Jered Dominguez-Trujillo

Candidate

Computer Science

Department

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

Approved by the Thesis Committee:

Patrick G. Bridges

Chairperson

Trilce Estrada

Member

Amanda Bienz

Member

Statistical Modeling of HPC Performance Variability and Communication

by

Jered Dominguez-Trujillo

B.E., Mechanical Engineering, Vanderbilt University, 2018

THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Computer Science

The University of New Mexico

Albuquerque, New Mexico

July, 2021

Dedication

To Grandpa Benny, for his dedication to family and for his joy of learning.
And to Uncle Mike, for his support and positivity. COVID took you too soon.

Acknowledgments

The path back home to the University of New Mexico and eventually, the Scalable Systems Lab, has been a winding, and unpredictable journey that, with the help and support of many people, has made this work possible and fulfilling.

I first want to thank my advisor, Patrick Bridges, for his enthusiasm in bringing me aboard, for sharing his knowledge and passion in the distributed systems and high-performance computing fields, and for his support and guidance during my time at UNM. Furthermore, I owe a debt of gratitude to my other committee members, Amanda Bienz and Trilce Estrada, for their guidance, insight, and support. Also, I owe a large thanks to Patrick Widener, who provided access to systems, and useful insights to many of the challenges along the way.

I wish to thank many of my lab peers, Quincy Wofford, Sahba Tashakkori, Soheila Jafari, Rei Haskins, Nick Abel, William Pepper Evans, Noah Evans, Muna Tageldin, Christopher Leap, Gerald Collom, Andres Quan, Donald Kruse, and Harshita Sahni for their support and companionship, especially while we weathered the move to remote learning during the COVID-19 pandemic. Thanks to the many folks that are part of the Center for Understandable, Performant Exascale Communication Systems (CUP-ECS), for providing input, feedback, and resources. I also want to thank our collaborators at Sandia, especially Scott Levy and Kurt Ferreira, for taking the time to provide feedback and much appreciated expertise on several research ideas and directions. Additionally, I want to thank Bob Bird of LANL and Stuart Slattery of ORNL and the other members of the ECP-CoPa project for taking the time to provide feedback, code comments, and helpful tips. Lastly, I want to thank Bob Robey for many of the early conversations we had that provided insight into the operation of a complex parallel scientific code such as CLAMR and for his support over the past year and a half.

I am incredibly thankful for the many professors, teachers, mentors, and colleagues I have had. In particular, I want to thank a few Vanderbilt professors: Haoxiang Luo for providing my first research opportunity and encouraging me to pursue graduate school and Amrutur Anilkumar for challenging me unlike I had ever been before. I want to thank my former MIT colleagues, specifically Mark Chang, Laurens Voet, Kanika Gahkar, Geoffrey Svensson, Aidan Dowdle, Yuankang Chen, Albert Gnadt, Andras Kiss, Alexander Joerger, and Jinwook Lee for their continuing support from afar and for their encouragement that empowered me to pursue a better situation in a different field while being closer to home.

I would like to thank the UNM Center for Advanced Research Computing, the Texas Advanced Computing Center, NERSC, and the many other facilities at LANL, LLNL, and SNL and the people that maintain these facilities, for providing the HPC resources to do this work. I would also like to thank the SMART Scholarship Program for funding my undergraduate and graduate education.

Lastly, I would like to thank my family for their unwavering support and encouragement. They have been a constant source of reassurance. Specifically, I want to thank my parents, Jerry and Cindy, for their love and support, and my brother, John, for his companionship.

Statistical Modeling of HPC Performance Variability and Communication

by

Jered Dominguez-Trujillo

B.E., Mechanical Engineering, Vanderbilt University, 2018

M.S., Computer Science, University of New Mexico, 2021

Abstract

Understanding the performance of parallel and distributed programs remains a focal point in determining how compute systems can be optimized to achieve exascale performance. Lightweight, statistical models allow developers to both characterize and predict performance trade-offs, especially as HPC systems become more heterogeneous with many-core CPUs and GPUs. This thesis presents a lightweight, statistical modeling approach of performance variation which leverages extreme value theory by focusing on the maximum length of distributed workload intervals. This approach was implemented in MPI and evaluated on several HPC systems and workloads. I then present a performance model of partitioned communication which also uses an expected maximum value method. This performance model was validated with benchmarked results from HPC systems. These lightweight, statistical models provide insight into the behavior of HPC applications and systems and allow developers to predict performance impacts as HPC systems evolve towards exascale.

Contents

List of Figures

List of Tables

Glossary

API	Application Programming Interface
BSP	Bulk Synchronous Parallel
DGEMM	Double-Precision Matrix-Matrix Multiply
EMMA	Expected Mean Maximum Approximation
FTQ	Fixed-Time Quantum
FWQ	Fixed-Work Quantum
GEV	Generalized Extreme Value
HPC	High-Performance Computing
HPCG	Higher-Performance Conjugate-Gradient
i.i.d.	Independent and Identically Distributed
LAMMPS	Large-scale Atomic/Molecular Massively Parallel Simulator
MLE	Maximum Likelihood Estimators
MOM	Method of Moments
MPI	Message Passing Interface

Glossary

PWM	Probability Weighted Moments
SPMV	Sparse Matrix-Vector Multiply

Chapter 1

Introduction

Performance modeling of parallel scientific codes executed on high performance computing systems will continue to shape the strategies used to understand and optimize their behavior and performance. As supercomputing platforms continue towards exascale speeds [66], system architects have diversified platform architectures with the large variety of many-core processors and graphics processing units (GPUs) available in order to leverage their platforms' ability to increase shared-memory parallelism [27]. As these systems evolve, performance variability between parallel segments of code and between runs exacerbates performance, scaling, and scheduling issues, while the increased parallelism places higher loads on their communication systems [45]. For many reasons, many application developers continue to use only basic MPI functionality, even in modern exascale applications. This could be due to a lack of familiarity with new MPI features, performance portability issues between MPI implementations, or the constraints imposed while maintaining large, legacy codebases. To address this, researchers are seeking to address these issues to allow developers to access and leverage modern MPI functionality to ease communication performance constraints and mitigate performance variation between highly threaded applications.

Lightweight statistical modeling methods can provide insight that allows application developers to understand and predict the magnitude of performance loss with minimal measurement and analysis overhead. It can also provide a mechanism to identify specific optimizations that can be used to improve performance at scale. My thesis is that statistical models can be applied to various performance issues of parallel scientific codes on HPC systems to help identify, characterize, and predict the performance of these applications as many-core CPUs and GPUs introduce interesting new trade-off spaces and performance issues. This document describes the approaches, methodologies, and evaluations of two statistical models characterizing performance variation and an emerging communication strategy, respectively.

1.1 Performance Modeling Goals

The goal of performance modeling is to capture, characterize, and approximate the behavior of a system with the use of simplifying assumptions. As with other models, performance modeling need not be perfect; it just has to be useful. Many other performance models for emerging and existing HPC systems and scientific applications have been developed [12, 17, 48], but few have utilized statistical methods and techniques to simplify the collection and analysis of data required of them.

Throughout this work, developing useful models means developing performance models which provide insights into scaling behavior and that can be used to predict and capture performance trends across various changing variables. With that in mind, this work presents two models which provide the ability to characterize performance, provide insight into causes and effects of changes in performance, are a useful tool to evaluate relevant trade-off spaces, and minimize the overhead and the complexity of usage.

1.2 Performance Modeling of Scientific Codes

The importance of performance models for scientific codes quickly became apparent as compute resources became cheaper and more readily available. As distributed and parallel computing systems rapidly scaled while scientists and engineers continued to develop applications to be run on these systems, the issue of scaling behavior became important to the success of running larger and higher fidelity scientific simulations. Understanding how an application is expected to perform as it is scaled across multiple compute nodes to be run in parallel became essential to developing scalable and performant applications. This led to the analysis of the performance and behavior of several subsystems using many different techniques. The compute and communication subsystems were primarily modeled, while the analysis techniques relied heavily on empirical measurements and basic performance models.

1.2.1 Background

Initial performance models allowed researchers and developers to estimate the expected speed-up they would observe in a parallel code given only the percentage of the code that could be parallelized and the number of tasks that could simultaneously execute a parallel portion of the code. These models enable an appropriate number of parallel tasks to be selected for a given problem in order to minimize diminishing returns. However, given that the percentage of code that can be parallelized is not necessarily simple to calculate in large, complex, and parallel codebases, more detailed and specialized performance models have been developed to characterize how well codes will scale in order to solve problems of interest that require enormous amounts of data and analysis. Furthermore, the increasing degree of heterogeneity of the systems only exacerbates this challenge. More detail on these models is provided in Section 2.5.

For this reason, many developers and researchers use profiling or tracing tools, such as Tau [79] and Likwid [85, 86], to gather detailed data about program path and performance, while hardware counter data can be gathered with the linux perf utility [22], PAPI [26, 71, 84], or PAPI++ [54, 55]. However, each of these methods requires the instrumentation of code, and the collection and analysis of gigabytes or terabytes of data in order to understand and model application performance. While performance models have been developed for many of the subsystems and events that can occur in a parallel code on a supercomputing system, developing lightweight models, that minimize data collection and analysis overheads, to characterize performance variation and communication performance are and will continue to be an area of interest and are the types of models discussed in this thesis.

1.2.2 Performance Variation

As in serial programs, parallel and distributed programs experience performance variation [80]. In serial programs, run-to-run variation is often due to noise or system load, but it is more generally predictable since there are fewer variables and possible sources of noise. Performance variation in parallel and distributed programs can manifest in four ways: (1) run-to-run variation between identical runs, (2) variation across processes during the same run, (3) communication loads of nearby jobs interfering with the current job, and (4) the topology of assigned resources for the current job. Run-to-run variation is still characterized by the overall variation in runtime across identical runs. However, run-to-run variation can be influenced and amplified by variation across processes during the same run.

For example, when a parallel code is performing a compute step, each process may be assigned to perform the same steps on partitions of the entire data to share the work (this is called a single instruction, multiple data, or SIMD, approach). However, noise, due to operating system processes [35], context switches, clock throttling [1], etc. on each process can produce variation of runtime length and performance on each processor. Additionally, if work is not split evenly, the load on each processor is then imbalanced and this will lead to additional performance variation between processors. If this compute step is followed by a synchronization step between all processors, as is commonly done between compute and communication, the effects of this variation will be propagated from the slowest process to all processes, exacerbating the overall observed performance variation between identical runs. Furthermore, other parallel approaches, such as multiple instruction, multiple data (MIMD), encounter similar issues.

In the absence of imbalanced workloads across processes, other factors such as nearby jobs and the topology of nodes assigned to a job, can induce performance variability. Nearby jobs may perform frequent communication and hog shared network resources in unpredictable patterns, and the network distance between nodes assigned to a job may change unpredictably from run-to-run. These sources of performance variability can be difficult to model due to their non-deterministic nature and statistical approaches may be better suited to capture their effect on performance variability.

As programs scale to be run across hundreds or thousands of processors, the variation across processors can become large and significantly impact overall program performance. Characterizing and predicting this variation through modeling can allow system administrators to develop better schedulers to allow more optimal use and sharing of HPC resources between users. Additionally, it can aid application developers with keeping their programs efficient by determining whether their application is expected to experience significant runtime variability. This can then provide insight on where to improve their program to reduce this variability.

Typical approaches to minimize performance variation between processors and across runs in parallel programs include ensuring load balance between processors and minimizing synchronization points. Commonly, overlapping communication with computation phases will improve performance, but is not always possible and may increase the program complexity significantly and the difficulty of the performance modeling task. To understand whether these solutions are necessary or viable, the variability across processors and across runs must be measured and characterized. This naturally points towards a statistical modeling approach that emphasizes expected maximum runtimes across processes to account for variability and synchronization between processes in order to predict total run-to-run variability.

1.2.3 Statistical Methods and Extreme Values

Statistical modeling approaches seek to better characterize variability by describing a group of processors performing a task as a distribution of runtimes. It follows that overall performance of an application can be described by the expected maximum runtime given the number of performing tasks, since each processor is essentially sampling this distribution of runtimes. Restated, the overall runtime of parallel routines is determined by the slowest process, and using the expected maximum runtimes can help develop a model to characterize the scalability and variability of parallel programs.

This branch of statistics, called General Extreme Value (GEV) theory, has commonly been used in hydrology to model the frequency of extreme weather events, but can also be applied to model the expected runtime of a program with a large number of parallel processes by using distributions of expected maximum values. It can also predict how the variability between processes and runs will scale as programs are run on more processes. This allows both performance variability to be modeled with distributions of maximums, and provides the basis for modeling various communication strategies that take advantage of the variability of runtimes between processes to overlap communication and computation and improve overall performance.

1.2.4 Communication Performance

Parallel and distributed scientific codes are typically structured either with a bulk-synchronous parallel (BSP) approach [89] or in a more loosely task-based approach where data dependencies and ordering are enforced by the creation of a directed acyclic graph (DAG). The Message Passing Interface (MPI) [69] has typically been used in the former and is the ubiquitous standard used to implement distributed-memory parallelism and communication. Therefore, this work focuses on the BSP approach. This approach is characterized by a period of parallel compute, where variables of interest, such as stress, strain, torsion, velocity, momentum, vorticity, pressure, density, temperature, etc. of regions of a domain (i.e. as a field and typically called an Eulerian description) or of particles themselves (a Lagrangian description) are calculated. This is followed by a synchronization and a period of data exchange where neighboring cells in a domain are exchanged or the particles themselves are exchanged between processes before proceeding to the next iteration of compute. During this data exchange, each process will post sends and receives to all of the processes it is exchanging data with using the MPI API and then all processes synchronize once again before continuing. This pattern of communication is commonly referred to as a halo exchange, or as a gather and scatter, and is prevalent in many scientific programs.

Communication performance has been traditionally modeled using two parameters: latency and bandwidth. Latency describes the start-up overhead of sending a message, while bandwidth describes the rate at which data traverses the network. Common tools and benchmarks, such as the OSU benchmark suite [73], are typically used to measure the latency and bandwidth of both shared-memory communication and distributed-memory communication. These parameters are typically functions of network hardware and topology, and can also be functions of message size, since various MPI implementations perform different protocols and optimizations based on the message size. More sophisticated performance models are described in Section 2.5.

As HPC systems continue to exploit parallelism to a greater degree, communication is going to become more frequent and integral to program performance, and models will be required to estimate the impact communication techniques have on overall program performance in addition to providing insight on how communication methods can be optimized.

1.3 Contributions

The major contributions of this work are:

- An MPI implementation of the measurement approach underlying a performance variability model that enables its application to a range of HPC workloads and systems.
- A new approach to modeling, measuring, and analyzing performance and performance variation in large-scale systems and applications based on parametric and non-parametric bootstrapping of empirical maxima distributions.
- An evaluation of parametric and non-parametric bootstrapping techniques for quantifying and predicting performance variation in large-scale systems, including an analysis of the strengths and limitations of the modeling assumptions underlying the different analysis techniques.

- A partitioned communication benchmark to measure partitioned communication performance as a function of message size and number of threads.
- A novel statistical performance model for partitioned communication.
- A demonstration of the predictive capability of this partitioned communication performance model.

1.4 Thesis Outline

The remainder of this thesis is outlined as follows: Chapter 2 discusses related work and background information regarding the previous work done in characterizing performance variability, partitioned communication in HPC systems, and the statistical and performance modeling frameworks built upon in this work to model performance variability and partitioned communication. Chapter 3 describes the design, implementation, and results of an initial performance and variability prediction method, which utilizes the GEV distribution, on a range of BSP workloads. Chapter 4 develops and evaluates a performance model for partitioned communication to provide insight on its performance advantages in highly multi-threaded systems and with workloads having various amounts of runtime variability or imbalance across its threads. Chapter 5 provides a summary of contributions and concluding thoughts in addition to future research directions.

Chapter 2

Related Work

2.1 Introduction

The statistical modeling approaches described in this thesis have their foundation in previous work to measure and characterize the sources and magnitudes of performance variability, statistical methods based in GEV theory, proposed additions to the MPI standard to add partitioned communication, and HPC performance modeling. This chapter discusses the previous work in these sub-fields and how the two statistical models developed and evaluated in this thesis build on these works to provide a novel contribution to the field.

Section 2.2 discusses previous studies that evaluated the sources and magnitude of performance variation due to noise and interference in HPC workloads. Next, Section 2.3 provides background on the statistical frameworks used to model the performance of parallel programs with synchronizations. Section 2.4 discusses a new communication technique, partitioned communication, that will be introduced to the MPI standard. Finally, Section 2.5 discusses commonly used performance models for HPC performance and communication along with their strengths and weaknesses.

2.2 Performance Variability

Performance variability studies of large-scale parallel applications on HPC systems have typically fallen into two categories: benchmarking performance and evaluating underlying causes of performance variability. The goal of these studies is to allow developers to tune their programs for optimal performance. For this to be feasible, developers must be able to distinguish natural run-to-run performance variation from performance improvements due to a change or optimization. The scale of natural performance variability in these applications and systems makes it difficult to discern whether changes in code result in runtime improvements or whether runtime improvements are solely an artifact of run-to-run variation. Hence, determining performance impact requires several runs and large amounts of data collection and analysis. Specifically, research has found the change in performance to be masked by certain tuning enhancements [75], posing challenges to the evaluation of the effectiveness of performance tuning. This has resulted in many studies which aim to characterize the underlying mechanisms that cause performance variation. From this, researchers have demonstrated broader sources of software and hardware architecture and application level variability, measured the performance loss caused by variability, and tried to control and decrease these sources and measure the resulting performance improvement [80].

In addition to evaluations of the underlying mechanisms that cause performance variation, there are several benchmarking studies of HPC systems in the literature. These have shown that CPU throttling and other power control mechanisms, such as Intel Turbo Boost, amplify the effect of manufacturing variability on CPU frequency and degrade the performance of HPC applications [1]. Other studies have shown that OS jitter and interference can affect application performance and may limit application scalability depending on the distribution and frequency of the noise [35]. Furthermore, the sharing of network resources among many jobs in HPC systems has been identified as a particularly unpredictable and difficult to model source of performance variation due to nearby jobs [10]. In an attempt to aid the evaluation, benchmarking, and reproducibility of performance variation, libraries have been developed which attempt to simulate common interference patterns and sources which stress various subsystems, such as the CPU, cache, memory, and the network [6].

Additionally, the presence of both spatial and temporal variability within HPC systems and applications can create other performance variation modeling challenges [57]. Spatial variability occurs as a result of the location within the HPC system that a job is scheduled. Temporal variability can take place over several time scales. On short time scales, variability can be manifested due to OS interference or CPU throttling while the evolution and maintenance of HPC systems and their software/hardware may contribute to variability on longer time scales.

Many authors have measured and modeled performance variability with a variety of approaches and techniques, such as machine learning techniques [87], statistical analysis methods [67], and neighborhood analysis [11]. Examples of these approaches include a machine learning model developed to identify runtime anomalies and performance variation in near real-time [87] and a method of performance analysis and extrapolation for bulk synchronous programs which utilizes extreme value theory [67]. Neighborhood analyses have used network counters to investigate the causes of performance variability in HPC systems due to neighboring jobs to forecast expected execution time [11]. However, these methods do not provide a framework to predict how the performance and performance variability of an application will scale.

2.3 Generalized Extreme Value Theory

Generalized extreme value theory [21] is the most commonly used statistical tool to characterize the distributions of maxima. It has typically been used to characterize and predict the frequency of extreme events in the field of hydrology [64]. Section 2.3.1 defines the GEV distribution and its relevant parameters. Next, Section 2.3.2 describes the relevant estimation methods used to estimate GEV parameters from empirical data. Section 2.3.3 then discusses the Effective Mean Maximum Approximation (EMMA) that can be used with general distributions, including the GEV distribution, to project maximum expected values at larger scales. Finally, Section 2.3.4 gives a brief overview of how the GEV distribution, estimation techniques, and EMMA, can be used to perform both parametric and non-parametric bootstrapping to predict maxima distributions at larger scales.

2.3.1 Generalized Extreme Value Distribution

GEV theory consists of a set of probability distributions, the Gumbel distribution [43], the Fréchet distribution [38], and the Weibull distribution [90], which characterize the independent and identically distributed (i.i.d.) random variables of sets of maximums taken from a previous distribution. The three distributions are distinguished from one another by their tail behavior. The Generalized Extreme Value (GEV) distribution [56] is able to characterize these three distributions with a set of three parameters: a shape parameter describing which of the three distributions a distribution of interest is most similar to and hence, the tail behavior, a location parameter that describes the center of the data, and a scale parameter that describes the spread of the data. The cumulative distribution function (CDF) of the GEV distribution is:

$$F\left(x|\kappa,\xi,\alpha\right) = e^{-\left(1+\kappa\left(\frac{x-\xi}{\alpha}\right)\right)^{\frac{-1}{\kappa}}} \tag{2.1}$$

where κ is shape, ξ is location, and α is a scale parameter. When the shape parameter κ is 0, the extreme value distribution is Type I or Gumbel. When ξ is greater than 0, it is Type II or Fréchet, and if κ is less than 0, it is Type III or Weibull.

