A REGRESSION-BASED SYSTEM FOR ACCURATE SCALABILITY PREDICTION ON LARGE-SCALE MACHINES

by

BRADLEY JAMES BARNES

(Under the direction of David K. Lowenthal)

ABSTRACT

Scalability prediction is one of the key problems facing high performance computing today. Methods to predict scalability accurately are necessary in order to improve throughput and overall efficiency on large-scale machines. This dissertation presents our novel, regression-based system for accurately predicting the scalability of scientific applications on large-scale machines. Our regression-based system provides accurate runtime predictions on large processor counts for multiple scientific applications when run using strong scaling. Our system is also able to provide input parameters leading to accurate time-constrained scaling on larger processor counts. We also discuss the impact of noise on scalability prediction. This work takes large steps towards a general scalability prediction system that could be deployed on supercomputing systems in the near future.

INDEX WORDS: Modeling, MPI, Prediction, Regression, Scalability, Noise, Theses (academic)

A REGRESSION-BASED SYSTEM FOR ACCURATE SCALABILITY PREDICTION ON LARGE-SCALE MACHINES

by

BRADLEY JAMES BARNES

B.S., College of Charleston, 2004

A Dissertation Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

ATHENS, GEORGIA

2011

A REGRESSION-BASED SYSTEM FOR ACCURATE SCALABILITY PREDICTION ON LARGE-SCALE MACHINES

by

BRADLEY JAMES BARNES

Major Professor: David K. Lowenthal

Committee: Bronis R. de Supinski
Jaxk Reeves
Kang Li
Lakshmish Ramaswamy

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
August 2011

DEDICATION

This dissertation is dedicated to my loving parents, Carolyn and Jim Barnes who have supported me throughout this eleven-year journey as a student.

ACKNOWLEDGEMENTS

An accomplishment of this magnitude is rarely achieved without the help of family and friends. I am incredibly grateful to all of the wonderful people who have helped me along the way. I would like to start off by saying that I would have never made it this far without the guidance of my advisor, Dr. David Lowenthal. Dave has been a great leader, teacher, and mentor to me since I first walked into his office in 2006. Dave has always had my best interest at heart and I have always known that I would be successful under his guidance. I would like to thank all of the wonderful people at LLNL who have helped me out over the years. Specifically, I am thankful for all of the helpful comments and suggestions made by Bronis R. de Supinski and Martin Schulz. I would also like to thank them for the opportunity to be a summer intern in 2008. In a similar vein, I would like to thank Jaxk Reeves, Jeonifer Garren, and Guannan Wang for providing invaluable knowledge and assistance in the statistical aspects of this dissertation. It is only through their suggestions and help that we were able to successfully implement a regression-based system. I am also thankful for the support and encouragement provided by all of my friends in (and out of) the computer science department. I have been lucky enough to meet a lot of amazing people (too many to name them all in this document) that I will undoubtedly keep in contact with long after we all leave Athens. Finally, I would like to thank my family (Mansfields and Barnes)

for their unconditional love and support. In particular, my parents for always giving me a home away from Athens where I always felt loved -- and never left hungry.

This dissertation work was partially funded by the Computer Science Department of the University of Georgia

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

1.1 Motivation

The top supercomputers in the world today average nearly 13,000 processing cores per system. Nine out of the top 500 systems contain more than 128,000 processors [23]. Systems of this size are extremely expensive to build and to maintain. For example, the fastest supercomputer in the world today, Tianhe-1A, costs over $88 million to build and requires roughly $2.7 million of power annually [64]. Annual operating costs including salaries of the hundreds of employees hired to run Tianhe-1A are estimated at $10-20 million [61]. In 2012, Oak Ridge National Laboratory and IBM are expected to complete 20 petaflop supercomputers named Titan and Sequoia, respectively. Without considering operating costs, Titan is estimated to cost $100 million, and Sequoia is expected to cost $200 million [11]. Given the increasing size and cost of supercomputers, it is reasonable to expect that total cost of ownership will soon reach over a billion dollars. With such large amounts of monetary resources applied to the construction and maintenance of large-scale systems, there is significant pressure on supercomputer centers to justify the expense.

Supercomputers must run jobs that are relevant to national security, prosperity, and improve our understanding of science while maintaining high efficiency and throughput. Each day, computational scientists (hereafter, referred to as scientists) around

the world use supercomputers to tackle some of the world's most difficult scientific problems in over 30 application domains [40]. For example, Sequoia will be used for classified nuclear weapons simulations, predicting the weather, studying the cosmos, and understanding the human genome. Recently, supercomputers at Lawrence Livermore National Lab were used to gauge radiation risks posed by the nuclear crisis in Japan. Unfortunately, scientists do not always run their jobs on an efficient number of processors, which leads to application slowdown and lower throughput on the machines.

In parallel computing, two of the most important performance metrics are speedup and efficiency. Speedup refers to how much faster the program runs when compared to the sequential version (Formula 1.1) [70]. The speedup on *p* processors (S_p) is the ratio of the sequential application time (T_s) to the parallel application time on *p* processors (T_p). Efficiency, on the other hand, estimates how well the processors are being utilized in a given application run. Parallel efficiency (E_p) is the ratio of the speedup on *p* processors to *p* (Formula 1.2).

$$Sp = \frac{Ts}{Tp} \tag{1.1}$$

$$Ep = \frac{Sp}{p} \tag{1.2}$$

The ideal number of processors to use for a scientific application varies with both the application and the machine under consideration. For example, Alam et al. [4] characterized performance of a biomolecular simulation and found that one time-consuming part of the program achieved a speedup of only 10 on 1024 processors (see Figure 1.1). In this case, the application would have run more efficiently on 16-64 processors. However, since the users are typically unaware of how an application is going

to scale, they will attempt to run their jobs on as many processors as possible, which leads to inefficient use of expensive supercomputing resources. This inefficiency is costly to both the owners and the other users of the systems. To the supercomputer center, it wastes the system in terms of money and power consumption. To the other users of the system, it reduces system availability because more processors than necessary are being allocated. Figure 1.2 shows the worst-case time to acquire nodes appears to increase exponentially in the number of nodes. So, in some cases, users are waiting exponentially longer in the scheduling queue only to have their jobs run slower. Given such a large downside to running applications inefficiently, why do users choose to run on so many processors? Predicting the parallel efficiency of applications without executing them at scale is an extremely complicated problem. In fact, it is one of the key problems facing high-performance computing (HPC) as we move toward exascale systems.

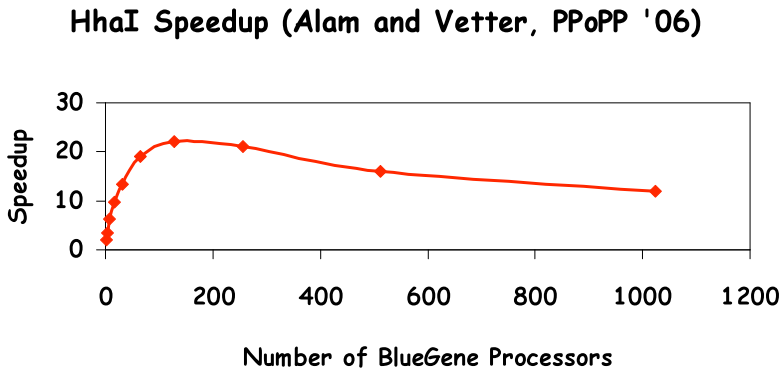

Figure 1.1: Speedup of the HhaI function. Speedup levels off around 128 and then begins to decline.

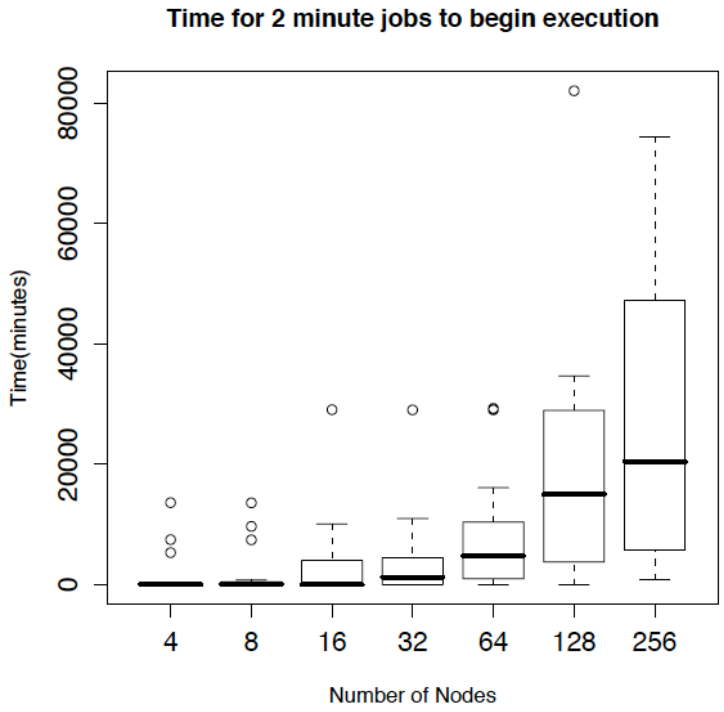

Figure 1.2: Time for short jobs to begin execution on increasing node counts on the Thunder cluster at Lawrence Livermore National Lab.

In order for a user to predict an application's scalability they would need an in-depth understanding of the parallel algorithm, how input parameters affect the program, the amount of potential parallelism that an application contains, the effects of the parallel overhead on runtime, and an understanding of the machine architecture. If the user knows the amount of potential parallelism, Amdahl provided an equation for theoretical application speedup when using N processors (Equation 1.3). In this equation, *P* represents the parallelizable portion of the code, 1 - *P* is the serial portion, and *N* represents the number of processors used. Amdahl's law shows that speedup is limited by the percentage of sequential code present in the application. If we fix the values of *N* and *P*, we quickly see how parallelism is limited (Table 1.1). In the right-most column, where *P* is .99, speedup is bounded by 100 at large processor counts. In this case, increasing the processor count by a factor of 10 (from 10,000 to 100,000) yields a less

than 1% increase in speedup, an extremely inefficient use of such a large number of processors. Amdahl's law provides a theoretical approximation of speedup, but a more detailed analysis is required in practice.

$$Speedup = \frac{1}{(1-P) + \frac{P}{N}} \tag{1.3}$$

N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02
100000	1.99	9.99	99.90

Table 1.1: Theoretical speedup of applications with varying amounts of potential parallelism (*P*) under different processor counts (*N*)

1.2 Contributions

The contributions of this dissertation are novel regression-based approaches to scalability prediction for large-scale scientific applications.

In this dissertation, we present a system to predict parallel program scalability. We use several program executions on a small subset of the processors to predict application performance on larger numbers of processors. Our system produces accurate predictions without having to understand low-level details about the applications and without executing them at scale. We argue that, if used on future supercomputers, our

system will help increase availability. Application scientists could use the system to determine how many processors to request so their applications run quickly without wasting resources beyond the point at which they achieve good speedup. The improved efficiency would not only reduce demand on the system's resources but would generally improve response time for the specific application. Our system could also be deployed as a service running on a separate partition to avoid using processors on the full partition.

In many HPC applications, the goal when given more processors is to run with larger input values, which allows the scientist to solve a larger problem (as opposed to running a given problem size as fast as possible). With time constraints, it is important to understand how the application is going to scale in order to provide input parameters that will result in the same execution time at a larger scale. Using input parameters that result in increased execution time could lead to a job exceeding the system's time constraints, which causes the job to be cancelled and the supercomputer resources to be wasted. The system presented in this dissertation is able to provide the application input parameters which result in accurate, time-constrained scaling at large processor counts. Our gray-box system does this with only a small amount of application-level information from the user. Our system works by using a focused regression technique, which uses a small focused set of training data closely resembling the application instance.

The presence of noise in a system causes random execution time variability which complicates performance prediction and leads to degraded accuracy for scalability models. The effects of system noise on a particular application are difficult to understand without running multiple replications of each program instance. In fact, the amount of training time required for our system to model the effects of system noise on an

application would be impractical without optimization. We believe that a logical first step to predicting application scalability in the presence of noise is to cut down the amount of training required. In this dissertation, we present a technique which reduces training time by predicting the execution time of scientific applications running for multiple iterations by using many samples from single-iteration executions on the same number of processors. While this technique does not extrapolate to larger processor counts, we believe it is an essential building block for a large-scale extrapolation system.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows. Chapter 2 discusses our system using a regression-based approach to scalability prediction. Then, Chapter 3 discusses an extension of our system for time-constrained scaling. In Chapter 4 we detail our work on prediction in the face of runtime variance. Next, we present related work on performance prediction for large-scale computer systems in Chapter 5. Finally, conclusions and future work comprise Chapter 6.

CHAPTER 2

SCALABILITY PREDICTION USING REGRESSION-BASED TECHNIQUES

Many applied scientific domains are increasingly relying on large-scale parallel computation. Consequently, several large clusters now have hundreds of thousands of processors. However, the ideal number of processors to use for these scientific applications varies with both the input variables and the machine under consideration, and predicting this processor count is rarely straightforward. Accurate prediction mechanisms would provide many benefits, including improving cluster efficiency and identifying system configuration or hardware issues that impede performance.

In this chapter we show how regression-based techniques can be used to predict execution time at large scale from training data gathered on a small subset of the processors. We explore novel regression-based approaches to predict parallel program scalability. We use several program executions on a small subset of the processors to predict execution time on larger numbers of processors. We compare three different regression-based techniques: one based on execution time only; another that uses per-processor information only; and a third one based on the global critical path. These techniques provide accurate scaling predictions, with median prediction errors between 6.2% and 17.3% for seven applications. In this chapter, we focus on strong scaling [69], where the scientist runs the same program instance, i.e., uses identical input parameters at larger processor counts. In the following chapter, we will present our focused regression

technique for predicting the input parameters that lead to accurate time-constrained scaling.

We investigate three techniques based on regression for predicting parallel program scalability. These techniques use several executions with different input sets on a *small subset of the processors* to predict performance on a larger number of processors. Our first technique is the most straightforward: simply fit total execution time from the data collected on training runs to a regression and extrapolate to larger configurations. This simple technique works well for some cases if we use a reasonable prediction function (a second-order polynomial). Our other two techniques refine this approach by handling computation and communication separately. One technique relies only on per-processor information; it gathers the computation and communication times of each processor, chooses the most representative pair, and separately regresses on each to form a prediction. Our third technique, unlike the per-processor method, ensures communication time never includes blocking by calculating computation and communication time via identification of the (global) critical path. Both techniques that separate communication from computation improve prediction quality in the common case that both quantities are significant.

This chapter makes several contributions. First, we show that the simple, "black-box" technique of regression can often accurately predict performance on a larger processor count. Second, we present a novel technique—separate regression on computation and communication—that improves prediction accuracy for processor counts at which applications scale poorly. Third, we identify two potential refinements to make higher-quality predictions: better prediction functions and special handling of

memory anomalies, including both NUMA and cache capacity effects. Fourth, our predictions for seven applications at processor counts up to 1024, based on runs on as few as 128 processors, demonstrate that accurate extrapolation of scaling behavior is possible. Specifically, we achieve median prediction errors of between 6.2% and 17.3% over all nontrivial programs. One of the nontrivial programs is Sweep3D, where we specifically chose a configuration that would obscure scaling behavior. We also provide a mechanism to estimate how large processor counts for training runs need to be for an accurate prediction.

The rest of this chapter is organized as follows. Section 2.1 describes our techniques for performance prediction. Next, we describe our experimental methodology in Section 2.2 and the results of using our techniques on seven applications in Section 2.3. Finally, Section 2.4 places our approach in the context of prior work, while Section 2.5 summarizes our findings and future directions.

2.1 Performance Prediction Techniques

In this section, we outline the definitions and assumptions of this chapter and we discuss the mathematical models used to make scalability predictions on large scale machines.

2.1.1 Definitions and Assumptions

In this section a processor configuration is simply a set of processors, with one or more processors (or cores) on each node. We investigate predictions using strong scaling where possible, where the total working set size is fixed over all processor

configurations. In some cases, strong scaling is impractical because of memory requirements at low-end processor counts, and in those cases we use a hybrid of strong scaling along with weak scaling. Applications that use weak scaling increase their total working set size proportionally as the number of processors increases, while the working set for each processor remains constant.

We make several assumptions in this work. First, we assume that all input variables to a given program (such as data set size and processor grid dimensions) are available to us. This assumption is reasonable, as most high-performance computing applications use (sometimes complex) configuration files that specify the input variables directly. Next, we assume that we know which input variables (e.g., sizes/constants given typically in input files) contribute significantly to execution time. Known techniques exist to find these variables [41]. Our procedure models execution time as some function of these input variables. The quality of our model depends on considering all of the important input variables when building the model. We also assume that the computational load is well balanced; we will explore load imbalance in future work. Finally, we assume that a program can be run using any configuration of the input variables. While this does not always hold—for example, many of the NAS Parallel Benchmarks [5] constrain the values of the input variables—we overrode this limitation in our training sets.

2.1.2 Approach

We predict execution time of a given program on p processors using several instrumented runs of the same program on q processors, where $q \in \{2, \ldots, p_0\}$, $p_0 < p$,

and p is arbitrary. We vary the values of the input variables ($x_1, x_2, \ldots, x_n$) on the instrumented runs. Because it is easier to acquire q processors than p, it is reasonable to perform many instrumented runs for different configurations of the input variables. We then use the relationship between the input variables and the observed execution time to develop a predictor, T', of the execution time T:

$$T' = F(x_1, x_2, \ldots, x_n, q). \tag{2.1}$$

The idea is that $T' \approx T$, with small error. Once we determine T', we use it to predict execution time for any arbitrary input variable set ($a_1, a_2, \ldots, a_n$) and p processors. We emphasize that we produce T' without any data from runs using p processors, since we choose $p_0 < p$.

The scale in which the error between T' and the true T is measured is crucial. For most applications, variability increases as T increases, so we use relative error:

$$E = |(T - T')|/T \tag{2.2}$$

Thus, our function F should minimize this relative error, subject to some feasibility constraints. Evaluation of different models in terms of relative error depends heavily on the input variables. Ideally, the function F minimizes the relative error by intelligent choices of the training set, i.e., the sets of input variables ($x_1, x_2, \ldots, x_n$) and number of processors (q) used to build the model.

Because we use relative error to evaluate models, F should be fit to T in log-scale. The particular log-scale (e.g., $\log_2$ or $\log_{10}$) does not matter statistically; we use $\log_2$ and fit models of the form:

$$log_2(T) = log_2(F(x_1, x_2, \ldots, x_n, q)) + error \tag{2.3}$$

such that the error is minimized in $\log_2$-scale. We can convert an individual $\log_2$-scale

error (e) into a relative error (RE): $RE = 2^{|e|} - 1$. However, for statistical accuracy we must minimize error in the log-scale when evaluating a model's fit over different input configurations. If we minimized error in the untransformed (T) scale, errors at the largest values of T would completely dominate those at smaller T values, making the model inaccurate.

Parameterization of the $log_2(F(x_1, x_2, \ldots, x_n, q))$ function is critical. A linear model like

$$log_2(T) = \beta_0 + \beta_1 \log_2(x_1) + \beta_2 \log_2(x_2) + \ldots + \beta_n \log_2(x_n) + \beta_q \log_2(q) + error \quad (2.4)$$

provides a reasonable first approximation, although it is too simple to capture the behavior of some applications. Statisticians refer to this as a linear model, since it is linear in the unknown parameters ($\beta_0, \beta_1, \ldots, \beta_q$) that are estimated so as to minimize the sum of squared error (in log-scale). In engineering contexts, one might call this a "log-log" model, because $\log_2$ is applied to both sides of Equation 2.3 to obtain Equation 2.4, but it is a linear model in the statistical sense, which means we can employ the vast statistical theory of linear models (of which multiple regression is a subset). The right-hand side of Equation 2.4 can be made considerably more general while remaining a linear model in the statistical sense. For example, one could include quadratic terms such as $(log_2(x_i)^2)$ or interaction terms such as $log_2(x_i) * log_2(x_j)$, or even try other transformations of the input variables, such as x_i or $\sqrt{x_i}$ rather than $log2(x_i)$. Our results for the seven applications that we examine show that most of the variability due to the input variables ($x_1, x_1, \ldots, x_n$) is explained by models of the form:

$$log_2(T) = \beta_1 \log_2(x_1) + \beta_2 \log_2(x_2) + \ldots + \beta_n \log_2(x_n) + g(q) + error \quad (2.5)$$

Thus, we focus on finding a good-fitting but parsimonious function $g(q)$ that explains the

effect of the number of processors, q. For three of the seven applications that we examine, the simple linear function:

$$g(q) = \gamma_0 + \gamma_1 \log_2(q) \tag{2.6}$$

is best, while quadratic (in $\log_2(q)$) models, where there is an additional term $\gamma_2(\log_2(q))^2$, fit the other four applications better. In general, we could use more complex $g(q)$ functions or include more parameters (β_i in the linear model). However, at some point the model adjustments will fit the sample data beyond their relation to the predicted input configurations. In this work we therefore do not consider higher-order polynomials.

