A Regression-Based Approach to Scalability Prediction*

Bradley J. Barnes
Dept. of Computer Science
The University of Georgia
barnes@cs.uga.edu

Barry Rountree
Dept. of Computer Science
The University of Georgia
rountree@cs.uga.edu

David K. Lowenthal
Dept. of Computer Science
The University of Georgia
dkl@cs.uga.edu

Jaxk Reeves
Department of Statistics
The University of Georgia
jaxk@stat.uga.edu

Bronis de Supinski
Lawrence Livermore
National Laboratory
bronis@llnl.gov

Martin Schulz
Lawrence Livermore
National Laboratory
schulzm@llnl.gov

ABSTRACT

Many applied scientific domains are increasingly relying on large-scale parallel computation. Consequently, many large clusters now have 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.

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.

Categories and Subject Descriptors

I.6.5 [**Model Development**]: Modeling Methodologies

General Terms

Measurement, Experimentation

Keywords

Modeling, MPI, Prediction, Regression, Scalability

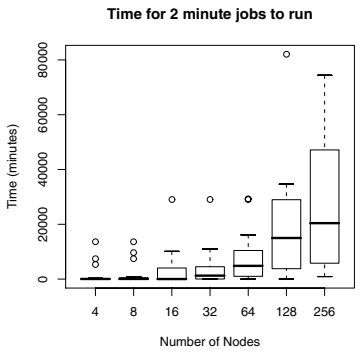

Figure 1: Median time for a short application to commence execution on the LLNL "Thunder" cluster when requesting different node counts. The system had 986 nodes.

1. INTRODUCTION

Nearly all applied sciences today rely on parallel computation. Applications from a wide variety of domains run on large systems with thousands of processors, such as BlueGene/L. However, these applications often achieve poor speedup and thus poor efficiency. For example, Alam et al. [2] characterized performance for a biomolecular simulation and found that one time-consuming part of the program achieved a speedup of only 10 on 1024 processors.

This inefficiency is costly to both the owners and the users of the systems. To the supercomputer center, it wastes the system in terms of money and possibly power consumption. To the user, it reduces system availability because other users allocate more processors than necessary. A mechanism to predict the parallel efficiency of applications—without having to understand their low-level details and without executing them at scale—could help increase availability. Application scientists could use the mechanism 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 reduce response time for the specific application. For example, Figure 1 shows that the worst-case time to acquire nodes appears to increase exponentially in the number of nodes. In our own experience—carrying out experiments for the performance section of this paper—it took nearly a month to be granted 256 nodes for one of our applications.

System procurement decisions are often based on results from

*This work was supported by NSF grant CCF-0429285. Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-CONF-400700).

ICS'08, June 7–12, 2008, Island of Kos, Aegean Sea, Greece.

prototype systems with many fewer processors than the system that will eventually be purchased. An accurate prediction mechanism that indicates what performance will be achieved on the larger (as yet unbuilt) system could improve its suitability for its actual workload. Predictions can also guide optimization and provide system diagnostics, for example identifying when operating system activity interferes with expected performance [24] or when non-uniform memory access necessitates finer control of processor to memory mappings.

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 paper 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 achieved median prediction errors of between 6.2% and 17.3% over all nontrivial programs. This includes 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 paper is organized as follows. Section 2 describes our techniques for performance prediction. Next, we describe our experimental methodology in Section 3 and the results of using our techniques on seven applications in Section 4. Finally, Section 5 places our approach in the context of prior work, while Section 6 summarizes our findings and future directions.

2. PERFORMANCE PREDICTION

This section describes our prediction techniques. We start with definitions along with our assumptions and then detail our basic regression-based approach. Finally, we describe our three techniques.

2.1 Definitions and Assumptions

In this paper 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 is a reasonable assumption, 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 [18]. Our procedure models execution time as some function of these input variables. The quality of our model depends on considering all 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 [3] constrain the values of the input variables—we overrode this limitation in our training sets.

2.2 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, ..., 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, $\hat{T}$, of the execution time T:

$$\hat{T} = F(x_1, x_2, \ldots, x_n, q). \tag{1}$$

The idea is that $\hat{T} \approx T$, with small error. Once we determine $\hat{T}$, we use it to predict execution time for any arbitrary input variable set (a_1, a_2, ..., a_n) and p processors. We emphasize that we produce $\hat{T}$ *without* any data from runs using p processors , since we choose $p_0 < p$.

