Master's Thesis

Performance Engineering on HPC clusters

2110404 Chiye Wang

Supervisor Kiyofumi Tanaka

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

August 2023

Abstract

Keywords: HPC, Performance Engineering, eBPF, Observability

The complexity of scientific computing tasks necessitates the design of intricate software and hardware stacks in HPC systems to meet the scalability requirements of computational kernels. However, the increased complexity of HPC systems poses challenges for designing efficient performance analysis methods. Currently, the HPC community employs diverse and complex performance analysis tools to model the performance of HPC systems implemented on different architectures. However, this fragmented analysis approach requires performance experiments that are relatively constrained in design for target applications and systems, leading to experimental methods and results lacking portability and generality.

Our contributions mainly focus on two aspects. Firstly, we propose a multi-level observability approach for HPC systems using eBPF (Extended Berkeley Packet Filter). We describe several widely-used industrial-grade HPC performance tool implementations and performance methodologies, and identify limitations in existing performance approaches. By leveraging native features of the Linux kernel, we enhance the portability and cross-platform generality of code instrumentation and dynamic tracing capabilities. Validation shows that eBPF incurs lower runtime overhead and offers high controllability compared to traditional methods. Moreover, eBPF is language-agnostic, providing robust support for programs implemented in languages which have simpler runtimes.

Secondly, we select specific performance modeling methods from conventional performance engineering approaches, including Profiling for gaining insights into the target application's execution flow and hot functions, Kernel Extraction for understanding the nature of computational kernels and calculating theoretical single-core performance, Dependency Chain Identification for assessing multi-threading performance degradation, Kernel Benchmark for evaluating the target workload's actual performance on the system, and I/O and Memory Benchmark for comprehending the target system's I/O performance. Based on these data, we can use theoretical models of parallel computing to fit data that closely approximates real performance. Furthermore, leveraging the eBPF performance sampling data from the first part, we can implement unsupervised system performance non-regression testing. We hope that this thesis can provide a potential, portable, cross-platform, and WORA performance methodology for future HPC systems.

Acknowledgment

I am deeply grateful to Prof. Tanaka Kiyofumi and Prof. Mineo Kaneko for their generous support and guidance, without which this work would not have been possible. Their invaluable input and encouragement have been instrumental in shaping the outcome of this research. I also extend my gratitude to my colleagues at Tanaka laboratory for their camaraderie and willingness to share helpful insights, which have contributed to the overall success of this project.

I would like to express my sincere appreciation to Japan Advanced Institute of Science and Technology (JAIST) for providing me with the studentship and unrestricted access to their magnificent HPC clusters, including HPE Superdome and KAGAYAKI. The computational resources made available to me have significantly enhanced the efficiency and scope of my research.

Furthermore, I am deeply indebted to my family members and friends, especially ZIAN YE formerly from Xiamen University of Technology, for their unwavering support, courage, and endurance during challenging times. Their constant presence has been a source of strength and inspiration, replacing moments of sorrow with blessings.

Additionally, I wish to express my gratitude to a friend whom I have never met in real life. His life philosophy, ”In the pursuit of virtue and curiosity, find joy; In the pursuit of significance, find purpose,” has left a profound impact on me. I hope that heaven will recognize his blessings in the afterlife.

Last but not least, I would like to extend my thanks to myself for continuously striving to overcome obstacles and challenge fate. Without my own courage, initiative, unwavering confidence, and determination, none of the accomplishments achieved in this journey would have been possible.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Motivation

The rapid evolution of High-Performance Computing (HPC) systems and their applications has introduced complexities in programming, increased abstraction levels, and higher degrees of parallelism. To meet the demand for computational power, scientific computing applications now leverage various parallel runtimes, thread models, and I/O frameworks to harness the parallelism of HPC systems. However, the increased levels of parallelism present challenges in designing efficient workload kernels and ensuring the scalability of HPC systems. Moreover, the growing complexity of workload kernels makes understanding the performance of exascale-scale applications difficult. To effectively utilize HPC systems, powerful, accurate, and robust performance analysis tools are needed in the software stack [2]. These tools can analyze code adaptability and scalability on the target system and guide the expansion of its parallel potential. Additionally, efficient performance analysis tools and application-aware runtimes allow domain experts to focus on fundamental science rather than wasting experimental budgets on debugging target applications and adjusting parallel parameters for specific HPC systems.

In the post-Moore's Law era, the growth of single-core CPU performance and the reduction of instruction-level parallelism are no longer sufficient to meet the computational needs of scientific problems. The end of Dennard scaling has led to increasing power consumption with each generation of CPUs. Consequently, HPC systems have shifted towards SMP architectures, utilizing numerous CPUs in parallel to achieve further performance gains while improving energy efficiency. Figure 1.1 demonstrated this trend by showing the correlation between the number of transistors on CPU and HPC performance through years. However, this transformation imposes higher requirements on tools for enhancing software scalability.

Scientific computing applications exhibit a duality in HPC system design, with computational kernels demonstrating homogeneity while I/O displays heterogeneity. On one hand, most nodes adopt architectures with pipelining, superscalar processing, and out-of-order processors combined with multi-level caching. Although occasionally different instruction set architectures are used due to distinct design objectives, microprocessors largely stem from the same design paradigm [3]. On the other hand, the complexity of I/O workloads requires cache hierarchy, storage infrastructure, high-speed interconnect networks, and parallel I/O subsystems designed to provide high configurability and scalability to support diverse I/O loads. Designing and configuring matching I/O workloads and I/O systems are crucial for the theoretical bandwidth and throughput of the entire HPC system [4]. Performance optimization tools supporting complex I/O stacks, distributed storage infrastructure, and parallel file systems are highly beneficial in optimizing HPC performance and are vital in understanding the behavior of scientific computing loads' I/O.

Currently, numerous mature industrial performance tools coexist with various experimental tools. While they share similarities and overlapping features, including parallelization paradigms (Message Passing Interface, Multi-threading, Parallel Runtimes), performance data collection (Code Instrumentation/Interpolation, Sampling, Performance Monitor Units(PMU)/Performance Counters support), and performance data visualization (events, metrics, statistics), they typically adopt different approaches and offer complementary specialized functionalities [5]. Consequently, performance engineers often need to combine multiple tools to understand application performance from different perspectives.

Performance analysis tools can be classified into online monitoring tools and post-execution analysis tools based on their relationship with the application's execution time and performance data sampling time. Online monitoring tools provide real-time performance data and probe timestamps during application runtime, offering immediate performance feedback. They can operate in either a non-intrusive (observability) mode or an involving user intervention (experimental) mode. However, such implementations usually require complex runtimes, hardware support, and system privileges, making them more challenging to use in HPC systems predominantly using batch processing. In contrast, post-execution analysis tools are more commonly used in HPC system performance engineering. Post-execution tools sample performance data during application runtime and perform aggregate analysis after the application execution. This provides a global view that real-time analysis methods cannot offer, presenting the overall correlation of the entire execution process and enabling the ability to observe performance data from multiple perspectives.

Performance analysis tools can also be classified based on their implementation methods, including measurement-based techniques and interrupt-timer-based event sampling techniques. Code instrumentation involves inserting sampling code at runtime into specially annotated functions or code sections, requiring specialized compilers and runtimes for analysis. It has an advantage in analyzing global communication, data synchronization operations, high-frequency repetitions, and highly variable workloads. However, it demands a profound understanding of computational kernels by performance engineers, which often requires their involvement in the development of scientific computing applications themselves, a challenge often difficult to achieve in most cases. Additionally, inserted sampling code may affect the program's control flow, leading to significant runtime overhead and reduced accuracy of performance sampling, or even affecting the stability of the target application. Nevertheless, its precise sampling capability, highly customizable ability, and breadth of observation are unparalleled by other techniques. In contrast, interrupt-timer-based event sampling techniques allow adjusting the sampling interval based on desired accuracy, balancing data accuracy and runtime overhead. Timing sampling features black-box properties, enabling performance engineers to identify potential bottlenecks and recognize performance patterns with limited knowledge of the source code.

Performance analysis paradigms in the HPC community usually combine the aforementioned two methods. This integrated approach provides a system-level global view, demonstrating primary factors influencing scalability and highlighting and quantifying design bottlenecks in the software and hardware cooperation environment, helping performance engineers better understand the computational nature and resource demands of the workload. By using filtered event probes from PMUs, User Statically-Defined Tracing(USDT) probes, and raw event counters, supercomputer designers can address bottleneck issues as needed, and compiler maintainers can determine suitable scenarios for Profile-Guided Optimization(PGO) and reduce overhead.

Considering the diversity of performance issues, data sources, sampling methods, evaluation methods, and the complexity of metrics and HPC systems themselves, performance engineering tools and methods for HPC system design must provide cross-platform consistency and flexibility, suitable for HPC systems and software configurations with different architectural designs and implementations. Lacking portability in performance assessment environments leads to the dispersion of performance evaluation methods and increased learning costs. This necessitates performance experiments that do not require designing performance experiments for specific platforms and includes highly configurable experimental mechanisms and environments.

Relying on a single, constrained performance method will limit the evaluation scope. Fortunately, the homogenization of computational kernels has led to homogenization of HPC system design, allowing general abstractions shared among different computing environments to be supported through a unified interface structure, providing the possibility of achieving portable, cross-platform unified performance methods. We believe this unified performance analysis ecosystem can bring mutual benefits to users and developers.

Research suggests that measuring and comparing sequences of multiple configurations are more meaningful for scalability analysis than collecting extensive run data for individual configurations [6]. However, the performance data sampling process itself is bound to influence the system or target application's execution flow and performance. Extensive measurements naturally lead to analysis tool overhead, with some studies indicating runtime overheads reaching 40% or higher [7] [8], impacting the efficiency of observations among different configurations. Therefore, designing a minimal measurement analysis approach to understand application performance helps reduce overhead and limits the interference caused by observation methods in program observation.

However, most existing performance tools mainly focus on the observability of application-level performance, lacking the ability to observe interactions between the operating system kernel and software [9]. The role of the operating system in modern large-scale parallel environments cannot be ignored. Tasks or I/O schedulers, Non-Uniform Memory Access(NUMA) support, and memory operations performed by the kernel directly impact the overall system's performance. Particularly, as modern HPC systems increasingly adopt Linux-based custom-optimized kernels instead of proprietary systems like Cray OS, which showed in Figure 1.2 and Figure 1.3, opens up the possibility of utilizing OS-based performance engineering methods. To provide portable OS-based performance engineering methods, we utilize eBPF integrated into the Linux kernel to observe kernel behavior. Essentially, it is a protected virtual machine running in kernel space, observing the kernel state by executing eBPF bytecode, thereby enabling instrumentation of critical events in the Linux kernel and providing non-intrusive observability. Based on eBPF, performance observation incurs minimal interference with the system, runtime, or the application itself, while maintaining accurate analysis capabilities and minimal overhead [10] [11].

Based on the above, we propose a top-down performance engineering method for HPC systems based on eBPF. This method leverages the native capabilities of the Linux kernel, providing unparalleled OS-level observability and portability. It is a low-intrusion, flexible, portable, cross-platform, and software-hardware cooperative approach that combines the advantages of code instrumentation and sampling. Additionally, our method supports both black-box and white-box analysis methods, offering highly customizable event observation and precise control to efficiently manage runtime overhead. Furthermore, our implemented black-box method does not rely on specific languages or compilers, making it particularly suitable for scientific computing applications written in C or Fortran. It offers high compatibility, does not require dependence on the source code, and ensures the preservation of the original control flow of the code, guaranteeing accuracy.

We demonstrate our method through the supercomputer KAGAYAKI at Japan Advanced Institute of Science and Technology(JAIST) and the classical Ab Initio computational tool Quantum Espresso [12]. We adopt a hierarchical approach that combines traditional tools with eBPF to address the challenges of performance pattern recognition. Through experiments, we reveal the feasibility of our method by demonstrating certain performance bottlenecks that can be identified and localized through traditional code instrumentation and sampling methods. We also showcase the unique strategies and analysis perspectives of our method. Finally, we explore the possibility of implementing automatic online performance monitoring using eBPF and XGBoost [13], achieving rapid and accurate modeling of application performance across different platforms.

1.2 State of the Art

The HPC community has developed numerous mature industrial-grade performance analysis tools, along with some experimental ones, which significantly contribute to the optimization of performance analysis for target parallel computing programs. This section presents several cutting-edge research achievements, analyzing their application scenarios and professional domains while explaining their limitations and constraints.

1.2.1 Performance Methods

Scalasca [14], the successor to the KOJAK [15] research project, is an open-source performance analysis toolset developed by the Jülich Supercomputing Centre. It is specifically designed for large-scale parallel computer clusters such as IBM Blue Gene and Cray XT. Scalasca uses Code Instrumentation for measurement-based performance analysis, distinguishing object types, including MPI communicator objects, for annotation and analysis in event tracing. It supports incremental performance analysis by progressively improving measurement configuration strategies, integrating runtime performance analysis summaries with parallel behavior during event tracing. Scalasca excels in recognizing waiting states, enabling the identification of performance challenges arising from workload imbalances in large-scale parallel I/O-intensive applications. By pinpointing abnormal waiting states, developers can address load imbalance issues through adjustments like process placement, cores binding, and cache affinity to optimize communication structures.

TAU [16], initiated by the University of Oregon in collaboration with the German Research Centre Juelich and the Los Alamos National Laboratory, is a flexible and portable performance tool framework closely associated with Scalasca. It adopts a three-layer modular design, providing users with highly customizable performance experiment configurations through sub-module configurations. TAU's flexible instrumentation allows users to insert code instrumentation at different levels, supporting various types of performance events, including communication library interface events and user-defined events. TAU's compatibility with Scalasca is achieved through standardized metrics defined for performance experiments. It also includes the PerfDMF [17] tool for parallel profiling management and collaboration with other performance analysis tools.

HPCToolkit [18], developed by Rice University, is a post-analysis tool based on interrupt timers and hardware counters. It eliminates the need for code instrumentation by sampling and collecting performance measurement data from fully optimized compiled applications. HPCToolkit offers binary application analysis techniques, associating measurement results with source code content and structure, presenting performance data in a top-down manner.

Vampir [19], developed by the German Research Centre Juelich and currently maintained by the ZIH at the Technical University of Dresden, is a well-known performance analysis tool in the HPC community. It consists of instrumentation and measurement components (VampirTrace), visualization component (Vampir), and parallelized extension (VampirServer). Vampir provides a powerful and versatile performance visualization interface [20], capable of utilizing performance data samples from Scalasca, TAU, and Paraver to generate intuitive chart-based analysis reports.

Paraver [5], developed by the Barcelona Supercomputing Center and the Polytechnic University of Catalonia, is a tool for visualizing and analyzing parallel time tracking files. It studies the effects of short-term scheduling strategies on multi-programming in distributed memory and MPI architecture computers. Paraver performs performance data collection by adding functions to the PVM [21] message-passing library and uses the distributed memory simulator DIMEMAS for event flow analysis.

Caliper [22], developed by Lawrence Livermore National Laboratory, is a cross-stack, generic performance analysis framework based on code instrumentation. It provides an easy-to-use timer API for application developers to merge information extracted from APIs across all software layers into a single context, allowing data analysis and performance enhancement.

Score-P [23], part of the SILC and PRIMA projects, is a collaborative effort involving several institutions. It provides performance analysis for other tools in the project, supports code instrumentation and sample-based post-analysis, and includes high versions of parallel runtimes and heterogeneous computing support. Score-P prioritizes portability, scalability, and controllable measurement overhead on HPC systems, encapsulating measurement data in the CUBE4 [24] format for visualization tools like Vampir.

PAPI [25], a milestone-level supporting technology in the performance analysis domain, offers generic interfaces for accessing hardware-provided precise counters on major processor platforms, providing necessary information for cross-platform tuning. PAPI is widely used as a hardware counter data source by performance analysis tools, providing portability across different operating systems and architectures. It offers highly customizable programmable low-level interfaces and highly automated simple measurement high-level structures, enabling users to control event detection range. Additionally, PAPI includes a predefined set of events, allowing unified tool counting of comparable events on different platforms using this standardized event set. It provides user callbacks based on counter overflow and hardware-based SVR4 compatibility analysis, offering infrastructure for any performance data source independent of the operating system. PAPI serves as a genuinely portable performance analysis infrastructure for any architecture equipped with hardware performance counters.

In the field of High-Performance Computing (HPC), in addition to the tools previously mentioned, there are numerous other powerful performance analysis support stacks and unique design approaches, such as Periscope [26], Jumpshot [27], Paradyn [28], and others. These performance engineering tools provide a theoretical foundation and inspiration for the methods discussed in this paper. However, due to space constraints, they cannot be exhaustively listed here. Their presence enriches the tool ecosystem in the HPC performance analysis domain, offering scientists and developers a broader range of options and resources to optimize and enhance the performance of parallel computing applications.

1.2.2 Limitations

[29] distinguishes three types of performance analysis methods:

1. **Measurement:** Measuring the performance of the target application on the actual system provides the most straightforward and intuitive performance data. Due to its convenience and generality, measurement is the most commonly used performance analysis method in HPC systems. Depending on the specific implementation, measurement techniques can be categorized into manual or automatic code instrumentation and interrupt sampling techniques based on hardware timers. However, since measurement tools run alongside the target application on the host machine, they inevitably introduce an impact on the execution process of the target system and application, known as the observer effect. Some hardware features, such as PMU, can reduce intrusiveness and runtime overhead during measurement. Nevertheless, when using software and system features for measurement, careful consideration is required to set the observation scope and sampling interval to avoid affecting the performance analysis results. As this method necessitates running the target application on the actual system, it may become unavailable if the required running conditions cannot be met.
2. **Simulation:** This method employs a simplified model of the target system implemented in software to simulate the behavior of the real system. In a simulation environment, the behavior of the target application can be observed at a microscopic time scale, allowing verification of various system implementations' impact on the target application before the actual system is implemented. In the simulation context, the observer effect present in the measurement method can be avoided, making it a widely used approach in traditional simple systems.
3. **Analytical:** This method applies numerical analysis to the mathematical model of the system to identify potential performance bottlenecks of the target application during the development phase. It also provides a global view of the target system and performance patterns. Analytical methods are generally faster than simulation methods, but to discover implicit performance patterns from extensive data, they require numerous assumptions and experimental verification to identify the causes of bottlenecks, thus necessitating a certain level of expertise.

The HPC performance tools mentioned earlier, such as Scalasca and TAU, provide measurement methods based on automatic code instrumentation. This approach offers high generality and does not depend on specific architectures or supercomputer designs. However, such methods rely on specific language runtimes and custom compilers, and their support for different language standards heavily relies on the specific tool's implementation. Therefore, for high-performance computing applications implemented in Fortran, it is challenging to ensure the availability of this method. When using Scalasca's MPI compiler and runtime to compile various scientific computing applications, we encountered language incompatibility issues. Additionally, research on HPCToolKit has pointed out that code instrumentation may impact the execution flow of the target application and system, affecting the accuracy of measurement results and the robustness of the method itself.

Among the most commonly used simulators in the HPC community is SimGrid. This method allows for performance simulation of the target machine, system, and application on any machine, providing relatively accurate performance evaluation results at a very fast speed. However, simulation methods require setting different parameters for different systems, and the simulator code itself requires extensive development and maintenance, adding complexity to the performance analysis process.

Commonly used performance analysis methods like Top-Down combine numerical analysis with measurement. By leveraging performance data sources such as PMUs and combining them with performance hierarchical modeling specific to the target system architecture, these methods can quickly identify potential performance bottlenecks in a particular architecture and gradually analyze their causes. However, numerical analysis methods require separate modeling for each system, and this process is challenging to apply to similar systems with only minor variations, making them almost non-portable.

1.3 Contributions of This Thesis

In this paper, we propose a novel performance engineering methodology tailored for HPC systems. The core idea of this new approach is to utilize black-box analysis to provide independent analysis capabilities from the source code while leveraging observable features provided by the operating system to offer flexible, highly customizable, and portable performance experimentation support.

The contributions of this paper are summarized as follows:

1. Comprehensive review of state-of-the-art supercomputer designs: The paper presents a thorough analysis of the performance contributions of various components in HPC systems, ranging from architectural choices to software-hardware stacks and code design of scientific computing applications.

2. Introduction of conventional software engineering approaches for software-hardware co-design and performance analysis tool designs tailored for HPC system workloads: The existing design implementations, application scenarios, operational overheads, and limitations of these approaches are discussed.

3. Introduction of a methodology employing generic tools to analyze application performance bottlenecks: Performance engineers can quickly identify performance issues using tools like Linux Perf without requiring an in-depth understanding of the target code.

4. Proposal of methods for modeling CPU-Memory system communication performance: Standard benchmarks are used to determine if system configurations lead to I/O bottlenecks in the target application.

5. Development of a cross-platform, cross-application, and cross-processor architecture generic performance analysis tool using Linux's native observability component, eBPF: The goal is to provide a unified, scalable, and Write Once, Run Everywhere (WORE) performance analysis component.

6. Implementation of the proposed methodology using eBPF to support mainstream analysis methods in the HPC community, including code instrumentation based on USTC and real-time performance sampling based on hardware counter interrupts: The approach supports both online real-time analysis and post-analysis.

7. Utilization of eBPF's powerful dynamic probe technology to ensure highly controllable runtime overhead during performance event sampling: eBPF can utilize JIT to accelerate the runtime speed of performance analysis logic, enabling broader observation capabilities with lower overhead compared to other performance sampling techniques.

8. Integration of performance data sampled by eBPF with performance visualization tools like Vampir and TAU using performance log conversion tools.

9. Demonstration of the capabilities and potential of the proposed methodology using Quantum Espresso: The paper analyzes the trade-offs between costs and performance of this approach.

Overall, our novel performance engineering methodology addresses the challenges of performance analysis in HPC systems by providing a powerful, flexible, and portable approach that leverages black-box analysis and observable features of the operating system to optimize application performance. The paper contributes to advancing performance engineering techniques in the HPC community and opens new possibilities for efficient performance analysis and optimization.

1.4 Organization and Synopsis

Following the introduction chapter, the subsequent chapters of this thesis are organized as follows:

- **Chapter 2:** Before proposing the novel performance analysis methodology, a comprehensive grasp of the elements contributing to "performance" in high-performance computing systems is of paramount importance. This chapter delves into the study of architecture evolution, design methods, system components, and specific implementations of HPC systems. Furthermore, it encompasses cutting-edge technologies employed in HPC. Chapters 1 and 2 jointly form the literature review, providing an exposition of the design and performance methodologies of state-of-the-art high-performance computing systems.

- **Chapter 3:** Understanding the adaptability of parallel computing programs on the target computing architecture is crucial as the main source of HPC system performance. This chapter provides a detailed exposition of our proposed performance analysis method, covering topics such as software-level hot path analysis, computational kernel extraction, and dependency chain identification. Our software performance modeling approach will utilize a combination of white-box and black-box performance sampling methods, aimed at providing architecture and operating system-independent generic performance analysis methods for exploring and identifying potential application performance bottlenecks. Additionally, this chapter will conservatively estimate the implementation cost, runtime overhead, and portability of the method and compare it with other conventional methods.

Furthermore, scientific computing applications exhibit a clear compute-I/O loop, where I/O performance significantly impacts their scalability. Therefore, specialized performance analysis modules are required for the I/O system and subsystems. This chapter introduces hardware-level computational kernel benchmarks, I/O benchmarks, and MPI memory bandwidth modeling to theoretically model the CPU-Memory subsystem, thereby understanding the performance gap between CPU and I/O bandwidth. Additionally, we will explore a novel I/O performance analysis method based on eBPF. This method utilizes Linux kernel probes and instrumented I/O libraries to provide performance event analysis capabilities for both the operating system-level and high-level parallel I/O library-level.

- **Chapter 4:** This chapter presents the initial validation of the proposed methodology, employing conventional tools and eBPF implementation. It engages in a comprehensive discussion of the costs and limitations associated with implementing the methodology using conventional methods and observability techniques.

- **Chapter 5:** The thesis culminates with a concise summary of the findings and outlines future research directions. Acknowledging the protracted and rigorous development process involved in performance analysis methodology and tools, along with the necessity for validation on industrial-grade HPC systems, the study will continue in collaboration with JAIST and BSC to explore the capabilities and potential of the proposed performance method.