2.3.2 Estimation of Maxima Distributions

Estimating the parameters (κ, ξ, α) of the GEV distribution given random samples is essential to correctly characterize the underlying distribution the samples came from. Several methods have been developed to estimate these parameters. Three different methods were compared and evaluated for the work in this thesis: maximum likelihood estimators (MLE), probability weighted moments (PWM), and the method of moments (MOM).

Maximum Likelihood estimators are widely used and are implemented in popular statistical libraries, such as in the Python SciPy Stats library [78]. However, multiple studies have found that the MLE is unstable when used with small samples sizes ($15 < n < 100$) [51] when compared to PWM. On the other hand, Madsen et al. [63] have found that MOM has lower root mean square error when dealing with smaller sample sizes. Given this information and experience gathered during the implementation and testing of the models in this thesis, the MOM and PWM methods were found to be more reliable than MLE for fitting GEV distribution parameters given the empirical data collected in this work. This section describes the MOM and PWM estimation methods.

Method of Moments

The method of moments [63] for estimating the GEV distribution parameters is given the sample mean ($\hat{\mu}$), standard deviation ($\hat{\sigma}$), and skewness ($\hat{\gamma}$) as inputs, and then minimizes Equation 2.2 to estimate the shape parameter ($\hat{\kappa}$).

$$\hat{\gamma} = sign(\hat{\kappa}) * \frac{-\Gamma(1+3\hat{\kappa}) + 3\Gamma(1+\hat{\kappa})\Gamma(1+2\hat{\kappa}) - 2[\Gamma(1+\hat{\kappa})]^3}{\{\Gamma(1+2\hat{\kappa}) - [\Gamma(1+\hat{\kappa})]^2\}^{3/2}} \tag{2.2}$$

Equation 2.2 is typically minimized with respect to $\hat{\kappa}$ using the Nelder-Mead method with an initial guess of $\hat{\kappa} = 0$ and where $\Gamma(n) = (n-1)!$. After the shape parameter ($\hat{\kappa}$) has been approximated, the scale ($\hat{\alpha}$) and location ($\hat{\xi}$) parameters are estimated using Equation 2.3 and Equation 2.4, respectively.

$$\hat{\alpha} = \frac{\hat{\sigma}|\hat{\kappa}|}{\{\Gamma(1+2\hat{\kappa}) - [\Gamma(1+\hat{\kappa})]^2\}^{3/2}} \tag{2.3}$$

$$\hat{\xi} = \hat{\mu} - \frac{\hat{\alpha}}{\hat{\kappa}}\{1 - \Gamma(1+\hat{\kappa})\} \tag{2.4}$$

Probability Weighted Moments

Similarly, the method of probability weighted moments [51] takes the first three moments, mean ($\hat{\mu}$), variance ($\hat{\sigma^2}$), and skewness ($\hat{\gamma}$), as inputs to compute the shape parameter ($\hat{\kappa}$) using Equation 2.5 below:

$$\hat{\kappa} = 7.8590c + 2.9554c^2 \tag{2.5}$$

Where c is defined in Equation 2.6 as:

$$c = \frac{2\hat{\sigma^2} - \hat{\mu}}{3\hat{\gamma} - \hat{\mu}} - \frac{log2}{log3} \tag{2.6}$$

The scale ($\hat{\alpha}$) and location ($\hat{\xi}$) parameters are then calculated with Equation 2.7 and Equation 2.8, respectively.

$$\hat{\alpha} = \frac{(2\hat{\sigma^2} - \hat{\mu})\hat{\kappa}}{\Gamma(1+\hat{\kappa})(1-2^{-\kappa})} \tag{2.7}$$

$$\hat{\xi} = \hat{\mu} + \hat{\alpha}\frac{\Gamma(1+\hat{\kappa})-1}{\hat{\kappa}} \tag{2.8}$$

2.3.3 Expected Maximum Values

The Effective Mean Maximum Approximation (EMMA) method can and has been extended to help predict the runtimes of parallel applications. Given a distribution, this method can help approximate the expected maximum when n values are randomly sampled from the defined distribution. In the context of parallel computing, the runtimes of a particular task on each thread or process can be described by such a distribution, and the overall runtime of the task can be determined using the EMMA method, since the overall runtime of synchronizing threads or processes is dependent on the slowest (i.e. maximum) overall runtime amongst the processes.

Sun et al. [82] developed a closed form formulation of this method using extreme value theory. It can be used to quickly estimate the expected maximum values given several distributions and can be extended to help model runtimes on heterogeneous systems. This can then be used to estimate the compute time in BSP [89] programs with GEV theory [21]. Since most parallel programs are composed of several MPI ranks performing a computation simultaneously, followed by a synchronization, the runtime of a program can be characterized by the process/rank that finishes last. Since various sources of noise and interference are present, such as OS interference [35], it is expected that even processes with a perfectly balanced set of work will have runtimes that are characterized by a distribution with some amount of nonzero variance, typically normal or exponential. The presence of load imbalance will only exacerbate this. As the number of processes grows, the likelihood of observing a process that takes an "extreme" amount of time increases as a function of the distribution that runtimes are characterized by.

The EMMA function is defined in Equation 2.9:

$$E(max_{i=1}^{m} X_i) \approx F^{-1}(P) \tag{2.9}$$

where $P \approx 0.570376002^{\frac{1}{m}}$. This function estimates the expected value of the maximum of m observations of a set of i.i.d random variables (X_i) with the estimated cumulative distribution function (CFD), F, of the observations.

This function, along with the GEV estimation methods, are utilized to perform parametric bootstrapping of empirical distributions (see Section 2.3.4) in Chapter 3. This calculation can also be leveraged to calculate when the average runtime of the slowest thread in a set of n threads is expected to finish. It is also later used in Chapter 4 as the foundation for calculating the average extent of overlap between computation and communication that can be utilized in partitioned communication.

2.3.4 Bootstrapping Maxima Confidence Intervals

Bootstrapping is a common statistical technique which can be used to randomly re-sample with replacement from an empirical distribution to simulate additional samples from a parent distribution. Given a large sample and the assumption that the observed samples from the empirical distribution are representative of the population distribution, the bootstrapping process can be utilized to produce more accurate population parameter estimates [16,33] than if the parameters were estimated based on an initial, but incorrect, assumption of a specific distribution.

It follows that two types of bootstrapping can be performed: parametric bootstrapping and non-parametric bootstrapping. Parametric bootstrapping involves randomly generating samples from a parametric distribution fitted to the data using PWM or MOM (e.g. a GEV distribution with shape, location, and scale parameters). Non-parametric bootstrapping performs re-sampling with replacement from the given sample data without estimating parameters which describe the distribution of the sample data. It can then calculate the required statistic from a large number of repeated samples. Both techniques allow the calculation of standard errors and confidence intervals. Both parametric and non-parametric bootstrapping approaches have been compared with respect to their ability to estimate uncertainties in extreme-value models. The non-parametric bootstrapping results in confidence intervals which are too narrow and underestimate the real uncertainties involved in the frequency models [58], unless a large number of samples are used.

2.4 Partitioned Communication

Partitioned communication is an addition to the MPI 4.0 specification [70] that is intended to allow application developers to better leverage multi-threaded MPI in MPI+X applications. The goal of this addition is to provide a simple primitive that can optimize multi-threaded communication performance by overlapping communication and computation when threading is used within an MPI rank.

This section provides background on the motivations, rationale, and background research that resulted in this addition to the MPI specification. Section 2.4.1 discusses adoption rates of newer MPI features and the challenges of adding functionality to MPI that can be integrated into existing codebases and new applications. Section 2.4.2 discusses the current performance challenges of using multi-threaded MPI in MPI+X applications. Finally, Section 2.4.3 discusses the two proposals to add partitioned communication to the MPI specification and contrasts their approaches.

2.4.1 Evolution of MPI

Parallel programs which require distributed-memory communication have ubiquitously used MPI [69]. The simplest method of communication has traditionally been sending individual messages between processes, i.e. point-to-point communication. Additionally, collective communication methods are standardized and have been used when groups of processors require aggregate and summary information or the redistribution of arrays. More recently, one-sided communication has been introduced in the form of remote memory access (RMA), with accompanying benchmarks [30], to enable performance optimizations by application programmers and within the MPI implementation itself. However, the usage of MPI in legacy and new applications has been limited to a small subset of available MPI functionality due to application developers reluctance to refactor large codebases to incorporate newer features and lack of awareness of new features [9,81]. Going forward, additions to the MPI standard must keep this in mind, where additions to the standard must be easy to integrate into legacy codebases and simple to understand in order for application developers to widely adopt and realize their performance benefits. Additionally, MPI implementations must take care to appropriately optimize functionality without loss of generality and portability.

2.4.2 Multi-Threaded MPI

As high performance systems continue to evolve towards exascale [66] and become more heterogeneous, where compute nodes may contain processors with many CPU cores and/or connected GPUs, understanding the performance of MPI+X codes has become paramount. For example, as of 2020, almost 30 percent of the Top500 supercomputers [27] utilize GPUs. This move has been motivated by the desire to maximize the on-chip parallelism of such systems, but considerations about the impact on the performance of the communication system have become a focused area of research as the number of threads and on-chip, shared-memory parallelism has substantially increased. This has led to the creation of partitioned communication as specified in the Finepoints [40] proposal to the MPI 4.0 [69, 70] specification.

Typically, large scientific codes are run on multiple compute nodes, and each of these compute nodes contains one or several sockets which each contain a multi-core processor. A common programming model is MPI+X, where MPI handles the distributed-memory parallelism (i.e. the communication between nodes) while X signifies one of the many shared-memory programming tools, such as OpenMP [20], OpenACC [72], Kokkos [32], RAJA [8], Pthreads, etc. that handle the shared-memory parallelism. Each node will typically utilize a single MPI process, each of which will spawn several threads that can work in parallel on the node, but this ratio between processes and threads can and should be optimized for performance.

In the simplest scenario, where each node spawns a single MPI process which utilizes many threads, this requires that either a single thread is responsible for sending and receiving all data from the node, or that each thread is responsible for sending and receiving the data it is responsible for (i.e. a fraction of the data sent to and from the node). To accommodate this, the MPI specification and many MPI implementations have developed and implemented multiple MPI thread settings (`MPI_THREAD_SINGLE`, `MPI_THREAD_SERIALIZED`, `MPI_THREAD_MULTIPLE`) that allow the user to toggle between the single thread communication paradigm and the multiple thread communication paradigm by simply calling `MPI_Init_Thread` with the desired setting. Single threaded communication has been the most widely used mode in the community, while multi-threaded communication has been found to have large performance problems [30, 46, 77] as the number of cores per processor has increased.

As the number of cores and threads increases, this has led to a few problems in single threaded communication. For example, in the single threaded mode of MPI, synchronization of the threads is required before performing the MPI send and receive operations and the data to be sent and received is aggregated into a single large message to be sent by MPI. This results in two issues: an overhead where threads may be waiting on other threads before synchronization due to imbalance, and under-utilization of network resources where network time is wasted. This network time is wasted while waiting to send a single large message during thread synchronization rather than sending data in the form of multiple small messages from completed threads during this waiting period.

2.4.3 Endpoints and Finepoints

Two main solutions and additions to the MPI framework have emerged: the Endpoints [23] proposal and the Finepoints [40] proposal. The Endpoints proposal attempted to utilize a large namespace which consisted of the endpoints of each thread on every node, where threads would be able to send to other threads on other nodes. This resulted in an enormous namespace and large overheads to perform message matching. The Finepoints proposal improved upon this by introducing the concept of partitioned communication, where threads will notify the communication system when they are ready to send their data, and the communication will then send data from each thread to the receiving node in partitions (i.e. several small messages). This kept the overall namespace small and the impact on existing applications minimal since the endpoint of messages are still processes on nodes, rather than individual threads. While this increases the total time spent communicating, it is spread out across threads and allows overlap between the computation performed by threads and the subsequent communication, improving overall performance by 3-5% [40].

2.5 Performance Modeling

The performance modeling that the work in this thesis is based on can be classified into two categories: scaling performance and communication performance. Scaling performance is interested in modeling and predicting how an application will perform as the size of the parallel system increases and/or as the problem size increases. These basic performance models have their basis in measuring the percentage of an application that can be parallelized and determining the expected speed-up and parallel efficiency based on problem size and number of parallel resources. Alternatively, communication performance focuses on the performance of the communication subsystem and the coupling of performance between processes to provide a more detailed understanding of how applications that require communication may scale and how communication overheads may negatively impact observed speed-up and overall program efficiency.

This section introduces much of the previous work done in these two performance modeling approaches. Section 2.5.1 discusses strong and weak scaling and the initial models used to quantify the expected speed-up and efficiency of parallel applications. Section 2.5.2 then provides the foundations on which communication modeling is built and the communication models leveraged in the models presented in this thesis.

2.5.1 Modeling Scaling Performance

Performance modeling of parallel codes began with the introduction of Amdahl's law [5], which characterizes strong scaling, and Gustafson's law [44], which describes weak scaling. Strong scaling refers to the scenario of having a constant problem size resulting in a set amount of total work and quantifying the magnitude of speed up that can be achieved when completing the entire task on a larger number of parallel tasks. Weak scaling refers to keeping the amount of work per task constant while increasing the number of tasks (and hence, the size of the problem). The remainder of this section describes how strong scaling and weak scaling were initially modeled.

Strong Scaling

Amdahl's law is used to model expected speed-up of a program with a problem of fixed size as more and more processes are used in parallel. This model is given the percentage of a program that can be performed in parallel and the number of processors used in parallel as inputs. This speed-up is calculated from Equation 2.10 below:

$$\textit{speed-up} = \frac{s+p}{s+p/N} = \frac{1}{s+p/N} \tag{2.10}$$

where s is the fraction of time spent doing serial work, p is the fraction of the program that can be parallelized with perfect scaling, such that $s + p = 1$, and N is the number of processors working in parallel.

This results in the theoretical speed-up being bound by the percentage of the program that can be parallelized (for example, a program that can be 90% parallelized will have a maximum speed-up of 10 as $N \to \infty$). This law indicates that the speed-up increases asymptotically as the number of processes, N, is increased.

Weak Scaling

On the other hand, weak scaling refers to understanding the performance of a code when the total number of parallel tasks increases, but the amount of work per task remains constant (i.e. the total work increases proportionally with the number of parallel tasks). This approach was proposed by Gustafson since he realized that in practical applications, the problem size and number of processes used were rarely independent. Gustafson's law provides a basic model to calculate the predicted scaled speed-up as shown in Equation 2.11.

$$\textit{scaled speed-up} = s + p * N = N + (1 - N) * s \tag{2.11}$$

This results in linear speed-up as the number of processors, N, are increased, and is more applicable to most applications - as problem size increases, a proportional increase in number of parallel processes is usually used. Other metrics, such as parallel efficiency, which is defined as $\textit{efficiency} = \frac{\textit{scaled speed-up}}{N}$, are commonly used to evaluate the ability of a program to weakly-scale well. Furthermore, metrics such as iso-efficiency [36] can be used in conjunction with these performance laws to reason about how the number of processes should scale with problem size to maximize the efficiency of a large-scale parallel program.

2.5.2 Modeling Communication Performance

Parallel programs are often structured in alternating phases of compute followed by communication. Typically this is used to implement time-stepping through or iterating towards the solution of a set of partial differential equations or matrix equations which describe some system. The compute phase implements the numerical scheme to calculate the state of the system and can be simply modeled by quantifying computational efficiency and cache and memory performance. The communication phase typically implements some form of point-to-point communication between a pair of processors for halo exchanges or collective communication between many processors for re-distribution of data or aggregation of results and is typically modeled using network parameters. In theory, collective communication can be broken down into many point-to-point communications, and hence, most of the previous literature focuses on developing models for the performance of point-to-point communication.

The implementation of point-to-point communication varies across MPI implementations. In general, each message consists of metadata and data, where the metadata contains features describing the message, such as length, tag, MPI communicator, and IDs identifying sending and receiving processes [70]. MPI implementations typically implement several approaches to sending data depending on the size of the data to attempt to optimize performance on the network. This includes "eager" sends where data is sent immediately once the sending call is posted and assumes buffer space is available on the receiving processes, and "rendezvous" sends where the metadata is initially sent, and the receiver must notify the sender that memory has been allocated before the sender actually sends the data. The threshold of appropriate message size to use as the cut-off point between the approaches to best optimize performance has previously been modeled and explored using queueing methods that model network bottlenecks [15].

The remainder of this section describes a few of the most widely used communication models, such as the Postal model and the Max-Rate model followed by a brief discussion on additional models that add complexity to address the shortcomings of the more simplistic communication models.

Postal model

The most simple description of communication performance must consider the time taken to initialize the communication (the start-up time) and the time taken to send the data. The start-up time can be modeled as a constant overhead, whereas the time taken to send the data, the transport time, is a function of the size of the message to be sent. This basic understanding describes the Postal model [7, 47], which takes an input latency to quantify the start-up time, and an input bandwidth to quantify the rate of data transfer, and finally a message size to calculate the transport time given the bandwidth. That is,

$$T_{send}(\alpha, \beta, s) = \alpha + \frac{s}{\beta} \tag{2.12}$$

where, α represents latency, β represents bandwidth (bytes sent per unit of time) and s represents the size of the message in bytes. Measurement of the latency and bandwidth is typically done using a communication benchmark, and the OSU microbenchmark [73] has been the de-facto benchmark used to determine the inputs to the Postal model. This benchmark performs a ping-pong test on messages of various sizes and gathers latency and bandwidth data as a function of message size. Note that latency and bandwidth performance can also be functions of MPI implementation and network configuration and hardware, and are measured empirically before applying the Postal model to predict how communication performance is going to behave and scale in parallel programs on a specific system.

Max-Rate model

More complex models have also been developed, which attempt to account for and parametrize network and network card limits as many-core CPUs have become common. In applications where many MPI processes co-exist on a single node, many MPI processes may simultaneously try to send data from a single node, overloading the network card and network as it tries to inject data into the network as rapidly as possible. This may lead to a bottle-necked condition. This results in a non-linear degradation of communication performance as message rate and communication processes per node increases and shifts many applications to a regime where the Postal model is no longer sufficient to capture performance behavior. The Max-Rate model [41] was then developed to address this issue by considering both the latency and bandwidth, but also the maximum rate that a node can inject data into the network. This improvement accounts for this injection bandwidth bottleneck and the model is defined in Equation 2.13,

$$T = \alpha + \frac{ppn * s}{min(R_N, ppn * R_b)} \tag{2.13}$$

where, α again represents the latency, *ppn* represents the number of actively communicating processes per node, R_b is the inverse of the measured bandwidth, β, and R_N is the maximum rate that a node can inject data into the network.

Additional Models

Additional models of point-to-point communication, which attempt to account for message queues and message matching to improve the accuracy of communication performance models, have also been explored [13] [14]. Additionally, the family of LogP [18,19,39], LogGP [4], LogGOP [50], as well as the LoPC [37], and LoGPC [68] models, which attempt to improve the fidelity of performance models by capturing more detail about computational bandwidth and compute-communicate coupling, have been developed. While useful, the added input parameters of these models may be hard to directly measure on some systems and do not lend themselves to a first-order model that can be easily developed with a simple statistical framework.

Chapter 3

Performance Variability Model

Portions of this chapter appear in the paper "Lightweight Measurement and Analysis of HPC Performance Variability" published in PMBS '20 [24]

3.1 Introduction

Large-scale HPC systems and applications are prone to performance variation due to various sources of hardware and software configurations and characteristics common in modern systems. Sources of variation and noise include operating system activities [35], memory bandwidth contention between individual cores and network cards [42], throttling of processor clock speeds due to power and temperature constraints [1], inconsistent cooling patterns [75], and changes in network characteristics, such as links, routes, and switch capacities [11]. Additionally, non-determinism and inherent run-to-run variation present in many applications may play a role in performance variation. Furthermore, as HPC systems continue to increasingly share node-level resources, resource contention between applications running on neighboring nodes within the system can contribute to performance variation [34, 62, 76, 88].

