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

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

*Department of Computer Science, The University of Georgia

†Department of Statistics, The University of Georgia

‡Department of Computer Science, The University of Arizona

§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 must scale their applications to exploit these new clusters. *Time-constrained scaling*, which is often used, tries to hold total execution time constant while increasing the problem size along with the processor count. However, complex interactions between parameters, the processor count, and execution time complicate determining the input parameters that achieve this goal.**

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 [1], ANL's Intrepid [2], LANL's Roadrunner [3] and LLNL's BG/L [4]. 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* [5]. 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 complex 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 that uses a small amount of application-level information as input. We choose a small series of training runs, varied over different, smaller processor counts and then use *focused regression* to predict the input parameters that need to be used in order to achieve time-constrained scaling. The training runs always use a processor count no more than half of the target number; to reduce training time, iterative applications can be executed for just a few timesteps. The scientist (or compiler/run-time system) must indicate the number of input parameters, whether they represent the dimensions of the main data structure or are unrelated, and whether the processor grid is part of the parameterization. Our focused regression technique 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 [6], 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* [7], where one runs the *same* program instance, i.e., uses identical input parameters. Strong scaling is the most frequent type that appears in computer science literature. However, *time-constrained scaling* [5], in which the scientist attempts to keep total run time constant, is becoming more commonplace. This approach solves problems that were previously unexplored and is generally more intuitive from the scientist's perspective.

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

However, time-constrained scaling poses many difficulties. First, most scientists assume that the *data set size per processor* should be fixed as the processor count increases, which is usually referred to as *weak scaling* [7] and tries to keep computation time per processor constant. Due to communication time, though, weak scaling alone will not keep total execution time constant.

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

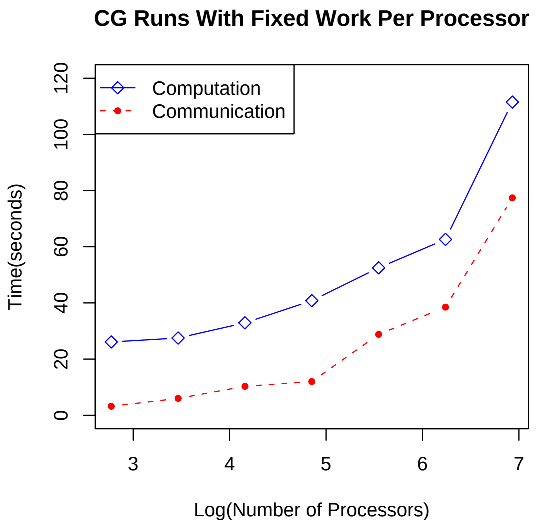

Figure 1. Computation and communication times for CG as the number of processors increases. The ratio of *SIZE/P* is fixed; *SIZE* ranges from 46,094 to 2,950,000, and P ranges from 16 to 1024. The value of *NONZER* is held constant.

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 such an obvious relationship between the parameters (e.g., CG from the NAS suite [8]).

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 does not require any program-level information. Finally, we discuss extensions that provide greater accuracy for more complicated applications.

A. Overall Technique

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

With the value of P_t and the input by the scientist, our technique proceeds as follows. For simplicity, we first present the case with only one input parameter. We assume that the scientist provides us with data from points with time $\approx T$, 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 the scientist provides. From this point, we use the techniques described in the next two subsections (basic and general regression models) to predict the value of the input parameter on P_t processors that will result in an execution time of $\approx T$. 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 computation time, denoted W is the other key characteristic for determining run time in programs with little communication. We can often easily determine W for simple programs from the input parameters (i.e., we can just use the product of the parameters (z_i's), or $W = f(z_1 \times z_2 \times \ldots z_n)$), and T is approximately proportional to W/P. More generally, we have

$$log_2(T) = \beta_0 + \beta_1 log_2(W) + \beta_2 log_2(P_t) + \epsilon \quad (1)$$

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

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.

Specifically, to collect the (T,W,Q) triplets, we execute the program on Q processors, where $Q \in \{2, \ldots, P_t/2\}$. We vary the values of W and Q on the sample runs and then use regression to generate Equation 1. Because it is easier to acquire Q processors than P_t, it is reasonable to perform multiple instrumented runs for different configurations of the input variables.

