribution problem, so we describe its overheads analytically. The aggregate memory overhead of our current reference implementation can be calculated as follows. Given m MPI processes under memnesia supervision, let n_p be the total number of trace events triggered by process $p, 0 \leq p < m$, where n_p equals the total number of times p called into the MPI library. For each trace event, two records are collected and subsequently stored, as detailed in Section 4.2.3. So total tool-induced memory overhead given m processes is proportional to $2s \sum_{i=0}^{m-1} n_i$, where s is a constant representing the size of a single trace record in bytes. Each trace record contains four entries: an 8-bit identifier naming the function that triggered data collection, two double-precision floating-point values storing timing information (start time and duration), and a 64-bit integer storing memory usage in kilobytes. Assuming 64-bit double-precision floating-point values, $s = 25\,\mathrm{B}$.

As previously described, our memory profiler is loaded into the target application binary at startup via the runtime loader. Consequently, memnesia's presence perturbs application heap behavior through its use of dynamic memory management (allocations and deallocations). For a single process, the primary unit of observation, the amount of tool-induced application perturbation is proportional to s times the number of trace records already collected by memnesia.



(a)



(b)

Figure 4.10: Results from the OSU multiple bandwidth/multiple message rate micro-benchmark, where the number of send/receive pairs vary. Figure a shows our performance baseline, while Figure b shows performance results with memnesia instrumentation enabled, both plotted using a log-log scale.

4.5 Discussion and Summary

Even though storage requirements for memnesia trace records are relatively small, structural improvements can be made to reduce their size, thereby decreasing overall tool-induced memory overhead and application perturbation. Because of the way our profiler is introduced into the application, tool-induced memory exhaustion manifests as an application runtime error that ultimately results in parallel job termination. The current implementation of memnesia requires calling `MPI_Finalize()` to flush memory usage statistics to disk for later analysis by other programs. This requirement is potentially problematic for long-running MPI applications because the amount of memory consumed by the tool grows without bound. A straightforward solution to limit memnesia's memory usage might include the use of `MPI_Pcontrol()`, which allows for a standard, user-accessible interface for controlling when tool data checkpoints are performed. This capability, in turn, could allow for user-defined *analysis extents*, trace data collected over a user-defined time span, that may then be used to attribute memory usage to specific application phases.

This work addresses the need for an easy to use, reasonably general, open source, and minimally intrusive tool for attributing dynamic memory usage to individual libraries comprising a distributed memory application. Our technique is able to capture features particular to a workload instance at a per-process and per-software-component level, a capability that is not readily available today through other means. The key techniques are function interposition and accurate memory map analysis. Our case study is MPI, addressing the growing need to understand and control the memory footprint of HPC applications on memory-constrained hardware. MPI already provides an interposition layer in the form of PMPI, obviating the need to create one for an arbitrary library via a mechanism such as `LD_PRELOAD`. Our results show that job size, communication workload, and hardware/software architecture influence peak runtime memory usage. As an example, our experimental

results show that different popular MPI implementations exhibit different memory usage behaviors, and such information could influence the choice of MPI implementation by application developers or users, and could also be of use to both MPI implementers and application/library developers to guide memory-use optimization of their implementations.

Chapter 5

Overhead of Adaptive Parallelism: A Case Study with Quo

In this chapter, we study the overhead of dynamically reconfiguring execution environments for coupled, thread-heterogeneous MT-MP programs. Focusing on runtime and memory costs brought on by the use of our dynamic approach, we examine Quo's overhead using proxy and full applications. We show that Quo's overheads are modest, imposing small runtime and memory usage penalties over the static baseline.

5.1 Runtime Overhead

We begin with an examination of Quo runtime overhead, quantifying three major sources: runtime operations, process quiescence, and data remapping. First, we measure the operational latencies of key Quo operations at different processor counts and discuss their performance and scaling characteristics up to 2,048 processes across 128 nodes. Next, using a scientific application as our case study, we contrast the performance of `MPI_Barrier()` and `QUO_barrier()` with that of the ideal case

requiring no process quiescence. Finally, we examine data remapping costs from the three applications used in our performance evaluation of Quo in Chapter 3.

5.1.1 Micro-Benchmark Results: Cost of Quo Operations

We quantify the individual overhead costs for a representative set of Quo operations using a micro-benchmark we developed named QuoBench. For each operation, we measure the time required to complete that operation 100 times in a tight loop—at each scale, processes co-located on the same compute node simultaneously execute this loop. Micro-benchmark results were collected on Cielo, detailed in Section 3.4.1. Figure 5.1 shows each operation's average execution time as a function of scale. All Quo operations, except `QUO_create()` and `QUO_free()`, are performed on a per-node basis, and their overheads are a function of the number of concurrent Quo processes within a single compute node. This phenomenon is observed in job sizes

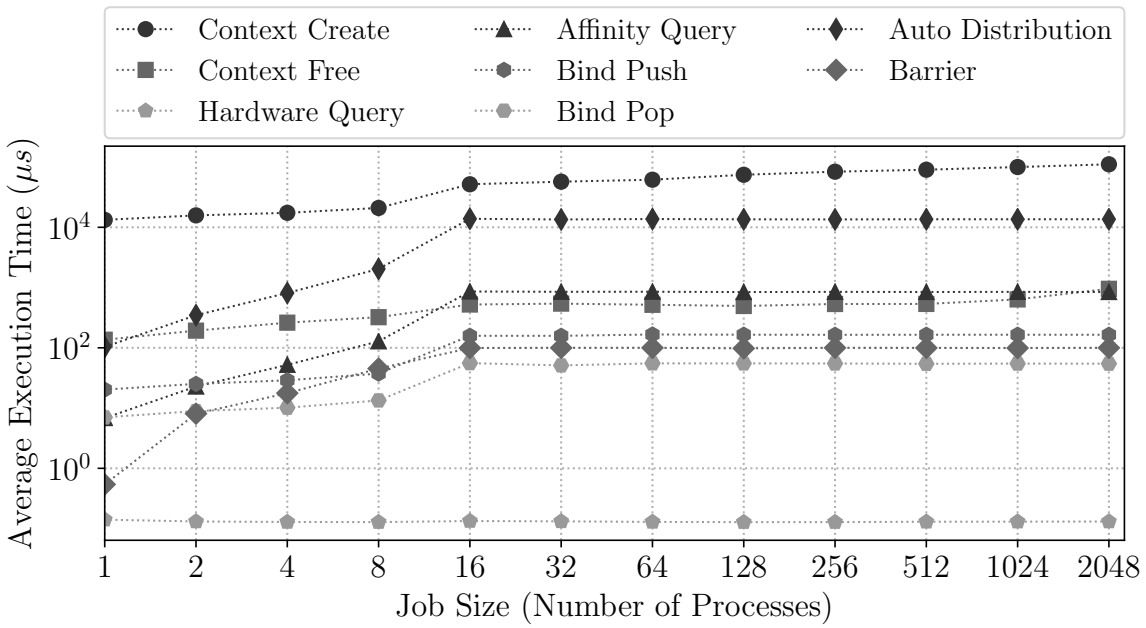


Figure 5.1: Log-log plot of average execution times of Quo operations on Cielo.

