



LAWRENCE  
LIVERMORE  
NATIONAL  
LABORATORY

# Using Focused Regression for Accurate Time-Constrained Scaling of Scientific Applications

B. Barnes, J. Garren, D. Lowenthal, J. Reeves, B.  
de Supinski, M. Schulz, B. Rountree

January 29, 2010

IPDPS 2010  
Atlanta, GA, United States  
April 19, 2010 through April 23, 2010

## **Disclaimer**

---

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

# Using Focused Regression for Accurate Time-Constrained Scaling of Scientific Applications

Brad Barnes<sup>\*</sup>, Jeonifer Garren<sup>†</sup>, David K. Lowenthal<sup>‡</sup>, Jaxk Reeves<sup>†</sup>, Bronis R. de Supinski<sup>§</sup>,  
Martin Schulz<sup>§</sup>, and Barry Rountree<sup>‡</sup>

<sup>\*</sup>*Department of Computer Science, The University of Georgia*

<sup>†</sup>*Department of Statistics, The University of Georgia*

<sup>‡</sup>*Department of Computer Science, The University of Arizona*

<sup>§</sup>*Lawrence Livermore National Laboratory*

**Abstract**—Many large-scale clusters now have hundreds of thousands of processors, and processor counts will be over one million within a few years. Computational scientists will want to take advantage by scaling applications to run on these new clusters. The goal of *time-constrained scaling*, which is often used, is to hold total execution time constant while increasing the problem size along with the processor count. However, determining the input parameters to use to achieve time-constrained scaling is not necessarily straightforward due to complex interactions between parameters, the processor count, and execution time.

In this paper we develop a novel gray-box, focused regression-based approach that assists the computational scientist with maintaining constant run time on increasing processor counts. Combining application-level information from a small set of training runs, our approach allows prediction of the input parameters that result in similar per-processor execution time at larger scales. Our experimental validation across seven applications showed that median prediction errors are less than 13%.

## I. INTRODUCTION

Nearly all applied sciences today make use of parallel computation. Applications from a wide variety of domains run on large systems with tens or hundreds of thousands of processors, such as ORNL’s Jaguar [33], ANL’s Intrepid [2], LANL’s Roadrunner [20] and LLNL’s BG/L [18]. However, these systems are a scarce resource in high demand. For example, we experimented on LLNL’s Thunder cluster and found that the worst-case node acquisition time increased roughly exponentially with the number of nodes, with the acquisition of 256 nodes (roughly one-quarter of the total) taking up to a month. We anticipate that on larger clusters, acquiring a similar fraction of the system will take a similar time, so the user may not easily get a “second chance” to determine the correct input parameters (for this paper, input parameters refer to those values input by the user that contribute significantly to execution time).

This paper focuses on *time-constrained scaling* [30]. Instead of using a larger number of processors to solve a problem faster, larger problems are solved and overall execution time is kept constant. Unfortunately, understanding how programs scale is difficult. 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 include nonlinear effects in computation and communication, along with non-obvious relationships between input parameters and execution time.

In this paper we develop a regression-based technique that allows accurate time-constrained scaling of applications. We use a gray-box technique, taking as input a small amount of application-level information. Our basic idea is to choose a small series of training runs, varied over different, smaller processor counts, and then to use *focused regression* to make predictions of input parameters to use 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 allows a small number of training runs and also improves prediction accuracy.

This paper 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 [5], which uses *non-focused regression* 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%. This includes applications for which there exists a complex interaction between multiple input parameters and execution time.

The rest of this paper is organized as follows. Section II provides motivation for this work. Section III describes our statistical techniques, in particular focused regressions. Next, Section IV describes our experimental methodology and results on seven applications. Finally, Section V places our approach in the context of prior work, while Section VI summarizes our findings and future directions.

## II. MOTIVATION

The computational scientist (“scientist” for the remainder of this paper) has several options when more processors become available. The first option is to use *strong scaling* [36], where one runs the *same* program instance, i.e., uses identical input parameters. This is the most frequent type of scaling that appears in the computer science literature. However, it is becoming more commonplace to use *time-constrained scaling*, a term coined by Singh et al. [30], in which the scientist attempts to keep total run time constant. This allows solutions to problems that were previously unexplored and is generally more intuitive from the scientist’s perspective.