2.1.3 Techniques

Our most straightforward approach uses the total execution time for T in Equation 2.5. Considering the two possible forms of $g(q)$ above, we have two possible ways to model T. Gathering the input for this approach is simple because our applications all report their execution times. We show in Section 4 that predictions using regression based solely on total execution time are effective in some cases.

However, computation and communication typically scale differently as processor count changes. To capture the individual scaling properties of computation and communication, we developed two techniques that separate computation and communication. The amount of computation in parallelizable code regions will generally scale proportionally to the increase in the number of processors, which holds for strong scaling of load-balanced applications. On the other hand, the behavior of communication time as the number of processors increases depends on the application. While it often increases with a rising numbers of processors, our experiments also show some cases of

decreasing communication time.

Our second approach uses the maximum computation time across all processors and the communication time from that same processor. We use the PMPI profiling interface to wrap all MPI calls to measure both quantities. Because our applications are well balanced computationally, the communication time usually contains the minimum amount of blocking time over all processors.

Our third technique avoids blocking time altogether by focusing on the parallel execution's *critical path*, the longest execution sequence *without blocking*. The critical path determines the execution time of a parallel program as Figure 1 shows. Any communication time on this path is purely communication (i.e., sending/receiving), which helps our model avoid overestimating it.

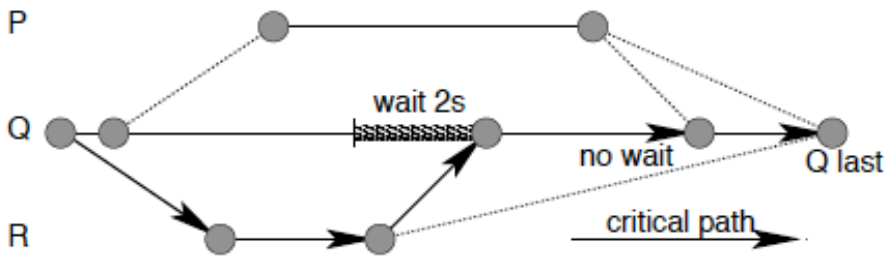

Figure 2.1: Critical path: P, Q, and R are MPI tasks with edges representing messages.

For each technique to separate computation and communication, we can fit the computation time two ways and fit the communication time two ways because we consider two possible forms for $g(q)$. Combined with two possible ways to split computation and communication, we therefore consider eight possible ways to predict total execution time when separating computation and communication.

2.2 Experimental Methodology

We tested our techniques using seven applications: five from the NAS parallel benchmark suite [5] and two from the ASC Purple/Blue suites [1, 38]. The NAS codes are BT and SP, which are computational fluid dynamic (CFD) applications that use different solution approaches; CG, an unstructured sparse linear solver; EP, an embarrassingly parallel program; and LU, a lower- and upper-triangular solution to implicit CFD problems. We omit some NAS programs because of their inherent constraints. Specifically, MG, FT, and IS require that the input sizes be powers of two, which does not allow us enough tests to achieve a statistically significant result. The ASC applications are SMG, a multigrid code, and Sweep3D, a 3D neutron transport code.

We make predictions of programs running on p processors using three different processor configurations for training: $p_0 = p/8$, $p_0 = p/4$, and $p_0 = p/2$. We follow this in our experiments below as closely as possible; BT and SP require a number of processors that is a perfect square, so we chose even-numbered processor configurations as close as possible to powers of two.

We currently make the decision as to whether to separate computation and communication as follows. We separate if either (1) a program is not computation bound (consisting of mostly computation) or (2) communication time increases with processors. If the computation time is 90% of the total runtime (on average) across all training runs for any processor configuration used for training, we deem the application computation bound (on both the per-node maximum and the critical path techniques).

Recall that we regress on the input variables that contribute significantly to

execution time as well as q, the number of processors. Table 2.1 shows the relevant input variables and the ranges that we used for our seven applications. The applications are all iterative, yet balance the work across the processors; thus, we do not include iteration count as a predictor variable. We use strong scaling with BT, CG, SMG and Sweep3D while we use a hybrid of strong and weak scaling in which we increase the problem size on large processor counts to get reasonable execution times with EP, SP, and LU.

Application	Name 1	Range	Name 2	Range	Name 3	Range
BT	*problem_size*	20-500	-	-	-	-
CG	*NA*	500K-3M	*NONZER*	14-24	-	-
EP	*m*	2^{28} - 2^{38}	-	-	-	-
LU	*isiz1*	200-1000	*isiz2*	200-1000	*isiz3*	200-1000
SMG	*DIM1*	290-315	*DIM2*	290-315	*DIM3*	290-315
SP	*problem_size*	40-1400	-	-	-	-
Sweep3D	IT_G	300-500	JT_G	300-500	KT_G	300-500

Table 2.1: The seven applications and their input variable names and ranges.

We observe results (T) for each instrumented run on q processors and fit a linear model of the form given in Equation 2.5 for various $g(q)$. For a specified $g(q)$, the set of β_i returned by the regression function are those that minimize the sum of squared error. The root-mean-squared-error (RMSE) for a particular regression measures the typical error in $\log_2$-scale when using the specified regression function to fit execution times over all input configurations of $(x_1, x_2, \ldots, x_n, q)$. As mentioned above, we choose the function $g(q)$ to be a second order polynomial, which we found sufficient for our experiments.

We use the statistical package R [54] for all regressions. We emphasize that we

run the program only on a small subset of the many possible input variable/processor combinations. Generally, we allowed between 10-64 tests (training runs) with various input values at each processor count. For the purposes of verification, we also run the program at the predicted processor count, p. We use the execution times, T, on p processors to evaluate how well a proposed function actually predicts a run on p processors, while the regressions we use to predict time on p processors *do not* include these results.

2.3 Results

This section discusses results of our performance prediction techniques. We used the "Atlas" cluster, which has 1152 four-way AMD Opteron nodes, for all experiments. Each node has four processors containing two cores running at 2.4 GHz, a 128KB split L1 cache, a 1MB L2 cache, and 16GB RAM. Each Opteron node is a NUMA architecture because each processor has one quarter of the memory connected to a local on-chip memory controller, while the rest must be accessed through remote memory controllers inside the remaining processors, which incurs longer memory latencies. Hereafter, we use the term processor to refer to a core to avoid confusion. We restrict our experiments to four processors per node since using all processors on a node risks high variance [52]. Atlas uses a priority-based batch queuing system that limited our ability to run sufficient experiments to (a maximum of) 1024 processors. Because Atlas nodes have a NUMA architecture, we used cpu_bind to ensure that Linux allocated memory for each processor locally. Separate experiments showed that Linux allocations do not otherwise account for locality, which leads to large, highly unpredictable variance in execution times with

identical input variables.

2.3.1 Summary of Results

We present the full results of our experiments in Sections 2.3.3 and 2.3.4 while Tables 2.2 and 2.3 show the median percentage error over all experiments for each application. All predictions are for 1024 except for SMG, which uses 256 due to memory limitations. We show the best and worst errors both for regressing on total execution time and (if applicable) for regressing on the separate times. For each, we also (if applicable) list the permutation that we used (type of fit and whether we used the critical path or maximum per-processor computation; when regressing on total time, time is not split into two quantities so we only have one degree of freedom). We give more details in Sections 2.3.2, 2.3.3, and 2.3.4.

We make the following general observations. First, prediction quality is often quite good, even on as few as $p_0 = p/8$ processors. Second, prediction quality for communication intensive applications is, except in one case (CG with $p_0 = p/8$), equal or better when we treat computation and communication separately (assuming that there is enough communication to merit separating). Third, CG is the primary case in which prediction quality on $p_0 = p/8$ processors is poor. Finally, CG is also the one case in which prediction quality for separate regression is better on $p_0 = p/4$ processors than on $p_0 = p/2$ processors. We discuss these issues, including how we might infer them automatically, in Section 2.3.4.

Application	Median Error, Total		Type (Best)
	Best (Procs)	Worst (Procs)	
BT	6.7% (484)	13.0% (100)	L
CG	16.0% (256)	120% (128)	L
EP	0.1% (512)	0.6% (128)	L
LU	13.8% (512)	15.8% (128)	Q
SP	7.7% (100)	12.8% (256)	L
SMG	14.4% (128)	92.7% (32)	Q
Sweep3D	32.8% (512)	59.3% (128)	Q

Table 2.2: Summary of results for all applications when regressing on total execution time. The best and worst median errors along with the number of processors used for prediction are shown. The right-most column shows, for the best error, what type of regressions (linear vs. quadratic) is used.

Application	Median Error, Separate		Type (Best)
	Best (Procs)	Worst (Procs)	
BT	--	--	--
CG	12.2% (256)	66.3% (128)	Q/L/CP
EP	--	--	--
LU	--	--	--
SP	7.7% (100)	12.1% (256)	L/Q/Max
SMG	6.7% (128)	25.6% (32)	Q/Q/Max
Sweep3D	17.3% (512)	33.2% (128)	Q/Q/Max

Table 2.3: Summary of results for all applications when regressing separately on computation and communication (if applicable). The best and worst median errors along with the number of processors used for predication are shown. The right-most column shows, for the best error, what type of regressions (linear vs. quadratic) and which type of separation (critical path vs. maximum), if any, are used.

2.3.2 Format of Detailed Results

In Figures 2.2, 2.3, and 2.4 we show two graphs for each computation-bound application. Figures 2.5, 2.6, 2.7, and 2.8 show three graphs for communication intensive applications. The first (leftmost) graph is a boxplot that shows the median, minimum, and maximum error for predicting 1024-processor performance (again, SMG uses 256 processors due to per-node memory limitations) using the three next largest processor configurations (e.g., for the 1024 processor tests, we use values of 128, 256, and 512 for p_0). The reported median, minimum, and maximum errors represent values over all permutations of the input variables (small circles represent outliers). As we explore different fits to model computation and communication, these boxplots display the prediction results. The way we chose the prediction model is as follows. For the total execution time prediction (TOT), we chose a linear or quadratic fit based on lowest root-mean-squared-error (RMSE) over all points up to and including the p_0 being investigated. Additionally, if the program is considered communication intensive (see Section 2.2), then for the predictions when separating computation and communication (SEP), we chose the model with the lowest weighted average RMSE. That is, we weighted the RMSE for computation and communication by the percentage each contributed to total execution time on p_0 processors.

The middle graph in Figures 2.5, 2.6, 2.7, and 2.8 (only for communication intensive applications) investigates three different models that treat computation and communication separately. This includes prediction using values of all input variables, including number of processors. Recall from Section 2.2, we have eight possible models

when regressing separately. For readability, we chose three of these eight possibilities: the one with the smallest, second smallest, and largest weighted average RMSE. Each of the three alternatives are labeled with a three-tuple that represents the type of fit used for computation and communication, respectively (Q or L for quadratic or linear), and whether the critical path (CP) or maximum per-processor computation (MAX) was used to separate the two quantities.

Finally, the rightmost graph (Figures 2.2 through 2.8) shows the quality of the fit obtained (over all processor configurations) using the three alternatives in the middle graph (or leftmost for computation intensive applications). For communication intensive applications, the regression line is shown for one particular permutation of the input variables, and this permutation is listed in the caption. We also graph the measured time and, for reference, baseline linear speedup relative to the smallest processor configuration. Finally, we extend the predictions to show our model results for processor counts beyond those with which we experimented.

2.3.3 Computation-Bound Applications

We first discuss applications for which computation time is dominant. This includes three applications: BT, EP, and LU. We first give an overview of all three applications, and then we cover BT in depth (the characteristics of LU are similar, and EP is a trivial application).

BT overview: BT yields nearly linear speedup up to 1024 processors, as shown in Figure 2.2, which makes it a straightforward application to predict. The left-hand figure shows that a linear function for $g(q)$ using total execution time to make predictions using

1024 processors always has a median error no more than 13%—it is 13% when using 100 processors for prediction, 7.2% when using 256 processors, and 6.7% when using 484 processors. The simplicity of modeling BT leads to scaling predictions (right-hand graph) that match the measured execution time almost perfectly.

LU overview: Like BT, LU yields good speedup (see Figure 2.3). While the results for $p_0 = p/8$ are good (14%) when considering the median, the worst case (not including outliers) is slightly over 50%. This error arises due to tests with small run times.

EP overview: EP is a rather straightforward application. Its only communication is at MPI initialization and at the end of the program (a barrier). Essentially, any technique works well to predict execution time for EP (our error was well below 1%; see Figure 2.4).

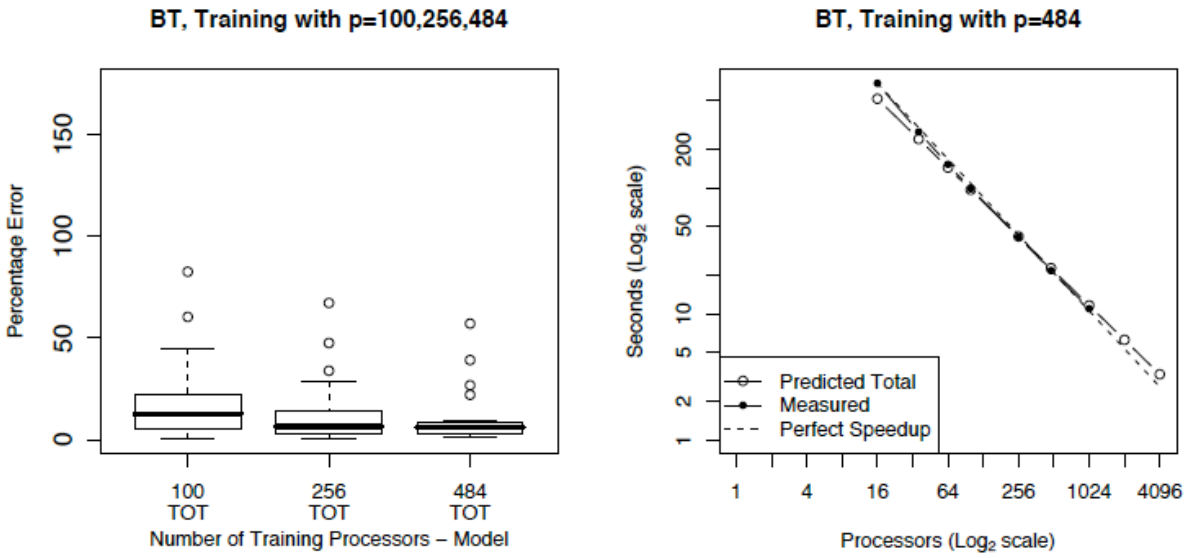

Figure 2.2: Results for predicting BT on 1024 processors. The right-hand graph shows results for *problem_size* = 500.

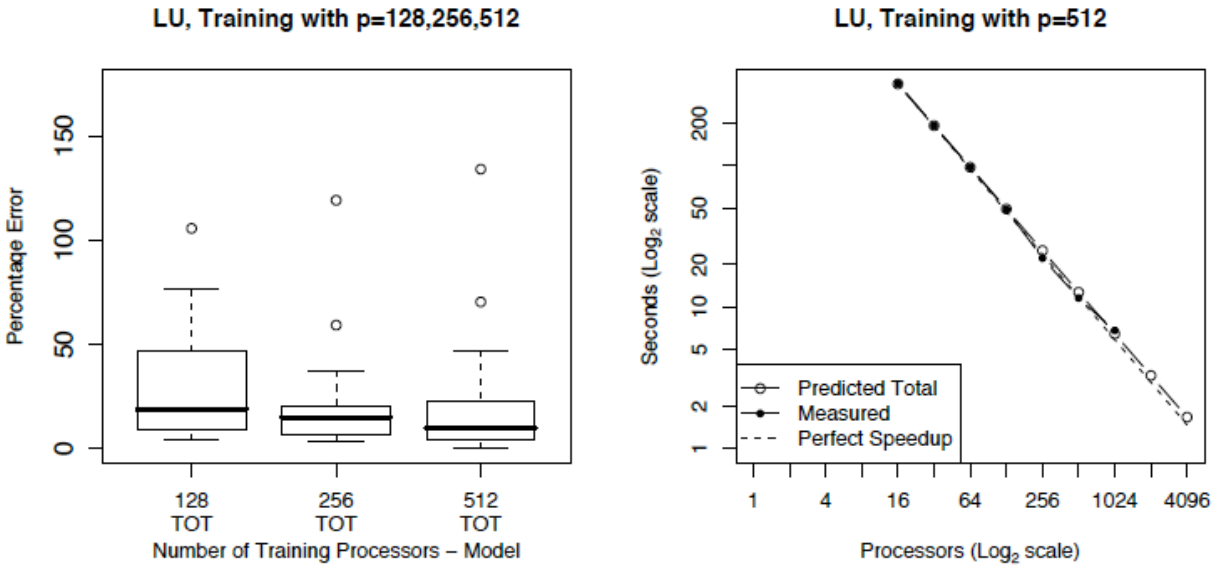

Figure 2.3: Results for predicting LU on 1024 processors. The right-hand graph shows results for all input variables equal to 450.

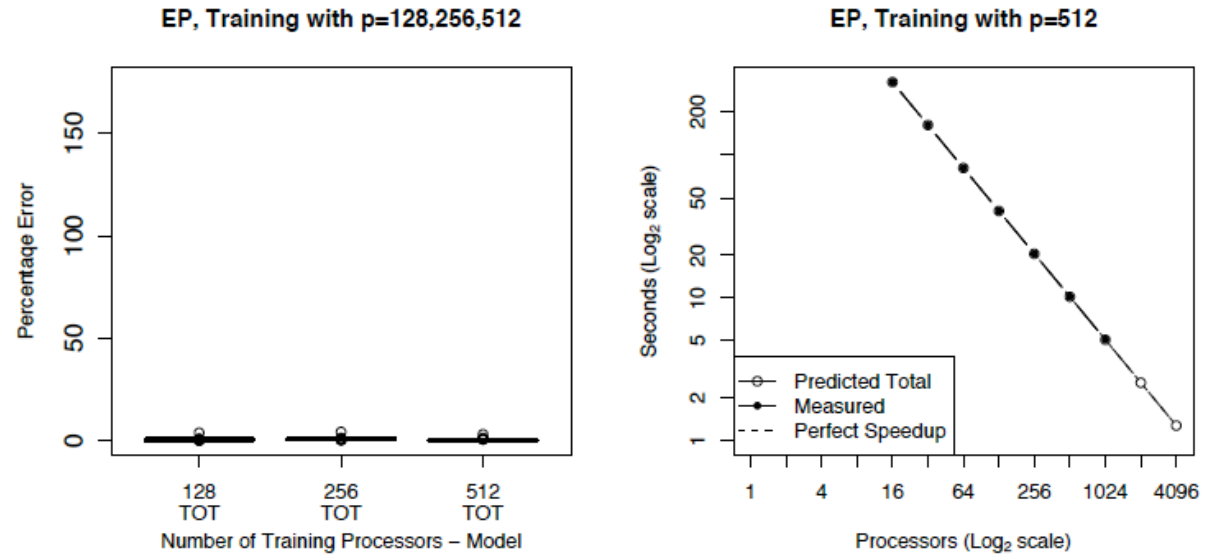

Figure 2.4: Results for predicting EP on 1024 processors. The right-hand graph shows results for $m = 2^{34}$.

Discussion: BT scales nearly perfectly because it has very little communication and a balanced computational workload. Because Atlas has no local disks to use for swap space, we were limited to relatively small input sizes for BT in order to fit the problem into the memory of smaller processor configurations, which limits the communication time. Thus, we omitted two runs with extremely small run times—less than one-half of a second; they are outliers with a large relative error, but are not indicative of any other results. Moreover, unlike many parallel programs, the communication time for BT actually decreases with an increase in the number of processors, meaning that the small amount of communication that is present using small numbers of processors will get even smaller when using larger numbers of processors. Thus, we conclude that regressing separately on computation and communication is not necessary for BT. In addition to BT, separate regressions are also not necessary for LU and EP. As mentioned earlier, we regress separately only when either (1) both computation and communication are significant or (2) communication is increasing.

2.3.4 Communication Intensive Applications

We next discuss applications for which there is a mix of computation and communication. Separating computation and communication is generally more important for these applications. This class includes four applications: CG, SMG, SP and Sweep3D.

CG overview: CG is significantly more difficult to predict than any of the compute-bound applications. The results are shown in Figure 2.5 and show several interesting characteristics of CG. First, the left-hand graph shows that when predicting

1024 processors and training with 256 processors, prediction quality is better than when training with 512 processors. The median error (for predicting total time using the best available fit, which is a quadratic) is 12% lower when training with 256 processors, and, when separating computation and communication, it is at least 10% better if $g(q)$ is chosen to be the same for both. Memory behavior causes this surprising result. For some input data sizes, the working set on 512 processors fits within the L2 cache, which we determined by using PAPI [14] to inspect the Opteron performance counters. On these tests, the number of L2 cache misses decreased by up to a factor of *six* (instead of the expected factor of two) when increasing from 256 to 512 processors and holding all other input variables constant. This phenomenon affects the fit (whether linear or quadratic) because the regression overcompensates: the predicted time by the regression is too low for the 1024 processor tests. This error occurs because the regression expects the ratio of the 256 to 512 processor tests to be fairly similar to that of the 512 to 1024 processor tests, but the superlinear memory hierarchy effects only occur in the former.