ranging from one to sixteen processes since our test platform contains sixteen-core compute nodes. `QUO_create()` and `QUO_free()` overheads depend on the total number of processes in the initializing communicator because they require inter-node process communication when processes are distributed. Figure 5.1 shows that even beyond 16 processes (i.e., the node width) the cost of these two operations continues to grow. Even so, their costs are modest at ∼100 ms at 2,048 processes across 128 nodes. Furthermore, these costs are amortized over the life of a Quo context: we expect most applications to use long-lived contexts that persist until library termination. Note that a long-lived context does not imply a single, static configuration; rather, it implies a single dynamic instance of Quo-maintained state.

5.1.2 Application Overhead from Process Quiescence

To evaluate the overhead of Quo process quiescence—a key Quo mechanism—we compare two approaches, namely `MPI_Barrier()` and `QUO_barrier()`. The benchmarking application is straightforward: an MPI-everywhere driver program that calls 2MESH's computationally intensive MPI+OpenMP library described in Table 3.3. Depending on the setup, before the multithreaded computation can be executed there is either no quiescence (ideally) or quiescence using one of the two approaches. We compare average wall-clock times reported by the application when using each mechanism. The single-node experiment is as follows.

1. 16 MPI processes are launched with a core binding policy, fully subscribing the cores in the compute node.

2. Four MPI processes are chosen using `QUO_auto_distrib()` such that each has an affinity to a different NUMA domain. The processes in this set P will enter the threaded compute phase.

3. Before executing the threaded 2MESH phase, processes in P push a NUMA

binding policy to accommodate the four OpenMP threads they will spawn in each NUMA domain, while the remaining processes are quiesced using either `MPI_Barrier()` or `QUO_barrier()`.

Table 5.1 contrasts the performance of `MPI_Barrier()` and `QUO_barrier()` with that of the ideal case in which four MPI processes (each with a NUMA binding policy) are launched across all four NUMA domains on the target architecture, thereby avoiding the need for quiescing any processes, thus mimicking what today's MPI+X codes do in practice. The results show that our `QUO_barrier()` implementation significantly outperforms `MPI_Barrier()` and is close to the ideal case where quiescence is not necessary. In particular, our approach introduces approximately an 8 % overhead, while the naive approach using `MPI_Barrier()` over `MPI_COMM_WORLD` introduces approximately 116 % overhead. Please note that because this experiment runs on a single compute node this setup mimics the shared-memory sub-communicator approach outlined in Section 3.3.7.

Quiescence Mechanism	Average Execution Time	Mechanism Overhead
Ideal	16.46 ± 0.05 s	—
QUO Barrier	17.82 ± 0.32 s	8.24 %
MPI Barrier	35.49 ± 0.17 s	115.63 %

Table 5.1: Average quiescence-induced overhead by mechanism.

5.1.3 Application Overhead from Data Remapping

Next, we quantify data remapping overhead at different scales and input configurations using the three applications introduced in Section 3.4, detailed in Table 3.3. We study data remapping cost because it is additional runtime overhead often brought

on by the use of our approach. As previously described in Section 3.3.5, before the number of active MPI processes can be safely increased or decreased, data must be exchanged among node-local processes to satisfy all inter-process data dependencies. Typically, this data remapping occurs via node-local gathers and scatters before and after Quo-enabled regions, respectively. Here we study two classes of applications: 1. those requiring data remapping irrespective of whether or not Quo is in use and 2. those only requiring data remapping when Quo is enabled. Examples of the former are 2MESH and ESMD, as data continuously are remapped from one library domain to another. That said, differences in data movement patterns still exist between the baseline and Quo-enabled versions, which we will discuss later. An example of the latter is RAGE, as data remapping occurs as a result of using Quo and would otherwise be unnecessary.

Table 5.2 details data remapping costs without and with Quo, where the former serves as our performance baseline. The first column names each experiment, organized as a hyphen-delimited string comprising an application identifier, input name, and scale (in the number of available PEs). The two remaining columns contain three entries each: average time spent on remapping data (Remap), average total execution time (Total), and percentage of total time spent remapping (R % of Total).

Across the board, our performance baselines spend little to no time remapping data between library domains, averaging across the 30 experiments approximately 2 % of total runtime—the worst case spending in total ∼6 % (2MESH-W1-512). For their baseline remapping phases, 2MESH and ESMD use all available MPI processes in `MPI_COMM_WORLD` to parallelize data transfers between library domains. Since N processes (i.e., the size of `MPI_COMM_WORLD`) participate in mapping data to and from each domain, we will call this exchange N-N. For Quo-enabled runs, 2MESH and ESMD use an N-M (or M-N) data exchange pattern wherein N processes exchange data with M processes before and after Quo-enabled regions, where $M < N$.