The scale in which the error between $\hat{T}$ and the true T is measured is crucial. For most applications, variability increases as T increases, so we use relative error:

$$E = |(T - \hat{T})|/T \tag{2}$$

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:

$$log_2(T) = log_2(F(x_1, x_2, \ldots, x_n, q)) + error \tag{3}$$

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

$$log_2(T) = \beta_0 + \beta_1 \log_2(x_1) + \beta_2 \log_2(x_2) + \ldots + \beta_n \log_2(x_n) + \beta_q \log_2(q) + error \quad (4)$$

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 3 to obtain Equation 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 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 $log_2(x_i)$. Our results for the seven applications that we examine show that most of the variability due to the input variables $(x_1, x_2, \ldots, x_n)$ is explained by models of the form:

$$log_2(T) = \beta_1 \log_2(x_1) + \beta_2 \log_2(x_2) + \ldots + \beta_n \log_2(x_n) + g(q) + error \quad (5)$$

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 applications that we examine in Section 4, the simple linear function:

$$g(q) = \gamma_0 + \gamma_1 \log_2(q) \quad (6)$$

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.

2.3 Techniques

Our most straightforward approach uses the total execution time for T in Equation 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 address this, 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 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.

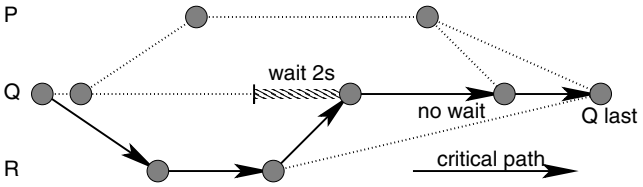

Figure 2: Critical path: P, Q, and R are MPI tasks with edges representing messages.

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 2 shows. Any communication time on this path is purely communication (i.e., sending/receiving), which helps our model avoid overestimating it.

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.

3. EXPERIMENTAL METHODOLOGY

We tested our techniques using seven applications: five from the NAS suite and two from the ASC Purple/Blue suites. 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 embarassingly 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 or (2) communication time increases with processors. The threshold for deeming an application computation bound is currently 90% (average) computation for any processor configuration used during training (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 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 use a hybrid of strong and weak scaling in which we increase the problem size on large

Application	Name 1	Range	Name 2	Range	Name 3	Range	Training runs per proc. count
BT	$problem\_size$	20–500	–	–	–	–	24
CG	NA	500K–3M	$NONZER$	14–24	–	–	24
EP	m	2^{28}–2^{38}	–	–	–	–	29
LU	$isiz1$	200–1000	$isiz2$	200–1000	$isiz3$	200–1000	16
SMG	$DIM1$	290–315	$DIM2$	290–315	$DIM3$	290–315	54
SP	$problem\_size$	40–1400	–	–	–	–	35
Sweep3d	IT_G	300–500	JT_G	300–500	KT_G	300–500	64

Table 1: The seven applications and their input variable names and ranges.

processor counts to get reasonable execution times with EP, SP and LU.

We observe results (T) for each instrumented run on q processors and fit a linear model of the form given in Equation 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 [12] 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 10-30 training runs per input variable. 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.

4. 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 CPU has 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 CPU 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 CPUs, 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 [24]. Atlas uses a priority-based batch queueing 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.

4.1 Summary of Results

We present the full results of our experiments in Sections 4.3 and 4.4 while Table 2 shows 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, there is no split and so only one degree of freedom). More details are given in Sections 4.2, 4.3, and 4.4.

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 4.4.

