ScoPred—Scalable User-Directed Performance Prediction Using Complexity Modeling and Historical Data

Benjamin J. Lafreniere and Angela C. Sodan
University of Windsor, Computer Science

Abstract

Using historical information to predict future runs of parallel jobs has shown to be valuable in job scheduling. Trends toward more flexible job-scheduling techniques such as adaptive resource allocation, and toward the expansion of scheduling to grids, make runtime predictions even more important. We present a technique of employing both a user's knowledge of his/her parallel application and historical application-run data, synthesizing them to derive accurate and scalable predictions for future runs. These scalable predictions apply to runtime characteristics for different numbers of nodes (processor scalability) and different problem sizes (problem-size scalability). We employ multiple linear regression and show that for decently accurate complexity models, good prediction accuracy can be obtained.

1 Introduction

The typical approach in parallel job scheduling is that users provide estimates about the runtimes of their jobs with such estimates being, in the general case, much higher than the actual runtime (ranging from 20% [12] to 16 times [11] higher in different studies for different supercomputing centers). The availability of more accurate information about runtimes of parallel programs has been shown to be valuable to improve average response times. However, results also exist suggesting that the overestimation of runtimes provides some benefits by creating holes in the schedule that can be filled with short jobs [10]. Thus, the need for accurate estimates in standard job scheduling is not yet fully decided. However, in the context of grid scheduling for simultaneous execution of jobs on multiple sites, reservations of resources on remote sites is required and prediction is unequivocally relevant. The simplest approach to obtain accurate estimates is recording runtimes of previous job executions and using them to predict future runtimes with the same job configuration and the same number of resources. For grid computing, more detailed runtime information may be needed to estimate performance on different systems. Furthermore, grid middleware may be composed of different components, and a more detailed recording of these components (and their performance for certain applications and machines) are needed to support optimal configuration of grid jobs [16].

To make matters more complicated, modern job scheduling employs advanced approaches such as flexible time sharing and adaptive resource allocation [6]. Flexible time sharing means relaxing global synchronous gang scheduling if jobs can be matched well [8][4] or could mean abandoning global control altogether [14]. In both cases the aim is improving resource utilization. Adaptive resource allocation means that the number of nodes allocated to a job changes during its runtime, typically driven by the changing workload of the machine [7][2]. Time sharing makes it necessary to know more detailed application characteristics such as the fractions of total runtime spent on communication and I/O, to allow us to find proper matches and estimate slowdowns. For adaptive resource allocation, it becomes relevant to estimate runtime on different numbers of resources; if we can predict resource times with different numbers of nodes, we can better predict the benefits of certain adaptation decisions. This leads to the challenge of generating scalable predictions, i.e. predicting resource times with allocations other than those previously measured. Such scalable prediction is also useful if the user moves to larger problem sizes for which no performance data is available yet.

Several approaches exist which utilize special compilers to provide performance models of applications. A difficulty of this approach is that abstract models may not always be extractable in a fully automated manner, particularly if the code behavior is complex. As well, runtime measurements or simulations are typically needed for quantification of the parameters. Our target is a standard job-execution environment running primarily MPI based applications, and not equipped with any such special compilers. However, we assume that users have some rough knowledge of their applications and know, for example, which parameters in their application

determine runtime. Furthermore, users may be able to provide rough cost estimations (closely resembling the steps needed to derive complexity estimates) as suggested in [15]. Whether estimated formulas for performance models are provided by the user or extracted by a special compiler, the system must then perform quantification of coefficients within the formula, to turn the rough estimate into a concrete model for the system being used. This challenge motivates the following goals for our ScoPred performance predictor:

- Support prediction of overall execution times as well as prediction of resource times (computation, communication, I/O)
- Support such prediction on the same and on different numbers of nodes (scaling the prediction)
- Assume that a rough model is available as input such as via cost/complexity estimations from the user (user-directed) or a formula provided by the compiler (compiler-directed) and leave all quantification (determination of coefficients) to the system
- Provide a practically feasible approach
- Provide a mathematically sound approach

In our ScoPred approach, we apply the following innovative solutions toward meeting these goals:

- Take the performance-determining parameters and a rough resource-usage model as input
- Employ multiple linear regression to determine the coefficients and provide mean values, confidence intervals, and prediction intervals

2 Related Work

Performance data bases or repositories were first proposed in [9], Due to a lack of models describing the application behavior, measurements such as runtime or memory usage were used to interpolate the runtime of future runs from previous measurements, i.e. no scaling was applied. The approach is extended in [13] to consider a number of additional criteria such as the application parameters. The difficulty addressed by considering the different criteria is to associate the runtimes with a particular application and the correct instance of that application. This can become difficult as a single user may run the same application with different parameters and under different names [9][13].

Furthermore, to estimate the cost under varying resource allocation, a proper cost model needs to consider the fact that speedup curves are not linear but that the curve typically flattens (and finally declines) if more resources are allocated. In [5], a general statistical model is proposed which is useful for studying general adaptation benefits in simulation studies. This, however, does not help us to find an application's specific cost model. The Grads project [17] employs performance models created by the compiler to predict performance and monitors whether these predictions are met. The approach in [19] considers floating-point operations and accesses to the memory hierarchy to predict performance on different architectures, while focusing on the memory accesses. Memory accesses are extracted via a simulator, and application characteristics are convolved with machine characteristics. In [18], a prediction model for different architectures is described which extracts the program structure by static analysis of the program binary, after which execution profiling is used to obtain dependence on parameters such as execution frequency for vertexes in the program graph and reuse distances for memory locations. Linear regression is applied, but only used for the memory-access cost, and the approach currently only considers single-CPU performance. Scalability is not considered.

The most closely related approach is presented in [15]. This approach takes cost/complexity estimations as input, while differentiating explicitly different sources of overhead such as communication, load imbalance, or synchronization loss. Coefficients are then obtained from actual program runs.

Both [18] and [15], consider linear combinations of cost terms. In [15], either additive or multiplicative combinations are possible, the latter describing interaction between cost terms. Predictions are compared to actual runtimes, with the average relative error found for 2D FFT to be only 12.5%, whereas simple linear interpolation (without interactions) resulted into an error of 750%. The approach considers different parameters and employs a least-squares method but does not provide confidence or prediction intervals.

3 The Prediction System

3.1 Overall Framework

The ScoPred performance predictor is embedded into a job-scheduling and job-control framework as shown in Figure 1. Several tools interact via an integrated design. The Dynamic Directory stores information about the current job execution, including user-provided estimates, and retrieves information about previous runs from a performance repository. The ScoPro monitor [1] provides information about the characteristics of jobs such as the fraction of computation, communication, and I/O time as well as slowdowns under coscheduling. Furthermore, it can monitor progress on heterogeneous resources. ScoPro employs dynamic instrumentation and can, for loosely synchronous applications, extract behavior from a few iterations. The adaptation controller interacts with the

application, determining potentially new workload targets per node and initiating the adaptation. In [3], we have presented an approach to perform such adaptation via overpartitioning or partitioning from scratch in either the time or space dimension.