	Without Quo		With Quo	
Experiment	Remap, Total (R% of T)		Remap, Total (R% of T)	
2MESH-W1-128	29.14 s,	522.73 s (5.58 %)	33.55 s,	229.62 s (14.61 %)
2MESH-W1-512	31.83 s,	518.99 s (6.13 %)	32.37 s,	266.40 s (12.15 %)
2MESH-W1-2048	33.67 s,	710.35 s (4.74 %)	36.93 s,	428.28 s (8.62 %)
2MESH-W2-128	18.18 s,	467.22 s (3.89 %)	10.09 s,	196.38 s (5.14 %)
2MESH-W2-512	18.81 s,	402.55 s (4.67 %)	11.86 s,	202.25 s (5.86 %)
2MESH-W2-2048	21.20 s,	461.08 s (4.60 %)	17.40 s,	304.17 s (5.72 %)
2MESH-W3-128	6.55 s,	106.59 s (6.15 %)	4.74 s,	88.89 s (5.33 %)
2MESH-W3-512	6.69 s,	123.24 s (5.43 %)	5.10 s,	117.83 s (4.33 %)
2MESH-W3-2048	8.03 s,	192.51 s (4.17 %)	5.85 s,	189.68 s (3.09 %)
2MESH-W4-128	1.13 s,	55.98 s (2.02 %)	3.21 s,	69.02 s (4.66 %)
2MESH-W4-512	1.51 s,	82.51 s (1.83 %)	3.35 s,	98.97 s (3.38 %)
2MESH-W4-2048	1.38 s,	133.59 s (1.03 %)	3.89 s,	184.23 s (2.11 %)
2MESH-S5-128	198.08 s,	6556.40 s (3.02 %)	98.04 s,	2764.60 s (3.55 %)
2MESH-S5-512	55.55 s,	1599.57 s (3.47 %)	34.35 s,	679.92 s (5.05 %)
2MESH-S5-2048	20.82 s,	461.07 s (4.52 %)	15.99 s,	305.65 s (5.23 %)
RAGE-W1-64	0.00 s,	1884.65 s (0.00 %)	5.00 s,	908.31 s (0.55 %)
RAGE-W1-256	0.00 s,	1854.61 s (0.00 %)	3.00 s,	1171.06 s (0.26 %)
RAGE-W1-512	0.00 s,	1924.41 s (0.00 %)	3.80 s,	1405.99 s (0.27 %)
RAGE-W2-64	0.00 s,	1581.83 s (0.00 %)	1.90 s,	558.63 s (0.34 %)
RAGE-W2-256	0.00 s,	1549.32 s (0.00 %)	2.00 s,	831.73 s (0.24 %)
RAGE-W2-512	0.00 s,	1629.10 s (0.00 %)	1.85 s,	1067.85 s (0.17 %)
RAGE-S3-64	0.00 s,	10782.28 s (0.00 %)	13.50 s,	1872.14 s (0.72 %)
RAGE-S3-256	0.00 s,	3095.27 s (0.00 %)	4.40 s,	1442.50 s (0.31 %)
RAGE-S3-512	0.00 s,	1929.49 s (0.00 %)	3.40 s,	1406.68 s (0.24 %)
ESMD-W1-80	2.80 s,	291.94 s (0.96 %)	17.23 s,	281.94 s (6.11 %)
ESMD-W1-320	3.36 s,	464.30 s (0.72 %)	17.64 s,	432.42 s (4.08 %)
ESMD-W1-640	17.81 s,	543.06 s (3.28 %)	37.84 s,	486.91 s (7.77 %)
ESMD-S2-80	2.69 s,	768.93 s (0.35 %)	24.56 s,	561.71 s (4.37 %)
ESMD-S2-320	1.74 s,	330.37 s (0.53 %)	14.38 s,	279.36 s (5.15 %)
ESMD-S2-640	1.89 s,	260.95 s (0.72 %)	12.06 s,	273.92 s (4.40 %)

Table 5.2: Application results: average data remapping costs without and with Quo.

Focusing now on results from Quo-enabled experiments, we notice that on average the percentage of time spent on remapping data has increased from approximately 2 % to about 4 %. This increase is not as pronounced as one would expect; the reasons are twofold. First, in the cases where the use of Quo necessitates data remapping, only a small fraction of total execution time (less than 1 %) is required. This phenomenon is observed in the nine RAGE experiments. Second, for the remaining cases requiring data remapping irrespective of whether or not Quo is in use, we see two distinct behaviors. The first, exhibited predominantly by ESMD, shows N-N outperforming N-M for all six experiments, though not substantially. The second shows that for the majority of the 2MESH experiments (9/15), N-M data exchange patterns improve remapping times. Note that since we use optimized execution times in our remapping overhead calculation, Quo-enabled overheads appear as a larger percentage of overall execution time.

In general, data remapping overhead is dependent on application-specific features. The most influential factors determining its impact include:

- *Data volume*: the amount of data exchanged to meet the remap requirements.
- *Frequency*: how often data must be exchanged to meet the remap requirements.
- *Data layout*: how key data structures are organized and accessed in memory.
- *Remap methodology*: how the remap operation is implemented and parallelized.

5.2 Memory Overhead

In this section, we examine the memory overhead of our reference implementation and approach. We begin by quantifying the memory requirements of Quo runtime state, i.e., the amount of memory overhead incurred by applications when using Quo. Because Quo's overall runtime memory footprint is influenced primarily by scale,

MPI implementation, and hardware architecture, we present data from experiments varying all three of these factors. We then study the impact of maintaining quiesced processes, as our approach for *task elasticity*, i.e., changing the number of active MPI processes at runtime via Quo-provided facilities to match a given parallelization strategy, often requires maintaining more MPI processes than would otherwise be necessary when using a static, under-subscribed approach. So, we compare MPI library memory usage of fully subscribed configurations to five under-subscribed configurations. The fully subscribed configurations serve as a proxy for Quo-enabled applications requiring full resource subscription, while the under-subscribed configurations replicate today's static approach for accommodating thread-level heterogeneity in coupled MT-MP programs at different threading levels. For example, the 4 PPN configuration represents a static, under-subscribed configuration with process threading factors of either 8 (for Trinitite) or 9 (for Snow).

5.2.1 Cost of Quo Runtime State

To quantify the cost of Quo runtime state, we compare the memory usage of Trivial, described in Section 4.3.1, with that of QuoBench, introduced in Section 5.1.1, where Trivial acts as our memory usage baseline. Using memnesia, usage statistics were collected from three MPI implementations running on the Trinitite and Snow systems (Table 4.3). Average memory overhead is broken into three categories as shown in Table 5.3. The first category, labeled as MPI, is Quo's impact on MPI library memory usage. Recall Quo uses MPI services for some of its operations, so we must therefore consider Quo's effect on the message passing system's memory footprint. The second category, labeled as App., measures the amount of memory overhead each Quo-enabled process will incur because of Quo-maintained state. Finally, aggregate per-process memory overhead (MPI and App.) is presented as Total. Our results show that for all 18 configurations Quo's impact on MPI and application memory

