UNIVERSITY OF CALIFORNIA
SANTA CRUZ

A SYSTEMS-CENTRIC APPROACH FOR IMPROVING SCIENTIFIC SIMULATION PERFORMANCE

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE AND ENGINEERING

by

Nilesh M Negi

June 2023

The Thesis of Nilesh M Negi
is approved:

Professor Scott Beamer, Chair

Professor Tyler Sorensen

Professor Nicholas Brummell

Peter Biehl
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

A Systems-Centric Approach for Improving Scientific Simulation Performance

by

Nilesh M Negi

Easy access to powerful computational resources (cloud platforms and supercomputers) has revolutionized scientific research in the 21st century. However, these resources are often under-utilized, resulting in sub-optimal performance and increased operational costs. But, improving application performance is a complex task, as various factors such as mismatch between hardware configuration, runtime environment, and source code can contribute to sub-optimal performance.

To address these challenges, our study describes a 'profile-and-optimize' approach that leverages profiling data obtained by running applications on supercomputing systems. By systemically analyzing this data and modifying both the runtime environment and source code, we aim to improve application performance and scalability. We apply our methodology to four case studies and showcase performance improvements by comparing baseline results with optimized results obtained through our approach. Through our methods and the insights gained from the case studies, we provide a valuable resource for optimizing similar applications. By following our process, researchers and developers can reduce application runtime, maximize the utilization of computational resources, and run finer-grained simulations.

To my parents, my sister, my grandparents

and to the amazing community of High-Performance Computing

Acknowledgments

I want to start by thanking my advisor and committee members. Prof. Scott Beamer and Prof. Tyler Sorensen have been guiding me in Student Cluster Competitions since my first quarter at UC Santa Cruz. Not-So-Slow Slugs (UC Santa Cruz's team for Student Cluster Competitions) would not have been possible without their support and encouragement. Prof. Nic Brummell taught me valuable lessons in his High-Performance Computing course that improved my understanding of Fortran and MPI programming. I am also grateful to my committee for all the time and effort spent reviewing my work.

I want to thank my Not-So-Slow Slugs teammates: Yiwei Yang, Esteban Ramos, Jessica Dagostini, Matthew Boisvert, Connor Masterson, Waylon Peng, Nathaniel Larsen, Victor Chen, Aidan Au-Yeung, Alexander Beloiu, Mark Forbush. I learned a lot from our brainstorming sessions and the time spent during competitions. Thank you for laying the foundations of the UCSC HPC club. I would also like to thank the Student Cluster Competition organizers and advisors for introducing and teaching the nuances of the scientific computing applications used in this study.

I am thankful to my parents and my sister for having my back and for emotional support during difficult times. I am grateful to my grandparents for instilling a sense of scientific curiosity and for always encouraging me to pursue graduate studies.

Last but not least, I would like to thank my roommates, friends, and colleagues at UC Santa Cruz. I am grateful for all the help, advice, and technical discussions.

Chapter 1

Introduction

1.1 Need for Application Profiling

In the past ten years, the fields of Scientific Computing and Machine Learning have made significant advancements in various domains by improving algorithm/model design, generating and ingesting larger input datasets, using fine-grained sampling, and increasing computational capacity[30]. These workloads are compute-intensive and require substantial resources[11], often run for many hours, and cost millions of dollars[42][69] to operate. Therefore, it is crucial to identify and optimize sub-optimal software implementations and inefficient use of hardware resources.

Traditionally, hardware and software design have been treated as separate processes, with hardware engineers designing the physical components of a system and software engineers developing the programs that run on that hardware. As a result of these siloed development efforts, we often observe inefficient utilization of resources

and poor application performance on modern computing systems. However, with the growing need for computing power to support finer-grained scientific simulations, very large input datasets, and the latest algorithmic innovations[51], the boundaries between hardware and software are blurring, necessitating a more integrated and collaborative approach, termed **hardware-software co-design**[60]. In hardware-software co-design, the design process is driven by both hardware and software constraints and requirements. This approach recognizes that the performance and efficiency of a system depend not only on the capabilities of the hardware but also on how the software utilizes these hardware capabilities.

Program analysis and profiling tools play a vital role in helping understand the intended program behavior and unintended side effects. These tools are essential for computer architects to evaluate the performance of programs on new architectures. Likewise, software developers rely on these tools to analyze their programs and identify critical sections of code. Understanding these interactions and optimizing these dependencies between hardware and software results in higher performance, faster execution, and more efficient resource utilization, which helps reduce operational costs and enable larger models and finer granularity simulations. **Application Profiling** is the first step in this performance optimization process. It can help identify code hotspots, communication hotspots, and system bottlenecks. Insights from these profiling results help improve the application source and runtime environment to enable faster execution time and more efficient resource utilization. This can, in turn, improve scalability and work with larger inputs and finer granularity in scientific simulations.

This study presents a profiling-based methodology for tuning application runtime and modifying source code to improve performance and demonstrates its effectiveness using four case studies on diverse supercomputing architectures.

1.2 Thesis Overview

Chapter 2 (Background) provides a brief overview of High-Performance Computing systems, scientific computing applications, and programming models.

Chapter 3 (Methodology) talks about the steps involved in application profiling and introduces the profiling tools used in this study.

Chapter 4 (Evaluation and Results) describes the scientific computing applications and supercomputing systems used in this study. We then present baseline results and profiling data for each case study and showcase the performance improvements achieved by using insights from profiling.

Chapter 5 (Related Work) talks about related work, draws analogies with performance optimization in large-scale Machine Learning, and talks about Student Cluster Competitions.

Finally, Chapter 6 (Conclusion) concludes this study. This includes conclusions to be drawn as a result of this study and discussions of future applications.

Chapter 2

Background

This section provides a background on High-Performance Computing (HPC), the characteristics of HPC systems, and information necessary for understanding application profiling.

2.1 High-Performance Computing (HPC) systems

High-Performance Computing (HPC) or Supercomputing refers to the practice of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large problems in science, engineering, or business[38].

Traditionally, this aggregation of compute power was achieved with monolithic systems and a **scale-up (vertical scaling)** methodology, which involves adding more resources to a single system to enhance its performance. This typically involves upgrading or adding components to an existing machine, such as increasing the number of

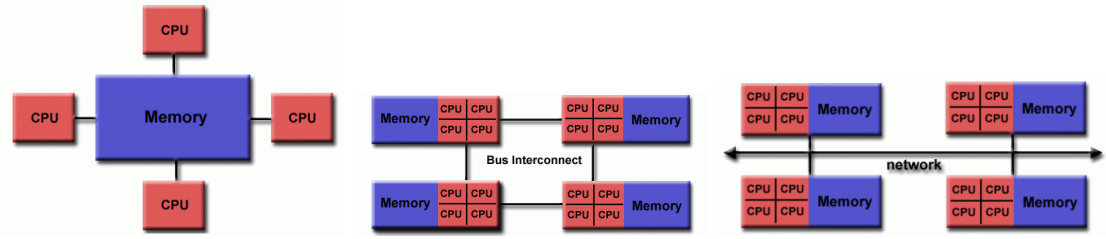
processors, increasing processor operating frequency, and increasing memory (DRAM), etc., on a single system. The idea is to make the existing system more powerful or capable of handling increased workloads. However, vertical scaling is constrained by the physical limits on how much the CPU or memory can be expanded, cost considerations, as well as observational laws like Moore's Law and Dennard Scaling.

To overcome these limitations of vertical scaling, HPC systems switched to a **scale-out (horizontal scaling)** methodology, which involves adding more machines or nodes to a system to handle larger workloads. Rather than upgrading a single system, scale-out focuses on distributing the workload across multiple machines and adding more nodes to form a distributed architecture. This choice was also influenced by the inherent parallelism of scientific simulations and machine learning workloads - the problem size can be divided into smaller subsets that can be computed individually. Additionally, horizontal scaling results in higher fault tolerance and resilience compared to a single scale-up system. If a single machine fails, it can cause a complete system outage until the hardware is repaired or replaced. In contrast, scale-out architectures can distribute workloads across multiple machines, providing better fault tolerance and resilience. This resulted in the era of parallel processing using multiple nodes.

Horizontal scaling is a better choice than vertical scaling in terms of cost-effectiveness, scalability, and overcoming hardware limitations. However, using distributed architectures introduce overheads like synchronization, communication, and the need for parallel programming frameworks and additional software libraries. HPC and ML workloads rely on OpenMP for shared memory systems and Message Passing

Interface (MPI) for distributed memory systems. Additionally, heterogeneous systems (where programs run on both CPUs and GPUs) require additional parallel programming frameworks like CUDA (NVIDIA GPUs), ROCm (AMD GPUs), and OpenACC (NVIDIA and AMD GPUs).

There are two types of Shared Memory systems: Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA). UMA systems do not scale-up reasonably with additional CPUs as the central bus is quickly saturated, so NUMA systems were introduced where memory is distributed physically but shared logically and connected via the system bus. Distributed Memory systems consist of NUMA Shared Memory nodes interconnected via a communication network. Figure 2.1[41] shows a visual representation of UMA Shared Memory systems, NUMA Shared Memory systems, and Distributed Memory systems.



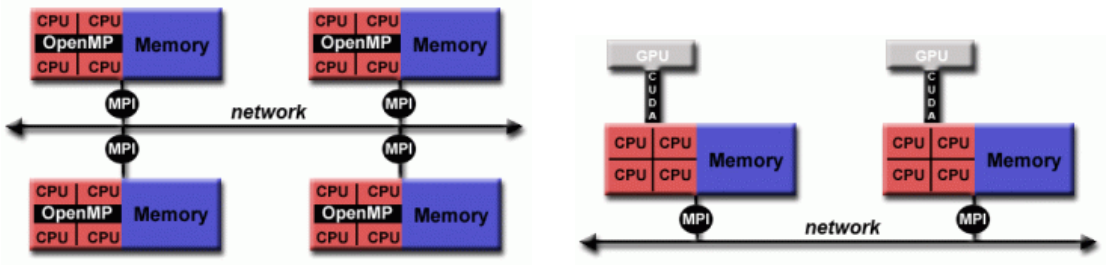
(a) UMA Shared Memory (b) NUMA Shared Memory (c) Distributed Memory[41]

Figure 2.1: Memory configurations in HPC systems

OpenMP is a popular parallel programming framework for shared memory systems (all processors can access system memory as global address space). It provides an API based on compiler directives and library routines for writing multi-threaded

applications in C, C++, and Fortran[5]. Message Passing Interface (MPI) is a message-passing standard for shared and distributed memory systems (processors communicate over a network to access inter-process memory). This MPI standard defines the syntax and semantics of library routines that are used for writing message-passing programs in C, C++, and Fortran[25].

Typically, on HPC systems, OpenMP provides lower overhead than MPI for shared memory systems (single node). So, hybrid runs, where OpenMP is used for intra-node parallelism and MPI is used for inter-node parallelism, offer the best performance and scalability on large-scale HPC systems. On heterogeneous systems (CPU+GPU), MPI is used for inter-node communication, and GPU programming frameworks (like CUDA[19] for NVIDIA GPUs, ROCm[33] for AMD GPUs, oneAPI[15] for Intel GPUs) provide CPU-GPU communication within the node. Figure 2.2[41] shows a visual representation of parallel programming paradigms used in homogeneous CPU-only systems and heterogeneous CPU=GPU systems.



(a) CPU-only systems[41] (b) Heterogeneous (CPU+GPU)[41]

Figure 2.2: Hybrid parallelism using MPI and OpenMP/CUDA

2.2 High-Performance Computing (HPC) applications

In parallel programs and HPC applications, the initial stage involves dividing the entire problem into distinct segments of work that can be assigned to multiple processes. This is known as decomposition or partitioning and can be accomplished in two ways: **domain decomposition** and **functional decomposition**. By employing these methods, the computational workload can be effectively distributed among processes, thus facilitating efficient parallel processing.

In domain decomposition, the data associated with a problem is decomposed and distributed amongst processes. Each process then works individually on a portion of the data and communicates intermediate values with other processes if/when needed. This is sometimes referred to as **data parallelism**, especially in Data Science and Distributed Machine Learning. In functional decomposition, the emphasis lies on the computation itself rather than the data involved in the computation. The problem is decomposed and distributed based on the specific tasks or computations that need to be executed. Each individual task is then responsible for executing a portion of the overall work, contributing to the collective computation. This is sometimes referred to as **model parallelism**, especially in Distributed Machine Learning.

For problems that can be decomposed and executed in parallel with virtually no need for tasks to share data, little or no communication is required. These types of problems are often called 'embarrassingly parallel' or 'pleasingly parallel'. A common example of an embarrassingly parallel problem is 3D video rendering handled by

a Graphics Processing Unit (GPU), where each frame or pixel can be handled with no inter-dependency. Some other examples would be adding two matrices or vectors, protein folding software that can run on any computer with each machine doing a small piece of the work, generation of all subsets, random numbers, and Monte Carlo simulations[26].