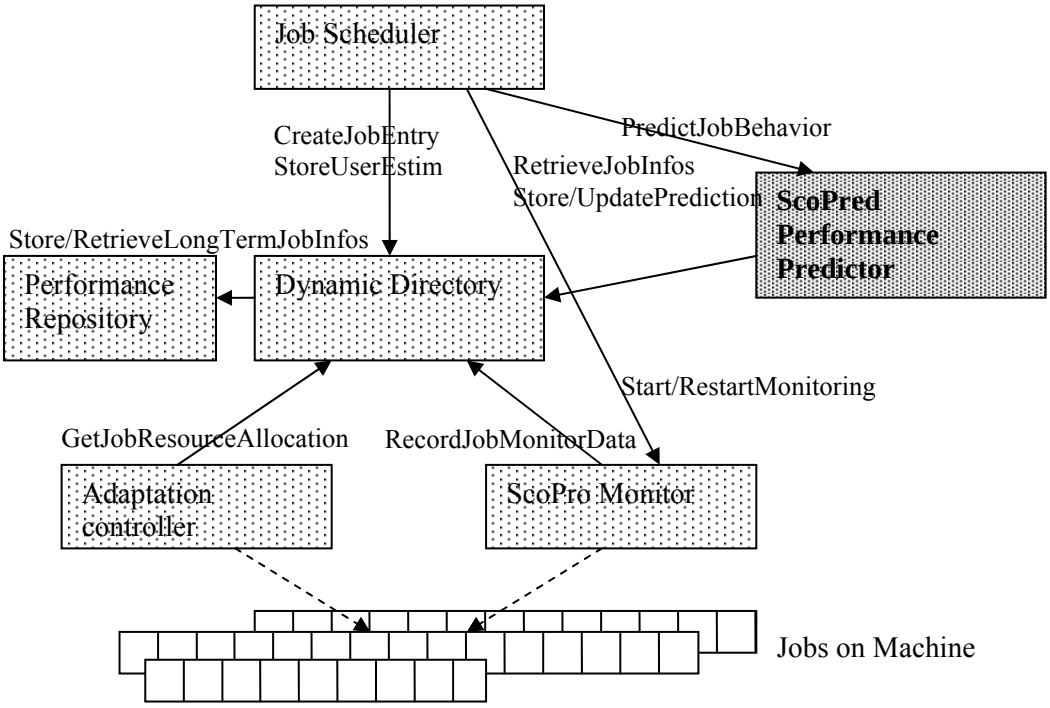

Figure 1. Overall integrated framework for job scheduling and job control. Solid arrows indicate invocations and a labeled with the corresponding function.

3.2 Application Model

We apply the view that application performance is dominated by certain data structures or computations that depend on a few critical parameters, such as the size of an array. Often these parameters determine the problem size of the application and changing the parameters changes the problem size. Then, it would be possible to model application performance by modeling the dependence of the cost on these parameters. In many cases, an application's user is aware of which application parameters are critical, as the user may explicitly want to change them to switch to a different problem size. In this case, the user should be able to specify which these parameters are.

Typically such parameters appear as input to the application but they may also be statically compiled into the code. We assume in the following that the values of the critical parameters are specified with the submission of the job (it would not be difficult to build a corresponding programming environment that makes such submissions straightforward). Furthermore, we assume that a unique identification is attached to the application (the name is not necessarily sufficient and may be different for program versions compiled for different numbers of nodes). We do not make any efforts to automatically match jobs as done in [9][13], but instead focus on the prediction aspect.

We assume that the user, in addition to the specification of the critical parameters, also provides a rough cost estimate describing the qualitative/structural relationship (without the quantification of system dependent coefficients). Such estimates could be close to the estimations represented by mathematically derived algorithmic complexity estimates. Importantly, complexity-oriented estimations would focus on the complexity of the model expressed in various terms such as $T = 2\ M^3 + log(N) + log(N) * M$ with T being the runtime, M being the size of one dimension in a matrix, and N being the number of nodes. Let us assume that the first term specifies computation cost in terms of computation steps and the second and third term specify communication cost in terms of sizes and numbers of messages. Note that the coefficients from

the replication in iteration steps as well as the computation time per step, message startup cost and transfer cost are omitted. These are the coefficients to be determined by the predictor. We assume that the estimate takes the form of a linear combination of different cost terms. However, as will be explained below, variables may be multiplied within the terms.

The idea is to specify a rough cost formula which only reflects critical, i.e. dominating performance influences. The system should provide the possibility to add an automatic correction to approximate missing cost terms which are not crucial but also not negligible if high prediction accuracy is to be obtained. What is exactly to be modeled depends on the application and on the desirable range of prediction. Thus, if a small problem size fits into the cache and a larger one does not, and this significantly influences performance, the memory access cost in dependence to the parameters should be modeled. Similarly, if load imbalance could become an issue, this should also be modeled, etc. Note that an alternative to the user providing specifications is the compiler providing them. In this case, the derived formulas may be more detailed though the compiler might require monitoring / runtime feedback or a simulator to prune irrelevant details and extract the relevant formulas.

Our current prediction system works for a particular machine, i.e. we do not include any machine model and prediction facilities across machines.

3.3 Architecture and Functionality

In the following, we describe the details of the predictor functionality. The ScoPred predictor takes as input:

- Specifications of the critical parameters and the estimation formulas depending on them
- The actual values of the critical parameters for the current job submission
- Information about previous runs of the same application (parameters values and performance of the job run)

Note that the number of nodes on which the application is run is typically one of the critical parameters (unless the application always runs with the same number of nodes).

The values predicted may be:

- Runtime of the whole job
- Differentiated time consumed on different resources such as computation time (CPU), communication time (network), and I/O time (disk).

The latter requires that the monitor provides the actual times for the different program execution components from previous runs to match them against the prediction. Providing cost estimations for different resources is typically not a problem for the user because these different aspects need to be considered (as far as applicable) to obtain an estimation of the runtime function for the whole job.

The goal of the estimation may be to predict

1. Future runtimes with the same parameter values
2. Future runtimes with different problem sizes (one or several of the parameters other than the number of nodes)
3. Future runtimes with a smaller or higher degree of parallelism (smaller or larger number of nodes)
4. Combinations of different problem sizes and a different degree of parallelism

Option 3 can be applied if resources are allocated adaptively (as for malleable or moldable applications), and is thus of particular interest. Option 4 is a typical case, as for production runs (runs other than tests for speedup graphs with varying numbers of nodes) there is typically a correlation between problem size and number of nodes, i.e. larger problem sizes require a larger number of nodes and vice versa. Note that this option is always at least a two-variable prediction, whereas the others potentially predict a single variable only. Options 2, 3, and 4 require the scalability features of our predictor.

The prediction system provides point estimates of the mean of values, and associated confidence and prediction intervals.

4 Multiple Linear Regression and Its Application in ScoPred

4.1 Overview of Multiple Linear Regression

Linear regression, also known as *linear least squares regression*, is a widely used mathematical modeling technique [20]. Given a data set and an appropriate function, least squares regression determines the values for coefficients within the function that produce an equation which best fits the data set. Simple linear regression applies to equations with a single dependent variable, and a single independent variable. Multiple linear regression applies to equations with a single dependent variable and multiple independent variables.

In order to be appropriate for linear regression, the supplied function must be *linear in the parameters* [21]. This is satisfied if and only if the function is of the form [20]:

$$f(\vec{x}:\vec{B}) = B_0 + B_1x_1 + B_2x_2 + \ldots + B_kx_k \quad (1)$$

where:

- Each independent variable (x_1, x_2, ..., x_k) in the function is multiplied by an unknown coefficient (B_1, B_2, ..., B_k).
- There is at most one unknown coefficient with no corresponding independent variable (B_0).
- The individual terms are summed to produce a final function value.

The independent variables may be the product of several application parameters, and particular application parameters may be components of more than one independent variable, though no two independent variable may be exactly the same. For instance,

$$y_i = B_0 + B_1 sqrt(x) + B_2 x^2 + B_3 xz + B_4 z \qquad (2)$$

is a valid function.

Given a function which satisfies the above conditions, we can relate the function to the data set by adding an error component, ε:

$$f(\vec{x} : \vec{B}) = B_0 + B_1 x_1 + B_2 x_2 + ... + B_k x_k + \varepsilon \qquad (3)$$

In all but perfect models, the value of ε will vary for each observation. The total set of ε values is referred to as the *error component*, or *set of residuals*. Three assumptions must hold regarding error component [22].

The set of values of ε must:

- Be independent, in the probabilistic sense (the value of a particular ε value should be unrelated to the other values of ε, except insofar as they satisfy these conditions)
- Have a mean of 0 and a common variance
- Have a normal probability distribution

Given a suitable function and data set, we can calculate the values for the coefficients such that the sum of the squares of the residual values is minimized. That is, we choose B_0, B_1, ... B_k such that

$$SSE = \sum_{i=1}^{k} (y_i - \hat{y}_i)^2$$

is minimized, where y_i refers to the i^{th} observed response value, and $\hat{y}_i$ refers to the value of (1) with the i^{th} observations of all dependent variables.