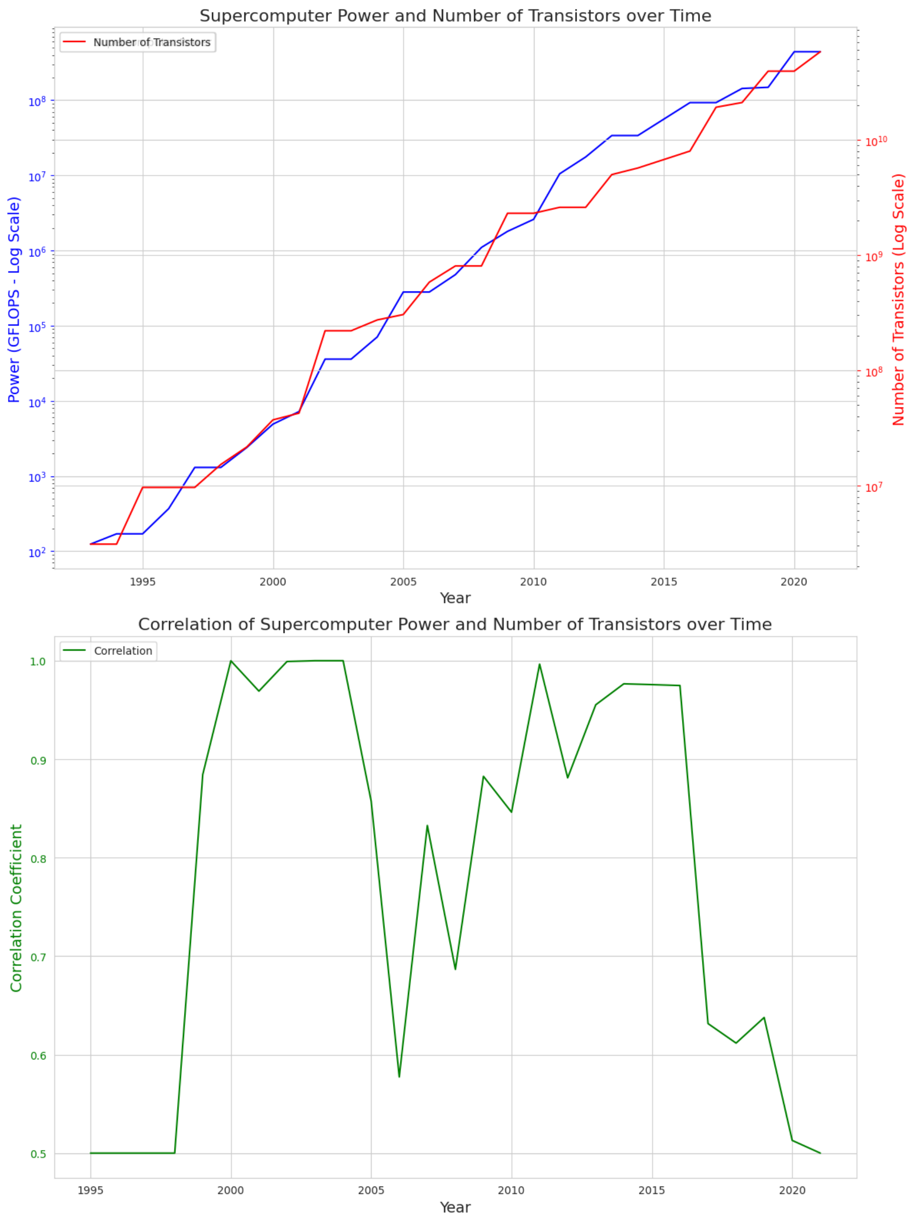


Figure 1.1: Correlations between CPU transistors and HPC performance.

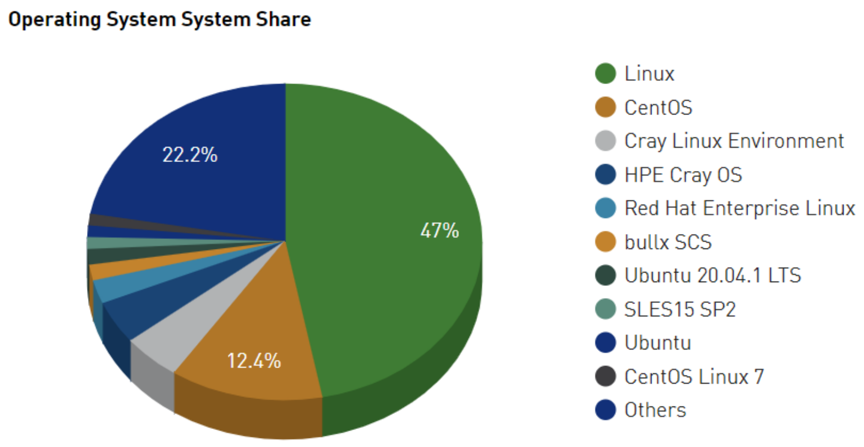


Figure 1.2: Operating systems on Top500 HPC systems, system share.

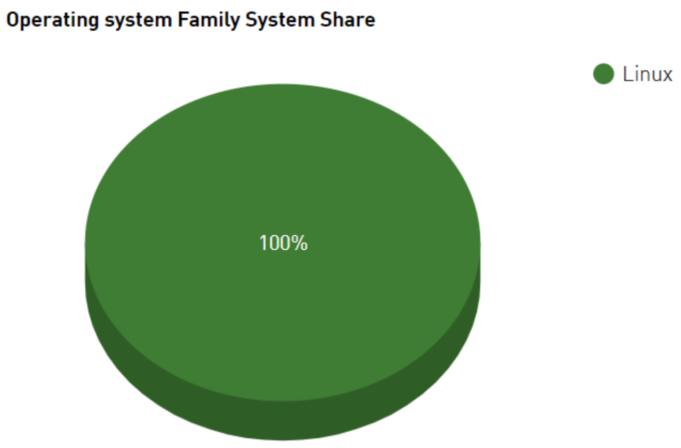


Figure 1.3: Operating system family on Top500 HPC systems, system share.

Chapter 2

Performance on HPC System

The concept of supercomputers or high-performance computing (HPC) systems has evolved significantly over time. In the past, "supercomputers" referred to systems that surpassed the performance of standard home computers. However, in today's context, the term "HPC system" describes computer clusters specifically designed to handle large-scale scientific computing problems that are beyond the capabilities of general-purpose computers. These systems distribute tasks across multiple high-performance nodes, employ domain decomposition, and utilize unique hardware and software designs to minimize the impact of distributed memory and sparse storage. As a result, HPC systems excel in certain application workloads, surpassing mere aggregation of computing power techniques such as grid or distributed computing. Nevertheless, traditional HPC applications often require a co-design approach [30], where software and hardware are carefully tailored to achieve optimal performance on the target HPC system. Moreover, strategies aimed at improving overall performance in HPC systems by upgrading CPU performance and increasing the number of computing hardware must consider factors such as energy efficiency and thermal design. This leads to challenges in scalability and extensibility compared to distributed computing methods, which can easily boost overall performance by adding more computing nodes. Notably, the Folding@home project, supported by BOINC, achieved a peak performance of 2.5 exaFLOPS in 2020, making it the first project to possess an Exascale-scale computing network.

Unlike conventional computers, HPC systems require a delicate balance between performance and power consumption, often necessitating joint hardware and software design for specific application workloads. Understanding the architectural design principles of HPC systems is crucial for comprehending the performance impact of each component. Therefore, this chapter first reviews representative HPC system architectural designs and analyzes the historical evolution of HPC systems from an engineering perspective, considering aspects such as architecture, hardware specifications, and co-design methods of software and hardware. Subsequently, the discussion delves into observability techniques and performance engineering methods developed for various components of HPC's software and hardware stack. Commonly used methodologies and tool implementations are summarized, and the usage scenarios, scalability, and portability of existing methods are explored, independently of specific computing architectures and workloads. Additionally, the potential integration of emerging technologies in the design of future supercomputers is investigated, analyzing progress in vector processors, parallel I/O, scheduling algorithms, and queuing simulators. This exploration aims to provide insights for the development of more adaptive, efficient, and robust performance analysis methods.

2.1 Evolution of Supercomputers : A Story

The evolution of HPC system architecture is closely tied to changes in CPU microarchitecture design. Initially, HPC systems used single powerful vector processors for SIMD arithmetic. Subsequently, with the acceptance of SMP, they transitioned to NUMA architectures and high-speed interconnect networks supporting large-scale parallel computing clusters. This paradigm shift transformed supercomputers from single-processor systems to computer clusters based on symmetric multiprocessing (SMP) with UMA architecture. Complex computational tasks that cannot be handled by a single computing node are now decomposed into several sub-tasks and scheduled and distributed across multiple computing nodes using batch processing systems. As scientific computing tasks become exponentially more complex and the demand for processing more data in shorter time increases, the design of high-performance computing systems also continuously evolves.

The concept of large-scale parallel computing, which is prevalent in modern HPC clusters, originated from the ILLIAC IV supercomputer cluster in 1975. This system was the first to use multiple CPUs and a large number of floating-point units (FPUs) for collaborative computation. However, limitations in symmetric multiprocessing (SMP) technology and the high cost of multi-CPU computing restricted its application.

In 1976, the CRAY-1 was introduced, offering a single-processor performance of 133 MFLOPS and becoming one of the most successful commercial supercomputers in history. Although reservations initially existed regarding SMP-based supercomputer systems, a significant shift occurred with the introduction of the CRAY-X-MP by Cray in 1982. This machine, utilizing four processors to achieve a peak performance of 800 MFLOPS, was considered the fastest supercomputer from 1983 to 1985, driving the transition from single-vector processor designs to large-scale parallel clusters based on shared-memory models.

Equally important was the LINKS-1 assembled at Osaka University in 1982. It utilized 257 Zilog Z8001 control processors and 257 iAPX 86/20 floating-point processors, ranking among the most powerful computer clusters worldwide until 1984.

In the 1990s, the maturity of high-speed interconnect technology led to the emergence of HPC clusters equipped with thousands of processors, shared-memory systems, custom operating system kernels, and fast interconnects between nodes. Until around 2005, HPC clusters commonly adopted CPU architectures based on single cores, with each CPU-memory unit (CMU) typically equipped with two processor slots. As the benefits of Dennard scaling diminished, CPU vendors shifted towards multi-core and hyper-threading technologies to continue improving the performance of individual CPUs. By 2023, all supercomputing systems listed on the HPC TOP500 adopted multi-core processors.

Although processors based on existing technologies face various physical limitations in maintaining performance parameters such as clock frequency and instructions per cycle (IPC), the progress of supercomputers has not stalled. From 2000 to 2016, there was a clear correlation between peak HPC computing capability and the number of transistors on CPUs. However, after 2016, these two factors began to decouple, and the impact of the number of CPU transistors on peak HPC performance gradually weakened. Factors such as thermal design, parallel I/O, and scientific computing software design have become critical determinants of HPC system peak performance.

In the face of these challenges, chip manufacturers have innovatively explored alternative paths to ensure continuous performance improvements for users. However, this diversity of technologies also introduced additional constraints in system design, increasing the cost of achieving efficient cluster computing. Moreover, due to the unique instruction set architecture (ISA), compilers, and runtimes of HPC systems, as well as the necessity for co-designing scientific computing code, the design of scientific computing applications has become increasingly complex. Nonetheless, we believe that evaluating HPC system performance should not be limited to summarizing the theoretical capabilities of its computing hardware alone. Instead, software performance engineering should be incorporated into the considerations of HPC system design, taking into account the computational loads of applications running on the system. This viewpoint was recognized in the design of the BlueGene/L supercomputer system by IBM and Lawrence Livermore National Laboratory (LLNL) in 2001, as well as in the development of the world's first pre-exascale supercomputer Fugaku by RIKEN in 2020. The effectiveness of this approach was further affirmed by the 2021 development of a pre-exascale supercomputing cluster by AMD and ORNL, making it the world's first exascale-level supercomputer.

2.2 Architecture

High-Performance Computing (HPC) system architecture involves the organization, functionalities, and Instruction Set Architecture (ISA) of its components, as well as their interactions with workloads. Factors such as programming methods, parallel programming paradigms, and memory access patterns determined by the architecture play a crucial role in the disparity between the potential peak performance and the actual performance of HPC systems.

Traditionally, the components of HPC systems were relatively homogeneous, but there have been changes in recent years. With an increasing demand for specific computing tasks, the use of domain-specific accelerators (such as Field-Programmable Gate Arrays - FPGAs) in general HPC systems has decreased. Instead, there is a growing preference for architectures that support Single Instruction Multiple Data (SIMD) operations, such as tensor-based processors, General-Purpose Graphics Processing Units (GPGPUs), and variable-length vector extensions. These architectures enable heterogeneous computing, utilizing offload techniques to dynamically allocate tasks with varying levels of vectorization to CPUs and heterogeneous computing hardware, allowing multiple processors with different microarchitectures and implementations to simultaneously execute different tasks.

With the rapid advancement of Large Language Models (LLMs), traditional dual-GPU AI servers or small-scale distributed computing using frameworks like Horovod [31] or DeepSpeed [32] are becoming insufficient to meet the massive computational and memory requirements of these models. Furthermore, servers running distributed training runtimes often lack high-speed interconnects, leading to inefficient inter-node I/O and burst I/O. Therefore, I/O-intensive tasks that require significant memory usage or memory swapping, such as training large neural network models like ChatGPT, may present future challenges for traditional distributed systems. Compared to distributed computing, HPC systems offer more efficient parallel I/O abstractions, a unified memory model among nodes, and a wide range of heterogeneous computing resources, making them well-suited for resource-intensive tasks. However, facing the increasing diversity and complexity of I/O loads, designing efficient I/O systems to cope with the demands of data explosion poses significant challenges for HPC system architecture design.

2.2.1 CPU Architecture

As the core of any general computing architecture, the CPU is primarily responsible for executing the assigned workloads. The most direct approach to creating high-performance supercomputer clusters is to have processors with higher frequencies. Following this trend, HPC systems use enterprise-grade CPUs specially designed for large-scale parallel performance, such as Intel Xeon, AMD EYPC, and other product lines, achieving performance, IPC, cross-node scalability, PCIe channel count, and support for specialized high-speed interconnect networks that ordinary CPUs find challenging to match.

In the 1990s, chip manufacturers adopted Dennard scaling technology, which involved reducing the size of transistors to achieve shorter switching times, resulting in up to 40% increase in single-core performance with each new generation of processors. This increase in IPC allowed software to run faster.

However, the linear relationship between CPU dynamic power and frequency led to a significant increase in CPU power consumption with increasing frequency. As transistors needed to be continuously scaled down for higher frequencies, the leakage current effect of transistors became more pronounced. This effect raised the junction temperature of transistors, potentially causing thermal issues. Eventually, as computing platforms approached the power limits, frequency scaling ended its decades-long exponential growth in 2006, remaining at the threshold of 5 GHz.

Since then, chip manufacturers have explored integrating more functional units, register sets, caches, and other resources onto processors to continue improving single-core performance. In response to the need for increased performance, chip manufacturers also turned to designing multi-core processors. Today, almost all consumer computers, smartphones, and even smart devices are equipped with processors based on multi-core architectures. In data centers, supercomputing centers, and other high-performance computing applications, a single CPU-Memory system can contain multiple CPU sockets, each with hundreds of cores.

The performance gain from using a multi-core architecture can only be seen as a palliative measure after the end of Dennard scaling. Due to the limitations of current processes, the power density of today's processors increases with each generation, making it impossible to supply all components on the processors with the rated voltage to maintain safe thermal design power. This phenomenon results in the presence of undervolted computational units in the processor, potentially preventing the entire CPU from achieving its rated performance. Given these constraints, Hennessy and Patterson estimate the performance growth rate of single-core processors under existing processes to be approximately 3% per year [33]. Compared to the 50% annual growth rate during the Dennard scaling era, the performance compensation provided by multi-core, multi-threading, and other technologies falls short. Therefore, exploring new semiconductor manufacturing processes or discovering the potential of heterogeneous computing to compensate for CPU performance shortcomings is essential for elevating HPC system performance to new levels.

2.2.2 I/O Architecture

To alleviate the complexity of the I/O system for software developers and hardware systems, while also mitigating the difficulty in software and hardware system maintenance, the I/O system is typically abstracted into several levels. Applications utilize the high-level abstraction interfaces provided by the I/O system to issue requests to large-scale storage infrastructure and file systems. These requests constitute an I/O request path through various components of the I/O system, a pathway referred to as the I/O stack.

Traditionally, applications directly use file abstractions provided by the operating system, such as generating I/O requests via the POSIX standard interface and conveying these requests to the underlying file system. However, the direct employment of POSIX in scientific computing results in inefficient access due to constraints such as consistency semantics requirements. Conventional local file systems also fall short in meeting the demand for large-scale storage devices and parallel access.

In response, scientific computing applications often utilize advanced I/O libraries such as MPI-IO [34] rather than the POSIX standard interface to initiate I/O requests to parallel file systems. These systems, used in high-performance computing (HPC) requiring large-scale parallelism and multiple tasks reading and writing from shared data simultaneously, are referred to as parallel I/O. Distinguished from the I/O stack for desktop computers, parallel I/O boasts higher storage capacity support, weak consistency semantics for parallel file access, advanced parallel I/O abstraction interfaces, and support for underlying parallel file systems.

One simple approach to implementing parallel I/O is allocating a separate file to each task, such as the checkpoint of neural network training or the progress archive used by VASP for interrupted restarts. In these instances, the data belonging to each task is easily distinguishable. As the data is not stored across nodes, it is regarded as parallel task local I/O.

A more intuitive approach is for part or all tasks within multiple tasks to hold a local offset pointer to a shared file and execute write or read operations on the same shared file. Two possible scenarios arise in this approach. In the first, each task accesses different partitions of the same file, such as HDF5-based categorized data. Here, there are no access conflicts between tasks and no need to handle consistency issues; only the file abstraction allowing partitioned access provided by the advanced I/O library is necessary. In the second scenario, different tasks may concurrently access the same position in the file. Here, the parallel I/O system needs to provide support at different software levels.

The third method, collective I/O, is a two-phase I/O method. By aggregating I/O requests from different tasks and serving these aggregated requests, this method reduces the I/O burden of large span discontinuous sections of multiple processes accessing the same file. Compared with the second method, collective I/O also redistributes the cost of inter-process communication to a lesser degree, hence significantly improving I/O performance.

Given the large volume, parallel access, and heterogeneity characterizing the I/O load of scientific computing applications [6] [4], the I/O system must enhance its bandwidth and Input/Output Operations Per Second (IOPS) to accommodate these traits of application load. Bandwidth refers to the data transfer rate achievable on a given transmission medium or path per unit of time, while IOPS is the number of I/O operations that can be performed in one second. Thus, the bandwidth of the parallel I/O system is the maximum stable read and write speed that the system can provide under full load, with the number of reads and writes being IOPs/R and IOPs/W, respectively.

However, a single computing node in current HPC architectures usually lacks high I/O capability. The I/O system often needs to be partitioned according to the application load and I/O access patterns, with the I/O efficiency of the task itself enhanced by accessing the I/O function area through a large number of nodes [35]. This partitioning can be achieved through special settings in different parts of the I/O stack.

Given the need for different computing systems to efficiently access parallel file systems without redesigning the I/O stack, parallel I/O systems generally possess robust reconfigurability and scalability. Scientific application developers merely need to use the file abstraction provided by the advanced I/O library without considering the specific implementation of the I/O system. The I/O stack can be divided into three layers: the computing system side, the I/O system side, and the I/O subsystem side. The computing system encompasses computing nodes, collective I/O dedicated I/O nodes, and cross-node interconnection network systems. The I/O system includes parallel file systems and storage infrastructure, defining the file abstraction interface used by the workloads running on the computing system. The I/O subsystem includes middleware used during the execution of

I/O requests, such as the peripheral network system of the computing system or the local non-parallel file system used for login nodes. I/O optimization for application load usually only occurs at the I/O layer.

2.2.3 Memory Architecture

HPC systems typically employ a shared memory multiprocessor architecture, where all processors can directly access the system's shared memory address space through high-speed interconnects between the processors and memory [3]. In this architecture, any processor can access or modify data created or used by other processors. Based on the locality of process and thread control instances, memory access in HPC systems can be categorized into two forms.

The first form is a shared memory system, where a single operating system runs on top of multi-core hardware, and the memory hardware is directly connected to all processing units and main memory, sharing the address space with all physical cores in the system. Each processing unit can access the entire shared memory address space. Processors employ inter-process communication (IPC) and memory operations through the shared memory model, while programmers ensure memory access compliance using programming models and interfaces provided by the operating system. However, the behavior of different processor cores accessing the same memory space in the shared memory model necessitates that the memory interconnect network provide cache coherence semantics to avoid memory access conflicts.

Cache coherence semantics ensure that modifications to data in one cache are reflected in the copy of the global data held in all potential copies of caches. It encompasses two fundamental requirements:

- **Write propagation:** Any modification to cached data must be broadcast to equivalent caches.
- **Transaction serialization:** All processors must observe reads and writes to a single memory location in the same order.

The earliest cache coherence protocol, the Modified Exclusive Shared Invalid (MESI) protocol, also known as snooping cache, was introduced in 1983. This approach utilizes cache controllers to monitor data accesses on the bus and employs three reserve bits on cache lines to record cache states. Each cache line can be in one of the following states: Modified (the local processor has updated the cached data), Exclusive, Shared, or Invalid. When a data block is first loaded into the cache, it is marked as Exclusive. In the event of a cache read miss, the read request is broadcasted on the bus and intercepted by the cache controller. If the address is cached and the cache line is in the Modified state, it is set to Exclusive and sent to the requester. In the case of a cache write miss, the cache snooping element can intercept the request and invalidate any copies held by other processor cores. If the cache is dirty and shared and there are requests for that memory copy on the bus, the dirty snooping element provides data to the requester. The cache system ignores the request when the marked address is invalid. The new cache is marked as dirty, valid, and non-shared, and it takes responsibility for that address. This method updates or invalidates the cache by detecting processor write requests to memory and checking if the same memory block is cached, thus avoiding consistency issues.

The approach to building parallel programs for large-scale parallel machines is known as the parallel programming paradigm. Based on the shared memory model, the simplest form of parallel programming paradigm is to clone multiple sub-threads within the same resource pool or address space using a single thread, with communication between threads achieved through shared variables. In this model, Inter-Process Communication (IPC) is implicit, and the memory model provides efficient utilization of data locality and instruction locality, meaning that program developers do not need to concern themselves with the layout of data in memory space. While this programming model simplifies the development of scientific applications, hardware can only perceive memory-to-processor communication at runtime as there are no annotations in the code indicating memory access patterns. This poses significant challenges to the parallel runtime [36].

Based on the relative time at which multiple processors in the shared memory model access shared memory blocks, HPC memory architectures can be categorized into two types: Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA).

In a UMA architecture based on Symmetric Multiprocessor (SMP), each memory block can be accessed at the same time. Under the SMP architecture, all processor cores are typically controlled by a single operating system kernel, and the bus multiplexing of memory units is provided by a bus controller or crossbar switch and other interconnect networks. Theoretically, the time for memory access requests is independent of the processor issuing the request or the memory chip containing the shared memory block. However, contention between multiple processors for a single memory block leads to delays in subsequent access requests to the same memory block. This means that for a large amount of memory, the bandwidth bottleneck of a single memory storage unit will affect the overall memory performance of the system, thereby impacting system scalability. Nonetheless, all processors still have equal access opportunities. SMP became the mainstream architecture for HPC system design in the 1980s, replacing single-processor architectures that used a single powerful vector processor. Pioneering HPC systems using

SMP included Sequent Balance 8000 and CRAY X-MP. Nowadays, SMP is used in most data center servers, HPC clusters, and home computers utilizing multi-core architecture chips. In particular, in HPC systems, SMP is often used as the memory architecture for individual CPU-Memory units to simplify the programming model on each node.

NUMA architecture emerged in the 1990s as an extension of SMP architecture. Unlike UMA, NUMA does not view memory as a single shared address space but comprises multiple CPU-Memory units forming an extended memory space. This architecture connects independent memory systems controlled by multiple operating system kernels using high-speed interconnect networks, forming a transparent unified memory unit for programs. In NUMA, memory access time depends on the spatial distribution of memory hardware relative to processors. NUMA does not guarantee equal access time for processors to all memory blocks like SMP does. However, it can leverage the reference locality in memory by providing each processor with an independent local cache to address the bottleneck issue of UMA and avoid performance losses when multiple processors are waiting for the same memory block. Therefore, NUMA improves the scalability of HPC systems by allowing more processor cores to be placed within the same shared memory system. However, due to the unevenness of memory access time, programmers need to pay attention to data locality to maximize cache utilization and exploit the advantages of the computing system. Cache-coherent NUMA (ccNUMA) is commonly used as a tightly-coupled shared memory solution across nodes in modern HPC systems, utilizing cache controllers to handle inter-processor communication and ensuring consistency among multiple memory copies of shared data.