Measuring, modeling, and predicting performance variation in large-scale systems is challenging for several reasons. Reproducibility and accuracy are concerns, since node and system behavior can vary over time, as well as across nodes [1, 57]. To address this and accurately characterize the system and its variability, it is necessary to collect large numbers of performance samples from many nodes over a period of many weeks or months. The result is the collection, storage, and analysis of large volumes of data that requires extensive overhead to organize and store while also requiring sophistication to meaningfully process and draw conclusions from. Additionally, researchers typically make simplifying modeling assumptions regarding behavior of the application and system performance in order to quantify the performance variation, and these assumptions may be implicit or may not be valid in practice. Empirical studies observe performance variation of applications on parallel and distributed systems which exhibit behaviors that may invalidate many of these modeling and analysis assumptions (e.g. continuity, temporal invariance, independence, identical distribution) [49].

Consequently, no current general approaches exist to measure, predict, and assess performance variation in real-world HPC systems. The community has made other efforts to measure and understand the performance variation and its underlying sources on specific nodes [1] or systems [35], but these efforts do not provide a general framework that can be extended to predict or assess how performance variability changes as a function of application scale. Furthermore, many performance modeling and prediction approaches emphasize predicting *average* performance [67] and overlook the performance variability common in real-world systems.

This work aims to provide users and applications developers an approach to quantifying and predicting performance variation in their applications and on their systems at scale. Challenges in developing efficient job scheduling protocols and routines that can maximize system throughput and efficiency are prevalent since users, system architects, and applications developers cannot accurately measure or predict performance variation. This work would allow systems to more accurately predict the required resources for a job. Rather than users making pessimistic resource allocation assumptions and decisions to accommodate for unexpected performance variation and anomalies, which result in an over-constrained scheduling system, system schedulers would make effective allocation decisions. This trade-off between user-defined and dynamic resource allocations could then be explored with respect to its effects on predictability and performance.

This chapter describes a novel approach to measuring, analyzing, and predicting performance variation in HPC systems. The approach combines a focus on the *maxima* length of distributed workload intervals with parametric and non-parametric statistical bootstrapping techniques. Compared to other approaches, focusing on estimating variation in maxima significantly simplifies measurement and analysis challenges since it avoids the need to reason about the complex distribution of runtimes on individual nodes. Overall, the goal of this work is to enable system and application architects to accurately assess and predict performance variation in large-scale systems and applications. This chapter presents the following contributions:

- A new approach to modeling, measuring, and analyzing performance and performance variation in large-scale systems based on bootstrapping either parametric or empirical maxima distributions (Section 3.2).
- An MPI implementation of the measurement approach that enables the application of this model to a range of HPC workloads and systems (Section 3.3.1).
- An evaluation of these techniques for quantifying and predicting performance variation in large-scale systems, including an analysis of the strengths and limitations of the modeling assumptions underlying the different analysis techniques (Sections 3.4 and 3.3.3).

These contributions were the result of collaborative work done at the University of New Mexico, with support from Sandia National Laboratories. These contributions are all presented within the context of the work in this thesis, and all contributions are a credit to the several students and principal investigators that participated in this research and the development of these measurement and modeling approaches. The specific contributions which are partially attributable to the author include the implementation of the measurement approach for a subset of the workloads tested, the evaluation and analysis of the modeling approach, including the gathering of data from the Cori system, and the organization, presentation, and analysis of the data and results from both the Cori and Attaway systems.

The remainder of this chapter is outlined as follows. Section 3.2 provides a description of the methodology of the parametric and non-parametric bootstrapping approaches. Next, Section 3.3 describes the MPI implementation of the measurement approach underlying the performance variability model in addition to enumerating the experiments performed, while Section 3.4 presents the gathered data and an analysis of the results. Finally, Section 3.5 provides a summary of the developed approaches and their results.

3.2 Methodology

While performance variation is a source of unpredictable behavior, it also inherently limits the scalability and performance of large-scale applications. In typical bulk-synchronous applications, where a loop characterized by a period of parallel compute is typically followed by a communication period with surrounding synchronization barriers, can describe the main program path, lagging processes can result in overheads on faster processes due to the synchronization requirement. This happens because periodic application-wide communication in distributed application requires all processes to wait for the slowest process before any other process can then proceed. Due to this, variability which manifests only within a single process is propagated to every other process and in turn, affects the overall performance of the application. For example, this frequently happens in MPI collective communication in bulk-synchronous programs. As the number of processes continues to grow, the likelihood of having slow, outlier processes that significantly affect runtime grows substantially and is heavily dependent on the tail behavior of the runtime distribution of the workload.

The methodology uses a combination of extreme value theory and statistical bootstrapping to achieve three goals:

1. Avoid making generalizing assumptions about the workload as a whole (i.e. specifically about the type of distribution governing the different workloads, as intra-node variability can and will impact those assumptions).

2. To reduce the amount of data that needs to be collected to accurately measure system and application performance characteristics.

3. To accurately quantify performance variation in large-scale systems and applications.

The remainder of this section describes a methodology that focuses on efficiently and accurately modeling this type of variability and its affect on performance. The basic performance quantification, prediction model, and its underlying assumptions used to develop this methodology are described in Section 3.2.1, while Section 3.2.2 discusses how this model can be leveraged to measure system performance with bootstrapping approaches to quantify the variation in measured and predicted workload performance.

3.2.1 Modeling Approach

This modeling approach measures, models, and analyzes the distribution of the *maximum* length of a fixed distribution compute/communicate workload at a given system scale. These workloads can largely be characterized by a computational kernel surrounded by synchronization points at which communication occurs, as is the typical application structure in bulk-synchronous programs. This can include relatively simple applications and complex applications, and only requires that they are delineated with `MPI_Init`/`MPI_Finalize` with clear compute and communicate phases fenced by synchronization operations.

Workload performance is modeled as a generally-distributed random variable of which a sample is the length of one execution of this workload. This random variable is then assumed to be the maximum of N random variables, each of which describes the length of program runtime on an individual node. Furthermore, it is assumed that these distributions are stationary (not time-varying). However, it is *not* assumed that the distributions of individual node runtimes are independent or identical, or that either the node or overall system runtime distributions are continuous.

This model then allows the measurement of only either:

1. The time between the end of successive synchronizing collectives, which is useful for the analysis of only the of workloads, such as in the non-parametric bootstrap approach described below, or;
2. the distribution of individual process inter-collective times such as in the inter-node and intra-node parametric bootstraps described below.

In both cases, the required data needed to be gathered and analyzed to quantify the system variation is significantly less than that required by approaches that rely on fine-grain tracing of all actions within the application.

3.2.2 Analytical Methods

Non-parametric and parametric bootstrapping make different underlying assumptions and have different sample size requirements. The non-parametric bootstrapping approach makes minimal assumptions about the distribution of maxima on nodes and focuses directly on the empirical distribution of the collected maxima. In contrast, the parametric bootstrapping approach assumes that either individual nodes or sets of nodes have i.i.d workload lengths. While the non-parametric bootstrapping approach has weaker assumptions, it requires a larger number of global maximum samples to be effective than the parametric bootstrapping approach. However, some parametric bootstrapping approaches require performance samples from individual nodes, and hence a higher data requirement, to avoid its stronger assumption that all workload samples are i.i.d.

Non-parametric bootstrapping

The non-parametric version of this approach does not utilize explicit model fitting. Rather, the maxima data is re-sampled to simulate the effects of scaling-up the workload. In particular, the non-parametric bootstrapping is implemented by randomly resampling the original maxima data k times with replacement and taking the maximum of these k values as a new sample on a system or application running on k times as many processes. This process is then repeated n times to generate the bootstrapped empirical samples for the larger system being modeled. The 95% confidence interval is then directly computed from the resulting set of maxima scaled up via this re-sampling. This process is described in Figure 3.1.

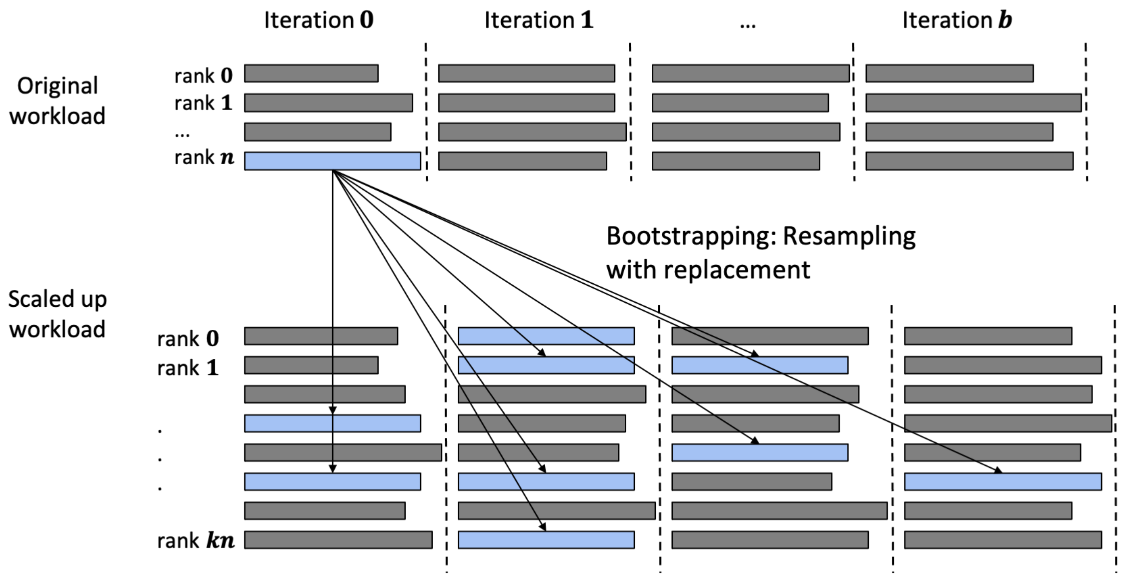


Figure 3.1: Summary of non-parametric modeling methodology.

Parametric bootstrapping

The parametric version of this approach fits a GEV distribution to a collection of empirical maxima, performs a prediction of the expected growth, and finally calculates the confidence intervals of expected performance. To perform the GEV fitting, the two methods described in Section 2.3.2 are used: PWM and MOM. Although mixed methods combining MLE and MOM have been previously explored to fit GEV [3], PWM has shown greater stability compared to MLE even though it can still break down in the presence of outliers [31]; thus, the need to combine them. By using both PWM and MOM in conjunction with bootstrapping, the strengths of each method can be leveraged while reducing the likelihood of instability in the fitting.

After a good fit is obtained, the Expected Mean Maximum Approximation [82] (EMMA) technique, which can be used to predict the performance correctly when the distribution of maxima are known a *priori*, can be used. The EMMA function is defined in Equation 2.9.

To expose the variability that occurs naturally within a node and across the system, this distribution is then bootstrapped in two ways:

- **Inter-node Parametric**, where one node at a time is used to generate the 50 bootstrap replicas and this process is repeated for the total number of nodes in the sample data (e.g. for 8 nodes, $8 \times 50 = 400$ replicas per sample workload are generated).
- **Intra-node Parametric**, where a single rank for each node is selected to generate the 50 replicas, and this process is repeated for the total number of ranks per node (e.g. for 32 ranks per node, $32 \times 50 = 1600$ replicas per sample workload are generated).

The 95% bootstrap confidence interval is then extracted by calculating the $2.5th$ and $97.5th$ percentiles of the distribution of EMMA projected runtimes. For a sample size n and a confidence interval ci, the position in the ordered collection of resampled projections corresponding to the bounds of the empirical bootstrap confidence interval are given by $[n(1-ci)/2, n(1+ci)/2]$. As for the number of bootstrap replicas that are needed in practice to compute a stable 95% confidence interval, Efron et al. [33] suggested 200 replicas for calculating the bootstrap standard error but 1000 or more for computing the bootstrap confidence interval. This entire parametric bootstrapping process can be summarized by Figure 3.2.

3.3 Implementation and Experimental Setup

The general methodology described in Section 3.2 was implemented in order to evaluate its ability to measure, analyze, and predict performance variation in modern systems when combined with various statistical performance prediction techniques.

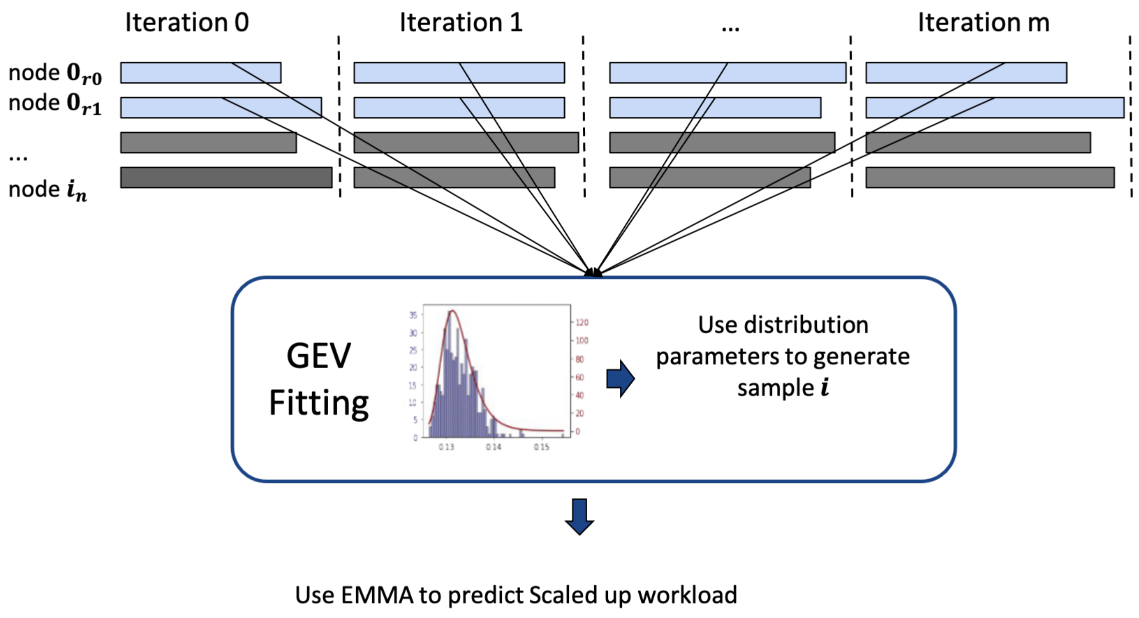


Figure 3.2: Summary of parametric modeling methodology.

This implementation was then configured to be run across six workloads of varying complexity on two supercomputing platforms.

3.3.1 Measurement Implementation

A simple MPI application for executing and measuring the performance and performance variation of a wide range of HPC workloads was designed and implemented to allow the collection of maxima runtime samples from a variety of workloads. This system is based on the ubiquitous bulk-synchronous approach to designing parallel applications. To achieve this, the measurement program executes k *intervals* in which n *processes* compute. Each interval begins with an `MPI_Barrier`, after which the test program runs a specified MPI/compute workload on each process (which may include communication through an MPI subcommunicator) before finally executing a second `MPI_Barrier`. Each process then logs and stores the elapsed time between synchronization points (i.e. the time to compute the workloads), and the rank 0 process logs the longest time taken by any process during the kth interval by using a call to `MPI_Reduce`. Upon program completion, these logs are then written to a parallel file system for later analysis in order to minimize runtime overhead.

3.3.2 Synthetic and Application Workloads

Table 3.1: Synthetic and application workload supported by measurement tool.

Workload	**Description**	**Parameters**
FTQ [2]	Spin for a statistically distributed length of time	Distribution and distribution parameters
FWQ [2]	Perform a statistically distributed number of integer additions	Distribution and distribution parameters
DGEMM	Single-Node Dense General Matrix Multiply	Size of matrix and number of multiplications per iteration
SPMV	Single-Node Sparse Matrix Vector Multiply	Size of matrix and number of multiplications per iteration
HPCG [25]	Distributed Pre-conditioned Conjugate Gradient Linear Solver	Global number of rows in matrix being solved
LAMMPS [74]	Molecular Dynamics Solver	Global number of molecules and timesteps to simulate

This benchmark includes several synthetic workloads that mimic common HPC computation and communication patterns, in addition to library versions of several HPC applications. These workloads are summarized in Table 3.1.

In addition, the benchmark supplements each of these synthetic compute workloads with a simple 2D halo exchange of a configurable size in each iteration to mimic BSP applications with different computational and communication characteristics. Two different distributed MPI applications were also integrated into this system, HPCG [25] and LAMMPS [74]. The distributed sparse matrix vector multiply (SPMV) workload within HPCG was utilized as its own unique workload in addition to the entire HPCG routine, which performs several SPMVs. Furthermore, the Lennard-Jones benchmark of LAMMPS was the benchmark of choice that was supported using API calls exposed by LAMMPS when compiled as a library. These application workloads perform a fully distributed solve or simulation, adding complex communication characteristics to the framework.

For each of these workloads, the global synchronization step guarantees that the distribution of runtimes is the maxima of the local compute/communication times on each node. When all local distributions are identical Gaussian distributions, either given directly or established by the central limit theorem, this distribution approaches a Gumbel distribution (i.e. a GEV distribution with a shape parameter of 0). More generally, when the local distributions are all continuous and i.i.d, the maximum distribution is the GEV distribution. It is important to note, however, that in many cases the distributions of local compute/communication times are *not* i.i.d. in real systems due to indirect coupling or interference between cores/processes on a single chip or due to resource sharing across processes. For example, explicit communication or indirect interference (e.g. L3 cache conflicts) between processes can violate independence assumptions. Similarly, differences in resource allocations between cores and nodes (e.g. dynamic clock frequency management) can violate identical distribution assumptions. These challenges can have potentially significant impact on the suitability of different analysis approaches, as evaluated in Section 3.4.1.

3.3.3 Experimental Setup

The evaluation of the statistical prediction techniques presented in this work was performed by running and collecting data from the six workloads outlined in Table 3.1 at various scales on two supercomputing systems (Table 3.2). The prediction techniques were evaluated based on their ability to quantify and predict performance variation. Each run of the various workloads provided a collection of maxima timings from each of the k intervals of each workload. The non-parametric and parametric bootstrapping techniques described in Section 3.2.2 were then used to evaluate the ability of each method to predict performance variation at various scales.

Table 3.2: Platform hardware details.

Platforms	**Cori**	**Attaway**
Processor	Intel Xeon E5-2698 v3 (Haswell) Intel Xeon Phi Processor 7250 (Knights Landing)	Intel Xeon Gold 6140
Clock Speed	2.3 GHz (Haswell) / 1.4 GHz (Knights Landing)	2.3 GHz
Cores per node	32 (Haswell) / 68 (Knight's Landing)	36
Total Nodes	2388 (Haswell) / 9688 (Knights Landing)	1488
Memory per Node	128 GB (Haswell) / 96 GB (Knights Landing)	192 GB

The data generated and used in this evaluation were collected on two supercomputing systems: Cori at the National Energy Research Scientific Computing Center, and Attaway at Sandia National Laboratories. Each workload was run on some combination of 8, 16, 32, and 64 nodes, with 32 MPI ranks per node, corresponding to 256, 512, 1024, and 2048 MPI ranks, respectively. Generally, 20 runs were performed on 256 ranks, 5-10 runs on 512 ranks, 5-10 runs on 1024 ranks, and 3-5 runs on 2048 ranks. This provided ample data at "smaller" scales to feed the bootstrapping techniques, while still providing test data at "larger" scales to test the accuracy of the predictive techniques in the limited provided system allocations.

Additionally, the FTQ, FWQ, DGEMM, and SPMV workloads were each run with and without a 1MB 2-D halo exchange, as described in Section 3.3.2. The halo exchange component was not included for the HPCG and LAMMPS workloads as each already contains communication and synchronization operations.

Each workload was controlled with several input parameters (Table 3.1). These were tuned to allow each workload to run in reasonable amounts of time, to generate enough data for bootstrapping, and to utilize appropriate amounts of memory to minimize cache effects. Specifically, the FTQ workload was run with 100 ms normally distributed intervals, while the FWQ workload was similarly run with normally distributed loop counts. Additionally, the SPMV and HPCG workloads were initialized to utilize 1GB per rank of memory to minimize cache effects, and the LAMMPS workload was performed on 64,000 atoms per rank for 250 time steps utilizing the Lennard-Jones potential computation.

Table 3.3: Experiment overview.