Strong scaling is preferred when a specific problem must be solved as quickly as possible. However, the amount of parallelism available is immutable and, therefore, strong scaling will fail to reduce runtime after a sufficiently large processor count has been reached. 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 (which is usually referred to as *weak scaling* [36] and has as its goal making computation time per processor constant), as the processor count increases. 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,

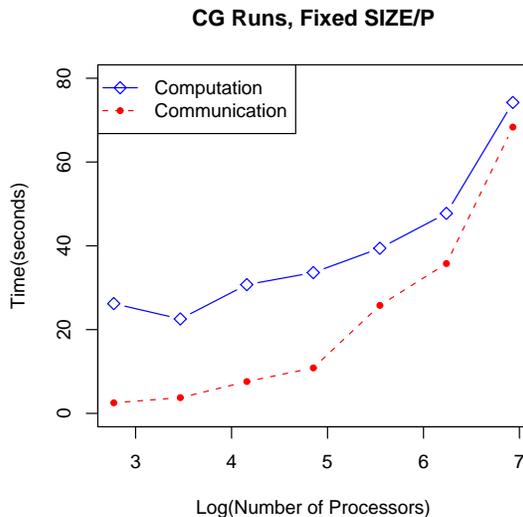


Figure 1. Computation and communication times for CG as the number of processors increases. The ratio of *SIZE/P* is fixed; *SIZE* ranges from 40,000 to 2,560,000, and *P* ranges from 16 to 1024. The value of *NONZER* is held constant.

defined by (global) dimensions  $N_1$  and  $N_2$ , that are 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  $N_1 \times N_2$  should be doubled when the processor count doubles. Some applications do not have an obvious relationship between the parameters (e.g., CG from the NAS suite [3]).

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 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.

## III. FOCUSED REGRESSION

This section discusses our focused regression technique. First, we describe the general idea. Then, we discuss our basic model, which we use for applications for which it needs no program-level information. Finally,

we discuss extensions that handle more complicated applications.

### A. Overall Technique

The scientist is responsible for providing appropriate input. 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 is responsible for certain application-level information, which is in Table I and described 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 will need to be executed. To control training time, these will be executed for only a limited number of timesteps. Therefore, we assume the timestep loop is known.

With the value of  $P_t$  input by the scientist, our technique proceeds as follows. For simplicity, we first present the case where there is only one input parameter. We assume that the scientist provides us with data from points with time  $\approx T$ , 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  $\approx 1.1 \times T$  and  $\approx 0.9 \times T$ . We determine the appropriate value of the input parameter to achieve this through inspection of the data that is provided by the scientist. 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  $\approx T$ . To extend this technique to multiple input parameters, please see the procedure described in Section III-C.

### B. 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 other key characteristic for determining run time in programs with low amounts of communication is the computation time, denoted  $W$ . For simple programs,  $W$  can often be easily

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 I  
APPLICATION-LEVEL INFORMATION NEEDED FROM THE SCIENTIST FOR OUR SEVEN PROGRAMS.

determined from the input parameters (i.e., all that is significant is the product of the parameters ( $z_i$ 's), or  $W = f(z_1 \times z_2 \times \dots \times z_n)$ ), and  $T$  is approximately proportional to  $W/P$ . More generally, we have

$$\log(T) = \beta_0 + \beta_1 \log(W) + \beta_2 \log(P_t) + \epsilon \quad (1)$$

where  $\beta_0$ ,  $\beta_1$ , and  $\beta_2$  are coefficients to be estimated from examining a set of observed (T,W,Q) triplets,  $Q$  is the number of processors used in a training run ( $Q < P_t$ ), and  $\epsilon$  is the error. The  $\beta$ 's are generally estimated in a way that minimizes 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  $Q \in \{2, \dots, P_t/2\}$ . We vary the values of  $W$  and  $Q$  on the sample runs and then use regression to generate Equation 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.

Note that the prediction of run time is performed in log-scale. This is common because the errors in prediction are well known to be proportional to the expected time—we are concerned with relative errors. Working in log-scale implicitly handles this. The base of the log makes no fundamental difference; we use  $\log_2$  in this paper for mathematical convenience. The coefficients  $\beta_1$  and  $\beta_2$  in Equation 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 1 (e.g., time is proportional to the quotient of  $W$  and  $P_t$ ).