The second type of memory architecture in HPC systems is the distributed memory system, and a computer employing this architecture is referred to as a Distributed Memory Machine (DMM). In this architecture, each CPU-Memory unit can only access its local memory. If a node requires data that does not exist in its local memory, it needs to communicate through a high-speed interconnect network that connects all nodes. The distributed memory machine decouples the network and nodes in system design by utilizing an improved interconnect network, making cross-node communication transparent to the host processors and executing local and remote memory communication in the background.

Due to the loosely coupled nature of CPU-Memory units in distributed memory machines, the complexity of system design and assembly is lower. However, a key challenge in distributed memory programming is data layout in memory. Programmers must decompose the computational data for a specified scientific problem into processing units with exclusive memory addresses. They must also explicitly specify communication between processors, ensuring that the requester and provider understand the data types, message formats, lengths, and other parameters of shared data. Another programming challenge in DMM is minimizing the number of cross-node data requests. Depending on the problem being solved, data can be statically distributed across memory systems, avoiding cross-node data transfers during runtime. Data can also be moved across nodes, with on-demand migration or pre-push to requesting nodes. With statically distributed computation, only edge-based data changes need to be reported to other nodes, resulting in less complexity in cross-node I/O. However, it is more difficult to implement. If dynamic data distribution is used, data distribution needs to be designed based on the topology of the HPC system to minimize memory latency.

2.2.4 Interconnection Architecture

In large-scale parallel computing systems with a distributed memory model, interconnect networks play a crucial role in enabling data sharing and communication across nodes. These networks can be broadly categorized into two types: direct interconnect and indirect interconnect. In direct interconnects, nodes have direct physical connections, enabling data transmission between neighboring nodes. On the other hand, indirect interconnects rely on multiple intermediate switching elements (SE) to relay data between shared memory blocks. Crossbar switches, buses, and multi-stage interconnect networks (MINs) are examples of indirect networks.

Direct interconnect networks, also known as point-to-point networks, exhibit a symmetric nature, meaning that their discovered topology is isomorphic from any node. This symmetry simplifies their design and implementation as they do not require relay nodes. Moreover, direct networks facilitate modular construction, expansion, and integration into computing clusters. Their scalability is not significantly impacted, resulting in strong scalability and the ability to construct large networks while maintaining structural flatness. The inherent symmetry of direct networks also makes designing efficient routing and switching algorithms more straightforward compared to indirect networks. Furthermore, the programming model for direct networks is simpler, enabling programmers to better utilize localities in network traffic and reduce complexity arising from data distribution.

Another critical aspect in designing interconnect networks is selecting the appropriate network topology, which refers to the arrangement of nodes and connection media. The chosen topology directly influences routing and switching strategies, as well as traffic control methods. For scientific computing applications that heavily rely on exchanging and synchronizing data through the interconnect network, the network topology can significantly impact data locality, leading to changes in network latency and throughput, thereby affecting overall network bandwidth. Thus, an efficient network topology is crucial for maximizing the performance of an HPC system.

Various network topologies can be employed in HPC applications, including single-router (crossbar switch topology), ring topology, mesh topology, torus topology, dragonfly topology [37], HyperX topology [38], among others. Notably, the Torus Fusion(Tofu) and Dragonfly topologies are commonly used in systems such as Fugaku and Frontier, respectively. Additionally, some topologies, like the Fat-Tree topology, can be configured with specific parameters to achieve unified link bandwidth and provide Fat-Tree network capabilities. However, modern HPC systems tend to deviate from standard Fat-Tree networks due to the pyramid-shaped architecture, which imposes increasingly higher network performance requirements as we move up the hierarchy.

Modern HPC network architectures often bypass higher levels of the tree to achieve lower network latency and a more flattened network structure while still maintaining a relatively consistent bandwidth distribution among nodes. These considerations are critical for designing efficient and high-performance interconnect networks in today's HPC environments.

2.3 Observability in Hardware

2.3.1 Performance Monitor Units

Performance Monitoring Units (PMUs) are a set of control registers and hardware counters reserved by the CPU for measuring and recording micro-architectural-level statistical data, providing performance infrastructure. Using hardware event sources provided by PMUs requires programming the control registers to capture specific events and count them. By further configuring the control registers, features such as generating interrupts when counters hit, collecting data only for user-mode processes or kernel-mode processes can be set. Although CPU vendors like Arm, Intel, and AMD have implemented some form of PMU features and provided corresponding support on their platforms, the types and quantities of measurable events in CPUs based on different micro-architectures and instruction set architectures, including the number of PMUs themselves, vary. The most common CPU measurement metrics include IPC (Instructions Per Cycle), cache hit rates at different levels, branch prediction success rate, etc.

Since PMUs belong to the hardware's low-level features, to access these hardware counters more easily, PMUs need to be programmed through software infrastructure to set the performance data to be monitored and control the start and stop of data collection. Additionally, PMU support software needs to read and aggregate values from the hardware counters to obtain performance sampling data. Several tools and system interfaces support collecting statistics from PMUs, including PAPI and eBPF mentioned in Chapter 1, as well as the Linux Perf tool. These tools use system calls to directly access the hardware counters of PMUs, and through the context switch during the system call process, the hardware counters can be easily saved and restored, enabling thread-level performance monitoring. This section mainly introduces the methods for obtaining PMU performance counter data using the Linux Perf interface.

The Perf tool can collect performance records for each thread, each process, and the entire CPU. Linux Perf provides various supports for PMUs, including raw counting, event-based sampling, fixed-rate sampling, and instruction-based sampling (only for AMD64). Taking event-based sampling as an example, the Perf_events function maintains a 64-bit virtual counter. When this event counter overflows, it indicates that the specified event has occurred a certain number of times, reaching the pre-defined threshold. The kernel then records information about the program's running state. This collection of information is referred to as a sample, and it includes the user-specified sampling event and an instruction pointer indicating the position where the timer interrupt was generated. Users can specify more events than actual counters, and in this case, the kernel will use time multiplexing (usually with a switching frequency of 100 or 1000 Hz) to give each event a chance to be sampled by the PMU. Table 2.1 and Figure 2.1 list the supported sampling events and their output results on Intel Xeon Platinum 8352Y with Perf. Since the target machine is a virtual machine, PMUs cannot be virtualized, and some events are shown as unsupported. However, HPC systems typically do not use virtualized environments, so this issue does not exist.

Nevertheless, event-based counter interrupt sampling is not accurate on modern processor architectures because the stored instruction pointer in the sample actually points to the interrupt handler of the program, not the position where the counter overflow occurred (i.e., the end of the sampling period). This does not pose an issue when the program is in a paused state, but in cases with jump instructions, the distance between these two points can be as large as several tens of instructions. Therefore, interpreting such reports requires combining the program's running conditions and experience.

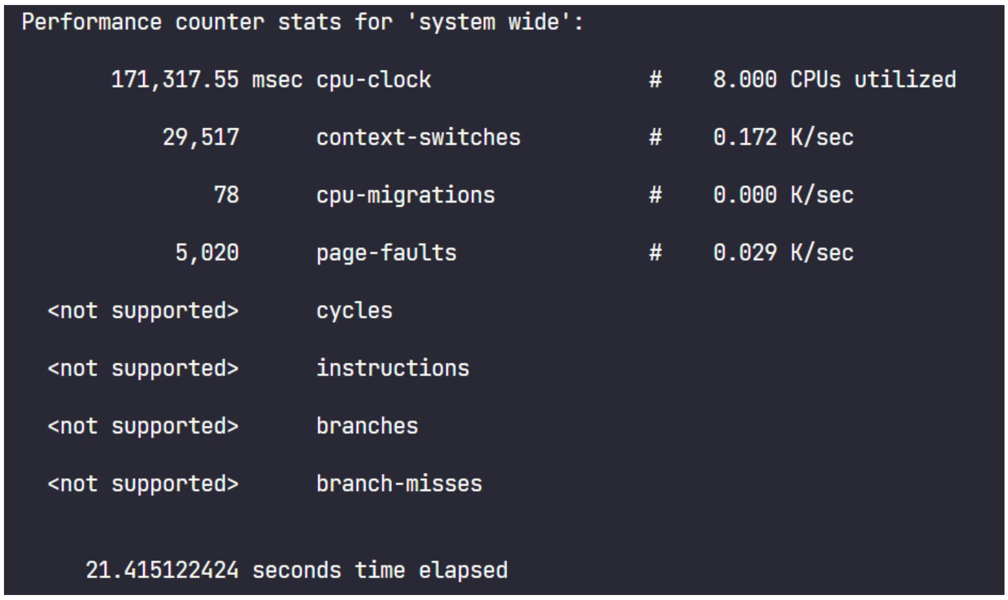
Event	Description
msr/pperf/	Kernel PMU event
msr/smi/	Kernel PMU event
msr/tsc/	Kernel PMU event
ref-cycles OR cpu/ref-cycles/	Kernel PMU event
rNNN	Raw hardware event descriptor
cpu/t1=v1[,t2=v2,t3 ...]/modifier	Raw hardware event descriptor
L1-dcache-load-misses	Hardware cache event
L1-dcache-loads	Hardware cache event
L1-dcache-stores	Hardware cache event
L1-icache-load-misses	Hardware cache event
branch-load-misses	Hardware cache event
branch-loads	Hardware cache event
dTLB-load-misses	Hardware cache event
dTLB-loads	Hardware cache event
dTLB-store-misses	Hardware cache event
dTLB-stores	Hardware cache event
iTLB-load-misses	Hardware cache event
mem:¡addr¿[/len][:access]	Hardware breakpoint

Table 2.1: List of Events and Descriptions

2.3.2 Top-Down Analysis

The Top-Down approach [39] is a performance optimization method aimed at accurately and rapidly identifying performance bottlenecks. It guides users to focus on the most critical performance issues during the optimization phase and helps them identify and understand the causes of performance problems through a hierarchical and progressive approach. The Top-Down approach is designed to be cost-effective, accurate, and low runtime overhead within the entire application development process and resource constraints. It has been well-controlled and widely recognized in the industry, with industrial-grade implementations such as Intel VTune.

The Top-Down method first classifies CPU execution time at a high level. In this step, the analysis tool marks areas that are worth further investigation. Then, users can apply the Top-Down decomposition on the interested hotspot functions until specific performance problems are identified or at least a subset of potential investigation questions is determined. This method categorizes performance issues originating from the CPU into four fundamental categories: Retiring, Bad Speculation, Frontend Bound, and Backend Bound. Based on these classifications, users can delve into



```
Performance counter stats for 'system wide':

      171,317.55 msec cpu-clock                 #    8.000 CPUs utilized

          29,517      context-switches          #    0.172 K/sec

              78      cpu-migrations            #    0.000 K/sec

           5,020      page-faults               #    0.029 K/sec

  <not supported>      cycles

  <not supported>      instructions

  <not supported>      branches

  <not supported>      branch-misses


     21.415122424 seconds time elapsed
```

Figure 2.1: Output of Perf.

these specific issues while disregarding problems unrelated to the hotspots.

2.3.2.1 Retiring

Retiring is a crucial metric in the Top-Down approach, reflecting the percentage of successfully executed and retired instructions. In modern processors, out-of-order execution is adopted, allowing instructions to be dynamically scheduled for execution based on data dependencies and executability, rather than following the program order sequentially. This design enhances instruction-level parallelism, fully utilizing processor resources, and improving program execution efficiency.

Ideally, all issued instructions would successfully retire, indicating that the processor fully utilizes its performance potential, achieving the maximum IPC (Instructions Per Cycle). For example, in a four-wide processor, if four instructions execute per cycle, achieving 100% Retiring means an IPC of 4.

However, Retiring is not the only performance factor to consider. Some instructions, like Floating Point Assists, can lead to performance degradation. Therefore, when measuring Retiring, it may be necessary to exclude such instructions that adversely affect performance to obtain more accurate performance analysis results.

Furthermore, Retiring also helps determine potential for optimization. For instance, if the Retiring value is high but the performance has not reached the desired level, code vectorization can be considered to enhance performance. Vectorization allows multiple operations to be completed simultaneously, increasing IPC and overall performance.

2.3.2.2 Bad Speculation

Bad Speculation is another key metric in the Top-Down approach, measuring performance losses caused by incorrect predictions. Modern processors employ branch predictors to anticipate the execution path of branch instructions in advance to enhance instruction-level parallelism. However, branch predictors are not always 100% accurate, leading to Bad Speculation. Bad Speculation can be understood in two parts:

1. **Instructions not retired due to prediction errors:** When the branch predictor inaccurately predicts the execution path of a branch, subsequent instructions are mistakenly dispatched to the processor's execution units, but these instructions will not ultimately retire successfully. These mispredicted instructions are canceled during actual execution, resulting in performance loss. This part of Bad Speculation reflects the accuracy of branch prediction and its impact on the execution of branch instructions.
2. **Performance loss due to recovery from prediction errors:** When the branch predictor inaccurately predicts the execution path of a branch, subsequent instructions have already been dispatched to the processor's execution units. However, due to the branch prediction error, recovery operations are required. These recovery operations typically involve clearing the erroneously predicted instructions from the execution units and re-executing the correct instructions. These recovery operations consume processor resources and lead to performance degradation. This part of Bad Speculation reflects the impact of branch prediction errors on performance.

The ratio of Bad Speculation can reflect the accuracy of the branch predictor. A higher ratio of Bad Speculation suggests that the branch predictor needs improvement. Additionally, the presence of Bad Speculation can affect the accuracy of other metrics. Therefore, when analyzing other metrics, addressing Bad Speculation should be given priority. For example, if the main performance bottleneck is caused by Bad Speculation, the observation results of other metrics might be overshadowed by Bad Speculation.

In the Top-Down approach, Bad Speculation is further divided into two subcategories: Branch Misspredict and Machine Clears.

1. **Branch Misspredict:** This subcategory reflects performance losses

caused by branch prediction errors. When the branch predictor inaccurately predicts the execution path of a branch, subsequent instructions are affected, leading to performance degradation.

2. **Machine Clears:** This subcategory reflects performance losses caused by other errors, such as those resulting from erroneous data predictions. This categorization helps pinpoint the issues more accurately and enables more effective performance optimization.

2.3.2.3 Frontend Bound

Frontend Bound is used to reflect performance bottlenecks in the processor's frontend. The frontend is the first stage of instruction execution in the processor, including operations such as branch prediction, instruction fetching, decoding, and transformation into micro-operations. In modern processors, the frontend is a critical component that affects the overall performance of the processor.

Frontend Bound mainly focuses on whether the frontend can supply enough bandwidth to meet the needs of the backend. When the frontend cannot provide a sufficient number of micro-operations to the backend in a timely manner, it may result in idle execution units in the backend, leading to performance loss. For example, if the branch predictor cannot accurately predict the execution path of branch instructions, causing delays in fetching and decoding subsequent instructions, the backend execution units may not have enough instructions to execute, resulting in stalls.

Frontend Bound's performance impact is also related to the program's branch density. In branch-intensive programs, the frontend may become a performance bottleneck, as the processor needs to frequently predict and handle branch instructions. In such cases, optimizing the frontend's performance can improve overall performance by fully utilizing the processor's execution resources and enhancing instruction-level parallelism.

In the Top-Down approach, Frontend Bound is further divided into two subcategories: Frontend Latency Bound and Frontend Bandwidth Bound.

1. **Frontend Latency Bound:** This subcategory reflects performance bottlenecks caused by frontend delays. For example, if the prediction latency of branch instructions is high, causing delays in fetching and decoding subsequent instructions, these delayed instructions may not be delivered to the backend execution units in time, resulting in performance loss.
2. **Frontend Bandwidth Bound:** This subcategory reflects performance bottlenecks caused by frontend bandwidth limitations. For example, if the frontend's bandwidth cannot satisfy the demands of the

backend execution units, the backend execution units may experience stalls, resulting in performance loss.

When conducting performance optimization, measures can be taken to improve the frontend's performance with respect to Frontend Bound. For example, improving the accuracy of the branch predictor, optimizing the instruction cache, and enhancing frontend bandwidth can be considered. By optimizing Frontend Bound, the processor's execution resources can be fully utilized, enhancing instruction-level parallelism, and improving overall program performance.

2.3.2.4 Backend Bound

Backend Bound is used to reflect performance bottlenecks in the processor's backend. The backend is the subsequent stage of instruction execution in the processor, including instruction execution and retirement. In modern processors, the backend typically consists of multiple execution units that can execute multiple micro-operations in parallel.

Backend Bound primarily focuses on whether backend resources are sufficient to meet the demands of instruction execution. When backend resources are insufficient to execute current instructions, it may lead to delays or blockages in instruction execution, resulting in performance loss. For example, if the backend execution units cannot satisfy all instructions that need to be executed, some instructions may wait in the backend, leading to idle backend resources.

Backend Bound is further divided into two subcategories: Memory Bound and Core Bound.

1. **Memory Bound:** This subcategory reflects performance bottlenecks caused by limitations in the memory subsystem. The memory subsystem includes various levels of cache and main memory. When the processor needs to frequently fetch data from memory, it may lead to waits for data in the backend execution units, resulting in performance loss. For example, if the execution of an application is mainly limited by memory access latency, it may be classified as Memory Bound.
2. **Core Bound:** This subcategory reflects performance bottlenecks caused by limitations in the core resources. Core resources include execution units, multipliers, dividers, etc. When the processor needs to perform complex computational operations, it may lead to busy backend execution units or partially idle execution units, resulting in performance loss. For example, if the execution of an application is mainly limited by complex computational operations, it may be

classified as Core Bound.

By analyzing Backend Bound, it is possible to understand the utilization of backend resources and identify the location of performance bottlenecks. For different Backend Bound situations, different optimization measures can be taken. For example, for Memory Bound, optimizing memory access patterns to reduce memory latency may be attempted. For Core Bound, optimizing computational operations to improve the utilization of execution units may be considered. By optimizing Backend Bound, the efficiency of backend resources can be improved, thus enhancing overall program performance.

2.3.3 I/O Tracing Techniques

In conventional software performance engineering, the analysis of the I/O system can be categorized into the following three types based on the sampling hierarchy of performance events:

1. Analysis based on monitoring tools: By utilizing monitoring tools such as iostat, real-time metrics such as I/O bandwidth, IOPS, and response time can be obtained, enabling the evaluation of an application's I/O performance.
2. Analysis based on tracing tools: Tracing tools like strace and DTrace are employed to capture the file access behavior of applications, allowing for the analysis of I/O patterns and performance bottlenecks.
3. Analysis based on file system: Gathering statistical information about the file system, such as file sizes, the number of files, and space utilization, provides insights into the file system's performance and structure, facilitating I/O optimization.

However, the I/O analysis tools used in conventional software engineering are inadequate in supporting I/O infrastructures for large-scale parallel computing. Therefore, specialized parallel I/O analysis tools, such as Darshan [40], have been designed specifically for large-scale scientific computing applications to support advanced I/O libraries, parallel I/O file formats, and parallel file systems. Darshan can capture an application's file access behavior and record key characteristics such as access patterns, access sizes, and the timing of I/O operations. Compared to traditional profiling methods, parallel I/O analysis tools accurately capture memory access patterns of large-scale scientific computing applications, ensuring a comprehensive understanding of parallel I/O behavior under the influence of advanced libraries or file systems, thus revealing performance bottlenecks and optimization opportunities.

2.4 Identify Software Activities

2.4.1 Source Code Inter-positioning

Code instrumentation is a common method used to understand the behavior of target functions or code segments at runtime. Depending on the performance analysis requirements, users inject probe code into the entry points of target functions or attach it to the memory addresses of target code segments. This allows pre-defined sampling functions to be executed when the target code starts, enabling the observation of behaviors such as the number of calls and execution times of the target function. Based on whether source code modifications are required, code instrumentation can be classified into the following two types:

1. Static instrumentation: Static instrumentation involves users wrapping the target function with special functions within the observation scope of the source code. In the context of performance engineering integrated into CI/CD pipelines, programmers often use static instrumentation combined with conditional compilation to minimize the runtime overhead introduced by probe code. This approach also allows for precise event collection and reduces difficulties in analysis caused by data explosion.
2. Dynamic instrumentation: In contrast to static instrumentation, dynamic instrumentation achieves runtime function interception by attaching sampling code to the memory addresses of target functions. Dynamic instrumentation is essentially a black-box method, as performance engineers do not require access to the source code, but they need to have at least the ability to debug the memory address distribution of the database or target functions. However, the lack of portability in dynamic instrumentation arises from changes in the memory addresses of the observed target functions during application upgrades or refactoring.

Probe technology is implemented by various tools, and although they are based on different underlying mechanisms, they generally offer similar observation capabilities. Some common implementation methods are:

- **LLVM Pass:** LLVM Pass is a widely used code instrumentation technique that allows the insertion of machine-independent, portable target code while traversing LLVM Intermediate Representation (IR). Probe technology is one application of LLVM Pass and is relatively straightforward for detecting branch instructions and loop bodies.

- **SystemTap/DTrace/eBPF:** Operating system kernels typically provide some degree of observability technology. DTrace, as a pioneer in this area, is a general-purpose debugging platform tightly integrated with the operating system kernel. The observability technology of DTrace and eBPF is essentially kernel virtual machines that enable kernel probe placement at the entry or exit of a kernel function, the entry or exit of a user-space process, or even at any program statement or machine instruction using restricted D language or C language bindings (BCC). While such observability technology is widely used in Linux SRE or cloud-native scenarios, there is still no systematic research for the HPC domain.
- **Scalasca/Tau, etc.:** The code instrumentation technology used in HPC workloads is essentially a form of probe technology. HPC performance analysis tools such as Scalasca and Tau achieve dynamic linking of event sampling libraries at runtime through specialized compilers and MPI runtimes, allowing them to collect and aggregate the runtime behavior of the program.

2.4.2 Profiling with Stack Tracing

Stack Tracing is the most commonly used method for CPU profiling and is also a prevalent approach for understanding software runtime behavior. Essentially, it is a sampling technique based on interrupt timers. Within a fixed time window, a CPU profiler registers a timed execution hook (e.g., SIGPROF signal) with the target program. During this hook, the current stack information is obtained by inspecting the stack address of the target application. At the end of the time window, all collected samples are aggregated to determine the number of times each function was sampled. By calculating the ratio of this value to the total number of samples, the relative proportion of each function is obtained. Through this process, we can easily identify functions with higher runtime proportions, known as hot functions. Moreover, due to the nature of function calls, we can effortlessly identify the longest function call chain in terms of runtime proportion. By analyzing this chain of dependencies, we can identify the hot execution path of the target program, thus helping users locate and identify CPU bottlenecks.

There are several common methods to implement Stack Tracing:

1. Language runtimes: Most high-level languages based on garbage collection (GC) or complex runtimes, such as Java, C#, Python, etc., have native Stack Tracing functionality. When an application crashes, it prints the stack trace to the console, which helps locate problematic

sections in the source code. This feature can also be utilized by profilers to design analysis tools that identify runtime hotspots in applications implemented in specific languages.

2. Profiling tools like Linux Perf: Tools like Perf can attach to a target application using hardware timer interrupts or software interrupts. They utilize system calls to record stack information from either kernel-mode programs or user-mode programs during interrupt occurrences and then aggregate and display the data at the end of the sampling process.
3. Kernel-level observability tools like eBPF: Kernel-level observability tools have the ability to perform event sampling at the kernel level. Their implementation is similar to Linux Perf, but they reside in the kernel space, allowing some performance overhead caused by context switches to be reduced. Additionally, eBPF-based stack tracing may have difficulty supporting high-level language runtimes; therefore, it is generally used for analyzing programs with minimal runtimes, such as those implemented in C language.

Chapter 3

eBPF-based Performance Method

This section introduces the architecture, methodology, and evaluation methods employed in the design of performance tools. It also includes discussions on key features, performance sampling modes, and performance pattern recognition methods. Additionally, the feasibility of using XGBoost for automated performance pattern recognition is proposed.

For the measurement, aggregation, and comparison of parallel application performance, we rely on multiple runs of eBPF probe code to monitor the execution state of working processes. eBPF, an extension of BPF (Berkeley Packet Filter), is a general-purpose kernel virtual machine. Compared to other data sampling techniques, eBPF provides highly efficient observability with minimal runtime overhead and excellent non-invasiveness. It enables low-cost online dynamic performance monitoring and tracing of various kernel activities, such as memory allocation and file I/O operations. Specifically, our work is divided into three parts:

1. Data Collection: As mentioned earlier, we leverage eBPF's low-overhead observability to monitor performance events. In this paper, we use C to write eBPF programs in both kernel and user space to implement code instrumentation based on USDT (User Statically Defined Tracing), supporting white-box measurement methods. Additionally, we utilize Linux kernel probes to support kernel sampling technologies like ptrace and SystemTap, providing black-box measurement methods based on hardware timer interrupts for post-analysis. Apart from the event visibility provided by eBPF, we use kernel extraction and kernel benchmarking to understand the computational nature and theoretical resource requirements of the workloads. We also employ LLVM's dependency chain analysis to identify coding issues that might cause scalability bottlenecks on specific architectures.
2. Data Preprocessing: Due to variations in measurement units, significant deviations between feature vectors may adversely affect subsequent data analysis and the training results of neural network models. Therefore, in this step, we use standardization methods to clean the

dataset.