System	**Cori (# of experiments)**		**Attaway (# of experiments)**	
Workload	*No Halo*	*Halo*	*No Halo*	*Halo*
FTQ	38	38	30	30
FWQ	53	53	30	30
DGEMM	53	53	40	40
SPMV	53	53	35	35
HPCG	38	0	30	0
LAMMPS	35	0	30	0

Overall, 467 runs on Cori and 330 runs on Attaway were performed, broken down between the 6 workloads of interest and with or without the inclusion of the additional 2D halo exchange as seen in Table 3.3. With these runs, it was possible to collect ample data to characterize the performance of a significant portion of the Cori and Attaway systems. Table 3.4 is a more detailed breakdown of the experiments performed for each system and workload combination.

Shown in Figure 3.3, the experiments utilized 1461 of the 2388 unique Haswell nodes on Cori and 319 of the 1488 unique Xeon Gold nodes on Attaway. On both Cori Haswell nodes and Attaway Xeon nodes, 32 cores per node were utilized. Cori nodes are configured to allow hardware use of Intel Turbo Boost by default and this feature was not explicitly disabled. Attaway nodes do not have Turbo Boost enabled.

3.3.4 Validation of Data Collection

To validate the scalability of this measurement technique, we measured the time spent in the barrier after each iteration of a workload using the DGEMM workload

Table 3.4: Detailed experiment breakdown.

System	Workload	Nodes	Ranks	Experiments
Cori	FTQ	8, 16, 32, 64	256, 512, 1024, 2048	20, 10, 5, 3
Cori	FWQ	8, 16, 32, 64	256, 512, 1024, 2048	20, 20, 10, 3
Cori	DGEMM	8, 16, 32, 64	256, 512, 1024, 2048	20, 20, 10, 3
Cori	SPMV	8, 16, 32, 64	256, 512, 1024, 2048	20, 20, 10, 3
Cori	HPCG	8, 16, 32, 64	256, 512, 1024, 2048	20, 10, 5, 3
Cori	LAMMPS	8, 16, 54	256, 512, 1728	20, 10, 5
Attaway	FTQ	8, 16, 32	256, 512, 1024	20, 5, 5
Attaway	FWQ	8, 16, 32	256, 512, 1024	20, 5, 5
Attaway	DGEMM	8, 16, 32	256, 512, 1024	20, 10, 10
Attaway	SPMV	8, 16, 32	256, 512, 1024	20, 10, 5
Attaway	HPCG	8, 16, 32	256, 512, 1024	20, 5, 5
Attaway	LAMMPS	8, 16, 54	256, 512, 1728	20, 5, 5

at 1024 ranks. On average, the DGEMM workload took 790 milliseconds on Cori and 820 milliseconds on Attaway, while the average time spent in the barrier was 24 microseconds and 15 microseconds on each system, respectively. This indicates that a negligible amount of the workload is spent in the barrier at some of the largest scales run at, showing that the measurement technique should be highly scalable on both systems.

3.4 Evaluation

This section provides the results of the measured performance variability and evaluates the performance of the performance variability prediction approach and its ability to characterize the potential performance variation of different workloads on several modern HPC systems. Section 3.4.1 provides the empirical performance variability observed across workloads on both the NERSC Cori and SNL Attaway supercomputing systems. Next, Section 3.4.2 outlines the evaluation approach steps utilized in Section 3.4.3 to validate the performance prediction of the model across

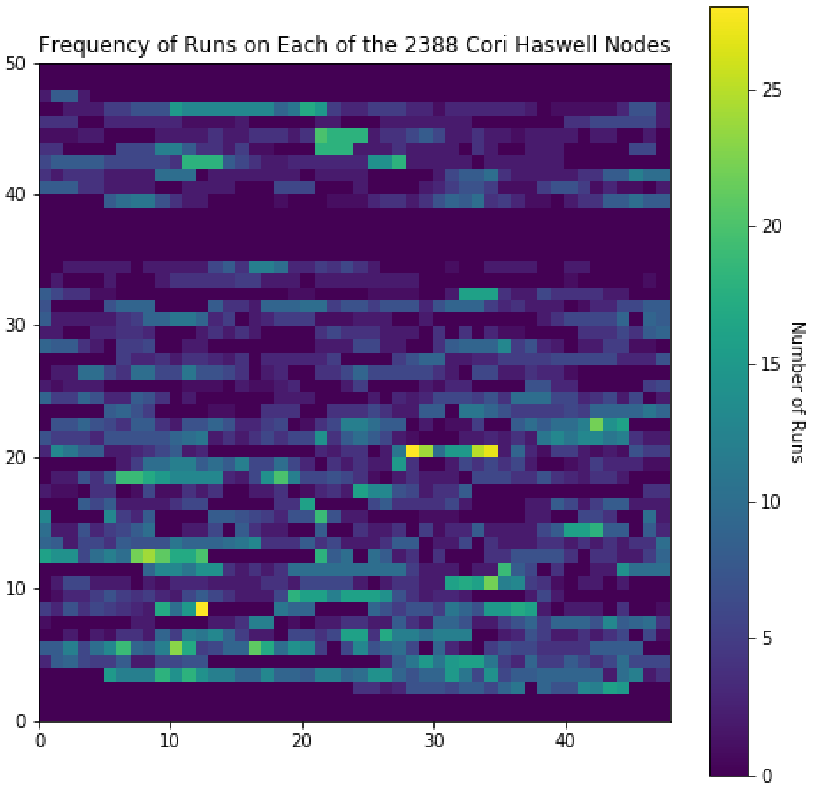


Figure 3.3: Heatmap of nodes utilized for Cori experiments. Overall coverage of 61% of Haswell nodes available on Cori.

the workloads tested. Finally, Section 3.4.4 discusses the trade-offs of various statistical performance prediction techniques as a function of workload complexity.

3.4.1 Assessment of Performance Variation

This section evaluates the overall runtimes of each run to assess the performance variation observed for each workload on each platform. We present runtimes both with and without halo exchanges, but this work does not specifically discuss variance in the halo exchange runs to minimize the parameter space and to simplify the evaluation of the various bootstrapping techniques. As seen in Figure 3.4, the runtime performance on Cori showed minimal variability for the FTQ workload (less than 1%), while compute-bound workloads (e.g. DGEMM) showed up to 10% variability across identical runs. The Attaway runtimes in Figure 3.5 showed less variability for the FTQ, FWQ, DGEMM, and SPMV workloads, but experienced 15% and 94% variability for the HPCG and LAMMPS workloads, respectively, most likely due to significant internal network dependencies.

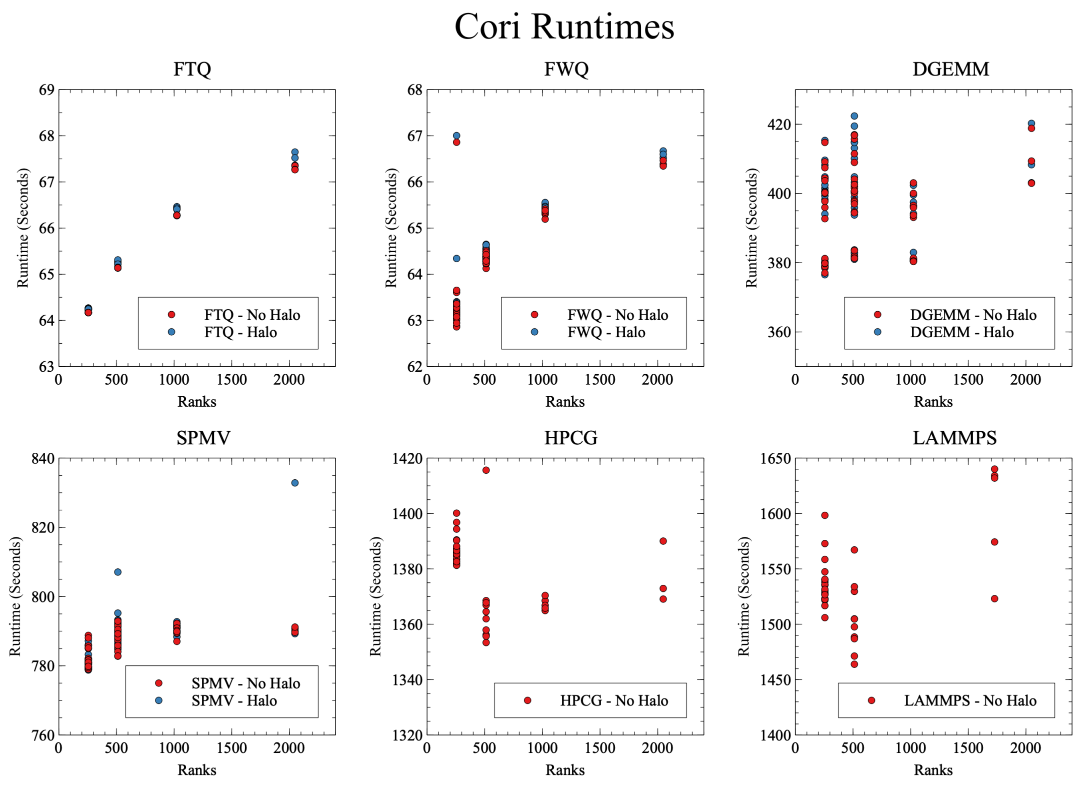


Figure 3.4: Runtimes of each workload at various scales on the NERSC Cori supercomputer.

While the workloads of FTQ and FWQ were well-behaved, the DGEMM and SPMV workloads exhibited larger variances since they are real compute-bound and memory-bound kernels subject to processor speed control and memory contention. Similarly, the HPCG and LAMMPS workloads exhibited more unpredictable behavior due to their internal communication patterns and network dependencies.

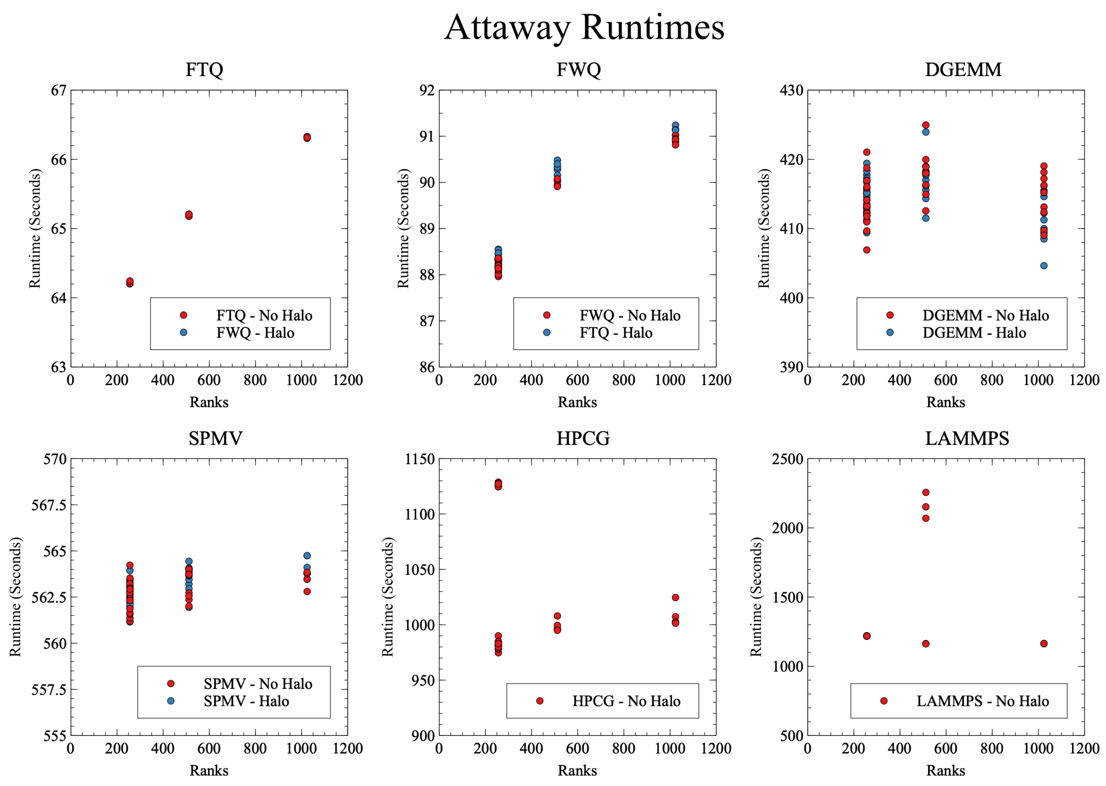
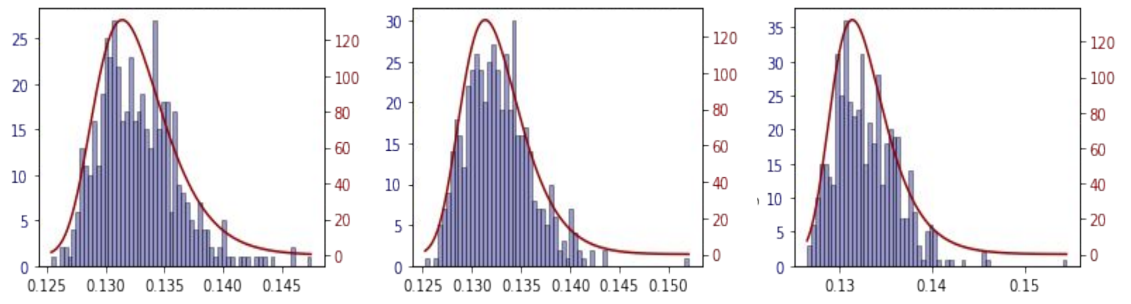
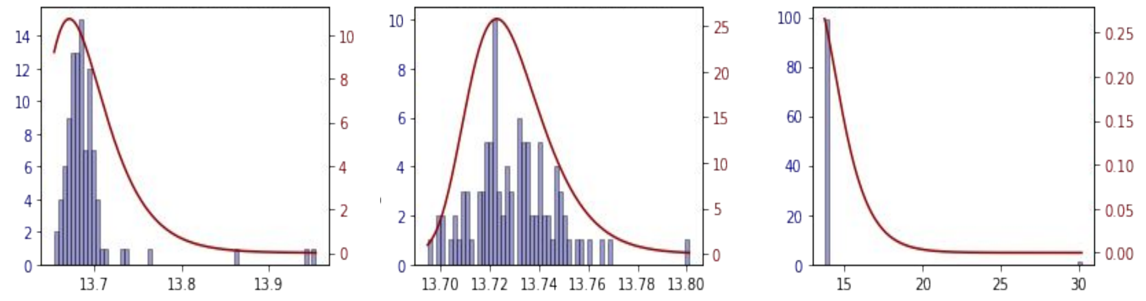


Figure 3.5: Runtimes of each workload at various scales on the SNL Attaway supercomputer.

Beyond the visual inspection, the variability of workloads across runs can be characterized by analyzing their GEV fitting parameters. For example, Figure 3.6 shows MOM fittings for the very stable FTQ workload, and for the more unpredictable HPCG. For space purposes only MOM fittings are depicted; however both PWM and MOM are compared. For FTQ, both PWM and MOM agree across runs that this is a Type II distribution. On the other hand, for HPCG the fitting methods do not reach an agreement. PWM characterizes the distribution as Type III, while MOM characterizes runs 1 and 3 as Type I, and run 2 as Type II (see Section 2.3 for the definition of the different types). This assessment of distribution type and agreement between fitting methods provides valuable insight into whether the predictive capabilities of these methods can be trusted for specific workloads.



a) FTQ workload variation and MOM fitting



b) HPCG workload variation and MOM fitting

Figure 3.6: Characterization of maxima for FTQ and HPCG across Cori runs.

3.4.2 Evaluation Approach

The evaluation approach consists of four steps:

- Usage of the collected data to calculate the total runtime of each experiment.
- Prediction of performance variation at larger scales with the bootstrapping techniques described in Section 3.2.2 applied to the smallest scale data collected (256 ranks).

- Evaluation of the ability of each technique to accurately predict and quantify performance variation.
- Discussion of the trade-offs of the various techniques, their underlying assumptions, and their performance.

For each method, the data resulting from running each workload with 256 ranks (the smallest data set collected) on Cori and Attaway was used to calibrate the performance variation prediction techniques of Section 3.2.2. These methods were then used to generate medians and confidence intervals of the predicted runtimes at various scales. The actual runtimes at 256, 512, 1024, and 2048 ranks were then used to evaluate whether each method was able to successfully quantify and predict the performance variation observed. We then determined whether the observed runtimes fell within the predicted confidence intervals, and whether the confidence interval reasonably reflected the observed variance at larger scales.

3.4.3 Prediction of Performance Variation

Workloads that are more controlled (e.g. FTQ) can be predicted well by techniques (e.g. EMMA, GEV) that further characterize variance, but those techniques make strong assumptions (e.g. known distribution or i.i.d.) which do not work well on workloads with more complicated behaviors. On those, more general techniques that cannot bound variation as tightly are necessary.

For controlled workloads, such as FTQ and FWQ, both non-parametric and parametric methods exhibited good predictive performance. As seen in Figures 3.7 - 3.8, the intra-node parametric technique was able to successfully predict the runtimes of the FTQ and FWQ workloads at 512, 1024, and 2048 ranks successfully while maintaining reasonable confidence intervals. The inter-node results were excluded for brevity, but exhibit the same behavior.

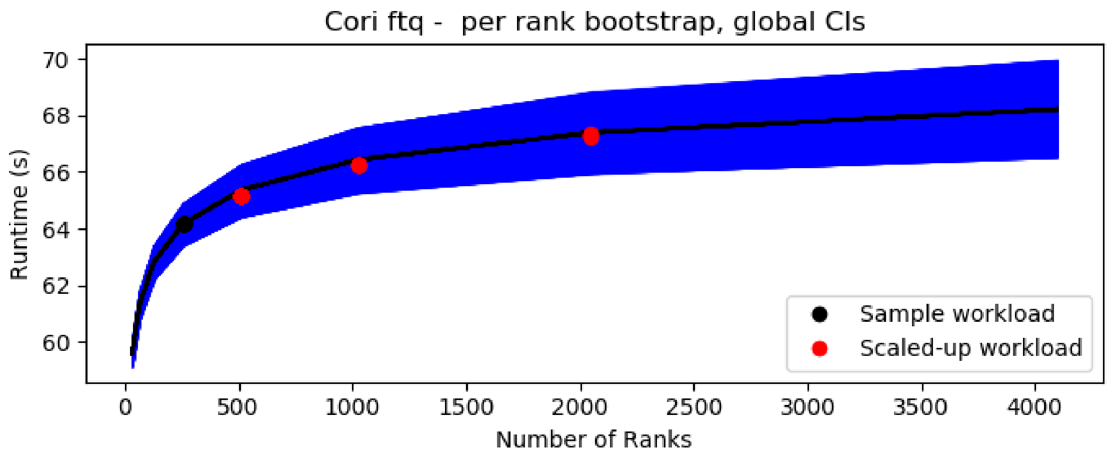


Figure 3.7: Parametric intra-node prediction of performance variation of Cori FTQ workload.

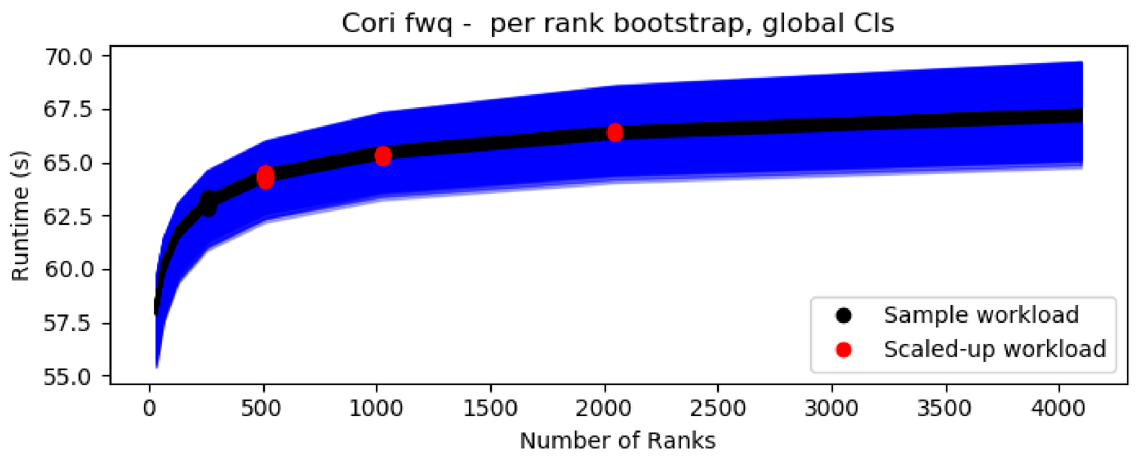


Figure 3.8: Parametric intra-node prediction of performance variation of Cori FWQ workload.