With this background knowledge, we can now discuss application profiling and performance optimization strategies for scientific computing applications running on distributed memory HPC systems.

Chapter 3

Methodology

3.1 Overview of Application Profiling

In High-Performance Computing (HPC), the pursuit of flawless code functionality goes hand in hand with a relentless quest for maximizing performance. To achieve maximum performance, we not only need to guarantee functional correctness but also ensure optimal utilization of resources. This demands a meticulous approach where applications and their software stack are fine-tuned to harmonize with hardware limitations and operational requirements. By embracing this synergy, HPC developers and users can achieve maximum application performance, pushing the boundaries of what's possible. However, this task becomes particularly challenging as hardware evolves over time, necessitating further optimization of applications. Also, it is impractical to account for all real-world use cases. So, in practice, developers create benchmarks specifically designed to assess the performance, such as execution time, within

an environment that mimics the real-world environment. These benchmarks serve as a valuable proxy by providing insights into application characteristics and identifying potential bottlenecks that may degrade performance during different operational scenarios. An effective benchmark should accurately represent the actual workload and not just rely on synthetic test cases.

At first glance, benchmarking may seem straightforward, often perceived as a simple comparison of execution times on different machines. However, to extract the maximum performance from emerging hardware, the application must undergo numerous tuning iterations. It requires more than just measuring raw execution time; it entails identifying the areas where the application spends the most time and exploring possibilities for further algorithmic improvements. The presence of heterogeneous systems adds additional complexities, emphasizing the importance of understanding critical paths and kernel execution. Hence, application profiling and performance tuning become indispensable components of the optimization process.

Application profiling enables developers to gain important insights into how efficiently their application is utilizing hardware and effectively diagnose potential bottlenecks contributing to poor performance.

3.2 Application Profiling Tools

There are several vendor-specific and third-party application profiling tools like IPM[47], Intel VTune Profiler[16], AMD μProf[32], NVIDIA Nsight Systems[20],

HPCToolkit[66], TAU[50], and many more. Most profiling tools operate in two modes: **offline profiling** via a command-line interface and **online profiling** via a GUI. Online profiling offers real-time profiling data for live runs but requires interactive access to systems. On the other hand, offline profiling collects and stores profiling data over the entire application run. This profiling data is then transferred to the local system for visualization and analysis. HPC systems are typically accessed via a remote connection, which is not well-suited for visualization tasks, so offline profiling is the preferred choice. For this study, we have used IPM, Intel VTune, and NVIDIA Nsight. We generate profiling data via the command-line interface that is transferred to the local system for analysis.

Integrated Performance Monitoring[47] (IPM) is a portable profiling infrastructure for parallel codes. It provides a low-overhead profile of application performance and resource utilization in a parallel program. IPM primarily focuses on communication, computation, and IO. In addition to timings, IPM provides MPI Communication topology and statistics for each MPI call and buffer size, arithmetic intensity metrics like Floating Point Operations Per second (FLOPs) via on-chip event counters, memory usage, network communication volume, and data written to and read from disk.

Intel VTune[16] is a performance analysis framework for serial and parallel applications. While it was originally intended for x86 architectures, the latest versions only support Intel CPUs and do not run on AMD CPUs. The first step is to use VTune's

Application Performance Snapshot (APS) for a quick view into various aspects of application performance, like MPI and OpenMP usage, CPU utilization, memory access efficiency, vectorization, I/O, and memory footprint. Apart from displaying key optimization areas, APS also displays potential bottlenecks and problem areas (very high Elapsed Time, low Instructions Per Cycle (IPC), or remarks on Memory Bound) and suggests specialized tools for tuning particular performance aspects. The next step is to identify code hotspots, i.e., the most time-consuming sections, and look into vector register utilization, memory access patterns, and MPI communication heatmap of these code hotspots. VTune provides individual components like Intel VTune Profiler for code hotspot analysis and Intel Trace Analyzer and Collector for identifying MPI load imbalance and communication hotspots.

NVIDIA Nsight Systems[20] is a system-wide performance analysis and profiling tool from NVIDIA. It is designed to visualize an application's algorithms, help identify the largest opportunities to optimize, and tune to scale efficiently across any quantity or size of CPUs and GPUs. Nsight Systems enables users to visualize CPU utilization, GPU utilization, CPU-GPU interactions, trace the execution of different GPU kernels, and network utilization.

3.3 Getting Started with Profiling

3.3.1 Baseline Single-node Runs

The first step in application profiling should be to build, run, and then profile out-of-the-box (baseline) single-node runs. For intra-node parallelism in these single-node runs, it is advisable to verify that the application source code supports using OpenMP directives and can be built with OpenMP support. Modern-day compilers like GNU compilers (gcc, g++, gfortran), LLVM (clang, clang++, flang), or vendor-specific compilers like Intel compilers (icc, icpc, ifortran), AMD compilers (aocc), and NVIDIA compilers (nvcc, nvc, nvc++, nvfortran), inherently support OpenMP, so it can be enabled using compiler optimization flags during the build process. As a best practice, a good starting place is using vendor-specific compilers optimized for target hardware, like using Intel compilers for Intel CPUs. Once the OpenMP-enabled build is successful, one can proceed with running and profiling the application. These initial results give us baseline performance numbers that can serve as the starting point for run-time optimizations like process pinning, memory pinning, and hybrid parallelism.

3.3.2 Baseline Multi-node Runs

The next step should be to build, run, and then profile out-of-the-box (baseline) multi-node runs based on pure MPI. For inter-node parallelism using MPI, one can choose between several open-source (OpenMPI, MVAPICH2, etc.) and vendor-specific (Intel MPI, NVIDIA NVHPC, HPE Cray MPI, etc.) implementations. Once it is

verified that the application source code supports MPI calls, one can use MPI compilers like mpicc, mpic++, mpifort/mpif90 (OpenMPI) or mpiicc, mpiicpc, mpiifort (Intel MPI) to build the application. These MPI compilers are simply wrappers that add in all the relevant compiler/linker flags and then invoke the underlying compiler. As a best practice, a good starting place is using vendor-specific MPI implementations optimized for target hardware, like using Intel MPI for Intel CPUs. Once the MPI-enabled build is successful, one can proceed with running and profiling the application at increasing node counts by using all processors per node. This method is used to observe **strong scaling** behavior where the problem size is fixed as the number of processors is increased to achieve faster runtime – ideally half the runtime for twice the number of processors. Scalability runs give us some baseline performance numbers and insights into the strong scalability of the application and the input dataset. Insights from scalability runs and initial profiling data serve as the starting point for run-time and MPI optimizations like process placement strategies, MPI communication strategies, and hybrid parallelism.

3.3.3 Analyzing Profiling Results

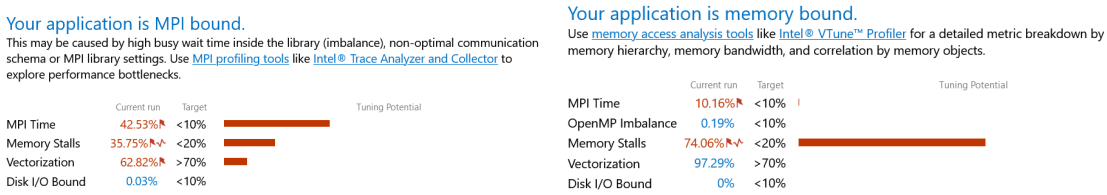
With most applications and corresponding input datasets, one does not obtain perfect linear scaling results from baseline single-node and multi-node runs. Typically, one observes inefficient utilization and bottlenecks in hardware resource utilization and sub-linear scaling if the input dataset is not large enough to utilize the compute capabilities of higher processor counts or if one encounters bottlenecks where computation is overshadowed by memory data movement or inter-node communication. To overcome

these bottlenecks, reducing the number of MPI processes per node or using hybrid runs (using MPI+OpenMP) is an effective strategy.

For efficient parallelism and utilization of resources in hybrid runs, one needs to find a good balance between the number of MPI processes and OpenMP threads per process. Having a large number of MPI processes can lead to inefficient domain decomposition for relatively small inputs as each MPI process gets a relatively small chunk of the input, where time spent on computation can be overshadowed by the overhead of MPI runtime. Additionally, having a large number of MPI processes with large input datasets can lead to memory imbalance (uneven memory usage across nodes) and memory bottlenecks (memory traffic due to data movement between processors and memory saturating bandwidth), resulting in intermittent idle processors during the application run. This occurs as the critical path in the application execution is influenced by system RAM speed which is lower than CPU processing speed, also known as **memory wall**. Another scenario can be communication imbalance (uneven distribution of MPI processes) and communication bottlenecks due to a large number of MPI processes when the time spent exchanging messages between MPI processes overshadows time spent in computation.

3.3.4 Optimization Strategies

Profiling insights from baseline single-node and multi-node results can point in the right direction and help define optimization strategies. One strategy can be using scaling-up techniques to improve and upgrade system components, like using faster



(a) Communication/MPI-bound (b) Memory-bound

Figure 3.1: Examples of insights from application profilers

RAM or faster network interconnects. However, this is not always feasible. So, another strategy is to transform a memory-bound or communication-bound bottleneck into a compute-bound bottleneck, which can, in turn, be resolved by using OpenMP threads or by using more nodes. Let's look at some scenarios:

Memory-bound applications As discussed earlier, when using a large number of MPI processes with a memory-bound application, performance can be bottlenecked by memory imbalance or memory bandwidth limitations, where a major portion of the execution time is spent in moving data between memory and processors while processors stay idle. For memory imbalance, one can look into profiler insights about memory usage per node and modify MPI process placement to better balance memory usage. In some cases, like functional decomposition, MPI process placement may not be sufficient to resolve memory imbalance, and one has to modify source code to ensure memory usage across nodes is as homogeneous and balanced as possible.

For memory bandwidth bottlenecks, using fewer MPI processes per node can be a solution, as it can reduce the overall amount of data moving between memory

and processors and transform the application to compute-bound. But, one drawback of reducing the number of MPI processes per node is inefficient usage of hardware resources as many CPU processors stay idle. This can be resolved in hybrid runs by using OpenMP threads to overcome the new compute-bound bottleneck, resulting in improved performance. However, one should ensure that OpenMP threads are placed on processors close to the processor used by MPI in order to utilize the same cache. There is a possibility that using additional OpenMP threads can degrade performance if the additional overhead of threading leads to cache contention (e.g., false sharing where one thread writes to memory that is very close to where another thread needs to read). In such a scenario, it is advisable to explore different domain decomposition schemes.

Communication-bound applications As discussed earlier, in a communication-bound application, performance can be bottlenecked by the network, where a major portion of the execution time is spent exchanging data between processes while processors stay idle. This leads to a poor computation/communication ratio. Profilers can provide information on the overall time spent in various MPI calls, communication topology (message sizes and time spent in point-to-point exchanges), and MPI imbalance during the application runtime. Based on these profiling metrics, one can modify MPI process placement by grouping MPI processes with the maximum number or maximum size of message exchanges and placing them on the same node to reduce network utilization.

Another reason for poor computation/communication ratio can be MPI syn-

chronization overhead as a result of time spent waiting for straggling MPI processes in point-to-point communication. This is often caused due to standard MPI calls, which rely on blocking communication, where the MPI function call does not return control until the data being sent/received has been copied out/into the buffer. To overcome this synchronization overhead for small messages, caching can be employed on the destination process for unmatched messages. Using non-blocking MPI calls is a more general solution, where the MPI function call immediately returns control and continues computation by asynchronously sending/receiving data. For correctness, one has to ensure that non-blocking communication calls do not re-use the send/receive buffers until the destination has been processed.

MPI collective calls, like Broadcast, Scatter, Gather, AllReduce, and AlltoAll, use blocking communication but enable higher efficiency due to their multi-cast messaging. To improve the computation/communication ratio, one can use the non-blocking variations of MPI collective calls, but this requires source code modifications. Alternatively, several MPI implementations offer different algorithmic choices for MPI collectives[54]. These algorithms can be chosen via runtime environment variables depending on insights gained from MPI profiling, network configuration, and application requirements.

In the following chapter, we introduce the scientific computing applications used in this study, talk about the hardware configuration of the testing environment, and present performance analysis and results based on the aforementioned profiling

methodology and optimization strategies.

Chapter 4

Evaluation and Results

In this study, we use four scientific computing applications: XCompact3D, Quantum ESPRESSO (QE), FluTAS, and POT3D. These are popular applications heavily used in computational fluid dynamics and molecular dynamics.