3. Modeling and Evaluation : Using the preprocessed eBPF sampling data along with the results from kernel benchmarking and dependency chain analysis, we model the architectural adaptability of workloads. We also attempt to use an XGBoost model to explore the potential of unsupervised online performance analysis.

3.1 Framework Architecture

3.1.1 Code Instrumentation

Code instrumentation serves as the fundamental module in the front-end of performance sampling, providing essential source code and runtime-level performance data for performance analysis tools. The design and implementation of this module not only influence the user-friendliness of code instrumentation but also determine the level of intrusiveness and overhead during the sampling process. While code instrumentation unavoidably involves some intrusion into the source code and control flow, it exhibits less intrusiveness and higher controllability compared to other techniques like debuggers with breakpoints or binary code analysis.

Developers of performance tools must optimize the instrumentation process to minimize overhead. Currently, for languages such as C/C++ and Fortran, code instrumentation can be achieved through compile-time dynamic linking of shared libraries, employing either automatic or manual instrumentation. Automatic instrumentation is suitable for black-box analysis, where understanding of the source code is limited, and modifications to the code are unnecessary. However, the generated data reports may lack specificity, making it challenging to pinpoint relevant information. Moreover, automated instrumentation often introduces a considerable amount of injected code for performance sampling, resulting in substantial runtime overhead.

In contrast, manual instrumentation is preferable when specific code segments within the application require inspection, irrespective of the code content. To ensure a performance engineering method that is language- and compiler-independent, this paper chooses to implement portability and WORA (Write Once, Run Anywhere) characteristics through manual instrumentation. Manual instrumentation is based on the User Statically Defined Tracing (USDT) feature provided by eBPF. USDT enables the insertion of static trace points into the application, allowing developers to insert probes for debugging and performance sampling at critical points in the program. These probes can be dynamically activated at runtime by tools like DTrace, SystemTap, or eBPF without interfering with the normal operation of the program, thereby obtaining internal states and performance metrics of the program.

Moreover, the universality of eBPF bytecode ensures that the same eBPF code and target program source code can achieve unified performance analysis functionality on supported Linux kernel versions without any modification to the program code. This feature enhances the overall portability and ease of use of the proposed performance engineering methodology, making it a flexible and widely applicable approach for performance analysis in HPC systems.

3.1.1.1 User Statically Defined Trace(USDT)

Static tracepoints, also known as User Statically Defined Tracing (USDT) probes in user space, represent a valuable user-level observability capability offered by eBPF. By incorporating tracing macros into specific functions of interest within the application, kernel trace code can be attached to examine code execution states and data. To utilize USDT, developers need to explicitly define it in the source code and enable it during compilation using flags such as ”–enable-dtrace.”

USDT code instrumentation supports predefined tracepoints provided by tools like BCC-tools, where the code already contains defined USDT probes, and users simply need to identify their presence in the distribution version of the application. Alternatively, custom tracepoints can be manually added using APIs provided by kernel tools like SystemTap or development packages like libstapsdt. This method offers various advantages, including tracepoint ABI stability, dynamic runtime safety, and fast reconfigurability. Consequently, it proves highly beneficial for swift analysis and resolution of performance issues without necessitating a restart of the target program. Moreover, static tracepoints are particularly advantageous for applications written in high-level languages like Python or Java, as the development toolset supporting USDT provides specific support for language runtimes and advanced features.

However, the explicit definition of static tracepoints in the source code introduces certain limitations. Modifying these tracepoints entails rebuilding the application, which poses challenges for maintainers concerning the stability and maintenance of existing tracepoints' ABI. Additionally, conventional eBPF bytecode operates in the kernel, leading to a kernel-user space context switch each time eBPF samples a user process. Although this context switch is relatively small compared to general debuggers and is often considered negligible, it can have implications for performance-sensitive scientific computing applications. Nevertheless, the impact of this switch can be minimized through specialized eBPF implementations like uBPF, a user-space BPF virtual machine, or JIT compilation, effectively mitigating any adverse effects on performance.

3.1.1.2 Dynamic Probes(uprobes)

Dynamic probes are a form of fully runtime dynamic user-space probes implemented based on the "uprobes" Linux kernel tracing module. Unlike static probes, dynamic probes do not require explicit definitions in the source code and have the capability to directly access the memory of a running application temporarily, without necessitating any special source code modifications. To attach an uprobe to a memory address corresponding to the target function within the running process, tools like objdump or /proc/maps can be used to manually calculate the offset. However, this process can be cumbersome and lacks portability. Fortunately, many Linux distributions offer debugging symbol databases, and using BCC (BPF's C language binding) can automatically resolve symbol names, simplifying the attachment of dynamic probes to target memory addresses.

In comparison to static probes, dynamic probes provide a more decoupled approach from the source code, resembling a black-box analysis. This attribute allows for effective real-time debugging of running instances and online performance analysis. However, the lack of stability in the Application Binary Interface (ABI) of dynamic probes across different application versions is a noteworthy limitation. Changes in the application's source code can render dynamic probes ineffective, as they may no longer match the target application's address space. Despite this limitation, dynamic probes remain highly valuable for performance analysis, especially in low-level languages such as C, C++, or Rust.

It is important to recognize that dynamic probes may encounter challenges when analyzing programs written in high-level languages. The memory-level operation of dynamic probes may restrict their access to language runtime implementations, thus limiting their effectiveness in such scenarios. Nevertheless, for performance analysis in low-level languages, dynamic probes prove to be a powerful and versatile tool.

3.1.1.3 Overhead Analysis

One crucial non-technical metric of performance analysis tools is the runtime overhead, which denotes the impact of sampling behavior or the instrumentation process on the program's control flow and execution time. Table 3.1

compared the tracing overheads on some common system calls and events. Both USDT and dynamic probes are activated only when the target function is encountered, resulting in minimal intrusion at the beginning and end of program execution. As a consequence, eBPF programs are initialized in both user and kernel space.

Event	**Typical Frequency**	**Tracing Overhead**
Thread sleeps	1 per second	$< 0.1\%$
Process execution	10 per second	$< 0.1\%$
File opens	10–50 per second	$< 0.1\%$
Profiling at 100 Hz	100 per second	$< 0.1\%$
New TCP sessions	10–500 per second	$< 0.1\%$
Disk I/O	10–1000 per second	$< 0.1\%$
VFS calls	1000–10,000 /s	$\sim 1\%$
Syscalls	1000–50,000 /s	$> 5\%$
Network packets	1000–100,000 /s	$> 5\%$
Memory allocations	10,000–1,000,000 /s	$> 30\%$
Locking events	50,000–5,000,000 /s	$> 30\%$
Function calls	Up to 100,000,000 /s	$> 300\%$
CPU instructions	Up to 1,000,000,000+ /s	$> 300\%$
CPU cycles	Up to 3,000,000,000+ /s	$> 300\%$

Table 3.1: Tracing overheads on typical events, data from [1].

The invasiveness during function hit depends primarily on factors such as the monitored function's call frequency, runtime, the number of processor units utilized during the process, and the specific measurement method employed. Furthermore, the user-space overhead of eBPF-based performance analysis is predominantly observed in the context switches that occur when data is transferred from the kernel to user space. Therefore, it is imperative to evaluate this aspect. To estimate the magnitude of this overhead, a simple benchmark test code can be employed to assess the observer effect introduced by eBPF operations during repeated calls to the sampling function. The Python code for this benchmark test is presented in Code 3.1.1.3.

Code 3.1.1.3 User-State eBPF Overhead Measurement

```c


#include <uapi/linux/ptrace.h>

BPF_HASH(start_time, u32, u64);
BPF_HISTOGRAM(exec_time_hist);

int trace_func_entry(struct pt_regs *ctx) {
    u32 pid = bpf_get_current_pid_tgid();
    u64 timestamp = bpf_ktime_get_ns();

    start_time.update(&pid, &timestamp);
    return 0;
}

int trace_func_return(struct pt_regs *ctx) {
    u32 pid = bpf_get_current_pid_tgid();
    u64 *timestamp = start_time.lookup(&pid);

    if (timestamp) {
        u64 delta = bpf_ktime_get_ns() - *timestamp;
        exec_time_hist.increment(bpf_log2l(delta));
        start_time.delete(&pid);
    }

    return 0;
}
```

The time taken for N function calls to execute, both with and without eBPF instrumentation, was meticulously recorded. The algorithm underwent execution with 1000, 2000, 3000, 4000, and 5000 function calls, allowing for the subsequent calculation of average, median, and variance values. The obtained results are thoughtfully depicted in Figure 3.1(a) and Figure 3.1(b).

Figure 3.1(a) presents a noteworthy observation, highlighting the varying mean runtimes of the function when subjected to eBPF instrumentation, across different call and measurement counts. To complement these findings, Figure 1b exhibits the corresponding variances for each execution. Please note that the specific runtime overhead depends on the particular computing architecture, but the proportion between the runtime overhead and the total runtime of the program should remain consistent.

For a comprehensive analysis, Table 3.2 provides a concise presentation of the experimental outcomes, offering a direct comparison between the execution times with and without eBPF instrumentation. It is evident from the table that the impact, or overhead, remains consistent as the interaction count increases.

Number of Runs	Time Difference (seconds)	Time Ratio
500	1.751e-05	1.006
1000	9.998e-06	1.003
1500	-3.762e-06	0.998
2000	3.162e-05	1.011
2500	2.549e-05	1.009
3000	-1.734e-05	0.994
3500	1.384e-05	1.005
4000	2.474e-05	1.009
4500	-5.993e-06	0.998
5000	4.190e-05	1.015
5500	-2.024e-06	0.999
6000	2.471e-05	1.009
6500	1.821e-05	1.006
7000	3.709e-05	1.013
7500	2.476e-05	1.009
8000	2.504e-05	1.009
Average	1.805e-05	1.005
Median	9.998e-06	1.008

Table 3.2: Comparison of eBPF Performance Overhead

3.1.2 Event Tracing Probes

The observation of operating system kernel-level events poses a significant challenge for most commonly used HPC performance analysis tools. Scientific computing software often involves cross-language linking (C/Fortran) and tracing of non-code components. While operating system-level monitoring can incur additional runtime overhead, which may be sensitive for HPC workloads, static tracing of kernel-provided functionalities, such as system calls, socket operations, resource management, and process scheduling, is crucial for effective performance modeling.

In this section, we first provide a brief introduction to the eBPF-based system call method, followed by a detailed discussion on the use of kernel probes for tracing system calls.

The primary functions of an operating system are to abstract the underlying system complexity from users and efficiently manage system resources. This abstraction is accomplished through services and a layered abstract API interface, with kernel methods exposed to programmers as system calls. While some HPC systems still rely on proprietary operating systems, they typically adhere to UNIX semantics, offering concepts such as process management, file descriptors, memory, IPC, and sockets.

Conventional kernel tracing involves wrapping the kernel source code through static instrumentation to generate essential low-level behavior logs for performance engineering and power management. However, this method introduces considerable overhead for frequently called functions like system calls, leading to an overall degradation in system performance. To address this issue, programmers have adopted dynamic instrumentation since the 1990s, which enables runtime monitoring of specific areas of interest in the operating system. This approach, known as observability technology, does not alter the target software execution flow and eliminates the need for modifying the target code. As a result, observability technology boasts lower runtime overhead and can be directly applied to production environments.

DTrace [41], developed by Oracle for the Solaris project in 2005 and later integrated into macOS (2007), FreeBSD (2008), and Windows (2019) platforms, attempted to merge into the Linux mainline tree but did not succeed. While DTrace did not directly provide its dynamic probe capabilities to Linux, its design concepts played a pivotal role in inspiring and influencing the extension of BPF to evolve into eBPF. Both eBPF and DTrace share fundamental concepts, such as probes, predicates, and actions, empowering users to capture kernel-level or user-level events and execute custom code within the kernel. DTrace has made substantial contributions to general software engineering aspects, performance overhead reduction, and security enhancements, offering safety mechanisms to ensure bytecode execution in a protected execution environment, such as a sandbox. Both dynamic instrumentation technologies have gained widespread usage and certification in industrial settings.

Bpftrace serves as an eBPF front-end inspired by DTrace and SystemTap, designed with a high-level language syntax. Utilizing LLVM as a backend, bpftrace translates programs into BPF bytecode and seamlessly interacts with the Linux BPF system using BCC. It supports existing kernel tracing techniques, such as kprobes, uprobes, and USDT. Bpftrace significantly simplifies the use of eBPF and enables complex performance analysis tasks to be accomplished with a single line of code.We have listed some one-liners in Code 3.1.2 to demonstrate its ability.

Code 3.1.2 bpftrace One-liners, data from bpftrace project on Github.

```
# Files opened by process
bpftrace -e 'tracepoint:syscalls:sys_enter_open { printf("%s %s\n", comm, str(args->filename)); }'

# Syscall count by program
bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'

# Read bytes by process:
bpftrace -e 'tracepoint:syscalls:sys_exit_read /args->ret/ { @[comm] = sum(args->ret); }'

# Read size distribution by process:
bpftrace -e 'tracepoint:syscalls:sys_exit_read { @[comm] = hist(args->ret); }'

# Show per-second syscall rates:
bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @ = count(); } interval:s:1 { print(@); clear(@); }'

# Trace disk size by process
bpftrace -e 'tracepoint:block:block_rq_issue { printf("%d %s %d\n", pid, comm, args->bytes); }'

# Count page faults by process
bpftrace -e 'software:faults:1 { @[comm] = count(); }'

# Count LLC cache misses by process name and PID (uses PMCs):
bpftrace -e 'hardware:cache-misses:1000000 { @[comm, pid] = count(); }'

# Profile user-level stacks at 99 Hertz, for PID 189:
bpftrace -e 'profile:hz:99 /pid == 189/ { @[ustack] = count(); }'


# Files opened, for processes in the root cgroup-v2
bpftrace -e 'tracepoint:syscalls:sys_enter_openat /cgroup == cgroupid("/sys/fs/cgroup/unified/mycg")/ { printf("%s\n", str(args->filename)); }'
```

With the advantages of eBPF-based observability technology, exemplified by tools like BPFtrace, represents a significant advancement in HPC performance analysis. By enabling dynamic tracing of operating system kernel-level events with low runtime overhead and offering a user-friendly interface, eBPF empowers performance analysts to delve deeper into the intricacies of HPC systems. The combination of dynamic probes and high-level language syntax makes eBPF-based tools like BPFtrace highly effective for real-time debugging, performance analysis, and optimization in scientific computing applications. With its versatility and compatibility across different kernel versions, eBPF-based observability technology holds great promise for further advancements in HPC performance analysis and engineering.

3.1.3 Profiling & Kernel Extraction

Profiling serves as a subset of sampling techniques based on interrupt timers and plays a crucial role in understanding application behavior. It aids programmers and performance engineers in identifying hotspots in the source code by analyzing the execution flow and hotspot areas of the workload. Profile Guided Optimization (PGO) is a dynamic debugging technique that utilizes runtime profiling data to optimize code. In scientific computing applications, where the execution flow remains relatively fixed, especially during program initialization, Profiling can directly pinpoint hot functions and execution paths through stack tracing and runtime timing. This information allows for the extraction of theoretical resource requirements for the computational workload using kernel extraction techniques. In this section, we elaborate on the stack tracing analysis method used in our study. We discuss both basic techniques for stack tracing in general software engineering and those specific to dealing with large-scale parallel application spaces, some of which have already been implemented in existing tools. As stack tracing can be computationally expensive, especially for high-cost scenarios such as HPC applications, Profiling is generally not utilized as the primary source of performance information when the source code structure is known. Moreover, due to the complexity of stack tracing, we aim to keep kernel operations to a minimum, focusing on discussing general methods of implementing stack tracing using eBPF. We then explore how stack tracing enables us to identify and analyze application behavior.

Basic stack tracing involves capturing the relationship between the caller and the called functions currently executing in a process. By representing function call paths as a tree or directed graph structure, we gain insights into the program's runtime behavior. For instance, we can often identify functions responsible for runtime initialization and resource recovery at the program's beginning and end, while the functions between them constitute the program's core business logic. However, general singleton stack tracing may not be suitable for large applications due to a large number of processes leading to different entry points and resulting in numerous stack traces, which can be overwhelming.

To address the limitations of singleton stack tracing, concepts like 2D-Trace have been proposed in projects like TotalView and Prism. This approach consolidates individual stacks of each application process into a call graph prefix tree, effectively mapping stack tracing to the application processes. This strategy reduces stack redundancy and streamlines the information users need to focus on. However, stack tracing methods alone may not fully address performance issues related to function runtime, such as application runtime state, hotspot analysis, and lock status. Therefore, by recording the runtime of functions at specific sampling intervals, we can analyze the process's behavior over time. However, it is essential to carefully select the sampling cycle length, control the precision and granularity of stack tracing, and ensure that the behavior of the target function can be adequately sampled while minimizing runtime overhead. It should be noted that stack tracing is highly reliant on the timer's accuracy and sampling interval; excessively large or small intervals can lead to severe runtime performance issues or overlook critical performance problems.

In this paper, our focus is on identifying the program's execution flow and hot functions through profiling. As such, we will employ the singleton stack tracing method to analyze the behavior of large-scale parallel programs on the host. Two main implementations of eBPF-based stack tracing will be explored, namely manually mapping stack addresses and using the automatic stack tracing feature provided by BCC, as depicted in Code 3.1.3 and Code 3.1.4.

Code 3.1.3 Manual stack address mapping, code from Brendan Gregg.

```
#define MAXDEPTH    10
struct key_t {
    u64 ip;
    u64 ret[MAXDEPTH];
};
BPF_HASH(counts, struct key_t);

static u64 get_frame(u64 *bp) {
    if (*bp) {
        // The following stack walker is x86_64 specific
        u64 ret = 0;
        if (bpf_probe_read(&ret, sizeof(ret), (void *)(*bp+8)))
            return 0;
        if (bpf_probe_read(bp, sizeof(*bp), (void *)*bp))
            *bp = 0;
        if (ret < __START_KERNEL_map)
            return 0;
        return ret;
    }
    return 0;
}

int trace_count(struct pt_regs *ctx) {
    FILTER
    struct key_t key = {};
    u64 zero = 0, *val, bp = 0;
    int depth = 0;

    key.ip = ctx->ip;
    bp = ctx->bp;

    // unrolled loop, 10 (MAXDEPTH) frames deep:
    if (!(key.ret[depth++] = get_frame(&bp))) goto out;
    if (!(key.ret[depth++] = get_frame(&bp))) goto out;
    if (!(key.ret[depth++] = get_frame(&bp))) goto out;
    if (!(key.ret[depth++] = get_frame(&bp))) goto out;
    if (!(key.ret[depth++] = get_frame(&bp))) goto out;
    if (!(key.ret[depth++] = get_frame(&bp))) goto out;
    if (!(key.ret[depth++] = get_frame(&bp))) goto out;
    if (!(key.ret[depth++] = get_frame(&bp))) goto out;
    if (!(key.ret[depth++] = get_frame(&bp))) goto out;
    if (!(key.ret[depth++] = get_frame(&bp))) goto out;
out:
    val = counts.lookup_or_init(&key, &zero);
    (*val)++;
    return 0;
}
```

Code 3.1.4 Auto BCC stack tracing, through BPF_STACK_TRACE function.

```c
// BPF Code
#include <uapi/linux/ptrace.h>
#include <linux/sched.h>

struct key_t {
    // no pid (thread ID) so that we do not needlessly split
        this key
    u32 tgid;
    int kernel_stack_id;
    int user_stack_id;
    char name[TASK_COMM_LEN];
};

BPF_HASH(counts, struct key_t);
BPF_STACK_TRACE(stack_traces, 1024);

// Count Function Statement
int trace_count(void *ctx) {
    FILTER
    struct key_t key = {};
    key.tgid = GET_TGID;
    STORE_COMM
    %s
    counts.atomic_increment(key);
    return 0;
}
```

Through profiling, valuable stack backtrace information can be obtained, as depicted in Figure 3.2 and Figure 3.3. This data includes the names of the most frequently called functions or functions with the longest running time, presented in a top-down manner. In scientific computing applications, common math libraries like MKL, FFTW [42], and ScaLAPACK [43] are often utilized, allowing us to readily identify the types of workloads predominantly used through function names. We refer to the core part of this workload as the "kernel," representing a code segment that significantly influences the entire lifecycle of the program. The computation kernel of a scientific computing application can be described as a set of basic blocks connected by some form of loop structure. Identifying the kernel is vital as it facilitates understanding the computational nature and resource requirements from the perspective of computational resources. For instance, when applying the Habakkuk method to extract and analyze the kernel from the computation code of an FFT library, we can obtain data similar to Figure 3.4, Figure 3.5 and Figure 3.6. By aggregating the computing performance achievable on the critical path, we can estimate the theoretical performance peak of the target application in a single-core scenario. This valuable insight enables us to assess the application's computational capabilities and resource utilization efficiency, providing a solid foundation for further performance optimization efforts.

3.1.4 Parallel Performance Modeling

3.1.4.1 Dependency Chain

Dependency chains represent a critical constraint that influences the throughput of multi-level pipelined processors, a characteristic inherent in the nature of pipelining. In the context of a pipelined processor, the execution can occur out-of-order, while the fetch and retirement stages must adhere to a specific order. Consequently, instructions forming a dependency chain, even if their results are ready for writing back to memory, must await the retirement of preceding instructions.

Parallel runtime systems, such as COMP Superscalar (COMPSs) [44], offer valuable data flow and dependency chain analysis capabilities, empowering programmers to comprehend the high-level behavior of their applications on the target system. However, the delay attributed to dependency chains is highly dependent on the specific microarchitecture implementation, necessitating a combined analysis that considers the instruction latency on a particular processor.

Furthermore, various microarchitecture implementations incorporate branch prediction mechanisms, enabling the machine to speculate on the taken control flow path ahead of time. These speculations can lead to the speculative execution of branch commands, even when the instruction still occupies CPU ports.

To prioritize portability, we leverage the uops.info Code Analyzer (UiCA) [45] framework and exploit the dependency chain analysis functionality offered by LLVM. We have shown its capability in Figure 3.7 and Figure 3.8. This approach ensures that our performance analysis methodology is versatile and can be applied across different systems and architectures. The combination of UiCA and LLVM's capabilities allows us to achieve comprehensive performance insights while maintaining compatibility with various computing environments.

3.1.4.2 Kernel & I/O Benchmark

Given the homogeneity of scientific computing workloads, benchmarking computational kernels is beneficial for theoretical performance modeling of unknown workloads. It also helps identify whether the target architecture can meet the computational resource and I/O bandwidth requirements for specific types of computations. The total execution time of a computational task can be roughly represented as shown in (3.1). Here, NB represents network bandwidth, BW represents memory bandwidth per CPU core, IOB represents I/O bandwidth, and f represents CPU frequency. MPI, IO, and SERIAL represent the execution times of MPI parallel execution, I/O requests, and serial code in the workload, respectively. All these parts are clearly dependent on input parameters.

$$\begin{aligned} T(f, BW, NB) = &\, MPI(BW, NB) \\ &+ IO(BW, NB, IOB) \\ &+ SERIAL(f, BW, NB) \end{aligned} \tag{3.1}$$

We can further represent (3.1) using two approaches:

$$\frac{T(f, BW, NB)}{T_{\text{ref}}} = \alpha_{\text{MPI}} \left(\frac{NB_{\text{ref}}}{NB} \right) + \alpha_{\text{CPU}} \left(\frac{f_{\text{ref}}}{f} \right) + \alpha_{\text{BW}} \left(\frac{BW_{\text{ref}}}{BW(f)} \right) + \dots \tag{3.2}$$

$$\begin{aligned} T(f, BW, NB) = &\, \sigma T_{\text{kernel}}(f, BW, NB, \text{IOB}) \\ &+ T_{\text{other}} \text{ with } T_{\text{kernel}}(f, BW, NB) \\ &\approx T_c(f) + T_{\text{mem}}(BW) + T_{\text{MPI}}(NB) + T_{\text{I/O}} \end{aligned} \tag{3.3}$$

(3.2) allows rapid modeling of the target application; however, it may lack strong representational ability and requires recalculations when application code changes. On the other hand, (3.3) exhibits greater representational power, particularly in predicting the total execution time of the target. However, (3.3) requires finer-grained functional partitioning of the source code.