4.2 Format of Detailed Results

We show two graphs for computation-bound applications and three 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). Additionally, if the program is considered communication intensive (see Section 3), 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 (shown 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 3 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 shows the quality of the fit obtained

Application	Median Error, Total		Type (Best)	Median Error, Separate		Type (Best)
	Best (Procs)	Worst (Procs)		Best (Procs)	Worst (Procs)	
BT	6.7% (484)	13.0% (100)	L	—	—	—
CG	16.0% (256)	120% (128)	L	12.2% (256)	66.3% (128)	Q/L/CP
EP	0.1% (512)	0.6% (128)	L	—	—	—
LU	13.8% (512)	15.8% (128)	Q	—	—	—
SP	7.7% (100)	12.8% (256)	L	7.7% (100)	12.1% (256)	L/Q/Max
SMG	14.4% (128)	92.7% (32)	Q	6.7% (128)	25.6% (32)	Q/Q/Max
Sweep3d	32.8% (512)	59.3% (256)	Q	17.3% (512)	33.2% (128)	Q/Q/Max

Table 2: Summary of results for all applications. The best and worst median errors are shown, both when regressing on total execution time and (if applicable) separately on computation and communication. For each, the number of processors used for prediction is 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.

(over all processor configurations) using the three alternatives in the middle graph. This is 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.

4.3 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 3, 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 4). 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 5).

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 experiments 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, as it is not for LU and EP. As mentioned earlier, we only regress separately when either (1) both computation and communication are significant or (2) communication is increasing.

4.4 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; we present results in detail for CG and SMG.

CG overview. CG is significantly more difficult to predict than any of the compute-bound applications. The results are shown in Figure 6 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 identically. 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 [7] 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 is 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.

SMG overview. SMG is written as a weak scaling application: the input parameters specify a single cube size for each processor.

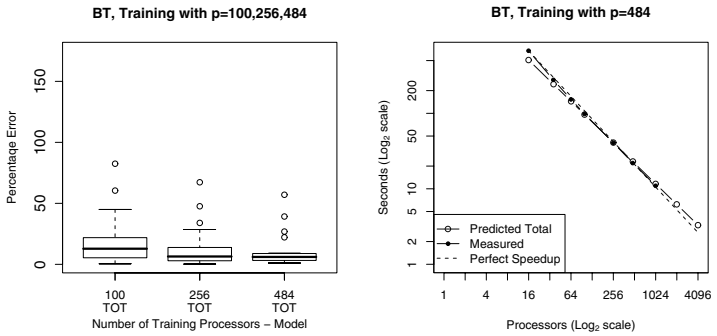

Figure 3: Results for predicting BT on 1024 processors. The right-hand graph shows results for $problem\_size = 500$.

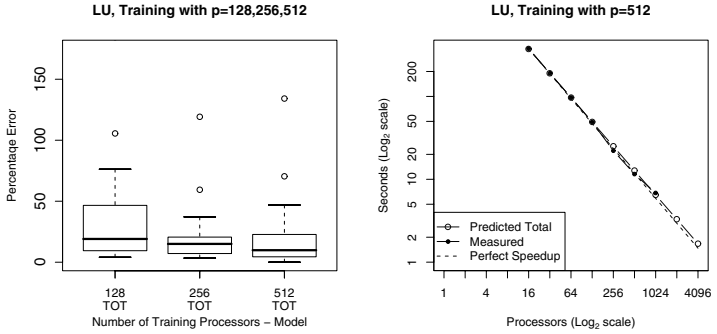

Figure 4: Results for predicting LU on 1024 processors. The right-hand graph shows results for all input variables equal to 450.

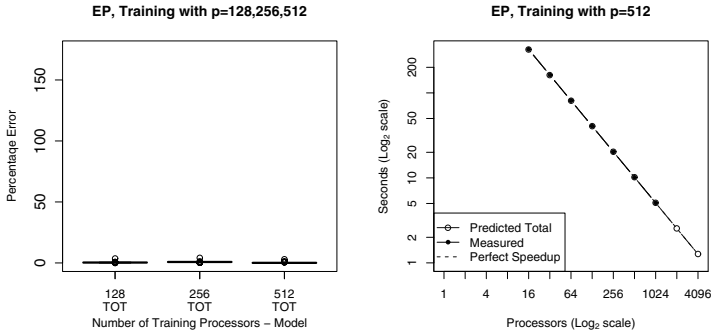

Figure 5: Results for predicting EP on 1024 processors. The right-hand graph shows results for $m = 2^{34}$.