While the model in Equation 1 is relatively simple, it works quite well for applications that can be described with computation-dominated, simple-array based programs. The applications we used to evaluate our focused regression technique in this paper (see Section IV) are

listed in Table I. The first three are well predicted with Equation 1: BT, LU, and SP (from the NAS suite [3]). All three have a high computation-to-communication ratio and have a single input parameter.

### C. General Model

For more complex applications, simply applying Equation 1 will be insufficient, for several reasons:

- 1) In some applications, computation time ( $W$ ) is not easily determined in advance. Rather, there are input parameters whose values can be specified in advance, and these parameters determine computation in an unknown, or at best non-obvious way. In such cases, one may need to examine a number of potential predictor parameters to determine which are significant predictors of time, and to model the relationship between  $T$  and these variables. This occurs in CG, as indicated by Table I.
- 2) For applications with a significant amount of time spent in communication, modeling only total execution time will produce inaccurate predictions. This is because computation and communication can scale at different rates, which the training runs will capture only if modeled separately. As mentioned earlier, this situation is shown in Figure 1. Currently, we do not subdivide further into phases, either computation or communication. For the most part, with our applications prediction quality is good without further subdividing. We are currently investigating breaking computation and communication into smaller phases. One example would be breaking communication calls into groups that have similar scaling behavior (e.g., logarithmic-scaling collectives versus linear-scaling collectives).
- 3) For applications in which the program specifies a processor grid to allow the scientist to control the data distribution,  $T$  is not only a function of  $P_t$ , but also of  $P_1, P_2, \dots, P_n$ , where  $n$  is the number of processor grid dimensions and  $P_t$  is the product of the  $P_i$ 's. Both SMG and Miranda fall into this category. Parameterizing this in a meaningful way can be difficult and depends to some extent on knowledge of the program structure. In cases where this knowledge can be exploited, significantly better fits can be obtained by using the values of the processor dimensions rather than just  $P$ . In such cases, in addition to providing time estimates for various input combinations, the models can be used to give scientists insight as to

what processor configurations, for a fixed  $W$  and  $P_t$ , allow the programs to run most quickly.

Our prototype handles each of these possibilities as follows.

*Case 1:* If the input parameters are not obviously related, we instead use the more general equation for execution time:

$$\log(T) = \beta_0 + \beta_1 z_1 + \beta_2 z_2 + \dots + \beta_n z_n + \beta_{n+1} \log(P_t) + \epsilon \quad (2)$$

Here,  $z_i$  is the  $i^{\text{th}}$  input parameter describing the data. We will 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. [19]).

*Case 2:* If communication is significant, we use separate regressions for computation and communication. Both follow the same form of either Equation 1 (if the input parameters are related) or 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 with smaller percentages it is sufficient to regress only on total time. 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 2 by replacing the  $P_t$  term with terms for  $P_1, P_2$ , etc. However, while intuitive, experiments showed that this is not an effective technique. Specifically, the problem is that the data distribution, as specified by the processor grid, greatly affects application execution time (as will be shown in Section IV) and in a nonlinear manner. Using a single regression will therefore result 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, it is necessary to use the input parameter space to determine a focal region. This is because the input parameter space is large, and it is quite difficult to cluster sample runs around execution time.

In the focal region, then, Equation 1 or Equation 2 is used, depending on whether the input parameters are

related or not, as described above. One interesting aspect here is that the typical strategy when creating regression models is to use 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.

Consider an example, with one of our applications, SMG, which has six input parameters—three processor dimensions,  $P_x$ ,  $P_y$ , and  $P_z$ , along with three grid dimensions,  $x$ ,  $y$ , and  $z$ . 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 a doubling of one of these three processor dimensions will result.

For each prediction, we use a *different* regression based on experiments in the focal region. Figure 2 shows two different focal regions, one of which,  $(1, 32, 32)$ , would be used in the preceding example. The figure shows that our prototype uses those processor grids (shown in black) at lower (total) numbers of processors which are most proportionally similar to the grid at the target number of processors. As the results in the next section show, if we include data from grids that are not proportionally similar, the results degrade. Note that this figure sets  $P_x = 1$ , because if  $P_x$  is also varied, the picture becomes quite complex. However, our prototype handles the general three-dimensional case.