First, we introduce the testing environment used for running, profiling, and optimizing these applications. After a brief description of the supercomputing architecture, we introduce each of these four applications, present out-of-the-box baseline results, profiling analysis, and optimized results obtained from runtime and code modifications based on profiling data. For two applications (Quantum ESPRESSO and POT3D), we also conducted a CPU vs. GPU comparison, which shows the advantages of GPU acceleration for application performance.

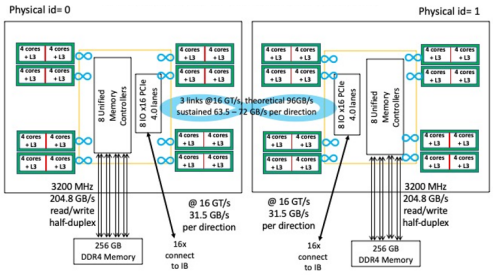
4.1 Testing Environment

For this study, simulations and performance profiling are conducted on the following supercomputing clusters:

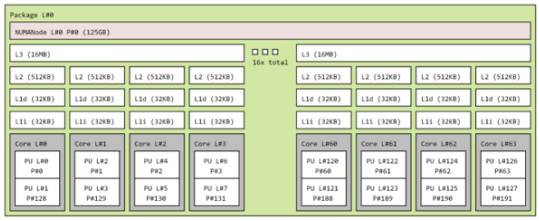
- Pittsburgh Supercomputing Center (PSC) **Bridges-2**
- University of Toronto's **Niagara**
- Erlangen National High-Performance Computing Center (FAU) **Fritz**

Bridges-2[7] is the largest supercomputer deployed at the Pittsburgh Supercomputing Center (PSC). It is intended for rapidly evolving and data-intensive research domains like AI/ML, data analytics, biomedical applications[8], and advanced networking. It is supported by the National Science Foundation (NSF) and is available at no cost for research and education through the NSF ACCESS/XSEDE program. Architecturally, Bridges-2 is a heterogeneous computing platform consisting predominantly of CPU-only nodes (based on AMD EPYC CPUs) and some GPU-accelerated nodes (based on NVIDIA Tesla V100 GPUs + Intel CascadeLake CPUs). It is interconnected by a novel Fat tree Clos topology with a modest 2.3:1 over-subscription using NVIDIA Mellanox Infiniband HDR-200Gbps adapters. Figure 4.1 shows the processor architecture and memory organization for Bridges-2 nodes.

Niagara is a homogeneous supercomputing cluster owned by the University of Toronto, and it has been optimized for throughput, ensuring efficient scaling, energy utilization, as well as network and storage capabilities for handling a variety of scientific



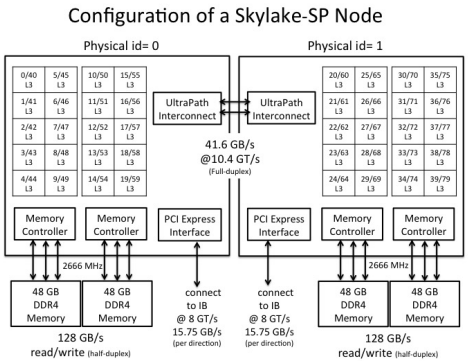
(a) Node configuration[28]



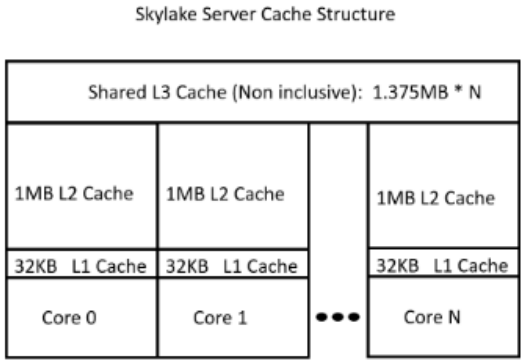
(b) Cache and Memory grouping

Figure 4.1: Bridges-2 AMD EPYC

workloads at large scales. It was the 53rd fastest supercomputer on the TOP500 list of June 2018 and occupies 177[65] on the (current) June 2023 Top500 list. Niagara uses Intel Skylake CPUs connected via NVIDIA Mellanox EDR-100Gbps Infiniband network in a Dragonfly+ topology.



(a) Node configuration[29]

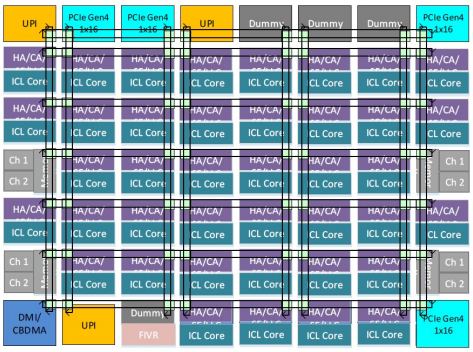


(b) Cache and Memory grouping[14]

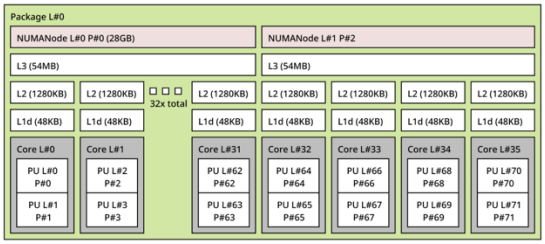
Figure 4.2: Niagara Intel SkyLake

FAU's Fritz is a homogeneous supercomputing cluster, named after FAU's founder Friedrich, Margrave of Brandenburg-Bayreuth. Financed by the German Re-

search Foundation (DFG), it placed 178[64] on the (current) June 2023 Top500 list and is used for research areas such as node-level performance engineering, performance modeling and tooling, numerical algorithms for sparse operations, and atomic structure simulations. Fritz is based on liquid-cooled Intel IceLake CPUs[63] connected via NVIDIA Mellanox HDR Infiniband network in a 1:4 blocking Fat tree topology.



(a) Node configuration[63]



(b) Cache and Memory grouping

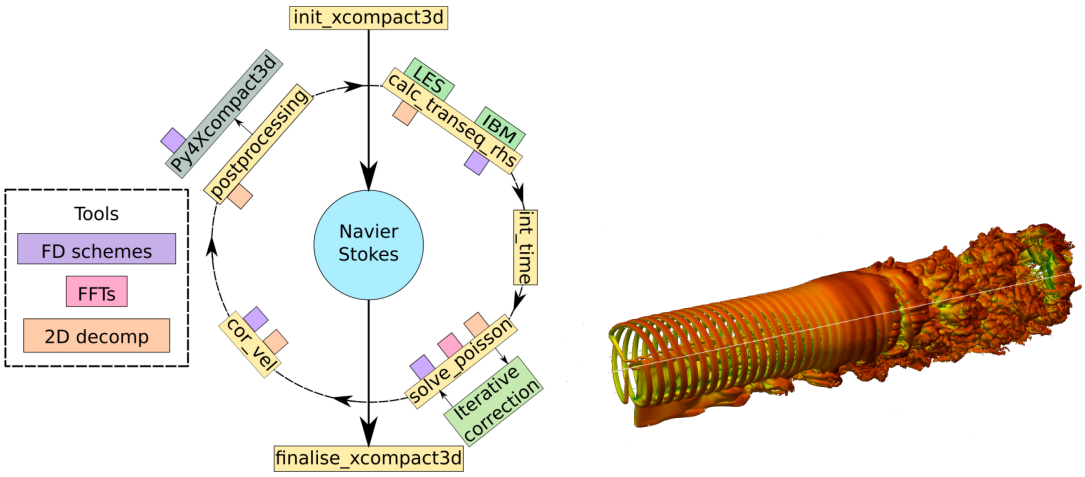
Figure 4.3: Fritz Intel IceLake

In this study, Bridges-2 and Niagara were used for the performance analysis of CPU-only runs of XCompact3D. Bridges-2 and Fritz were used for the performance analysis of CPU-only runs of QE, FluTAS, and POT3D. In addition, QE and POT3D were evaluated on Bridges-2 GPU nodes.

4.2 XCompact3D

4.2.1 Introduction

XCompact3D is a free, open-source Fortran90 framework consisting of high-order finite-difference flow solvers[3]. These solvers target Direct and Large Eddy Simulations (DNS/LES) to investigate turbulence and heat transfer problems. XCompact3D is currently able to solve the incompressible and low-Mach number variable density Navier-Stokes equations using sixth-order compact finite-difference schemes on a Cartesian mesh. Figure 4.4 shows the software architecture of XCompact3D and the visualization of wind-flow simulation in this study generated using ParaView.



(a) XCompact3D subroutines[3] (b) Simulation visualization

Figure 4.4: XCompact3D Software Architecture

Theoretically, XCompact3d can efficiently scale up to hundreds of thousands of CPU cores. It uses the open-source library 2DECOMP&FFT, a 2-dimensional pencil

decomposition and Fast Fourier Transform (FFT) framework that supports building large-scale parallel applications on distributed memory systems using MPI. For XCompact3D, the Poisson equation is solved in spectral space using modified wave numbers[43] and Dirichlet boundary conditions with high-order finite-difference schemes and FFT routines.

2D FFT operations are usually communication-bound (performance is bottlenecked by the network, where a major portion of the execution time is spent exchanging data between processes while processors stay idle). One of the reasons is that 2D FFT operations are based on a multi-dimensional (2D) global transpose of large matrices. From an MPI and domain decomposition perspective, this involves MPI AlltoAll communication, where all processes exchange data with each other via multi-casting, resulting in $O(P * log_2 P)$ message exchanges[49], where P is the number of MPI processes.

For this study, we use the open-source implementation of XCompact3D v4.0, available on GitHub, under the BSD 3-Clause license.

4.2.2 Build and Input description

XCompact3D does not require external software libraries and can be built by using only the Fortran compiler and MPI. But it supports using an external FFT library. Its source code includes the 2DECOMP&FFT library (generates generic FFT subroutines), which is used by default if an external library is not used during the build process.

The input dataset is a wind turbine benchmark where two generic wind tur-

bines are aligned and subject to an incoming uniform wind[4]. These model wind turbines have a three-bladed rotor with diameters of `DT1=0.944m` and `DT2=0.894m`. The small difference in rotor diameter stems from a slightly different hub geometry of the rigs. Apart from that, the blade geometry is exactly the same. Both turbines rotate counter-clockwise when observed from an upstream point of view. This experimental data was measured in the closed-loop wind tunnel at the Norwegian University of Science and Technology (NTNU), Trondheim.

4.2.3 Results

Cores	MPI ranks	Time (s)
128	128	3202.06
256	256	2310.95
512	512	1675.46

(a) Bridges-2

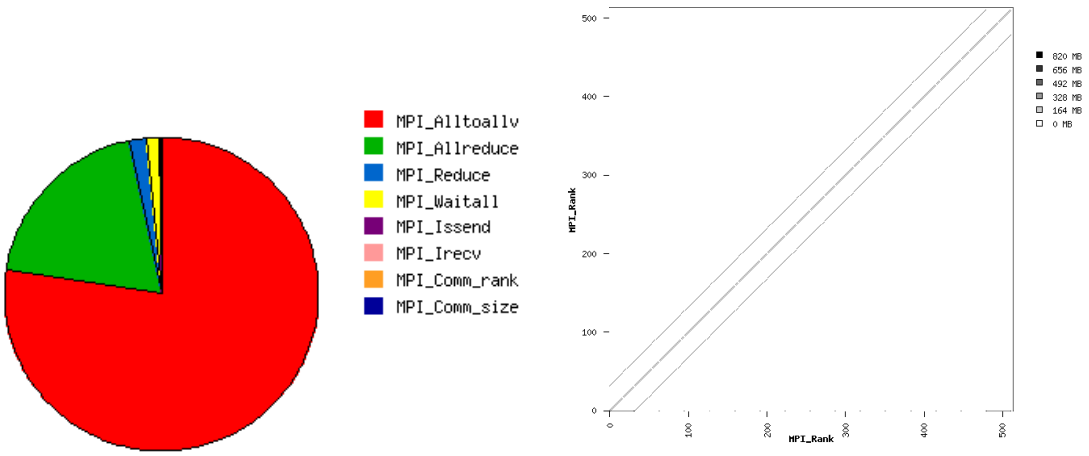
Cores	MPI ranks	Time (s)
40	40	16863.16
80	80	8361.69
160	160	5168.16

(b) Niagara

Table 4.1: XCompact3D Baseline results

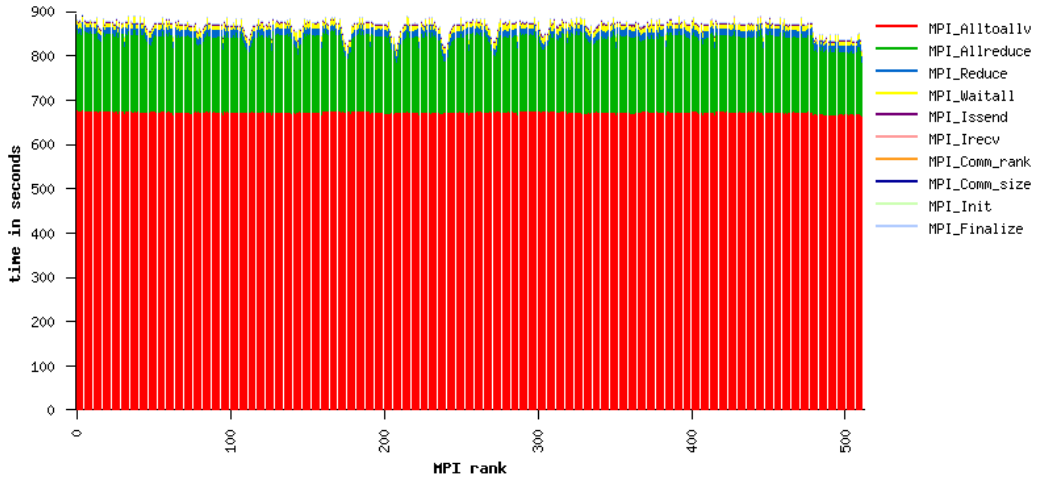
Baseline XCompact3D results on Bridges-2 and Niagara are in Table 4.1. Profiling data from the baseline results shows that **XCompact3D is communication-bound**. We observe that XCompact3D spends 40% of its runtime in MPI communication and 60% in computation. MPI communication is dominated by AlltoAllv (80% of MPI WallTime) and AllReduce (19.25% of MPI WallTime) calls, as seen in Figure 4.5.