To find values for the coefficients which minimize SSE, partial differentials of (1) are taken with respect to each unknown coefficient. The resulting set of partial differential equations is then solved as a system of linear equations. For further details, refer to [21][20].

4.2 Statistical Tests and Metrics

There are several ways of evaluating how well a calculated regression equation fits the data. Two commonly used measures of a fit's significance are the coefficient of determination, and the analysis of variance F-test [22].

The coefficient of determination (often called R^2 or *multiple* R^2) [22] expresses the proportion of the variance that is explained by the regression model. Informally, this is a measure of how much better the model is at expressing the relationship, as compared to simply using the mean of the response data. An R^2 value of 0 indicates that the mean of the response data is a better predictor of the response, whereas an R^2 value of 1.0 indicates that the response fits the curve perfectly, explaining away 100% of the deviation from the mean in each response value. For convenience, we consider any value less than 0 to be equivalent to 0, and this allows us to interpret the coefficient of determination as a percentage. A value related to the coefficient of determination is the *adjusted* R^2, or *shrunken* R^2. As the number of independent variables in a regression formula rises, the R^2 value becomes artificially inflated. To compensate for this effect, the *adjusted* R^2 value takes into account the number of independent variables [22].

Another way of testing whether a regression model is significant is through a statistical test. For any given configuration of coefficient values we can test whether the data supports that configuration at a given confidence level. If the regression model is significant, then at least one of the independent variables is contributing significant information for the prediction of the response variable [22]. To prove this, we attempt to reject the contrapositive statement that no independent variable contributes any information for the prediction of the response. This is referred to as the null hypothesis, and we attempt to reject it with 95% confidence. If successful, we have shown with 95% confidence that at least one of the independent variables is contributing significant information for the prediction of the response variable. That is, the regression model is significant.

4.3 Using a Regression Model for Prediction

Once the regression model is generated and we have verified that the model effectively describes the relationship between the predictor variables and the response, the model can be used to predict the response for given values of the independent variables. These predictions take the form of a point estimate which is a prediction of the mean function value for a certain parameter-value combination. In addition, the confidence interval (typically 95%) is provided, describing the probability with which the mean value of future observations for this parameter-value combination falls into the corresponding interval.

Furthermore, a prediction interval is provided which describes the probability that future observed function values fall into the corresponding interval. Note that both the confidence and the prediction interval become wider as the parameter-value combinations are further away from the means of the observed values of the independent variables used to build the regression model, i.e. the prediction becomes less reliable.

4.4 Multiple Linear Regression in Parallel Performance Prediction

Given the above explanations, we can now explain how a scalable model of an application can be built through a straightforward application of linear regression. Given an application that we wish to model, we proceed as follows. To start, the static (static variables) or dynamic parameters (arguments of the program invocation) of an application, which are specified to influence the application's characteristics, are considered as components of the independent variables of our model. Similarly, application characteristics that we wish to construct a model of (runtime, I/O time, etc) are considered as the response variables. By storing the parameters with which application runs are submitted, and the corresponding observations of the characteristic under study, we build a data set of our independent and response variables.

At this point we have a data set, but we still need a function to fit to the data. As described above, the function is supplied by the user (or by a compiler) as an estimation formula (see Section 3.2,) of the relation between the characteristic under study and the application parameters. However, it is still lacking a quantification of the coefficients. Coefficients (that unknown at this point) are automatically added into the equation by our system in a manner that ensures the resulting equation is linear in the parameters. Once we have added the unknown coefficients, we have an equation of the form described in Section 4.1. We can now use linear regression to determine the quantification by calculating values for the coefficient variables that best fit the equation to the data, giving us a model of the application.

By substituting in the parameters, the model can be used to predict the performance of the characteristic under study for a future run of the application. We use statistical measures such as the coefficient of determination, R^2 and the analysis of variance F-test to determine how successful the regression model fits the data, and to calculate confidence intervals for our predictions.

Since fitting the function values into a regression model always involves some inaccuracy regarding the individual values for certain parameter-value combinations (points), employing the prediction from the model for points with available observations is not very meaningful. We can gain better predictions by only considering the different historical values for exactly the corresponding point. The main benefit from the regression model comes from predicting performance for new parameter-value combinations.

4.5 Implementation

We have used Java to write the overall interface for our performance predictor. This interface handles the interaction with the job scheduler and the dynamic directory in regards to storage of new or modified application profiles and retrieval of existing profiles. All statistical calculations are performed through calls to the Waterloo Maple symbolic algebra system [27] using custom functions programmed in Maple's native language. To allow the Java components access to the Maple components, we used the OpenMaple API for Java [28], which allows us to call code in a running Maple environment from external Java programs. The calls to the Maple environment to calculate a model, or make a prediction based on an existing model take in the order of a second, which is acceptable in a job-scheduling environment for parallel machines.

5 Experimental Evaluation

5.1 Experimental Setup

To evaluate our system, we have chosen the Linpack benchmark [29] and two applications from the NAS Parallel Benchmarks Version 2.4 [23]. The selected NAS benchmarks are the EP embarrassingly parallel benchmark, and the FT 3D Fast Fourier transformation benchmark.

Some of the tests were run on a local 16 node Debian GNU/Linux cluster running Linux kernel 2.6.6. Each node of this cluster is equipped with 2 Intel Xeon 2.0Ghz processors, of which our tests only use one. Other tests requiring more than 16 nodes were run on 64 nodes of a Redhat GNU/Linux cluster running Linux kernel 2.6.8.1. Each node of this cluster is equipped with 2 Intel Opteron/244 1.8Ghz processors, of which our tests only use one. Each cluster uses a Myrinet network for application-level communication, using MPIch version 1.2.6 over GM.

Each benchmark was run for varying problem sizes and different numbers of nodes (NAS provides a number of different categories: S, W, A, B, etc. which have increasing problem sizes, and Linpack allows problem sizes and node configurations to be specified). For each configuration of problem size/nodes, each benchmark was run 3 times, unless specified otherwise. For the subset of gathered data selected for each test, we feed all data into our predictor but for the sake of clarity only show the mean values in the tables below.

After providing a subset of the gathered data to our predictor, predictions are made for the excluded problem size/nodes configurations.

We test the following cases:

- Processor Scalability: Prediction towards larger number of nodes given data for a smaller number of nodes (malleability test or test for user choosing new number of nodes)
- Problem-size Scalability: Prediction towards larger problem sizes given data for smaller problem sizes (user switching to larger problem size)
- Problem-size/Processor Scalability: Prediction towards larger problem sizes and larger number of processors given data from situations with data for smaller numbers of nodes/problem size (user switching to larger problem size on larger number of nodes)

Whenever possible, we omit observations with runtimes of less than one second and in some of our experiments below this limits the number of experiments we can perform on the gathered data sets.

5.2 EP

The EP benchmark represents the simplest of parallel programs. Communication occurs only twice: once when the job begins, sending a segment of the total work to each of the nodes; and once right before the job terminates, collecting back the results of each node's calculations. The lack of communication time makes this benchmark very simple to model, and we use it as a demonstration of our technique.

We ran the EP benchmark on the local 16 node cluster, varying the number of nodes from 2 to 16 in increments of 2. The mean values of the three gathered observations for each configuration are shown in Table 1.

Before this data can be entered into our system, we must provide an estimate of how the benchmark's runtime varies with its parameters. To do so, we consider the benchmark's algorithm. The EP benchmark accepts two parameters, the number of nodes (P), and the problem size (N). Each node is assigned the task of generating (N/P) pairs of Gaussian random deviates according to a specific scheme, and tabulates the number of pairs in successive annuli [23]. Since the time to generate a pair of random numbers is relatively constant, we estimate that the runtime (T) of a particular job will be given by:

$$T = N / P$$

This estimate is entered into our system. As the first step toward turning this estimate into a detailed model, the system adds unknown coefficients into the equation, giving us:

$$T = B_1 (N / P) + B_0$$

where B_1 and B_0 represent the unknown constants added by our system.