Second, unlike the compute-bound applications, communication in CG is significant, even though it also decreases as the number of processors increases. Because the computation and communication times decrease at different rates, treating them separately improves prediction quality compared to using a monolithic execution time. Overall, the median error decreases from 16% when using total execution time to 12% when separating the communication and computation for CG.

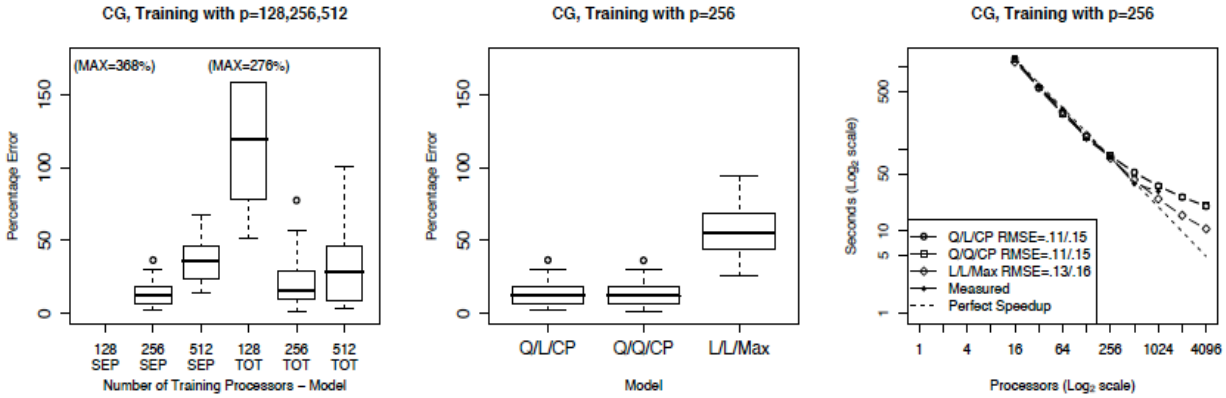

Figure 2.5: Results for predicting CG on 1024 processors. The right-hand graph shows results for *NA*=2M and *NONZER*=24.

SMG overview: SMG is written as a weak scaling application: the input parameters specify a single cube size for each processor. We converted SMG to strong scaling by reducing the cube's volume by half when we doubled the processor count. We could only test up to 256 processors since using more than one processor per node causes SMG to run out of memory quickly on smaller processor configurations.

Figure 2.6 shows the results for SMG. The left-hand graph shows that separating computation and communication makes a more significant difference than with CG (see also the discussion below). Unlike CG, increasing the number of processors used for training always helps because SMG does not have as pronounced differences in memory behavior. The middle graph shows that a quadratic regression for both computation and communication performs best.

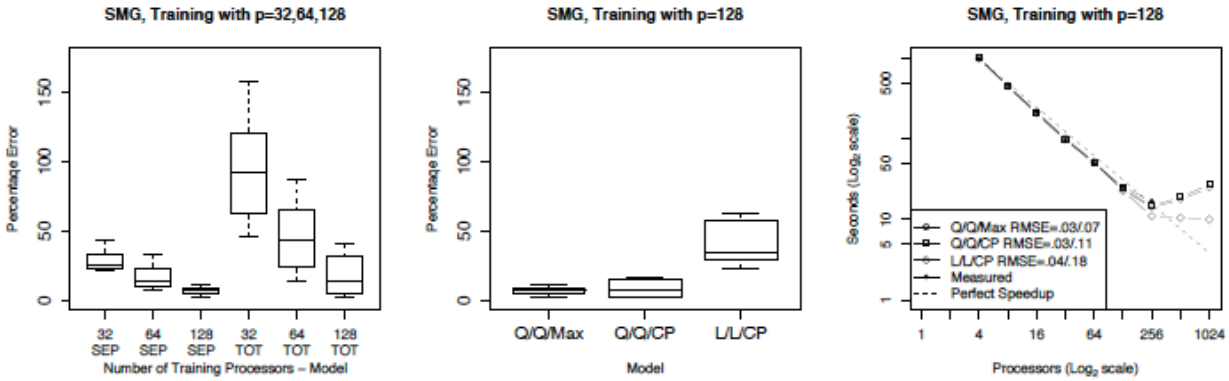

Figure 2.6: Results for predicting SMG on 256 processors. The right-hand graph shows results for *DIM1*=300.

SP overview: Figure 2.7 shows the results for SP. The results are quite good; prediction error is always within 13% and as close as 8%. The primary difference between SP and the other communication intensive applications is that separating computation and communication improves predictions only slightly. The one unusual aspect of our SP predictions is exactly the same as that of CG: predictions using smaller numbers of processors (100) is better than larger numbers (484). As with CG, our inspection of hardware performance counters show that this is due to memory hierarchy issues.

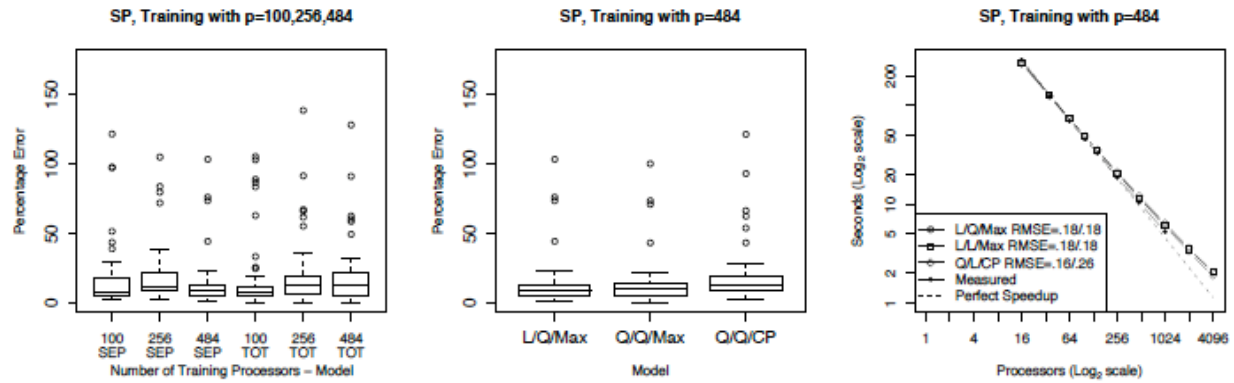

Figure 2.7: Results for predicting SP on 1024 processors. The right-hand graph shows results for *problem_size* = 450.

Sweep3D overview: Sweep3D is an application with which we intentionally stressed our prediction system, by choosing a *P* X 1 processor grid—which is not typical for this application. The atypical processor grid causes any processor rank larger than the x-dimension (ITG) of the data grid to be assigned no work and therefore remain idle. Figure 2.8 shows that while prediction quality is not as good for Sweep3D as for the other applications, it is reasonable. The best median error is 17%, but the significant algorithmic change caused by the atypical processor grid (which is Sweep3D-specific) limits the ability of RMSE to select this regression. Sweep3D demonstrates the challenges facing any black box system. Clearly, a regression-based approach will not generate high quality predictions in the (fairly unusual) case when program behavior is vastly different when increasing the processor configuration. With a program like Sweep3D—with a *P* X 1 processor grid—modest programmer input is needed for better predictions. We can probably limit this input to just the knowledge of when the granularity of work is too small to keep processors busy.

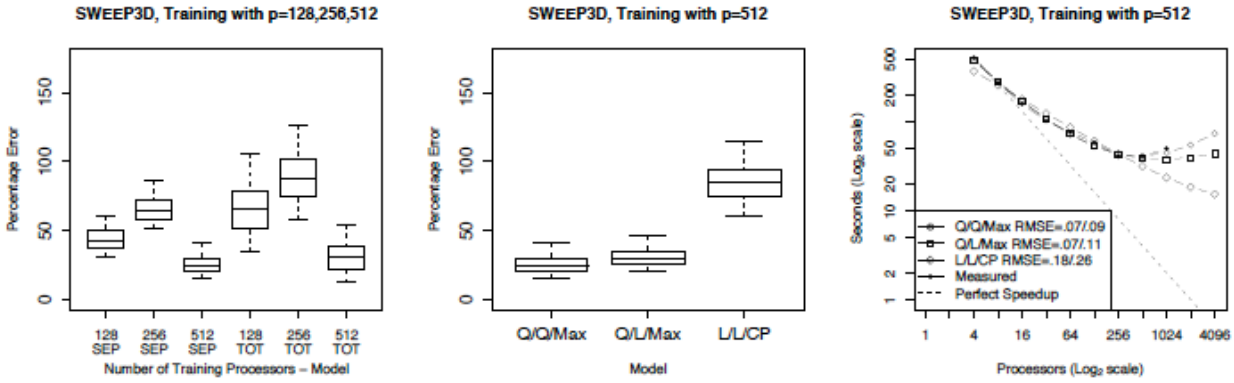

Figure 2.8: Results for predicting Sweep3D on 1024 processors. The right-hand graph shows results for $IT_G = 400$, $JT_G = 400$, and $KT_G = 400$.

Discussion: Generally speaking, training with as many processors as possible will produce a better fit. However, the case of CG shows that sometimes using fewer

processors for training can provide a more accurate result. Our system can detect when memory behavior is causing super-linear speedup, because the training runs can collect the relevant performance counters. However, the ramification might be more training runs since machines often have a relatively small number of performance counters and we are interested in not just L1 and L2 cache misses, but, for example, also local and remote references to memory caused by a NUMA architecture. More importantly, we can choose the best fit within a processor configuration based on RMSE even though we cannot easily use it to compare across different processor configurations. Through SP and SMG, we see two different extremes of the disparity in fit quality. For SP, using RMSE determined that we should use the critical path technique to separate computation and communication times. However, the RMSE difference, and the difference in fit quality, is small, as Figure 2.7 shows.

For SMG, the difference in fit quality is quite significant. The middle graph in Figure 2.6 shows that the difference between linear and quadratic for predicting 256 processors is much more pronounced than with CG. The RMSE is often over twice as high for a linear fit as a quadratic fit. The right-hand graph clearly shows that the quadratic fit predicts the observed times much better (7% median error using 128 processors to predict 256) than the linear fits. The worst fit has a median error over 30% and worst-case error of over 60%. Thus, even if using RMSE to select the regression method might provide only a small improvement, as with CG, selecting the regression method based on RMSE avoids cases with very large error.

Finally, training with up to 128 processors clearly produces poor results for CG. First, CG is the one case where separating computation and communication actually hurt

prediction accuracy (on 128 processors). As we have said, memory hierarchy effects are the problem. More generally, however, we need a method to determine the processor count required to produce good results a priori. We suggest predicting *within the training runs*. For example, in the case of CG, we train with up to 16 processors, and investigate how that predicts 128 processors. The results show significant error, an indication that training with up to 128 processors will produce poor results when predicting 1024 processors. As a comparison, we used the same method for LU, and predictions within the training set were quite good—and when 128 processors were used, 1024 processor predictions were accurate. However, we need more evidence to verify that this approach works consistently, which we leave to future work.

2.4 Summary

This chapter has presented a novel, black-box technique to predict parallel program scaling behavior. The basic idea is to use multivariate regression to predict the performance on large processor configurations using training data obtained from smaller numbers of processors.

Our results are encouraging. The error of our predictions for programs up to 1024 processors was in most cases 13% or less. We showed that we can formulate several different predictions based on the training data and that the one with the lowest root-mean-squared-error generally provides the best prediction. We also showed that understanding memory behavior—specifically the capacity of different memory hierarchy levels—is critical to making good predictions.

CHAPTER 3

USING FOCUSED REGRESSION FOR ACCURATE TIME-CONSTRAINED SCALING OF SCIENTIFIC APPLICATIONS

As mentioned in Chapter 2, several large-scale clusters now have hundreds of thousands of processors, and processor counts will be over a million within a few years. When more processors become available, scientists can take advantage of the extra resources in several ways. In the previous chapter, we looked at the strong scaling [69] option, where the scientist uses a larger number of processors to solve a problem faster. Strong scaling is the most frequent type that appears in computer science literature. However, time-constrained scaling [59], in which the scientist attempts to keep total run time constant, is becoming more commonplace. This approach solves larger problems and keeps the execution time the same on larger processor counts. Time-constrained scaling allows scientists to run problem sizes that were previously unexplored and is generally more intuitive from the scientist's perspective. While time-constrained scaling for simple applications seems simple (just increase the total problem size by the same factor as the number of processors), several factors complicate it in the general case. These factors include nonlinear effects in computation and communication, along with complex relationships between input parameters and execution time.

In this chapter we present an extension to our regression-based technique that allows accurate time-constrained scaling of applications. We have extended our system to

use a small amount of application-level information as input (gray-box) for more accurate predictions. Similar to our previous work, we choose a small series of training runs, varied over different, smaller processor counts. However, we now use focused regression to predict the input parameters that need to be used in order to achieve time-constrained scaling. The training runs always use a processor count no more than half of the target number; to reduce training time, iterative applications can be executed for just a few timesteps. The scientist (or compiler/run-time system) must indicate the number of input parameters, whether they represent the dimensions of the main data structure or are unrelated, and whether the processor grid is part of the parameterization. Our focused regression technique reduces the required training runs to a small number and also improves prediction accuracy.

This chapter makes two primary contributions. First, we provide a technique that the computational scientist can use to guide time-constrained scaling accurately. It builds on our prior work, which uses a black-box, non-focused regression approach to predict execution time using strong scaling (rather than time-constrained scaling). Second, we show that our focused regression technique makes accurate time-constrained scaling predictions with little (and often no) program-level information—predictions that are better in some cases by a wide margin compared to naive ones. Specifically, over all applications, median prediction error is within 13% including applications that exhibit a complex interaction between multiple input parameters and execution time.

The rest of this paper is organized as follows. Section 3.1 provides motivation for this work. Section 3.2 describes our statistical techniques, in particular focused regressions. Next, Section 3.3 describes our experimental methodology and results on

seven applications. Finally, Section 3.4 summarizes our findings and future directions.

3.1 Motivation

Strong scaling is preferred when a scientist must solve a specific problem as quickly as possible. However, the available parallelism is immutable and, therefore, strong scaling beyond a sufficiently large processor count will fail to reduce runtime. Time-constrained scaling, on the other hand, avoids limits imposed by Amdahl's law and allows scientists to solve problems at the limit of their system capacity. For example, a scientist often tries to run a problem twice as large when given twice as much computing power.

However, time-constrained scaling poses many difficulties. First, most scientists assume that the *data set size per processor* should be fixed as the processor count increases, which is usually referred to as *weak scaling* [69] and tries to keep computation time per processor constant. Due to communication time, though, weak scaling alone will not keep total execution time constant.

Second, even if communication is insignificant for a given application, proportionally increasing the problem is often not well defined. For example, consider an application that has a two dimensional data structure, defined by (global) dimensions N_1 and N_2, that is partitioned among the processors at a given processor count. Given twice as many processors, it is not clear how N_1 and N_2 should change.

Worse, the dimensions might not be correlated. In the above example, we knew that N_1 X N_2 should be doubled when the processor count doubles. Some applications do not have such an obvious relationship between the parameters (e.g., CG from the NAS

suite [5]).

Finally, overall execution time may not remain constant even when we know how to increase the problem size proportionally based on the input parameters. Computation time or communication time (or both) can increase at a greater than linear rate (which may not be obvious to even the experienced scientist). Figure 3.1 shows the complexities of time-constrained scaling for CG from the NAS suite. Here, *both* computation and communication times increase when holding *SIZE*/*P*, where P is the number of processors, constant for a given value of *NONZER*. In general, scientists would benefit from tools that help navigate through the complexities of time-constrained scaling.

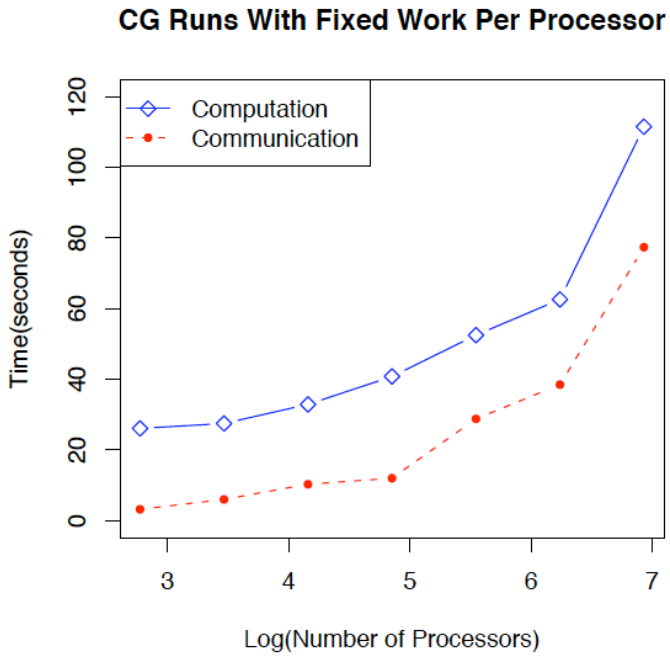

Figure 3.1: Computation and communication times for CG as the number of processors increases. The ratio of *SIZE*/*P* is fixed; *SIZE* ranges from 46,094 to 2,950,000, and *P* ranges from 16 to 1024 (shown in Log_2 scale). The value of *NONZER* is held constant at a value of 20.

3.2 Focused Regression

This section discusses our focused regression technique. First, we describe the general idea. Then, we discuss our basic model, which does not require any program-level information. Finally, we discuss extensions that provide greater accuracy for more complicated applications.

3.2.1 Overall Technique

The scientist must provide appropriate inputs for our technique. Our current prototype requires the scientist to present the application and input parameters used on the largest processor count that is smaller than the target number of processors (denoted P_t). For example, in this paper P_t is always 1024, so the scientist must present the input parameters used on the 512-processor version. In addition, the scientist must provide certain application-level information, which Table 3.1 shows and we describe further below. The output is the set of input parameters—or sets, when there are multiple input parameters—that will result in application run time that is equal to that of the program executing on P_t /2 processors. To find these parameters, we must in part run experiments on smaller numbers of processors. While we expect that some of these experiments will already be run (e.g., the scientist has run the program with the desired input parameters on 512 processors, and now wants to scale to 1024), a few others usually must be executed. To control training time, these executions cover only a limited number of time steps. Therefore, we assume the time step loop is known.

With the value of P_t and the input by the scientist, our technique proceeds as

follows. For simplicity, we first present the case with only one input parameter. We assume that the scientist provides us with data from points with time ≈ T, at both $P = P_t$ /2 and $P = P_t$ /4. Then, we sample points (assuming that this data is not made available by the scientist) where the times are ≈ 1.1 X T and ≈ 0.9 X T. We determine the appropriate value of the input parameter to achieve this through inspection of the data that the scientist provides. From this point, we use the techniques described in the next two subsections (basic and general regression models) to predict the value of the input parameter on P_t processors that will result in an execution time of ≈ T. We extend this technique to multiple input parameters with the procedure described in Section 3.2.3.

Program	Parameter Relatedness	Processor Grid Used
BT	Yes	No
LU	Yes	No
SP	Yes	No
CG	No	No
Miranda	Yes	No
SMG	No	Yes
Sweep3d	Yes	Yes

Table 3.1: Application-level information needed from the scientist for our seven programs

3.2.2 Basic Model

In order to determine the proper input parameters for constant run time at P_t, we need a model that predicts total run time T of a given application. This model expresses T as a function of the values of its input parameters and P_t. Aside from P_t, the size, denoted W is the other key characteristic for determining run time in programs with little

communication. We can often easily determine W for simple programs from the input parameters (i.e., we can just use the product of the parameters (z_i's), or $W = f(z_1 \text{ X } z_2 \text{ X. . . } z_n)$), and T is approximately proportional to W/P. More generally, we have

$$log_2(T) = \beta_0 + \beta_1 log_2(W) + \beta_2 log_2(P_t) + \varepsilon \qquad (3.1)$$

where β_0, β_1, and β_2 are coefficients that we estimate based on a set of observed (T,W,Q) triplets, Q is the number of processors used in a training run ($Q < P_t$), and ε is the error. Generally, we estimate the β values so as to minimize the error between the predicted values and the observed values.

Specifically, to collect the (T,W,Q) triplets, we execute the program on Q processors, where $Q \in \{2, \ldots, P_t/2\}$. We vary the values of W and Q on the sample runs and then use regression to generate Equation 3.1. Because it is easier to acquire Q processors than P_t, it is reasonable to perform multiple instrumented runs for different configurations of the input variables.