Generally, the computational kernels of scientific computing applications consist of the following parts:

1. **Serial LAPACK or ScaLAPACK:** LAPACK or its extension version ScaLAPACK for distributed memory MIMD parallel computers provides routines for solving linear systems of equations, linear least squares problems, eigenvalue problems, and singular value decomposition in both serial and parallel modes. It also includes a set of routines for matrix factorizations. LAPACK relies on underlying BLAS to provide efficient and portable building blocks for its routines, utilizing modern cache-based architectures and scalar processors to achieve instruction-level parallelism.

2. **BLAS Kernel (BLIS [46], MKL, etc.):** BLAS defines a set of low-level routines and architecture-oriented optimisations for performing common linear algebra operations, such as scalar arithmetic, vector operations and general matrix multiplication(GEMM). BLAS often uses uniform function names and return values to allow for straightforward implementation of special optimization versions for different platforms. BLAS-based LINPACK does not require modification of its code.
3. **FFT Kernel (FFTW3, etc.):** FFT is one of the most common computational kernels apart from linear algebra operations. Libraries like FFTW provide multiple FFT algorithms and evaluate and select the best trade-off between speed and accuracy based on the problem type and size.

The parallel runtimes that need performance modeling include:

1. **OpenMP:** OpenMP [47] is a parallel programming paradigm for SMP-UMA architectures, offering a cross-platform shared-memory multithreading API. It provides C/C++ and Fortran language bindings and can run on various operating systems such as Linux, Windows, and MacOS. OpenMP provides a high-level abstraction of parallel algorithms, allowing programmers to indicate their intent through pragmas in the source code. The compiler automatically parallelizes the program based on these annotations, entering synchronization, mutual exclusion, and communication at critical regions.
2. **OpenMPI:** OpenMPI [48] is a standardized, portable message-passing standard for NUMA architectures, designed to support virtual topology, communication, and synchronization between parallel processes running on distributed memory systems, without relying on specific language syntax.

In this study, the BLIS Benchmark, benchFFT, Intel MPI Benchmark, pmbw, and STREAM 1.0 are used to perform benchmark tests for Math Kernel, FFT Kernel, OpenMPI communication, OpenMP-based multi-core memory access, and CPU-Memory coupling, respectively, in order to obtain the required bandwidth values for Equation 3.1.

3.2 Define the Metrics

In performance engineering, the indicators used to assess performance changes introduced by code or architectural modifications are referred to as metrics. The definition and calculation methods of these metrics are the essence of the performance engineering methodology. In this section, we will introduce the POP Parallel Metrics and I/O Metrics that we have utilized.

3.2.1 Single Core Metrics

The most direct way to evaluate application performance is by recording its execution time, known as CPU time. CPU time refers to the time taken by a given program from the start of execution to its completion. It encompasses the time allocated by the CPU hardware to the target program and includes time introduced by the operating system for resource management, such as interrupt handling and scheduling. However, it does not include time spent waiting for I/O operations or time when execution is paused due to scheduling. CPU time can be expressed as the product of the total number of clock cycles required to execute all instructions of the program and the clock cycle time, or it can be calculated by dividing the total number of clock cycles by the clock frequency, as shown in Equation 3.4.

$$\begin{aligned} Time_{CPU} &= Clock\ Cycles_{total} \times Clock\ Cycle\ Time \\ &\text{or} \quad Clock\ Cycles_{total} \times Clock\ Rate \end{aligned} \tag{3.4}$$

CPU time can also be obtained by multiplying the total number of instructions a given program needs to execute with the average clock cycles per instruction (CPI) and dividing it by the clock frequency, as shown in Equations 3.5 and 3.6, respectively.

$$Time_{CPU} = \frac{Count_{instruction} \times CPI}{Clock\ Rate} \tag{3.5}$$

$$CPI = \frac{Count_{clock\ cycles}}{Count_{instructions}} \tag{3.6}$$

3.2.2 Parallel Metrics

$$Efficiency_{parallel} = \frac{sum(comp)}{n \times runtime} = \frac{average(comp)}{runtime} \tag{3.7}$$

Parallel efficiency is defined as the ratio of the total execution time spent on the computational part of parallel code to the overall execution time. It is obtained by summing up the runtime of the parallel portion of the code and dividing it by the total runtime.

$$Efficiency_{LoadBalance} = \frac{average(comp)}{max(comp)} \tag{3.8}$$

$$Efficiency_{Communication} = \frac{max(comp)}{runtime} \tag{3.9}$$

Equations 3.8 and 3.9 define load balance efficiency and communication efficiency, respectively, to determine whether the target task is evenly distributed among computational resources and to identify the proportion of parallel communication in the total execution time. These two metrics can indicate performance issues such as oversubscription, load imbalance, and abnormal lock states. The multiplication of these two metrics is equivalent to Equation 3.7, i.e., parallel efficiency.

$$Scaling_{strong} = \frac{sum(comp_{ref})}{sum(comp)} \tag{3.10}$$

$$Scaling_{weak} = \frac{average(comp_{ref})}{average(comp)} \tag{3.11}$$

Equations 3.10 and 3.11 are the most important metrics for evaluating parallel computing efficiency, which reflect whether the scalability of hardware can be translated into linear performance growth in the target application. The scaling of a computational architecture can be defined by three components: instruction scaling, IPC (Instructions Per Cycle) scaling, and frequency scaling. Instruction scaling represents whether the number of instructions can achieve uniform scaling. IPC scaling assesses whether software can achieve the same speed gain as hardware performance increases. Frequency scaling reflects the trend of target system performance with CPU frequency growth. The three metrics can be multiplied together to form the unified expression of computation scaling, as shown in Equation 3.12.

$$Scaling_{computation} = Scaling_{instruction} \times Scaling_{IPC} \times Scaling_{Frequency} \tag{3.12}$$

By combining $Efficiency_{Parallel}$ and $Scaling_{Computation}$, we obtain the global efficiency of the target software-hardware system, as shown in Equation 3.13.

$$Efficiency_{global} = Scaling_{computation} \times Efficiency_{Parallel} \tag{3.13}$$

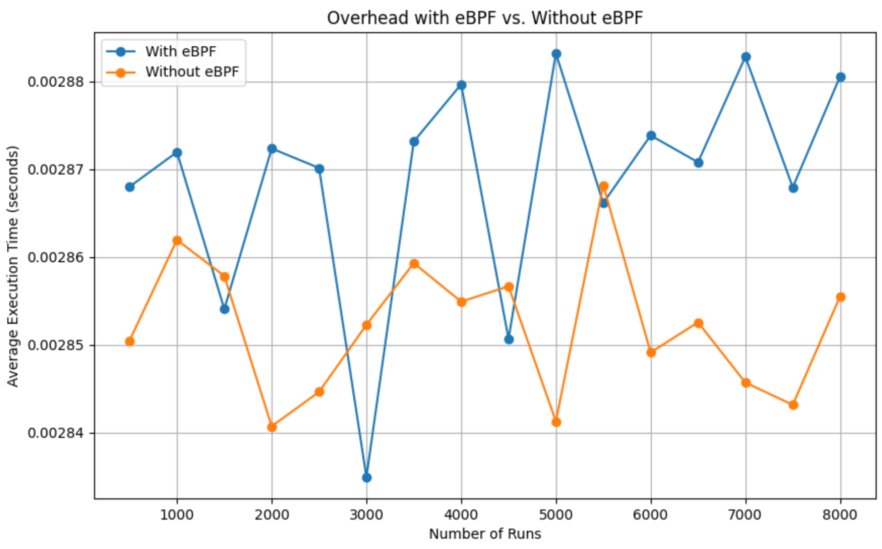
For the commonly used hybrid parallel architecture of OpenMP + OpenMPI in HPC systems, parallel efficiency can also be defined by Equation 3.14.

$$Efficiency_{parallel} = Efficiency_{MPI} \times Efficiency_{OpenMP} \tag{3.14}$$

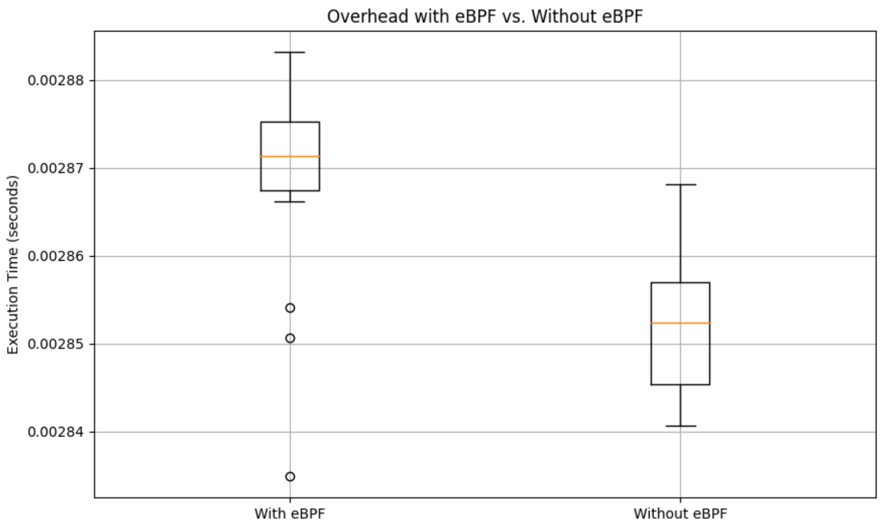
Here, MPI parallel efficiency and OpenMP parallel efficiency are defined as shown in Equations 3.15 and 3.16, respectively.

$$Efficiency_{MPI} = \frac{average(out\_MPI)}{runtime} \tag{3.15}$$

$$Efficiency_{OpenMP} = \frac{Hybrid\ Parallel\ Efficiency}{MPI\ Efficiency} \tag{3.16}$$

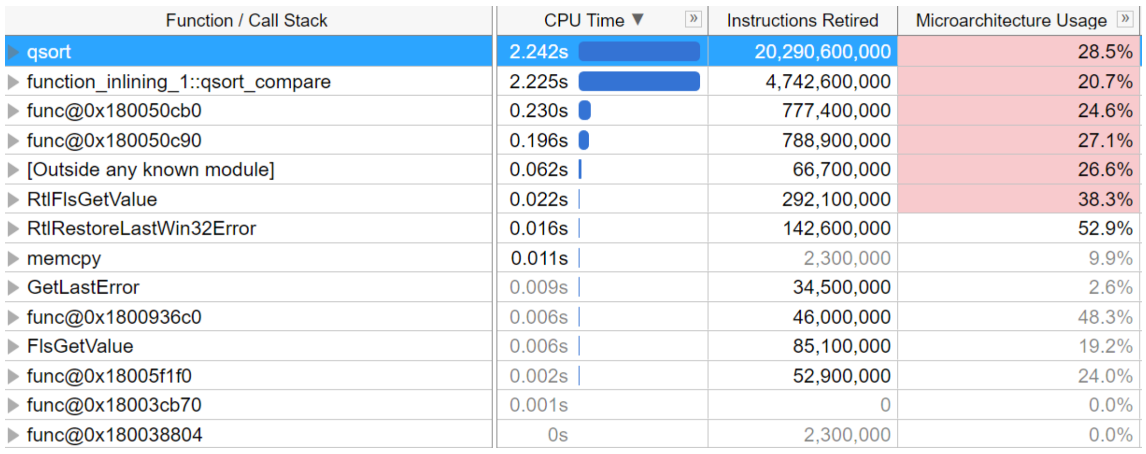


(a) Time consumed from sampling a region one time in seconds, demonstrated in average values.



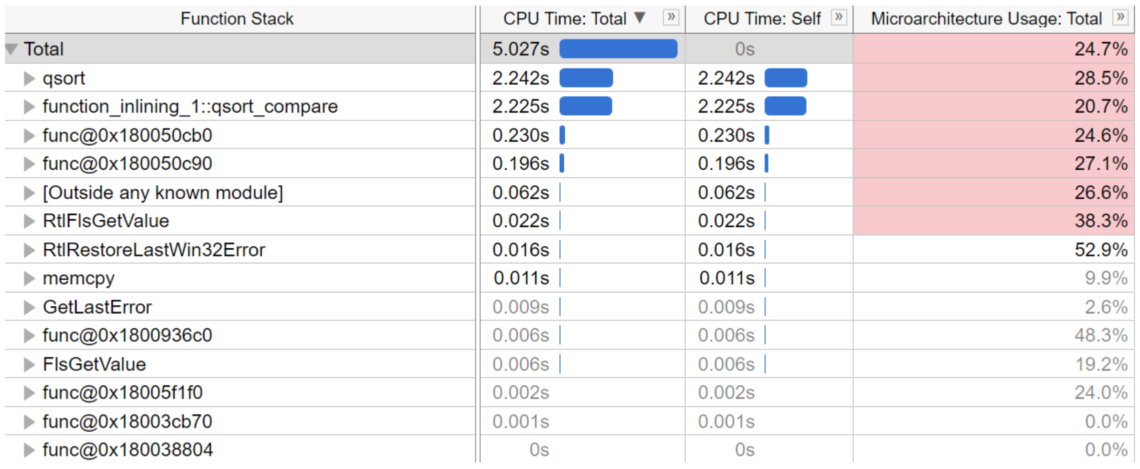
(b) Box plot showing the statistics of the sampling time of a single region.

Figure 3.1: Measuring variance of the time to single instrumentation, i.e., attach eBPF process onto the target application while varying the number of measurements.



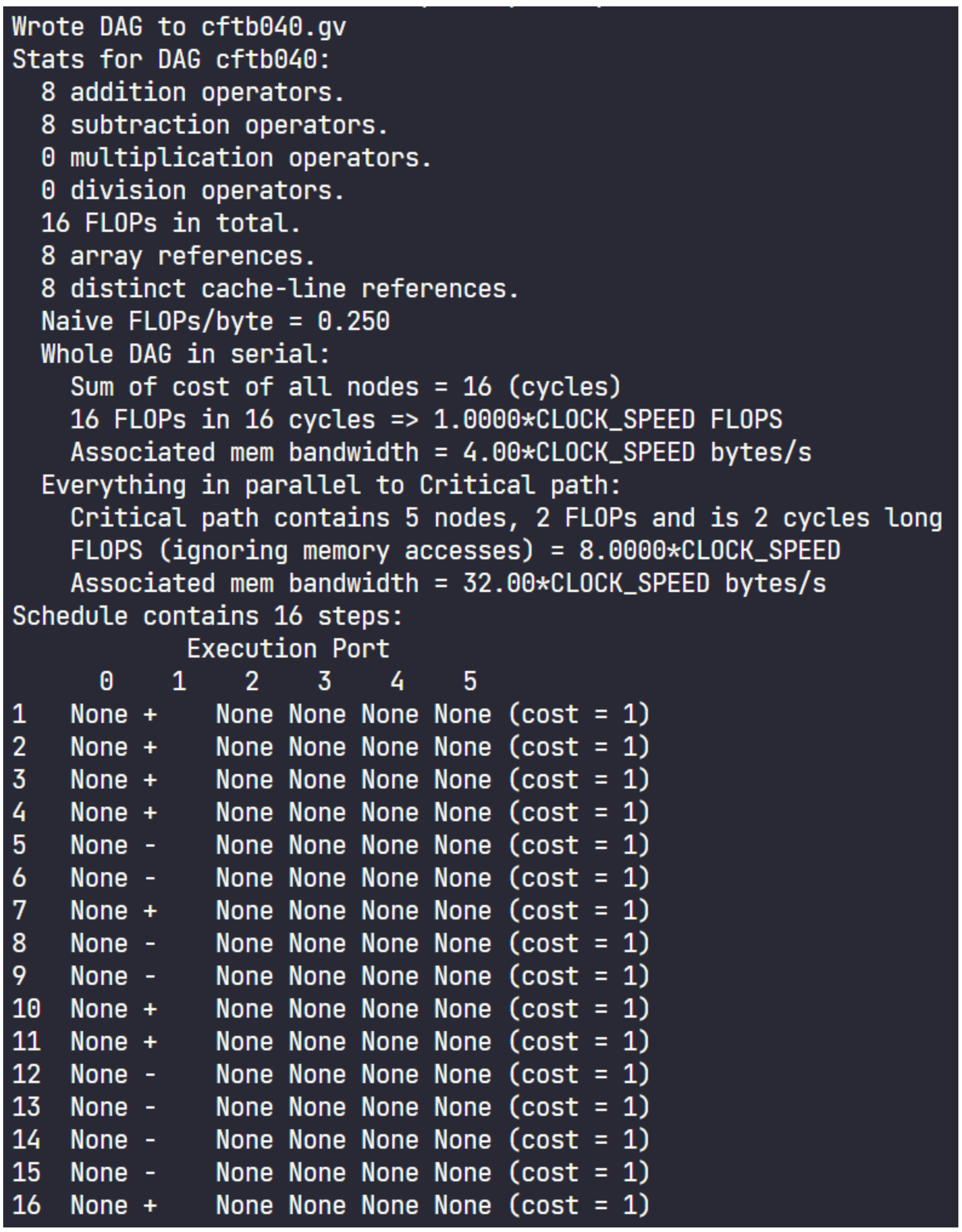
Function / Call Stack	CPU Time	Instructions Retired	Microarchitecture Usage
qsort	2.242s	20,290,600,000	28.5%
function_inlining_1::qsort_compare	2.225s	4,742,600,000	20.7%
func@0x180050cb0	0.230s	777,400,000	24.6%
func@0x180050c90	0.196s	788,900,000	27.1%
[Outside any known module]	0.062s	66,700,000	26.6%
RtlFlsGetValue	0.022s	292,100,000	38.3%
RtlRestoreLastWin32Error	0.016s	142,600,000	52.9%
memcpy	0.011s	2,300,000	9.9%
GetLastError	0.009s	34,500,000	2.6%
func@0x1800936c0	0.006s	46,000,000	48.3%
FlsGetValue	0.006s	85,100,000	19.2%
func@0x18005f1f0	0.002s	52,900,000	24.0%
func@0x18003cb70	0.001s	0	0.0%
func@0x180038804	0s	2,300,000	0.0%

Figure 3.2: Profiling through Intel VTune Profiler, shown in bottom-up.



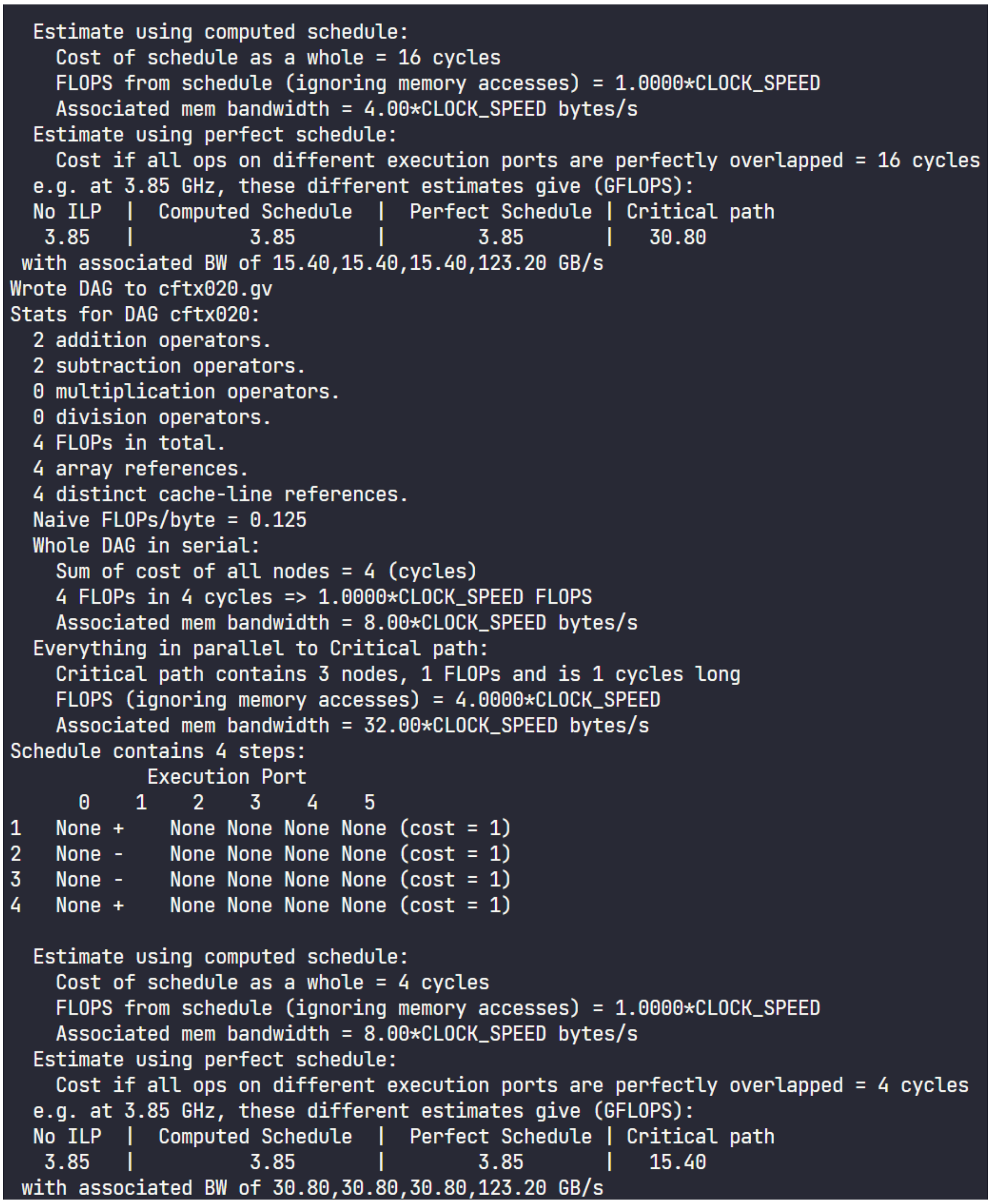
Function Stack	CPU Time: Total	CPU Time: Self	Microarchitecture Usage: Total
Total	5.027s	0s	24.7%
qsort	2.242s	2.242s	28.5%
function_inlining_1::qsort_compare	2.225s	2.225s	20.7%
func@0x180050cb0	0.230s	0.230s	24.6%
func@0x180050c90	0.196s	0.196s	27.1%
[Outside any known module]	0.062s	0.062s	26.6%
RtlFlsGetValue	0.022s	0.022s	38.3%
RtlRestoreLastWin32Error	0.016s	0.016s	52.9%
memcpy	0.011s	0.011s	9.9%
GetLastError	0.009s	0.009s	2.6%
func@0x1800936c0	0.006s	0.006s	48.3%
FlsGetValue	0.006s	0.006s	19.2%
func@0x18005f1f0	0.002s	0.002s	24.0%
func@0x18003cb70	0.001s	0.001s	0.0%
func@0x180038804	0s	0s	0.0%

Figure 3.3: Profiling through Intel VTune Profiler, shown in top-down.



```
Wrote DAG to cftb040.gv
Stats for DAG cftb040:
  8 addition operators.
  8 subtraction operators.
  0 multiplication operators.
  0 division operators.
  16 FLOPs in total.
  8 array references.
  8 distinct cache-line references.
  Naive FLOPs/byte = 0.250
  Whole DAG in serial:
    Sum of cost of all nodes = 16 (cycles)
    16 FLOPs in 16 cycles => 1.0000*CLOCK_SPEED FLOPS
    Associated mem bandwidth = 4.00*CLOCK_SPEED bytes/s
  Everything in parallel to Critical path:
    Critical path contains 5 nodes, 2 FLOPs and is 2 cycles long
    FLOPS (ignoring memory accesses) = 8.0000*CLOCK_SPEED
    Associated mem bandwidth = 32.00*CLOCK_SPEED bytes/s
Schedule contains 16 steps:
            Execution Port
      0    1    2    3    4    5
1   None +    None None None None (cost = 1)
2   None +    None None None None (cost = 1)
3   None +    None None None None (cost = 1)
4   None +    None None None None (cost = 1)
5   None -    None None None None (cost = 1)
6   None -    None None None None (cost = 1)
7   None +    None None None None (cost = 1)
8   None -    None None None None (cost = 1)
9   None -    None None None None (cost = 1)
10  None +    None None None None (cost = 1)
11  None +    None None None None (cost = 1)
12  None -    None None None None (cost = 1)
13  None -    None None None None (cost = 1)
14  None -    None None None None (cost = 1)
15  None -    None None None None (cost = 1)
16  None +    None None None None (cost = 1)
```

Figure 3.4: FFT kernel performance abstract generated by habakkuk.



```
  Estimate using computed schedule:
    Cost of schedule as a whole = 16 cycles
    FLOPS from schedule (ignoring memory accesses) = 1.0000*CLOCK_SPEED
    Associated mem bandwidth = 4.00*CLOCK_SPEED bytes/s
  Estimate using perfect schedule:
    Cost if all ops on different execution ports are perfectly overlapped = 16 cycles
  e.g. at 3.85 GHz, these different estimates give (GFLOPS):
  No ILP  |  Computed Schedule  |  Perfect Schedule | Critical path
   3.85   |          3.85       |        3.85       |   30.80
 with associated BW of 15.40,15.40,15.40,123.20 GB/s
Wrote DAG to cftx020.gv
Stats for DAG cftx020:
  2 addition operators.
  2 subtraction operators.
  0 multiplication operators.
  0 division operators.
  4 FLOPs in total.
  4 array references.
  4 distinct cache-line references.
  Naive FLOPs/byte = 0.125
  Whole DAG in serial:
    Sum of cost of all nodes = 4 (cycles)
    4 FLOPs in 4 cycles => 1.0000*CLOCK_SPEED FLOPS
    Associated mem bandwidth = 8.00*CLOCK_SPEED bytes/s
  Everything in parallel to Critical path:
    Critical path contains 3 nodes, 1 FLOPs and is 1 cycles long
    FLOPS (ignoring memory accesses) = 4.0000*CLOCK_SPEED
    Associated mem bandwidth = 32.00*CLOCK_SPEED bytes/s
Schedule contains 4 steps:
            Execution Port
      0    1    2    3    4    5
1   None +    None None None None (cost = 1)
2   None -    None None None None (cost = 1)
3   None -    None None None None (cost = 1)
4   None +    None None None None (cost = 1)

  Estimate using computed schedule:
    Cost of schedule as a whole = 4 cycles
    FLOPS from schedule (ignoring memory accesses) = 1.0000*CLOCK_SPEED
    Associated mem bandwidth = 8.00*CLOCK_SPEED bytes/s
  Estimate using perfect schedule:
    Cost if all ops on different execution ports are perfectly overlapped = 4 cycles
  e.g. at 3.85 GHz, these different estimates give (GFLOPS):
  No ILP  |  Computed Schedule  |  Perfect Schedule | Critical path
   3.85   |          3.85       |        3.85       |   15.40
 with associated BW of 30.80,30.80,30.80,123.20 GB/s
```

Figure 3.5: FFT estimated core scheduling and theoretic performance.