#### IV. RESULTS

This section discusses results of our focused regression prototype in making time-constrained scaling predictions. For evaluation, we used two different clusters at Lawrence Livermore National Laboratory: the *Atlas* cluster and the *Hera* cluster. The former has 1152 four-socket, dual-core AMD Opteron nodes with 16GB RAM, 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 [26]. 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.

##### A. 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, to get finer-grain results (e.g., omitting blocking time and collecting only network and copying time) requires instrumenting the entire MPI library. Then, we 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 conserves machine time as well as produces better results by using focal regions (as described in the previous section).

An important assumption that we make is that an application can be run 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). Without the ability to execute the program in configurations of our choice, we may not be able to collect the data that we need to make accurate predictions. This essentially means that we are assuming that scientists write programs that are flexible and provide meaningful timing results, if not physical results, for any combination of the input parameters.

For 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 the 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.

##### B. Applications

We tested our techniques using seven applications. Four are from the NAS suite [3]; these include BT,

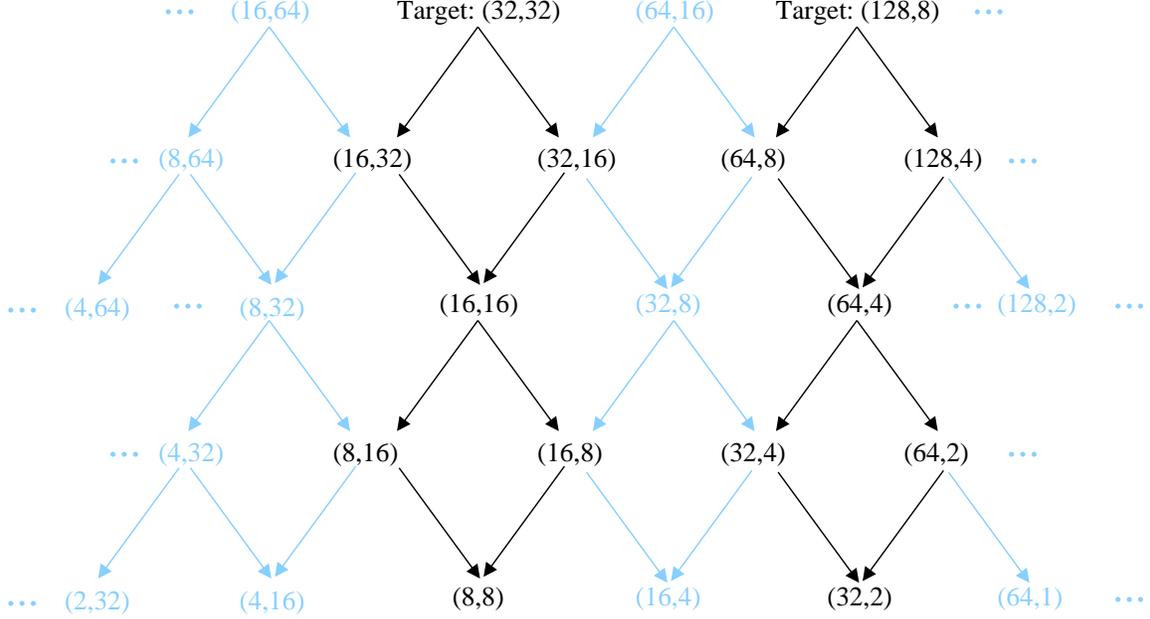


Figure 2. Processor grids (only shown down to 64 processors) used in SMG to predict  $P_x = 1$ ,  $P_y = 32$ , and  $P_z = 32$ , and  $P_x = 1$ ,  $P_y = 128$ , and  $P_z = 8$ , respectively. For all vertices in the graph,  $P_x = 1$ .

SP, CG, and LU<sup>1</sup>. CG is a conjugate gradient program, and LU, BT and SP solve PDEs using three different techniques: lower-upper symmetric Gauss-Seidel, block tridiagonal, and scalar pentadiagonal. The two others, SMG and Sweep3d, are from the ASC suite; the former is a three-dimensional multigrid solver, and the latter is a three-dimensional neutron transport code. The last application is Miranda, which is an industrial-strength hydrodynamics application.