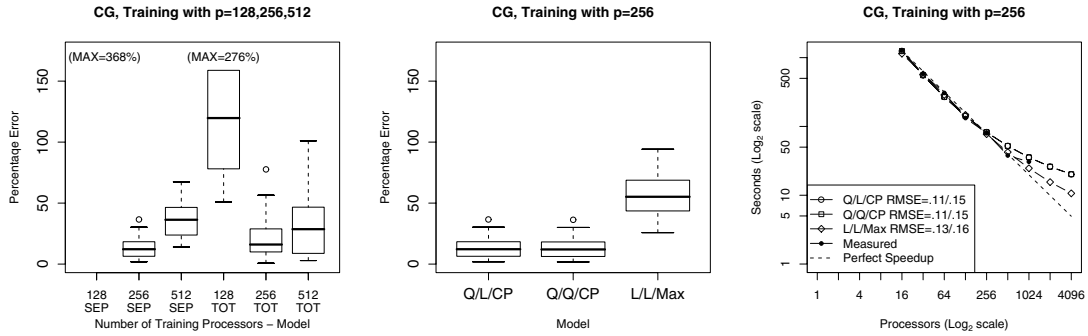

Figure 6: Results for predicting CG on 1024 processors. The right-hand graph shows results for $NA = 2M$ and $NONZER = 24$.

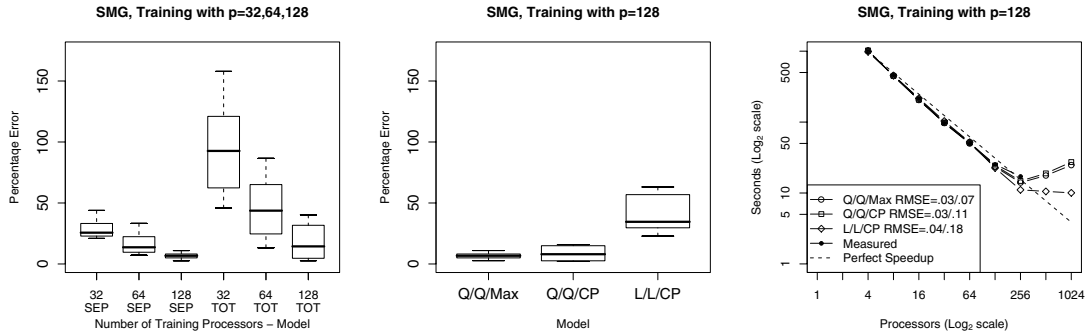

Figure 7: Results for predicting SMG on 256 processors. The right-hand graph shows results for $DIM1 = 300$.

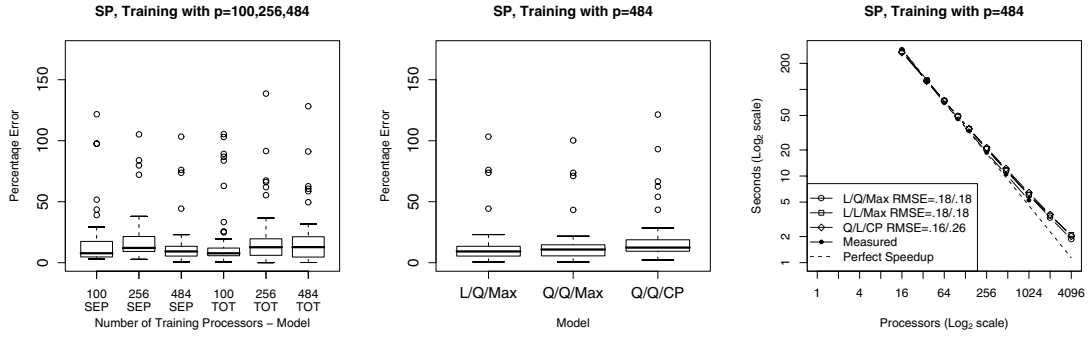

Figure 8: Results for predicting SP on 1024 processors. The right-hand graph shows results for $problem\_size = 450$.