We predict run time using a log-scale because the prediction errors are well known to be proportional to the expected time—we are concerned with relative errors. Working in log-scale implicitly handles this. The base of the log makes no fundamental difference; we use log_2 in this paper for mathematical convenience. The coefficients β_1 and β_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 well for computation-dominated, simple-array based applications. The applications upon which we 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 [8]). All three have a high computation-to-communication ratio and a single input parameter.

C. General Model

For more complex applications, simply applying Equation 1 is insufficient for three reasons. First, we cannot easily determine computation time (W) in advance in some applications. Rather, we can specify the values of some input parameters in advance, and these parameters determine computation in an unknown or complex way. In such cases, one may need to examine several potential predictor parameters to determine which ones are significant predictors of time and to model the relationship between T and these variables. For example, CG has this characteristic, as indicated by Table I.

Second, modeling only total execution time produces inaccurate predictions for applications with a significant amount of time spent in communication. Computation and communication can scale at different rates, which the training runs capture only if we model them separately. As mentioned earlier, Figure 1 shows this situation. Currently, we do not subdivide either computation or communication further into phases because the prediction quality we achieve for our applications is usually accurate without it. We are currently investigating breaking computation and communication into smaller phases. For example, we could break communication calls into groups that have similar scaling behavior (e.g., logarithmic-scaling collectives versus linear-scaling collectives).

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

Our prototype handles each of these possibilities.

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

$$log_2(T) = \beta_0 + \beta_1 z_1 + \beta_2 z_2 + \ldots + \beta_n z_n + \beta_{n+1} log_2(P_t) + \epsilon \qquad (2)$$

Here, z_i is the i^{th} input parameter describing the data. We use additional training runs to determine which of the z_i are important in predicting T, as well as to model the functional form of these variables (similar to what was done by Lee et al. [9]).

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 regressing only on total time suffices with smaller percentages. We collect computation and communication time using the PMPI profiling layer of MPI.

Case 3: The most interesting case occurs when the application uses a processor grid. We considered simply extending Equation 2 by replacing the P_t term with terms for the processor grid terms (e.g., P_1 and P_2). However, while intuitive, experiments showed that this technique is ineffective because the data distribution, as specified by the processor grid, significantly affects application execution time in a nonlinear manner, as we show in Section IV. Thus, using a single regression results in significant errors.

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

We then use Equation 1 or Equation 2 in the focal region, depending on whether or not the input parameters are related, as described above. The typical strategy when creating regression models uses more data to achieve a better result. However, in our particular case, more data is worse, if it is not nearby in the processor dimension space on $Q < P_t$ processors. Also, while the focal region idea is quite useful and necessary when handling an application that uses a processor grid, it also improves regression quality for all applications. Therefore, we use the focal region idea in general—restrict tests to those around the values of the input parameters (adjusted for processor count) presented by the scientist. Using only a subset of available data for prediction via regression is not new; for example, Lee and Brooks use this technique for predicting performance and power [10]. We apply this technique to scalability analysis.

