Dynamic Performance Prediction for Scalable Parallel Computing

Mark J. Clement and Michael J. Quinn

July 22, 1994

Abstract

Performance prediction is necessary in order to deal with multi-dimensional performance effects on parallel systems. The compiler-generated analytical model developed in this paper accounts for the effects of cache behavior, CPU execution time and message passing overhead for real programs written in high level data-parallel languages. The performance prediction technique is shown to be effective in analyzing several non-trivial applications as the problem size and number of processors vary. We leverage technology from the Maple symbolic manipulation system and the S-PLUS statistical package in order to present users with critical performance information necessary for performance debugging, architectural enhancement and procurement of parallel systems. The usability of these results is improved through specifying confidence intervals as well as predicted execution times for parallel applications. Cost optimal analysis techniques are also developed using the symbolic equations for execution time.

1 Introduction

Several implementations of "Grand Challenge" problems require far more processing power than any sequential processor can deliver [1] . Although significant progress is being made in the speed of microprocessor based computers, there is nearly universal agreement that continuing increases in computational power for large scientific applications will require parallel processing.

Massively Parallel Processing (MPP) systems promise to provide continued increases in performance by employing thousands of computing elements in the solution of a single problem. These distributed memory machines can be assembled using the fastest microprocessors with relatively inexpensive communications networks. The actual performance achieved on these machines for real applications, however, can be disappointing. For example, traditional supercomputers such as the Cray C-90 deliver 50% of their peak advertised performance on the NAS benchmarks while multicomputers like the IBM SP1 and the Meiko CS-2 currently achieve less than 5% of the advertised rate [2]. The causes of this low level of efficiency can be traced to unbalanced hardware systems and inefficient algorithmic implementations.

Several problems have been noted in the parallel processing environment that cause reported performance results to be much poorer than would be expected when actual programs are developed to run on multicomputers:

- **Performance Debugging**
  It is difficult for a programmer to predict the impact that system parameters will have on parallel code. As a result, software developers are frequently unable to evaluate different algorithmic implementations in order to arrive at an optimal solution. There is also little information available to indicate which parallel architecture, if any, will execute a given program in an efficient manner.

- **Architectural Improvement**
  Parallel system designers are handicapped by the lack of information on the effect changing system parameters will have on the performance of actual parallel programs. As a result, systems may be designed with such a high level of imbalance that few parallel programs will be able to execute in an efficient manner.

- **Machine Selection**
  The high performance computing environment has changed significantly with the advent of high speed workstations. Much of the work that was previously performed on large centralized computers is now done on desktop workstations. Clusters of workstations are often available for larger jobs with Massively Parallel Processing (MPP) machines being reserved for the most demanding applications. Selecting the appropriate computing platform to achieve efficient execution can be an important part of program development.

- **Compiler Optimizations**

  High level parallel languages are important because they allow a program to be written in a machine independent manner, and abstract away the complexity of explicit message passing. Compilers for these languages are often unable to choose optimizations which will result in the best performance for an application.

Solving the problems of performance debugging, architectural tuning, machine selection and compiler optimizations is a critical step in improving the efficiency of multicomputers. For sequential environments, system performance can be adequately described in terms of the amount of computation time required together with the processor instruction rate. The performance of parallel systems is influenced by multiple factors related to the structure of applications and to their ability to exploit the parallel features of a particular system. As a result of the multidimensional nature of this environment, users must be able to account for and visualize the effects of multiple variables on system performance [3]. Without an accurate model that takes all of these factors into account, it is difficult to determine what changes are needed to improve the performance of multicomputers. This multi-dimensional nature of distributed memory architectures makes sophisticated performance tools necessary in order to effectively utilize these systems [4, 5]. Performance prediction tools have been identified as an important technology in achieving the solution to grand challenge problems [1].

Although predicted execution times can be useful in improving the performance of multicomputers, their value can be limited unless there is some way of determining the accuracy of the prediction. By utilizing multivariate statistical techniques, this research specifies a confidence interval for predictions, indicating the expected range of values. This estimate of accuracy is particularly important for applications with variable problem size which we will be focusing on in this paper.

Previous research has investigated models which assume a statistical characterization of application programs [6]. These methods cannot provide accurate estimates of performance for real applications. Simulation based systems [7] require excessive time to arrive at predicted values. Although task graph representations [8] of algorithms expose many of the details of parallel execution, rewriting an application in another language may impose an unreasonable requirement on a developer. Static performance prediction techniques [9, 10, 11] are often unable to estimate execution time for applications with variable problem sizes. The isoefficiency metric [12] deals effectively with problem size scaling, but the asymptotic equations generated are not sufficiently detailed to generate exact execution times. Although multiple runs of an application with different problem sizes can be used to determine the constants for these equations [13], this curve fitting technique may be impractical for some users of performance prediction information.