Tracing the code execution shows that the bulk of this MPI communication oc-



(a) MPI calls

(b) MPI topology



(c) MPI imbalance

Figure 4.5: XCompact3D Profiling data

curs in the `decomp2d` subroutines that use `MPI_ALLTOALLV` for matrix transposition from X-axis to Y-axis, Y-axis to X-axis, Y-axis to Z-axis, and Z-axis to Y-axis. This blocking MPI communication requires synchronous pairwise exchanges of messages. Converting

these calls to non-blocking can improve the computation/communication ratio by using asynchronous `MPI_IALLTOALLV`, as it allows threads to continue doing computations while communication with another thread is pending. Using external FFT libraries like FFTW3, AMD-FFTW3, or Intel MKL that can generate non-blocking FFT routines optimized for the target hardware can also help improve performance.

Based on the hardware configuration, we choose **gfortran+HPC-X OpenMPI +AMD-FFTW3** for the optimized build on Bridges-2 (AMD), and **Intel Fortran+Intel MPI+Intel MKL** for the optimized build on Niagara (Intel). Additionally, we modify Fortran compiler flags in the Makefile to enable loop and math optimizations using `-Ofast -funroll-loops` on GNU Fortran (Bridges-2) and `-O3 -fp-model fast=2` on Intel Fortran (Niagara). To enable and optimize vector instructions for the target CPU, we use `-march=znver2` on Bridges-2 to target AMD's Zen2 microarchitecture, and `-xCORE-AVX512` on Niagara to target Intel's Skylake microarchitecture.

Vendor-specific MPI implementations like HPC-X OpenMPI and Intel MPI support specialized transport-layer protocols for intra-node and inter-node communication. The Mellanox Infiniband network can be tuned to use Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) capabilities[55] that can offload collective communication to the network and reduce CPU involvement. To further reduce the impact of network congestion in this communication-bound application, we instruct the SLURM HPC scheduler to allocate nodes close to each other using the `--contiguous` directive.

Optimized XCompact3D results on Bridges-2 and Niagara are in Table 4.2.

Cores	MPI ranks	Optimized Time (s)	Baseline Time (s)	Speedup
128	128	2585.56	3202.06	1.23
256	256	1787.65	2310.95	1.29
512	512	1162.71	1675.46	1.44
			Average Speedup	**1.32**

(a) Bridges-2

Cores	MPI ranks	Optimized Time (s)	Baseline Time (s)	Speedup
40	40	4208.73	16863.16	4.01
80	80	1975.51	8361.69	4.23
160	160	1125.31	5168.16	4.59
			Average Speedup	**4.27**

(b) Niagara

Table 4.2: XCompact3D Optimization results

Using third-party FFT implementations that generate tuned FFT subroutines, compiler optimizations to increase CPU resource utilization, and vendor-specific MPI features like SHARP to streamline communication over the Infiniband network improve the performance of XCompact3D by **32%** on Bridges-2 and **4.27x** on Niagara. This performance improvement on Niagara is largely due to using AVX-512 on Intel Skylake CPUs.

4.3 Quantum ESPRESSO (QE)

4.3.1 Introduction

Quantum ESPRESSO (QE), where ESPRESSO stands for "opEn Source Package for Research in Electronic Structure, Simulation, and Optimization", is a suite of

Fortran90 and C code for electronic-structure calculations and materials modeling based on density-functional theory, plane waves, and pseudopotentials[27].

For this study, we rely on Quantum ESPRESSO's Car-Parrinello (CP) package. CP is a type of computational method for ab-initio molecular dynamics that uses the fundamental laws of nature to simulate the motion of atoms in a system. This type of molecular dynamics simulation does not rely on empirical potentials or force fields to describe the interactions between atoms but rather calculates these interactions directly from the electronic structure of the system using quantum mechanics.

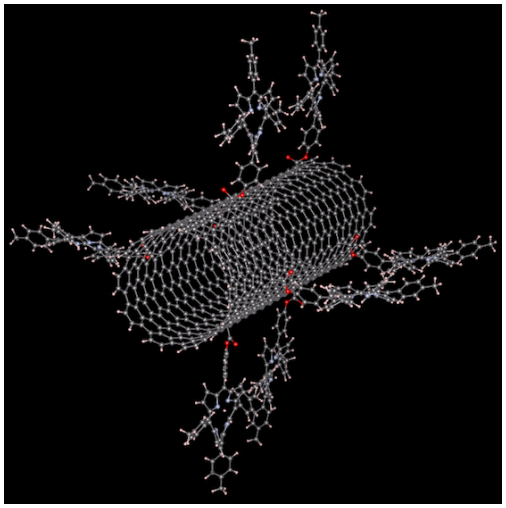


Figure 4.6: Visualization of a 1500-atom carbon nanotube system using QE[53]

QE typically consists of force calculations and FFT operations. It supports both 1D domain decomposition (slab decomposition) and 2D domain decomposition (pencil decomposition). By default, QE uses 1D slab decomposition, as 2D pencil decomposition is convenient and faster only when FFT operations are parallelized over a very large number of MPI processes. For example, the two input datasets used for

QE in this study have array dimensions $200 \times 200 \times 200$ and $180 \times 192 \times 240$, so the scalability of 1D slab decomposition flattens out when the number of MPI processes used for FFT parallelization exceeds the third FFT dimension (i.e. 200 and 240 respectively). 2D pencil decomposition is more communication-intensive than 1D slab decomposition, but it scales better. Note that the choice of the total number of MPI ranks is also influenced by memory usage during force calculations, so 2D pencil decomposition is not necessarily the clear-cut choice for a large number of MPI ranks.

For this study, we use the open-source implementation of Quantum ESPRESSO v7.1, available on GitHub, under the GPL-2.0 license.

4.3.2 Build and Input description

Quantum ESPRESSO does not require external software libraries and can be built with only a C compiler, a Fortran compiler (compliant with the F2008 standard), and MPI. But QE documentation recommends using an external Linear Algebra library (for BLAS and LAPACK subroutines) and an FFT library.

We use two input datasets for the Car-Parinello (cp) molecular dynamics simulation: **CP-W256** and **supercell_11layer**. CP-W256 is a small-size benchmark based on 256 H_2O (Water) molecules consisting of Hydrogen (H) and Oxygen (O) atoms. supercell_11layer is a medium-size benchmark based on 264 ZrO_2 molecules consisting of Zirconium (Zr) and Oxygen (O) atoms. The simulation starts 'from scratch' and runs for 10 Car-Parinello steps with a time step of 2.4189×10^{-17} seconds.

4.3.3 Results

Cores	MPI ranks	Baseline Time (s)	Comments
128	128	158.41	Runs with 1D slab decomposition
256	256	–	Fails with 1D slab decomposition
256	256	558.34	Runs with 2D pencil decomposition
512	512	–	Fails with 1D slab decomposition
512	512	760.50	Runs with 2D pencil decomposition

(a) CP-W256 on Bridges-2

Cores	MPI ranks	Baseline Time (s)	Comments
128	128	3204.00	Runs with 1D slab decomposition
256	256	–	Fails with 1D slab decomposition
256	256	2508.96	Runs with 2D pencil decomposition
512	512	–	Fails with 1D slab decomposition
512	512	1492.09	Runs with 2D pencil decomposition

(b) supercell_11layer on Bridges-2

Cores	MPI ranks	Baseline Time (s)	Comments
72	72	111.52	Runs with 1D slab decomposition
144	144	71.7	Runs with 1D slab decomposition
288	288	–	Fails with 1D slab decomposition
288	288	56.05	Runs with 2D pencil decomposition

(c) CP-W256 on Fritz

Cores	MPI ranks	Baseline Time (s)	Comments
72	72	2390.00	Runs with 1D slab decomposition
144	144	1328.12	Runs with 1D slab decomposition
288	288	–	Fails with 1D slab decomposition
288	288	816.30	Runs with 2D pencil decomposition

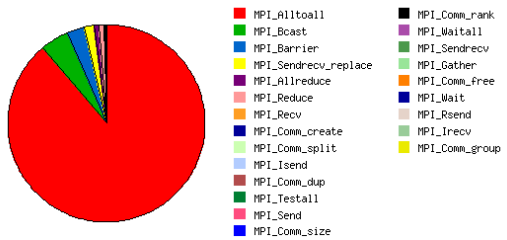
(d) supercell_11layer on Fritz

Table 4.3: Quantum ESPRESSO (QE) Baseline results

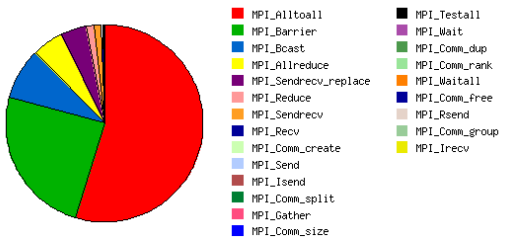
Baseline QE results on Bridges-2 and Fritz are in Table 4.3. We can observe that QE runs with a large number of MPI ranks fail to run with (default) 1D slab domain decomposition, as the number of CPUs used in these runs exceeds the number of planes of charge density along the 3rd dimension of the FFT grid – $200 \times 200 \times 200$ for CP-W256 and $180 \times 192 \times 240$ for supercell_11layer. This is a limitation of the benchmark. Using the runtime parameter `.pd. true` enables the 2D pencil decomposition scheme. Runs with 200 MPI ranks are successful, but performance and scalability degrade with respect to lower MPI process counts. Also, as the number of MPI ranks increases, the communication overhead of 2D pencil decomposition exceeds the time spent on computation. This difference is exacerbated when using the medium-size CP-W256 benchmark.

Profiling data from QE baseline results on Bridges-2 and Fritz (Figure 4.7 show that **QE is communication-bound** especially with 2D pencil domain decomposition. CP-W256 spends 89.54% of its runtime in MPI communication and 10.44% in computation. supercell_11layer spends 65.76% of its runtime in MPI communication and 34.21% in computation. For both, MPI communication is dominated by AlltoAll (90% of MPI WallTime for CP-W256 and 55% of MPI WallTime for supercell_11layer). Interestingly, supercell_11layer also shows 25% of MPI WallTime spent in MPI Barrier calls. This is a synchronization mechanism to ensure that all MPI calls reach a certain stage in communication before proceeding. Since supercell_11layer does not show any stark MPI Imbalance, this can be credited to memory stalls due to **memory bottlenecks**.