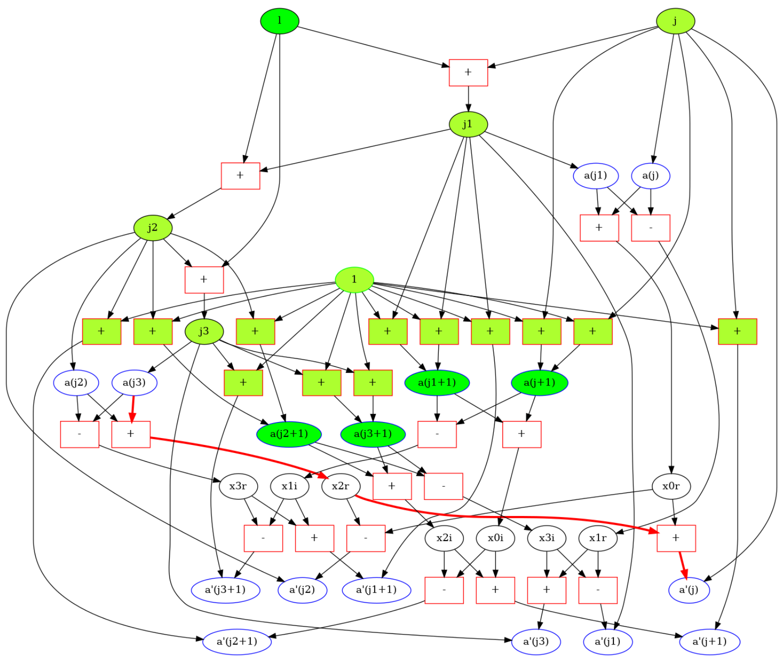
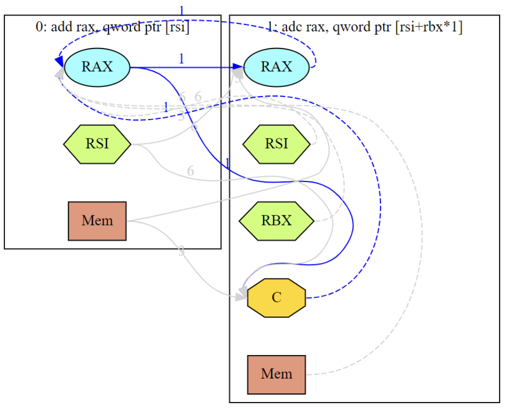


Figure 3.6: Dependency plot on one of the fft components, generated by habakkuk.



0: add rax, qword ptr [rsi]
1: adc rax, qword ptr [rsi+rbx*1]
1
RAX
1
RAX
6
1
RSI
RSI
6
1
Mem
RBX
C
Mem

Figure 3.7: Dependency chain analyzed by UiCA, first part.

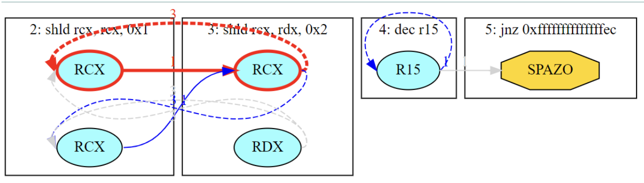


Figure 3.8: Dependency chain analyzed by UiCA, second part.

Chapter 4

Evaluate the Implementation

In this chapter, we summarized the outcomes of our performance experiments. This included the type of computation and input/output loads, the effect of parallel parameters on the speed of calculation, and any potential memory limitations. Furthermore, we compared different probing techniques to reduce the uncertainty caused by overhead.

4.1 Define the Experiment

4.1.1 Target System

The top-down and computational load analysis experiments were carried out on a high-performance computing cluster featuring the AMD EPYC Zen 2 processor. Each node in the cluster is equipped with two sockets, forming a NUMA (Non-Uniform Memory Access) domain comprising eight nodes. Each socket contains 64 physical cores, with SMP enabling two threads per physical core. This configuration results in a total of 128 threads per socket.

For the GPU experiments, we utilized a separate computing cluster equipped with Intel Xeon Ice Lake processors and Nvidia A100 40G graphics cards. Each node in this cluster houses two CPU sockets and two graphics cards. The Intel Xeon Ice Lake processors provide powerful computational capabilities, while the Nvidia A100 40G graphics cards offer high-performance parallel processing capabilities.

Detailed specifications of the CPU and GPU architectures used in the experiments are provided in Table 4.1 for the AMD EPYC Zen 2 processor and Table 4.2 for the Intel Xeon Ice Lake processor and Nvidia A100 40G graphics cards. These specifications offer valuable insights into the hardware configurations employed in our performance analysis experiments.

In this study, we conducted performance tests using Quantum Espresso version 6.7, which was compiled with the Intel OneAPI compiler suite. The hardware used for the experiments encompassed diverse systems to ensure a comprehensive analysis. The support libraries employed in our

CPU Arch	processor	clock(MHz)	# sockets	cores/socket
X86_64	AMD EPYC 7H12	3312	2	64
X86_64	Intel Xeon Gold 5320	2797	2	26

Table 4.1: CPU Used in This Study

Architecture	GPU	clock(MHz)	memory	memory bandwidth
Ampere	NVIDIA A100	765	40G	1555 GB/s
Ampere	NVIDIA A40	1305	48G	695.8 GB/s

Table 4.2: GPU Used in This Study

test environment included Intel MPI version 2021.1.1, CUDA version 11.3, Intel MKL version 2021.1.1, Intel TBB version 2021.1.1, and Intel Compiler-rt version 2021.1.1. The remaining support libraries were maintained in line with the official default configuration of Quantum Espresso.

To assess the performance impact of AOCL on AMD CPUs, we utilized a specific configuration. For this purpose, we employed the QE6.7 version hosted by Spack, which was compiled using FFTW version 3.3.10 and OpenBLAS version 0.3.21 as support libraries. We incorporated AOCC version 3.0.0 and flang corresponding to it via make and cmake. Throughout the installation process, we adhered to the default Spack configuration, ensuring consistent and standardized settings.

It is important to note that the CPU and GPU versions were distinguished primarily by the inclusion or exclusion of the NVHPC SDK and CUDA support. This approach allowed us to effectively gauge the performance disparities and draw meaningful comparisons between the two versions across different hardware configurations.

4.1.2 Workload Properties

In this study, the AUSURF112 configuration under the UEABS benchmark was chosen as the subject of investigation.

The AUSURF112 structure comprises a total of 112 atoms. Ab-initio calculations of the electronic structure were performed using the density functional theory (DFT) with the Perdew-Burke-Ernzerhof (PBE) exchange-correlation pseudopotential and the generalized gradient approximation PBE functional. A plane wave basis was utilized with a cut-off energy set to 200 Ry. The Marzari-Vanderbilt smearing method with a degauss value of 0.05 was used for the occupation method. The diagonalization method chosen for the calculations was Davidson, and 2 k-points were employed.

The complexity of ab-initio electronic structure calculations lies in the convergence of wave functions within each time step during DFT calculations. The Quantum Espresso (QE) software adopts the plane wave self-consistent field method (PWscf) to describe electronic states using the self-consistent Kohn-Sham equation. Additionally, QE employs the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm for structure optimization. The self-consistent field (SCF) calculations involve iterative evaluations of the electronic density to determine convergence by assessing the difference between input and output densities within a predefined threshold. Each iteration is referred to as a time step. The core load of SCF calculation involves diagonalization of the Hamiltonian and FFT (Fast Fourier Transform) calculations for converting charge density and potential energy between real and reciprocal domains. Both tasks pose challenges for high-performance computing systems. QE provides multiple parallel levels to handle various computing tasks, where k-points can be evenly distributed among different Pool levels, with each Pool processing a fraction of the k-points. However, different parallel levels exhibit varying scalability and memory distribution strategies, necessitating dynamic adjustment of parallel parameters based on the computing environment and structure for optimal performance. Additionally, the convergence iteration process of wave functions is influenced by factors such as initial trial wave function, floating-point and sparse matrix operations, dense node intra/inter-node communication, and the specific density functional used.

The key computational kernels in QE have been highly parallelized, although significant variations in parallelism exist among different parallel levels. The Image, Pool, and PW levels demonstrate linear to close-to-linear CPU scaling. However, the Task level, responsible for FFT calculations of electronic states, and the Linear Algebra (LA) level, dealing with subspace Hamiltonians and constraint matrices, exhibit suboptimal CPU scaling. This phenomenon can be attributed to Amdahl's law and Gustafson's law, where non-parallelized code within the kernel becomes a bottleneck for parallel scalability.

4.2 Before the Experiment

Before conducting performance experiments, it is essential to set initial I/O parameters and parallel parameters at different levels of parallelism reasonably. Previous research has indicated that achieving ideal parallelism involves setting the parameter "nimage" to its maximum possible value and multiplying the number of OpenMP threads by the MPI rank to obtain the total number of CPU logical cores. Consequently, when increasing the number of OpenMPI ranks, the number of OpenMP threads per rank decreases proportionally to maintain this constraint. Based on the hardware configuration of KAGAYAKI, we have determined a series of initial parallel parameters, which are detailed in Table 4.3.

nodes	ranks	threads	Placement
1	64	1	default, round robin
1	32	2	default, round robin
1	16	4	default, round robin
2	128	1	default, round robin
2	64	2	default, round robin
2	32	4	default, round robin
4	256	1	default, round robin
4	128	2	default, round robin
4	64	4	default, round robin
8	512	1	default, round robin
8	256	2	default, round robin
8	128	4	default, round robin

Table 4.3: Process Placement and Affinity Settings

To analyze process placement, we utilize various tools, including NUMACTL, taskset, vmtouch, and sysbench. OpenMPI's Process Placement consists of three components: mapping, ranking, and binding. Mapping involves determining the relationship between computational loads and computational resources, such as mapping a computational instance to a CPU socket, a NUMA domain, or a node. Ranking involves ranking computation loads according to specific computation resources, allowing instances with intensive communication to be distributed near adjacent computational resources. Binding refers to the binding relationship between computational loads and computational resources during actual computation. Incorrect process placement can lead to performance issues such as oversubscription and overload, resulting in an imbalance in computational load distribution and resource utilization. OpenMPI supports process binding at different levels, such as core, socket, and NUMA. In this study, we use the term "default" to refer to OpenMPI's default behavior for process placement.

4.3 Evaluation Procedures

4.3.1 Runtime Profiling

Our investigation revealed that the overhead of SCF calculations across nodes is significantly higher compared to that across sockets. To achieve near-ideal scalability with same-node scaling, we bound OpenMPI to physical CPU cores and disabled OpenMP. The linear acceleration ratio achieved through MPI binding to adjacent cores may benefit from shared caches or NUMA domains. Figure 4.1 and 4.2 present specific experimental data when increasing only the number of MPI ranks and using a combination of OpenMP threads and OpenMPI ranks, respectively.

We observed that binding OpenMP threads to CPU sockets and mixing them with OpenMPI ranks leads to higher acceleration ratios compared to using OpenMPI alone. Even without setting affinity, increasing OpenMP threads on the same node achieves a 1.2x acceleration ratio. This indicates that combining OpenMP and OpenMPI better utilizes the peak performance of all processor cores. However, it is essential to consider the problem size when selecting the number of computing hardware, as excessive computing resources can lead to increased computation time, particularly when crossing nodes. In such cases, I/O and communication overheads across nodes can become the primary bottleneck, offsetting the acceleration effect brought by parallelism. Consequently, choosing the right number of computing hardware is crucial to achieving efficient computing performance.

Scope	CPU TIME (sec) (I)	CPU TIME (sec) (E)
run_pwscf_	3.30e+04 (97.9%)	1.60e-02 (0.0%)
electrons_	2.86e+04 (84.8%)	2.52e-02 (0.0%)
electrons_scf_	2.86e+04 (84.8%)	2.66e-01 (0.0%)
c_bands_	2.59e+04 (76.7%)	1.46e-02 (0.0%)
diag_bands_	2.58e+04 (76.6%)	1.59e-02 (0.0%)
pcegterg_	2.55e+04 (75.8%)	1.03e+01 (0.0%)
h_psi_	9.03e+03 (26.8%)	5.57e-02 (0.0%)
h_psi__	9.03e+03 (26.8%)	5.03e+01 (0.1%)

Table 4.4: Hotspot Processes in PWSCF

Our runtime tracing results revealed that the hot functions in the SCF calculation mainly come from the GEMM function in MKL and the communication function in Intel MPI. Further details of the hotspots can be found in Tables 4.4 and 4.5.

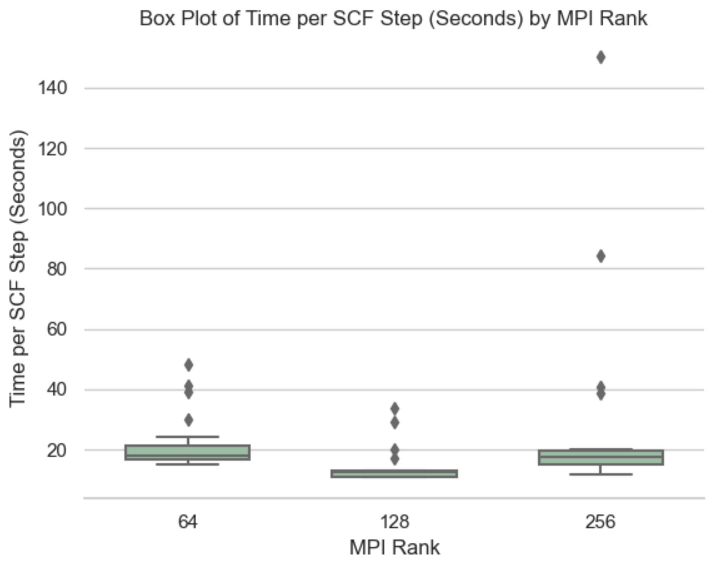


Figure 4.1: Time per SCF step / MPI ranks, with each MPI rank bound to a single physical core.

The "Scope" in the table represents the function call stack obtained by sampling the target program using eBPF, sorted in descending order based on the percentage of CPU time it occupies in the total execution time. "CPU Time (I)" and "CPU Time (E)" respectively indicate the total execution time of each function during the sampling period and the single execution time of the function. The total runtime of a function is obtained by multiplying the average single execution time by the number of executions during the sampling period. From the data, it can be observed that the Z-General Matrix Multiplication (ZGEMM) is the computational kernel of this workload, and "electrons" represent the main process of this workload. Additionally, there are other computation flows involving LAPACK and FFT kernels, such as "*diag_bands*" and "*h_psi*," but these computation flows do not become bottlenecks of this workload. In other words, according to Amdahl's Law, optimizing these flows would not significantly improve the overall performance. Apart from the CPU computation part, we also notice from the profiling that a substantial portion of execution time is occupied by MPI collective operations and Infiniband communication, indicating the significance of optimizing the OpenMPI communication process.

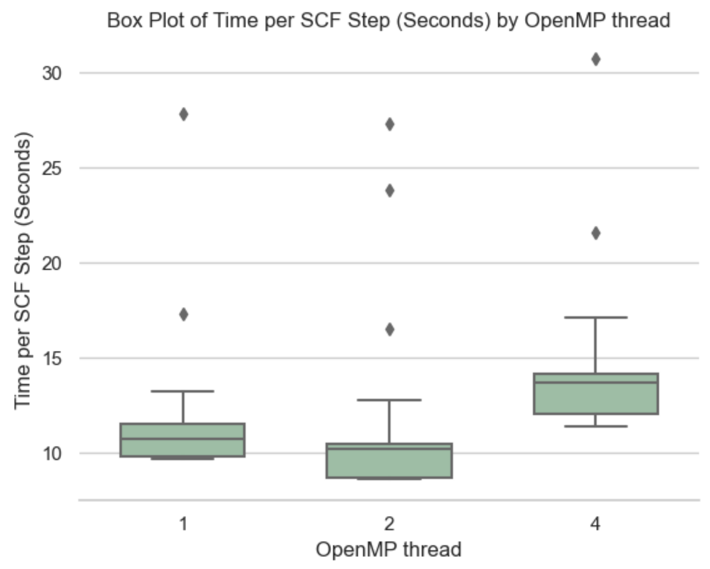


Figure 4.2: Time per SCF step / OpenMP thread, with physical cores fixed to 256, and OpenMPI rank = 256 / OpenMP thread.

4.3.2 Kernel Benchmark

We conducted a performance comparison of linear algebra operations using BLIS 0.9.0, MKL 2022.2.0, OpenBLAS zenp-r0.3.21.a, and Eigen 3.4.0 on the Zen2 architecture. As the most commonly used operation type in SCF calculations is ZGEMM, we evaluated the single-core, 64-core, and 128-core performance of ZGEMM operations across different linear algebra libraries on the Zen2 architecture. Detailed measurement data can be found in Graph 4.3, 4.4, and 4.5.

The results demonstrate that in the single-core scenario, OpenBLAS and BLIS exhibit approximately 1.25x performance improvement compared to MKL on the Zen2 architecture. In the 64-core scenario, MKL performs exceptionally well for extremely small tasks; however, as the task size increases, MKL encounters scalability issues, leading to peak performance at $m = n = k = 2000$, approximately 25 GFLOPS/core. On the other hand, BLIS performs admirably in the 64-core scenario, showcasing both excellent performance and scalability. For $m = n = k > 3000$, BLIS's performance surpasses MKL by about 1.25x.

In the 128-core scenario, all linear algebra libraries face scalability challenges to varying degrees during ZGEMM operations, and the overhead caused by cross-socket communication may be a contributing factor. It

Scope	CPU TIME (sec) (I)	CPU TIME (sec) (E)
zgemm	1.58e+04 (46.8%)	7.87e-01 (0.0%)
zgemm	1.58e+04 (46.8%)	2.39e-01 (0.0%)
avx2_xzgemm	1.57e+04 (46.7%)	8.19e-01 (0.0%)
avx2_z_generic_fullacopybcopy	1.57e+04 (46.7%)	1.32e+01 (0.0%)
avx2_zgemm_ker0	1.48e+04 (43.8%)	1.40e+00 (0.0%)
avx2_zgemm_kernel_0	1.32e+04 (39.3%)	1.32e+04 (39.3%)
MPIDI_coll_select	1.12e+04 (33.3%)	1.20e+00 (0.0%)
MPIDI_coll_invoke	1.12e+04 (33.3%)	1.29e+00 (0.0%)
MPIDI_OFI_progress	9..97e+03 (29.6%)	2.33e+02 (0.7%)

Table 4.5: Hotspot Functions in PWSCF

is essential to note that since $m = n = k < 1500$ rarely occurs in QE calculations, we conclude that BLIS is the most suitable choice for QE.

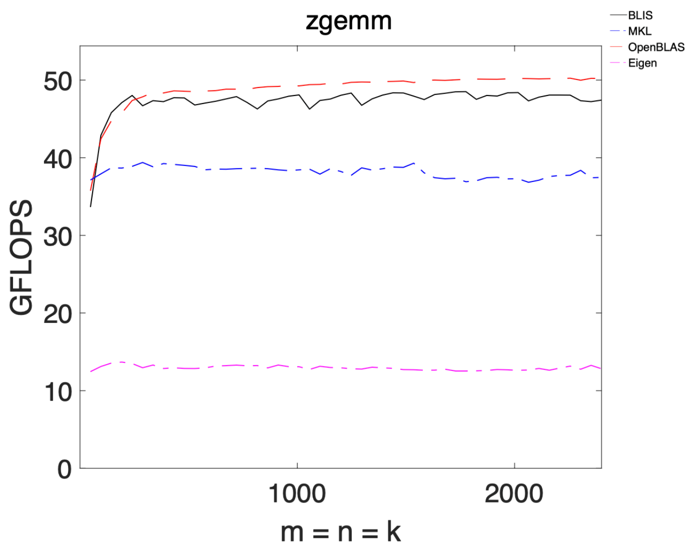


Figure 4.3: ZGEMM Benchmark on Single Core.

In addition to conducting benchmark tests on linear algebra computational kernels, we also performed benchmark tests on FFT operations using benchFFT. Experiment data is demonstrated in Figure 4.6.

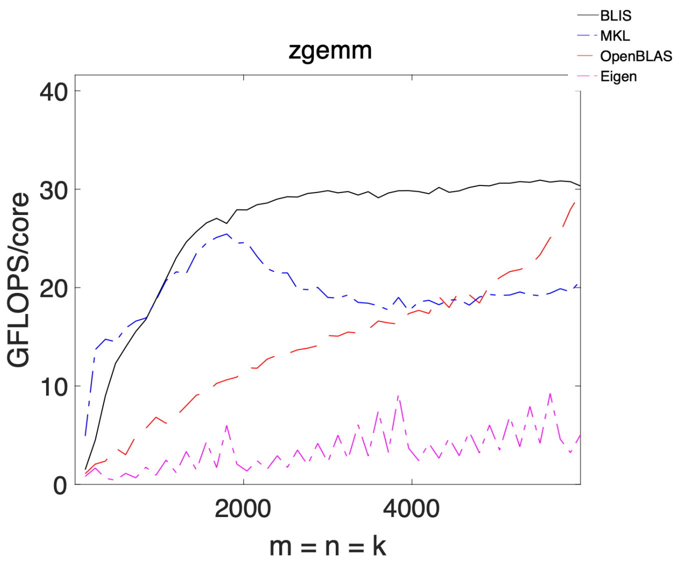


Figure 4.4: ZGEMM Benchmark on 64 Cores.

4.4 I/O and Memory Benchmark

Understanding the memory architecture of the target system is crucial for computing performance and scalability. To assess both single-threaded and multi-threaded memory bandwidth of the target machine, we employed the pmbw memory benchmarking framework. This tool offers performance testing capabilities for 64-bit and 128-bit standard loops and unrolled loops, along with a multi-threading interface. We selected representative memory bandwidth results, specifically, the access speed test outcomes. Benchmark results are listed in Figure 4.7, Figure 4.8, Figure 4.9, Figure 4.10, Figure 4.11 and Figure 4.12.

IO performance is another crucial factor that impacts computing performance. To evaluate the performance of input/output operations, we utilized the Intel MPI Benchmark and conducted STREAM analysis to assess the I/O performance and communication performance within the context of OpenMPI and OpenMP. MPI experiment data is listed in Table 4.6, Table 4.7 and Table 4.8. Multi-core and OpenMP scaling are listed in Figure 4.13 and Figure 4.14.