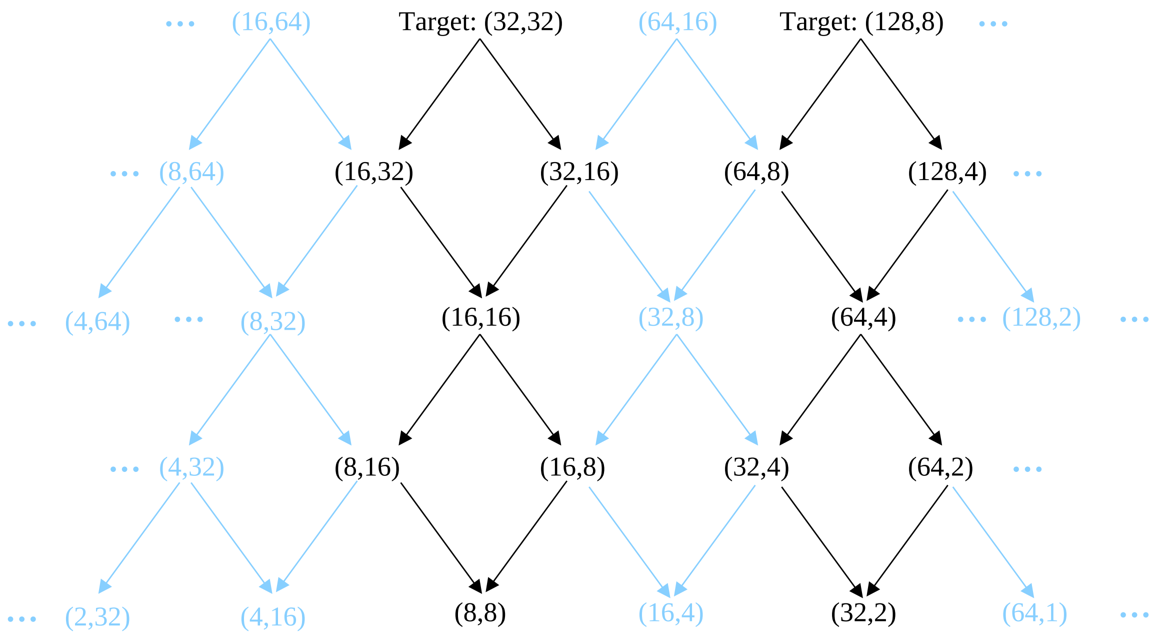

Figure 2. Processor grids (only shown down to 64 processors) used in SMG to predict $P_x = 1$, $P_y = 32$, and $P_z = 32$, and $P_x = 1$, $P_y = 128$, and $P_z = 8$, respectively. For all vertices in the graph, $P_x = 1$.

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

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

IV. Results

This section discusses results of our focused regression prototype in making time-constrained scaling predictions. For our evaluation we used two different clusters at Lawrence Livermore National Laboratory: the *Atlas* cluster and the *Hera* cluster. The former 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 [11]. 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, getting finer-grain results (e.g., omitting blocking time and collecting only network and copying time) requires instrumenting the entire MPI library. We then use measured execution times to fit a linear model. We use the statistical package SAS for all regressions. We emphasize that we run the program only on a small subset of the many possible input parameter/processor combinations; this choice conserves machine time as well as produces better results by using focal regions (as described in the previous section).

We make the important assumption that we can run an application with the input parameters set to values of our choosing. Essentially, the parameter space is quite large and sparse for applications like SMG (5 free parameters). This ability to execute the program in configurations of our choice ensures that we can collect the data that we need to make accurate predictions. Essentially, we assume that scientists write programs that are flexible and provide meaningful timing results, if not physical results, for any combination of the input parameters.

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

B. Applications

We tested our techniques using seven applications. Four are from the NAS suite [8]. In particular, we use BT, SP, CG, and LU; our approach does not apply to other NAS programs (FT, IS, MG) because we have insufficient input data due to input restrictions. Further, EP is trivial because it has only one parameter and zero communication. 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. We further use SMG and Sweep3d from the ASC suite; the former is a three-dimensional multigrid solver, and the latter is a three-dimensional neutron transport code. The final application is Miranda, which is an industrial-strength hydrodynamics application.

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.

C. Summary of Results

Overall, our prediction quality is quite good: median prediction error ranges from 3% to 12.2%, and predictions are almost always within 20% and usually much better. For the three more complex applications, we *must* generate different regressions for different focal regions to achieve accuracy. In particular, we obtain a median error as high as 75% if we do not use a focal region.

D. Single Parameter Programs

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

Table 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 to create a single monolithic regression. We study SMG first and in depth because it presents the most challenges.

SMG: SMG has six input parameters: three processor dimensions, p_x, p_y, and p_z, along with three grid dimensions, n_x, n_y, and n_z. The application specifies grid dimensions in terms of a per-processor local grid; one can recover the global grid by taking the product of each grid dimension with the associated processor dimension. For time-constrained scaling, four of the six input parameters are unconstrained, which still leaves many different ways to scale SMG. Note that SMG is not symmetric in all dimensions [12], 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, we must create a regression for different focal regions with SMG. Specifically, a different regression predicts each processor configuration. We used six of the possible processor configurations at 1024 processors for these results.

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

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

Figure 3 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. Figure 4 shows the effect of using our prototype to predict *SIZE*, as opposed to using naive weak scaling.