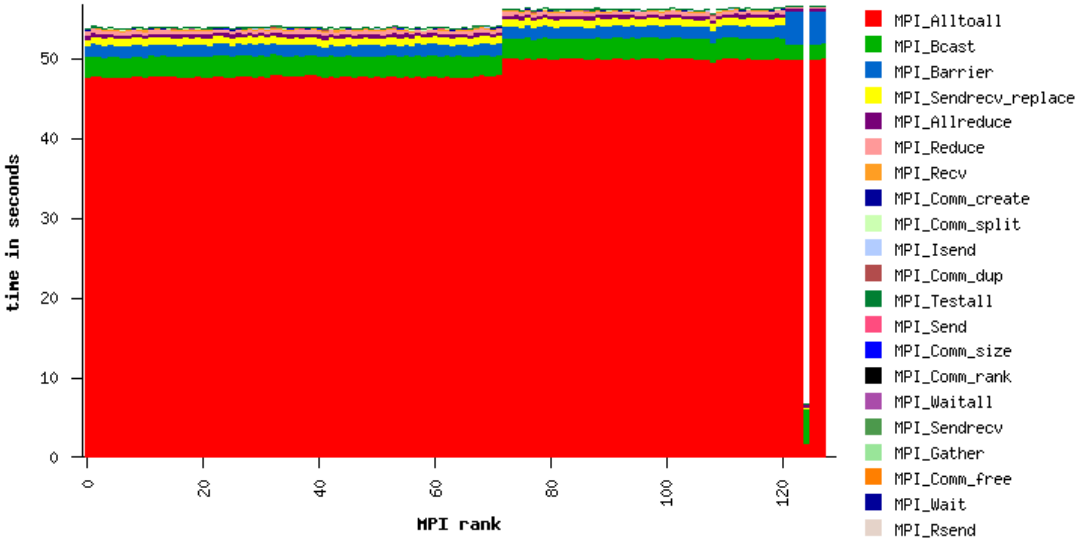
Since QE relies on dense matrices and FFT operations, it is recommended to



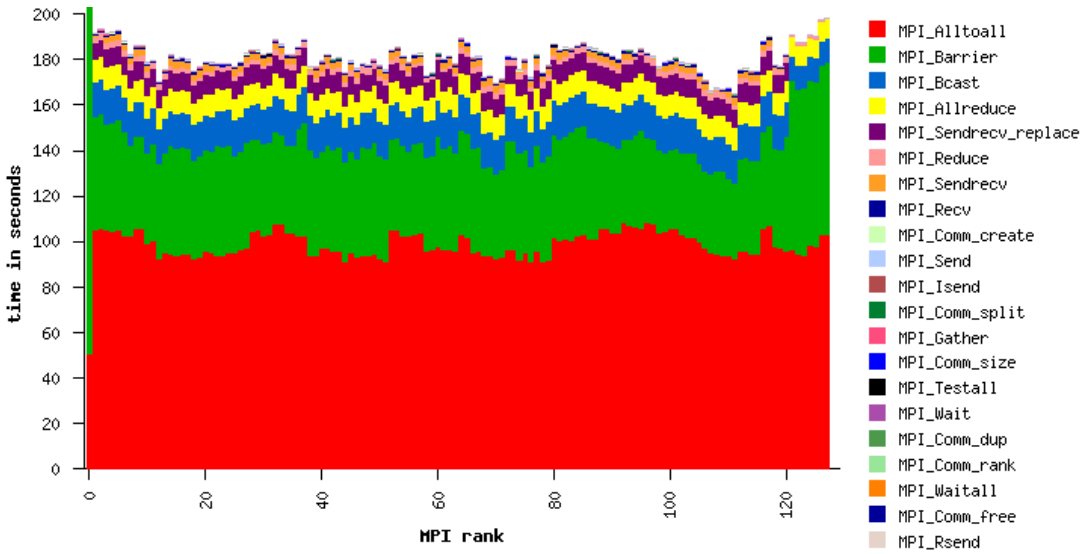
(a) MPI calls CP-W256



(b) MPI calls supercell_11layer



(c) MPI imbalance CP-W256



(d) MPI imbalance supercell_11layer

Figure 4.7: Quantum ESPRESSO (QE) Profiling data

use the Eigenvalue Solvers for Petaflop Applications (ELPA)[2] package, a specialized scientific library for electronic structure simulations that supports efficient and scalable direct eigensolvers for symmetric (hermitian) matrices.

Based on the hardware configuration and ELPA compatibility, we choose **Intel compilers+Intel MPI+Intel MKL+ELPA** for the optimized build on both Bridges-2 (AMD) and on Fritz (Intel). Additionally, we modify the Makefile to enable loop and math optimizations using `-O3 -fp-model fast=2`, enable the use of OpenMP using `-qopenmp`, and link the correct BLAS, LAPACK, and ScaLAPACK libraries from Intel MKL. To enable vector instructions optimized for the target CPU, we use `-march=core-avx2` on Bridges-2 to target AMD's Zen2 microarchitecture, and `-xCORE-AVX512` on Fritz to target Intel's Skylake microarchitecture.

QE performs better with 1D slab domain decomposition, so we use hybrid runs (MPI +OpenMP) to reduce the MPI communication bottleneck. Additionally, to reduce the impact of network congestion in this communication-bound application, we instruct the SLURM HPC scheduler to allocate nodes close to each other using the `--contiguous` directive.

Optimized QE results on Bridges-2 and Fritz are in Table 4.4. Using hybrid runs (MPI+OpenMP) to reduce MPI communication overhead without idling compute resources and building with the ELPA library to improve FFT computation improves the performance of QE by **5.53x** for CP-W256 and by **44%** for supercell_11layer on Bridges-2, and by **13%** for CP-W256 and **8%** for supercell_11layer on Fritz. We observe a larger performance delta on Bridges-2 as AMD CPUs typically have a larger core count

Cores	MPI ranks	Threads/rank	Optimized Time (s)	Baseline Time (s)	Speedup
128	128	1	143.31	158.41	1.10
256	128	2	108.32	558.34	5.15
512	128	4	73.43	760.50	10.35
				Average Speedup	**5.53**

(a) CP-W256 on Bridges-2

Cores	MPI ranks	Threads/rank	Optimized Time (s)	Baseline Time (s)	Speedup
128	32	4	3062.54	3204.00	1.05
256	64	4	1531.70	2508.96	1.63
512	128	4	898.92	1492.09	1.65
				Average Speedup	**1.44**

(b) supercell_11layer on Bridges-2

Cores	MPI ranks	Threads/rank	Optimized Time (s)	Baseline Time (s)	Speedup
72	72	1	109.74	111.52	1.02
144	72	2	66.46	71.7	1.08
288	72	4	43.58	56.05	1.29
				Average Speedup	**1.13**

(c) CP-W256 on Fritz

Cores	MPI ranks	Threads/rank	Optimized Time (s)	Baseline Time (s)	Speedup
72	36	2	2366.49	2390.00	1.01
144	72	2	1257.79 1	1328.12	1.06
288	72	4	701.90	816.30	1.16
				Average Speedup	**1.08**

(d) supercell_11layer on Fritz

Table 4.4: Quantum ESPRESSO (QE) Optimization results

than Intel CPUs, so the baseline runs use more MPI ranks on Bridges-2, which leads to higher MPI communication overhead when using 2D pencil decomposition for the baseline runs.

4.3.4 CPU vs GPU comparison

QE also supports GPU-enabled runs. Based on experiments, QE v7.1 does not support CUDA-aware MPI but QE v7.2 support CUDA-aware MPI. Table 4.5 compares the performance of CPU-only runs vs. GPU-accelerated QE runs on Bridges-2.

Configuration	CPUs	GPUs	MPI ranks	Threads/rank	Time (s)	Speedup
CPU	512	0	128	4	73.43	1.00
GPU	40	4	4	10	80.31	0.91
GPU (CUDA-MPI)	40	4	8	5	60.27	1.22

(a) CP-W256

Configuration	CPUs	GPUs	MPI ranks	Threads/rank	Time (s)	Speedup
CPU	512	0	128	4	898.92	1.00
GPU	40	4	4	10	264.89	3.39
GPU (CUDA-MPI)	40	4	4	10	Ran out of memory	

(b) supercell_11layer

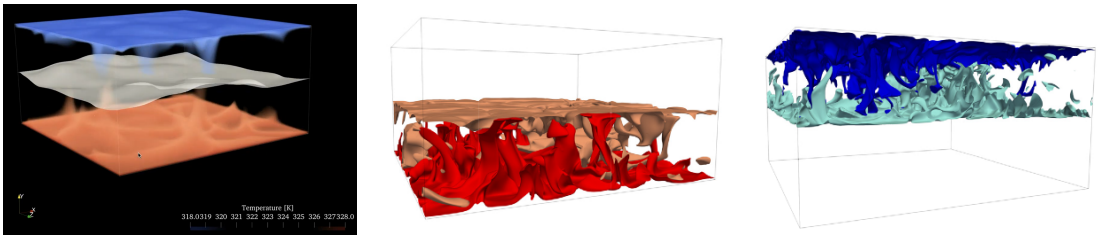
Table 4.5: Quantum ESPRESSO (QE) CPU-only vs GPU-accelerated comparison

GPU-accelerated QE using 4 NVIDIA V100 GPUs for the CP-W256 benchmark does not outperform the CPU-only runs, and performance degrades by 10%. Using CUDA-aware MPI improves performance, and QE runs **22%** faster than 4 CPU-only Bridges-2 nodes. This can be attributed to the small-size CP-W256 benchmark, as the cost of data movement between CPU-GPU overshadows acceleration on the GPUs. GPU-accelerated QE using 4 NVIDIA V100 GPUs for the supercell_11layer benchmark improves performance significantly by **3.39x**.

4.4 FluTAS

4.4.1 Introduction

FluTAS (Fluid Transport Accelerated Solver) is an open-source code targeting multi-phase fluid dynamics simulations. It uses a numerical method for fast, massively-parallel numerical simulations of turbulent flows. The corresponding code, CaNS (Canonical Navier–Stokes), benefits from the efficiency of the FFT for the finite-difference discretization of the pressure Poisson equation and allows for simulating a wide range of canonical flows like turbulent mixing and Rayleigh-Bénard (RB) convection[23]. FluTAS uses very efficient FFT solvers in problems with different combinations of homogeneous pressure boundary conditions. It relies on 2D pencil-like domain decomposition, which enables efficient massively parallel simulations.



(a) RB convection[23] (b) Hot fluid phase[23] (c) Cold fluid phase[23]

Figure 4.8: Visualization of Rayleigh-Bénard (RB) convection flow using FluTAS

For this study, we use the open-source implementation of FluTAS v1.5, available on GitHub, under the MIT license.

4.4.2 Build and Input description

FluTAS relies on the FFTW interface to perform Fast Fourier Transforms, so FFTW3 should be installed and linked with the Fortran compiler and MPI during the build process.

The input dataset is a two-layer Rayleigh-Bénard convection benchmark. This simulates the flow developed inside a fluid layer that is heated from below and cooled from above. It is driven by the density differences that arise due to the temperature variation inside the fluid. The grid size is $768 \times 284 \times 768$, and the simulation runs for 1000 timesteps.

4.4.3 Results

Cores	MPI ranks	Time (s)
128	128	3477.54
256	256	2340.78
512	512	1347.98

(a) Bridges-2

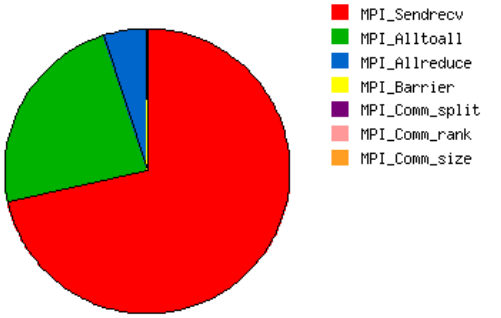
Cores	MPI ranks	Time (s)
72	72	2658.75
144	144	1660.51
288	288	954.92

(b) Fritz

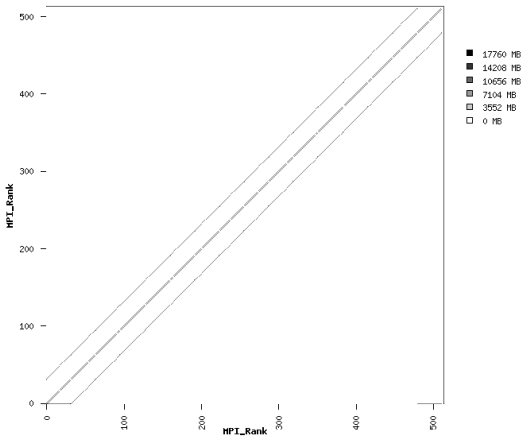
Table 4.6: FluTAS Baseline results

Baseline FluTAS results on Bridges-2 and Fritz are in Table 4.6. Profiling data from baseline results (Figure 4.9 show that **FluTAS is both compute-bound and communication-bound**. It spends 48.41% of its runtime in MPI communication and 51.55% in computation. We also observe that FluTAS is susceptible to communication imbalance as MPI calls are dominated by point-to-point SendRecv (70% of MPI

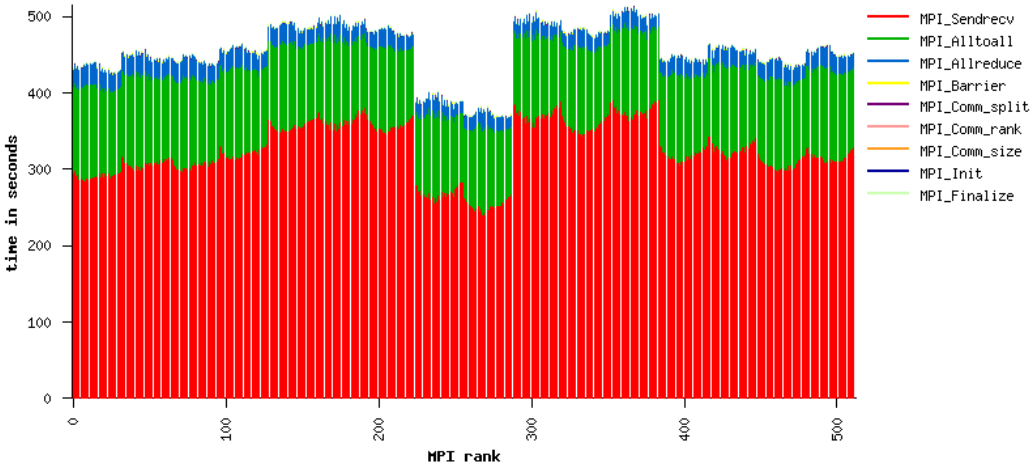
WallTime) and collective AlltoAll (23% of MPI WallTime).