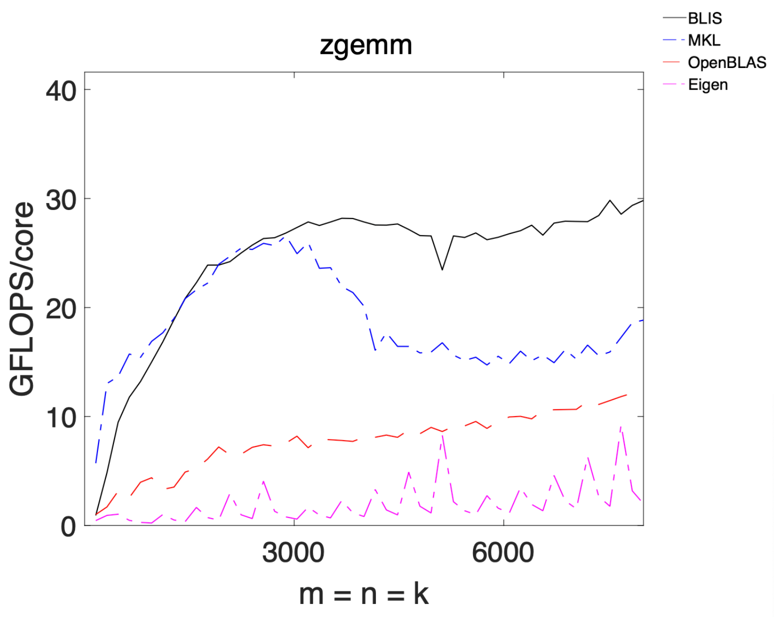


Figure 4.5: ZGEMM Benchmark on 128 Cores.

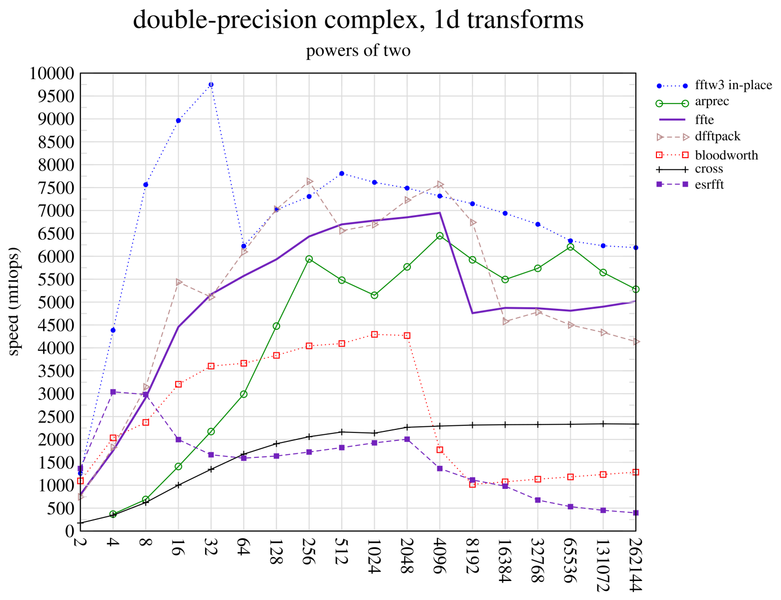


Figure 4.6: FFT Benchmark on hybrid FFT libraries.

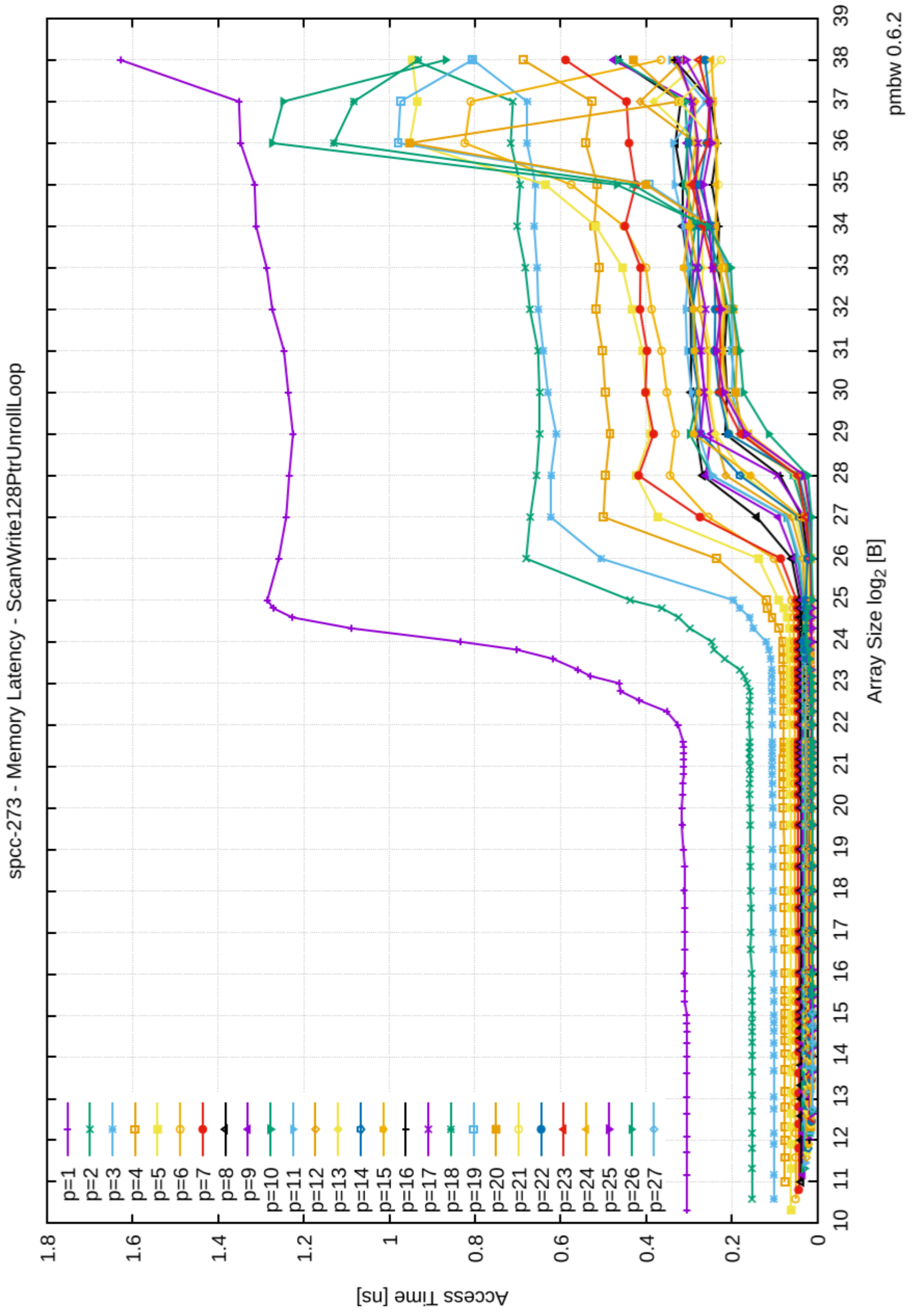


Figure 4.7: Multithreading memory latency, on Scan Write 128 bit pointers on unrolled loop.

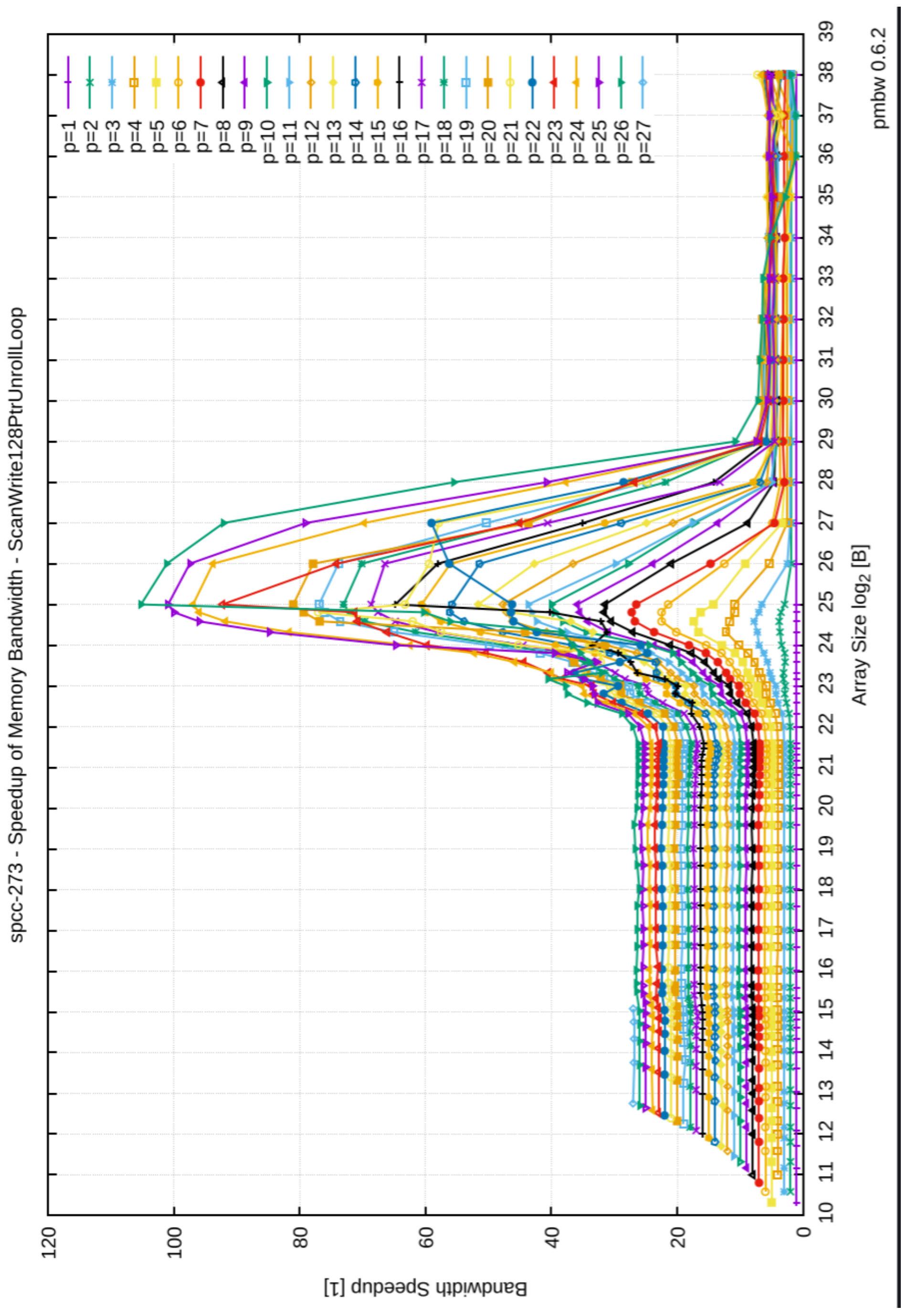


Figure 4.8: Speedup of memory bandwidth, on Scan Write 128 bit pointers on unrolled loop.

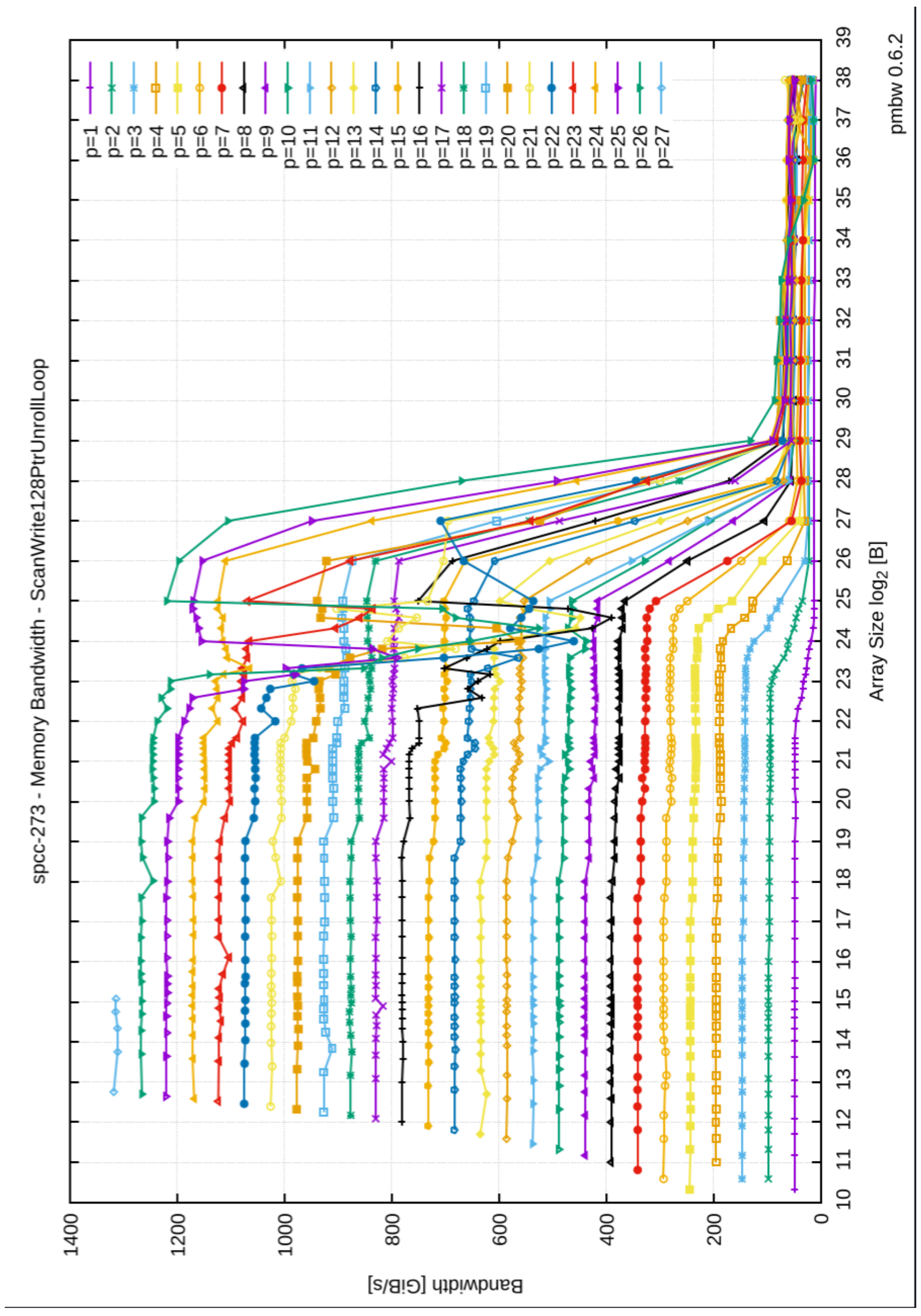


Figure 4.9: Multithreading memory bandwidth, on Scan Write 128 bit pointers on unrolled loop.

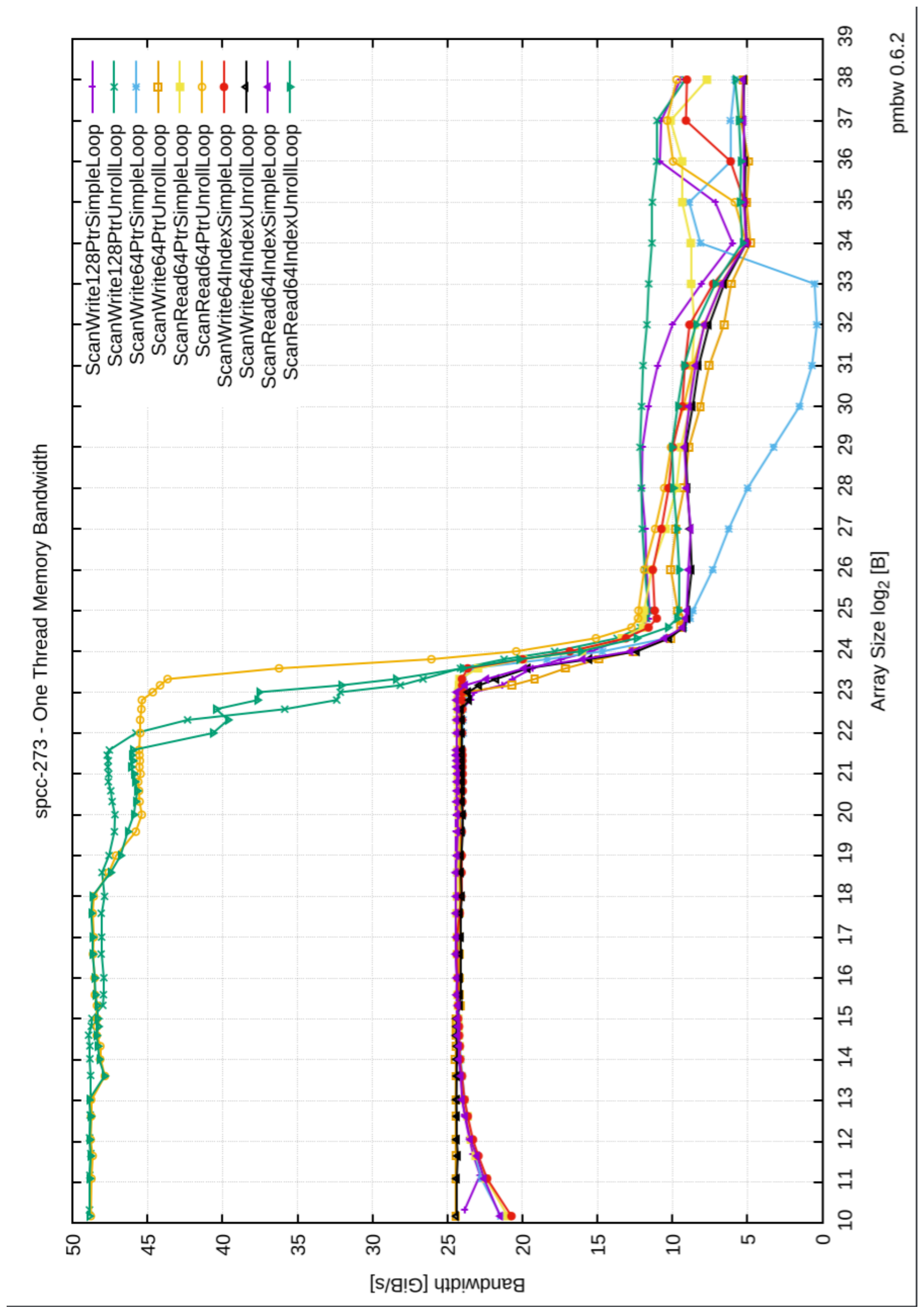


Figure 4.10: Single thread memory bandwidth, on hybrid cases.

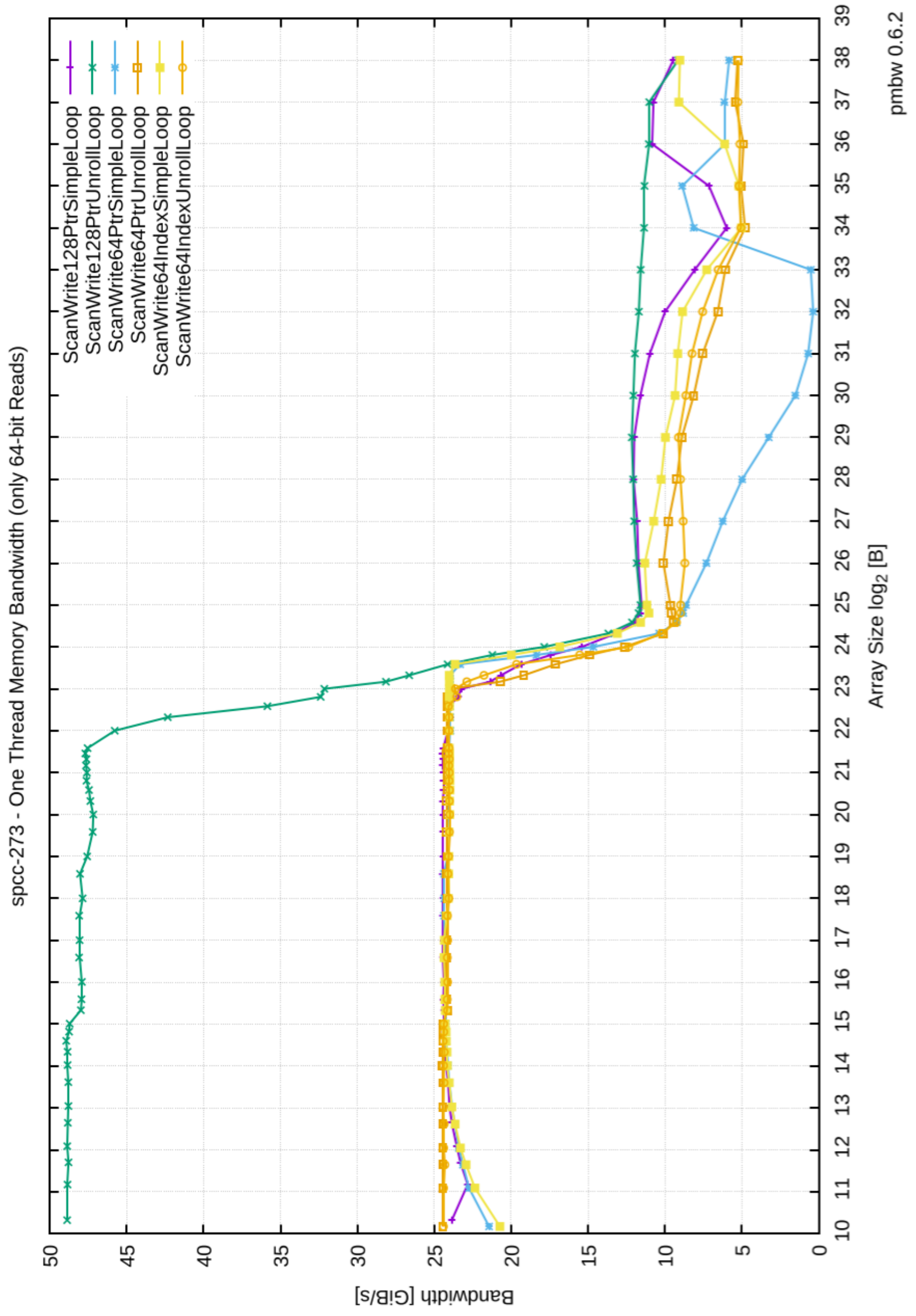


Figure 4.11: Single thread memory bandwidth, on 64bit read cases.

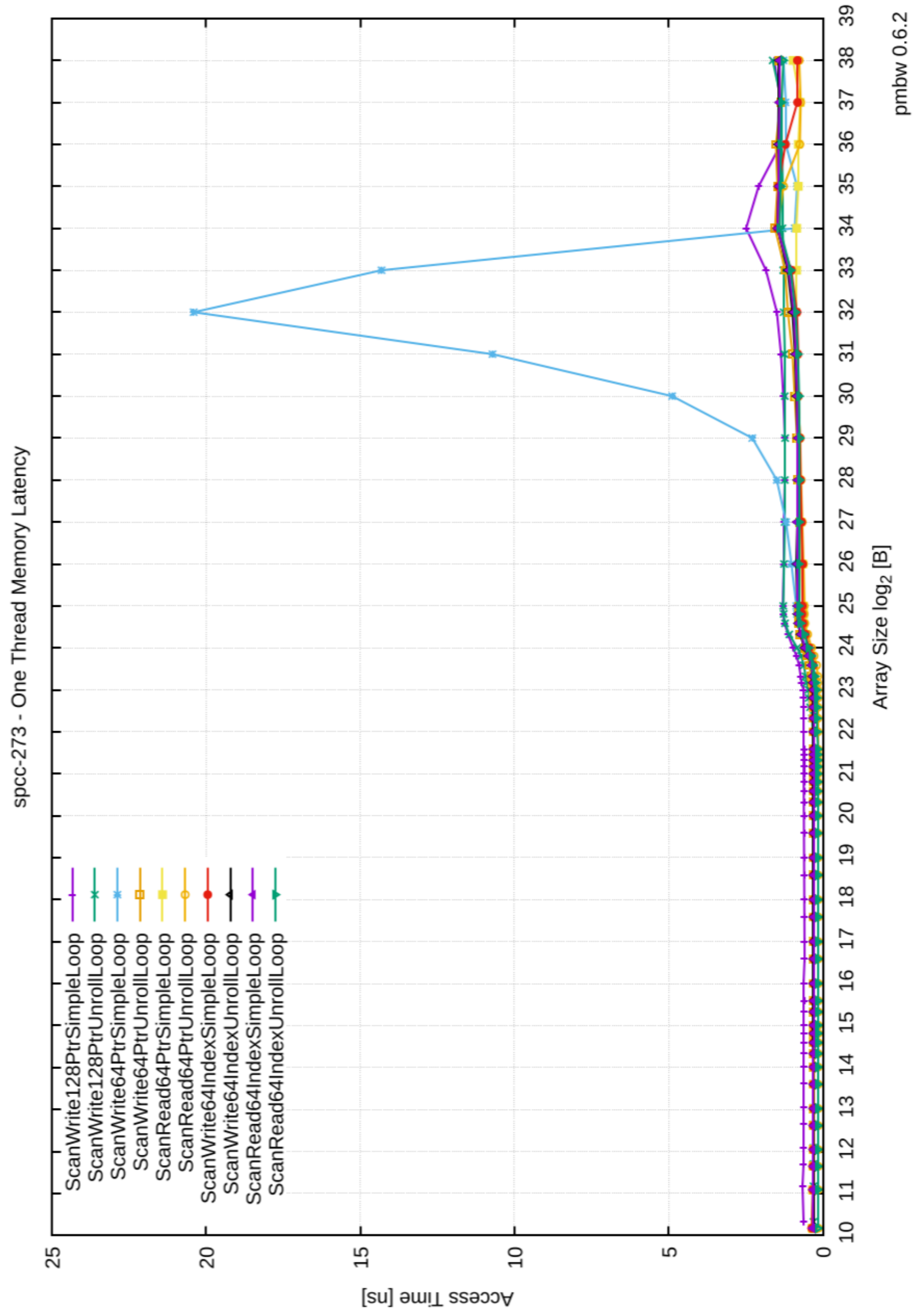


Figure 4.12: Single thread memory latency, on hybrid cases.