The dynamic performance model developed here uses a single instrumentation run of the program with a reduced problem size to generate an equation for execution time which includes the contributions of each basic block in a program. Previous research has shown that compile time analysis can be used to characterize loop iterations for several static applications [14, 11], but for many programming constructs a sample execution of the application will be required. The equation can be differentiated to determine the sensitivity of an application to changes in the hardware architecture. With compiler inserted instrumentation code, little effort is required from the programmer. The instrumentation run can also be performed on a single processor workstation, freeing MPP systems for running production code.

In this paper we first describe the class of scalable algorithms which we focus on in this research. We then detail the performance modeling system implementation. Multiple regression is then examined as a means for determining the coefficients for this model. Several examples are then given to show the utility of this approach in examining performance issues along with conclusions and future work.

2 Scalable Algorithms

Scalable, data parallel applications are an important sub-class of problems which can be solved on multicomputers. It has been estimated that 90% of scientific and engineering applications are data parallel in nature [15]. Many of these programs can be scaled up by varying the number and size of data elements.

Scalable algorithms are able to utilize increasing numbers of processing elements efficiently. The number of processors that a fixed problem size application can effectively use is limited by the fraction of the program which is sequential in nature. This limitation does not apply to scalable applications since the ratio of parallel to sequential code can be increased as more processors are added to the problem [16].

Performance prediction is particularly important for scalable applications because they are not amenable to traditional performance debugging techniques. Given a sample application that runs for one hour on 1000 processors, it is impractical to gather trace data or to perform a simulation of the program to gather performance information. If we assume that accurate trace-based analysis requires 2 Mbyte/sec of data [17] then our sample application would generate 6 Tbytes of trace data. This data volume is clearly impractical even if the data could be written in real time without significant performance perturbation. Methods which achieve constant reductions in trace volume [18] are still unacceptable when viewed in light of the current trend towards machines with rapidly increasing CPU speeds and relatively constant disk bandwidth.

Performance debugging for scalable applications can be simplified by using the mathematical equation generated through dynamic performance prediction. Given the equation for execution time with variables corresponding to the problem size and number of processors, an application can be analyzed to determine the relative contribution of each line of code under selected scaling schemes. The speedup metric can be examined by increasing the number of processors in the equation as the problem size is held constant. Scaleup can be examined to determine the impact of each basic block as the problem size increase with a fixed number of processors. The scaled speedup metric examines performance as both the problem size and number of processors vary. Figure 1 summarizes the different views of an application under these different scaling schemes. Figure 2 shows the contributions of basic blocks in the Gaussian elimination application under speedup scaling. The symbolic equation resulting from dynamic performance prediction allows this kind of analysis to be performed.

The complexity of manual performance analysis for scalable applications is significantly greater than that found on parallel programs with constant problem size. This research uses dynamic performance prediction techniques to make progress in solving this important problem.

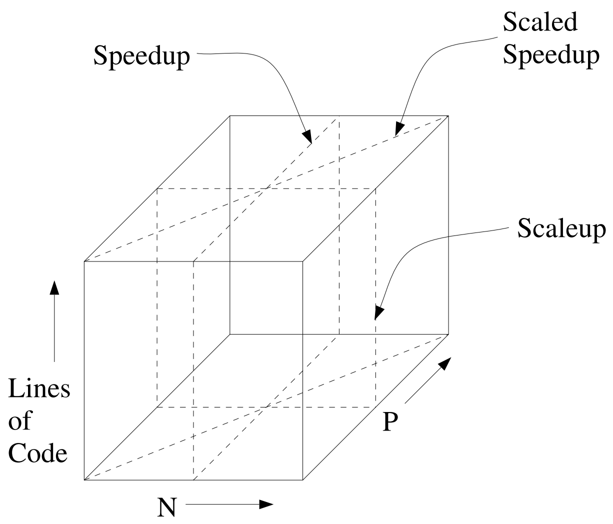


Figure 1: Views of basic block contributions to overall execution time under different scaling schemes.