We also further investigated the naive time-constrained scaling prediction. However, the question is: how would the scientist scale CG to keep the execution time constant without our approach? As we mentioned earlier, CG has two parameters: *SIZE* and *NONZER*. The scientist has three intuitive potential choice to scale CGs: double *SIZE*, holding *NONZER* constant; double *NONZER*, holding *SIZE* constant; or increase each by $\sqrt{2}$. We ruled out the third case for two reasons. First, increasing both parameters by $\sqrt{2}$ seems physically unrealistic since CG is at its core a one-dimensional data structure (sparse matrix). Second, CG requires 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.

Miranda: Figure 3 shows the results from Miranda for both focused and non-focused regressions, and Table III summarizes this data. In this case, we vary only two processor grid dimensions, which substantially reduces the number of processor grids at 1024 processors.

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

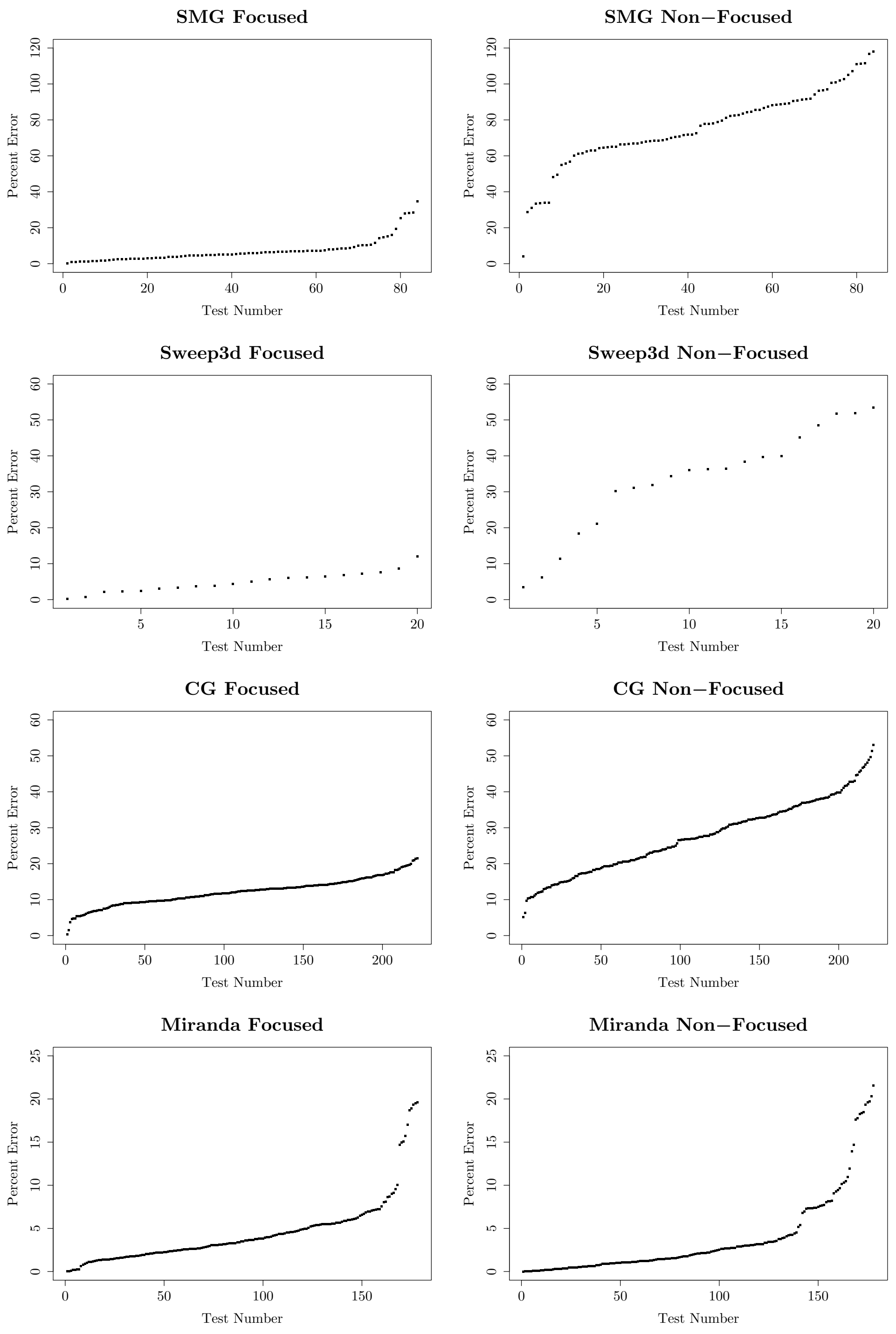

Figure 3. Scatterplots showing prediction error for focused and non-focused regressions for SMG, Sweep3d, CG, and Miranda.