We predict run time using a log-scale because the prediction errors are well known to be proportional to the expected time—we are concerned with relative errors. Working in log-scale implicitly handles this. As in the previous chapter, the base of the log makes no fundamental difference; we use log_2 for mathematical convenience. The coefficients β_1 and β_2 in Equation 3.1 measure the relative increase in time due to changes in computation. Finally, working in log-scale implicitly handles interactions between the different terms in Equation 3.1 (e.g., time is proportional to the quotient of W and P_t).

While the model in Equation 3.1 is relatively simple, it works well for computation-dominated, simple-array based applications. The applications upon which

we evaluate our focused regression technique in this paper (see Section 3.3) are listed in Table 3.1. The first three are well predicted with Equation 3.1: BT, LU, and SP (from the NAS suite [5]). All three have a high computation-to-communication ratio and a single input parameter.

3.2.3 General Model

For more complex applications, simply applying Equation 3.1 is insufficient for three reasons. First, we cannot easily determine size (W) in advance in some applications. Rather, we can specify the values of some input parameters in advance, and these parameters determine computation in an unknown or complex way. In such cases, one may need to examine several potential predictor parameters to determine which ones are significant predictors of time and to model the relationship between T and these variables. For example, CG has this characteristic, as indicated by Table 3.1.

Second, modeling only total execution time produces inaccurate predictions for applications with a significant amount of time spent in communication. Computation and communication can scale at different rates, which the training runs capture only if we model them separately. As mentioned earlier, Figure 3.1 shows this situation. Currently, we do not subdivide either computation or communication further into phases because the prediction quality we achieve for our applications is usually accurate without it. We are currently investigating breaking computation and communication into smaller phases. For example, we could break communication calls into groups that have similar scaling behavior (e.g., logarithmic-scaling collectives versus linear-scaling collectives).

Third, for applications in which the program specifies a processor grid to allow

the scientist to control data distribution, T is not only a function of P_t, but also of $P_1, P_2, \ldots, P_n$, where n is the number of processor grid dimensions and P_t is the product of the P_i values. Both SMG and Miranda fall into this category. Parameterizing this aspect to some extent requires knowledge of the program structure. We can obtain significantly better fits when we have this information by using the values of the processor dimensions rather than just P. In such cases, our models can give scientists insight into the processor configurations, for a fixed W and P_t, that run fastest, in addition to providing time estimates for various input combinations.

Our prototype handles each of these possibilities.

Case 1: We use a more general equation for execution time with complex input parameter relationships:

$$log_2(T) = \beta_0 + \beta_1 z_1 + \beta_2 z_2 + \ldots + \beta_n z_n + \beta_{n+1} log_2(P_t) + \varepsilon \qquad (3.2)$$

Here, z_i is the i^{th} input parameter describing the data. We use additional training runs to determine which of the z_i are important in predicting T, as well as to model the functional form of these variables (similar to what was done by Lee et al. [41]).

Case 2: If communication is significant, we use separate regressions for computation and communication. Both follow the same form of either Equation 3.1 (if the input parameters are related) or 3.2 (if they are not, as in case 1 above). Our current prototype splits the regressions only if the percentage of time spent in communication is greater than 50% at the largest number of processors used for training (512); we found that regressing only on total time suffices with smaller percentages. We collect computation and communication time using the PMPI profiling layer of MPI.

Case 3: The most interesting case occurs when the application uses a processor grid. We

considered simply extending Equation 3.2 by replacing the P_t term with terms for the processor grid terms (e.g., P_1 and P_2). However, while intuitive, experiments showed that this technique is ineffective because the data distribution, as specified by the processor grid, significantly affects application execution time in a nonlinear manner, as we show in Section 3.3. Thus, using a single regression results in significant errors.

Instead, we restrict the sample runs used in the regression to a narrow range or *focal region* around the processor grid at the target number of processors, P_t. In general, the focal region is trivial when the number of input parameters is small (e.g., 1); in this case, using a fixed execution time to determine the focal region suffices. However, for nontrivial applications with several input parameters, such as SMG and Sweep3d, we must determine a focal region based on the input parameter space since that space is large, and it is quite difficult to cluster sample runs around execution time.

We then use Equation 3.1 or Equation 3.2 in the focal region, depending on whether or not the input parameters are related, as described above. The typical strategy when creating regression models uses more data to achieve a better result. However, in our particular case, more data is worse, if it is not nearby in the processor dimension space on $Q < P_t$ processors. Also, while the focal region idea is quite useful and necessary when handling an application that uses a processor grid, it also improves regression quality for all applications. Therefore, we use the focal region idea in general—restrict tests to those around the values of the input parameters (adjusted for processor count) presented by the scientist. Using only a subset of available data for prediction via regression is not new; for example, Lee and Brooks use this technique for predicting performance and power [42]. We apply this technique to scalability analysis.

Consider an example, with one of our applications, SMG, that has six input parameters—three processor dimensions, p_x, p_y, and p_z, along with three grid dimensions, n_x, n_y, and n_z. We next illustrate what predictions our prototype makes, along with what focal region it selects to make each prediction. Suppose the scientist has run SMG on 512 processors using a processor grid where $p_x = 1$, $p_y = 16$, and $p_z = 32$, denoted for convenience as (1, 16, 32). We assume that if the scientist wants to use time-constrained scaling of SMG to 1024 processors, then it is necessary to double one of the three processor dimensions.

For each prediction, we use a *different* regression based on experiments in the focal region. Figure 3.2 shows two different focal regions, one of which, (1, 32, 32), we would use in the preceding example. The figure shows that our prototype uses those processor grids (shown in black) at lower (total) numbers of processors that are most proportionally similar to the grid at the target number of processors. As the results in the next section show, the results degrade if we include data from grids that are not proportionally similar. Our figure sets $p_x = 1$, because if we also vary p_x, the picture becomes quite complex. However, our prototype handles the general three-dimensional case.

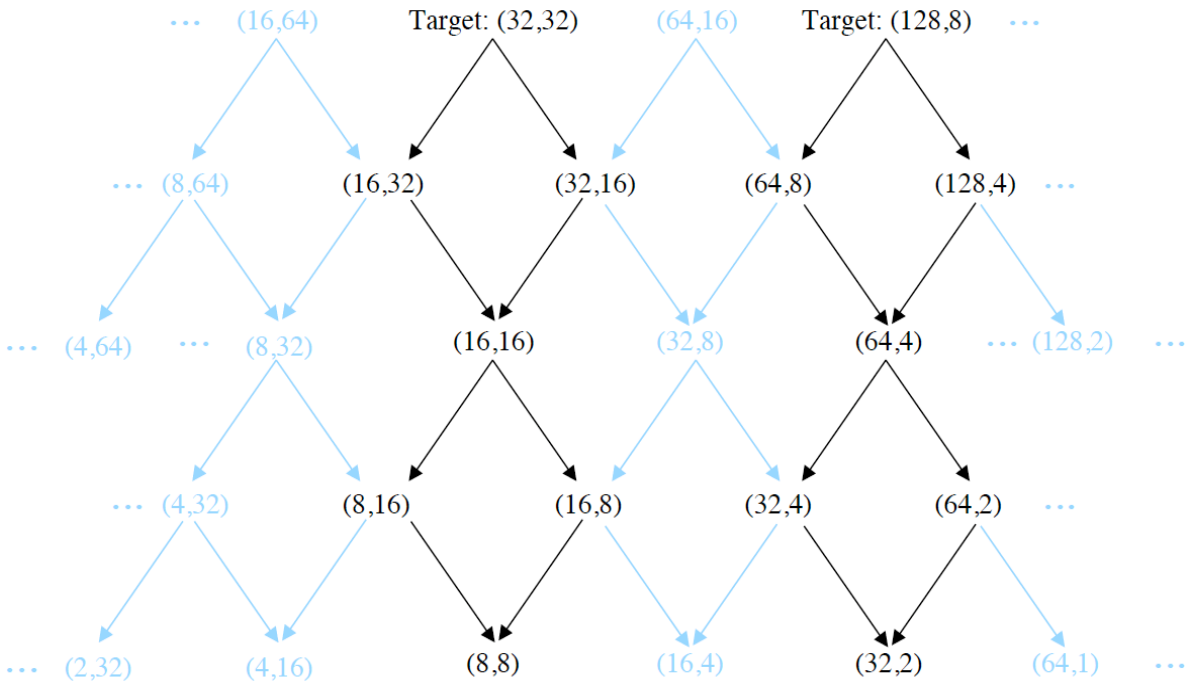

Figure 3.2: Processor grids (only shown down to 64 processors) used in SMG to predict $P_x = 1$, $P_y = 32$, $P_z = 32$, and $P_x = 1$, $P_y = 128$, $P_z = 8$, respectively. For all vertices in the graph, $P_x = 1$. The two values listed are P_y and P_z.

3.3 Results

This section discusses results of our focused regression prototype in making time-constrained scaling predictions. For our evaluation we used two different clusters at Lawrence Livermore National Laboratory: the *Atlas* cluster and the *Hera* cluster. The former is the same as described in Chapter 2, while the latter has 864 four-socket, quad-core AMD Opteron nodes with 32 GB RAM. We used Hera (which is similar to Atlas) to execute Miranda because of time constraints on Atlas. Each Opteron node is a NUMA architecture; each socket has local memory, and all others are accessed through longer-access remote memory controllers. Our experiments use four cores on each node (one per

socket on Atlas and Hera) to avoid potential variance if all cores are in use [52]. Note that in the rest of this section, we use the term processor to refer to a core.

To eliminate potential NUMA effects, we used cpu_bind to ensure that Linux allocates memory for each core out of the socket's local memory. Without binding, Linux may allocate remote memory (arbitrarily), which introduces significant variance across runs.

3.3.1 Methodology

Our prototype collects results for each instrumented training run; these runs occur on a variety of processor counts, but never on the target processor count (P_t). We use the PMPI layer to collect computation and communication times; we count any time in the MPI library as communication time. While this is not completely precise, getting finer-grain results (e.g., omitting blocking time and collecting only network and copying time) requires instrumenting the entire MPI library. We then use measured execution times to fit a linear model. We use the statistical package SAS for all regressions.

We emphasize that we run the program only on a small subset of the many possible input parameter/processor combinations; this choice conserves machine time as well as produces better results by using focal regions (as described in the previous section).

We make the important assumption that we can run an application with the input parameters set to values of our choosing. Essentially, the parameter space is quite large and sparse for applications like SMG (5 free parameters). This ability to execute the program in configurations of our choice ensures that we can collect the data that we need

to make accurate predictions. Essentially, we assume that scientists write programs that are flexible and provide meaningful timing results, if not physical results, for any combination of the input parameters.

For our evaluation we executed the program at the target processor count (1024 processors), and we find the input parameters that are predicted to cause the program to run in the same time as for a 512-processor run (which is the goal). We measure effectiveness by reporting error based on the relative difference between the observed execution times on 1024 and 512 processors.

3.3.2 Applications

We tested our techniques using seven applications. Four are from the NAS suite [5]. In particular, we use BT, SP, CG, and LU (described in Section 2.2); our approach does not apply to other NAS programs (FT, IS, MG) because we have insufficient input data due to input restrictions. Further, EP is trivial because it has only one parameter and zero communication. We further use SMG and Sweep3d from the ASC suite; the former is a three-dimensional multigrid solver, and the latter is a three-dimensional neutron transport code. The final application is Miranda, which is an industrial-strength hydrodynamics application.

3.3.3 Summary of Results

Overall, our prediction quality is quite good: median prediction error ranges from 3% to 12.2%, and predictions are almost always within 20% and usually much better. For the three more complex applications, we must generate different regressions for different

focal regions to achieve accuracy. In particular, we obtain a median error as high as 75% if we do not use a focal region.

3.3.4 Single Parameter Programs

First, we studied three programs that have only one important parameter (other than p): BT, SP, and LU. These programs are computation intensive; they serve as programs for which the scientist could perform accurate time-constrained scaling in a straightforward manner. Proportional scaling, which we define as increasing the parameter by an identical factor as the number of processors increases, will be relatively effective.

Table 3.2 shows the results of all three programs. Focused regression produces predictions within 6% of the actual time, whereas predicting using simple proportional scaling of the single input is over 17% for BT and 11% for LU. For SP, proportional scaling is slightly better, 3.6% to 5.2%, but both predictions are quite good.

These results show focused regression performs well and avoids the larger errors incurred by proportional scaling. More importantly, it shows that performing time-constrained scaling even on seemingly simple applications is not necessarily trivial.

Program	Focused Regression Error	Proportional Scaling Error
BT	1.8%	17%
LU	5.3%	11%
SP	5.2%	3.6%

Table 3.2: Percentage error between actual and predicted times for one-parameter programs (BT, SP, and LU) when using 512 processors for training. For reference, the error when scaling proportionally is shown. All predictions are for programs executing on 1024 processors.

3.3.5 Multiple Parameter Programs

Next, we studied four programs that have at least two important parameters: SMG, Sweep3d, CG, and Miranda. All of these applications serve as challenges for our focused regression approach; time-constrained scaling is difficult either because the parameters have nontrivial interactions or the application specifies processor grid dimensions. We compare our results to an approach, denoted *non-focused*, in which we use all the sample runs below 1024 processors to create a single monolithic regression. We study SMG first and in depth because it presents the most challenges.

***SMG**:* SMG has six input parameters: three processor dimensions, p_x, p_y, and p_z, along with three grid dimensions, n_x, n_y, and n_z. The application specifies grid dimensions in terms of a per processor local grid; one can recover the global grid by taking the product of each grid dimension with the associated processor dimension. For time-constrained scaling, four of the six input parameters are unconstrained, which still leaves many different ways to scale SMG. Note that SMG is not symmetric in all dimensions [13], so modeling it is not at all straightforward.

We chose to scale the global grid equally in all three dimensions (e.g., if we double the processor count, we increase each global grid dimension by a factor of $\sqrt[3]{2}$), which corresponds physically to decreasing the grid point resolution by a factor of 2. Furthermore, we assume that if the user is scaling a program with processor dimensions p_x, p_y, and p_z, that one of these dimensions will increase by a factor of 2. Therefore, we make predictions for all three possibilities.

As described in Section 3.2, we must create a regression for different focal

regions with SMG. Specifically, a different regression predicts each processor configuration. We predict input parameters for a range of input sizes for six of the possible processor configurations (focal regions) at 1024 processors for these results.

Figure 3.3 shows the median prediction errors using focused and non-focused regressions for SMG, and Table 3.3 summarizes the results across all applications.

In the particular case of SMG, using focused regressions allows accurate predictions, while the non-focused technique is clearly inferior. Also, the median error is just 5.6% for the points predicted. The non-focused technique has median prediction errors that are higher (76%). Furthermore, the worst case has an even larger disparity—up to 117% with the non-focused approach. While the worst case for focused regression is 34%, we note that 90% of the predictions are within 10%.

Finally, we do not give the prediction error when using proportional scaling for SMG because we lack a clear definition of proportional scaling with six input parameters, and some parameters (the processor grid dimensions) have strict restrictions on their values.

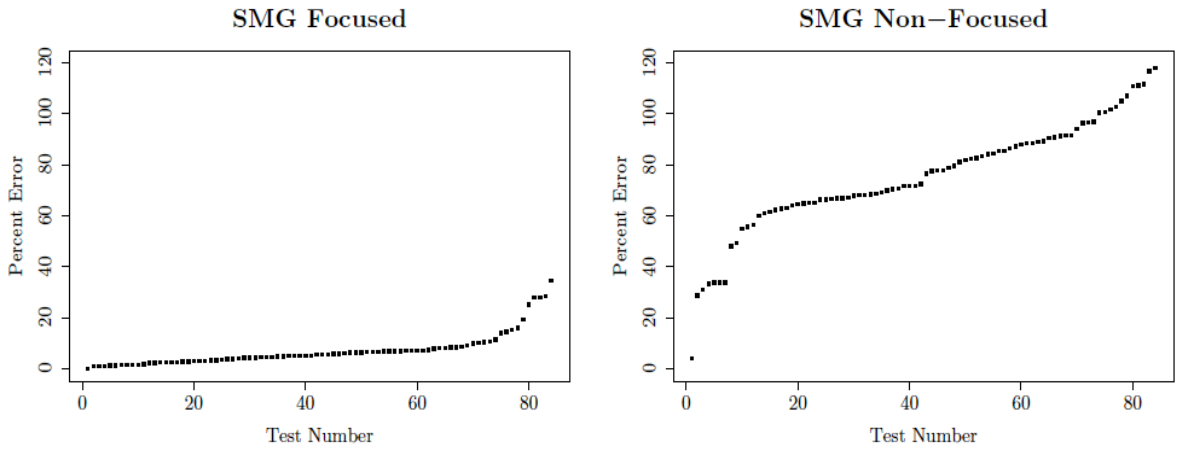

Figure 3.3: Scatterplots showing prediction error for focused and non-focused regressions for SMG. The tests are ordered by percentage error.

Prediction	SMG			Sweep3d		
Error (%)	Max	Avg	Median	Max	Avg	Median
Focused	34	7.1	5.6	12	4.9	5.0
Non-focused	117	75	76	53	33	36

Prediction	CG			Miranda		
Error (%)	Max	Avg	Median	Max	Avg	Median
Focused	22	12	12	20	3.7	2.2
Non-focused	53	27	27	21	3.7	3.2

Table 3.3: Maximum, average, and median prediction error in SMG, Sweep3d, CG, and Miranda for focused and non-focused regressions.

Sweep3d: Sweep3d has fewer input parameters (five) than SMG (six) because it has a two dimensional processor grid. Also, the specification of the grid is global, not local. Three of the five input parameters are unconstrained for time-constrained scaling. Thus we can scale Sweep3d in many ways, as with SMG. We choose the same approach for scaling as for SMG and use focal regions in exactly the same way.

Figure 3.4 shows the results for Sweep3d and Table 3.3 summarize the results.

The results are similar to those of SMG; the median prediction error is quite low for our focused regression (5.0%) and poor for the non-focused regression (36%).

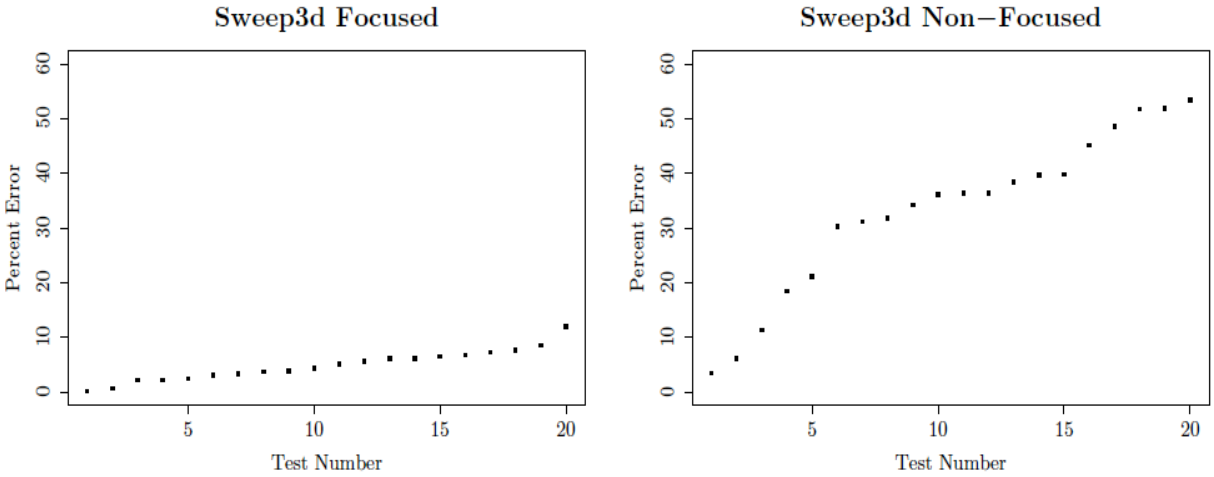

Figure 3.4: Scatterplots showing prediction error for focused and non-focused regressions for Sweep3d.

***CG**:* Figure 3.5 shows the results when applying focused regression to CG, and Table 3.3 summarizes this data. The figure shows that we produce predictions with median error of 12% and worst-case error less than 23%. For comparison, we also show the error when using a non-focused regression—for CG, we focus the regression on different values of the *NZ* input parameter, along with splitting computation and communication and regressing on them separately. Prediction quality is much better with focused regression. Figure 3.6 shows the effect of using our prototype to predict *SIZE*, as opposed to using naive weak scaling.

We also further investigated the naive time-constrained scaling prediction. However, the question is: how would the scientist scale CG to keep the execution time constant without our approach? As we mentioned earlier, CG has two parameters: *SIZE* and *NONZER*. The scientist has three intuitive potential choices to scale CG: double

SIZE, holding *NONZER* constant; double *NONZER*, holding *SIZE* constant; or increase each by $\sqrt{2}$. We ruled out the third case for two reasons. First, increasing both parameters by $\sqrt{2}$ seems physically unrealistic since CG is at its core a one-dimensional data structure (sparse matrix). Second, CG requires integers for both input parameters, so increasing *NONZER* by $\sqrt{2}$ will lead to experiments that we cannot actually run.