### C. Summary of Results

We make the following general observations. First, 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. Second, for the three more complex applications, it is clear that we *must* generate different regressions for different focal regions to achieve accuracy. In particular, if one does not use a focal region, the median error can be as high as 75%.

### D. Single Parameter Programs

First, we studied three programs that have only one important parameter: BT, SP, and LU. These pro-

<sup>1</sup>The others (FT, IS, MG, EP) are unsuitable for our approach because either they restrict the input sizes to the extent that there is insufficient data available, or, in the case of EP, it is trivial (one parameter and zero communication).

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

Table II  
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.

grams 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 II 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.

### E. 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, which is creating 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 [8], 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 III, with SMG, we must create a regression for different focal regions; specifically, there is one different regression that predicts for each processor configuration. For these results, we used six of the possible processor configurations at 1024 processors.

Figure 3 shows the median errors for all program execution times that we predicted using each of the three techniques, and Table III summarizes the results.

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 all 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, for SMG we do not give the prediction error when using proportional scaling. This is because it is completely unclear what it means to do proportional scaling when there are six input parameters, and some of them have strict restrictions on their values (the processor grid dimensions).

*Sweep3d*: Sweep3d has fewer input parameters (five) than SMG (six) because its processor grid is only two dimensional. In addition, the specification of the grid is global, not local. For time-constrained scaling, three of the five input parameters are unconstrained, which means that like SMG, there are many ways to scale Sweep3d. We chose the same approach for scaling as SMG (see above), and used focal regions in exactly the same way.

Figure 3 and Table III 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%).

*CG*: Figure 3 shows the results when applying focused regression to CG, and Table III summarizes this data. The figure shows that we produce predictions whose median error is 12%, and the worst-case error is 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.

We also investigated the naive time-constrained scaling prediction. However, the question is, if not using our approach, how would the programmer scale CG to keep the execution time constant? As mentioned earlier, CG has two parameters: *SIZE* and *NONZER*. There are three intuitive potential choices: 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: (1) as CG is at its core a one-dimensional data structure (sparse matrix), increasing both parameters by  $\sqrt{2}$  seems physically unrealistic, and (2) CG requires both parameters be integers, and 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.