Table 1. Mean values for EP benchmark observations.

nodes \ class	S = 2^{24}	W = 2^{25}	A = 2^{28}	B = 2^{30}
2	2.10	4.19	33.60	138.24
4	1.09	2.10	16.81	67.28
6	0.72	1.46	11.26	44.82
8	0.54	1.09	8.40	34.86
10	0.43	0.87	6.74	29.03
12	0.36	0.71	5.62	22.47
14	0.31	0.62	5.17	20.11
16	0.28	0.56	4.36	17.38

.

In the first experiment, we test our system for processor scalability, using data from the A and B columns. Table 2 shows the results of using our system to model either the A or B column, i.e. the table shows the result of two separate sets of processor-scalability predictions. In each column, a subset of the observations is input into the system (shown in light grey). The cells highlighted in darker grey are the predictions of our system given the input data for that column. For each value we provide a point estimate, followed by the 95% confidence interval, and the 95% prediction interval (in parenthesis). Below the point estimate and confidence/prediction intervals is the percentage that the estimate deviates from the mean of the observations.

For the A column test outlined in Table 2, the system assigns a value of 1/3996971.35 for B_1 and a value of 0.025 for B_0, giving the model:

$$T = (N / 3996971.35\ P) + 0.025$$

For the B column, the values of B_0 and B_1 are similar, giving us the model:

$$T = (N / 3906469.72\ P) + 0.085$$

We observe in Table 2 that our predictions tend to be a bit low, though all predictions are well within a 10% deviation from the mean of the observations. While our predictions for Class A come very close, the mean of the observations does not fall within the 95% confidence or prediction intervals for the 14 and 16 node predictions. For Class B, the mean of the observations is within both the 95% confidence and prediction intervals for all predictions. This does not necessarily indicate that the predictions for Class B are better as we observe that the confidence and prediction intervals for Class B are considerably wider. This is likely due to greater variation observed between data points of the same configuration for Class B.

Table 2. Results of processor-scalability predictions for 12, 14, and 16 node cases in the EP benchmark.

nodes \ class	A = 2^{28}	B = 2^{30}
2	33.60	138.24
4	16.81	67.28
6	11.26	44.82
8	8.40	34.86
10	6.74	29.03
12	5.62 +/- 0.03 (0.08)	22.99 +/- 2.77 (8.09)
	- 0.06%	+ 2.31%
14	4.82 +/- 0.03 (0.08)	19.72 +/- 2.89 (8.13)
	- 6.71%	- 1.96%
16	4.22 +/- 0.03 (0.08)	17.26 +/- 2.98 (8.16)
	- 3.21%	- 0.73%

Table 3. Results of problem-size scalability prediction of the B problem size for the EP benchmark.

nodes \ class	S = 2^{24}	W = 2^{25}	A = 2^{28}	B = 2^{30}
2	2.10	4.19	33.60	134.43 +/- 0.18 (0.19)
				- 2.76%
4	1.09	2.10	16.81	67.17 +/- 0.27 (0.29)
				- 0.17%
6	0.72	1.45	11.26	44.92 +/- 0.34 (0.36)
				+ 0.22%
8	0.55	1.09	8.40	33.49 +/- 0.28 (0.30)
				- 3.94%

In our second experiment, we test our system with the task of problem-size scalability, modeling the 2, 4, 6, and 8-processor rows respectively. We provide the data for problem sizes S, W, and A for each processor row, and use the generated model to predict problem size B for that row. The results are shown in Table 3.

We see that our predictions are very good, with all predictions less than 5% from the mean of the actual observations. Also, in all but the 2 processor case, the mean of the actual observation is within the 95% confidence and prediction intervals.

In our final experiment for the EP benchmark, our system is tested with the task of modeling both problem size and number of processors simultaneously. We provide the system with columns S, W, and A and rows 2, 4, 6, 8, and 10, and use the generated model to predict A on 12, 14, and16 processors, and B on 8, 10, 12, 14, and 16 processors. The results are shown in Table 4.

The multivariate model created by entering observations varying in both number of nodes and problem-size is almost as successful as the models of one or the other alone. All values are within 6% of the observed mean, and none of the observed means falls outside of the 95% prediction interval (only one falls outside the 95% confidence interval).

Finally, we want to note that the R^2 values for these predictions are close to 1 and that the F tests are passed at a greater than 95% confidence level in all cases.

Table 4. Results of problem-size/processor scalability predictions for the EP benchmark.

nodes \ class	**S = 2^{24}**	**W = 2^{25}**	**A = 2^{28}**	**B = 2^{30}**
2	2.10	4.19	33.60	
4	1.09		16.81	
6	0.72	1.45	11.26	
8	0.55	1.09	8.40	34.39 +/- 0.61 (3.95) - 1.36 %
10	0.43	0.87	6.74	27.51 +/- 0.59 (3.95) - 5.23 %
12			5.73 +/- 0.63 (3.95) + 1.90 %	22.93 +/- 0.58 (3.94) + 2.05 %
14			4.91 +/- 0.64 (3.95) - 4.97 %	19.65 +/- 0.58 (3.94) - 2.30 %
16			4.29 +/- 0.64 (3.95) - 1.61 %	17.19 +/- 0.59 (3.94) - 1.13 %

5.3 FT

The FT benchmark solves a partial differential equation (PDE) using a 3-D Fast Fourier Transform (FFT) [23]. The 3-D FFT is solved using a standard transpose algorithm. Due to the nature of the algorithm, it can only be run on a power-of-two number of nodes. The FT benchmark is more complex to model than the EP benchmark, as there is extensive communication throughout the application run.

The algorithm employs a 3-dimensional array which determines the problem size. Though the size of the FFT array is set for each class of benchmark, the user may vary the number of *iterations* performed for each class. We have set all tests to 6 iterations to make the runs with different problem sizes comparable. In each iteration, a 3-D FFT is performed, and the resulting data set is evolved before being used as the input for the next iteration.

The FT benchmark was run for 2, 4, 8, 16, 32 and 64 nodes with problem sizes varying from Class S (64x64x64) to Class C (512x512x512). The mean values of the application runs are summarized in Table 5. We were unable to obtain any data points for two configurations, which are marked N/A. For most configurations, we gathered 6 data For the remaining configurations (marked with an ◄) we only gathered 3 data points.

Table 5. Mean values for FT benchmark observations.

nodes \ class	S = 64x64x64	W = 128x128x32	A = 256x256x128	B = 512x256x256	C=512x512x512
2	0.21	0.48	9.91	43.94	N/A
4	0.11	0.25	5.30	23.23	101.69 ◄
8	0.06	0.14	2.68	12.16	N/A
16	0.03	0.08	1.51	6.69	29.36 ◄
32	0.02	0.04	0.87	3.69	16.57
64	0.01 ◄	0.02 ◄	0.47 ◄	1.95 ◄	8.63 ◄

The time complexity of computing a 3D FFT using the transpose algorithm is well studied and easily found in many textbooks. In [24], the following formula is given in an analysis of the 3-D FFT algorithm.

$$T = (N/P)\log(N) + 2(\sqrt{P} - 1) + 2(N/P)$$

where T is the runtime, N is the problem size, and P is the number of processors. However, an inspection of the code for the FT benchmark reveals that it utilizes all-to-all communications within processor subgroups rather than the point-to-point communications used by the algorithm analyzed in [24]. This motivates us to form an alternate estimate that assumes that scatter/gather is being used for the implementation of all-to-all. Based on the analysis of CPU time given in [24] and the complexity of scatter/gather operations given in [30], we get:

$$T = (N/P)\log(N) + (N/P)\log(N)$$

In this formula, the first $(N/P)\log(N)$ represents the time spent on processing in the FFT algorithm. The second $(N/P)\log(N)$ represents the time spent on communication.

We test the FT benchmark with three tests similar to those used for the EP benchmark: testing the processor scalability, problem-size scalability, and finally both together.

As with EP, we test processor scalability by providing a subset of the observations and predicting the observations not provided in that column. This time we perform three tests for each problem size. In the first test, we provide all but the 64 processor observations, in the second, we provide all but the 32 and 64 processor observations, and in the third we provide all but the 16, 32 and 64 processor observations. We perform these tests for Class A and Class B, as they are the only full columns which have a significant number of observations with runtime greater than one second. The results are summarized in Table 6 and Table 7.