(a) MPI calls



(b) MPI Topology



(c) MPI Imbalance

Figure 4.9: FluTAS Profiling data

Based on the hardware configuration, we choose **Intel Fortran+HPC-X MPI+FFTW3** for the optimized build on Bridges-2 (AMD), and **Intel Fortran+Intel**

MPI+FFTW3 for the optimized build on Fritz (Intel). Additionally, we modify the Fortran compiler flags in the Makefile to enable loop and math optimizations using `-O3 -fp-model fast=2` on Intel Fortran. To enable optimized vector instructions for the target CPU, we use `-march=core-avx2` on Bridges-2 to target AMD's Zen2 microarchitecture, and `-xCORE-AVX512 -qopt-zmm-usage=high` on Fritz to target Intel's IceLake microarchitecture.

Tracing the code execution shows that the bulk of computation occurs in the `cmpt_nor_curv_2o` subroutine, which solves the quadratic surface function for the symmetric Cartesian curvature tensor[31]. Profiling shows that this subroutine operates on scalar floating-point data. Vectorization can transform this scalar operation acting on individual data elements to concurrently operate on multiple data elements with a single instruction.

Since MPI communication is dominated by `MPI_SendRecv` in halo exchanges with nearest neighbors, MPI process placement strategies can help by mapping MPI processes to NUMA domains within the node. This ensures that MPI ranks will use intra-node shared memory instead of inter-node distributed memory for halo exchanges.

To further reduce the impact of network congestion and communication imbalance, we instruct the SLURM HPC scheduler to allocate nodes close to each other using the `--contiguous` directive.

Optimized FluTAS results on Bridges-2 and Fritz are in Table 4.7. Using compiler optimizations and enforcing the usage of vector instructions increases CPU resource utilization. Binding MPI ranks to NUMA domains on the same node transfers

Cores	MPI ranks	Optimized Time (s)	Baseline Time (s)	Speedup
128	128	2368.35	3477.54	1.47
256	256	1510.16	2340.78	1.55
512	512	795.63	1347.95	1.69
			Average Speedup	**1.57**

(a) Bridges-2

Cores	MPI ranks	Optimized Time (s)	Baseline Time (s)	Speedup
72	72	2310.22	2658.75	1.15
144	144	1337.16	1660.51	1.24
288	288	730.12	954.92	1.31
			Average Speedup	**1.23**

(b) Fritz

Table 4.7: Optimized results for FluTAS

the majority of point-to-point communication from the Infiniband network to the intra-node shared memory. This improves the performance of FluTAS by **57%** on Bridges-2 and **23%** on Fritz.

4.5 POT3D

4.5.1 Introduction

POT3D is a Fortran90 code that computes approximations of the solar coronal magnetic field (called potential fields) using observations of the solar surface magnetic field as a boundary condition[10]. It can be used to generate potential field source surfaces, potential field current sheets, and open field models. It is used in numerous studies of coronal structure and dynamics, solar/heliophysics, and space weather.

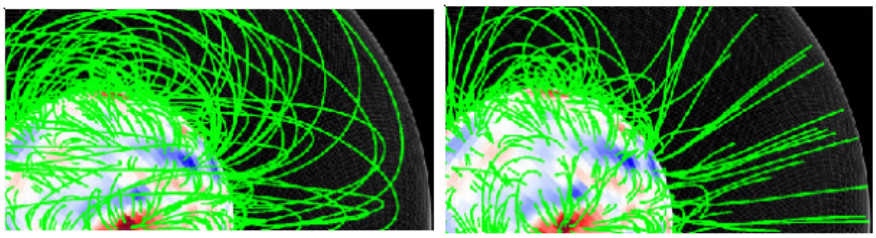


Figure 4.10: Visualization of field lines in a solar coronal magnetic field using POT3D

This Fortran90 code is highly parallelized using MPI and Fortran standard parallelism (`do concurrent`). It can also be offloaded to GPUs by using OpenMP and OpenACC, along with an option to use the NVIDIA cuSparse library for sparse matrix/vector operations. POT3D is included in the Standard Performance Evaluation Corporation's (SPEC)[6] SPEChpc(TM) 2021 benchmark suite[62].

POT3D solves the 3D Laplace equation on a non-uniform logically-rectangular spherical grid using an iterative Preconditioned Conjugate Gradient (PCG) method consisting of array operations, matrix-vector products, and dot products. Conjugate Gradient operations are typically memory-bandwidth bound (which involves system bottlenecks due to data movement to and from memory, as it is too large to store in caches).

For multi-node MPI, POT3D uses domain decomposition to distribute workload, where each MPI rank takes one subsection of the grid (as cubed as possible) and treats it as its own local domain for all operations. The only MPI communication needed is point-to-point messages for the local boundaries in the matrix-vector product and collective operations for the inner products and polar boundary conditions.

For this study, we use the open-source implementation of POT3D v3.1.0r, available on GitHub, under the Apache-2.0 license.

4.5.2 Build and Input description

POT3D uses the HDF5 file format for input/output, so HDF5 needs to be installed and linked with the Fortran compiler and MPI during the build process. Parallelism is achieved through the use of Fortran's standard parallelism (DC) along with OpenACC for loops that are not yet supported with DC.

The input dataset is the 'ISC2023' benchmark, derived from the 'small' POT3D benchmark run from the SPEChpc 2021 benchmark suite. A successful CG solver run should converge in `25112` steps.

4.5.3 Results

Cores	MPI ranks	Time (s)
128	128	6983.87
256	256	3754.16
512	512	2136.42

(a) Bridges-2

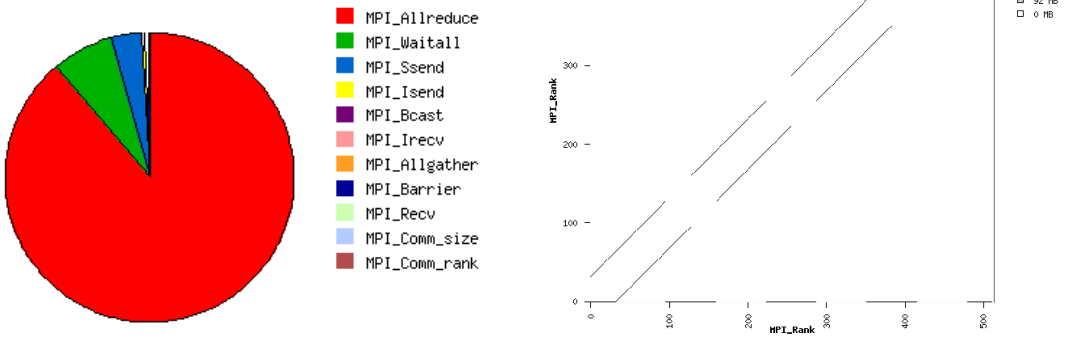
Cores	MPI ranks	Time (s)
72	72	5175.15
144	144	2781.74
288	288	1583.12

(b) Fritz

Table 4.8: POT3D Baseline results

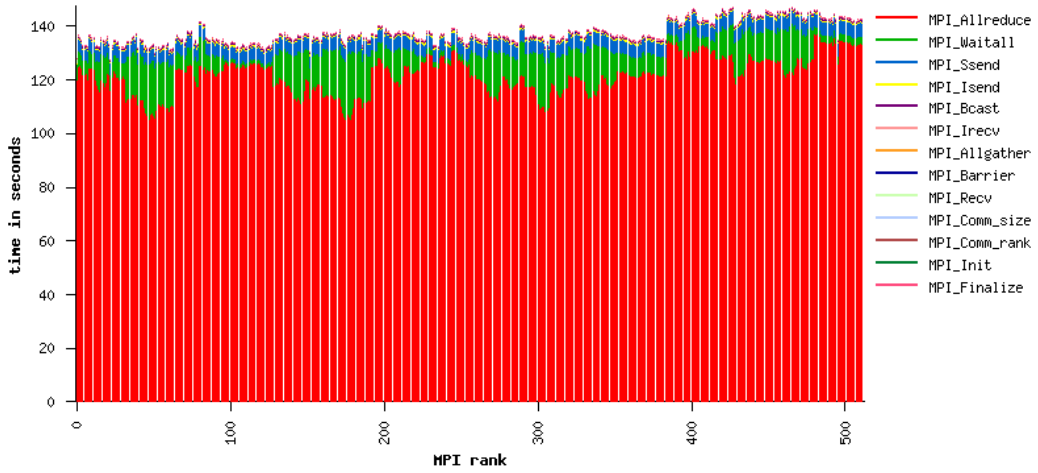
Baseline POT3D results on Bridges-2 and Fritz are in Table 4.8. Profiling data from baseline results (Figure 4.11) show that **POT3D is memory-bound**. It spends 16.62% in MPI communication and 83.30% in computation. This is expected,

as the Preconditioned Conjugate Gradient solver has a high Byte/FLOP ratio, and the amount of FLOPs that can be executed is limited by the maximum data throughput for sparse matrix-vector operations.



(a) MPI calls

(b) MPI topology



(c) MPI imbalance

Figure 4.11: POT3D Profiling data

Based on the hardware configuration, we choose **Intel Fortran+HPC-X MPI+HDF5** for the optimized build on Bridges-2 (AMD), and **Intel Fortran+Intel MPI+HDF5** for the optimized build on Fritz (Intel). Additionally, we modify the Fortran compiler flags in the Makefile to enable loop and math optimizations using `-O3 -fp-model fast=2` on Intel Fortran. To enable optimized vector instructions for the target CPU, we use `-march=core-avx2` on Bridges-2 to target AMD's Zen2 microarchitecture, and `-xCORE-AVX512` on Fritz to target Intel's IceLake microarchitecture. However, vectorization is not very effective as POT3D is a memory-bound application.

Tracing the code execution shows that the bulk of memory accesses occurs in the `call ax(p,ap,N)` subroutine used for matrix-vector multiplication, and the `r(i)=r(i)-alphai*ap(i)` statement for concurrently updating the matrix. In terms of MPI communication, these subroutines are designed to use asynchronous point-to-point `MPI_ISEND` and `MPI_IRECV` calls, so there is not a lot of scope for MPI optimizations.

To improve the memory bottleneck, changing the MPI process placement such that the L3 cache is utilized by a single MPI process can reduce contention. So, we use hybrid runs (MPI+OpenMP) by binding where MPI processes are mapped by L3 cache (i.e. 1 MPI process per L3 cache and 4 OpenMP threads per MPI process). This is particularly effective on Bridges-2, as the AMD EPYC CPU has 1 shared L3 cache between every 4 CPU cores(Figure 4.1b), so each socket has 16 L3 caches. A similar strategy is not as effective on Fritz as the Intel IceLake CPU has 1 shared L3 cache per socket(Figure 4.3b), so all 36 cores on a socket share the same L3 cache.

Finally, to reduce the impact of network congestion, we instruct the SLURM

HPC scheduler to allocate nodes close to each other using the `--contiguous` directive.