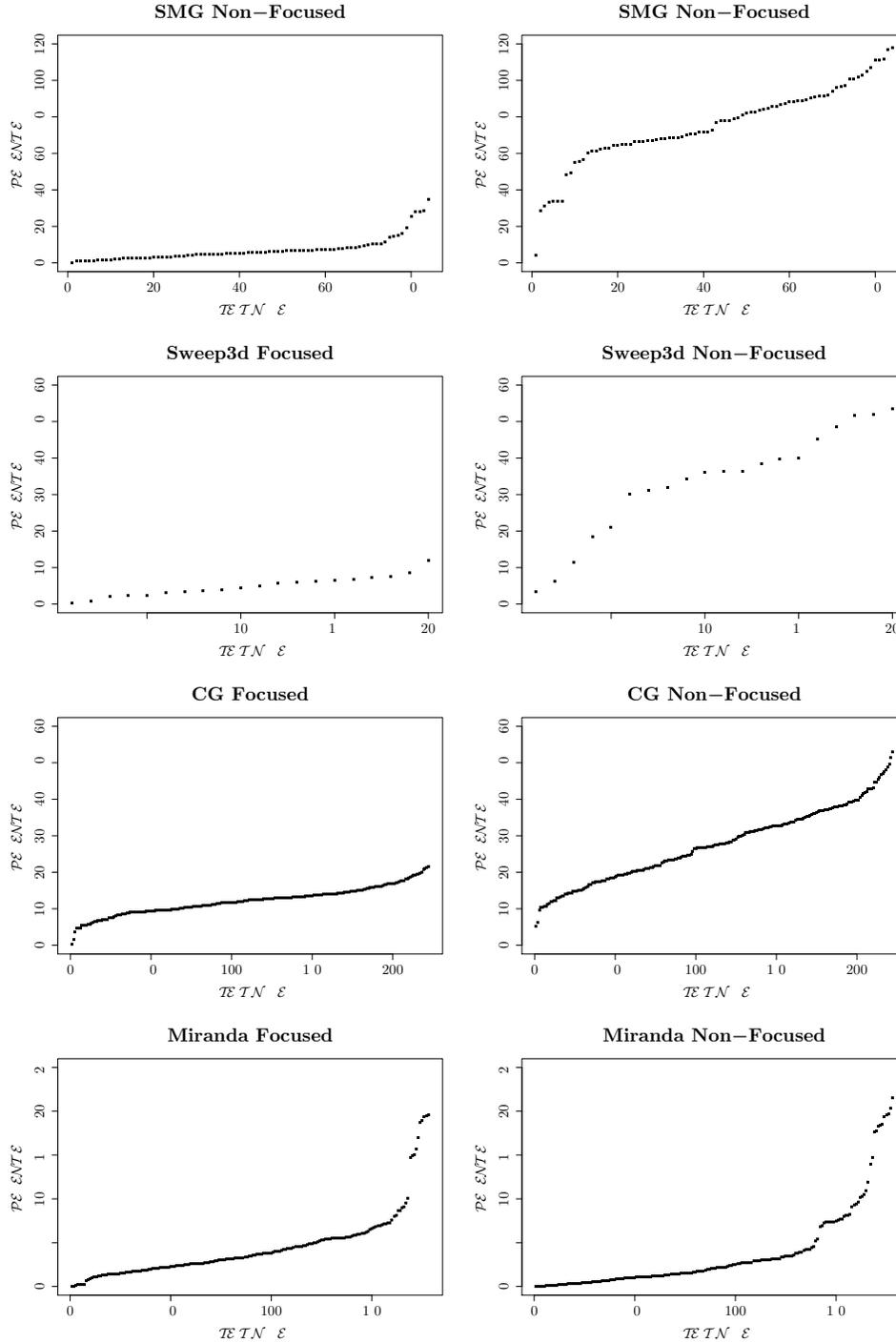


Figure 3. Scatterplots showing prediction error for focused and non-focused regressions for SMG, Sweep3d, CG, and Miranda.

*Miranda*: Figure 3 shows the results from Miranda for both focused and non-focused regressions, and Table III summarizes this data. In this case, there are

only two processor grid dimensions that are varying, which cuts down the number of processor grids at 1024 processors substantially.

Prediction	SMG			Sweep3d			CG			Miranda		
Error (%)	Max	Avg	Median	Max	Avg	Median	Max	Avg	Median	Max	Avg	Median
Focused	34	7.1	5.6	12	4.9	5.0	22	12	12	20	3.7	2.2
Non-focused	117	75	76	53	33	36	53	27	27	21	3.7	3.2

Table III  
MAXIMUM, AVERAGE, AND MEDIAN PREDICTION ERROR IN SMG, SWEEP3D, CG, AND MIRANDA FOR FOCUSED AND NON-FOCUSED REGRESSIONS.

The data shows that prediction quality is quite good with *either* technique. 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.

## V. RELATED 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.

First, this work extends our previous work on predicting strong scaling used black-box predictions and regression [5]. Our work here is different in multiple ways: it uses focused regressions, is targeted to time-constrained scaling (which in many ways proves more difficult than strong scaling), and uses gray-box techniques.

Another approach uses machine learning to make predictions on multicore machines [35]. Also in a similar vein, Curtis-Maury et. al. predict the power-performance tradeoff on single multicore machines [11]. These are similar to our approach, but they are limited to single multi-core processors and does not address the multiprocessor or cluster cases.

The other work most closely related to ours uses regression to predict application performance across a range of input parameter values. This includes neural networks [15] and piecewise regression [19]. Neither performs extrapolation, which is our focus.

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 [39].

Lyon *et al.* use the theoretical approach of Taylor expansions to understand execution behavior, including scalability properties [21]. Combining static and dynamic analysis to predict performance on different architectures for different inputs offers greater possibilities for extrapolating across process counts than these other statistical methods [22]. Later work showed that the technique could locate performance bottlenecks [23]). In contrast, Our framework only requires relinking of the application with the PMPI library to gather data during training runs.

There are a variety of simulation- or trace-based approaches to performance modeling [31], [32], [17]. Although techniques could extrapolate those traces to larger numbers of processors, our approach to scaling predictions is more direct.