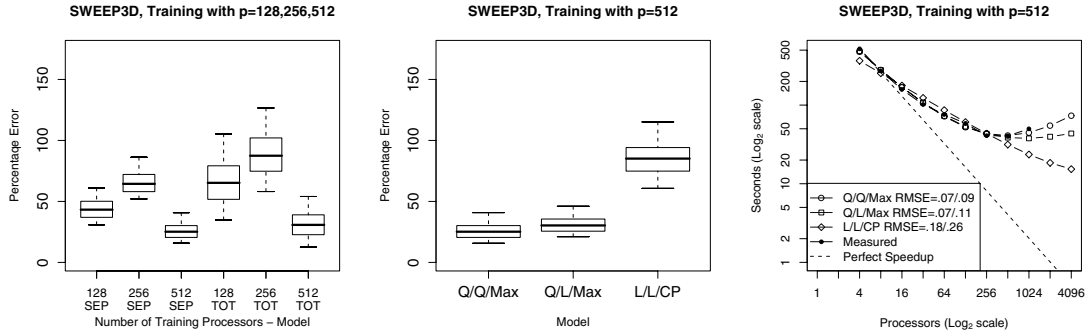

Figure 9: Results for predicting Sweep3d on 1024 processors. The right-hand graph shows results for $IT_G = 400$, $JT_G = 400$, and $KT_G = 400$.

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 7 (previous page) 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.

SP overview. Figure 8 (previous page) 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.

Sweep3d overview. Sweep3d is an application with which we intentionally stressed our prediction system, by choosing a $P \times 1$ processor grid—which is not typical for this application. This caused any processor rank larger than the x-dimension (IT_G) of the data grid to be assigned no work and therefore simply remain idle. Figure 9 (previous page) 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 (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 $P \times 1$ 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.

Discussion. Generally speaking, training with as many processors as possible will produce a better fit. However, the case of CG shows this does not always hold. 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, are small, as Figure 8 shows.

For SMG, the difference in fit quality is quite significant. The middle graph in Figure 7 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, this 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.

5. RELATED WORK

We extend on a significant body of prior work. Many researchers have explored predicting performance of parallel applications. Most prior work either predicts for processor counts explored on other systems or uses extensive manual analysis to derive analytic models.

The work most closely related to ours uses various regression approaches to predict application performance across a range of input parameter values. For example, neural networks model the parameter space to predict execution time, generally with errors of 10% or less [15]. Direct comparisons demonstrated that piecewise polynomial regression provides similar accuracy [18]. Unlike these previous regression-based approaches, we identify techniques to separate computation and communication that support extrapolations to larger processor counts.

Many other black-box modeling approaches exist. Lyon *et al.* reason about execution behavior using Taylor expansions to determine scalability properties [19]. This more theoretical approach is complementary to our empirical approach and could conceptually be used in tandem with it. Another black box technique [34] predicts performance across platforms through partial execution of iterative programs but is restricted to the system sizes used for the partial executions. A final approach is to use a combination of static and dynamic analysis to predict performance on different architectures for different inputs [20] (and in later work, the approach is extended to find performance bottlenecks [21]). 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. Our framework only requires only relinking of the application with the PMPI library.

MetaSim provides a general performance modeling framework [29]. MetaSim uses Atom [30] or other instrumentation mechanisms like Dyninst [8] 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 [17], which consumes an MPI trace to sim-

ulate network performance. The memory and MPI traces are tied to the original processor count and, thus, unlike our approach, this work does not support scaling predictions.

Kerbyson et al. derive an analytical performance model for the ASCI Sage application [16]. This white-box approach requires detailed analysis of data structures and program constructs, such as loop nests. They combine this analysis 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 approach is mostly automated and requires little analytic effort. Brehm *et al.* make predictions using regression, including ones that separate computation and communication [6]. Their approach requires creating computation and communication models for the program that is being modeled. Several other researchers have explored similar white-box scalability analysis approaches, both from an algorithmic and an architectural perspective [13, 33, 23, 9, 28, 32]. In general, they derive application or architecture specific models through detailed analysis, which requires significant effort that is not readily automated. Other white-box approaches, such as *modeling assertions* [1], require code modifications in order to predict workload and memory requirements. Our techniques at most use the MPI profiling interface for instrumentation, which only requires relinking the application.