	Number of MPI Processes					
	1	8	27	64	125	216
	Open MPI (Snow)					
MPI	0.00 MB	0.04 MB	0.06 MB	0.57 MB	0.59 MB	0.57 MB
App.	0.98 MB	0.77 MB	0.73 MB	0.74 MB	0.74 MB	0.75 MB
Total	0.98 MB	0.81 MB	0.79 MB	1.31 MB	1.33 MB	1.32 MB
	MVAPICH2 (Snow)					
MPI	0.02 MB	0.60 MB	0.60 MB	1.11 MB	1.08 MB	1.11 MB
App.	0.57 MB	0.44 MB	0.47 MB	0.47 MB	0.47 MB	0.48 MB
Total	0.59 MB	1.04 MB	1.07 MB	1.58 MB	1.55 MB	1.59 MB
	Cray MPICH (Trinitite)					
MPI	0.11 MB	0.52 MB	0.47 MB	1.29 MB	0.98 MB	1.28 MB
App.	1.63 MB	0.97 MB	0.93 MB	0.91 MB	0.92 MB	0.92 MB
Total	1.74 MB	1.49 MB	1.40 MB	2.20 MB	1.90 MB	2.20 MB

Table 5.3: Average per-process cost of Quo runtime state.

usage is small, imposing in total an average of ∼1.4 MB per process—in the worst case requiring only 2.2 MB.

5.2.2 Cost of Quiesced Processes

Because our approach often requires that more MPI processes be maintained (i.e., started and kept for the entirety of the parallel application's execution) than would otherwise be necessary when using other approaches, we must examine the amount of memory required to keep additional MPI process state. For this study, we compare the memory usage of five hardware resource subscription levels, where the number of single-threaded MPI processes placed on each compute node varies, to fully subscribed configurations (36 PPN for Snow and 32 PPN for Trinitite). Our experiment is designed to measure average process- and node-level memory overhead brought on

	Average Process Memory Footprint	Average Node Memory Footprint
Open MPI (Snow)	20.69 ± 0.02 MB	744.97 MB (36 PPN)
MVAPICH2 (Snow)	26.32 ± 0.02 MB	947.43 MB (36 PPN)
Cray MPICH (Trinitite)	5.46 ± 0.25 MB	174.83 MB (32 PPN)

Table 5.4: MPI library memory footprint of fully subscribed node configurations.

by fully subscribing a compute node—the highest level of resource subscription that would be considered in practice—to understand worst-case memory usage overheads, as the use of Quo does not necessitate full resource subscription. For each subscription level, we present MPI library memory usage and resource-subscription-induced overhead data collected from runs of our Trivial micro-benchmark under memnesia supervision. The experimental setup is as follows. For each system/MPI pair, we collect per-process MPI library memory usage statistics from six fully subscribed compute nodes. Similarly, we also measure MPI library memory usage from five under-subscribed configurations (e.g., 1 PPN, 2 PPN, 4 PPN), again running across six compute nodes for each subscription level and system/MPI pair.

Table 5.4 details results from three fully subscribed experiments. From those data, we see that the amount of memory consumed by MPI libraries again depends on both the underlying system architecture and MPI implementation. On average each process requires ∼17 MB of memory to store MPI library state—in the worst case requiring approximately 26 MB (MVAPICH2 at 36 PPN on Snow) and in the best case requiring about 5 MB (Cray MPICH at 32 PPN on Trinitite). Table 5.5 details results from 15 under-subscribed experiments. At each scale (PPN), we present average per-process MPI library storage requirements (Proc. Footprint), their respective translation to per-node memory requirements (Node Footprint), and finally node-level resource-subscription-induced overhead (Node Overhead)—

	Number of MPI Processes Per Node				
	1	2	4	8	16
	Open MPI (Snow)				
Proc. Footprint	21.51 MB	20.93 MB	20.61 MB	20.46 MB	20.46 MB
Node Footprint	21.51 MB	41.85 MB	82.42 MB	163.71 MB	327.43 MB
Node Overhead	723.46 MB	703.12 MB	662.55 MB	581.26 MB	417.54 MB
	MVAPICH2 (Snow)				
Proc. Footprint	21.87 MB	25.36 MB	25.10 MB	25.12 MB	25.41 MB
Node Footprint	21.87 MB	50.72 MB	100.40 MB	200.96 MB	406.64 MB
Node Overhead	925.55 MB	896.71 MB	847.03 MB	746.46 MB	540.79 MB
	Cray MPICH (Trinitite)				
Proc. Footprint	8.02 MB	6.77 MB	6.16 MB	4.91 MB	4.97 MB
Node Footprint	8.02 MB	13.54 MB	24.62 MB	39.25 MB	79.50 MB
Node Overhead	166.81 MB	161.30 MB	150.21 MB	135.58 MB	95.34 MB

Table 5.5: Resource-subscription-induced memory overhead.

calculated as the difference of fully subscribed and under-subscribed compute node memory footprints. For example, node overhead for Cray MPICH at 4 PPN is $150.21\,\text{MB} = 174.83\,\text{MB} - 24.62\,\text{MB}$.

Our results show that the resource subscription differential most impacts the amount of memory overhead brought on by maintaining additional MPI processes to achieve dynamic task elasticity between corresponding dynamic (i.e., with Quo) and static (i.e., under-subscribed) configurations, as the difference in PPN acts as a node memory usage multiplier. For example, the three 1 PPN cases yield the highest memory overheads, averaging across the experiments 605.27 MB per node, because when considered against their fully subscribed counterparts they exhibit the highest resource subscription differentials in our study.

Given today's MPI library memory requirements, the most evident source of Quo

memory overhead comes from maintaining additional MPI process state. Even so, the trade-off between space and speedup is not prohibitive in practice. For example, in the most severe case (MVAPICH2 at 1 PPN) our approach consumes less than 1 % of total available node memory. As message-passing middleware becomes more memory efficient, a topic of active research [17, 56, 98, 104], resource-subscription-induced overheads will follow suit; in turn, making our approach viable in severely memory-constrained environments.

5.3 Summary

We have presented a study concerning our approach's runtime and memory overheads to dynamically reconfigure phased MT-MP programs. Using proxy and full applications as our case studies, we examined five sources of Quo-induced overhead. Our runtime overhead results show for the three sources considered that 1. Quo operations exhibit low operational latencies, averaging ∼2 ms at 16 PPN across 128 nodes, 2. `QUO_barrier()` significantly outperforms `MPI_Barrier()`, imposing an 8.24 % runtime penalty for MPI process quiescence versus 115.63 %, and 3. Quo-induced data remapping overhead averages approximately 2 % over the time required by the 30 static baseline configurations tested. Our memory overhead results show that Quo runtime state is small, having a memory footprint of about 1.4 MB per process. The most apparent source of memory overhead stems from our approach for task elasticity. Our results show that resource-subscription-induced overheads average 516.91 MB per node—less than 1 % available in our case.