Therefore, we investigated the first two possibilities. When doubling *SIZE* and holding *NONZER* constant, the average error is 53%; when doubling *NONZER* and holding *SIZE* constant, the average error is 13%. Both are worse than the average error with focused regression, and the potential for large error exists.

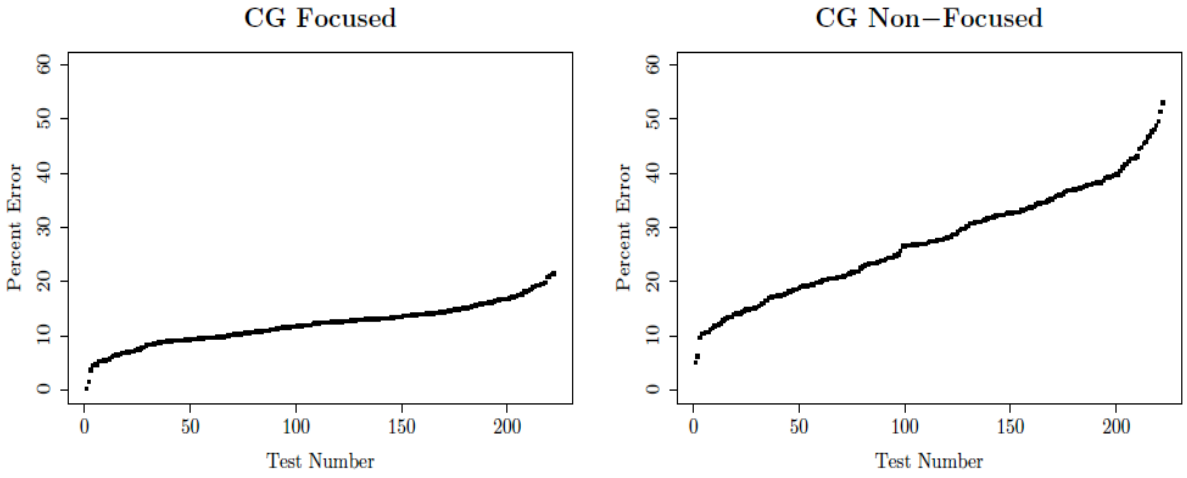

Figure 3.5: Scatterplots showing prediction error for focused and non-focused regressions for CG.

P	Size (Wk Scaling)	Time (s)	Size(Prototype)	Time (s)
16	46,094	29.3	46,094	29.3
32	92,188	33.5	78,682	27.9
64	184,375	43.2	124,979	27.4
128	368,750	52.8	237,656	31.4
256	737,000	81.3	299,536	28.7
512	1,475,000	101	458,171	29.2
1024	2,950,000	189	558,273	29.7

Table 3.4: Time-constrained scaling with our prototype for 16 through 1024 processors. (with *NONZER* fixed at 20). Our prototype predicts the value of *SIZE* at 1024 (given a set of experiments using 16 through 512 processors) that will match the time at 16 processors, which is 29.3 seconds. For completeness, we also show the predicted values for 32 through 512 processors. Clearly, our prototype leads to time-constrained scaling, while naive weak scaling does not. Figure 3.4 (Below) displays these results graphically.

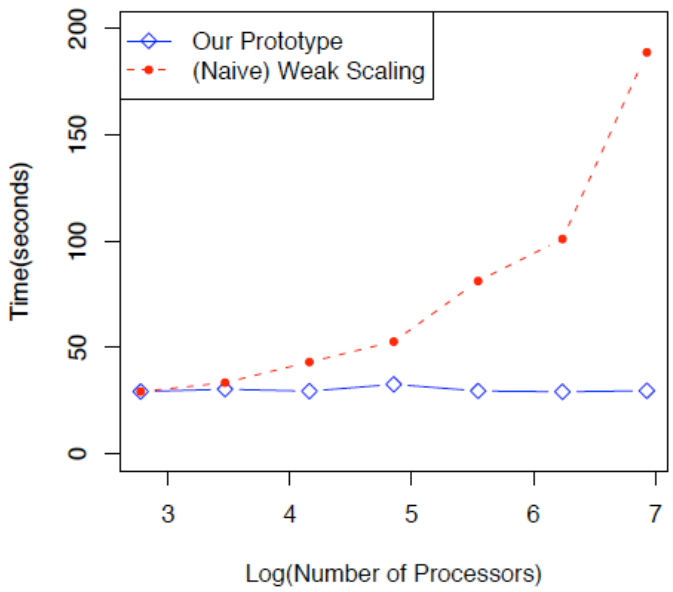

Figure 3.6: A graphical representation of the results presented in Table 3.4

***Miranda**:* Figure 3.7 shows the results from Miranda for both focused and non-focused regressions, and Table 3.3 summarizes this data. In this case, we vary only two processor grid dimensions, which substantially reduces the number of processor grids at 1024 processors.

The data shows that *either* technique achieves good prediction quality. The median is slightly better when using the non-focused approach, while we have fewer prediction errors over 10% (17 to 11). Recall, however, that for SMG, prediction quality was much better with focused regression, and the non-focused regression produced consistently poor results.

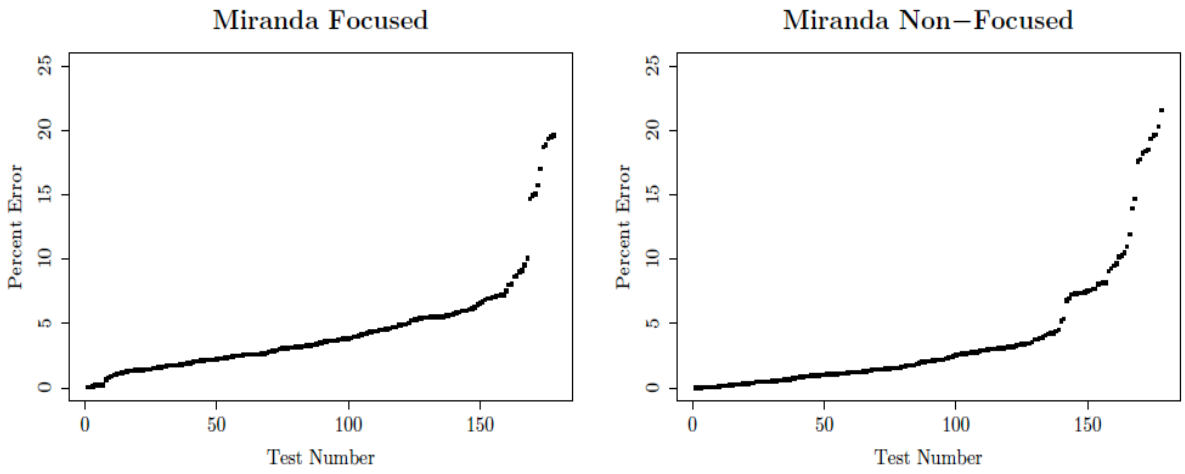

Figure 3.7: Scatterplots showing prediction error for focused and non-focused regressions for Miranda.

3.4: Summary

This chapter has described the design, implementation, and evaluation of an approach that uses *focused regression*, which assists computation scientists in scaling

their application so that execution time is kept constant. Our approach only requires the application scientist to provide a small amount of application-level information—specifically whether the input parameters are related and if the application uses a processor grid. Our approach then provides values of input parameters that will yield approximately the same execution time on a larger number of processors. Notably, our technique never requires a run of the application at the target scale.

CHAPTER 4

PREDICTION IN THE PRESENCE OF RUNTIME VARIANCE

With a balanced workload, a deterministic algorithm, and dedicated nodes, one would expect only slight variation in execution time (runtime variability) when running multiple replications of an application with the same input parameters. However, when running the original tests for our work in Chapters 2 and 3, we observed significant performance variability, especially at large scale. Depending on the complexity of the system, different sources of variation may exist, including non-uniform memory access (NUMA) effects, operating system activity, network contention, and even the layout of the nodes assigned for a given run. We minimized variability in previous chapters by leaving a core idle to handle system processes and by binding each process to a single core to avoid potential NUMA effects. By adding these constraints, we were able to minimize (but not eliminate) the prediction error caused by noise. However, constraining the runtime environment and leaving cores idle is not a viable solution to the noise problem. In this chapter, we explore the difficulties of prediction in the presence of runtime variability and discuss our approach to make accurate predictions when noise is present on the system.

Without runtime constraints, large-scale applications are prone to significant variation. Figure 4.1a shows the variation of a single instance of Sweep3D executed 65 times on a range of processor counts from 4 to 128. The percent variation on most

processor counts is +/- 7% of the mean for that processor count. However, on 64 processors, the runtime ranges from 5.58 seconds to 10.11, which is approximately a +/- 35% variation about the mean. Variability of this magnitude is not surprising, as other research shows up to 100% performance degradation due to noise on the Parallel Ocean Program (POP) [27].

Significant runtime variability causes problems for performance prediction and leads to degraded accuracy for scalability models. Figure 4.1b demonstrates the effect of noise on our prediction models presented in Chapters 2 and 3. In this example, we select a single Sweep3D experiment out of the possible 65 tests in Figure 4.1a at each processor count (including the maximum observed runtime at 64 processors). A single test is selected because the techniques presented in earlier chapters require a single experiment at each processor count. We model the data using a quadratic regression up to 64 ($p/2$) processors to predict at $p = 128$. The test selected on 64 processors causes the regression model to curve upward, resulting in a prediction of 8.44 seconds for a test that actually takes only 3.5 seconds to run (141% relative prediction error). In an environment with minimal noise, the small decrease in runtime from 32 to 64 processors indicates the point at which speedup begins to level off and at which it becomes inefficient to run on more processors. However, in this case, the regression model is altered by a test that is highly impacted by noise on the system. While it is true that noise of this magnitude was not present in earlier chapters, small amounts of noise (including operating system activity) may have degraded the accuracy of our predictions. In order to model scalability on increasingly complex supercomputer architectures accurately, we must consider the impact of noise on application runtimes.

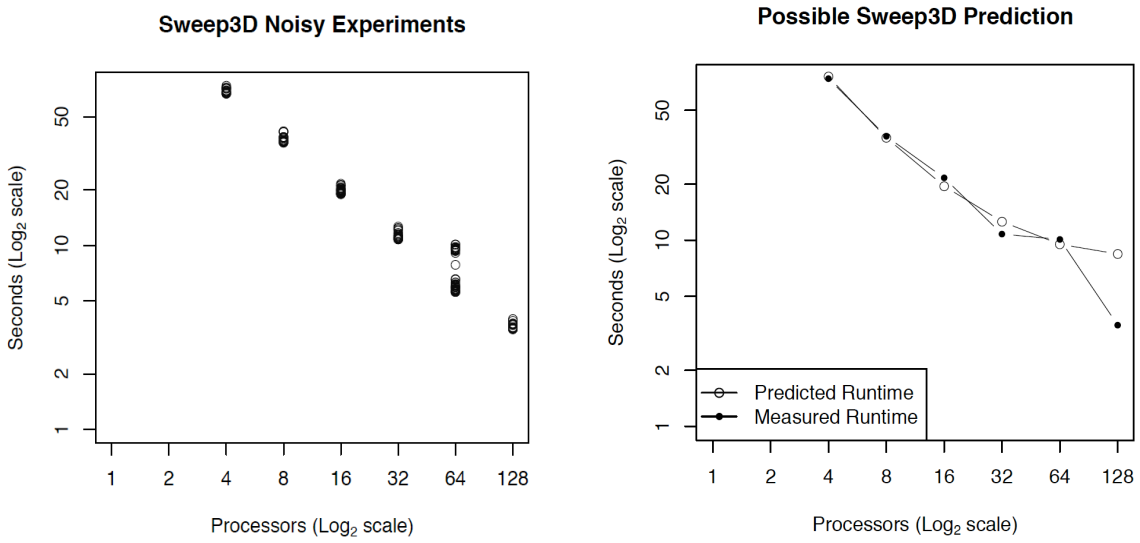

Figure 4.1: Sweep3D experiments run in a noisy environment using a 200x200x200 processor grid on various processor counts. Figure 4.1a (left) shows all 65 tests run on each processor count. Figure 4.1b (right) shows the effects of noise on scalability predictions. We predict 128 processors using data from 4 through 64.

In principle we can modify our system to provide the user with a range of runtime predictions at large scale using runs from small processor counts. This approach would require many replications of each test at all processor counts in order to gather the approximate minimum, maximum and probability distribution of runtimes for each test. The system could then use regression techniques to show the projected range at large processor counts. However, the amount of training runs required makes this solution impractical. For example, non-replicated training tests took roughly 24 hours for the CG predictions obtained in Chapter 2. Repeating these tests enough times to capture all types of variability on the machine could take months or even longer. A more practical starting point is to predict the distribution of execution time accurately for a given processor count before moving to the more complicated problem of extrapolation. Our approach consists of gathering many execution time samples from a single iteration (either by

simulation or real application runs) of an iterative application and then predicting the execution time of the same application run for many iterations. Both the single iteration tests and the long-running tests are executed in a noisy environment (either simulated or real system). Solving this first allows us to focus on noise effects within a given processor count without extrapolation, and it also serves as a building block for a large-scale extrapolation prediction system. The ability to understand the noise effects of an application running for many iterations by running the application for only a single iteration will greatly reduce the amount of time required to run training tests on an extrapolation-based system.

This chapter makes three important contributions. First, we present two methods for gathering single-iteration runs of scientific applications. One is by the use of simulation techniques, and the other is by running many single-iteration tests on the actual systems. We also demonstrate how the Central Limit Theorem can be used to predict application runtime in the presence of runtime variability of multiple iterations using only information about single-iteration runs. Finally, we discuss reasons for prediction error and demonstrate how various sources of noise can affect application runtime.

The rest of this chapter is organized as follows. Section 4.1 outlines the types of noise that can be found on modern supercomputers. Next, Section 4.2 provides an overview of the two methods for gathering single-iteration information and discusses our prediction formula, based on the Central Limit Theorem. Section 4.3 discusses our approach to predictions using simulations. Then, Section 4.4 demonstrates predictions

made using real runs on the system. Finally, Section 4.5 summarizes our findings and future directions.

4.1 Sources of Noise on Modern Supercomputers

The complex architectures and network interconnects of modern supercomputers result in many potential sources of runtime variability on the systems. We outline a few of the main sources of variability in this section.

Operating System Activity - This type of variability is caused by system events that interrupt the execution of the application. It is the most well-studied type of noise found in the literature. Most authors on the subject recognize two types of operating system interference: short-lived timer interrupts (15-30 us) and longer-lived daemon processes. Type 1 includes the OS tick, which keeps up with the clock for process scheduling. Type 2 (daemon processes) includes servers that may be running on the system (sshd, httpd, etc.), monitoring daemons, cron jobs, and many others. These can run up to a second or more [33]. If a core is not left unused to process these events, they can introduce significant delays to a large-scale application. These delays have a detrimental effect on the application runtime at large scale due to the increased probability of a "straggler" process at each synchronization point.

Non-Uniform Memory Access - This type of variability is caused by references to non-local memory, which cause a process to delay on each remote reference. When multiple processes have different numbers of remote references throughout an application run, runtime variation results. In our experience on the Atlas cluster at LLNL, remote memory references can happen for three reasons. First, they can be the result of an

overflow on a memory module, which occurs when the processes are not balanced across the cores and the total memory requirements for a given processor exceed the available local memory (large problem sizes increase the probability that this kind of variability will occur). The second cause of remote references is when a task migrates away from the core on which it has allocated its data [50], which can happen at any point during the execution of the application and may result in varying numbers of remote references (the probability increases with processor count). The final source of remote references, which we have not experienced on the Atlas cluster, is when a process assigns data remotely even though local memory is free.

Network Contention - When other applications are executing simultaneously on the cluster, they may create traffic on the network. The amount of traffic (which changes across runs of an application) will affect the communication time of the application. This traffic variability will cause the time for communication calls to vary, thus creating variability in overall application runtime. In our observations, network contention has only a minor impact on applications being run on the systems on which we are working, so we leave it as a topic of future work.

Node Allocation - Supercomputers often consist of hundreds or thousands of nodes grouped into physically separate racks. When a job is assigned to nodes located on separate racks on the system, runtime variability is likely to occur. We have not yet explored the significance of allocating a process on a remote node in detail, but we are currently investigating it. Our initial findings show that node allocation has a distinct effect on runtime (discussed further in Section 4.4.3). Scheduler bias makes the probability of being assigned nodes within the same rack is higher than that of being

assigned nodes located in separate racks (unless, of course, the job size exceeds the rack size). Predicting the effect of node allocation requires knowledge about the system layout and the ability to choose the nodes where the training tests will run. System layout information would allow us to match the allocated nodes for the job to their rack locations in the system. Matching the locations of the nodes to the measured execution time would give a clear idea of how node allocation affects runtime. Also, if given the ability to choose the allocated nodes, we could run experiments on nodes located on various racks in the system in order to estimate the effect of remote node allocation without having to rely on chance (which would require far more tests). Since machine layout information and node selection privileges are not available on our machines, we leave this study to future work.

4.2 Prediction Methodologies

As previously mentioned, the goal of this chapter is to predict a range of runtimes for applications running for many iterations, using times obtained from many runs of a single iteration of the program. In order to predict execution time, we must first have an understanding of the amount of runtime variability (in terms of mean and standard deviation) that exists on a single iteration of the application. Once we understand the variability for a single iteration, we can then use the information to predict the runtime at larger iteration counts.

4.2.1 Understanding Single Iteration Variability

In this section, we discuss two different approaches to this problem. The first uses

micro-benchmarks to capture the noise signature of the machine and then injects noise based on this signature into a single-iteration simulation of the application. The noise signature is a collection of all operating system noise events on the system and their duration over a fixed time interval. Since other types of noise are not repeatable and therefore impossible to capture using micro-benchmarks, this approach focuses on operating system noise. The second approach captures the runtime variability of a single iteration by running one iteration of the application repeatedly on the system. This approach is able to capture other sources of noise on the system, in addition to operating system activity. We discuss the benefits and drawbacks of each approach in the following sections.

4.2.2 Making Predictions

Under each model, once we can model the variability of a single iteration of the application, in terms of its mean and standard deviation, we can use the central limit theorem to make predictions about the behavior of the application when run for multiple iterations. If a random variable has a mean (μ) and standard deviation (σ), the approximate range of the sum of n independent replications from this distribution will, for large n, be:

$$((n * \mu) - (3.30 * \sqrt{n} * \sigma)) \text{ to } ((n*\mu) + (3.30 * \sqrt{n} * \sigma)) \qquad (4.1)$$

with 99.9% confidence. There are other confidence intervals that can be calculated using this theorem. However, we focus on the 99.9% interval in this chapter, since we desire an interval that is almost sure to contain any realizable result. All predictions in the following sections of this chapter are made using this formula.

We note several caveats concerning use of this formula:

a) The number of iterations, *n*, must be known in advance.

b) The procedure used to estimate the mean (μ) and SD (σ) must be unbiased. The estimates of μ and σ based on many runs of a single iteration must be representative of the many iteration experiments.

c) The formula given above depends crucially on the assumption that runtimes for successive iterations are independent of one another. If something in the application process causes subsequent experiments to be similar to one another with respect to run-time, then there will be positive serial auto-correlation, and the formula given above will under-estimate (perhaps significantly) the true variability that will occur.

4.3 Prediction By Simulation

In this section, we discuss our technique that simulates a single iteration of an application and then show our results when predicting the performance of 100 iterations on a real system. Simulating a single iteration of a scientific application requires one to understand the noise signature on the machine along with the application's runtime for a single iteration. As previously mentioned, this technique focuses on operating system noise on the supercomputer. The application studied in this section is a program that mimics a wavefront application with an *a* x *b* processor grid layout (often denoted Wavefront(*a*,*b*) in this section). In a wavefront application, each processor receives from its northern and western neighbors (if neighbors exist), computes for approximately one second (in a noisy environment), and then sends to its southern and eastern neighbors

(Figure 4.2). The wavefront communication begins at the top-left corner of the *a* x *b* processor grid. We perform these tests on 32 processors (using an 4x8 processor grid, Wavefront(4,8)) on the Atlas cluster. A process cannot begin execution until it has received from both of its neighbors, if both exist. One pass through the wavefront is considered to be one iteration,and there is an MPI_Barrier call at the end of each iteration. Our wavefront application represents an application with long computation phases followed by short communication phases.