As shown in Table 6, our scalability model yields fairly accurate point estimates for 16 and 32 processors, with all below 12% deviation from the mean of the observations. The point estimates for 64 processors seem quite poor, with deviations between 29% and 43% from the mean of the observations. This might be explained by the extremely small (less than a second) runtime values for the 64 node case. In all cases the mean of the observations falls within the 95% prediction interval, and two cases fall within the 95% confidence interval.

The scalability tests for Class B are slightly worse than Class A. However, the 16 processor case in the right column, and the 32 processor case in the center column are fairly accurate predictions, both with point estimates less than 15% from the mean of observations, and both with 95% prediction intervals that accurately predict the location of the mean of observations. As with Class A, some of the higher deviations of 64 processor predictions may be explained by the small size of the mean of observations.

Table 6. Results of processor scalability prediction of the 64, 32-64, and 16-64 node cases of Class A of the FT benchmark.

nodes \ class	A=256x256x128	A=256x256x128	A=256x256x128
2	9.91	9.91	9.91
4	5.30	5.30	5.30
8	2.68	2.68	2.68
16	1.51	1.51	1.57 +/- 0.1 (0.25) + 4.20%
32	0.87	0.94 +/- 0.06 (0.21) + 8.25%	0.97 +/- 0.11 (0.25) + 11.71%
64	0.61 +/- 0.05 (0.19) +29.79%	0.64 +/- 0.07 (0.21) + 36.17%	0.67 +/- 0.11 (0.26) + 42.55%

Table 7. Results of processor scalability prediction of the 64, 32-64, and 16-64 node cases of Class B of the FT benchmark.

nodes \ class	B=512x256x256	B=512x256x256	B=512x256x256
2	43.94	43.94	43.94
4	23.23	23.23	23.23
8	12.16	12.16	12.16
16	6.69	6.69	7.09 +/- 0.21 (0.56) + 6.06%
32	3.69	4.22 +/- 0.17 (0.56) + 14.52%	4.45 +/- 0.23 (0.56) + 20.76%
64	2.68 +/- 0.16 (0.64) + 37.20%	2.89 +/- 0.18 (0.56) + 47.95%	3.13 +/- 0.25 (0.57) + 60.24%

.

Table 8 shows the results of our problem-size scalability tests for the FT benchmark. Due to the small runtimes of all of the observations for Class S, these observations were excluded from the tests. The observations for Class W were included despite their small runtimes to provide us with enough data points to get meaningful predictions. We performed four tests, separately modeling 2, 16, 32, and 64 processors, and predicting Class C from the observations of Class W, A, and B. The point-estimates are very close to the mean of the observed values, all less than 4%. However, the means of the observed values do not fall into any of the 95% confidence or prediction intervals. This is likely due to the small number of configurations used to build the model. When very few observations are used to build a model, the model can easily pass very close to all provided observations, and based on this predicts tight confidence and prediction intervals.

Table 8. Results of the problem-size scalability prediction of the C problem size for the FT benchmark.

nodes \ class	W=128x128x32	A=256x256x128	B=512x256x256	C=512x512x512
4	0.25	5.30	23.23	100.47 +/- 0.12 (0.13) - 1.20%
16	0.08	1.51	6.69	28.91 +/- 0.30 (0.33) - 1.54%
32	0.04	0.87	3.69	15.91 +/- 0.13 (0.14) - 4.00%
64	0.02	0.47	1.95	8.43 +/- 0.09 (0.09) -2.36%

Table 9. Results of problem-size/processor scalability predictions for the FT benchmark. In the upper table, the 64 processor case of Class B, and the 16-64 processor cases of Class C are predicted. In the lower table, the 32-64 processor cases of Class B and the 16-64 processor cases of Class C are predicted.

nodes \ class	A=256x256x128	B=512x256x256	C=512x512x512
2	9.91	43.94	
4	5.30	23.23	
8	2.68	12.16	
16	1.51	6.69	24.22 +/- 0.21 (1.13) - 17.52%
32		3.69	12.42 +/- 0.15 (1.12) - 25.06%
64		1.99 +/- 0.19 (1.13) + 1.88 %	6.52 +/- 0.16 (1.12) - 24.48%

nodes \ class	A=256x256x128	B=512x256x256	C=512x512x512
2	9.91	43.94	
4	5.30	23.23	
8	2.68	12.16	
16	1.51	6.69	24.20 +/- 0.22 (1.17) - 17.58 %
32		3.29 +/- 0.21 (1.17) - 10.72%	12.38 +/- 0.17 (1.17) - 25.30%
64		1.92 +/- 0.22 (1.17) - 1.71%	6.46 +/- 0.19 (1.17) - 25.17%

As with the EP benchmark, we tested our system with the task of modeling both problem size and number of processors simultaneously for the FT benchmark. The very small runtimes of many of the observations for the FT benchmark made this difficult, and limited the number of useful observations that we could use. Including only observations greater than one second, we tested two configurations, with results shown in Table 9. In the first configuration, we provided observations from Class A and Class B for 2, 4, 8, and 16 processors, and 2, 4, 8, 16, and 32 processors respectively. We then predicted Class B with 64 processors, and Class C with 16, 32, and 64 processors. In the second configuration, we provided observations from Class A and Class B for 2, 4, 8, and 16 processors, and predicted Class B with 32 and 64 processors and Class C with 16, 32, and 64 processors. In both tests, the predictions for Class C deviate from the mean of observations by between 17% and 25%, with the 95% confidence and prediction intervals failing to capture the mean of observations. The predictions for Class B are better, with point estimates less than 11% from the mean of observations, and all of the prediction intervals (and most of the confidence intervals) accurately capturing the mean of observations. In the second test, the mean of observations for Class B with 32 processors falls outside of the 95% confidence interval, but only by 0.19 seconds. A possible explanation for the poorer predictions for Class C is the relative size of the Class C problem size as compared to the problem sizes of Class A and Class B. When making predictions for Class C, we are predicting for a problem size that is very far from the provided observations (4 times larger than Class B and 8 times larger than Class A).

Since the formula used for the FT benchmark provides us with a breakdown of the complexity into communication time and processing time, and the FT benchmark allows us to measure the time spent on various tasks, we can take a closer look at why some of our predictions for FT are inaccurate. Furthermore, this differentiation gives us a chance to demonstrate the capability of our system to predict different resource characteristics separately.

By taking a closer look at the algorithm, we find that the entire algorithm is made up of setup time and a number of iterations in which an FFT is performed and between which, the data is evolved. The FFT time consists of CPU (FFTcpu) and communication time (FFTcomm). We notice that two terms (setup time and evolve time) were not considered in the original model but have significant effect on the total runtime. Motivated by this, we also model these additional terms (SetupTime and EvolveTime).

$$T = SetupTime + EvolveTime + FFTcpu + FTcomm$$

Table 10. Mean values for the FT benchmark observations, broken down into Setup, Evolve, FFTcpu, and FFTcomm time for Class B.

nodes \ class	Total	Setup Time	Evolve Time	FFTcpu	FFTcomm	% fft comm
2	43.87	1.27	2.23	33.68	6.58	16.34%
4	23.27	0.63	1.13	16.80	4.67	21.74%
8	12.19	0.32	0.56	8.35	2.95	26.10%
16	6.78	0.16	0.29	3.96	2.32	36.89%
32	3.72	0.08	0.13	1.96	1.48	42.98%
64	1.95	0.04	0.07	0.97	0.85	46.62%

Table 10 shows the breakdown of the runtimes for the B problem size, according to the four cost components described above. Furthermore, the percentage of communication in relation to the overall runtime is given. Unlike our other FT tests, in these tests we only use three observations for each configuration to calculate the mean of observations and to input into the system for prediction.

We model SetupTime and EvolveTime both as N/P (obvious from the observations). The original CPU-time model becomes FFTcpu = (N/P)log(N). The model for communication remains the same and is now explicitly described in FFTcomm = (N/P)log(N).