For less-controlled workloads that are representative of common kernels in HPC applications, the parametric methods again provided good predictive power. Since the DGEMM and SPMV workloads were run such that each rank was responsible only for its own operations, no internal communication except for the synchronization after each iteration was performed. These kernels exhibit almost perfect weak scaling, and the parametric methods are able to accurately capture this as seen in Figures 3.9 - 3.10. However, since the DGEMM kernel is compute bound and subject to CPU frequency variation and other sources of noise, and because SPMV is memory bound and subject to memory latency variations, there can be substantial variation of up to 10% in runtimes across identical runs, as seen in Figures 3.4 - 3.5. As seen in Figures 3.9 - 3.10, the inter-node parametric method also captures this variation accurately. This time, the intra-node results have been excluded for brevity, but they exhibit similar behavior.

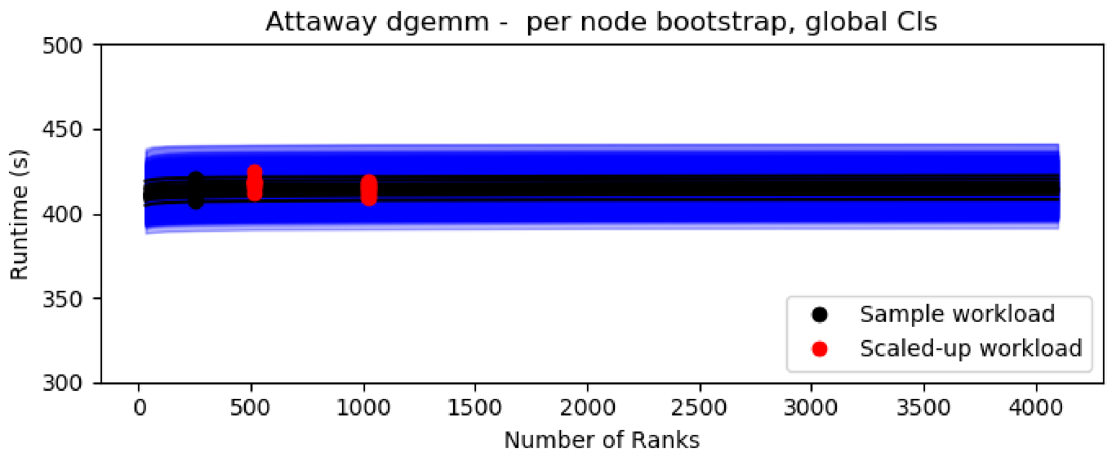


Figure 3.9: Parametric inter-node prediction of performance variation of Attaway DGEMM workload.

The resampling non-parametric prediction method and the intra-node and inter-node parametric prediction methods can behave differently in evaluation of real-world applications such as the HPCG and LAMMPS workloads. The communication and synchronization within each iteration of HPCG and LAMMPS result in an additional dependency on network performance which may produce extreme outliers as network traffic fluctuates. This internal communication and synchronization occurs before

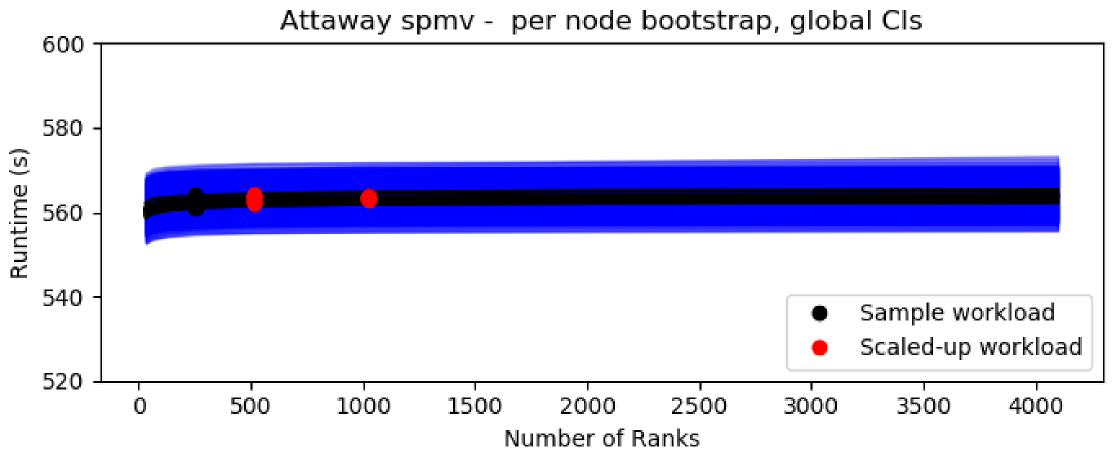


Figure 3.10: Parametric inter-node prediction of performance variation of Attaway SPMV workload.

the measurement framework runs at each step, effectively masking the variation in performance between the various ranks. The same synchronization ahead of the measurement point can cause all ranks of the application to be reported as outliers in an iteration if even one rank is delayed due to communication overhead. The net effect is that the measured time of each rank is equal to the maximum time of all the ranks since synchronization has already occurred within the HPCG and LAMMPS applications at the time of measurement. Instrumenting and estimating variation using the bulk-synchronous communication points in the applications themselves would potentially address this.

As seen in Figures 3.11 - 3.12, a consequence of this is increased sensitivity of the parametric methods to tail behavior and larger predicted confidence intervals and variation than observed. This may happen more frequently on larger systems whose networks are more likely to experience performance variations; in the specific case of the presented results, Cori is much larger than Attaway. Hence, the likelihood of having large outliers due to internal synchronization barriers within HPCG is higher on Cori. This results in an overestimation of variability on Cori, compared to the accurate prediction of variability on Attaway for the HPCG workload. Similar behavior was observed for both parametric methods, but only the intra-node results are provided for brevity. Note that LAMMPS was run at 1728 ranks instead of 1024 ranks to satisfy exact weak scaling at 64000 atoms per rank.

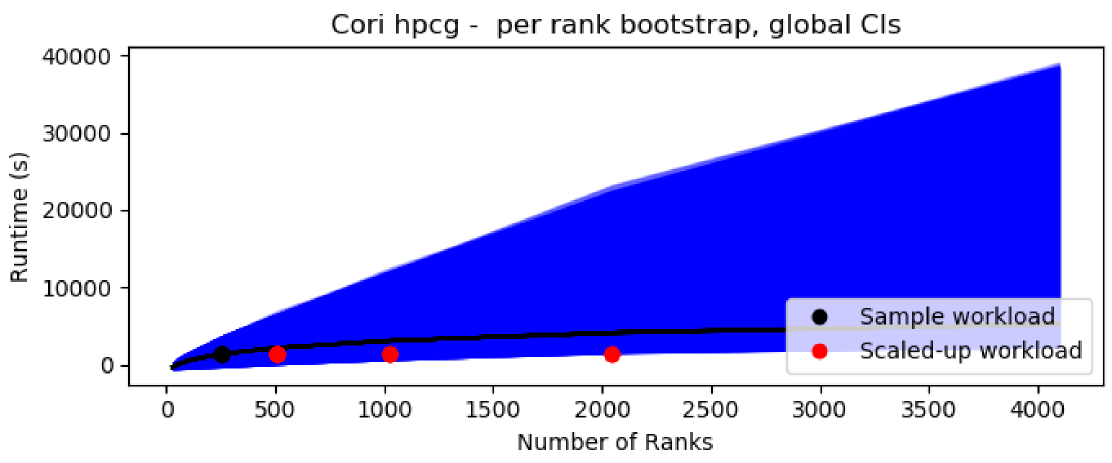


Figure 3.11: Parametric intra-node prediction of performance variation of Cori HPCG workload.

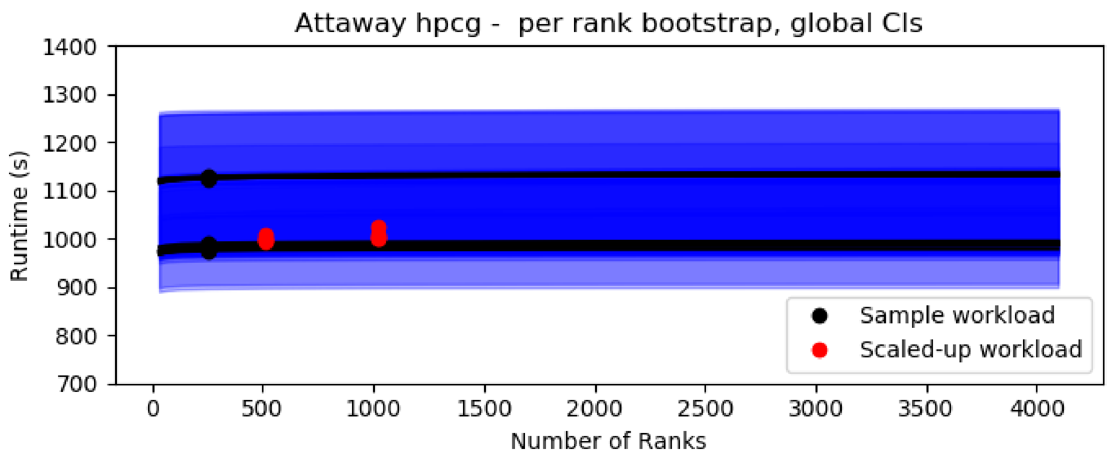


Figure 3.12: Parametric intra-node prediction of performance variation of Attaway HPCG workload.

The non-parametric prediction method exhibits a similar behavior, but to a lesser degree. Rather than utilizing GEV estimation and iteration runtime data from each rank, it simply re-samples the maximas. The result is that the tail is still amplified due to the internal communication and synchronization overhead within HPCG and LAMMPS, but the tail is not overcounted from each rank since the non-parametric re-sampling method discards all non-maxima data. The end result is Figures 3.13 - 3.14, where the Cori prediction still overpredicts runtime due to the communication and synchronization overhead, but by a much lesser degree than the parametric methods. Similarly, since Attaway is a quieter system it is less affected by internal network overheads and still provides a relatively accurate characterization and prediction of performance variation.

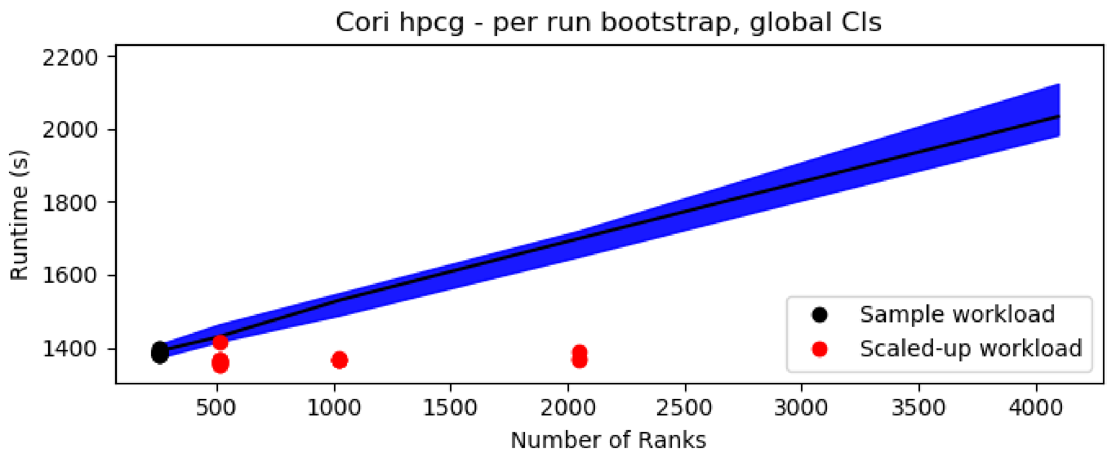


Figure 3.13: Non-parametric re-sampling prediction of performance variation of Cori HPCG workload.

We observe a similar pattern when evaluating the quantification and prediction of performance variation on the LAMMPS workloads. The parametric methods overestimate the performance variability, resulting in large confidence intervals. The inter-node method also results in non-physical negative runtime predictions as seen in Figure 3.16, since the GEV estimation is unable to negotiate the amplified tail. In-

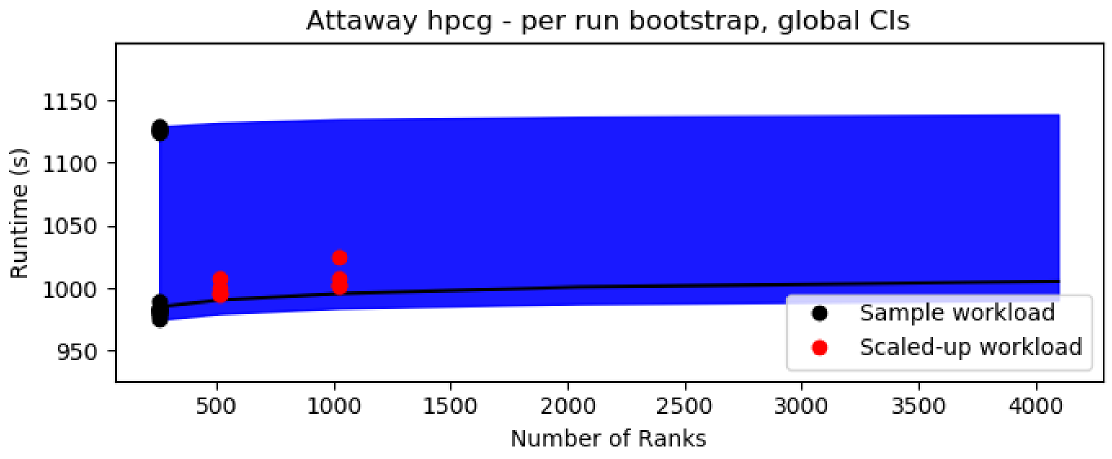


Figure 3.14: Non-parametric re-sampling prediction of performance variation of Attaway HPCG workload.

creasing the granularity of GEV estimation to per-rank data alleviates this problem (Figure 3.17) as the GEV estimation is able to provide higher fidelity parameters. On the other hand, the non-parametric re-sampling method slightly overestimates performance variability, but does an acceptable job of capturing and predicting the performance variation of the LAMMPS workload. It does this despite the internal communication and synchronization overhead present within the LAMMPS workload. Even though the parametric methods overestimate confidence intervals for workloads like HPCG and LAMMPS, the predicted medians are still very close to the actual scaled-up workloads. Following the discussion of assessment of performance variation, the insight provided by the GEV parameter estimation can be used to determine when these confidence intervals are likely to be overestimated.

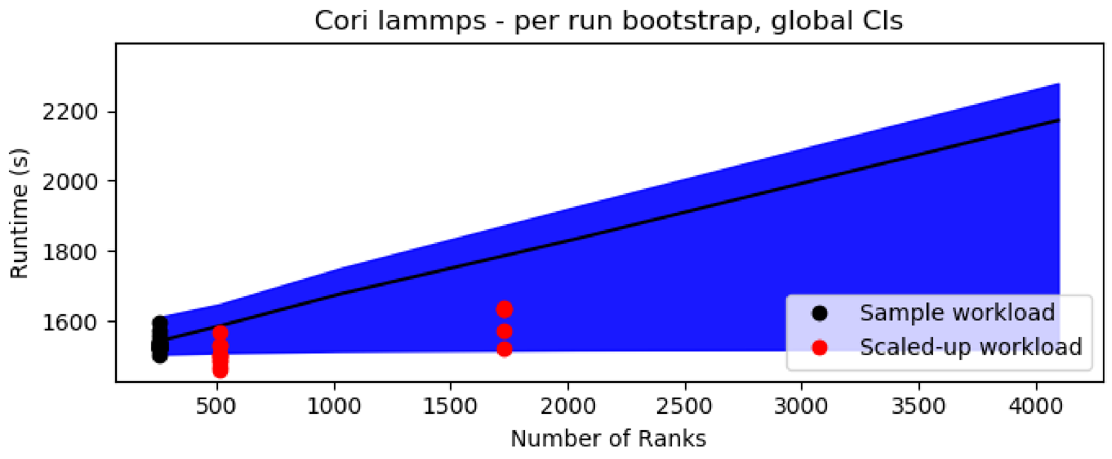


Figure 3.15: Non-parametric re-sampling prediction of performance variation of Cori LAMMPS workload.

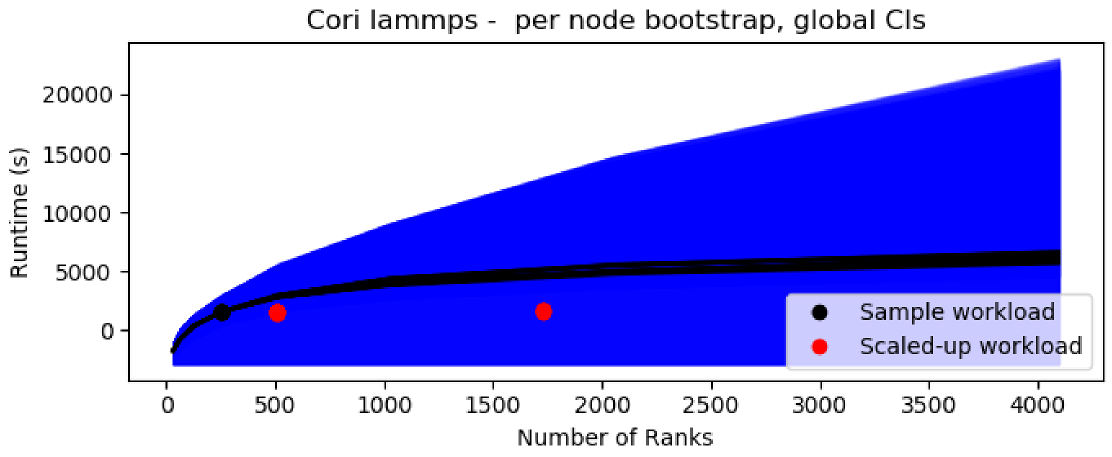


Figure 3.16: Parametric inter-node prediction of performance variation of Cori LAMMPS workload.

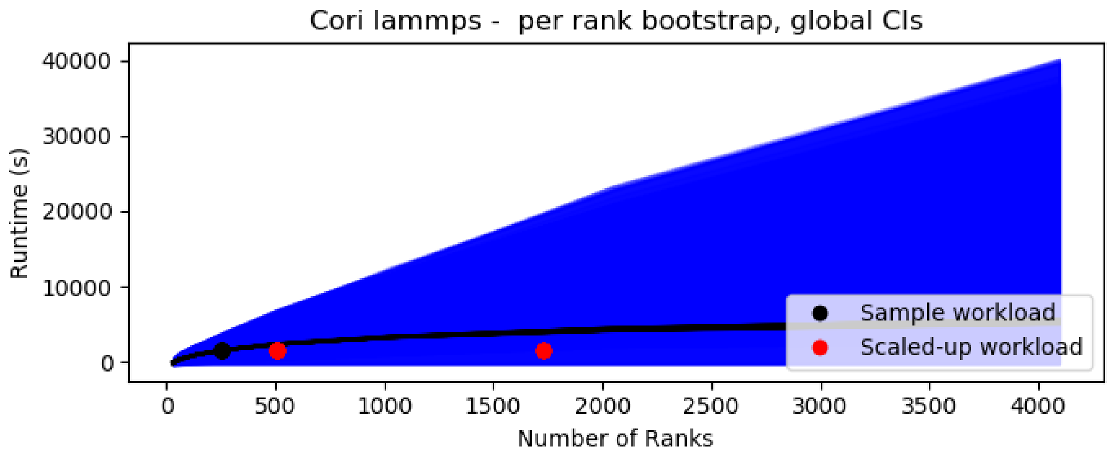


Figure 3.17: Parametric intra-node prediction of performance variation of Cori LAMMPS workload.

3.4.4 Discussion

The performance of the parametric and non-parametric methods on these six workloads performed on two different systems leads to the following conclusions:

- The performance variation of trivial and controlled workloads, such as FTQ and FWQ, can be accurately modeled and predicted by parametric and non-parametric methods.
- Compute-bound and memory-bound kernels without any local (e.g. halo) communication can be accurately measured, and their performance variation accurately quantified and predicted using both parametric and non-parametric methods.
- Parametric methods that rely on GEV estimation and either node or rank level granularity of data rather than iteration maxima granularity of data overestimate performance variation on workloads with internal communication and synchronization overhead. However, their median estimates remain accurate.

- Parametric methods can still be useful in evaluation and predicting performance variation on workloads with internal communication and synchronization overhead if and only if the network behaves predictably and with minimal fluctuations.
- Non-parametric methods can provide a technique to characterize and predict real workloads with internal communication and synchronization overhead with less overestimation than parametric methods.