The diagram in Figure 4.2 displays an example Wavefront(4,8) iteration with noise injected based on the Atlas noise signature (described in Section 4.3.1 and seen in Figure 4.3). The per-processor times are shown, with the critical path (among all 120 possible south-east paths going from top-left processor to the bottom-right processor) highlighted in red. In the absence of all noise, the time on each processor would take exactly 1.000 seconds and the total execution time would be 11.000 seconds for every path. However, because of random noise events occurring on each processor, the computation time of each processor is greater than 1.000 seconds and not all paths take the same amount of time. The critical, or longest, path [57] is the one that determines the true execution time, 11.144 seconds in the example shown. Under our method, we perform many simulations of a single Wavefront(4,8) iteration like the one shown in Figure 4.2. The single iteration simulations are then used to estimate the mean and standard deviation of the one-iteration critical path. Then, we use those values in the formula of Section 4.2 to obtain prediction intervals for the time needed for *n* iterations, where *n* is a larger number of interest, such as 100. In the following sections, we further explain how we generate the per-processor times and run our single-iteration simulations.

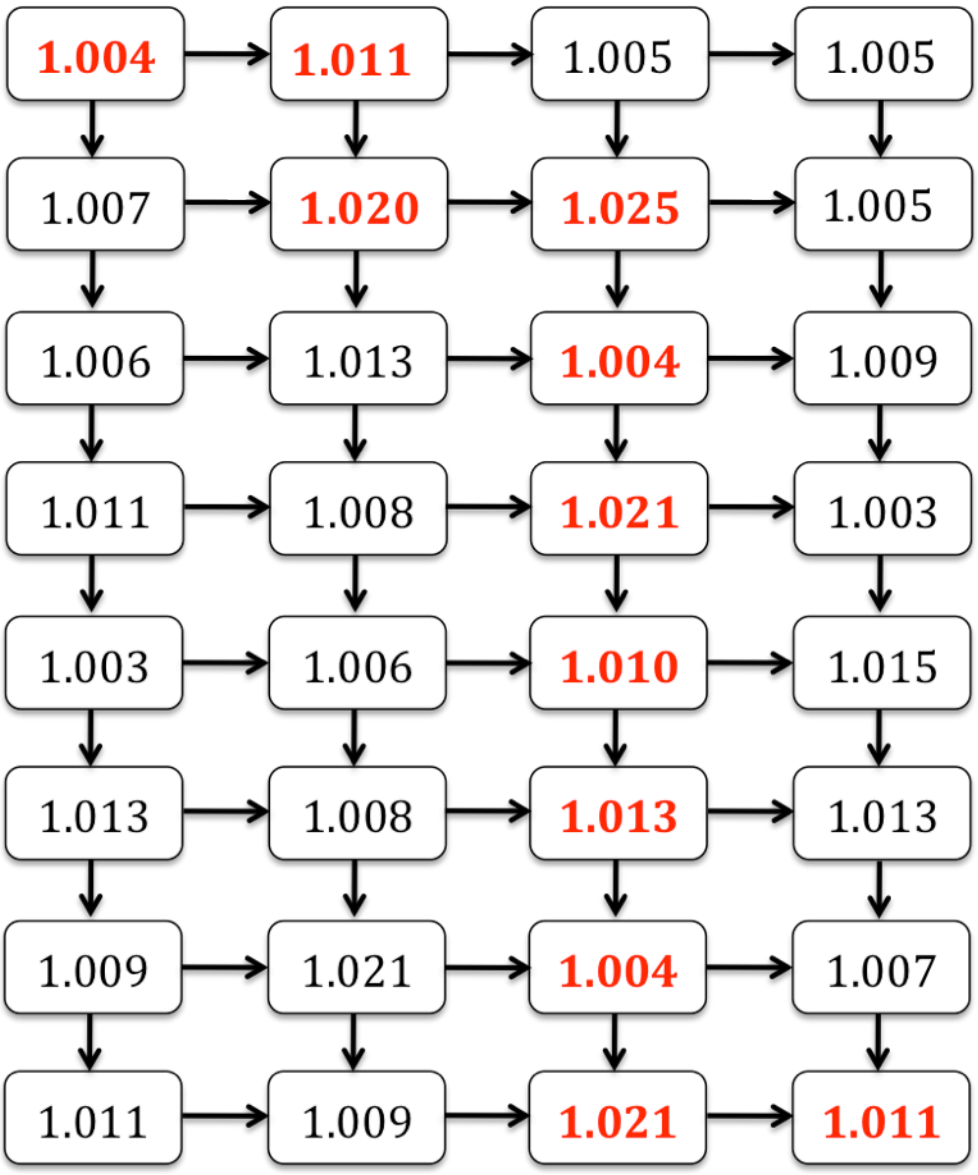

Figure 4.2: Example Wavefront(4,8) iteration with noise injected based on the Atlas noise signature. Each block represents a processor in the 4x8 configuration. Times represent the time to complete the computation on a given processor (in seconds). The critical path is labeled in red.

4.3.1 Measuring Operating System Noise With Micro-benchmarks

A common approach to measuring noise on a machine is to use micro-benchmarks. These are specifically written algorithms to capture operating system

activity on the processors [8, 27, 28, 61], and we briefly discuss these in Chapter 5. For this work, we use the selfish benchmark, which is publicly available as part of the Netgauge network performance measurement framework [28]. The selfish benchmark runs a carefully calibrated spin loop and records any deviations from the minimum loop time (found empirically). Figure 4.3 shows the noise events recorded during a 250 second period on the Atlas supercomputer at LLNL (described in Chapter 2).

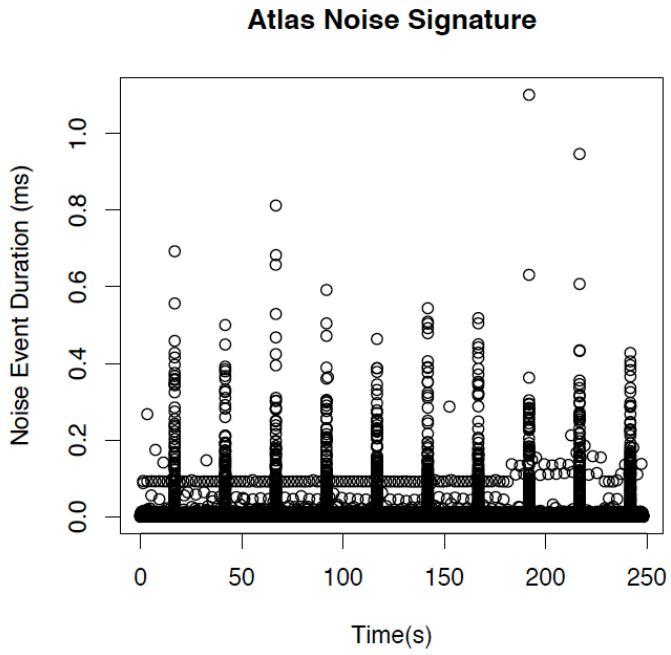

Figure 4.3: Noise events recorded on Atlas during a 250 second period.

4.3.2 Summarizing the Noise Signature

Once the noise signature is obtained, we group the noise events into distinct types. We look for periodic repeating noises first and group them based on similar frequency and duration. For example, the first type of noise classified from the Atlas signature occurs once every millisecond and, when it occurs, is approximately normally distributed

with a mean of 615ns and a standard deviation of 7ns. Occasionally, we see a probability distribution within a noise type. For example, type 1 noise on Atlas has a 62% chance of N(615,7), a 14% chance of N(715,17), and a 24% chance of N(815,45), when it occurs, where N(μ, σ) represents a Normal Distribution with a mean of μ and standard deviation of σ. This means that when a type 1 noise event occurs, it has a chance of following one of the three possible distributions in column 3 of Table 4.1. A summary of the high-frequency noise summary on Atlas based on a 250-second sample is given in Table 4.1.

Noise	Period (1/freq) in ns	Noise length (ns)	%
1	1,000,000	N(615; 7)	62%
		N(715; 17)	14%
		N(815; 45)	24%
2	1,000,000	N(3,400; 23)	83%
		N(3,500; 33)	17%
3	250,000,000	N(9,200; 260)	100%
4	550,000,000	N(12,300; 1060)	100%

Table 4.1: A summary of the high-frequency noise based on a 250-second sample from the Atlas cluster.

The four noise types listed in Table 4.1 are considered high-frequency, low-duration noise. The effect of these four types of noise on the wavefront application is expected to be low due to the relatively large amount of computation time. High-frequency, low-duration noise has a much larger affect on applications with fine-grained communication calls [49], i.e. those where computation times are short relative to their communication times. However, for most of the applications considered in this

dissertation, and especially for the wavefront application considered in this chapter, computation dominates communication. Thus, in such cases, the four types of noise noted in Table 4.1 will not be a large source of variation. In addition to these four types of noise, we also observed high-duration, low frequency noise events, which did not follow a normal distribution and were not periodic. For these noise types, we found an average frequency and an approximate distribution. In one case, when the noise event occurred (if at all), it occurred many times in succession, causing roughly a 20ms delay. High-duration, low-frequency noise can have a dramatic impact on application runtime and is usually hard to capture in sample runs based on their relatively low probability of occurrence.

4.3.3 Application Simulation

One advantage of our micro-benchmark simulation approach is that the single iteration execution range is gathered using inexpensive simulation instead of having to use valuable supercomputer resources. Given the noise-free runtime for all processors of a single iteration of an application, we generate random noise according to the different distributions (obtained from the noise signature distribution) and accumulate them into the overall noise impact. For example, suppose we want to simulate a wavefront application with 1.0 seconds of noise-free execution time per processor. First, we generate the expected number of events of each type of noise that occur during the one second of computation. For each noise event that occurs, the duration is randomly generated from the distribution for its noise type. The durations of each event for a specific noise type are then summed to get the total noise for that type. For the short,

periodic noise types, this value is essentially deterministic for this type of computation-bound application. The more interesting cases are the high-duration events that occur less frequently, such as a noise event that occurs 15% of the time over a one-second interval. This means that 85% of the time there will be no noise of this type, but when it occurs, it will delay the application longer relative to the other noise.

4.3.4 Results

In this section, we present results from our simulations, which inject noise according to the Atlas noise signature into a simulation of the wavefront program using a 4x8 processor configuration. These random noise events are accumulated to generate computation times for each of the *a* x *b* nodes in the Wavefront(*a*,*b*) program. The values shown in the diagram in Figure 4.2 represent a particular set of random noise events that could have occurred on the 32 processors in Wavefront(4,8) if each node's noiseless computation time were 1.000 seconds and noise events were generated according to mixtures of distributions that approximate the Atlas noise signature shown in Figure 4.3 . This entire process was simulated many times (10,000) to obtain 10,000 simulations of one iteration of Wavefront(4,8). The mean and SD of these 10,000 values could then be used in the formula of Section 4.2 to make predictions about the expected length of time for n=100 iterations. We then predict the estimated runtime using the formula discussed in Section 4.2.

Our simulation results show that the 99%-ile estimates under the Atlas noise signature are less than 11.155 seconds. However, when 100 actual single-iteration Wavefront(4,8) tests were run on Atlas, we observed that the actual execution times

range from 11.227 to 11.798 seconds (Figure 4.4). All tests on the real machine are greater than the 99%-ile estimate, which strongly indicates that the machine has other sources of noise (in addition to operating system noise) that cause the application to slow down.

As mentioned in Section 4.1, complex systems have various sources of noise including non-uniform memory access (NUMA) effects, operating system daemons, network contention, and the layout of the nodes assigned for a given run. Clearly, other sources of noise affect our predictions, so we must try another approach to capture additional sources of noise in the system.

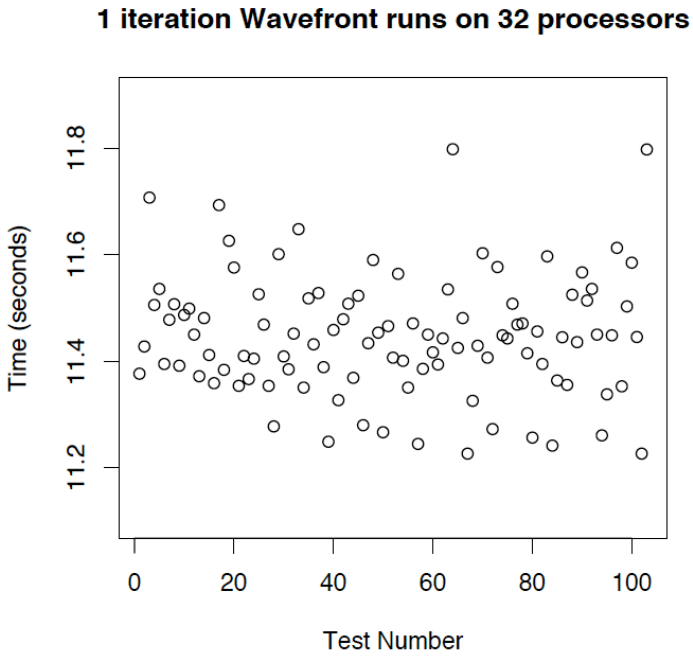

Figure 4.4: 1 iteration runs of Wavefront(4,8) on 32 processors.

4.4 Application Measurement

The micro-benchmark simulation approach has the advantage of avoiding supercomputer usage. However, it will work only if the noise signature obtained from the machine is representative of the true noise on the system. Since there is clearly other noise is present on the system, we must find another approach.

Our second approach involves running a single iteration of the application many times on a real machine in order to understand the variability due to noise for a single iteration. We then use this information (via the formula of Section 4.2) to predict the runtime of multiple iterations. This technique allows us to capture the architectural variability such as NUMA noise and cache effects, in addition to operating system variability. We cannot accurately capture network contention and node allocation variability since these sources of variance are not easily captured (as mentioned in Section 4.1). However, these sources of noise likely still impact our tests.

4.4.1 Experimental Methodology

To investigate this method, we run two different synthetic applications. The first is the wavefront application discussed in section 4.3. We run the synthetic wavefront on 32 processors (Wavefront(4,8)) and 256 processors (Wavefront(16,16)) in this section. The second program is a small segment of the Parallel Ocean Program (POP). Many research studies point out that the large number of MPI_Allreduce calls in POP cause it to be highly affected by noise events. Many researchers claim that the reason for POP's noise sensitivity is that it contains mostly fine-grained computation phases followed by

global synchronization calls (typically MPI_Allreduce). Since POP has roughly 75K lines of code, we decided to start by analyzing a segment of the program that represents its typical behavior. Specifically, our synthetic program computes for a few hundred microseconds; executes an MPI_Bcast and an MPI_Barrier; and then does 4 separate computation phases of a few hundred microseconds with MPI_Allreduce calls made in between each computation phase. The end of the iteration is marked with an MPI_Barrier. The POP segment is executed on 32 processors. All experiments are run on the Atlas cluster at LLNL.

4.4.2 Results

The first program that we examine is the synthetic wavefront application. First, we ran 5000 runs of a single iteration of both Wavefront(4,8) and Wavefront(16,16). From the former, we estimated 11.034 seconds for the mean (μ) and 0.009 for the standard deviation (σ) of the critical path. For the latter, the estimates were 32.138 for the mean (μ) and 0.072 for the standard deviation (σ).

We then ran the wavefront application on 32 and 256 processors, respectively, for 100 successive iterations. Figure 4.5 shows 100 replications of the 100-iteration synthetic Wavefront(4,8) application on 32 processors. Each point on the graph represents a single run of the application. Using the equation 4.2 and the results from the 5000 1-iterations, we can obtain an expected time of 1103.350 seconds (dotted line) along with a 99.9% upper and lower prediction bounds (solid horizontal lines) at 1102.750 and 1103.930. Of the 100 tests run, 96% of them fall within the predicted range with four tests falling outside of the range being within 0.005% of the upper bound of the range. So, in this

case, the procedure worked reasonably well, although the true variability is slightly more than predicted (points lie outside of the predicted range) and the mean is slightly higher than predicted.

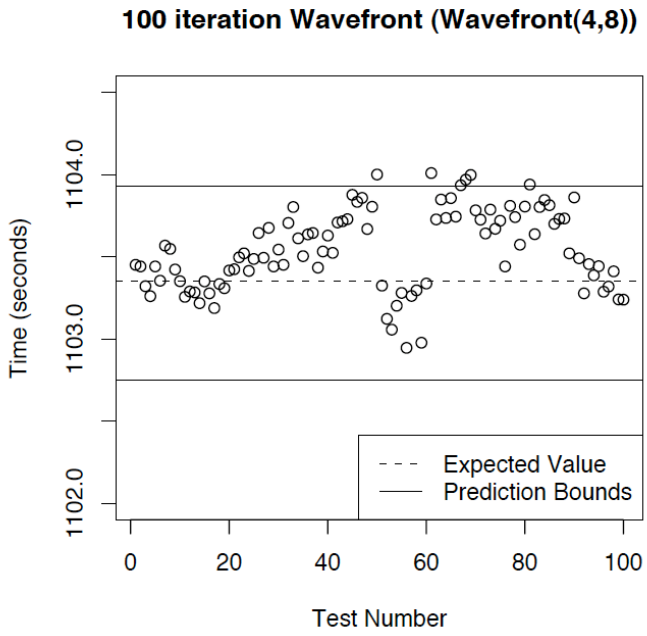

Figure 4.5: 100 replications of the 100-iteration wavefront application (Wavefront(4,8)). Dotted and solid horizontal lines represent expected value and 99.9% upper and lower prediction bounds, respectively, based on single-iteration sampling.

Figure 4.6 shows the predictions for the synthetic wavefront application (Wavefront(16,16)). Our technique predicts that the 100-iteration tests should fall between 3111.404 seconds and 3116.153 seconds with an expected value of 3113.780. The figure shows that 23 of the 100 tests lie outside of the prediction range. However, the points lie within 0.010% of the prediction lines in the worst case. The variation for Wavefront(16,16) is more extreme than the Wavefront(4,8) case. We believe the increased variation is due to the fact that Wavefront(16,16) runs for a longer amount on an increased number of processors, which leaves it vulnerable to additional sources of noise.

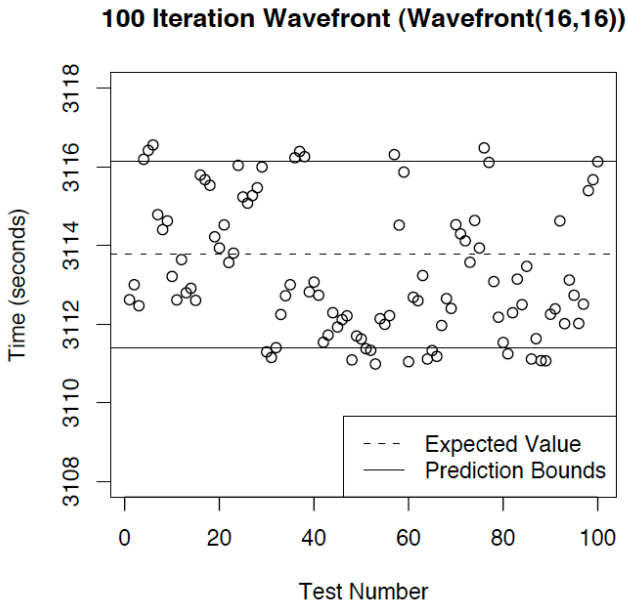

Figure 4.6: 100 replications of the 100-iteration wavefront (Wavefront(16,16)).

The second application that we examine is the synthetic POP segment mentioned in the previous section. This program's total execution time is comprised of a much higher percentage of communication than the synthetic wavefront application. We make predictions for this application at increasing iteration counts. Using a single set of 5000 single-iteration POP tests and the formula described in Section 4.2, we predict POP execution time at both $n = 100$ and $n = 1000$. Figure 4.7 shows a comparison of our predictions to the actual results of $n = 100$ iterations of the POP segment on 32 processors. In this graph, the tests fall within the prediction range 99% of the time with only one test lying outside of the prediction range. However, the mean estimate is biased, as shown by comparison with the dotted line in the figure. Figure 4.8 shows predictions for $n = 1000$ iterations of the POP segment on 32 processors. The tests in this graph fall within the prediction range only 62% of the time with a worst-case error of 0.8%. However, there is again a bias in estimating the mean.

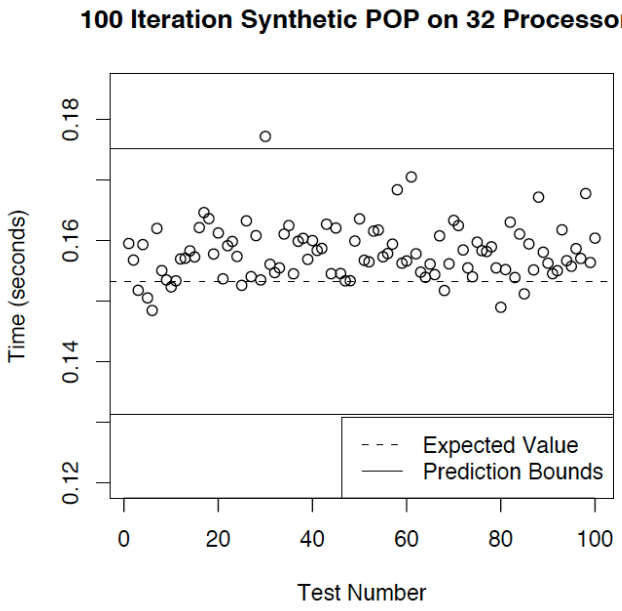

Figure 4.7: 100 replications of the 100-iteration POP segment on 32 processors.