White-box approaches typically require detailed analysis of data structures and program constructs, such as loop nests [16]. Several other researchers have explored white-box scalability analysis approaches that provide algorithmic or architectural perspectives [13], [38], [25], [9], [30], [37]. 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.* use regression and explore separating computation and communication [7]. 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* [1], require code modifications. Our techniques at most use the MPI profiling interface for instrumentation, which only requires relinking the application.

Analytic modeling of parallel machines include LogP [10] and BSP [34]. Another approach that requires no user intervention to create a static cost model [4] has only been applied to simple programs and architectures.

Several tools trace or analyze MPI performance through the MPI profiling interface, including VampirTrace [24], svPablo [12], TAU/ParaProf [6], and Paraver [27]. These tools generally focus on providing

assistance in optimizing applications, particularly for very large processor counts [28]. We build on algorithms to capture the critical path in MPI programs that were developed to support optimization [14], [29].

## VI. SUMMARY

This paper 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. Only a small amount of application-level information need be provided by the scientist—specifically whether the input parameters are related and if a processor grid is used. Then, our approach 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 scale at which the scientist desires.

Future work is proceeding in several directions. First, we are investigating breaking computation and communication into smaller phases. In particular, different computation or communication phases may scale quite differently; the idea is analogous to dividing total time into computation and communication time—which improved prediction accuracy. The challenge is to ensure that phases are combined when their execution time is sufficiently small, to protect against variance that is more striking in small phases. Second, we are looking at more applications that have many input parameters with complex relationships. While our approach is effective for all applications in our set, we may yet find that different techniques are required to achieve accurate time-constrained scaling predictions on other applications. Finally, we are investigating how to further reduce the number of experiments needed at smaller scales through the use of a field of statistics called experimental design.

## 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] Argonne National Laboratory. Intrepid. [www.alcf.anl.gov/news/detail.php?id=122](http://www.alcf.anl.gov/news/detail.php?id=122).
- [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] B. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz. A regression-based approach to scalability prediction. In *International Conference on Supercomputing*, June 2008.
- [6] 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.
- [7] 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.
- [8] P. N. Brown, R. D. Falgout, and J. E. Jones. Semi-coarsening multigrid on distributed memory machines. *SIAM Journal on Scientific Computing*, 21(5):1823–1834, 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. Schauer, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, Nov. 1996.
- [11] 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*, pages 250–259, 2008.
- [12] 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.
- [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*, 1124:665–674, 1996.
- [18] Lawrence Livermore National Laboratory. Blue Gene/L. <http://www.research.ibm.com/bluegene/>.
- [19] 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.
- [20] Los Alamos National Laboratory. Roadrunner. <http://www.lanl.gov/roadrunner/>.
- [21] G. Lyon, R. Kacker, and A. Linz. A scalability test for parallel code. *Software — Practice and Experience*, 25(12):1299–1314, Dec. 1995.
- [22] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *SIGMETRICS 2004*, pages 2–13, June 2004.
- [23] G. Marin and J. Mellor-Crummey. Application insight through performance modeling. In *IEEE International Performance Computing and Communications Conference*, Apr 2007.
- [24] 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.
- [25] D. Nussbaum and A. Agarwal. Scalability of parallel machines. *Communications of the ACM*, 34(3):56–61, Mar. 1991.
- [26] 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.
- [27] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PAR-AVER: 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.
- [28] P. C. Roth and B. P. Miller. On-line automated performance diagnosis on thousands of processes. In *PPOPP*, pages 69–80, Mar 2006.
- [29] M. Schulz. Extracting critical path graphs from MPI applications. In *IEEE Cluster*, Sep 2005.
- [30] 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.
- [31] A. Snively, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *Supercomputing*, Nov. 2002.
- [32] 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.
- [33] The National Center for Computational Science/Oak Ridge National Laboratory. Jaguar. <http://www.nccs.gov/jaguar/>.
- [34] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.
- [35] 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*, pages 75–84, 2009.
- [36] Wikipedia. Scalability web page. <http://en.wikipedia.org/wiki/Scalability>.
- [37] 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.
- [38] P. H. Worley. The effect of time constraints on scaled speedup. *SIAM J. Sci. Stat. Computing*, 11(5):838–858, Sept. 1990.
- [39] L. T. Yang, X. Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Supercomputing*, 2005.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.