Chapter 6

Conclusion

A tremendous amount of software has and continues to be designed and built around message passing, as this model has shown to be incredibly robust for expressing parallelism in scientific applications. Even so, as computer architectures have evolved, so too have the techniques used to program them. Our research aims to improve upon long-standing approaches concerning how coupled, parallel, and distributed message passing software is structured, executed, and analyzed in the HPC domain.

6.1 Summary of Contributions

We have presented new ways to structure and execute coupled MT-MP programs with dynamic, phased configuration conflicts. Our design and implementation are influenced by requirements for generality, composability, efficiency, and pragmatism in the face of production HPC software realities. The key techniques used are low-latency runtime hardware/software environment queries, programmable dynamic task affinities, and efficient node-level process quiescence. Our case study is MPI, addressing the growing need for adaptive parallelism in massively parallel MT-MP

programs. Our performance results show that for a majority of the tested scientific workloads our approach and corresponding open-source reference implementation render speedups greater than 50 % over the static under-subscribed baseline—the previous, long-standing mode for production runs of these applications.

We have also presented a new memory usage analysis technique for parallel and distributed message passing applications. The results show that our approach and corresponding implementation accurately quantify memory resource usage as a function of time, scale, communication workload, and software or hardware system architecture, clearly distinguishing between application and communication library memory usage at a per-process level—a capability that is not readily available today through other means. With this new capability, we show that job size, communication workload, and hardware/software architecture influence peak runtime memory usage. The key techniques used are function interposition and accurate memory map analysis. Our case study is MPI, addressing the growing need to understand and control the memory footprint of HPC applications on memory-constrained hardware.

6.2 Open Related Studies

New questions and potential opportunities for follow-on research have arisen from our work. Here we discuss open topics that remain work for the immediate future.

6.2.1 Transparent Data Dependency Satisfaction

Recall that only after outstanding inter-process data dependencies are satisfied can Quo safely enact new task configurations. Currently, the integrator of Quo assumes this burden. Meaning in practical terms that inter-process data dependencies are managed explicitly and programmatically through message passing. Often, ad-

ditional source code specific to the use of Quo implements this logic. While in practice this requirement is not prohibitive, it nevertheless adds complexity and runtime overhead. For these reasons, we are investigating ways to transparently handle data dependencies existing between processes that share RAM on a compute node. Specifically, our prototype implements a cooperative memory allocator that allows for inter-process data structure access via shared memory. Through the use of a custom Kokkos memory space [37], we are investigating the efficacy of this approach.

6.2.2 Examination of Other Dynamic Configurations

In this dissertation, we have thoroughly examined the costs and benefits of dynamically (at runtime) resolving configuration conflicts in a way that favors full utilization of available compute resources throughout all phases (both MP and MT-MP) of a coupled parallel application. The following dynamic configurations, which in contrast run under-subscribed during some computational phases, have not yet been subject to the same level inquiry as to the ones already considered in this work.

- **Under-Subscribed, Fully Subscribed**: A variation of the standard static approach used to accommodate thread-level heterogeneity in coupled MT-MP applications today. Recall that the conventional method statically (at application launch time) under-subscribes compute resources such that the computational phase with the highest degree of threading per process has one PE per software thread. Because of this, MP phases with fewer threads per process run with a potentially suboptimal scheduling policy. In contrast, our approach can enact affinity schedules specifically tailored for each phase. MP phases under-subscribe hardware resources such that each single-threaded MP process is given affinity to a dedicated PE. MT-MP phases fully subscribe hardware resources such that each MT-MP process is given affinity to an appropriate set

of PEs determined by the threading degree desired, hardware availability, and hardware locality. For the reasons described in Chapters 3 and 5, when in this regime two sources of runtime overhead are avoided, namely process quiescence and data remapping.

- **Fully/Under-Subscribed, Under-Subscribed**: A configuration used to accommodate coupled MP phases with different strong scaling characteristics. Consider a situation where some MP phases demonstrate strong scaling up to a given processor count, while the others do not. In this case, our efficient process quiescence mechanism could enable such a configuration.

6.3 Concluding Remarks

At LANL the practicality and utility of our pragmatic approach for adaptive parallelism in coupled MT-MP programs have been demonstrated by continuous production use for over two years—running at scales well in excess of 200k PEs and servicing demanding scientific workloads running on a variety of HPC platforms.

Appendices

Appendix A

Example Quo Policies

A.1 Pseudocode for a Caller-Driven Quo Policy

```
QUO_create(&ctx, MPI_COMM_WORLD); // Create a context.
// Query runtime software/hardware environment and gather
// information to influence configuration selection algorithm.
tres = QUO_OBJ_NUMANODE; // Set target resource to NUMA.
// Let QUO find a set of node-local MPI processes (optimized
// for maintaining data locality) that satisfy the
// distribution criterion that no more than max_pe processes
// be assigned to each NUMA domain on the host.
QUO_auto_distrib(ctx, tres, max_pe, &in_dset);
// All MPI processes participate in data dissemination (via
// message passing) to satisfy outstanding data dependencies.
// If in_dset is true, then the calling process is a member of
// the distribution set and will perform the calculation.
if (in_dset) {
    // Change binding policy to cover resources with
    // affinity to the caller's closest NUMA domain.
    QUO_bind_push(ctx, tres);
    // Perform threaded computation with newly
    // enacted process hardware affinity policy.
    result = A_threaded_library_call(in_args);
    // Revert to prior process binding policy
    // before entering node-local QUO barrier.
    QUO_bind_pop(ctx);
}
// Quiesce set of active MPI processes not in distribution set
// by yielding their use of compute resources, while those who
// are spawn threads onto those resources.
QUO_barrier(ctx);
// Barrier complete, all MPI processes participate in result
// dissemination (via message passing) to relay result to all
// cooperating processes in calculation.
...
QUO_free(ctx); // Relinquish context resources.
```

Listing A.1: Pseudocode showing a *caller-driven* policy using hardware queries and application characteristics to guide a dynamic affinity schedule with Quo.