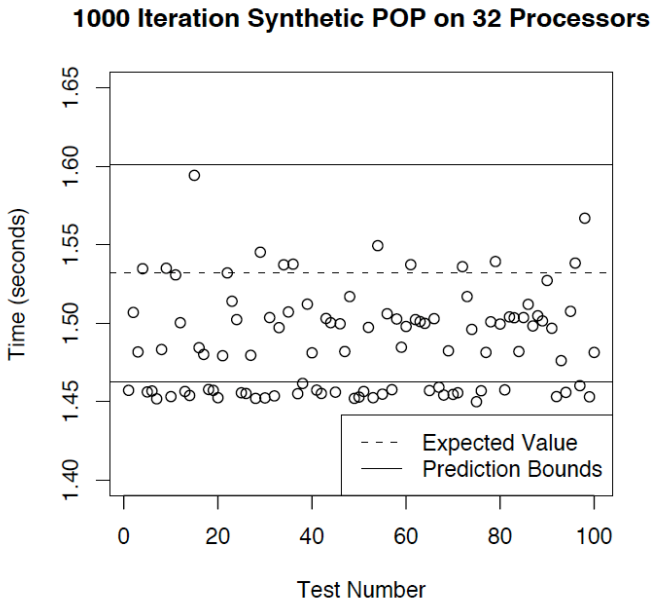

Figure 4.8: 100 replications of the 1000-iteration POP segment on 32 processors.

4.4.3 Discussion

We observe two reasons why our technique does not predict the range more accurately in some cases. The first problem is that the estimation of the mean is slightly incorrect. This error probably arises from additional sources of noise that our technique fails to capture, indicating the need for more single-iteration tests. Second, sequential observations appear to be dependent. They come from blocks with similar means. We believe that much of this variation is due to node allocation. Figure 4.9 shows over 900 single iteration runs of the Wavefront(32,32) application on 1024 processors (256 nodes). Each group of tests (roughly 90) is allocated a different block of nodes (randomly) on the supercomputer. The vertical lines in the figure show the points at which node allocations changed. The effect of node allocation is clearly seen in this figure. Similar block variations are seen in other runs that we have considered in order to find the distribution of the critical path times for one iteration of various applications, but they are more obvious when the runtime for one iteration is large, as it is here.

The two problems noted in the previous paragraph are both potentially serious. The first is more so, since, as noted in caveat (b) of Section 4.2.2, if the mean can't be estimated accurately, there is a high potential for error in the range predictions. We believe that some of the mean estimation errors are due to additional (unmeasured) sources of noise on the system. The second problem (underestimation of spread) can potentially be handled by gaining a better understanding of the block nature of the node allocation process.

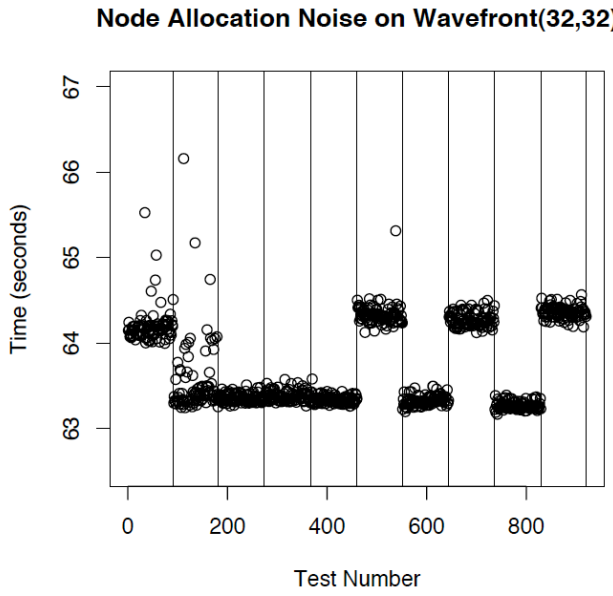

Figure 4.9: Single iteration wavefront runs (Wavefront(32,32)). The effect of different node allocations is clearly seen. Vertical lines show points at which node allocations change.

4.5 Summary

Clearly, more sources of noise are present in the system when applications run on more processors and run for more iterations. We believe that running for thousands of iterations may cause other sources of noise to show up that are yet to be seen in 1-iteration experiments. Also, running applications on more processors may make the experiments prone to larger node allocation and network contention variability (two factors that we did not consider in this work). In this chapter, we have presented a technique that can accurately predict the range of runtimes for two synthetic applications. This promising start for runtime prediction in the presence of noise lays the groundwork for a large-scale extrapolation system.

CHAPTER 5

RELATED WORK

In this chapter, we provide a brief overview of other work in the area of scalability prediction. Section 5.1 discusses alternative prediction techniques, and section 5.2 discusses the effects of system noise on scalability and the difficulties it causes for accurate predictions.

5.1 Scalability Prediction

This dissertation extends a significant body of prior work. Extensive study into methods to predict the performance of parallel applications has explored a variety of approaches. Prior work has frequently focused on cross-platform predictions in which the processor count is held constant but the system under consideration is changed. Other research has used extensive manual analysis to derive analytic models. We extend a significant body of prior work that has developed statistical methodologies to predict performance.

The work most closely related to ours uses various regression approaches to predict application performance across a range of input parameter values when run on the same number of processors. For example, neural networks model the parameter space to predict execution time, generally with errors of 10% or less [32]. Direct comparisons demonstrated that piecewise polynomial regression provides similar accuracy [41].

Unlike these previous regression-based approaches, we identify techniques to separate computation and communication that support extrapolations to larger processor counts.

Another similar approach uses machine learning to make predictions on multicore machines [68]. Also in a similar vein, Curtis-Maury et al. predict the power-performance tradeoff on multicore machines [18]. While similar to our approach, these approaches are limited to single multicore processors and do not address cluster systems.

Similarly, other black-box modeling approaches offer at best limited abilities to extrapolate to larger processor counts. Yang et al. predict performance across platforms through partial execution of iterative programs but only for system sizes used for the partial executions [73]. Lyon et al. use the theoretical approach of Taylor expansions to understand execution behavior, including scalability properties [43]. This more theoretical approach is complementary to our empirical approach and could conceptually be used in tandem with it. Combining static and dynamic analysis to predict performance on different architectures for different inputs offers greater possibilities for extrapolating across processor counts than these other statistical methods [44]. Later work showed that the technique could locate performance bottlenecks [45]). This approach has the advantage of avoiding runs on different hardware architectures, but requires compiler infrastructure that can be difficult to integrate into existing application build environments. In contrast, our framework only requires re-linking the application with the PMPI library to gather data during training runs.

A variety of simulation- or trace-based approaches to performance modeling exist in the literature. MetaSim provides a general performance modeling framework [60]. MetaSim uses Atom [63] or other instrumentation mechanisms like Dyninst [15] to

gather memory traces, which then support simulation of memory performance on a variety of architectures. MetaSim extends those results to a distributed memory setting with Dimemas [37], which consumes an MPI trace to simulate network performance. The memory and MPI traces are tied to the original processor count and, thus, unlike our approach, their work does not directly support scaling predictions. Although techniques could extrapolate those traces to larger numbers of processors, our system provides a more direct approach to scaling predictions.

White-box approaches typically require detailed analysis of data structures and program constructs, such as loop nests. Kerbyson et al. derive an analytical performance model for the ASCI Sage application [36]. They combine detailed analysis of program constructs with microbenchmarks that measure basic machine characteristics such as network, memory and computation capability. This powerful approach does support scaling predictions. However, it requires significant performance analysis effort that would be difficult, if not impossible, to automate. Our system requires, at most, a small amount of application-level information. When predicting the input parameters for time-constrained scaling, the users must provide our system with information about parameter relatedness and whether or not a processor grid is used (Chapter 3). Obtaining this information requires little analytic effort. Several other researchers have explored white-box scalability analysis approaches that provide algorithmic or architectural perspectives [16, 26, 51, 59, 71, 72]. In general, they derive application or architecture specific models through detailed analysis, which requires significant effort that is not readily automated. In a strongly related white-box approach, Brehm et al. make predictions using regression and explore separating computation and communication [12]. However, their approach

requires detailed analysis to create the computation and communication models. Other white-box approaches that predict workload and memory requirements, such as *modeling assertions* [3], require code modifications in order to predict workload and memory requirements. Our system uses the MPI profiling interface for instrumentation, which at most requires relinking the application.

Many have investigated analytic modeling of parallel machines. The best known examples are LogP [17] and BSP [66]. The former uses latency, overhead, gap, and number of processors to determine an effective parallel algorithm while the latter uses *supersteps* to indicate computational phases, which are terminated by communication points. Both of these techniques require significant programmer intervention. For example, with LogP, while the programmer can model a computational step as taking constant time, it is still necessary to model the communication that exists precisely. As the number of processors increases, modeling communication becomes increasingly challenging. Another approach requires no user intervention to create a static cost model [6]. However, it has so far only been used effectively for simple programs and on simple architectures.

Several tools trace or analyze MPI performance through the MPI profiling interface, including mpiP [67], Open|SpeedShop [58], VampirTrace [48], svPablo [22], TAU/ParaProf [10], and Paraver [53], just to name a few. These tools generally focus on providing assistance in optimizing applications, particularly for very large processor counts [56]. We build on algorithms to capture the critical path in MPI programs that were developed to support optimization [30, 57].

5.2 Effects of System Noise

When an application is affected by seemingly random operating system noise (jitter), it becomes extremely difficult to predict its scalability. Doing so requires in-depth knowledge about the types of noise on the system and how the noise affects the application at large-scale. The tests performed in Chapters 2 and 3 of this dissertation were run under settings that result in negligible amounts of system noise. Important strides have been made towards understanding and modeling noise, but a general scalability prediction system in the face of noise is yet to be realized. Our work presented in Chapter 4 demonstrates a necessary first step towards this goal.

Over many years, researchers have conducted numerous studies on the effects of operating system noise on large-scale parallel computer systems. Noise has been widely accepted as a major problem in parallel computing. In 2006, noise was declared one of the leading issues facing petascale computing [7]. Over the years, researchers have focused on different aspects of the problem including: finding the exact source of noise, attempting to reduce the noise effect, understanding application characteristics that amplify noise, and understanding the degree to which noise slows down large, parallel applications.

5.2.1 Finding Sources of Noise

Quantifying the amount of noise on a system and determining the cause is an important area of research. One approach to quantifying the amount of noise on a system is to use micro-benchmarks that are able to measure and to record individual noise events

on a machine [28, 61]. These studies attempt to find an overall noise signature for a given machine. They are not concerned with which activity in the kernel caused each event. These micro-benchmarks find the minimum time to execute some amount of work (or minimum work to execute in a given amount of time) and then repeatedly record any deviations from the minimum and deem these variations to be caused by the operating system. These micro-benchmarks provide a quick and accurate way to sample operating system noise. Most research on operating system noise uses some variation of this technique. As the researchers point out, determining precisely how long a given amount of work should take and determining the minimum iteration time for a given benchmark loop are sources of potential error. Also, noise events that are smaller than the combined overhead from the loop and the timer calls will not be recorded. However, some researchers state that such tiny noise events are really of only theoretical value and that they do not affect the performance of the application [27]. In Chapter 4, we use the selfish benchmark [28] to measure operating system noise and then inject the recorded noise events into our application simulations.

Other work has focused on finding all sources of noise in the operating system kernel and quantifying the total amount of noise from each source [19, 49, 61]. Nataraj et al. provide a technique to provide noise information to the application program using kernel injection [49]. In this work, the authors are able to provide a kernel noise profile showing the amount of time spent in each activity for each node during the run of an application. Gioiosa et al. [25] use micro-benchmarks, kernel profiling and a kernel module to gather results and to find the sources of noise in the operating system. Kernel instrumentation methods produce valuable information about the total noise of each

kernel activity. However, kernel instrumentation is not always an option as it was not available on the supercomputers used for this research.

5.2.2 Reducing the Impact of Noise

Beckman et al. state that operating system interference must be reduced and synchronized in order to create scalable petascale systems [7]. Many researchers have realized this same fact and, therefore, a broad number of techniques to alleviate noise exist in the literature. These include: elevating an application's priority, leaving a processor open to handle system daemons, using a lightweight kernel, synchronizing noise, and reducing the noisiest activities in the operating system.

The first noise study was conducted in 1994 [47]. In this study, the author realized that noise was affecting the application and tried to eliminate it by raising the application priority above operating system activity. Since then, many researchers have set out to find solutions on ways to minimize the impact of noise on scalability. Multiple researchers have shown that leaving a processor available to handle noise events can greatly reduce the impact of noise [33, 34, 52]. Petrini showed that MPI_Allreduce takes 300us on 3 processes per node (~900 nodes) and 3ms on 4 processors per node (a factor of 10 degradation). While this is an important find, giving up a processor to handle noise is not a viable solution since it limits the scalability of the system. Also, this technique only reduces daemon activity, not periodic timer interrupts.

Another way to reduce the impact of operating system noise is to use a specialized, lightweight kernel. Lightweight kernels attempt to remove noise events by simplifying the operating system [35, 46]. For example, Blue Gene/L [39] does not have any daemon

processes or a process scheduler and, therefore, has almost no system noise. These customized, lightweight kernels are only deployed on compute nodes. They usually have a limited amount of operating system functionality and do not support multitasking or interrupts. Also, they require that applications be ported to run on the specialized system.

Other researchers have suggested enhancements to commodity operating systems. The most common suggestion is to synchronize the noise on the machine so that the noise penalty is paid at the same time by all processors [8, 9, 20, 21, 27, 33, 34, 52]. When noise events are unsynchronized, even a small amount of noise on each processor can lead to severe performance degradation (synchronization reduces this amplification effect). Jones et al. were some of the first researchers to examine the performance gained by coscheduling noise [33, 34]. In this research, the authors show that coscheduling noise leads to a 300% speedup in a benchmark performing 4096 allreduce calls of 8 bytes each. Beckman et al. show that the impact of unsynchronized noise on MPI_Allreduce performance adds a constant 35us (100% increase at 1024 processors and 50% increase at 32K processors) when injecting 1.6% noise on a noiseless machine, whereas synchronized noise has almost no effect. De et al. show that synchronizing noise can reduce the noise impact from 99% slowdown to 6% when injecting real machine noise traces into a small MPI_Barrier micro-benchmark on BG/L. Hoefler et al. show that unsynchronized noise causes 100% slowdown for POP whereas co-scheduling POP results in less than 0.5% slowdown [27]. Hoefler's results are shown using simulation on 16K processors with operating system noise injected from a noise trace of a real machine.

Some researchers believe that by determining the dominant sources of noise (in terms of total amount over the application runtime) and removing them from the system,

scalability will be improved [19, 49]. However, it has been shown that the largest sources of noise do not always cause the biggest problem. Evidence has shown that the frequency and duration of the noise (system's noise signature) has a larger effect on scalability than a single noise source [24, 27]. We noticed the same effect when looking at operating system noise injection in Chapter 4. Tsafrir et al. promote the idea of removing fine-grained operating system clock ticks and replacing them with "smart timers" [65]. The authors show that the impact of these fine-grained noise events is linearly proportional to the number of processors used if the probability of noise is small, which means that adding more nodes increases the probability of a noise event on a collective operation. When the probability is large, adding processors (beyond the convergence point) will not affect the runtime since a noise event is a virtual certainty. One issue that is not discussed is the possibility of other fine-grained noise events. As we discuss in Chapter 4, modern systems contain many sources of variability other than operating system noise. In order to realize a large-scale prediction system, all sources of noise must be considered.

While it is important to reduce the effects of noise in large-scale systems, it is impossible to eliminate all noise on complex supercomputers. For this reason, we must consider the impact of noise on scalability in order to make accurate predictions. Our work focuses on understanding the impact of noise at large scale instead of ways to reduce the noise in the system.

5.2.3 Understanding the Impact of Noise

A broad array of work on understanding the impact of noise at large-scale is available in the literature. In order to quantify this impact, researchers have conducted

theoretical studies, tests using micro-benchmarks, full application runs with noise injection, and large-scale application simulators.

Agarwal et al. took a theoretical approach to understanding noise [2]. In this work, the authors make the first attempt to explain the impact of noise with a mathematical model. They study bulk-synchronous applications under three types of noise distributions: exponential, heavy-tailed, and Bernoulli. Performance is shown to be best when noise follows an exponential distribution. When Bernoulli noise is introduced, the effect of noise scales linearly with the number of processors. At large processor counts, the impact converges to the amount of noise injected. The convergence is due to the fact that noise occurrence is a virtual certainty at large processor counts. That noise converges at large scale has been verified in other work [9, 27, 65]. Tsafrir et al. demonstrate a probabilistic model of fine-grained noise (clock ticks) [65]. They claim that when the probability of noise is small, the impact of noise grows linearly with the number of processors. When either the probability of noise or the number of processors becomes large, they also show noise convergence. Hoefler et al. analyzed the effects of noise on point-to-point messages using the LogGOPS model [27, 29, 31]. They show that blocking point-to-point messages often propagate noise. The authors state that non-blocking point-to-point communication has a higher potential to absorb noise. They then show the noise bottleneck (convergence point) and where it occurs on different machines with different applications. Through the use of simulation and noise traces gathered on individual machines, the authors are able to show that each system has a precise point at which the noise converges (for collective operations). They state that this knowledge could be used for tuning new systems so that the noise bottleneck occurs after the maximum system

size. Understanding the noise convergence point may allow our system to further reduce the number of training runs required. If we can show the point where noise converges, we could eliminate the need to run tests at processor counts beyond the convergence point since noise would no longer affect the application execution time.

Tests on micro-benchmarks have provided valuable information about the effects of noise at large scales. Since most HPC applications are bulk synchronous, micro-benchmarks often consist of collective communication calls with little to no computation in between. Beckman et al. have conducted two studies on the impact of noise on collective communication calls at large scale using artificial noise injection [8, 9]. In these studies, the authors look at MPI_Allreduce, MPI_Barrier, and MPI_Alltoall on 1024 through 32K processors on BG/L (a good platform for noise injection since it is inherently noiseless). The authors inject noise at different frequencies (1ms, 10ms, 100ms) and different durations (16us, 50us, 100us, 200us) resulting in a maximum noise of 20% (200us every 1ms). Although noise signatures on modern machines are much lower (usually less than 1.0%), this work demonstrates the noise effect at extreme amounts. The results of these tests show that high duration, low frequency noise is the most detrimental. Also, when injecting the same percentage of noise, large noise events affect MPI_Allreduce time more than smaller ones. This information leads the authors to claim that high-duration, low-frequency (coarse-grained) noise also affects fine-grained applications, and they refute Petrini's claim that performance loss is substantial when an application resonates with system noise (i.e. fine-grained applications are most affected by low duration, high frequency noise and vice versa) [52]. Results also show that MPI_Alltoall is most affected (up to 173% degradation) followed by MPI_Allreduce.

Under a constant percentage of noise, the authors show that large detours affect MPI_Allreduce more than smaller detours. Like the earlier theoretical work, Beckman et al. also show that the noise effect levels off at large node counts. They claim that increasing the node count any higher than the convergence point will not cause more noise effects. The leveling off occurs at lower processor counts with a larger noise percentage injected.

In a later work, De et al. take a similar approach of artificial noise injection on BG/L to understand the scaling of collectives [20]. One of the main differences between this work and Beckman's work is that the authors collect noise signatures from real machines and replay them on a noiseless machine in order to measure scalability. They use a simple micro-benchmark, which does repeated iterations of computation followed by MPI_Barrier. Noise is injected on BG/L because of its low noise signature and high processor count. Their experiments are run on up to 2048 processors. Their "Jitter Emulator" is an extra function in the code, which the process calls to inject noise according to the noise signature. Noise events are scheduled using alarms. Another difference from Beckman is that they increase the duration of all noise events at the beginning of the run and then scale back down at the end to ensure that they can inject small amounts of noise. Without scaling the values, the authors would be unable to inject small noise events due to the overhead of the alarms, function calls, etc. Results show that unsynchronized jitter leads to 99% slowdown in the worst case (2048 nodes).

Other authors have provided techniques for using simulation to understand noise effects on collective operations. Sottile et al. wrote a discrete event simulator to analyze the effects of noise [62]. In this work, they simulate variations in message latency and

compute time (noise) and observe the effect on runtime. They create a message-passing graph by analyzing traces and then parameterize the simulation using noise and latency variations (found using micro-benchmarks). The results focus on a simple, token ring code. The authors show that on 128 nodes that the runtime increases by the product of the number of ring iterations, how often noise is injected in cycles, and the number of nodes.

De et al. also took a simulation approach to understanding noise [21]. They claim that simulation is necessary in order to explore the effects of noise beyond the limits of currently existing systems since other approaches are limited by the largest jitter free system available. In this work, simulations are based on noise traces (from real machines), network latency measurements, MPI stack latency measurements, and shared memory latency measurements. The authors claim that this approach can be used to predict scalability up to any number of processors (results show up to 16K processors). Each task starts at random points in the jitter trace in order to achieve unsynchronized noise. The authors simulated a bulk-synchronous micro-benchmark with computation phases followed by MPI_Barrier synchronization. The results show a 45% slowdown in runtime at 16K processors.