Cores	MPI ranks	Threads/rank	Optimized Time (s)	Baseline Time (s)	Speedup
128	32	4	5498.11	6983.87	1.27
256	64	4	3052.16	3754.16	1.23
512	128	4	1707.27	2136.42	1.25
				Average Speedup	**1.25**

(a) Bridges-2

Cores	MPI ranks	Threads/rank	Optimized Time (s)	Baseline Time (s)	Speedup
72	72	1	5024.42	5175.15	1.03
144	144	1	2674.75	2781.74	1.04
288	288	1	1461.12	1583.12	1.08
				Average Speedup	**1.05**

(b) Fritz

Table 4.9: POT3D Optimization results

Optimized POT3D results on Bridges-2 and Fritz are in Table 4.9. Mapping each MPI process by L3 cache and using hybrid runs for computation is an effective strategy for POT3D on Bridges-2, improving the performance of FluTAS by **25%**. A similar strategy does not show an equivalent increase in performance on Fritz due to Intel IceLake's cache hierarchy and core grouping. Vectorization and close-node placement yield a **5%** improvement on Fritz.

4.5.4 CPU vs GPU comparison

The POT3D implementation also supports GPU-enabled runs and uses the NVIDIA cuSparse library[17] for accelerating sparse matrix-vector multiplications on GPUs. Table 4.10 compares the performance of CPU-only runs vs. GPU-accelerated

QE runs on Bridges-2.

Configuration	CPU Cores	GPUs	MPI ranks	Threads/rank	Time (s)	Speedup
CPU	512	0	128	4	1707.27	1.00
GPU	40	4	4	10	550.64	3.10
GPU (cuSparse)	40	4	4	10	288.23	5.92

Table 4.10: POT3D CPU-only vs GPU-accelerated comparison

GPU-accelerated POT3D using 4 NVIDIA V100 GPUs performs **3.10x** faster than 4 CPU-only Bridges-2 nodes. Using cuSparse further improves performance to **5.92x**. Profiling data from NVIDIA Nsight systems shows that without cuSparse, a lot of time is spent in loading sparse matrix data instead of arithmetic operations, which lowers the arithmetic intensity and reduces performance. This aligns with our earlier observation that POT3D is memory-bound. With cuSparse, POT3D relies on the `cusparseSpSV_solv` subroutine to solve a system of linear equations whose coefficients are represented in a sparse triangular matrix. cuSparse enables asynchronous execution using streams even when they are generated by a previous kernel resulting in maximum parallelism.

4.6 Conclusion

In Sections 4.2-4.5, we showed the impact of compiler optimizations and profile-guided optimizations on the performance of multi-node runs.

Communication/MPI is a common bottleneck to the performance and scalability of multi-node HPC applications. One of the reasons for this bottleneck is the

prevalence of Fast Fourier Transforms (FFT) and global communication in molecular dynamics and fluid dynamics applications. To overcome this bottleneck, we utilize optimization strategies like hybrid runs (MPI+OpenMP), custom MPI process placement, non-blocking MPI communication, specialized MPI implementations that enable offloading communication to network interconnect adapters, and choosing contiguous nodes to reduce interconnect congestion. Combined with compiler optimizations, this results in an average **2x** runtime improvement for XComapct3D, **1.5x** runtime improvement for Quantum Espresso, and **80%** runtime improvement for FluTAS. We also observe that GPUs can further improve QE runtime by **3x**.

Memory bandwidth can be a bottleneck for HPC applications that rely on Preconditioned Conjugate Gradient solvers and sparse matrix-matrix or matrix-vector operations. This can be resolved by using hybrid runs (MPI+OpenMP) and custom MPI process placement to reduce memory traffic and cache contention. We observe an average **15%** runtime improvement for POT3D. We also observe that GPUs and customized libraries like cuSparse can further improve POT3D runtime by **3x**.

One thing to note is that the input dataset size can affect application scalability at higher core counts or when using GPUs, as we observe with the CP-W256 dataset in QE runs.

Chapter 5

Related Work

This section covers related topics like the Roofline Performance Model and how HPC system optimizations are applicable to Distributed Deep Learning. Roofline Model is an alternative approach that presents visual insights into improving parallel software and hardware for floating-point computations. The section on Distributed Deep Learning shows the similarities between HPC systems and large-scale ML systems.

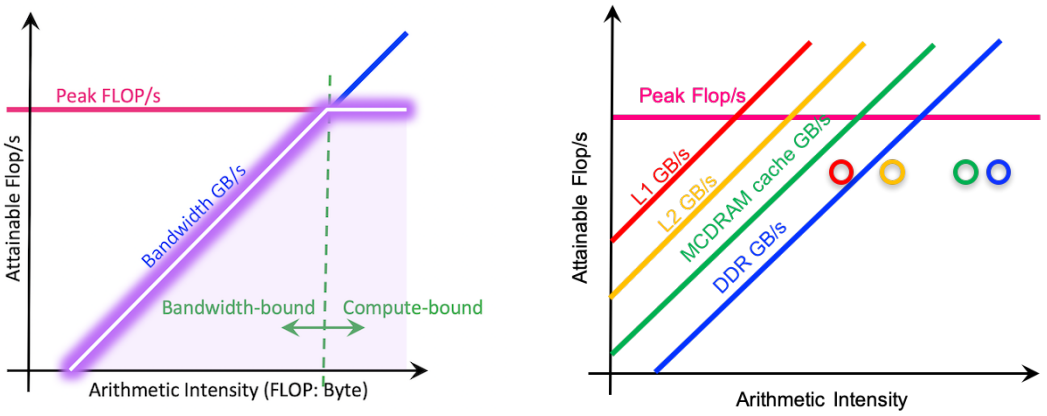
In the end, we also talk about Student Cluster Competitions (SCCs) and their potential for teaching students about application profiling and performance optimization.

5.1 Roofline Model

The Roofline performance model[68] provides a user-friendly and informative approach for evaluating application performance in relation to the capabilities of a machine. This model enables tracking progress toward optimal performance, identifying

bottlenecks, inefficiencies, and limitations in software and architecture designs. By extracting computational characteristics and simplifying the complexities of memory hierarchies, Roofline-based analysis can help optimize application performance.

This performance model allows us to set bounds on the floating-point performance (FLOP/s) using the machine's peak performance (FLOP/s), peak bandwidth (GB/s), and the arithmetic intensity (FLOP/Byte) of the application. The resulting curve, represented by a hollow purple line in Figure 5.1, forms a performance envelope that shows where the application performance falls.



(a) Base Roofline Model[48] (b) Hierarchical Roofline Model[48]

Figure 5.1: Roofline Performance Model

On this Roofline plot in Figure 5.1a, there is a ridge point called the 'machine balance' point. If an application's arithmetic intensity is lower than this point, it is considered to be **memory-bandwidth bound**. This means that the application's performance is limited by how quickly data can move through the memory system rather

than how fast calculations can be performed on the CPU core or GPU. To optimize performance in this case, it is important to examine memory inefficiencies, such as the memory access pattern, data locality, and cache reuse. On the other hand, if the application's arithmetic intensity is higher than the machine balance point, it is more likely to be **computation-bound**. In this scenario, improving vectorization (making better use of the vector units on each CPU core) or implementing multi-threading (utilizing multiple or many cores effectively) can usually lead to performance improvements.

To gain insight into the complex memory system of modern architectures, multiple Rooflines can be overlaid to represent different cache levels in the memory hierarchy. This approach is known as the hierarchical Roofline model (shown in Figure 5.1b). It helps analyze the application's data locality, cache reuse patterns, and how efficiently data is flowing through the memory system.

To my knowledge, the Roofline Performance model does not incorporate multi-node distributed memory runs, so it does not provide insights into network communication bottlenecks. As a result, it is not used in this study.

5.2 Distributed Deep Learning

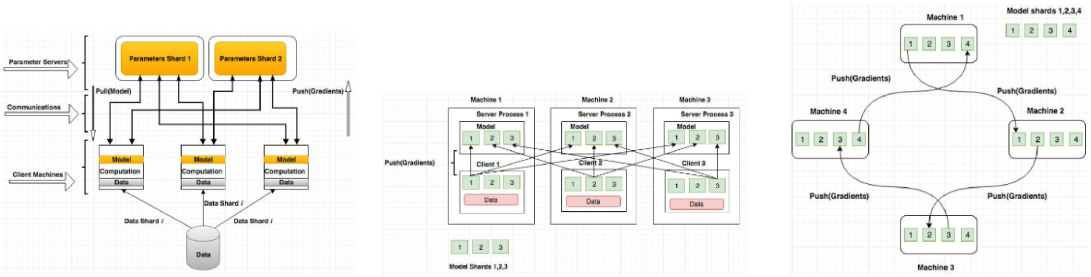
Distributed Deep Learning workloads can benefit greatly from High-Performance Computing (HPC) systems and programming models, as neural network training demands substantial computational power, and relying solely on vertical scaling for high-end single-node machines is insufficient to meet this demand. Among the scale-up

solutions, adding programmable GPGPUs is the most common method. An alternative to using generic GPUs for acceleration is the use of Application Specific Integrated Circuits (ASICs) like Google TPUs[40], Amazon Tranium[37], Amazon Inferentia[36], Tesla Dojo D1[12], Meta Training and Inference Accelerator[45], etc., which implement specialized functions through a highly optimized design targeting specific domains.

On the other hand, distributed training architectures and frameworks that leverage horizontal scaling with supercomputing clusters have gained popularity in recent years. However, these distributed training frameworks face numerous challenges in efficiently coordinating nodes within the cluster for tasks such as sharing states, parameters, and gradients. Consistency, fault tolerance, communication overhead, and resource management pose significant obstacles to achieving optimal performance.

When it comes to distributed training of deep neural networks, two primary paradigms are commonly used: Data parallelism and Model parallelism. Data parallelism involves dividing the large dataset into batches and distributing them among processing elements that train the same model simultaneously. This approach is similar to the Domain Decomposition strategy used in HPC systems. On the other hand, Model parallelism entails distributing the model itself among different processing elements. This is similar to the Functional Decomposition strategy employed in HPC systems. In practice, Data and Model parallelism are not mutually exclusive but rather complementary. They can be combined and used together to optimize the training process, much like how Domain and Functional Decomposition strategies are often used in tandem.

Three data-parallel distributed training architectures are popular in this field[1]. The first is the **Parameter Server** (PS) architecture[61], which involves multiple parameter servers responsible for coordinating and synchronizing model updates across several worker processes. The workers retrieve the model from the parameter servers, perform computations on the neural network, and then send the computed gradients back to the parameter servers. The second is the **Peer-to-Peer** (P2P) model[44], where both worker and server processes co-exist on the same machine. The worker process retrieves the model locally from the server process within the same machine, performs computations, and sends the computed gradients to all other machines in the system. The third is the **Ring AllReduce** (RA) model[52], which involves having only a server process on each machine. The server reads the model from its buffer, performs computations, and then sends the computed gradients to its neighboring machine in a ring-like structure.



(a) Parameter-Server (b) Peer-to-Peer (c) Ring-AllReduce

Figure 5.2: Distributed Deep Learning Training techniques

The Ring-AllReduce (RA) architecture was popularized by Uber Inc. when they incorporated the Baidu RA algorithm and `MPI_AllReduce` in **Horovod**[59] which

is a distributed training framework for TensorFlow. Since then, Ring-AllReduce is heavily used in modern deep learning frameworks when training on modern GPUs with fast interconnect. Hardware vendors like NVIDIA[18], AMD[34], Microsoft[46], and Amazon[35] have designed highly-tuned inter-GPU communication libraries with optimized implementations for intra-node GPU communication (over the GPU interconnect) and inter-node GPU communication (over the network).

We can observe similarities in the programming models and inter-node communication between Distributed Deep Learning systems and High-Performance Computing systems. In fact, modern distributed ML systems are designed using HPC systems, GPUs, and interconnects. The methodology presented in this study can also be applied to increase GPU utilization and improve the computation/communication ratio for large-scale Deep Learning workloads.

5.3 Student Cluster Competitions

Student Cluster Competitions (SCCs) provide students with a hands-on learning experience about HPC systems. These competitions are integrated within the largest supercomputing conferences like the International Conference for High-Performance Computing, Networking, Storage and Analysis (SC)[58] and the International Supercomputing Conference (ISC) HPC[39]. The Winter Classic Invitational is a stand-alone competition that aims to attract new talent from diverse communities to HPC from Historically Black Colleges and Universities (HBCUs) and Hispanic Serving Institutions

(HSIs)[56].

SCCs simulate a modern HPC 'micro-verse', allowing students to gain practical knowledge in operating a small cluster and optimizing scientific computing applications. In on-site competitions, a team of six students work together to design and construct a small cluster (within a certain power limit) with assistance from mentors and hardware/software vendors. During a 48-hour hackathon-like challenge, teams compete against each other to complete a real-world scientific workload while keeping their cluster operational. Virtual SCCs offer a similar experience, but teams compete remotely using provided supercomputing clusters and are mentored by HPC industry experts throughout the months leading up to the conference, allowing for a more extended learning experience. SCCs serve as platforms for students to demonstrate their systems expertise in a friendly and enthusiastic competition, which nurtures technical skills, professional connections, a competitive mindset, and lasting camaraderie.