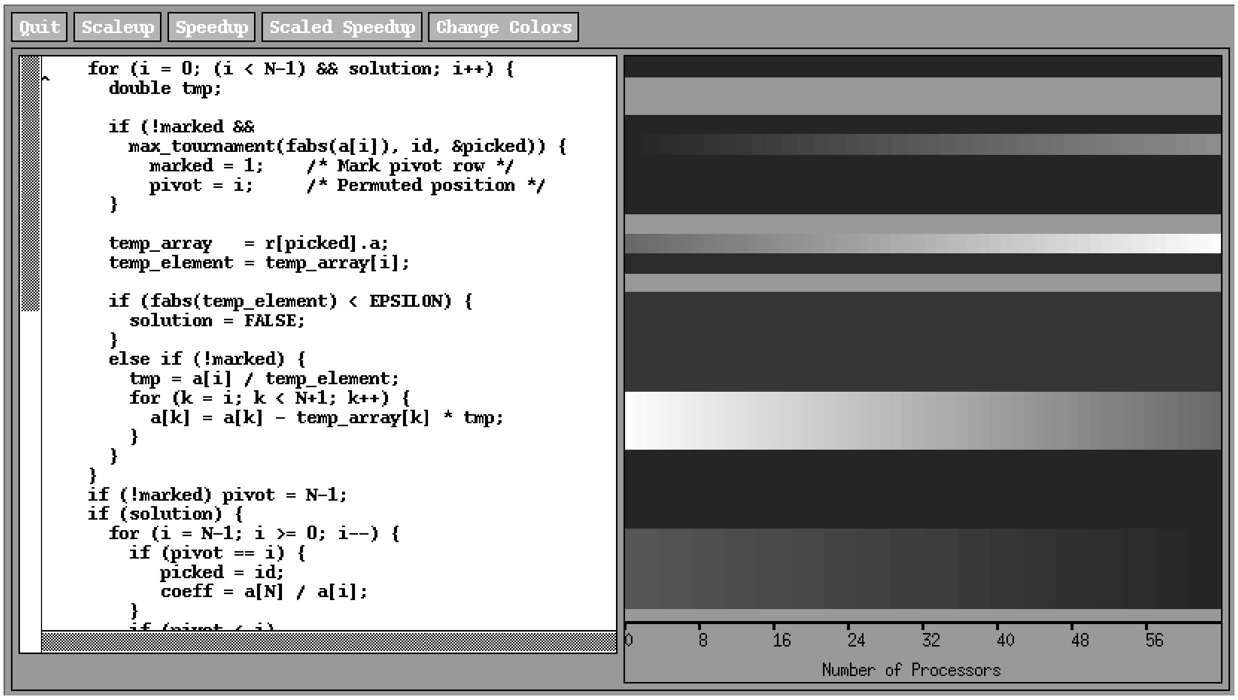


Figure 2: Contributions of basic blocks in Gaussian elimination application with speedup scaling. Lighter colors indicate a higher percentage of total execution time.

3 Dynamic Performance Prediction

The direction for this research was motivated by meetings with a commercial MPP vendor to determine requirements for a performance analysis tool. Members of the programming tools and system architecture groups suggested the following specifications for a performance prediction system.

- The model should allow a user to predict performance for larger problem sizes and larger number of processors than the machine used during performance debugging. This allows smaller parallel machines or workstations to be used during program development with large MPP machines being reserved for solving computational problems. It also speeds up instrumentation runs of a program since smaller problem sizes can be used during performance debugging.

- One means of determining the portability of an application is to determine the sensitivity of a program to changes in critical system parameters. A performance model should allow the user to view the sensitivity to system parameters as the problem size and number of processors vary. The validity of the model can also be determined through analyzing the sensitivity of the model to a particular parameter and the confidence level for that parameter.

- Several of the MPP systems currently in production support advanced operating systems with virtual memory. An effective performance prediction model should account for paging activity as the data size grows to the point that it does not fit in physical memory.

- In order to enable performance debugging, a performance model must allow the user to determine the relative contribution of each basic block in an application program.

Figure 3 shows a block diagram of the dynamic performance prediction system. Our implementation focuses on analysis for the Dataparallel C programming language [19]. The basic constructs can be extended to other explicitly data-parallel languages. We have found that programs written in high level parallel languages enable more accurate and complete performance analysis techniques than code written in an imperative language with message