A.2 Pseudocode for a Callee-Driven Quo Policy

```
result_t A_threaded_library_call(args_t in_args) {
    if (!ctx) { // Initialization of encapsulated QC required?
        QUO_create(&ctx, in_args->mpi_comm);
        // Query and cache hardware configuration.
    }
    if (need_new_target_resource) {
        // Determine appropriate runtime configuration
        // using library-specific heuristics.
        tres = QUO_OBJ_SOCKET; // Target resource.
        // Let QUO find a set of node-local MPI processes
        // that satisfy the given distribution criteria.
        QUO_auto_distrib(ctx, tres, max_pe, &in_dset);
    }
    // MPI processes participate in data dissemination (via
    // message passing) to satisfy data dependencies.
    if (in_dset) { // Selected to perform the calculation?
        // Change binding policy to cover resources with
        // affinity to the caller's target resource.
        QUO_bind_push(ctx, tres);
        // Perform threaded computation with newly
        // enacted process hardware affinity policy.
        result = A_threaded_kernel(in_args);
        // Revert to prior process binding policy
        // before entering node-local QUO barrier.
        QUO_bind_pop(ctx);
    }
    // Quiesce set of active MPI processes not in distribution
    // set by yielding their use of compute resources, while
    // those who are spawn threads onto those resources.
    QUO_barrier(ctx);
    // Barrier complete, MPI processes in in_args->mpi_comm
    // participate in result dissemination (via message
    // passing) to relay result to all cooperating processes.
    return result;
}
```

Listing A.2: Pseudocode showing a *callee-driven* policy using hardware queries and application characteristics to guide a dynamic affinity schedule with Quo.

References

[1] libhugetlbfs (7) Linux User's Manual, September 2008.

[2] ld.hugetlbfs (1) Linux User's Manual, March 2012.

[3] proc (5) Linux User's Manual, December 2015.

[4] ptrace (2) Linux User's Manual, April 2018.

[5] Ishfaq Ahmad and Muhammad Kafil. A Parallel Algorithm for Optimal Task Assignment in Distributed Systems. In *Advances in Parallel and Distributed Computing*, pages 284–290. IEEE, 1997.

[6] Yuichiro Ajima, Yuzo Takagi, Tomohiro Inoue, Shinya Hiramoto, and Toshiyuki Shimizu. The Tofu Interconnect. In *19th Annual Symposium on High Performance Interconnects (HOTI)*, pages 87–94. IEEE, 2011.

[7] B.J. Alder and T.E. Wainwright. Phase Transition for a Hard Sphere System. *The Journal of Chemical Physics*, 27(5):1208–1209, 1957.

[8] Bob Alverson, Edwin Froese, Larry Kaplan, and Duncan Roweth. Cray XC Series Network. *Cray Inc., White Paper WP-Aries01–1112*, 2012.

[9] Robert Alverson, Duncan Roweth, and Larry Kaplan. The Gemini System Interconnect. In *18th Annual Symposium on High Performance Interconnects (HOTI)*, pages 83–87. IEEE, 2010.

[10] Alverson, Bob and Froese, Edwin and Kaplan, Larry and Roweth, Duncan. Cray XC Series Network. *Cray Inc., White Paper*, WP-Aries01–1112, 2012.

[11] Gene M Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pages 483–485. ACM, 1967.

[12] Björn Andersson and Eduardo Tovar. Multiprocessor Scheduling with Few Preemptions. In *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 322–334. IEEE, 2006.

[13] A. Arnold, O. Lenz, S. Kesselheim, R. Weeber, F. Fahrenberger, D. Roehm, P. Košovan, and C. Holm. ESPResSo 3.1 — Molecular Dynamics Software for Coarse-Grained Models. In M. Griebel and M. A. Schweitzer, editors, *Meshfree Methods for Partial Differential Equations VI*, volume 89 of *Lecture Notes in Computational Science and Engineering*, pages 1–23. Springer, 2013.

[14] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[15] Allison H Baker, Robert D Falgout, Tzanio V Kolev, and Ulrike Meier Yang. Scaling Hypre's Multigrid Solvers to 100,000 Cores. In *High-Performance Scientific Computing*, pages 261–279. Springer, 2012.

[16] Henry Baker and Carl Hewitt. Laws for Communicating Parallel Processes. 1977. MIT Artificial Intelligence Laboratory.

[17] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing Lusk, Rajeev Thakur, and Jesper Larsson Träff. MPI on a Million Processors. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 20–30. Springer, 2009.

[18] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 66. IEEE Computer Society Press, 2012.

[19] Heiko Bauke, Stephan Mertens, and Andreas Engel. Phase Transition in Multiprocessor Scheduling. *Physical Review Letters*, 90(15):158701, 2003.

[20] Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, and Charles V. Packer. BEOWULF: A Parallel Workstation for Scientific Computation. In *International Conference on Parallel Processing*, volume 95, pages 11–14, 1995.

[21] Abhinav Bhatelé, Gagan Raj Gupta, Laxmikant V Kalé, and I-Hsin Chung. Automated Mapping of Regular Communication Graphs on Mesh Interconnects. In *International Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2010.

[22] Abhinav Bhatele and Laxmikant V Kale. Heuristic-Based Techniques for Mapping Irregular Communication Graphs to Mesh Topologies. In *13th International Conference on High Performance Computing and Communications (HPCC)*, pages 765–771. IEEE, 2011.

[23] Shahid H. Bokhari. On the Mapping Problem. *IEEE Transactions on Computers*, C-30(3):207–214, 1981.

[24] Ryan L Braby, Jim E Garlick, and Robin J Goldstone. Achieving Order Through CHAOS: the LLNL HPC Linux Cluster Experience. In *The 4th International Conference on Linux Clusters: The HPC Revolution*, 2003.

[25] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 180–186, 2010.

[26] A Canning, J Shalf, N Wright, S Anderson, and M Gajbe. A Hybrid MPI/OpenMP 3D FFT for Plane Wave First-principles Materials Science Codes. In *Proceedings of CSC12 Conference*, 2012.

[27] R. H. Castain, T. S. Woodall, D. J. Daniel, J. M. Squyres, B. Barrett, and G .E. Fagg. The Open Run-Time Environment (OpenRTE): A Transparent Multi-Cluster Environment for High-Performance Computing. In *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.