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

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.

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

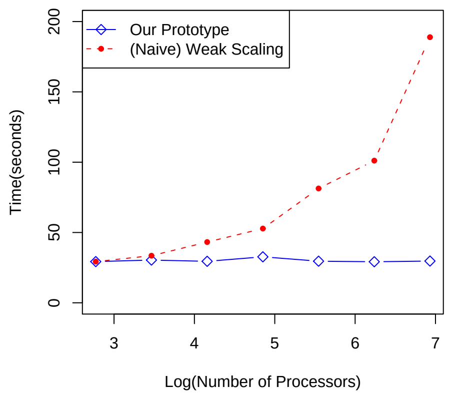

Figure 4. Time-constrained scaling with our prototype for 16 through 1024 processors (with *NONZER* fixed at 20). Our prototype predicts the value of *SIZE* at 1024 (given a set of experiments using 16 through 512 processors) that will match the time at 16 processors, which is 29.3 seconds. For completeness, we also show the predicted values for 32 through 512 processors. Clearly, our prototype leads to time-constrained scaling, while naive weak scaling does not. The right-hand side displays the results graphically.

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 using black-box predictions and regression [6]. Our work here is different in multiple ways: it uses focused regressions; it targets time-constrained scaling, which is more difficult than strong scaling in many ways; and it uses gray-box techniques.

Another approach uses machine learning to make predictions on multicore machines [13]. Also in a similar vein, Curtis-Maury et al. predict the power-performance tradeoff on multicore machines [14]. While similar to our approach, these approaches are limited to single multicore processors and do not address cluster systems.

The other, most closely related work uses regression to predict application performance across a range of input parameter values. Prior work here includes neural networks [15] and piecewise regression [9]. 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 [16]. Lyon et al. use the theoretical approach of Taylor expansions to understand execution behavior, including scalability properties [17]. 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 [18]. Later work showed that the technique could locate performance bottlenecks [19]). In contrast, our framework only requires relinking 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 [20], [21], [22]. Although techniques could extrapolate those traces to larger numbers of processors, we provide a more direct approach to scaling predictions.

White-box approaches typically require detailed analysis of data structures and program constructs, such as loop nests [23]. Several other researchers have explored white-box scalability analysis approaches that provide algorithmic or architectural perspectives [24], [25], [26], [27], [5], [28]. 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 [29]. 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* [30], require code modifications. Our techniques use the MPI profiling interface for instrumentation, which at most requires relinking the application.

Analytic modeling of parallel machines include LogP [31] and BSP [32]. Another approach that requires no user intervention to create a static cost model [33] has only been applied to simple programs and architectures.

Several tools trace or analyze MPI performance through the MPI profiling interface, including mpiP [34], Open|SpeedShop [35], VampirTrace [36], svPablo [37], TAU/ParaProf [38], and Paraver [39]. These tools generally focus on providing assistance in optimizing applications, particularly for very large processor counts [40]. We build on algorithms to capture the critical path in MPI programs hat were developed to support optimization [41], [42].

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

We are exploring several directions of future work. 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. This extension requires that we combine phases when their execution time is sufficiently small, to protect against variance that is more striking in small phases. 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 find that other applications require different techniques to achieve accurate time-constrained scaling predictions. Finally, we are investigating how to reduce the number of experiments needed at smaller scales further through experimental design.

Acknowledgements

This work was supported in part by the National Science Foundation under award number CNS-0936251. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the National Science Foundation. 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-PROC-422989).

References

[1] The National Center for Computational Science/Oak Ridge National Laboratory, "Jaguar," http://www.nccs.gov/jaguar/.

[2] Argonne National Laboratory, "Intrepid," www.alcf.anl.gov/news/detail.php?id=122.

[3] Los Alamos National Laboratory, "Roadrunner," http://www.lanl.gov/roadrunner/.

[4] Lawrence Livermore National Laboratory, "Blue Gene/L," http://www.research.ibm.com/bluegene/.