UC Santa Cruz students have participated in the 2021 Winter Classic Invitational, 2022 ISC SCC (Virtual), 2023 Winter Classic Invitational, and 2023 ISC SCC (Virtual). The methodology presented in this study enabled the UC Santa Cruz Not-So-Slow Slugs team to achieve **2nd place** in the 2023 Winter Classic Invitational[57] and the 2023 ISC SCC (Virtual)[39].

Chapter 6

Conclusion

This work presents a methodology for observing, analyzing, and optimizing the performance of four scientific computing applications: XCompact3D, Quantum ESPRESSO (QE), FluTAS, and POT3D.

This methodology can be used to conduct an in-depth analysis of the runtime characteristics, scalability, and performance bottlenecks in shared (single-node) and distributed (multi-node) memory applications that are heavily used in High-Performance Computing and Distributed Machine Learning. All applications and profiling tools we use in this study are either open-source or free to download from third-party/hardware vendors, so this methodology can be easily extended to study other applications.

This study can also serve as a good starting point for future Student Cluster Competitions teams and enable them to streamline their optimization efforts.

Bibliography

[1] Salem Alqahtani and Murat Demirbas. Performance analysis and comparison of distributed machine learning systems. *CoRR*, abs/1909.02061, 2019.

[2] T. Auckenthaler, V. Blum, H.-J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer, and P.R. Willems. Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations. *Parallel Computing*, 37(12):783–794, 2011. 6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10).

[3] Paul Bartholomew, Georgios Deskos, Ricardo A.S. Frantz, Felipe N. Schuch, Eric Lamballais, and Sylvain Laizet. Xcompact3d: An open-source framework for solving turbulence problems on a cartesian mesh. *SoftwareX*, 12:100550, 2020.

[4] J. Bartl and L. Sætran. Blind test comparison of the performance and wake flow between two in-line wind turbines exposed to different turbulent inflow conditions. *Wind Energy Science*, 2(1):55–76, 2017.

[5] OpenMP Architecture Review Boards. What is openmp. https://www.openmp.org/resources/tutorials-articles/.

[6] Swen Boehm and Veronica Melesse Vergara. Spechpc 2021 benchmark suites for modern hpc systems. 7 2022.

[7] Shawn T. Brown, Paola Buitrago, Edward Hanna, Sergiu Sanielevici, Robin Scibek, and Nicholas A. Nystrom. Bridges-2: A platform for rapidly-evolving and data intensive research. In *Practice and Experience in Advanced Research Computing*, PEARC '21. Association for Computing Machinery, 2021.

[8] P. A. Buitrago and N. A. Nystrom. Neocortex and bridges-2: A high performance ai+hpc ecosystem for science, discovery, and societal good. *Communications in computer and information science*, 1327, 2021.

[9] Ronald M. Caplan, Cooper Downs, Jon A. Linker, and Zoran Mikic. Variations in finite-difference potential fields. *The Astrophysical Journal*, 915(1):44, jul 2021.

[10] Ronald M. Caplan, Zoran Mikic, and Jon A. Linker. From mpi to mpi+openacc: Conversion of a legacy fortran pcg solver for the spherical laplace equation, 2017.

[11] Erin Carson and Zdeněk Strakoš. On the cost of iterative computations. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 378(2166):20190050, 2020.

[12] Chips and Cheese. Tesla's dojo microarchitecture. https://chipsandcheese.com/2022/09/01/hot-chips-34-teslas-dojo-microarchitecture/.

[13] Intel Corporation. Get started with intel application performance snap-

shot. https://www.intel.com/content/www/us/en/docs/vtune-profiler/get-started-application-snapshot/2021-3/overview.html.

[14] Intel Corporation. Intel 64 and ia-32 architectures optimization reference manual. https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual.html.

[15] Intel Corporation. Intel oneapi. https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html#gs.21jbzv.

[16] Intel Corporation. Intel vtune profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.000y0c.

[17] NVIDIA Corporation. Basic linear algebra for sparse matrices on nvidia gpus. https://docs.nvidia.com/cuda/cusparse/index.html.

[18] NVIDIA Corporation. Nvidia collective communication library (nccl). https://developer.nvidia.com/nccl.

[19] NVIDIA Corporation. Nvidia cuda. https://developer.nvidia.com/cuda-toolkit.

[20] NVIDIA Corporation. Nvidia nsight systems. https://developer.nvidia.com/nsight-systems.

[21] Pedro Costa. A FFT-based finite-difference solver for massively-parallel direct numerical simulations of turbulent flows. *Computers & Mathematics with Applications*, 76(8):1853–1862, oct 2018.

[22] HPC-AI Advisory Council. Student cluster competition results- isc23. https://www.hpcadvisorycouncil.com/events/student-cluster-competition/index.php.

[23] Marco Crialesi-Esposito, Nicolò Scapin, Andreas D. Demou, Marco Edoardo Rosti, Pedro Costa, Filippo Spiga, and Luca Brandt. Flutas: A gpu-accelerated finite difference code for multiphase flows. *Computer Physics Communications*, 284:108602, 2023.

[24] Georgios Deskos, Sylvain Laizet, and Rafael Palacios. Winc3d: A novel framework for turbulence-resolving simulations of wind farm wake interactions. *Wind Energy*, 23(3):779–794, 2020.

[25] MPI Forum. What is mpi. https://www.mpi-forum.org/.

[26] freeCodeCamp. Embarrassingly parallel algorithms explained. https://www.freecodecamp.org/news/embarrassingly-parallel-algorithms-explained-with-examples/.

[27] Paolo Giannozzi, Stefano Baroni, Nicola Bonini, Matteo Calandra, Roberto Car, Carlo Cavazzoni, Davide Ceresoli, Guido L Chiarotti, Matteo Cococcioni, Ismaila Dabo, Andrea Dal Corso, Stefano de Gironcoli, Stefano Fabris, Guido Fratesi, Ralph Gebauer, Uwe Gerstmann, Christos Gougoussis, Anton Kokalj, Michele Lazzeri, Layla Martin-Samos, Nicola Marzari, Francesco Mauri, Riccardo Mazzarello, Stefano Paolini, Alfredo Pasquarello, Lorenzo Paulatto, Carlo Sbraccia, Sandro Scandolo, Gabriele Sclauzero, Ari P Seitsonen, Alexander Smogunov, Paolo

Umari, and Renata M Wentzcovitch. QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials. *Journal of Physics: Condensed Matter*, 21(39):395502, sep 2009.

[28] NASA High-End Computing Capability (HECC). Amd rome processors. https://www.nas.nasa.gov/hecc/support/kb/amd-rome-processors_658.html.

[29] NASA High-End Computing Capability (HECC). Intel skylake processors. https://www.nas.nasa.gov/hecc/support/kb/skylake-processors_550.html.

[30] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory F. Diamos, Heewoo Jun, Hassan Kianinejad, Md. Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically. *CoRR*, abs/1712.00409, 2017.

[31] Satoshi Ii, Kazuyasu Sugiyama, Shintaro Takeuchi, Shu Takagi, Yoichiro Matsumoto, and Feng Xiao. An interface capturing method with a continuous function: The thinc method with multi-dimensional reconstruction. *Journal of Computational Physics*, 231(5):2328–2358, 2012.

[32] Advanced Micro Devices Inc. Amd μprof. https://www.amd.com/en/developer/uprof.html.

[33] Advanced Micro Devices Inc. Amd rocm. https://www.amd.com/en/graphics/servers-solutions-rocm.

[34] Advanced Micro Devices Inc. Amd rocm collective communication library (rccl). https://github.com/ROCmSoftwarePlatform/rccl.

[35] Amazon Web Services Inc. Amazon elastic fabric adapter (efa). https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/efa.html.

[36] Amazon Web Services Inc. Aws inferentia. https://aws.amazon.com/machine-learning/inferentia/.

[37] Amazon Web Services Inc. Aws tranium. https://aws.amazon.com/machine-learning/trainium/.

[38] InsideHPC. What is high-performance computing. https://insidehpc.com/hpc-basic-training/what-is-hpc/.

[39] ISC-HPC23. Student cluster competition - isc23. https://www.isc-hpc.com/student-cluster-competition.html.

[40] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A domain-specific supercomputer for training deep neural networks. *Commun. ACM*, 63(7):67–78, jun 2020.

[41] Lawrence Livermore National Laboratory. Parallel computer memory architectures. https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial#MemoryArch.

[42] Lambda Labs. Openai's gpt-3 language model: A technical overview. https://lambdalabs.com/blog/demystifying-gpt-3.

[43] Sanjiva K Lele. Compact finite difference schemes with spectral-like resolution. *Journal of computational physics*, 103(1):16–42, 1992.

[44] Hao Li, Asim Kadav, Erik Kruus, and Cristian Ungureanu. Malt: Distributed data-parallelism for existing ml applications. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[45] Meta. Mtia v1: Meta's first-generation ai inference accelerator. https://ai.facebook.com/blog/meta-training-inference-accelerator-AI-MTIA/.

[46] NVIDIA Corporation Microsoft Corporation. Microsoft collective communication library (msccl). https://github.com/microsoft/msccl.

[47] NERSC. Integrated performance monitoring (ipm) for hpc. https://github.com/nerscadmin/IPM.

[48] NERSC. Roofline performance model. https://docs.nersc.gov/tools/performance/roofline/.

[49] Nick Netterville, Ke Fan, Sidharth Kumar, and Thomas Gilray. A visual guide to mpi all-to-all. In *2022 IEEE 29th International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW)*, pages 20–27, 2022.

[50] University of Oregon. Tau - tuning and analysis utilities. https://www.cs.uoregon.edu/research/tau/home.php.

[51] OpenAI. Ai and compute. https://openai.com/research/ai-and-compute.

[52] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distrib. Comput.*, 69(2):117–124, feb 2009.

[53] National Supercomputer Centre at Linköping University Peter Larsson. Quantum espresso 1500 atom system. https://www.nsc.liu.se/~pla/blog/2013/12/18/qevasp-part3/.

[54] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E Fagg, Edgar Gabriel, and Jack J Dongarra. Performance analysis of mpi collective operations. *Cluster Computing*, 10:127–143, 2007.

[55] Bharath Ramesh, Kaushik Kandadi Suresh, Nick Sarkauskas, Mohammadreza Bayatpour, Jahanzeb Maqbool Hashmi, Hari Subramoni, and Dhabaleswar K. Panda. Scalable mpi collectives using sharp: Large scale performance evaluation on the tacc frontera system. In *2020 Workshop on Exascale MPI (ExaMPI)*, pages 11–20, 2020.

[56] Intersect360 Research. 2023 winter classic invitational student cluster competition. https://www.winterclassicinvitational.com/2023-winter-classic-the-finale/.

[57] Intersect360 Research. 2023 winter classic: The finale. https://www.winterclassicinvitational.com/.

[58] SC23. Student cluster competition - sc23. https://sc23.supercomputing.org/students/student-cluster-competition/.

[59] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[60] Advanced Simulation and Computing (ASC). Asc co-design strategy. https://

www.hpcwire.com/2010/11/02/compilers_and_more_hardware_software_codesign/, https://asc.llnl.gov/exascale/co-design, and https://asc.llnl.gov/sites/asc/files/2020-06/ASC_Co-design.pdf.

[61] Alexander Smola and Shravan Narayanamurthy. An architecture for parallel topic models. *Proc. VLDB Endow.*, 3(1–2):703–710, sep 2010.

[62] Standard Performance Evaluation Corporation (SPEC). 628.pot3d spechpc 2021 benchmark description. https://www.spec.org/hpc2021/Docs/benchmarks/628.pot3d_s.html.

[63] Stackhouse Publishing Inc. The Next Platform. Deep dive into intel's "ice lake" xeon sp architecture. https://www.nextplatform.com/2021/04/19/deep-dive-into-intels-ice-lake-xeon-sp-architecture/.

[64] Top500.org. Fritz - megware d50tnp, xeon platinum 8360y 36c 2.4ghz, infiniband hdr100; rank 178/500, june 2023 top500 list. https://www.top500.org/system/180074/.

[65] Top500.org. Niagara - thinksystem sd530, xeon gold 6248 20c 2.5ghz, infiniband hdr100; rank 177/500, june 2023 top500 list. https://www.top500.org/system/179408/.

[66] Rice University. Hpctoolkit. http://hpctoolkit.org/.

[67] ROBERTO VERZICCO and ROBERTO CAMUSSI. Numerical experiments on

strongly turbulent thermal convection in a slender cylindrical cell. *Journal of Fluid Mechanics*, 477:19–49, 2003.

[68] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[69] Wired. Openai's ceo says the age of giant ai models is already over. https://www.wired.com/story/openai-ceo-sam-altman-the-age-of-giant-ai-models-is-already-over/.