Table 11 shows two tests in which models are generated and predictions are made for all terms separately. In the first tests, we provide observations for 2, 4, 8, 16, and 32 processors and predict the 64 processor values. In the second test, we provide observations for 2, 4, 8, and 16 processors and predict the 32 and 64 processor cases. We see that FFTcpu is modeled fairly accurately, with point predictions less than 11% from the mean of observations. However,

FFTcomm is quite poor, with deviations of 44% to 134% off the mean of observations. The Total Runtime prediction is simply the sum of the predictions of the individual components. These predictions are about the same as the predictions of the total runtime presented in Table 7. The results indicate that the inaccuracy of our predictions is due to an inaccurate communication model – which is critical, considering that the communication amounts to up to about 47%. It is well known that different algorithms may be chosen for the implementation of collective operations. For instance, the all-to-all which dominates in FT can be implemented using scatter/gather, pairwise-exchange, or a linear algorithm [31] [32]. Without knowing which algorithms are employed by the MPI library in use, the model cannot properly capture communication behavior. As a future extension, such information could be made available for all applications per collective operation in the dynamic directory. Another possible explanation is the small size of the communication observations for 32 and 64 processors. In a further test, we provided 2, 4, and 8 processors and predicted 16, 32, and 64 processors. The 16 processor prediction was fairly accurate (11.21% off the mean of observations).

Table 11. Results of the problem-size scalability prediction for Class B (512x256x256) of the FT benchmark, broken down into Setup, Evolve, FFTcpu, and FFTcomm time. In the upper table, we provide observations from 2-32 processors and predict the 64 processor case. In the lower table, we provide observations from 2-16 processors, and predict the 32 and 64 processor cases.

nodes \ class	Total Runtime	Setup Time	Evolve Time	FFTcpu	FFTcomm	% fft comm
2	43.87	1.27	2.23	33.68	6.58	16.34%
4	23.27	0.63	1.13	16.80	4.67	21.74%
8	12.19	0.32	0.56	8.35	2.95	26.10%
16	6.78	0.16	0.29	3.96	2.32	36.89%
32	3.72	0.08	0.13	1.96	1.48	42.98%
64	2.71 +/- 0.34 (0.97)	0.04 +/- 0.00 (0.01)	0.07 +/- 0.01 (0.02)	0.88 +/- 0.06 (0.18)	1.72 +/- 0.27 (0.76)	46.62%
	+ 38.74%	0.00%	0.00%	- 9.59%	+ 102.35%	

nodes \ class	Total Runtime	Setup Time	Evolve Time	FFTcpu	FFTcomm	% fft comm
2	43.87	1.27	2.23	33.68	6.58	16.34%
4	23.27	0.63	1.13	16.80	4.67	21.74%
8	12.19	0.32	0.56	8.35	2.95	26.10%
16	6.78	0.16	0.29	3.96	2.32	36.89%
32	4.30 +/- 0.35 (0.84)	0.08 +/- 0.00 (0.01)	0.15 +/- 0.01 (0.02)	1.93 +/- 0.09 (0.21)	2.14 +/- 0.25 (0.60)	42.98%
	+ 15.49%	0.00%	+ 15.38%	- 1.70%	+44.59%	
64	2.98 +/- 0.36 (0.84)	0.04 +/- 0.00 (0.01)	0.08 +/- 0.01 (0.02)	0.87 +/- 0.09 (0.21)	1.99 +/- 0.26 (0.60)	46.62%
	+ 52.56%	0.00%	+ 14.29%	- 10.62%	+134.12%	

5.4 Linpack

The Linpack benchmark is the standard tests of a supercomputer's performance, and is used to establish the Top 500 list of supercomputers. The benchmark application generates and then solves a random dense linear system using LU factorization [29]. We use the High-Performance Linpack (HPL) implementation of the Linpack benchmark provided in [29]. Unlike the FT benchmark, HPL can be run on any number of processors in a variety of configurations, and can be

run with user-specified problem sizes. As well, HPL allows the user to specify several characteristics of the algorithm, such as whether the benchmark will use a binary exchange swapping algorithm, a spread-roll swapping algorithm, or a hybrid of the two. The user is also able to specify the block size that is used by the algorithm. The complexity of the benchmark is well studied and is provided in [32]. Dropping the constant terms which do not make a difference in our system, the formula is as shown below:

$$T = N^3 / 3PQ + N^2(3P+Q)/(2PQ) + N \log(P) + NP$$

where T is the runtime, N is the size of one side of the square matrix constituting the problem size, and P and Q describe the arrangement of processors in a P by Q grid. Thus PQ gives the total number of processors used.

The benchmark was run on square configurations ($P = Q$) of nodes varying from 2x2 to 8x8, and with N varying from 8000 to 14000 in increments of 1000. Each configuration was run three times and in all cases we used the binary exchange swapping algorithm, and a block size of 64. The mean values of the observations are shown in Table 12.

Table 12. Mean values for the Linpack observations.

node \ size	8000^2	9000^2	10000^2	11000^2	12000^2	13000^2	14000^2
2x2	174.07	255.81	365.05	492.14	661.67	857.91	1066.88
3x3	69.12	104.54	148.43	201.67	264.84	344.20	436.65
4x4	32.37	49.69	74.84	105.76	140.54	183.68	233.86
5x5	20.38	29.72	42.49	59.68	81.12	111.18	145.49
6x6	14.30	20.48	27.31	38.11	51.30	68.43	90.51
7x7	11.50	15.10	20.76	29.13	38.36	46.98	59.28
8x8	8.63	11.75	15.76	20.50	27.16	34.47	43.87

Table 13. Processor-scalability tests for the Linpack benchmark.

node \ size	8000^2	9000^2	10000^2	11000^2	12000^2	13000^2	14000^2
2x2	174.07	255.81	365.05	492.14	661.67	857.91	1066.88
3x3	69.12	104.54	148.43	201.67	264.84	344.20	436.65
4x4	32.37	49.69	74.84	105.76	140.54	183.68	233.86
5x5	20.38	29.72	42.49	59.68	81.12	111.18	145.49
6x6	14.30	20.48	27.31	38.11	51.30	68.43	90.51
7x7	11.50	15.10	20.76	29.13	38.36	46.98	59.28
8x8	7.20 +/- 2.19 (2.43) -16.56%	8.57 +/- 1.75 (1.94) -27.10%	19.49 +/- 1.40 (1.55) + 23.70%	31.17 +/- 13.16 (14.55) + 52.02%	39.64 +/- 3.00 (3.33) + 45.96%	36.85 +/- 6.16 (6.82) + 6.90%	35.60 +/- 8.05 (8.91) - 18.85%

As with the EP and FT benchmarks, we test processor-scalability, problem-size scalability, and both processor and problem-size scalability simultaneously.

To test processor-scalability, we ran seven tests, one for each problem size. In each test, the problem size was kept constant and the processor configuration was varied. The observations for 2x2, 3x3, 4x4, 5x5, 6x6, and 7x7 processor configurations was provided, and 8x8 was predicted. The results are shown in Table 13.

The quality of the predictions varies between the different problem sizes, with point estimates from 6.90% to 52.02% from the mean of the corresponding observations. A possible explanation for the poor predictions is the small number of provided observations for a relatively complex formula. Multiple linear regression works best with large numbers of observations, and relatively simple formulas, with few coefficients to calculate. In this case, we have a formula with five coefficients to calculate (one in front of each term in the formula, and one constant term at

the end) and observations for six distinct processor configurations. The EP and FT benchmarks only have only two coefficients to calculate, though they also have few distinct configurations.

To test problem-size scalability we ran six separate tests. In each test, the processor configuration was kept constant, and the problem size was varied. The observations for 8000^2 to 12000^2 were provided, and 13000^2 and 14000^2 were predicted. The results are shown in Table 14. In all of the tests, the predictions are quite accurate, with all point estimates less than 9% from the mean of observations, and in all but two predictions, with the mean of observations falling within both the 95% confidence and prediction intervals.