[5] J. P. Singh, J. L. Hennessy, and A. Gupta, "Scaling Parallel Programs for Multiprocessors: Methodology and Examples," *IEEE Computer*, vol. 26, no. 7, pp. 42–50, Jul. 1993.

[6] B. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. R. de Supinski, and M. Schulz, "A Regression-Based Approach to Scalability Prediction," in *International Conference on Supercomputing*, Jun. 2008.

[7] Wikipedia, "Scalability web page," http://en.wikipedia.org/wiki/Scalability.

[8] D. Bailey, J. Barton, T. Lasinski, and H. Simon, "The NAS Parallel Benchmarks," NASA Ames Research Center, RNR-91-002, Aug. 1991.

[9] 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*, 2007, pp. 249–258.

[10] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *ASPLOS*, 2006, pp. 185–194.

[11] 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.

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

[13] 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*, 2009, pp. 75–84.

[14] 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*, 2008, pp. 250–259.

[15] E. Ipek, B. R. de Supinski, M. Schulz, and S. A. McKee, "An Approach to Performance Prediction for Parallel Applications," in *Euro-Par*, Aug 2005, pp. 196–205.

[16] L. T. Yang, X. Ma, and F. Mueller, "Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution," in *Supercomputing*, 2005.

[17] G. Lyon, R. Kacker, and A. Linz, "A Scalability Test for Parallel Code," *Software — Practice and Experience*, vol. 25, no. 12, pp. 1299–1314, Dec. 1995.

[18] G. Marin and J. Mellor-Crummey, "Cross-Architecture Performance Predictions for Scientific Applications Using Parameterized Models," in *SIGMETRICS 2004*, June 2004, pp. 2–13.

[19] ——, "Application Insight Through Performance Modeling," in *IEEE International Performance Computing and Communications Conference*, Apr 2007.

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

[21] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 1994, pp. 196–205. [Online]. Available: ftp://gatekeeper.dec.com/pub/DEC/WRL/research-reports/WRL-TR-94.2.ps

[22] J. Labarta, S. Girona, V. Pillet, and T. Cortes, "DiP: A Parallel Program Development Environment," *Lecture Notes in Computer Science*, vol. 1124, pp. 665–674, 1996.

[23] 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.

[24] J. L. Gustafson, "Reevaluating Amdahl's Law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, May 1988. [Online]. Available: http://www.acm.org/pubs/toc/Abstracts/0001-0782/42415.html

[25] P. H. Worley, "The Effect of Time Constraints on Scaled Speedup," *SIAM J. Sci. Stat. Computing*, vol. 11, no. 5, pp. 838–858, Sep. 1990.

[26] D. Nussbaum and A. Agarwal, "Scalability of Parallel Machines," *Communications of the ACM*, vol. 34, no. 3, pp. 56–61, Mar. 1991.

[27] 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*, 1992, pp. 387–396.

[28] 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.

[29] J. Brehm, P. H. Worley, and M. Madhukar, "Performance Modeling for SPMD Message-Passing Programs," *Concurrency: Practice and Experience*, vol. 10, no. 5, pp. 333–357, Apr. 1998.

[30] S. R. Alam and J. S. Vetter, "Hierarchical Model Validation of Symbolic Performance Models of Scientific Applications," in *Euro-Par*, Aug. 2006.

[31] 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*, vol. 39, no. 11, pp. 78–85, Nov. 1996.

[32] L. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[33] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, "A Static Performance Estimator to Guide Data Partitioning Decisions," in *Proceedings of the*

Third ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, Apr. 1991, pp. 213–223.

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

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

[36] 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*, Sep. 2007.

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

[38] 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)*, Aug. 2003, pp. 17–26.

[39] 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*, ser. Transputer and Occam Engineering, vol. 44, Apr. 1995, pp. 17–31.

[40] P. C. Roth and B. P. Miller, "On-Line Automated Performance Diagnosis on Thousands of Processes," in *PPOPP*, Mar 2006, pp. 69–80.

[41] J. K. Hollingsworth, "Critical Path Profiling of Message Passing and Shared-Memory Programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 29–40, 1998. [Online]. Available: citeseer.ist.psu.edu/hollingsworth98critical.html

[42] M. Schulz, "Extracting Critical Path Graphs from MPI Applications," in *IEEE Cluster*, Sep 2005.