The next logical step, after understanding noise impacts on micro-benchmarks, is to move to real parallel applications. Ferreira et al. look at the noise impact on three different parallel applications (SAGE, CTH, and POP). To study the effects of noise, the authors create a kernel level noise injection framework. 2.5% net processor noise is injected into the applications at varying frequencies and durations on an otherwise noiseless machine (Catamount). The results show that changes in the frequency and duration greatly affect the performance of the applications even when keeping the total

amount of noise the same (this finding is later verified via simulation by Hoefler et al. [27]). Similar to the findings of Beckman et al. [8, 9], the authors show that high-frequency, low-duration signatures do not affect the applications as much as low-frequency, high-duration signatures. Out of the three applications that are studied, POP is the most affected by noise (nearly 1900% at 2500 nodes) whereas CTH is virtually unaffected. The authors explain these findings by pointing out that certain features of the application lead to noise absorption or amplification (also verified by Hoefler et al. [27]). They show that POP spends most of its time in collective operations at high node counts. Also, SAGE spends more time in MPI_Allreduce making it more susceptible to noise than CTH. They also show that MPI_Allreduce is more affected by noise than MPI_Bcast (later verified by Hoefler et al. [27]). MPI_Allreduce is especially sensitive at small sizes (in bytes). Their results show that applications tend to absorb high-frequency, low-duration noise and amplify low-frequency, high-duration noise. A coarse-grained application, like CTH, is able to absorb frequent, short noise but unable to absorb high-duration noise (consistent with Beckman et al. [8, 9]). One possible improvement to this work would be to test noise signatures from real systems. The authors say 2.5% processor noise is common, although it seems to be slightly higher than the amount of noise found in other noise studies [8, 9, 27].

Hoefler et al. took a simulation approach to understanding noise effects on large-scale parallel applications [27, 29]. Similar to De [21], they use discrete-event simulation and noise traces from real machines (gathered using the selfish benchmark). However, their simulations are based on the LogGOPS model [31] and they study noise impacts on full applications. Simulated applications include Sweep3D, POP, and AMG. Out of the

three applications that were studied, POP is by far the most affected by system noise (similar to the findings of Ferreira et al. [24]). AMG is only slightly affected, while Sweep3D has nearly no impact. In agreement with Ferreira [24], Hoefler says that in order to understand the impact of noise, you must consider the communication patterns of each application. Sweep3D spent only 7.6% of time in collectives and has mostly blocking point-to-point operations. On 16K processors, this application shows less than .7% slowdown on a noisy machine. AMG spends 9.72% of runtime in collective communication with 45% of communication coming from non-blocking point-to-point messages. The worst slowdown on 16K processors is still less than 5%. POP, on the other hand, spends 77.2% of time in communication. 77% of this (overall time) is in collective calls. Because of its large amount of collective communication, POP slows down by 100% on the noisiest machine studied (CHiC), which has 0.26% operating system noise. Ferreira's study went up to 2.5% operating system noise, which may explain why POP slowed down by 1900%. Based on the results from this paper, it is clear that the influence of noise depends on the noise signature of the machine. The noisiest machine does not always have the largest noise impact. Random, long detours can be more detrimental than short, periodic ones (as seen in other studies). One possible improvement would be verification of their simulations. They simulated full runs on 16K processors and it seems like 4K-8K should be obtainable on real machines for verification.

Hoefler's paper is one of the few to point out that collective and point-to-point calls both influence the application's sensitivity to noise. They demonstrate results from AMG and POP, which show that P2P messages can help absorb noise or even amplify the noise

depending on the situation/application.

Most of the work described in this chapter attempts to identify and to reduce the impact of noise on large-scale systems. In our work, instead of reducing the amount of noise, we focus on predicting a range of application execution times in the presence of noise. While there is no current work that does execution time prediction in the presence of noise, studies showing the effects of noise on runtime at large-scale are the most similar.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this chapter we summarize the contributions made in this dissertation and discuss possible future extensions.

6.1 Summary

In this dissertation, we have presented a system that promotes efficient use of supercomputers by providing scalability predictions to the users. Our system guides users in selecting an efficient number of processors and/or selecting the correct input parameters for time-constrained scaling, thereby avoiding wasted processors and job cancellation. Our system can increase supercomputer throughput, improve availability, save power, and reduce application runtime.

Specifically, this dissertation has shown that regression-based techniques are a viable solution for performance prediction. We used regression-based techniques to achieve median prediction errors of 6.2% and 17.3% across seven scientific applications under strong scaling. Also, when predicting input parameters for time-constrained scaling, prediction error was always less than 13% for seven applications. We started with a monolithic total time regression model that works well for simple applications with mostly computation (Chapter 2). We then extended the base system to predict applications with large amounts of communication accurately (Chapter 2). We then

showed how allowing for some application-level information from the user can improve input parameter predictions for time-constrained scaling (Chapter 3). All of our techniques make predictions based on training runs from processor counts that are *strictly less than* the target run.

In summary, this dissertation has made the following novel contributions:

1. A focused regression technique that uses training runs with similar input parameters to the target run. This technique allows for more accurate predictions than non-focused regression techniques for complex applications with many input parameters.
2. A black-box technique to predict parallel program scalability. This technique uses multivariate regression to predict performance on large processor counts using training runs from smaller counts.
3. A technique using separate regressions to predict computation and communication. Our separate regression technique accurately captures the individual scaling behavior of each quantity.
4. A gray-box technique that provides the input parameters leading to accurate time-constrained scaling on large processor counts. Our technique uses minimal amounts of application-level information to provide predictions.
5. A technique for predicting application behavior at large iteration counts using runs on a single iteration. This technique serves as a building block for a large-scale extrapolation system by reducing the amount of system time required for training runs.

6.2 Future Work

While this dissertation has taken important steps towards the realization of a general scalability prediction system that can be run on large-scale machines, many improvements remain to be made. We outline some of those improvements in this section.

1. **Experimenting with larger processor configurations**. Due to the limited size of the systems used in this dissertation, our predictions could only be validated up to 1024 processors. It remains to be seen what happens to prediction accuracy when we reach larger processor counts.
2. **Investigating more applications**. An important extension to this work is to test the prediction accuracy when using large, industrial strength applications with many input parameters that have complicated interactions. While our approach is effective for all applications in our set, we may find that other applications require different techniques to achieve accurate scalability predictions.
3. **Breaking computation and communication into smaller phases**. In particular, different computation or communication phases may scale quite differently. This idea is analogous to dividing total time into computation and communication time—which improved prediction accuracy. This extension requires that we combine phases when their execution time is sufficiently small, to protect against variance that is more striking in small phases.
4. **Determining the smallest number of processors needed for quality predictions**. We need to make sure that we are using the smallest number of

processors for training while still providing accurate predictions. Using fewer processors for training would decrease the time it takes for our system to provide predictions to the users.

5. **Investigating how to reduce the number of experiments.** Using experimental design techniques could allow our system to generate a smaller, representative set of training runs automatically. This approach would allow fewer runs and avoid running tests that only slightly improve prediction accuracy due to their similarity to other tests in the set.
6. **Exploring additional prediction techniques.** In this dissertation, we focus on regression-based techniques for scalability prediction. However, research shows that machine learning techniques such as neural networks are also a reasonable approach to scalability prediction [41]. In the future, we will explore the difference in accuracy between our scalability prediction system and a system based on machine learning techniques.
7. **Exploring the effects of additional noise sources.** We have looked at techniques for capturing noise due to operating system activity and NUMA effects. However, other sources of noise, such as network contention and node allocation must be addressed to create a more accurate prediction system.

We have discussed in detail how our system could be utilized on large-scale machines in order to promote more efficient use of the resources. However, our performance prediction system could be helpful in other areas as well. For example, understanding scalability is also an important topic in cloud computing. In these systems,

users run applications on a commercially available supercomputer where there is a charge per hour for each processor. For example, Amazon has a system that is available for ten cents per hour per processor. The main concern for the users of cloud computing systems is the cost to run a job. Our system allows users to analyze the tradeoff of performance and cost at higher processor counts to decide if increasing the number of processors is cost efficient. Our system could also be used to guide the development of newer, larger systems. Providing system designers with a model of how applications will scale allows them to make more intelligent decisions about system creation and/or expansion. Our system could also be used in system debugging. Observed application performance deviations from the model predictions could indicate a problem (as described by Petrini et al. [52]).

REFERENCES

[1] Accelerated Strategic Computing Initiative. The ASCI Sweep3D Benchmark Code. http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/, Dec. 1995.

[2] S. Agarwal, R. Garg, N.K. Vishnoi, "The Impact of Noise on the Scaling of Collectives: A Theoretical Approach", in *International Conference on High Performance Computing,* Dec. 2005.

[3] S. R. Alam and J. S. Vetter, "Hierarchical Model Validation of Symbolic Performance Models of Scientific Applications," In *The International European Conference on Parallel and Distributed Computing*, Aug. 2006.

[4] S. R. Alam, J. S. Vetter, P. K. Agarwal, and A. Geist, "Performance Characterization of Molecular Dynamics Techniques for Biomolecular Simulations," In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, Mar 2006.

[5] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. RNR-91-002, NASA Ames Research Center, Aug. 1991.

[6] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, "A Static Performance Estimator to Guide Data Partitioning Decisions," In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, Apr. 1991.

[7] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, "Operating System Issues for Petascale Systems," in *Special Interest Groups on Operating Systems*, Jul. 2006.

[8] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, "The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale," in *International Conference on Cluster Computing*, Sep. 2006.

[9] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, A. Nataraj, "Benchmarking the Effects of Operating System Interference on Extreme-Scale Parallel Machines", in *International Conference on Cluster Computing*, Sep. 2007.

[10] R. Bell, A. Malony, and S. Shende, "ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis," In *Proceedings of the International Conference on Parallel and Distributed Computing*, Aug. 2003.

[11] A.W. Berry. Chinese Supercomputer Tianhe-1A, named world's fastest. http://www.helium.com/items/2017071-worlds-fastest-computer/.

[12] J. Brehm, P. H. Worley, and M. Madhukar, "Performance Modeling for SPMD Message-Passing Programs," *Concurrency: Practice and Experience*, Apr. 1998.

[13] P. N. Brown, R. D. Falgout, and J. E. Jones, “Semicoarsening Multigrid on Distributed Memory Machines,” *SIAM Journal on Scientific Computing*, vol. 21, no. 5, pp. 1823–1834, 2000.

[14] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, "A Scalable Cross-Platform Infrastructure for Application Performance Tuning using Hardware Counters," In *Supercomputing*, Nov. 2000.

[15] B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching," In *The International Journal of High Performance Computing Applications*, Winter 2000.

[16] G. Carey, J. Schmidt, V. Singh, and D. Yelton, "A Scalable, Object-Oriented Finite Element Solver for Partial Differential Equations on Multicomputers", In *International*

Conference on Supercomputing, Jul. 1992.

[17] D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, and T. von Eicken, "LogP: A Practical Model of Parallel Computation," *Communications of the ACM*, Nov. 1996.

[18] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz, "Prediction Models for Multi-Dimensional Power- Performance Optimization on Many Cores," in *International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2008.

[19] P. De, R. Kothari, V. Mann, "Identifying Sources of Operating System Jitter through Fine-Grained Kernel Instrumentation," in *International Conference on Cluster Computing*, Sep. 2007.

[20] P. De, R. Kothari, V. Mann, "A Trace-Driven Emulation Framework to Predict Scalability of Large Clusters in Presence of OS Jitter," in *International Conference on Cluster Computing*, Sep. 2008.

[21] P. De, V. Mann, "jitSim: A Simulator for Predicting Scalability of Parallel Applications in Presence of OS Jitter," in *The International European Conference on Parallel and Distributed Computing*, Sep. 2010.

[22] L. DeRose and D. A. Reed, "SvPablo: A Multi-Language Architecture-Independent Performance Analysis System," In *Proceedings of the International Conference on Parallel Processing (ICPP'99)*, Sep. 1999.

[23] J. Dongarra, H. Meuer, H. Simon, and E. Strohmaier (2009). Top 500 Supercomputer Sites. http://www.top500.org/.

[24] K. B. Ferreira, P. Bridges, R. Brightwell, "Characterizing Application Sensitivity to OS Interference Using Kernel-Level Noise Injection," in *International Conference on Supercomputing*, Nov. 2008.

[25] R. Gioiosa, F. Petrini, K. Davis, F. Lebaillif-Delamare, "Analysis of System Overhead on Parallel Computers," in *IEEE International Symposium on Signal Processing and Information Technology*, Dec. 2004.

[26] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, May 1988.

[27] T. Hoefler, T. Schneider, A. Lumsdaine, "Characterizing the Influence of System Noise on Large-Scale Applications by Simulation," in *International Conference on Supercomputing*, Nov. 2010.

[28] T. Hoefler, T. Mehlan, A. Lumsdaine, W. Rehm, "Netgauge: A Network Performance Measurement Framework," in *High Performance Computing and Communications*, Sep. 2007.

[29] T. Hoefler, T. Schneider, A. Lumsdaine, "LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model," in *ACM Workshop on Large-Scale System and Application Performance*, Jun. 2010.

[30] J. K. Hollingsworth, "Critical Path Profiling of Message Passing and Shared-Memory Programs," *IEEE Transactions on Parallel and Distributed Systems*, Jan. 1998.

[31] F. Ino, N. Fujimoto, K. Hagihara, "LogGPS: A Parallel Computational Model for Synchronization Analysis," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, Jun. 2001.

[32] E. Ipek, B. R. de Supinski, M. Schulz, and S. A. McKee, "An Approach to

Performance Prediction for Parallel Applications," In *The International European Conference on Parallel and Distributed Computing*, Aug 2005.

[33] T. R. Jones, L. B. Brenner, J. M. Fier, "Impacts of Operating Systems on the Scalability of Parallel Applications," Technical Report UCRL-MI-202629, Lawrence Livermore National Laboratory, Mar. 2003.

[34] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, M. Roberts, "Improving the Scalability of Parallel Jobs by Adding Parallel Awareness to the Operating System," in *International Conference on Supercomputing*, Nov. 2003.

[35] S. M. Kelly and R. Brightwell, "Software Architecture of the Light Weight Kernel, Catamount," in *Proceedings of the 47th Cray User Group Conference*, May 2005.

[36] D. Kerbyson, H. Alme, A. Hoisie, F. Petrini, H. Wasserman, M. Gittings, "Predictive Performance and Scalability Modeling of a Large-Scale Application," In *Supercomputing*, Nov. 2001.

[37] J. Labarta, S. Girona, V. Pillet, and T. Cortes, "DiP: A parallel program development environment," *Lecture Notes in Computer Science*, Apr. 1996.

[38] Lawrence Livermore National Lab, ASC Purple Benchmark Codes, http://www.llnl.gov/asci/purple/benchmarks/limited/code_list.html.

[39] Lawrence Livermore National Laboratory, “Blue Gene/L,” http://www.research.ibm.com/bluegene/.

[40] Lawrence Livermore National Lab. Parallel Computing Tutorial. https://computing.llnl.gov/tutorials/parallel_comp/

[41] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee,

"Methods of inference and learning for performance modeling of parallel applications," In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Mar. 2007.

[42] B. C. Lee and D. M. Brooks, “Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction,” in *International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[43] G. Lyon, R. Kacker, and A. Linz, "A Scalability Test for Parallel Code," *Software — Practice and Experience*, Dec. 1995.

[44] G. Marin and J. Mellor-Crummey, "Cross-Architecture Performance Predictions for Scientific Applications using Parameterized Models," In *SIGMETRICS*, June 2004.

[45] G. Marin and J. Mellor-Crummey, "Application Insight through Performance Modeling," In *IEEE International Performance Computing and Communications Conference*, Apr. 2007.

[46] J. E. Moreira, G. Almasi, C. Archer, R. Bellofatto, P. Bergner, J. R. Brunheroto, M. Brutman, J. G. Castanos, P. Crumley, M. Gupta, T. Inglett, D. Lieber, D. Limpert, P. McCarthy, M. Megerian, M. P. Mendell, M. Mundy, D. Reed, R. K. Sahoo, A. Sanomiya, R. Shok, B. E. Smith, G. G. Stewart, "Blue Gene/L Programming and Operating Environment," in *IBM Journal of Research and Development*, vol. 49, issue 2.3, pp. 367-376, Mar. 2005.

[47] R. Mraz, "Reducing the Variance of Point to Point Transfers in the IBM 9076 Parallel Computer," in *Supercomputing*, Nov. 1994.

[48] M. Muller, H. Brunst, M. Jurenz, A. Knupfer, M. Lieber, H. Mix, and W. Nagel, "Developing Scalable Applications with Vampir, VampirServer and VampirTrace," In

Proceedings of the Minisymposium on Scalability and Usability of HPC Programming Tools at PARCO, Sep. 2007.

[49] A. Nataraj, A. Morris, A. D. Malony, M. Sottile, P. Beckman, "The Ghost in the Machine: Observing the Effects of Kernel Operation on Parallel Application Performance," in *International Conference on Supercomputing*, Nov. 2007.

[50] D.S. Nikolopoulos, T.S. Papatheodorou, C.D. Polychronopoulos, J. Labarta, E. Ayguade. User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors. in *International Conference on Parallel Programming*, Aug. 2000.

[51] D. Nussbaum and A. Agarwal, "Scalability of Parallel Machines," *Communications of the ACM*, Mar. 1991.

[52] F. Petrini, D. J. Kerbyson, S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q," in *International Conference on Supercomputing*, Nov. 2003.

[53] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A Tool to Visualise and Analyze Parallel Code," In *Proceedings of WoTUG-18: Transputer and Occam Developments*, volume 44 of *Transputer and Occam Engineering*, Apr. 1995.

[54] The R Project for Statistical Computing. http://www.r-project.org/.

[55] M. Raman. China Makes World's Fastest Supercomputer. http://www.ibtimes.com/articles/76731/20101028/tianhe-1a-tianhe-supercomputer-fastest-supercomputer-china-us-nvidia-amd-gpum-cpu-chip-semiconductor.htm.

[56] P. C. Roth and B. P. Miller, "On-line Automated Performance Diagnosis on Thousands of Processes," In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Mar. 2006.

[57] M. Schulz, "Extracting Critical Path Graphs from MPI Applications," In *International Conference on Cluster Computing*, Sep 2005.

[58] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Monotya, and S. Cranford, "Open|SpeedShop: An Open Source Infrastructure for Parallel Performance Analysis," *to appear in Special Issue of Scientific Programming on Large- Scale Programming Tools and Environments*, 2008.

[59] J. P. Singh, J. L. Hennessy, and A. Gupta, "Scaling Parallel Programs for Multiprocessors: Methodology and Examples," *IEEE Computer*, Jul. 1993.

[60] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha, "A Framework for Performance Modeling and Prediction," In *Supercomputing*, Nov. 2002.

[61] M. Sottile, R. Minnich, "Analysis of Microbenchmarks for Performance Tuning of Clusters," in *International Conference on Cluster Computing*, Sep. 2004.

[62] M. Sottile, V. P. Chandu, D. A. Bader, "Performance Analysis of Parallel Programs via Message-Passing Graph Traversal," in, *IEEE International Parallel & Distributed Processing Symposium*, Feb. 2006.

[63] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 1994.

[64] E. Sundararajan. Oak Ridge Looks Toward 20 Petaflop Supercomputer. http://elankovansundararajan.wordpress.com/2011/03/14/oak-ridge-looks-toward-20-petaflop-supercomputer/.

[65] D. Tsafrir, Y. Etsion, D.G. Feitelson, S. Kirkpatrick, "System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications," in *International Conference on Supercomputing*, Jun. 2005.

[66] L. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM*, Aug. 1990.

[67] J. Vetter and C. Chambreau, “mpiP: Lightweight, Scalable MPI Profiling,” http://www.llnl.gov/CASC/mpip/, Apr. 2005.

[68] Z. Wang and M. F. O’Boyle, “Mapping Parallelism to Multi-cores: A Machine Learning Based Approach,” in *Symposium on Principles and Practice of Parallel Programming*, Feb. 2009.

[69] Wikipedia, “Scalability web page,” http://en.wikipedia.org/wiki/Scalability.

[70] Wikipedia, “Speedup web page,” http://en.wikipedia.org/wiki/Speedup.

[71] F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler, "Architectural Requirements and Scalability of the NAS Parallel Benchmarks," In *Supercomputing*, Jun. 1999.

[72] P. H. Worley, "The Effect of Time Constraints on Scaled Speedup," *SIAM J. Sci. Stat. Computing*, Sep. 1990.

[73] L. T. Yang, X. Ma, and F. Mueller, "Cross-Platform Performance Prediction of Parallel Applications using Partial Execution," In *Supercomputing*, Nov. 2005.