[28] Chris Chambreau. memP: Parallel Heap Profiling. `http://memp.sourceforge.net`, 2010.

[29] Dong Chen, Noel Eisley, Philip Heidelberger, Sameer Kumar, Amith Mamidala, Fabrizio Petrini, Robert Senger, Yutaka Sugawara, Robert Walkup, Burkhard Steinmacher-Burow, et al. Looking Under the Hood of the IBM Blue Gene/Q Network. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 69. IEEE Computer Society Press, 2012.

[30] Edmond Chow and David Hysom. Assessing Performance of Hybrid MPI/OpenMP Programs on SMP Clusters. *Lawrence Livermore National Laboratory Technical Report UCRL-JC-143957*, 2001.

[31] Edward G. Coffman, Jr, Michael R. Garey, and David S. Johnson. An Application of Bin-Packing to Multiprocessor Scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.

[32] Phil Colella. Defining Software Requirements for Scientific Computing. January 2004.

[33] Timothy W. Curry and et al. Profiling and Tracing Dynamic Library Usage Via Interposition. pages 267–278 . USENIX Summer, 1994.

[34] Frederica Darema, David A George, V Alan Norton, and Gregory F Pfister. A Single-Program-Multiple-Data Computational Model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11–24, 1988.

[35] Jack J. Dongarra, James R. Bunch, Cleve B. Moler, and Gilbert W. Stewart. *LINPACK Users' Guide*. Siam, 1979.

[36] N. Drosinos and N. Koziris. Performance Comparison of Pure MPI vs Hybrid MPI-OpenMP Parallelization Models on SMP Clusters. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004.

[37] H Carter Edwards and Daniel A Ibanez. Kokkos' Task DAG Capabilities. *SAND2017-10464, Technical Report*, 2017.

[38] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling Manycore Performance Portability Through Polymorphic Memory Access Patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[39] Michael J Flynn. Very High-Speed Computing Systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[40] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. Budapest, Hungary, September 2004. 11th European PVM/MPI Users' Group Meeting.

[41] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[42] Richard Gerber, James Hack, Katherine Riley, Katie Antypas, Richard Coffey, Eli Dart, Tjerk Staatsma, Jack Wells, Deborah Bard, Sudip Dosanjh, et al. Crosscut Report: Exascale Requirements Reviews. Technical report, Oak Ridge National Laboratory; Argonne National Laboratory; Lawrence Berkeley National Laboratory, Tysons Corner, Virginia, March 2017.

[43] Brice Goglin. Managing the Topology of Heterogeneous Cluster Nodes with Hardware Locality (hwloc). In *High Performance Computing & Simulation (HPCS), 2014 International Conference on*, pages 74–81. IEEE, 2014.

[44] Brice Goglin. Towards the Structural Modeling of the Topology of Next-Generation Heterogeneous Cluster Nodes with hwloc. Research report, Inria, November 2016.

[45] Google. Google Performance Tools—gperftools. `https://github.com/gperftools/gperftools`, 2017.

[46] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[47] Samuel K. Gutiérrez. A Memory Consumption Benchmark for MPI Implementations. `https://github.com/hpc/mpimemu`, October 2017. Los Alamos National Laboratory.

[48] Samuel K. Gutiérrez. memnesia: MPI Memory Consumption Analysis Utilities. `https://github.com/hpc/mpimemu/tree/master/memnesia`, Oct 2017. Los Alamos National Laboratory.

[49] Samuel K. Gutiérrez. The Quo Runtime Library. `https://github.com/lanl/libquo`, January 2017. Los Alamos National Laboratory LA-CC-13-076.

[50] Samuel K. Gutiérrez, Dorian C. Arnold, Kei Davis, and Patrick McCormick. On the Memory Attribution Problem: A Solution and Case Study Using MPI. *Journal on Concurrency and Computation: Practice and Experience (ExaMPI Special Issue Paper)*. To Appear.

[51] Samuel K. Gutiérrez, Kei Davis, Dorian C. Arnold, Randal S. Baker, Robert W. Robey, Patrick McCormick, Daniel Holladay, Jon A. Dahl, R. Joe Zerr, Florian

Weik, and Christoph Junghans. Accommodating Thread-Level Heterogeneity in Coupled Parallel Applications. In *2017 IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Orlando, Florida, 2017.

[52] Francois Gygi, Erik W Draeger, Martin Schulz, Bronis R De Supinski, John A Gunnels, Vernon Austel, James C Sexton, Franz Franchetti, Stefan Kral, Christoph W Ueberhuber, et al. Large-Scale Electronic Structure Calculations of High-Z Metals on the BlueGene/L Platform. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 45. ACM, 2006.

[53] Salman Habib, Vitali Morozov, Hal Finkel, Adrian Pope, Katrin Heitmann, Kalyan Kumaran, Tom Peterka, Joe Insley, David Daniel, Patricia Fasel, et al. The Universe at Extreme Scale: Multi-Petaflop Sky Simulation on the BG/Q. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 4. IEEE Computer Society Press, 2012.

[54] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[55] Charles Antony Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[56] Michael Hofmann and Gudula Rünger. An In-Place Algorithm for Irregular All-to-All Communication with Limited Memory. In *European MPI Users' Group Meeting*, pages 113–121. Springer, 2010.

[57] Edwin Hou, Nirwan Ansari, and Hong Ren. A Genetic Algorithm for Multiprocessor Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):113–120, 1994.

[58] Xiche Hu, William L Hase, and Tony Pirraglia. Vectorization of the General Monte Carlo Classical Trajectory Program VENUS. *Journal of computational chemistry*, 12(8):1014–1024, 1991.

[59] Chao Huang, Orion Lawlor, and Laxmikant V Kale. Adaptive MPI. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 306–322. Springer, 2003.

[60] Emmanuel Jeannot and Guillaume Mercier. Near-Optimal Placement of MPI Processes on Hierarchical NUMA Architectures. In *European Conference on Parallel Processing*, pages 199–210. Springer, 2010.

[61] Emmanuel Jeannot, Guillaume Mercier, and François Tessier. Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques. *IEEE Transactions on Parallel and Distributed Systems*, 25(4):993–1002, 2014.

[62] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[63] Erich L. Kaltofen. *The "Seven Dwarfs" of Symbolic Computation*, pages 95–104. Springer Vienna, Vienna, 2012.