#bytes	#repetitions	t_min[usec]	t_max[usec]	t_avg[usec]	Mbytes/sec
0	1000	2.97	3.24	3.13	0.00
1	1000	2.67	2.86	2.77	0.70
2	1000	2.66	2.88	2.77	1.39
4	1000	2.68	2.88	2.77	2.77
8	1000	2.66	2.94	2.79	5.44
16	1000	2.62	2.91	2.79	10.98
32	1000	2.65	2.95	2.78	21.67
64	1000	3.40	3.79	3.58	33.75
128	1000	2.96	3.31	3.12	77.42
256	1000	3.33	3.65	3.47	140.25
512	1000	3.99	4.19	4.09	244.36
1024	1000	5.99	6.23	6.10	328.99
2048	1000	11.29	11.47	11.39	357.14
4096	1000	22.02	22.12	22.08	370.30
8192	1000	42.97	43.48	43.32	376.78
16384	1000	85.51	86.42	86.17	379.16
32768	1000	170.99	172.62	172.19	379.65
65536	640	276.07	350.46	331.48	373.99
131072	320	681.35	688.02	686.36	381.01
262144	160	1362.07	1372.33	1369.50	382.04
524288	80	2692.39	2739.75	2725.52	382.73
1048576	40	5327.38	5473.76	5433.15	383.13
2097152	20	10704.04	10941.79	10872.46	383.33
4194304	10	21521.24	21878.73	21770.60	383.41

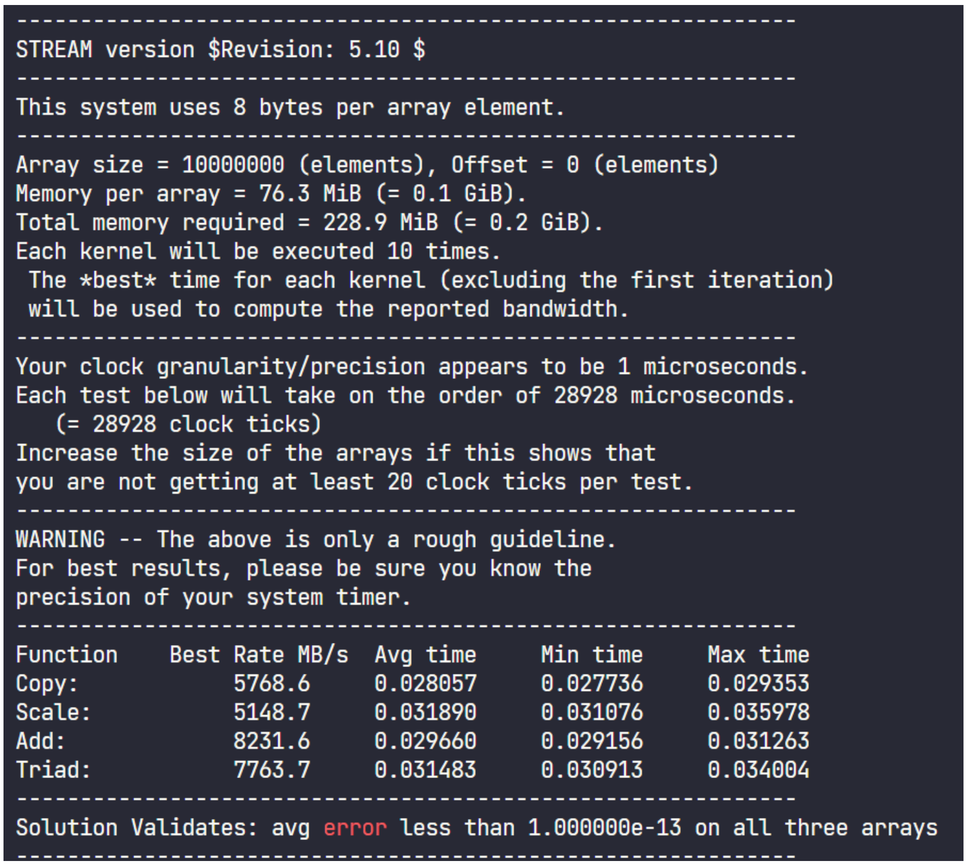
Table 4.6: MPI Benchmarking on Sendrecv

#bytes	#repetitions	t_min[usec]	t_max[usec]	t_avg[usec]	Mbytes/sec
0	1000	4.40	4.59	4.52	0.00
1	1000	4.56	4.74	4.68	0.84
2	1000	4.16	4.36	4.29	1.84
4	1000	3.95	4.11	4.06	3.89
8	1000	3.96	4.14	4.08	7.73
16	1000	4.02	4.22	4.14	15.18
32	1000	3.92	4.11	4.04	31.14
64	1000	4.30	4.50	4.43	56.85
128	1000	4.38	4.60	4.52	111.34
256	1000	4.95	5.13	5.04	199.44
512	1000	7.25	7.40	7.34	276.85
1024	1000	11.98	12.13	12.08	337.71
2048	1000	22.25	22.53	22.43	363.54
4096	1000	43.67	44.13	44.00	371.28
8192	1000	85.60	86.91	86.48	377.02
16384	1000	171.79	172.88	172.51	379.09
32768	1000	367.88	370.07	369.29	354.18
65536	640	613.15	714.49	689.12	366.89
131072	320	1517.03	1527.82	1524.76	343.16
262144	160	2960.96	3003.43	2992.04	349.13
524288	80	5664.69	5738.52	5718.17	365.45
1048576	40	11032.76	11178.56	11147.51	375.21
2097152	20	21946.83	22123.74	22070.19	379.17
4194304	10	43701.33	44116.94	43963.14	380.29

Table 4.7: MPI Benchmarking on Exchange

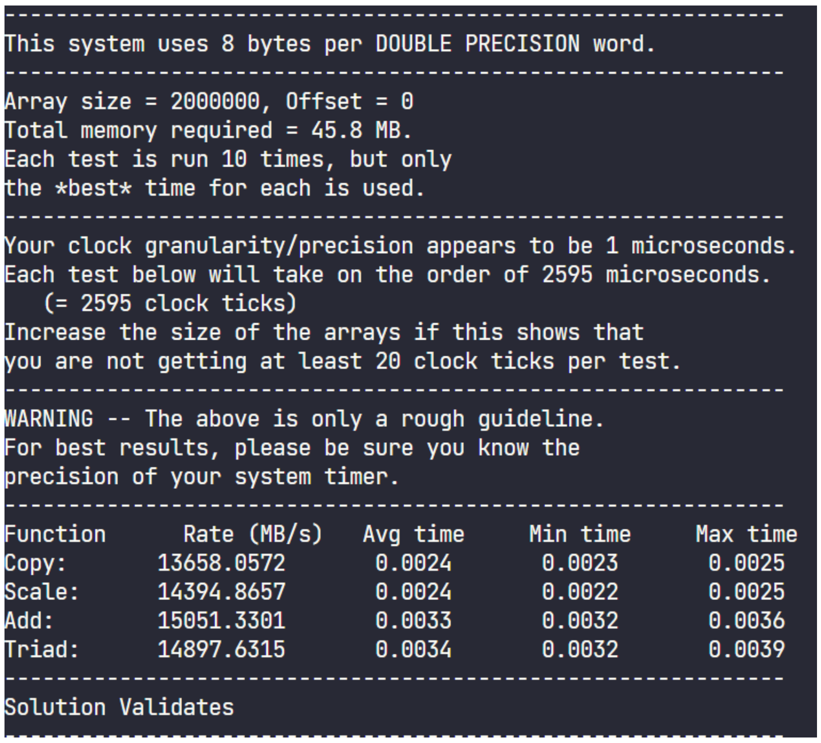
#bytes	#repetitions	t_min[usec]	t_max[usec]	t_avg[usec]
0	1000	0.04	0.05	0.04
4	1000	26.17	33.15	30.02
8	1000	25.60	33.55	30.03
16	1000	25.23	33.63	30.08
32	1000	26.18	34.01	30.77
64	1000	27.92	35.31	32.10
128	1000	27.79	35.12	31.74
256	1000	31.94	41.17	36.76
512	1000	38.00	47.37	42.44
1024	1000	57.05	70.38	63.88
2048	1000	98.09	123.16	109.51
4096	1000	172.10	242.95	207.84
8192	1000	333.30	473.40	412.05
16384	1000	662.14	940.50	820.85
32768	1000	1332.34	1881.41	1655.16
65536	640	2815.87	3816.69	3448.29
131072	320	6010.13	8011.72	7167.84
262144	160	11918.42	16189.37	14446.38
524288	80	23560.10	32227.11	28848.99
1048576	40	46580.94	64124.03	57433.16
2097152	20	95764.18	124618.57	114364.00
4194304	10	186929.60	242391.55	223517.80

Table 4.8: MPI Benchmarking on Allreduce



```
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 10000000 (elements), Offset = 0 (elements)
Memory per array = 76.3 MiB (= 0.1 GiB).
Total memory required = 228.9 MiB (= 0.2 GiB).
Each kernel will be executed 10 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 28928 microseconds.
   (= 28928 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:            5768.6     0.028057     0.027736     0.029353
Scale:           5148.7     0.031890     0.031076     0.035978
Add:             8231.6     0.029660     0.029156     0.031263
Triad:           7763.7     0.031483     0.030913     0.034004
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
```

Figure 4.13: STREAM scaling benchmark on multi-core.



```
-------------------------------------------------------------
This system uses 8 bytes per DOUBLE PRECISION word.
-------------------------------------------------------------
Array size = 2000000, Offset = 0
Total memory required = 45.8 MB.
Each test is run 10 times, but only
the *best* time for each is used.
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 2595 microseconds.
   (= 2595 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function      Rate (MB/s)   Avg time     Min time     Max time
Copy:        13658.0572     0.0024       0.0023       0.0025
Scale:       14394.8657     0.0024       0.0022       0.0025
Add:         15051.3301     0.0033       0.0032       0.0036
Triad:       14897.6315     0.0034       0.0032       0.0039
-------------------------------------------------------------
Solution Validates
-------------------------------------------------------------
```

Figure 4.14: STREAM scaling benchmark on OpenMP.

Chapter 5

Conclusion & Future Work

Designing efficient performance analysis subsystems has become an increasingly challenging task within the modern HPC community. As we approach the exascale era, relying solely on CPU power is no longer sufficient, leading to more complex designs in HPC system software and hardware stacks, and a greater emphasis on software performance engineering. However, the lack of a unified performance analysis framework in the HPC community results in tools tailored to specific system designs, limiting their portability and flexibility. Moreover, traditional HPC performance methods heavily rely on specific languages and runtimes, necessitating unique configurations for different architectures.

The growing trend of HPC systems adopting Linux kernel support and utilizing Linux native functionality for observability presents a new avenue for HPC performance engineering. By leveraging eBPF's USDT and uprobes, fine-grained cross-platform sampling of performance data becomes feasible, with subsequent analysis using visualization tools like Vampir and TAU. Although eBPF is not yet extensively used in HPC system performance engineering, it is believed to bring about a paradigm shift and shape the future of large-scale parallel software analysis architecture.

5.1 Conclusions

In this thesis, we conduct a comprehensive analysis of the performance requirements of HPC systems with varying emphases and designs. We investigate existing industrial-grade HPC performance tool implementations to understand the challenges they pose for the development of scientific computing applications. We also identify inefficiencies, fragmentation, and limitations in some existing methods. This thesis examines several challenges in modern HPC system performance methods, with a specific focus on emerging observability methods, parallel runtime, and bottleneck identification. Moreover, we propose new technologies that can guide the future implementation of HPC performance methods.

The first major contribution of this thesis is a systematic review of the architecture and software-hardware design of HPC systems in recent years. By gaining a comprehensive understanding of the performance components in high-performance computing systems, we can better grasp the co-design ideas between HPC systems and scientific computing applications. Our approach involves mapping the workload characteristics of scientific computing applications to compute systems, heterogeneous computing systems, and I/O subsystems, thus observing the combined effects of performance events at different levels and avoiding biases that may arise from solely focusing on source code or parallel runtime. This methodology provides a higher level of performance feature integration and reduces the risk of falling into local traps.

Additionally, building on traditional performance engineering methodologies, we explore the potential of implementing a unified and portable performance analysis tool as the second contribution of this thesis. We contend that current HPC performance methods have not achieved true portability, resulting in a proliferation of performance analysis tools with redundant features and limited insights for unknown HPC systems that lack specific adaptations. Through the demonstration of eBPF's cross-platform portability and non-intrusive dynamic instrumentation, we highlight the significant advantages of this performance frontend, particularly in terms of reduced runtime overhead.

5.2 Future Work

In the following sections, we will provide a detailed introduction to some potential future work that can further explore HPC performance methods based on eBPF, including utilizing kernel-level observable technologies such as kprobes and uprobes. These future research areas are:

5.2.1 Workload-Aware Runtime

Designing a workload-aware runtime system that leverages eBPF-based performance instrumentation can significantly enhance the adaptability and efficiency of HPC applications. By dynamically adjusting performance monitoring and profiling based on the specific characteristics of the workload, such a runtime system can reduce overhead and provide more accurate performance insights. This involves developing intelligent algorithms and heuristics to detect workload patterns, identify relevant performance metrics, and automatically configure the eBPF-based instrumentation accordingly.

5.2.2 Generative Heuristic Micro-benchmark

Creating a generative heuristic micro-benchmark suite that can automatically generate representative synthetic workloads is another area of potential research. These synthetic workloads can be used to stress test HPC systems and performance analysis tools under various conditions. By using eBPF to profile these synthetic workloads, researchers and developers can gain valuable insights into system behavior and identify potential performance bottlenecks.

5.2.3 Reproducible Performance Settings Database

Building a comprehensive and reproducible performance settings database is crucial for the HPC community. This database can store a wide range of performance configurations, including eBPF instrumentation settings, runtime parameters, and system characteristics. Researchers and practitioners can then use this database to share and compare performance results across different HPC systems and applications, promoting transparency and reproducibility in performance analysis.

These future research areas can extend the capabilities of eBPF-based HPC performance methods and address some of the challenges faced in designing efficient and scalable performance analysis tools. By exploring these avenues, the HPC community can advance towards a more unified and standardized approach to performance analysis, benefiting a wide range of applications and architectures.

Bibliography

[1] B. Gregg, *BPF Performance Tools: Linux System and Application Observability*, 1st ed. Addison-Wesley Professional, 2019.

[2] T. Cornebize, "High performance computing : Towards better performance predictions and experiments." [Online]. Available: https://theses.hal.science/tel-03328956

[3] V. García Flores, A. J. Peña Monferrer, and E. Ayguadé Parra, "Memory hierarchies for future HPC architectures." [Online]. Available: http://hdl.handle.net/2117/113684

[4] W. Frings, "Efficient Task-Local I/O Operations of Massively Parallel Applications," pp. xiv, 140 S. [Online]. Available: https://juser.fz-juelich.de/record/811621

[5] D. Computadors, V. Pillet, J. Labarta, T. Cortes, and S. Girona, "PARAVER: A tool to visualize and analyze parallel code," vol. 44.

[6] S. El Sayed Mohamed, "Analysis of I/O Requirements of Scientific Applications," pp. xv, 199 S. [Online]. Available: https://juser.fz-juelich.de/record/851081

[7] S. Hunold, J. I. Ajanohoun, I. Vardas, and J. L. Träff, "An overhead analysis of mpi profiling and tracing tools," in *Proceedings of the 2nd Workshop on Performance EngineeRing, Modelling, Analysis, and VisualizatiOn Strategy*, ser. PERMAVOST '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 5–13. [Online]. Available: https://doi.org/10.1145/3526063.3535353

[8] T. Ponweiser, "Profiling And Tracing Tools For Performance Analysis Of Large Scale Applications." [Online]. Available: https://zenodo.org/record/830390

[9] C. Coti, K. Huck, and A. D. Malony. STaKTAU: Profiling HPC applications' operating system usage. [Online]. Available: http://arxiv.org/abs/2304.11205

[10] V. R. G. Da Silva, A. B. N. Da Silva, C. Valderrama, P. Manneback, and S. Xavier-de Souza, "A Minimally Intrusive Approach for Automatic Assessment of Parallel Performance Scalability of Shared-Memory HPC Applications," vol. 11, no. 5, p. 689. [Online]. Available: https://www.mdpi.com/2079-9292/11/5/689

[11] Z. Lian, Y. Li, Z. Chen, S. Shan, B. Han, and Y. Su, "eBPF-based Working Set Size Estimation in Memory Management," in *2022 International Conference on Service Science (ICSS)*, pp. 188–195. [Online]. Available: http://arxiv.org/abs/2303.05919

[12] P. Giannozzi, S. Baroni, N. Bonini, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, G. L. Chiarotti, M. Cococcioni, I. Dabo, A. D. Corso, S. de Gironcoli, S. Fabris, G. Fratesi, R. Gebauer, U. Gerstmann, C. Gougoussis, A. Kokalj, M. Lazzeri, L. Martin-Samos, N. Marzari, F. Mauri, R. Mazzarello, S. Paolini, A. Pasquarello, L. Paulatto, C. Sbraccia, S. Scandolo, G. Sclauzero, A. P. Seitsonen, A. Smogunov, P. Umari, and R. M. Wentzcovitch, "QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials," *Journal of Physics: Condensed Matter*, vol. 21, no. 39, p. 395502, sep 2009. [Online]. Available: https://doi.org/10.1088%2F0953-8984%2F21%2F39%2F395502

[13] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 785–794. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939785

[14] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," vol. 22, no. 6, pp. 702–719.

[15] G. Aguilera, P. Teller, M. Taufer, and F. Wolf, "A systematic multi-step methodology for performance analysis of communication traces of distributed applications based on hierarchical clustering," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, 2006, pp. 8 pp.–.

[16] S. Shende and A. Malony, "The tau parallel performance system." vol. 20, pp. 287–311.

[17] K. Huck, A. Malony, R. Bell, and A. Morris, “Design and implementation of a parallel performance data management framework,” in *2005 International Conference on Parallel Processing (ICPP’05)*, 2005, pp. 473–482.

[18] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, “HPCTOOLKIT: Tools for performance analysis of optimized parallel programs,” pp. n/a–n/a. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/cpe.1553

[19] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, “The Vampir Performance Analysis Tool-Set,” in *Tools for High Performance Computing*, M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, Eds. Springer Berlin Heidelberg, pp. 139–155. [Online]. Available: http://link.springer.com/10.1007/978-3-540-68564-7_9

[20] M. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. Nagel, “Developing scalable applications with vampir, VampirServer and VampirTrace.” ser. Parallel Computing: Architectures, Algorithms and Applications, vol. 15, pp. 637–644.

[21] A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam, “Pvm: Parallel virtual machine: a users’ guide and tutorial for networked parallel computing,” *Computers in Physics*, vol. 9, pp. 607–607, 1995.

[22] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz, “Caliper: Performance introspection for HPC software stacks,” in *SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 550–560.

[23] A. Knüpfer, C. Rössel, D. A. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, “Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir,” in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds. Springer Berlin Heidelberg, pp. 79–91. [Online]. Available: http://link.springer.com/10.1007/978-3-642-31476-6_7

[24] M. Geimer, P. Saviankou, A. Strube, Z. Szebenyi, F. Wolf, and B. Wylie, "Further improving the scalability of the scalasca toolset," 06 2010, pp. 463–473.

[25] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," vol. 14, no. 3, pp. 189–204. [Online]. Available: https://doi.org/10.1177/109434200001400303

[26] S. Benedict, V. Petkov, and M. Gerndt, "Periscope: An online-based distributed performance analysis tool," 01 2009, pp. 1–16.

[27] O. Zaki, E. Lusk, and D. Swider, "Toward scalable performance visualization with jumpshot," *International Journal of High Performance Computing Applications*, vol. 13, 02 1999.

[28] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, B. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall, "The paradyn parallel performance measurement tool." *Computer*, vol. 28, pp. 37 – 46, 12 1995.

[29] J.-Y. Le Boudec, *Performance Evaluation of Computer and Communication Systems*. EPFL Press, Lausanne, Switzerland.

[30] M. Sato, Y. Ishikawa, H. Tomita, Y. Kodama, T. Odajima, M. Tsuji, H. Yashiro, M. Aoki, N. Shida, I. Miyoshi, K. Hirai, A. Furuya, A. Asato, K. Morita, and T. Shimizu, "Co-design for a64fx manycore processor and "fugaku"," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[31] A. Sergeev and M. Del Balso. Horovod: Fast and easy distributed deep learning in TensorFlow. [Online]. Available: http://arxiv.org/abs/1802.05799

[32] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 3505–3506. [Online]. Available: https://doi.org/10.1145/3394486.3406703

[33] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990.

[34] J.-P. Prost, *MPI-IO.* Boston, MA: Springer US, 2011, pp. 1191–1199. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_297

[35] P. Swartvagher, "On the interactions between HPC task-based runtime systems and communication libraries." [Online]. Available: https://theses.hal.science/tel-03989856

[36] J. Milton and P. Zarkesh-Ha, "Impacts of Topology and Bandwidth on Distributed Shared Memory Systems," vol. 12, no. 4, p. 86. [Online]. Available: https://www.mdpi.com/2073-431X/12/4/86

[37] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," *SIGARCH Comput. Archit. News*, vol. 36, no. 3, p. 77–88, jun 2008. [Online]. Available: https://doi.org/10.1145/1394608.1382129

[38] J. Domke, S. Matsuoka, I. R. Ivanov, Y. Tsushima, T. Yuki, A. Nomura, S. Miura, N. McDonald, D. L. Floyd, and N. Dubé, "Hyperx topology: First at-scale implementation and comparison to the fat-tree," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356140

[39] A. Yasin, "A Top-Down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* IEEE, pp. 35–44. [Online]. Available: http://ieeexplore.ieee.org/document/6844459/

[40] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *ACM Trans. Storage*, vol. 7, no. 3, oct 2011. [Online]. Available: https://doi.org/10.1145/2027066.2027068

[41] B. Gregg and J. Mauro, *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*, 1st ed. USA: Prentice Hall Press, 2011.

[42] M. Frigo and S. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[43] L. S. Blackford, J. Choi, A. Cleary, A. Petitet, R. C. Whaley, J. Demmel, I. Dhillon, K. Stanley, J. Dongarra, S. Hammarling, G. Henry, and D. Walker, "Scalapack: A portable linear algebra library for distributed memory computers - design issues and performance," in *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '96. USA: IEEE Computer Society, 1996, p. 5–es. [Online]. Available: https://doi.org/10.1145/369028.369038

[44] F. Lordan, E. Tejedor, J. Ejarque, R. Rafanell, J. Álvarez, F. Marozzo, D. Lezzi, R. Sirvent, D. Talia, and R. M. Badia, "ServiceSs: An Interoperable Programming Framework for the Cloud," vol. 12, no. 1, pp. 67–91. [Online]. Available: https://doi.org/10.1007/s10723-013-9272-5

[45] A. Abel and J. Reineke, "uiCA: Accurate throughput prediction of basic blocks on recent intel microarchitectures," in *Proceedings of the 36th ACM International Conference on Supercomputing*. ACM, pp. 1–14. [Online]. Available: https://dl.acm.org/doi/10.1145/3524059.3532396

[46] F. G. Van Zee and R. A. van de Geijn, "Blis: A framework for rapidly instantiating blas functionality," *ACM Trans. Math. Softw.*, vol. 41, no. 3, jun 2015. [Online]. Available: https://doi.org/10.1145/2764454

[47] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[48] R. L. Graham, T. S. Woodall, and J. M. Squyres, "Open mpi: A flexible high performance mpi," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Waśniewski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 228–239.