Many have investigated analytic modeling of parallel machines. The best known examples are LogP [10] and BSP [31]. 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, this becomes increasingly challenging. Another approach requires no user intervention to create a static cost model [4]. 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 VampirTrace [22], svPablo [11], TAU/ParaProf [5], and Paraver [25], just to name a few. Most of these tools focus on providing assistance in optimizing applications, particularly for very large processor counts [26]. Previous work focusing on optimization has investigated techniques to capture the critical path in MPI programs [14, 27]; we build on the algorithm developed by Schulz [27] in our work.

6. SUMMARY AND FUTURE WORK

This paper has contributed a new 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.

Future Work. This work represents the beginning of our effort to achieve accurate predictions of parallel programs, and much future work remains to be done. Our next task is to experiment with larger processor configurations; while the number of processors used in training will also be larger, it remains to be seen what happens to prediction accuracy. Second, we need to investigate more applications, including ones that exhibit load imbalance. Third, we will investigate further ways to determine the smallest number of processors for training that will produce quality predictions. Fourth, we are investigating a novel clustering scheme to predict more accurately the effect of the memory hierarchy on computation time. Finally, we will further refine communication predictions by separately modeling calls that scale differently.

7. REFERENCES

[1] S. R. Alam and J. S. Vetter. Hierarchical model validation of symbolic performance models of scientific applications. In *Euro-Par*, Aug. 2006.

[2] S. R. Alam, J. S. Vetter, P. K. Agarwal, and A. Geist. Performance characterization of molecular dynamics techniques for biomolecular simulations. In *PPOPP*, pages 59–68, Mar 2006.

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

[4] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 213–223, Apr. 1991.

[5] 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 (Euro-Par 2003)*, pages 17–26, Aug. 2003.

[6] J. Brehm, P. H. Worley, and M. Madhukar. Performance modeling for SPMD message-passing programs. *Concurrency: Practice and Experience*, 10(5):333–357, Apr. 1998.

[7] 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.

[8] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[9] 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*, pages 387–396, 1992.

[10] 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*, 39(11):78–85, Nov. 1996.

[11] 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)*, Sept. 1999.

[12] T. R. P. for Statistical Computing. http://www.r-project.org/.

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

[14] J. K. Hollingsworth. Critical path profiling of message passing and shared-memory programs. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):29–40, 1998.

[15] E. Ipek, B. R. de Supinski, M. Schulz, and S. A. McKee. An approach to performance prediction for parallel applications. In *Euro-Par*, pages 196–205, Aug 2005.

[16] D. Kerbyson, H. Alme, A. Hoisie, F. Petrini, H. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Supercomputing*, Nov. 2001.

[17] J. Labarta, S. Girona, V. Pillet, and T. Cortes. DiP: A parallel program development environment. *Lecture Notes in Computer Science*, 1124:665–??, 1996.

[18] 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 *PPOPP*, pages 249–258, 2007.

[19] G. Lyon, R. Kacker, and A. Linz. A scalability test for parallel code. *Software — Practice and Experience*, 25(12):1299–1314, Dec. 1995.

[20] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *SIGMETRICS 2004*, pages 2–13, June 2004.

[21] G. Marin and J. Mellor-Crummey. Application insight through performance modeling. In *IEEE International Performance Computing and Communications Conference*, Apr 2007.

[22] M. Müller, H. Brunst, M. Jurenz, A. Knüpfer, 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 2007, to appear*, Sept. 2007.

[23] D. Nussbaum and A. Agarwal. Scalability of parallel machines. *Communications of the ACM*, 34(3):56–61, Mar. 1991.

[24] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Supercomputing*, 2003.

[25] 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*, pages 17–31, Apr. 1995.

[26] P. C. Roth and B. P. Miller. On-line automated performance diagnosis on thousands of processes. In *PPOPP*, pages 69–80, Mar 2006.

[27] M. Schulz. Extracting critical path graphs from MPI applications. In *IEEE Cluster*, Sep 2005.

[28] J. P. Singh, J. L. Hennessy, and A. Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *IEEE Computer*, 26(7):42–50, July 1993.

[29] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *Supercomputing*, Nov. 2002.

[30] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

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

[32] 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*, 1999.

[33] P. H. Worley. The effect of time constraints on scaled speedup. *SIAM J. Sci. Stat. Computing*, 11(5):838–858, Sept. 1990.

[34] L. T. Yang, X. Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Supercomputing*, 2005.