[64] Ian Karlin, Abhinav Bhatele, Bradford L. Chamberlain, Jonathan Cohen, Zachary Devito, Maya Gokhale, Riyaz Haque, Rich Hornung, Jeff Keasler, Dan Laney, Edward Luke, Scott Lloyd, Jim McGraw, Rob Neely, David Richards, Martin Schulz, Charle H. Still, Felix Wang, and Daniel Wong. Lulesh programming model and performance ports overview. Technical Report LLNL-TR-608824, December 2012.

[65] Ian Karlin, Jeff Keasler, and Rob Neely. LULESH 2.0 Updates and Changes. Technical Report LLNL-TR-641973, August 2013.

[66] Michael Karo, Richard Lagerstrom, Marlys Kohnke, and Carl Albing. The Application Level Placement Scheduler. *Cray User Group*, pages 1–7, 2006.

[67] Hironori Kasahara and Seinosuke Narita. Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing. *IEEE Transactions on Computers*, 33(11):1023–1029, 1984.

[68] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. Technology-Driven, Highly-Scalable Dragonfly Topology. In *ACM SIGARCH Computer Architecture News*. IEEE Computer Society, 2008.

[69] Géraud Krawezik. Performance Comparison of MPI and Three OpenMP Programming Styles on Shared Memory Multiprocessors. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 118–127. ACM, 2003.

[70] Adam J. Kunen, Teresa S. Bailey, and Peter N. Brown. *Kripke—A Massively Parallel Transport Mini-App*. United States. Department of Energy., 2015.

[71] Argonne National Laboratory. Hydra Process Management Framework, June 2014.

[72] Charles E. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE transactions on Computers*, 100(10):892–901, 1985.

[73] John M. Levesque, Ramanan Sankaran, and Ray Grout. Hybridizing S3D into an Exascale Application Using OpenACC: An Approach for Moving to Multi-petaflops and Beyond. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 15:1–15:11. IEEE Computer Society Press, 2012.

[74] Xiang-Ke Liao, Zheng-Bin Pang, Ke-Fei Wang, Yu-Tong Lu, Min Xie, Jun Xia, De-Zun Dong, and Guang Suo. High Performance Interconnect Network for Tianhe System. *Journal of Computer Science and Technology*, 30(2):259–272, 2015.

[75] Franklin T Luk. Computing the Singular-Value Decomposition on the ILLIAC IV. *ACM Transactions on Mathematical Software (TOMS)*, 6(4):524–539, 1980.

[76] Charles L Mader. FORTRAN BKW: A Code for Computing the Detonation Properties of Explosives. Technical report, Los Alamos Scientific Laboratory, N. Mex., 1967.

[77] Evangelos P. Markatos and Thomas J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, 1994.

[78] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.1, June 2015.

[79] Robin Milner. Elements of Interaction: Turing Award Lecture. *Commun. ACM*, 36(1):78–89, 1993.

[80] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. Number 6, pages 89–100. ACM Sigplan Notices, 2007.

[81] Marc Pérache, Patrick Carribault, and Hervé Jourdren. MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 94–103. Springer, 2009.

[82] Marc Pérache, Hervé Jourdren, and Raymond Namyst. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In *European Conference on Parallel Processing*, pages 78–88. Springer, 2008.

[83] Howard Pritchard, Igor Gorodetsky, and Darius Buntinas. A uGNI-Based MPICH2 Nemesis Network Module for the Cray XE. Proceedings of the European MPI Users' Group Meeting, 2011.

[84] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436, 2009.

[85] Rolf Rabenseifner. Hybrid Parallel Programming: Performance Problems and Chances. In *Proceedings of the 45th Cray User Group Conference, Ohio*, pages 12–16, 2003.

[86] Philip C Roth, Dorian C Arnold, and Barton P Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 21. ACM, 2003.

[87] Larry Rudsinski and Gail W Pieper. Evaluating Computer Program Performance on the CRAY-1. Technical report, Argonne National Laboratory, 1979.

[88] Ponnuswamy Sadayappan and Fikret Ercal. Nearest-Neighbor Mapping of Finite Element Graphs onto Processor Meshes. *IEEE Transactions on Computers*, 100(12):1408–1424, 1987.

[89] Chien-Chung Shen and Wen-Hsiang Tsai. A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion. *IEEE Transactions on Computers*, 100(3):197–203, 1985.

[90] Albert Solernou, Jeyarajan Thiyagalingam, Mihai C. Duta, and Anne E. Trefethen. The Effect of Topology-Aware Process and Thread Placement on Performance and Energy. In Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer, editors, *Supercomputing*, pages 357–371, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[91] Mark S Squillante and Edward D Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, 1993.

[92] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. *ACM Transactions on Computer Systems (TOCS)*, 23(1):77–110, 2005.

[93] Rajeev Thakur and William Gropp. Test Suite for Evaluating Performance of MPI Implementations That Support MPI_THREAD_MULTIPLE. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 46–55. Springer, 2007.

[94] The Ohio State University. MVAPICH Benchmarks. `http://mvapich.cse.ohio-state.edu/benchmarks`, September 2017.

[95] The smem authors. smem memory reporting tool. `https://www.selenic.com/smem/`, October 2017.

[96] Dominique Thiebaut and Harold S. Stone. Footprints in the Cache. *ACM Transactions on Computer Systems (TOCS)*, 5(4):305–329, 1987.

[97] Top500.org. Top500 Supercomputing Sites, March 2018.

[98] Jesper Larsson Träff. Compact and Efficient Implementation of the MPI Group Operations. In *European MPI Users' Group Meeting*, pages 170–178. Springer, 2010.

[99] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[100] Bart Veltman, BJ Lageweg, and Jan Karel Lenstra. Multiprocessor Scheduling with Communication Delays. *Parallel Computing*, 16(2-3):173–182, 1990.

[101] Florian Wende, Thomas Steinke, and Alexander Reinefeld. The Impact of Process Placement and Oversubscription on Application Performance: A Case Study for Exascale Computing. In *Proceedings of the 3rd International Conference on Exascale Applications and Software*, EASC '15, pages 13–18, 2015.

[102] Pete Wyckoff and Jiesheng Wu. Memory Registration Caching Correctness. Number 2, pages 1008–1015. IEEE, 2005.

[103] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[104] Judicael A Zounmevo and Ahmad Afsahi. A Fast and Resource-Conscious MPI Message Queue Mechanism for Large-Scale Jobs. *Future Generation Computer Systems*, 30:265–290, 2014.