3.5 Summary

This chapter introduced a new approach to quantifying and measuring performance variation in HPC applications and systems. Overall, the combination of parametric and non-parametric bootstrapping with a maxima-based approach to measuring, quantifying, and predicting performance variation significantly reduces the scale of measurement and analysis required compared to methods which require handling and integrating performance variation from individual nodes or processes. The parametric and non-parametric bootstrap methods examined for predicting how performance variation scales each had different strengths. The non-parametric bootstrap is more effective at predicting applications with complex communication behaviors. The parametric bootstrap can quantify application variance and scaling while using fewer parameters and sample data points, but is less accurate for complex applications whose communications can violate GEV distribution modeling assumptions.

In summary, these approaches can quantify and predict the performance variation of applications and HPC systems at scale while reducing the required amount of measurement for modeling and analysis. Furthermore, this chapter analyzed and discussed the trade-offs between parametric and non-parametric methods, specifically relating to their predictive capability across workloads of varying complexity.

Chapter 4

Partitioned Communication Model

4.1 Introduction

This chapter describes the development and performance of a statistical model that captures the expected performance of partitioned communication with the Finepoints approach. In doing so, the benchmarked performance results of partitioned communication on the Lassen [59] and Quartz [60] supercomputers are presented and compared to the expected performance computed from the statistical model. This chapter presents the following contributions:

- A novel statistical performance model for the proposed Finepoints partitioned communication approach (Section 4.3).
- A partitioned communication benchmark (Section 4.4.1).
- A demonstration of the predictive capability of the performance model on two supercomputing platforms (Section 4.4.3).

The remainder of this chapter is outlined as follows. Section 4.2 describes the

Finepoints proposal and the proposed semantics of partitioned communication. The development and implementation of the partitioned communication performance model is discussed in Section 4.3 along with an example calculation. Section 4.4 describes the developed partitioned communication benchmark and how the model compared to empirical benchmark results. Finally, Section 4.5 summarizes this work.

4.2 Background

This section describes the statistical and modeling background that was leveraged in the development of the partitioned communication performance model. Section 4.2.1 contrasts the typical single-send approach with the proposed Finepoints partitioned communication approach. Afterwards, an example (Example 4.2.1) calculation of single-send performance is provided to later contrast with an example calculation of the predicted partitioned communication performance (Example 4.3.1) of the proposed partitioned model in Section 4.3.

4.2.1 Single-Send and Partitioned Approach

Figure 4.1 depicts the typical single-send approach that is commonly used with MPI+X. Each thread begins a compute task at an initial time on the partition of data that it owns, and each thread may complete slower or faster than other threads due to load imbalance or other sources of noise. The threads then wait for the slowest thread to finish its compute task before synchronizing and joining. Only then will the communication initiate, where all of the data computed on by all of the threads are aggregated and sent as one large message. Alternatively, in the proposed Finepoints partitioned approach [40], each thread marks its data partition as ready and the MPI runtime can then initialize the send of the small partition of data while other threads complete (Figure 4.1). The overall effect is that the network does not spend as much time idle in hopes of eliminating the overhead caused by imbalanced thread compute times. This results in multiple small messages being sent, so while the overall time on the network might be increased, the time until all data is sent is reduced because of the ability to overlap communication with computation.

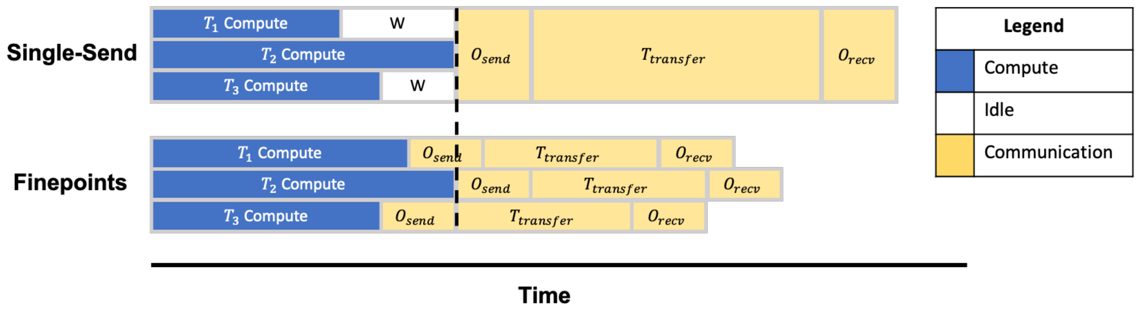


Figure 4.1: Top: typical single-send model used in MPI+X codes. Bottom: proposed partitioned (Finepoints) model for MPI+X codes. Adapted from [40].

Example 4.2.1 walks through a simple performance calculation of the single-send approach with $N = 4$ threads sharing a compute load, characterized by normally distributed runtimes with a mean of 100,000 microseconds with 10% runtime standard deviation, before sending a 16 megabyte message. The Postal model is used to calculate the message transmission time of the single send, while EMMA is used to calculate the average runtime of the slowest of the 4 threads given this distribution of runtimes.

Example 4.2.1. As an example, consider the case where 4 threads are being used to compute and send 16 megabytes of data on a system with a constant latency (not a function of message size) of 5 microseconds and a constant bandwidth of 10,000 MB/s, to simplify the calculation. The completion time of these threads is characterized by a normal distribution with a mean of 100,000 microseconds and 10% runtime standard deviation, giving a variance of 1,000 microseconds. A time of last thread completion time of 101,121 microseconds is calculated using EMMA given this distribution and 4 threads. Since the normal distribution is symmetrical, this corresponds to an expected finish time of the fastest (first) thread of 98,878 microseconds, for an expected total spread between the maximum and minimum thread completions times of 2,243 microseconds.

In the typical single-send case, one large message is sent that is 16 megabytes in size once all threads finish (at 101,121 microseconds). Using the Postal model and assuming $O_{send} = O_{recv} = 0$, such that these overheads are fully captured by the Postal model, the expected communication time can be calculated from Equation 2.12. For this message size, the calculation yields:

$$T_{send} = O_{send} + T_{transfer} + O_{recv} = \alpha + \frac{s}{\beta} = 1605\ microseconds \tag{4.1}$$

Hence, the single-send model is expected to complete at a time of 102,726 microseconds, with an elapsed time of 1605 microseconds from slowest thread completion to communication completion ($T_{elapsed} = T_{send}$, since there is no overlap between communication and computation), and an effective bandwidth of 9,969 MB/s (or 10,453 MiB/s) calculated from Equation 4.2.

$$BW_{eff} = \frac{Buffer\ Size}{T_{elapsed}} = \frac{16\ MB}{1605 * 10^{-6}\ seconds} = 9,969\ MB/s \tag{4.2}$$

In the Finepoints case, up to four small messages are sent. Assuming the data is split evenly and that four messages are sent, each message is then only 4 megabytes. Again, using the Postal model, the expected communication time for each message is found to be $T_{send} = 405\ microseconds$. As will be seen later, it might be expected that a significant chunk of data will have already been sent by the time the last thread finishes its compute, since partitioned communication allows this overlap of computation and communication, yielding better performance. Example 4.3.1 provides a detailed calculation of how this improvement in performance is calculated and predicted using the model described in Section 4.3.

4.3 Performance Model

As described in Section 4.2.1, the Finepoints partitioned communication [40] approach attempts to minimize thread waiting time by overlapping communication and compute. While Finepoints defines the API exposed to application developers, it does not specify lower-level details of implementation or optimization techniques that may be used. For example, it may be of interest to agglomerate certain partitions together to optimize performance, or to schedule sends from each thread given knowledge of the gaps between thread completion times. These are areas of future work, both in terms of modeling and implementation.

Given this information, I developed a model using a set of simplifying assumptions:

- Data is partitioned as evenly as possible between threads.
- Each thread individually sends its partitioned data as a single message. If there are 10 threads working on a buffer, there will be 10 messages sent, each of equal size (1/10th the size of the entire buffer).
- Thread runtimes are distributed normally or as any other symmetrical distribution. Extending the model to other distributions is left as future work.
- Message transmission times can be described by the Postal model of communication [7, 47].

- An optimistic sending assumption: data transmission will be assumed to begin as soon as the fastest thread finishes its compute and will proceed continuously until the slowest thread finishes its compute (i.e. maximum overlap between computation and communication). The remaining data will then be assumed to send as individual messages after the completion of the compute phase of the slowest thread.
- At least one message will remain unsent at the time that the slowest thread finishes its compute workload (i.e. the slowest thread cannot send data until it finishes its compute).

These assumptions allowed the development of a basic statistical model which utilized the EMMA method [82] for calculating the expected maximum time of the slowest thread. The symmetrical distribution assumption allowed the EMMA result to be extended to calculate the average spread between the runtimes of the fastest and the slowest thread. The optimistic assumption provided the basis for calculating the average amount of data able to be sent during the overlap period, while the last remaining message assumption assured that order of operations was preserved. Visually, this model can be summarized by Figure 4.2.

Figure 4.2 shows $n = 7$ threads with normally distributed runtimes. Thread 4 is the slowest on average and thread 7 is the fastest. The average runtime of thread 4 is determined with the EMMA method (Equation 2.9) while the average runtime of thread 7 is determined with the symmetry assumption. Once thread 7 completes, messages begin to be sent in a serialized fashion, with the time each message takes calculated using the Postal model (Equation 2.12). The Postal model is given bandwidth and latency values from the OSU microbenchmark [73] as inputs in addition to the partitioned size of the message (calculated from the assumption that data is evenly distributed amongst threads). This continues until all messages have been sent and is followed by a small (but significant at smaller message sizes)

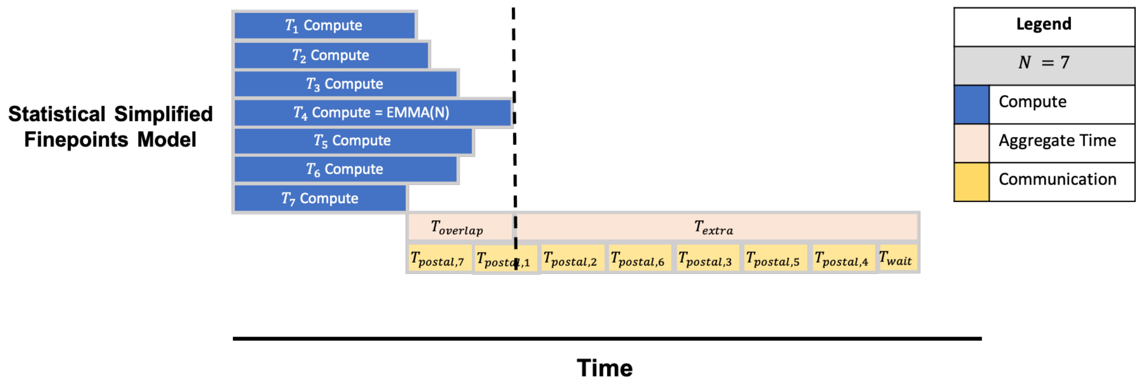


Figure 4.2: Simplified visualization of the Finepoints communication approach that is approximated in the developed Finepoints communication model.

overhead of calling `MPIX_Wait`. At this point, the time elapsed from the completion of the final thread's compute to the completion of the entire partitioned send and the effective bandwidth (Equation 4.3) can be calculated. This process is described formally in Algorithm 1.

Given input latencies and bandwidths from the OSU microbenchmark [73], the number of threads utilized, the total sizes of the buffers to be sent, the average thread compute time, the runtime variability (or standard deviation), and the overhead of `MPIX_Wait` (on the order of 10s of microseconds), Algorithm 1 calculates the expected effective bandwidth as a function of total buffer size.

Example 4.2.1 is continued below as Example 4.3.1 and walks through the model calculation of the predicted performance of a partitioned send compared to the previously shown calculation for single-send performance. Again, the bandwidth and latency are assumed to be constant and not a function of message size for simplicity.

Example 4.3.1. Continuing Example 4.2.1 with $N = 4$ threads performing a computation on a 16 megabyte buffer ($M_{size} = 4\ megabytes$) that they will eventually send, where runtimes are normally distributed with a mean of $\mu = 100,000\ microseconds$ and a variance of $\sigma = 1,000\ microseconds$ (i.e. a 10% runtime standard deviation):

Algorithm 1 Modeling effective bandwidth in Finepoints.

```
    Input: $osu_{Lat}[]$, $osu_{BW}[]$, $threads$, $sizes[]$, $T_{compute}$, $variability$, $T_{wait}$
    Output: $bw[]$: Effective bandwidth
$N = threads$, $M_{size} = size/N$
$\mu = T_{compute}$, $\sigma = T_{compute} * (variability^2)$, $D = Dist_{Norm}(\mu, \sigma)$
for size in sizes do
    $T_{message} = postal(M_{size}, osu_{Lat}(M_{size}), osu_{BW}(M_{size}))$
    $T_{lastThread} = EMMA(D, N)$
    $T_{overlap,avg} = 2 * (T_{lastThread} - \mu)$
    $M_{overlap,avg} = min(T_{overlap}/T_{message}, N - 1)$
    $T_{extra} = T_{message} * (N - M_{overlap,avg}) + T_{wait}$
    $E = size/T_{extra}$
    $bw.append(E)$
end for
```

The expected spread between the fastest finishing thread at 98,878 microseconds and the slowest finishing thread at $T_{lastThread} = 101,121$ microseconds is $T_{overlap,avg} = 2,243\ microseconds$. Given $T_{message} = 405\ microseconds$, as calculated from the Postal model (Equation 2.12) with a latency of 5 microseconds, a bandwidth of 10,000 MB/s, and a message size of $M_{size} = 4\ megabytes$, the average number of messages able to be sent during the overlapping period from 98,878 microseconds to 101,121 microseconds can be calculated, $M_{overlap,avg} = 3$. This indicates that 3 threads will be able to complete their compute and also post and complete their sends by the time the slowest thread completes it's compute. At the point when the last thread finishes it's compute at 101,121 microseconds, only the slowest thread will need to send data, resulting in a single additional message of 4 megabytes being sent. This results in the partitioned send completing at 101,526 microseconds with $T_{extra} = 405\ microseconds$, for an improvement of about 1200 microseconds from the single-send approach. This would result in a predicted effective bandwidth of

$E = 39,506\ MB/s$ (or, converted to mebibytes per seconds $E = 41,425\ MiB/s$).

There will be cases where more than one message will need to be sent after the completion of the slowest thread. As message size increases and more time is spent communicating, one would expect that the spread between the first and last thread times would not be adequate to send a majority of the data, and the partitioned send will collapse back to the single-send performance at very large message sizes. Additionally, as seen in Example 4.2.1, the partitioned send spends more time communicating ($405 * 4 = 1620\ microseconds$) than the single-send would ($1605\ microseconds$) due to multiple start-up costs from sending multiple smaller messages. However, this increase in overall communication time is offset by the overlap in computation and communication that partitioned communication allows, resulting in better network utilization.

4.4 Evaluation

The following section is organized as follows. Section 4.4.1 describes the benchmark developed to measure Finepoints performance which was then compared with modeled performance. Section 4.4.2 describes the selection of MPI settings on the system of interest to run the benchmark and discusses potential causes of performance variation across MPI modes, features, and launch mechanisms. Finally, Section 4.4.3 evaluates the predictive capability of the partitioned communication model on the Lassen and Quartz supercomputers at two runtime standard deviation levels and discusses potential sources of error.

4.4.1 Benchmark

The developed benchmark, which collects empirical data, uses a library called the MPI Partitioned Communication Library (MPIPCL), that implements the Finepoints API calls using existing MPI functionality. The developed benchmark utilizes 2 MPI ranks, where the root rank sends data to the second rank and the second rank receives the data from the root rank, similar to a typical MPI ping-pong test to measure latency and bandwidth. While partitioned communication can be used on both the send and receive ends of a communication channel, this particular benchmark only uses partitioned communication on the send side. The benchmark took the following inputs: number of threads, the size of the buffer to send, average compute time of each thread, and thread runtime variance level. After performing 5 warm-up loops, it then initializes the partitioned send with a call to `MPIX_Psend_init` and `MPIX_Start`, then performs normally distributed synthetic computes, simulated by a sleep, across the threads (i.e. each thread slept for an amount of time that was randomly sampled from the normal distribution defined by the average compute time and the amount of runtime variance) before performing the partitioned send by calling `MPIX_Pready` once each thread completes its synthetic compute. After the threads join, the benchmark checks message completion by calling `MPIX_Wait`. The benchmark performs a measurement of the time between the last completing thread (i.e. when the slowest throw finishes its synthetic compute) and the completion of the `MPIX_Wait`. The benchmark uses OpenMP as the shared-memory threading model and requires an OpenMP reduction to record the time of the last completing thread. The benchmark then performs the final calculation of effective bandwidth as shown in Equation 4.3, where BW_{eff}, represents the calculated effective bandwidth, t_{last} represents the measured completion time of the slowest thread, t_f represents the measured time after `MPIX_Wait` returns indicating all messages have completed, and

s represents the total size of the buffer.

$$BW_{eff} = \frac{s}{t_f - t_{last}} \tag{4.3}$$

Algorithm 2 outlines the implementation of this process to describe how the benchmark is used to perform effective bandwidth measurements. The plots presented in this section visualize the effective bandwidth BW_{eff} calculated from Equation 4.3 and the time elapsed between the slowest thread completing its compute and the completion of all communication (represented as $t_{elapsed} = t_f - t_{last}$).

4.4.2 Performance Sensitivity Analysis

I performed an initial analysis of Finepoints performance on Lassen [59] using many of the different options available for MPI and launching parallel jobs on the Lassen system. This was done to characterize the variation observed as options were toggled and to understand which options corresponded to or break various assumptions of the performance model. Since the Lassen supercomputer uses SpectrumMPI [53], tunable options, such as turning on hardware tag matching [28] (`-hwtm`) and enabling the asynchronous progress thread (`-async`), were available by launching the benchmark using the typical call to `mpirun` with the appropriate flags. Additionally, Lassen uses IBM's lsf [52, 61] scheduler and corresponding `lrun` command to simplify the launching of parallel jobs, and hence does not provide the ability to control whether hardware tag matching or the asynchronous progress thread are enabled through the use of flags. Finally, partitioned communication through Finepoints can be run with both of the multi-threaded modes from MPI, `MPI_THREAD_SERIALIZED` and `MPI_THREAD_MULTIPLE`. Note that only the hardware tag matching flag was enabled for the `MPI_THREAD_SERIALIZED` mode. This resulted in 8 possible combinations of options, launch methods, and MPI modes, listed below:

Algorithm 2 Finepoints performance benchmark.

```
  Input: $threads$, $size$, $T_{compute}$, $variability$
  Output: $t_{elapsed}$: Elapsed time
$N = threads$, $s = size$, $M_{size} = s/N$
$\mu = T_{compute}$, $\sigma = T_{compute} * (variability^2)$, $D = Dist_{Norm}(\mu, \sigma)$
if rank == 0 then                                                    ▷ Sender
    Do Warmup Loops
    MPI_Barrier

    Initialize Partitioned Send
    parfor thread in threads do                  ▷ Do compute and partitioned send
        $sleep(D)$          ▷ Sleep for length of random sample from distribution D
        $t_{thread}$ = MPI_Wtime
        Post MPI_PReady ($M_{size}$)                  ▷ Partition is ready to send
    end parfor
    $t_{last}$ = reduction(max: $t_{thread}$)
    MPIX_Wait
    $t_f$ = MPI_Wtime
    $t_{elapsed} = t_f - t_{last}$
end if
if rank == 1 then                                                  ▷ Receiver
    Do Warmup Loops
    MPI_Barrier

    Initialize Partitioned Receive
    MPIX_Wait
end if
```

1. `mpirun`, `MPI_THREAD_SERIALIZED`, no flags
2. `mpirun`, `MPI_THREAD_SERIALIZED`, `-hwtm`
3. `mpirun`, `MPI_THREAD_MULTIPLE`, no flags
4. `mpirun`, `MPI_THREAD_MULTIPLE`, `-hwtm`
5. `mpirun`, `MPI_THREAD_MULTIPLE`, `-async`
6. `mpirun`, `MPI_THREAD_MULTIPLE`, `-hwtm` `-async`
7. `lrun`, `MPI_THREAD_SERIALIZED`, no flags
8. `lrun`, `MPI_THREAD_MULTIPLE`, no flags

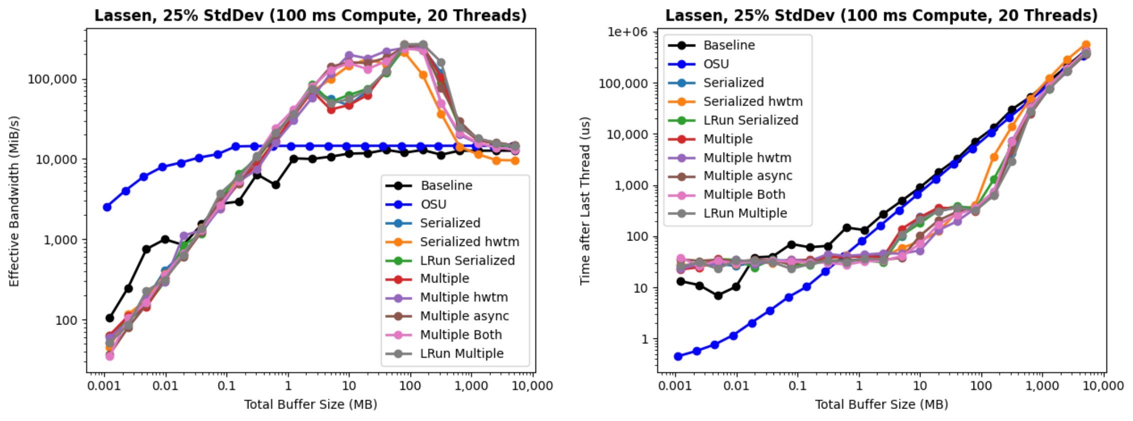
These 8 combinations were all run with the developed Finepoints benchmark on the Lassen supercomputer at Lawrence Livermore National Laboratories on a variety of total buffer sizes, ranging from 160 bytes - 5 gigabytes. The specifications of the Lassen supercomputer are shown in Table 4.2. For the current study, only a single socket per node was utilized, and two separate nodes were reserved to measure inter-node performance. Since each socket contains an IBM Power9 CPU with 20 cores, the benchmark was run with 20 threads per rank, resulting in average message sizes of 8 bytes - 256 megabytes. Finally, the compute time distribution selected was a normal distribution with a mean of 100 ms and 25% runtime standard deviation. The benchmark was run 5 times for each buffer size and the average results are presented. The effective bandwidth and elapsed time (i.e. time from last thread completion to completion of data transfer) are shown in Figure 4.3. Additionally, the results from the OSU microbenchmark (see Tables A.1 - A.2) are plotted plus a baseline MPI ping-pong test to allow visualization of when Finepoints underperforms and overperforms a typical send and receive.