Table 14. Problem-size scalability tests for the Linpack benchmark.

node \ size	8000^2	9000^2	10000^2	11000^2	12000^2	13000^2	14000^2
2x2	174.07	255.81	365.05	492.14	661.67	870.83 +/- 23.26 (24.66) + 1.51%	1130.51 +/- 65.55 (66.06) +5.96%
3x3	69.12	104.54	148.43	201.67	264.84	338.75 +/- 1.92 (2.04) -1.58%	424.11 +/- 5.41 (5.46) -2.87%
4x4	32.37	49.69	74.84	105.76	140.54	177.14 +/- 2.58 (2.74) -3.56%	213.59 +/- 7.28 (7.34) -8.67%
5x5	20.38	29.72	42.49	59.68	81.12	107.58 +/- 4.00 (4.24) -3.24%	139.34 +/- 11.28 (11.36) -4.23%
6x6	14.30	20.48	27.31	38.11	51.30	69.2 +/- 2.86 (3.03) +1.13%	92.20 +/- 8.05 (8.11) +1.87%
7x7	11.50	15.10	20.76	29.13	38.36	48.59 +/- 3.57 (3.78) +3.43%	58.99 +/- 10.05 (10.13) -0.49%
8x8	8.63	11.75	15.76	20.50	27.16	35.83 +/- 2.13 (2.26) +3.96%	47.18 +/- 6.00 (6.05) +7.56%

Next, we test processor and problem-size scalability at the same time. In this test, we provide the observations for problem sizes 80002 to 130002, and processor configurations 2x2 to 7x7, and we predict the 140002 problem size for all processor configurations, and the 8x8 processor configuration for all problem sizes. The results are shown in Table 15. As with the separate processor and problem-size tests above, the model predicts problem-size more accurately than processor-size. However, it is interesting how the processor-size predictions are improved considerably when both are modeled simultaneously.

In our final set of tests, we test our system with the task of modeling three parameters (N, P, and Q) of Linpack simultaneously, demonstrating the capability of our multiple linear regression approach. That is, we are modeling all three parameters from the complexity estimate above. For these tests, we gathered observations on our local 16 node cluster for N ranging from 3000 to 9000, for all possible configuration of 16 processors (1x16, 2x8, 4x4, 8x2, and 16x1). The mean values for the observations are shown in Table 16.

As a first test, we provide all observations except those for the 6000^2 problem size and 4x4 processor configuration, and then predict the excluded row and column. The results are shown in Table 17. The results are very good, with all of the point estimates less than 6% from the mean of observations, and the 95% prediction interval accurately capturing the mean of observation.

Table 15. Problem-size/processor scalability test for the Linpack benchmark.

node \ size	8000^2	9000^2	10000^2	11000^2	12000^2	13000^2	14000^2
2x2	174.07	255.81	365.05	492.14	661.67	857.91	1084.47 +/- 3.61 (6.13)
							+1.65%
3x3	69.12	104.54	148.43	201.67	264.84	344.20	443.60 +/- 1.35 (5.14)
							+1.59%
4x4	32.37	49.69	74.84	105.76	140.54	183.68	231.75 +/- 1.36 (5.14)
							-0.90%
5x5	20.38	29.72	42.49	59.68	81.12	111.18	138.66 +/- 1.26 (5.11)
							-4.70%
6x6	14.30	20.48	27.31	38.11	51.30	68.43	90.12 +/- 1.29 (5.12)
							-0.42%
7x7	11.50	15.10	20.76	29.13	38.36	46.98	61.56 +/- 1.77 (5.26)
							+3.84%
8x8	6.88 +/- 1.44 (5.16)	9.77 +/- 1.49 (5.18)	13.68 +/- 1.62 (5.21)	18.78 +/- 1.81 (5.28)	25.25 +/- 2.06 (5.37)	33.28 +/- 2.36 (5.49)	43.03 +/- 2.69 (5.64)
	-20.36%	-16.89%	-13.22%	-8.41%	-7.01%	-3.46%	-1.90%

Table 16. Mean values for observations of problem sizes 3000^2 to 9000^2 run on 16 nodes in various configurations.

config \ size	3000^2	4000^2	5000^2	6000^2	7000^2	8000^2	9000^2
1x16	2.95	5.93	10.41	16.66	25.22	36.22	50.15
2x8	2.49	5.22	9.45	15.37	23.42	33.59	46.71
4x4	2.50	5.14	9.26	14.96	22.74	32.87	45.57
8x2	2.95	5.95	10.42	16.60	24.84	35.48	48.81
16x1	4.23	8.09	13.68	21.08	31.01	43.51	58.97

Our second set of tests is somewhat more challenging, predicting the 16x1 processor configuration, and the 9000^2 problem size. The challenge comes from the 16x1 problem size, which shows significantly different runtimes from the other processor-configurations. The results are shown in Table 18. The predictions are quite accurate, this time with all point estimates less than 4% from the mean of the observations, and nearly all of the means of observations within the 95% confidence and prediction intervals (the exceptions being 4x4 9000^2 and 8x2 9000^2).

Table 17. Problem-size/processor-configuration test for the Linpack benchmark, predicting 6000^2 problem size, and 4x4 processor-configuration.

config \ size	3000^2	4000^2	5000^2	6000^2	7000^2	8000^2	9000^2
1x16	2.95	5.93	10.41	16.7 +/- 0.15 (0.72)	25.22	36.22	50.15
				+ 0.21%			
2x8	2.49	5.22	9.45	14.96 +/- 0.11 (0.71)	23.42	33.59	46.71
				-2.63%			
4x4	2.62 +/- 0.13 (0.71)	4.97 +/- 0.14 (0.71)	8.84 +/- 0.15 (0.71)	14.57 +/- 0.15 (0.71)	22.51 +/- 0.16 (0.72)	33.01 +/- 0.19 (0.72)	46.42 +/- 0.27 (0.75)
	+ 5.09%	-3.22%	-4.53%	-2.63%	-1.01%	+ 0.43%	+ 1.87%
8x2	2.95	5.95	10.42	16.05 +/- 0.15 (0.71)	24.84	35.48	48.81
				-3.32%			
16x1	4.23	8.09	13.68	20.85 +/- 0.18 (0.72)	31.01	43.51	58.97
				-1.09%			

Table 18. Problem-size/processor-configuration test for the Linpack benchmark, predicting the 9000^2 problem size, and 16x1 processor-configuration.

config \ size	3000^2	4000^2	5000^2	6000^2	7000^2	8000^2	9000^2
1x16	2.95	5.93	10.41	16.66	25.22	36.22	49.88 +/- 0.32 (0.65)
							-0.52%
2x8	2.49	5.22	9.45	15.37	23.42	33.59	46.88 +/- 0.44 (0.71)
							+ 0.36%
4x4	2.50	5.14	9.26	14.96	22.74	32.87	46.75 +/- 0.33 (0.66)
							+2.59%
8x2	2.95	5.95	10.42	16.60	24.84	35.48	50.18 +/- 0.27 (0.63)
							+ 2.81%
16x1	4.29 +/- 0.74 (0.93)	7.84 +/- 0.79 (0.97)	13.21 +/- 0.80 (0.98)	20.74 +/- 0.78 (0.96)	30.75 +/- 0.78 (0.97)	43.61 +/- 0.83 (1.01)	59.64 +/- 0.95 (1.10)

6 Summary and Conclusion

We have presented an approach to employ both complexity estimates from the user and historical information from previous runs to make scalable predictions in the processor-number dimension (processor scalability), problem-size dimension (problem-size scalability), and processor-number/problem-size dimensions simultaneously. The solution applied is multiple linear regression which not only provides predictions of mean values but also confidence and prediction intervals. The user provides rough complexity estimates and the coefficients are determined by the predictor.

In our tests on the NAS EP and FT benchmarks, and the Linpack benchmark we have demonstrated that this approach is capable of making reliable predictions if the complexity estimate / model provided by the user are decently accurate. This requirement can create problems when communication libraries are utilized, as they may employ different algorithms, or even switch between algorithms depending on different parameters of the communication operations. The former problem can be addressed by documenting the complexity of the communication algorithms implemented in popular libraries, and possibly making this information available for automatic retrieval. Another difficulty can be that problem-sizes and the number of nodes used often grow exponentially, as observed in the FT benchmark. This leads to the prediction of data points far away from the set of observations.