Table 4.1: LLNL Lassen supercomputer CPU specifications (excluding GPUs) [59].

CPU Architecture	Total Nodes	Sockets/Node	Cores/Node	Total Cores	CPU Clock Speed	Peak Performance (CPU)
IBM Power9	795	2	44	34,848	3.5 GHz	855 TFLOPS

Starting at total buffer sizes of 0.25 megabytes, Finepoints begins to see substantial effective bandwidth performance improvements over what would be expected from the OSU benchmark and what is observed during a typical MPI ping-pong test. This continues until Finepoints performance reduces back to the expected OSU performance for large message sizes when bandwidth completely dominates. This performance improvement peaks above 200,000 MiB/s with total buffer sizes between 72 and 144 megabytes (average message sizes of 3.6 - 7.2 megabytes with 20 threads), compared to the 14,000 MiB/s expected from OSU and a single-send MPI ping-pong test, for a factor of 15 improvement in effective bandwidth. Similarly, at this buffer size, the elapsed time decreased from 11,500 microseconds in the single-send ping-pong test to only 800 microseconds in the Finepoints partitioned benchmark. These aggregate results are consistent with previously published results run on a Cray system [40].

Noticeably, when the benchmark was run in `MPI_THREAD_MULTIPLE` mode without any additional flags, in addition to both runs initiated with `lrun` (no flags were able to be configured), a slight dip in performance was observed after an early peak at total buffer sizes around 2.25 megabytes. I hypothesize this difference is due to the overhead MPI spends searching through send and receive queues to perform tag matching when in the `MPI_THREAD_MULTIPLE` mode, as has been documented previously when using multi-threaded MPI without hardware tag matching [28, 29, 65, 77] or any other optimizations. Simply using `mpirun` allows us to utilize hardware tag matching with the `-hwtm` flag and this appears to alleviate the performance issue. Finally, the asynchronous progress thread does not appear to significantly affect performance once hardware tag matching has been enabled, and it appears that enabling



(a) Calculated effective bandwidth.

(b) Elapsed time, in microseconds.

Figure 4.3: Finepoints benchmark results on Lassen with a 100 ms workload, 20 threads, and 25% runtime standard deviation. The benchmark was launched with `MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE`, with `mpirun` or `lrun`, and with and without hardware tag matching and asynchronous progress thread. Right: effective bandwidth. Left: elapsed time between the completion of data transfer and the completion of last thread compute.

only one of these flags is able to alleviate the performance issue. Therefore, since the developed model does not consider overheads that MPI may incur from searching through receive and send queues to perform tag matching, `mpirun` is used to launch the benchmark with the hardware tag matching flag enabled and the MPI threading mode is selected to be `MPI_THREAD_SERIALIZED` for all subsequent runs to compare measured performance with modeled performance.

4.4.3 Model Performance

To evaluate and demonstrate the predictive capability of the model presented in Section 4.3, I run the benchmark at two runtime standard deviation levels, 10% and 25%, on a 100 millisecond, normally distributed, synthetic compute with various buffer sizes on the Lassen and Quartz supercomputers. Lassen benchmarks were run with 20 threads per rank and Quartz benchmarks were run with 18 threads per rank. Each test was performed on 2 ranks on separate nodes (each rank using the cores on a single socket with a 1:1 thread to core mapping) to test the inter-node performance. The specifications of the Lassen supercomputer are provided again in Table 4.2 for convenience, along with the specifications of the Quartz supercomputer in Table 4.3. As discussed in Section 4.4.2, this was done with the MPI threading mode set to `MPI_THREAD_SERIALIZED` and with hardware tag matching enabled by launching with `mpirun` rather than `lrun` in order to avoid the overhead of searching through receive and send queues to perform tag matching. The model was then run with the corresponding inputs of 20 threads, 10% and 25% runtime standard deviation on a 100 millisecond normally distributed compute workload, and inter-node OSU microbenchmark data gathered on the Lassen and Quartz supercomputers (Table A.1) as inputs. The output effective bandwidth and elapsed time between last thread completion and completion of data transfer of the model was compared with that measured by the benchmark and plotted as a function of total buffer size.

Table 4.2: LLNL Lassen supercomputer CPU specifications (excluding GPUs) [59].

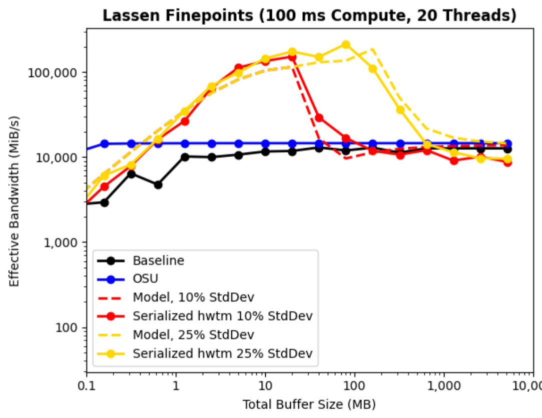
CPU Architecture	Total Nodes	Sockets/Node	Cores/Node	Total Cores	CPU Clock Speed	Peak Performance (CPU)
IBM Power9	795	2	44	34,848	3.5 GHz	855 TFLOPS

Evaluation on Lassen

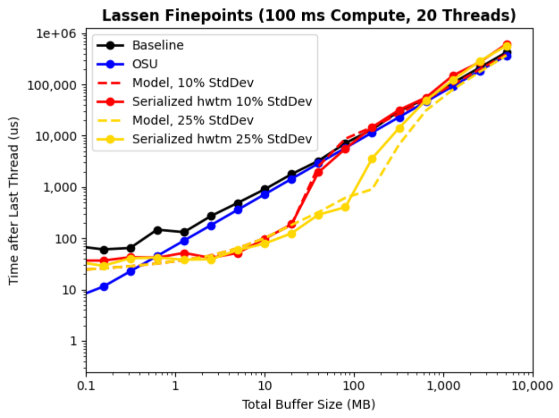
The results for Lassen are shown in Figure 4.4, in addition to the measured OSU bandwidth (blue) and the single-send MPI ping-pong bandwidth (black, labeled as Baseline).

Table 4.3: LLNL Quartz supercomputer specifications [60].

CPU Architecture	Total Nodes	Sockets/Node	Cores/Node	Total Cores	CPU Clock Speed	Peak Performance
Intel Xeon E5-2695 v4	3,018	2	36	108,648	2.1 GHz	3,251.4 TFLOPS



(a) Effective bandwidth comparison.



(b) Elapsed time comparison.

Figure 4.4: Finepoints benchmark results on Lassen, with a 100 ms workload, 20 threads, and 10% and 25% runtime standard deviations, compared to the model predicted results. Lassen OSU bandwidth and single-send MPI performance are shown as baselines. Right: measured vs. modeled effective bandwidth. Left: measured vs. modeled elapsed time between the completion of data transfer and the completion of last thread compute.

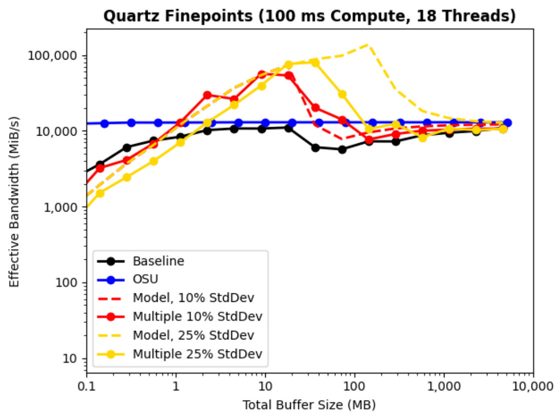
As seen in Figure 4.4, the model predicts the region of overperformance compared to the single-send MPI approach for both runtime standard deviation levels of 10% and 25%. At 10% runtime standard deviation, it predicts the overperformance to occur between total buffer sizes of 0.56 megabytes and 72 megabytes (average message sizes of 0.028 megabytes and 3.6 megabytes with 20 threads), while overperformance actually occurs from total buffer sizes of 0.56 megabytes to 144 megabytes. Additionally, the model predicts peak performance with 10% runtime standard deviation to occur at a total buffer size of 18 megabytes (average message size of 0.9 megabytes) and with an effective bandwidth of 115,000 MiB/s, while the measured peak performance also occurred at a total buffer size of 18 megabytes with an average effective bandwidth of 152,000 MiB/s. Over the course of 5 runs, this peak bandwidth measured at a total buffer size of 18 megabytes ranged from 53,000 MiB/s to 211,000 MiB/s, indicating that the model was able to predict both the location of peak performance and the relative magnitude of the performance improvement within the expected variation of a real system.

The corresponding drops in elapsed time compared to the expected elapsed time using a single-send MPI approach is also visible in Figure 4.4b, forming a corresponding overperformance "bubble" in the elapsed time figure. Note that this large variation in empirically measured effective bandwidth from run-to-run is due to the randomized amount of spread present in the thread runtime distribution affecting performance and expected noise within the system. For example, if the spread between the runtimes of the fastest thread and slowest thread happens to be smaller on one run and the next run observes a larger spread, more overlap in communication and computation would be expected to occur in the second run and a higher effective bandwidth would be the result. However, on average over many runs, the effective bandwidth would be expected to be well-represented by the model. Finally, the overall trend between the modeled effective bandwidth (and the resulting elapsed time) matches the empirical data with 10% runtime standard deviation and should provide robust predictions of communication performance.

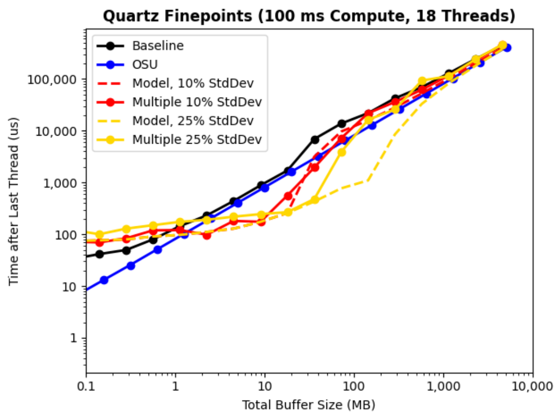
Next, at the 25% runtime standard deviation level, a longer and higher peak of performance was measured to occur than the workload with 10% runtime standard deviation. The modeled performance appears to capture this shift in performance well. The region of overperformance in the measured data now encompasses a region of total buffer sizes from 0.56 megabytes up to 2,304 megabytes (average message size of 0.026 megabytes to 115.2 megabytes), while the model predicted a region of overperformance from 0.28 megabytes up to 4,608 megabytes. This larger region of overperformance can be explained by the increased spread in thread runtimes due to increased runtime variability, allowing more communication to be overlapped with computation, even at large message sizes. This results in the overall bandwidth of the system (represented by the OSU bandwidth) only becoming a significant limiting factor at very large message sizes. This also results in a higher peak of effective bandwidth, increasing from a measured average of 152,000 MiB/s at 10% runtime standard deviation to a measured average of 214,000 MiB/s at 25% runtime standard deviation (with a range of 153,000 MiB/s to 233,000 MiB/s). This peak also occurs at a larger total buffer size of 72 megabytes, compared to 18 megabytes in the 10% case. At 25% runtime standard deviation, the model predicts a peak of 186,000 MiB/s at a total buffer size of 144 megabytes (it predicts an effective bandwidth at 136,000 MiB/s at a total buffer size of 72 megabytes). These peaks and the region of overperformance can be seen in the elapsed time figure (Figure 4.4b) as well, and the relationship between effective bandwidth and elapsed time is previously defined in Equation 4.3. Again, the overall trend between the modeled and measured effective bandwidth provides a useful tool to allow prediction of overall Finepoints performance in addition to predicting the regions where Finepoints partitioned communication would be expected to outperform a single-send communication approach. The model achieves this only given the number of threads, the OSU benchmark results of the system, and the expected distribution of thread compute times.

Evaluation on Quartz

Figure 4.5 shows the comparison of empirical results with the performance model results on the Quartz supercomputer system [60], run with 18 threads and a normally distributed 100 millisecond compute with 10% and 25% runtime standard deviations. Note that Quartz does not use SpectrumMPI, but mvapich2 [83] instead. Quartz also utilizes the slurm scheduling system [91], giving users access to the `srun` command to launch parallel jobs, but does not allow `mpirun` to be used with mvapich2, nor are command line flags provided to enable hardware tag matching and the asynchronous progress thread. Additionally, the mvapich2 (version 2.3) installation on Quartz only carries the `MPI_THREAD_MULTIPLE` mode and not the `MPI_THREAD_SERIALIZED` mode. These limitations allowed the exploration of model performance with confounding variables and provided a basis for a discussion about model assumptions, their effects, and how the model could be refined in the future to capture the variabilities seen between MPI implementations, options, and various HPC systems and networks.



(a) Effective bandwidth comparison.



(b) Elapsed time comparison.

Figure 4.5: Finepoints benchmark results on Quartz, with a 100 ms workload, 18 threads, and 10% and 25% runtime standard deviations, compared to the model predicted results. Quartz OSU bandwidth and single-send MPI performance are shown as baselines. Right: measured vs. modeled effective bandwidth. Left: measured vs. modeled elapsed time between the completion of data transfer and the completion of last thread compute.

As can be seen in Figure 4.5, the model at 10% runtime standard deviation predicted the region of overperformance and the peak of the partitioned communication performance well. The improved performance was modeled to occur from a total buffer size of 1.13 megabytes to 72 megabytes (average message size of 0.063 megabytes to 4 megabytes), with peak performance occurring at a total buffer size of 9 megabytes and an effective bandwidth of 54,200 MiB/s, while empirically, the improved performance window ended at a total buffer size of 36 megabytes and had a peak performance of 56,400 MiB/s (ranging from 37,000 MiB/s to 65,000 MiB/s) also at a total buffer size of 9 megabytes. This indicates great predictive performance.

However, the model at 25% runtime standard deviation significantly overpredicted the performance benefit of Finepoints on large buffer sizes, predicting that improved performance will continue up to a buffer size of 2,304 megabytes when it really only continues until 288 megabytes. Similarly, the peak prediction is skewed and the model overpredicts peak performance. However, good agreement between the modeled bandwidths and the observed bandwidths are observed until a total buffer size of 36 megabytes, where the model begins to diverge from the observed.

The current hypothesis is that several factors may have played a role in this difference, such as the use of mvapich2 rather than SpectrumMPI, the inability to explicitly control hardware tag matching with mvapich2 (which may hurt performance substantially as seen in Figure 4.3 and discussed in Section 4.4.2) or the usage of `MPI_THREAD_MULTIPLE` rather than `MPI_THREAD_SERIALIZED`. This performance model assumes no queueing times in the network and that messages are sent in a serialized fashion from the moment the fastest thread finishes its computation, even though communication systems and hardware have much more complexity. Finally, the assumption that messages are sent continuously from the moment the fastest thread finishes may be inaccurate if the gaps between thread finish times are regularly longer than the time it would take to send a single slice of the partitioned buffer, resulting in slower data transfer and the lower effective bandwidths observed. Sources of error and directions of future work to improve the ability of the model to account for differences in MPI implementations, network characteristics, and MPI optimizations is discussed further in Section 5.2.

4.5 Summary

This chapter has introduced a partitioned communication performance model and benchmark based on the Finepoints proposal to the MPI standard. This benchmark was utilized to gather performance data from the Lassen and Quartz supercomputers, and then the performance model was evaluated against this gathered data. It was found that the model predicted performance on the Lassen supercomputer well at two separate runtime variability levels, in terms of peak performance, location of peak performance, and buffer sizes at which partitioned communication can be expected to outperform a single-send MPI approach. MPI parameters and optimizations were enabled on the SpectrumMPI implementation used by Lassen to satisfy underlying assumptions of the performance model. The model was also found to provide predictive capability on Quartz at low levels of runtime variability, but over-predicted performance at high runtime variabilities. Finally, the importance of MPI optimizations, parameters, and network characteristics were discussed as considerations for future directions of this work to address the predictive performance of the model on Quartz and other systems that do not provide a high level of flexibility of MPI parameters.

Chapter 5

Conclusions

This thesis developed and demonstrated the ability of lightweight statistical models to characterize and predict the performance of various aspects of distributed and parallel programs on HPC systems. In this chapter, I summarize this work, its results and contributions, and possible directions for future work.

5.1 Summary

This thesis presents two lightweight, statistical modeling approaches to understand the performance of parallel and distributed programs on HPC systems as they become more highly threaded with many-core CPUs and GPUs and more heterogeneous in architecture. First, Chapter 3 introduced a statistical model based on general extreme value theory to characterize and project the performance variation of various parallel workloads on HPC systems [24]. Additionally, this model provided the flexibility to adjust the volume of data collected to account for the complexity of the workload by utilizing parametric and non-parametric bootstrapping approaches which leveraged the GEV distribution. I then discussed the model's trade-off between granularity of measurement and capability of projection at scale in complex workloads. The models ability to project the performance variability at scale was then evaluated on two HPC systems with six representative BSP workloads that encompassed a range of complexity.

Next, Chapter 4 discussed partitioned communication, and presented a statistical model to help understand the performance benefits expected from using partitioned communication on highly-threaded heterogeneous HPC systems. I then discussed the simplifying assumptions used to develop the performance model and explored potential model limitations. This model was then evaluated with data generated by a partitioned communication benchmark that was run on two HPC systems using a variety of settings and parameters.

5.2 Future Work

There are several directions of future work, including the development of additional statistical models to understand and to predict the performance trends of parallel scientific codes in addition to the improvement of the statistical models presented in this thesis.

5.2.1 Hybrid Parametric Models

Chapter 3 presented parametric and non-parametric approaches for understanding and projecting the expected performance and variation in the performance of workloads as they scale. These approaches were challenged by large, complex workloads that contained minor synchronization points amongst subsets of processes between major synchronization points across all processes. They were also challenged by the granularity of information gathered due to system and network noise. Additional work that further investigates the empirical performance variation of large and commonly used applications, such as LAMMPS [74], at large scale and on various platforms, would provide additional insight into the magnitude of performance variation seen by scientists and application developers in practice and would provide additional data to test the approaches developed and presented. Secondly, the development of hybrid parametric models that utilize some combination of parametric bootstrapping and non-parametric bootstrapping to tune the projections of performance variability and scalability would allow the model to characterize more complex workloads. Furthermore, such a model could provide performance anomaly detection in large scale systems. Finally, continued work on utilizing statistical and lightweight models to characterize and predict the performance and runtimes of parallel and distributed applications would provide an opportunity to optimize HPC system schedulers to best utilize available resources and minimize wasted compute time and energy as these systems continue to grow and strive to become more energy efficient.