We have also demonstrated that different characteristics such as computation and communication time can be considered and predicted separately, which is useful for coscheduling in a time-sharing environment.

Future work includes automatic checking of whether the assumptions regarding the error term hold and rejecting or modifying a model for which they are not met. A more advanced extension would be to experiment with automatically correcting inaccurate models by tear-down and build-up approaches of the function. That such approaches are feasible has been shown in [33]. How feasible it is for a user to provide the necessary models requires further exploration, as does the possibility of replacing or supplementing user estimates with compiler-derived models. The required models could also potentially be generated in a fully automated manner [34].

Acknowledgement

This research was supported by CFI via Grant No. 6191 and partially by NSERC. We thank SHARCNET for allowing us time on 64 nodes of their cluster at McMaster University, and in particular Mark Hahn for assisting us with running our tests.

References

[1] Angela C. Sodan and Lun Liu. Dynamic Multi-Resource Monitoring for Predictive Job Scheduling with ScoPro. Technical Report 04-002, U of W, CS Department, February 2005.

[2] Angela C. Sodan and Xuemin Huang. Adaptive Time/Space Scheduling with SCOJO. Int. Symp. on High-Performance Computing Systems (HPCS), Winnipeg/Manitoba, May 2004, pp. 165-178.

[3] Angela C. Sodan and Lin Han. ATOP—Space and Time Adaptation for Parallel and Grid Applications via Flexible Data Partitioning. 3rd ACM/IFIP/USENIX Workshop on Reflective and Adaptive Middleware, Toronto, Oct. 2004.

[4] Angela C. Sodan and Lei Lan. LOMARC—Lookahead Matchmaking in Multi-Resource Coscheduling. JSSPP (Workshop on Job Scheduling Strategies for Parallel Processing), New York / USA, June 2004, to appear in Springer.

[5] W. Cirne and F. Berman. A Model for Moldable Supercomputer Jobs. *Proc. Internat. Parallel and Distributed Processing Symposium (IPDPS)*, April 2001.

[6] Angela C. Sodan. Loosely Coordinated Coscheduling in the Context of Other Dynamic Approaches for Job Scheduling—A Survey. Concurrency&Computation: Practice&Experience. Accepted for publication. *(57 pages).*

[7] V. K. Naik, S. K. Setia, and M. S. Squillante. Processor Allocation in Multiprogrammed Distributed-Memory Parallel Computer Systems. *J. of Parallel and Distributed Computing*, Vol. 46, No. 1, 1997, pp. 28-47.

[8] Eitan Frachtenberg, Dror Feitelson, Fabrizio Petrini, and Juan Fernandez. Flexible CoScheduling: Mitigating Load Imbalance and Improving Utilization of Heterogeneous Resources. Proc. Int. Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France, April 2003.

[9] R.A. Gibbons Historical Application Profiler for Use by Parallel Schedulers. Proc. IPPS Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), April 1997, Lecture Notes in Computer Science **1291**, Springer Verlag.

[10] Mu'alem A and Feitelson D G. 2001. Utilization, Predictability, Workloads and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Transactions Parallel & Distributed Systems* June 2001, **12**(6).

[11] Perkovic D and Keleher P J. Randomization, Speculation, and Adaptation in Batch Schedulers. *Proc. ACM/IEEE Supercomputing (SC)*, Dallas/TX, Nov. 2000.

[12] Chiang S-H and Vernon M K. Characteristics of a Large Shared Memory Production Workload. *Proc. Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, June 2001, *Lecture Notes in Computer Science* **2221**, Springer-Verlag, pp. 159-187.

[13] Smith W, Taylor V, and Foster I. Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance. *Proc. Workshop*

on Job Scheduling Strategies for Parallel Processing (JSSPP), 1999, *Lecture Notes in Computer Science* **1659**, Springer Verlag.

[14] Arpaci-Dusseau A C, Culler D E, and Mainwaring A M. Scheduling with Implicit Information in Distributed Systems. *Proc. SIGMETRICS'98/PERFORMANCE'98 Joint Conference on the Measurement and Modeling of Computer Systems*, Madison/WI, USA, June 1998.

[15] M.E. Crovella and T.J. LeBlanc. Parallel Performance Prediction Using Lost Cycles Analysis. Proc. Supercomputing (SC), 1994.

[16] K. Keahey, P. Beckman, and J. Ahrens. Ligature: Component Architecture for High Performance Applications. The International Journal of High Performance Applications, 14(4):347-356, Winter 2000.

[17] Frederik Vraalsen, Ruth A. Aydt, Celso L. Mendes, and Daniel A. Reed. Performance Contracts: Predicting and Monitoring Grid Application Behavior. In Proc. 2nd Internat. Workshop on Grid Computing, Nov. 2001.

[18] G. Marin and J. Mellor-Crummey. Cross-Architecture Predictions for Scientific Applications Using Parameterized Models. Proc. Joint. Internat. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS), New York, NY, USA, June 2004.

[19] A. Snavely, L. Carrington, and N. Wolter. Modeling Application Performance by Convolving Machine Signatures with Application Profiles. In Proc. IEEE 3th Annual Workshop on Workload Characterization, 2001.

[20] NIST/SEMATECH e-Handbook of Statistical Methods, available at http://www.itl.nist.gov/div898/handbook, retrieved October, 2004.

[21] J. Cohen, P. Cohen, S.G. West, and L.S. Alken. Applied Multiple Regression/Correlation Analysis for the Behavioural Sciences, 3rd edition. Mahwah, New Jersey, USA: Lawrence Erlbaum Associates, 2003.

[22] W. Mendenhall, R.J. Beaver, and B.M. Beaver. Introduction to Probability and Statistics, 10th edition. Pacific Grove, CA, USA: Brooks/Cole Publishing Company, 1999.

[23] D.H. Bailey, T. Harris, W.C. Saphir, R.F. Van der Wijngaart, A.C. Woo, M. Yarrow. *The NAS Parallel Benchmarks 2.0.* NAS Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, 1995.

[24] A. Grama, A. Gupta, G. Karypis, V. Kumar. (2003) *Introduction to Parallel Computing,* 2nd ed. Addison Wesley.

[25] M. Yarrow, R.F. Van der Wijngaart. Communication Improvement for the LU NAS Parallel Benchmark: A Model for Efficient Parallel Relaxation Schemes. NAS Technical Report NAS-97-032, NASA Ames Research Center, Moffett Field, CA, 1997.

[26] E. Barszcz, R. Fatoohi, V. Venkatakrishnan, S. Weeratunga, *Solution of Regular, Sparse Triangular Linear Systems on Vector and Distributed-Memory Multiprocessors,* NAS Applied Research Branch Report RNR-94-007, NASA Ames Research Center, Moffet Field, CA, 1993.

[27] Maple 9.5—Advanced Mathematics Software for Engineers, Academics, Researchers, and Students, http://www.maplesoft.com/products/maple/index.aspx, retrieved December 2004.

[28] OpenMaple—An API into Maple, http://www.adaptscience.com/products/mathsim/maple/html/OpenMaple.html, retrieved December 2004.

[29] HPL – A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, available at http://www.netlib.org/benchmark/hpl/, retrieved June, 2005.

[30] Ian Foster. Designing and Building Parallel Programs. Reading, MA: Addison-Wesley, 1995.

[31] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra. *Automatically Tuned Collective Communications*. IEEE/ACM Supercomputing, Nov. 2000.

[32] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. *Performance Analysis of MPI Collective Operations*. PMEO-PDS, Apr. 2005.

[33] Ljupčo Todorowski, Peter Ljubič, and Sašo Džeroski. *Inducing Polynomial Equations for Regression*. ECML, 2004.

[34] E. Schmidt, A. Schulz, L. Kruse, G. von Cölln, and W. Nebel. *Automatic Generation of Complexity Functions for High-Level Power Analysis*. PATMOS, 2001.