5.2.2 Queueing Theory and Communication Models

The partitioned communication performance model presented in Chapter 4 provided useful insights on the magnitude and the regions of performance gains expected from using partitioned communication in place of typical single-send communication in MPI+X programs. Additionally, work to refine this model to consider on-node vs. off-node communication, receive-side partitioned communication, and entire halo exchanges amongst subsets of processes would allow this performance model to become even more directly applicable to full scientific applications with complex communication patterns. While the current model made many simplifying assumptions to ensure ease of use and understandability, future models may pursue different approaches and levels of complexity in order to consider information about final implementation level details of partitioned communication. MPI implementations across vendors will determine these low-level details. For example, a hypothetical future model based on queueing theory, which has previously been successfully used to model eager and rendezvous performance in an MPI ping-pong test [15], would allow the inclusion and consideration of many lower-level details of partitioned communication that may impact performance. Using a queueing model would allow the reduction of simplifying assumptions and would likely be able to capture the performance impacts of MPI implementation differences and optimizations, as well as impacts from details of network structure and hardware, in ways that a simple statistical model cannot. Development of a queueing based model, not just for partitioned communication, but for general and common communication patterns, would be an exciting area of future work that could build off of the model presented in this thesis.

5.2.3 Communication with Many-Core CPUs and GPUs

As many-core CPUs and GPUs, with hundreds or thousands of threads that implement shared-memory parallelism, continue to become the standard in HPC systems, partitioned communication and future communication techniques will only become more important. This is especially true as partitioned communication continues to be explored and hopefully implemented with GPU-triggered sends and receives. As the degree of parallelism available to exploit grows, parallel and distributed scientific codes will be able to perform computations much faster, resulting in a higher frequency of communication to exchange data. In some cases, this has already been observed to result in upwards of 10% of total run time being spent in communication libraries and routines [45]. As a result, it is paramount that the community develops models and benchmarks to understand the expected performance impacts of various communication techniques and which allow researchers to explore the trade-off spaces of different techniques. This will result in new communication primitives which application developers can leverage, and will provide tools that allow application developers and systems experts to easily understand and diagnose communication system performance. Statistical models, such as the partitioned communication model presented in Chapter 4, can provide a starting point for this future work that would allow researchers to develop and understand the expected performance of new communication techniques and allow practitioners to evaluate communication methods and techniques through simulation and modeling before choosing the best technique for their application.

Appendices

Appendix A

OSU Benchmark Data

Table A.1: OSU inter-node benchmark data.

	Lassen		**Quartz**	
Size (bytes)	**Latency (μs)**	**Bandwidth (MB/s)**	**Latency (μs)**	**Bandwidth (MB/s)**
1	1.09	3.81	1.25	3.41
2	1.09	7.52	1.26	6.93
4	1.09	15.46	1.23	14.24
8	1.11	30.64	1.22	28.79
16	1.12	59.65	1.49	52.17
32	1.16	117.70	1.50	101.67
64	1.17	233.26	1.50	210.05
128	1.26	448.30	1.53	407.20
256	1.57	772.24	1.59	798.79
512	1.70	1444.99	1.67	1479.75
1024	1.91	2439.89	1.88	2063.03
2048	2.70	3843.35	2.31	2956.79
4096	3.33	5740.88	3.08	3822.00
8192	4.85	7567.50	4.71	4769.81
16384	6.77	8557.06	6.42	4811.08
32768	9.07	9903.44	14.71	5121.45
65536	11.82	10899.60	22.05	11828.67
131072	16.91	13661.78	30.81	12023.30
262144	25.85	13777.30	38.21	12241.34
524288	44.44	13838.00	63.17	12226.83
1048576	89.99	13865.86	105.97	12244.56
2097152	157.73	13883.41	197.04	12294.29
4194304	308.25	13891.20	378.19	12328.31

Table A.2: OSU intra-node benchmark data.

	Lassen		**Quartz**	
Size (bytes)	**Latency (μs)**	**Bandwidth (MB/s)**	**Latency (μs)**	**Bandwidth (MB/s)**
1	0.86	3.32	0.68	3.78
2	0.86	6.69	0.64	7.59
4	0.86	13.71	0.62	14.89
8	0.87	27.84	0.61	30.01
16	0.87	55.02	0.59	60.32
32	0.87	107.62	1.00	54.92
64	1.00	152.17	0.96	110.59
128	1.01	303.24	0.97	219.17
256	1.08	559.89	1.01	423.83
512	1.24	1083.95	1.03	832.16
1024	1.26	2106.08	1.26	1251.21
2048	1.46	3548.90	1.68	1865.48
4096	2.30	3087.46	2.05	3654.81
8192	2.82	3954.93	3.18	4142.93
16384	5.19	4884.31	4.59	4854.89
32768	5.70	8640.95	7.61	5231.72
65536	6.57	14126.10	6.79	11082.04
131072	8.99	18945.78	12.62	11299.32
262144	14.12	21533.40	26.20	10343.10
524288	22.27	25929.79	52.75	10057.74
1048576	39.19	28558.42	101.60	10356.57
2097152	71.82	29890.29	199.67	10390.80
4194304	167.41	28233.50	404.25	10548.84

References

[1] Bilge Acun, Phil Miller, and Laxmikant V. Kale. Variation Among Processors Under Turbo Boost in HPC Systems. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, New York, NY, USA, 2016. Association for Computing Machinery.

[2] Advanced Simulation and Computing, Lawrence Livermore National Laboratory. ASC Coral Benchmark Codes, 2014. `https://asc.llnl.gov/sites/asc/files/2020-06/FTQFTW_Summary_v1.1.pdf`.

[3] Pierre Ailliot, Craig Thompson, and Peter Thomson. Mixed methods for fitting the GEV distribution. *Water Resources Research*, 47(5), 2011.

[4] Albert Alexandrov, Mihai F Ionescu, Klaus E Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95–105, 1995.

[5] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[6] Emre Ates, Yijia Zhang, Burak Aksar, Jim Brandt, Vitus J. Leung, Manuel Egele, and Ayse K. Coskun. HPAS: An HPC Performance Anomaly Suite for Reproducing Performance Variations. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Amotz Bar-Noy and Shlomo Kipnis. Designing broadcasting algorithms in the Postal model for message-passing systems. *Mathematical systems theory*, 27(5):431–452, 1994.

[8] David A Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J Kunen, Olga Pearce, Peter Robinson, Brian S Ryujin, and Thomas RW Scogland. RAJA: Portable performance for large-scale scientific applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 71–81. IEEE, 2019.

[9] David E Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E Grant, Thomas Naughton, Howard P Pritchard, Martin Schulz, and Geoffroy R Vallee. A survey of MPI usage in the US exascale computing project. *Concurrency and Computation: Practice and Experience*, 32(3):e4851, 2020.

[10] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs. There goes the neighborhood: Performance degradation due to nearby jobs. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013.

[11] A. Bhatele, J. J. Thiagarajan, T. Groves, R. Anirudh, S. A. Smith, B. Cook, and D. K. Lowenthal. The Case of Performance Variability on Dragonfly-based Systems. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 896–905, 2020.

[12] N. Bhatia, S. R. Alam, and J. S. Vetter. Performance Modeling of Emerging HPC Architectures. In *2006 HPCMP Users Group Conference (HPCMP-UGC'06)*, pages 367–373, 2006.

[13] Amanda Bienz. *Reducing communication in sparse solvers.* PhD thesis, University of Illinois at Urbana-Champaign, 2018.

[14] Amanda Bienz, William D Gropp, and Luke N Olson. Improving performance models for irregular point-to-point communication. In *Proceedings of the 25th European MPI Users' Group Meeting*, pages 1–8, 2018.

[15] Patrick G. Bridges, Matthew G. F. Dosanjh, Ryan Grant, Anthony Skjellum, Shane Farmer, and Ron Brightwell. Preparing for Exascale: Modeling MPI for Many-Core Systems Using Fine-Grain Queues. In *Proceedings of the 3rd Workshop on Exascale MPI*, ExaMPI '15, New York, NY, USA, 2015. Association for Computing Machinery.

[16] Guillaume Calmettes, Gordon B. Drummond, and Sarah L. Vowler. Making do with what we have: use your bootstraps. *Advances in Physiology Education*, 36(3):177–180, 2012. PMID: 22952254.

[17] Laura Carrington, Allan Snavely, and Nicole Wolter. A performance prediction framework for scientific applications. *Future Generation Computer Systems*, 22(3):336–346, 2006.

[18] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12, 1993.

[19] David E Culler, Richard M Karp, David Patterson, Abhijit Sahay, Eunice E Santos, Klaus Erik Schauser, Ramesh Subramonian, and Thorsten Von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, 1996.

[20] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[21] Laurens De Haan and Ana Ferreira. *Extreme value theory: an introduction.* Springer Science & Business Media, 2007.

[22] Arnaldo Carvalho De Melo. The new linux 'perf' tools. In *Slides from Linux Kongress*, volume 18, pages 1–42, 2010.

[23] James Dinan, Ryan E Grant, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev Thakur. Enabling communication concurrency through flexible MPI endpoints. *The International Journal of High Performance Computing Applications*, 28(4):390–405, 2014.

[24] J. Dominguez-Trujillo, K. Haskins, S. J. Khouzani, C. Leap, S. Tashakkori, Q. Wofford, T. Estrada, P. G. Bridges, and P. M. Widener. Lightweight Measurement and Analysis of HPC Performance Variability. In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 50–60, 2020.

[25] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *The International Journal of High Performance Computing Applications*, 30(1):3–10, 2016.

[26] Jack Dongarra, Kevin London, Shirley Moore, Phil Mucci, and Dan Terpstra. Using PAPI for hardware performance monitoring on Linux systems. In *Conference on Linux Clusters: The HPC Revolution*, volume 5. Citeseer, 2001.

[27] Jack Dongarra, Erich Strohmaier, and Horst Simon. Top500 List. `https://www.top500.org/`.

[28] M. G. F. Dosanjh, W. Schonbein, R. E. Grant, P. G. Bridges, S. M. Gazimirsaeed, and A. Afsahi. Fuzzy Matching: Hardware Accelerated MPI Communication Middleware. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 210–220, 2019.

[29] Matthew G.F. Dosanjh, Ryan E. Grant, Whit Schonbein, and Patrick G. Bridges. Tail queues: A multi-threaded matching architecture. *Concurrency and Computation: Practice and Experience*, 32(3):e5158, 2020. e5158 cpe.5158.

[30] Matthew GF Dosanjh, Taylor Groves, Ryan E Grant, Ron Brightwell, and Patrick G Bridges. RMA-MT: a benchmark suite for assessing MPI multi-threaded RMA performance. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 550–559. IEEE, 2016.

[31] D.J. Dupuis and C.A. Field. A Comparison of confidence intervals for generalized extreme-value distributions. *Journal of Statistical Computation and Simulation*, 61(4):341–360, 1998.

[32] H Carter Edwards and Christian R Trott. Kokkos: Enabling performance portability across manycore architectures. In *2013 Extreme Scaling Workshop (xsw 2013)*, pages 18–24. IEEE, 2013.

[33] Bradley Efron. Better Bootstrap Confidence Intervals. *Journal of the American Statistical Association*, 82(397):171–185, 1987.

[34] Y. El-Khamra, H. Kim, S. Jha, and M. Parashar. Exploring the Performance Fluctuations of HPC Workloads on Clouds. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 383–387, 2010.

[35] K. B. Ferreira, P. Bridges, and R. Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2008.

[36] Ian T Foster. *Designing and building parallel programs: concepts and tools for parallel software engineering.* Addison-Wesley, 1995.

[37] Matthew I Frank, Anant Agarwal, and Mary K Vernon. LoPC: modeling contention in parallel algorithms. *ACM SIGPLAN Notices*, 32(7):276–287, 1997.

[38] Maurice Fréchet. Sur la loi de probabilité de l'écart maximum. *Ann. Soc. Math. Polon.*, 6:93–116, 1927.

[39] Phillip B Gibbons. A more practical PRAM model. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 158–168, 1989.

[40] Ryan E. Grant, Matthew G. F. Dosanjh, Michael J. Levenhagen, Ron Brightwell, and Anthony Skjellum. Finepoints: Partitioned Multithreaded MPI Communication. In Michèle Weiland, Guido Juckeland, Carsten Trinitis, and Ponnuswamy Sadayappan, editors, *High Performance Computing*, pages 330–350, Cham, 2019. Springer International Publishing.

[41] William Gropp, Luke N Olson, and Philipp Samfass. Modeling MPI communication performance on SMP nodes: Is it time to retire the ping pong test. In *Proceedings of the 23rd European MPI Users' Group Meeting*, pages 41–50, 2016.

[42] T. Groves, R. E. Grant, and D. Arnold. NiMC: Characterizing and Eliminating Network-Induced Memory Contention. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 253–262, 2016.

[43] Emil Julius Gumbel. The Return Period of Flood Flows. *The Annals of Mathematical Statistics*, 12(2):163–190, 1941.

[44] John L Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[45] Nathan Hanford, Ramesh Pankajakshan, Edgar A León, and Ian Karlin. Challenges of GPU-aware Communication in MPI. In *2020 Workshop on Exascale MPI (ExaMPI)*, pages 1–10. IEEE, 2020.

[46] Nathan Hjelm, Matthew GF Dosanjh, Ryan E Grant, Taylor Groves, Patrick Bridges, and Dorian Arnold. Improving MPI multi-threaded RMA communication performance. In *Proceedings of the 47th International Conference on Parallel Processing*, pages 1–11, 2018.

[47] RW Hockney, CR Jesshope, and CR Jesshope. *Parallel Computers: Architecture, Programming and Algorithms*. Hilger (Bristol), 1981.

[48] Torsten Hoefler. Bridging Performance Analysis Tools and Analytic Performance Modeling for HPC. In Mario R. Guarracino, Frédéric Vivien, Jesper Larsson Träff, Mario Cannatoro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, and Michael Alexander, editors, *Euro-Par 2010 Parallel Processing Workshops*, pages 483–491, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[49] Torsten Hoefler and Roberto Belli. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[50] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. LogGOPSim: simulating large-scale applications in the LogGOPS model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 597–604, 2010.

[51] J.R.M. Hosking, J.R. Wallis, and E.F. Wood. Estimation of the Generalized Extreme-Value Distribution by the Method of Probability-Weighted Moments. *Technometrics*, 27(3):251–261, 1985.

[52] IBM Spectrum LSF Session Scheduler. `https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=lsf-session-scheduler`.

[53] IBM Spectrum MPI Version 10 Release 1 User's Guide, 2016. `https://www.ibm.com/docs/en/SSZTET_EOS/eos/guide_101.pdf`.

[54] Heike Jagode, Anthony Danalis, and Jack Dongarra. Formulation of Requirements for new PAPI++ Software Package, PDN No. 1. 2020.

[55] Heike Jagode, Anthony Danalis, and Damien Genet. Roadmap for Refactoring classic PAPI to PAPI++, PDN No. 2. 2020.

[56] Arthur F Jenkinson. The frequency distribution of the annual maximum (or minimum) values of meteorological elements. *Quarterly Journal of the Royal Meteorological Society*, 81(348):158–171, 1955.

[57] Brian Kocoloski. *Scalability in the Presence of Variability.* PhD thesis, University of Pittsburgh, 2018.

[58] Jan Kyselý. A Cautionary Note on the Use of Nonparametric Bootstrap for Estimating Uncertainties in Extreme-Value Models. *Journal of Applied Meteorology and Climatology*, 47(12):3236 – 3251, 01 Dec. 2008.

[59] Lawrence Livermore National Laboratories. Lassen Supercomputing Platform. `https://hpc.llnl.gov/hardware/platforms/lassen`.

[60] Lawrence Livermore National Laboratories. Quartz Supercomputing Platform. `https://hpc.llnl.gov/hardware/platforms/Quartz`.

[61] Lawrence Livermore National Laboratories. Running jobs on Lassen. `https://hpc.llnl.gov/training/tutorials/using-lcs-sierra-system`.

[62] Philipp Leitner and Jürgen Cito. Patterns in the Chaos — A Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Trans. Internet Technol.*, 16(3), April 2016.

[63] Henrik Madsen, Peter F. Rasmussen, and Dan Rosbjerg. Comparison of annual maximum series and partial duration series methods for modeling extreme hydrologic events: 1. At-site modeling. *Water Resources Research*, 33(4):747–757, 1997.

[64] Eduardo S Martins and Jery R Stedinger. Generalized maximum-likelihood generalized extreme-value quantile estimators for hydrologic data. *Water Resources Research*, 36(3):737–744, 2000.

[65] W. Pepper Marts, Matthew G. F. Dosanjh, Whit Schonbein, Ryan E. Grant, and Patrick G. Bridges. MPI Tag Matching Performance on ConnectX and ARM. In *Proceedings of the 26th European MPI Users' Group Meeting*, EuroMPI '19, New York, NY, USA, 2019. Association for Computing Machinery.

[66] Paul Messina. The exascale computing project. *Computing in Science & Engineering*, 19(3):63–67, 2017.

[67] O. H. Mondragon, P. G. Bridges, S. Levy, K. B. Ferreira, and P. Widener. Understanding Performance Interference in Next-Generation HPC Systems. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 384–395, 2016.

[68] Csaba Andras Moritz and Matthew I Frank. LoGPC: Modeling network contention in message-passing programs. In *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 254–263, 1998.

[69] Message Passing Interface Forum. `https://www.mpi-forum.org/mpi-40/`.

[70] MPI: A Message-Passing Interface Standard Version 4.0, November 2020. `https://www.mpi-forum.org/docs/drafts/mpi-2020-draft-report.pdf`.

[71] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710. Citeseer, 1999.

[72] The OpenACC organization members. The OpenACC Application Programming Interface, November 2020. `https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.1-final.pdf`.

[73] OSU Micro-Benchmarks 5.7. `https://mvapich.cse.ohio-state.edu/benchmarks/`.

[74] Steve Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 117(1):1–19, 1995.

[75] A. Porterfield, S. Bhalachandra, W. Wang, and R. Fowler. Variability: A Tuning Headache. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1069–1072, 2016.

[76] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proc. VLDB Endow.*, 3(1–2):460–471, September 2010.

[77] Whit Schonbein, Matthew GF Dosanjh, Ryan E Grant, and Patrick G Bridges. Measuring multithreaded message matching misery. In *European Conference on Parallel Processing*, pages 480–491. Springer, 2018.

[78] Scipy. scipy.stats.rv_continuous.fit, 2020. Last accessed September 11, 2020.

[79] Sameer S Shende and Allen D Malony. The TAU parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[80] D. Skinner and W. Kramer. Understanding the causes of performance variability in HPC workloads. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, pages 137–149, 2005.

[81] Nawrin Sultana, Martin Rüfenacht, Anthony Skjellum, Purushotham Bangalore, Ignacio Laguna, and Kathryn Mohror. Understanding the use of message passing interface in exascale proxy applications. *Concurrency and Computation: Practice and Experience*, page e5901, 2020.

[82] J. Sun and G. D. Peterson. An Effective Execution Time Approximation Method for Parallel Computing. *IEEE Transactions on Parallel and Distributed Systems*, 23(11):2024–2032, 2012.

[83] The MVAPICH Team. MVAPICH2 2.3 User Guide, March 2017. `http://mvapich.cse.ohio-state.edu/static/media/mvapich/mvapich2-2.3a-userguide.pdf`.

[84] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010.

[85] Jan Treibig, Georg Hager, and Gerhard Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th International Conference on Parallel Processing Workshops*, pages 207–216. IEEE, 2010.

[86] Jan Treibig, Georg Hager, and Gerhard Wellein. LIKWID: Lightweight performance tools. In *Competence in High Performance Computing 2010*, pages 165–175. Springer, 2011.

[87] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. Brandt, V. J. Leung, M. Egele, and A. K. Coskun. Online Diagnosis of Performance Variation in HPC Systems Using Machine Learning. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):883–896, 2019.

[88] Y. Ueda and T. Nakatani. Performance variations of two open-source cloud platforms. In *IEEE International Symposium on Workload Characterization (IISWC'10)*, pages 1–10, 2010.

[89] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[90] Waloddi Weibull et al. A statistical distribution function of wide applicability. *Journal of applied mechanics*, 18(3):293–297, 1951.

[91] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*, pages 44–60. Springer, 2003.