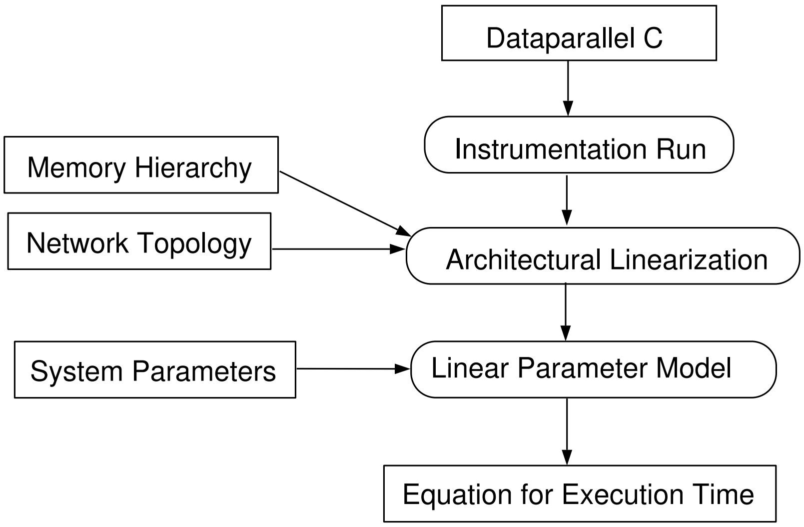


Figure 3: Block diagram for dynamic performance prediction system

passing. This is another argument for high level languages as opposed to the message passing added to an imperative language style which is often used today. Given Dataparallel C source code, instrumentation code is inserted to gather execution statistics which will be used to build a call graph for the application. Architecture specifications for the target machine are then passed to a linearization phase which outputs operation counts for significant system parameters. These counts are then combined with costs in time for each operation type resulting in a symbolic equation for execution time. Since the result of this model is an equation rather than a time estimate for a given problem size the execution time can be differentiated with respect to a given system parameter. The resulting equation can be used to determine the sensitivity of the application to changes in that parameter as the problem is scaled up.

3.1 Instrumentation Run

Instrumentation required by the dynamic performance prediction system can be inserted by a source-to-source compiler. It consists of static declarations of predefined data types and calls to a prediction library routine. The time spent in instrumentation library routines has been minimized to reduce the execution time for the prediction library calls. By using existing techniques, the perturbation caused by instrumentation could be removed from the time for the instrumentation run [20]. This time could then be used to improve the operation count for the program.

The Dataparallel C compiler replicates scalar values on all processors and distributes parallel variables across the nodes of the system. Total data volume for scalable applications will be dominated by parallel variables. The instrumentation code accounts for the size of parallel variables by declaring a shape descriptor structure which will specify the number of dimensions in a shape and the number of positions in each dimension.

Iteration constructs are instrumented with a loop descriptor structure which specifies the symbolic name of the iteration variable along with the initialization, termination and increment expressions. This symbolic information will be used to analyze complex iteration constructs. Conditional code is also instrumented to determine the true ratios. Virtual processor emulation loops are instrumented with shape instance descriptor structures which are organized in a linked list. Information on the data size accessed during each virtual processor loop will be used to determine the number of cache misses. The type of communication and the size of the data instance to be transferred are also accounted for in each communication block.

At the conclusion of the instrumentation run, the performance prediction system builds a call graph data structure which is used to scale the number of loop iterations and the size of each shape to the problem size. The instrumentation library code recursively descends the call graph and for each loop construct the following attempts are made to determine the complexity of the loop:

1. The most accurate results occur when the number of loop iterations can be determined from symbolic information. A search is made beginning from the parent node of the loop under analysis to the top of the call graph tree. If the initialization or termination values for the loop are iteration variables for an enclosing loop, then that symbolic value is used in subsequent analyses.

2. If a symbolic value cannot be found then a simplified fit is attempted to determine if the number of iterations found in the instrumentation run is an even multiple of the problem size.

3. If the initialization or termination expression cannot be scaled to problem size then it is assumed to be constant.

This instrumentation strategy is effective in structured programs where the iteration variables are not modified within the loop body. It is also restricted to non-recursive applications without "goto" statements. In a survey of 112 supercomputer applications from the College of Oceanographic and Atmospheric Sciences at Oregon State University, 98% of the loop constructs were amenable to this analysis strategy. Shape descriptor structures are also scaled to the problem size in a similar manner. Since the call graph structure is constructed symbolically, program complexities which cannot be analyzed computationally can be entered by a programmer in symbolic form if additional accuracy is required.

The instrumentation library outputs counts of important predictors in terms of the problem size. The equations for large programs can be highly complex since there is an element corresponding to each basic block in every equation. These equations are independent of the particular machine architecture used and will be passed to the linearization phase where a specific architecture will be modeled.

3.2 Architectural Linearization

The architectural linearization phase of performance prediction reduces complex machine characteristics to equations linear with respect to the speed of hardware subsystems. The major architectural features we analyze here are:

- On-chip cache and page fault behavior.
- Message startup times for interprocessor communications.
- Bandwidth characteristics for different communication patterns.

Previous research has investigated the impact that memory hierarchy can have on the performance achieved by a parallel program. A combination of local analytical models and empirical observations can characterize the performance of the memory system [21]. References to parallel variables in virtual processor emulation loops have a highly sequential access pattern, enabling accurate prediction of the number of cache misses and page faults which will occur during the execution of a scalable application. For our analysis we assume that if the size of all data accessed during a virtual processor emulation loop is greater than the cache size, then the processor will miss in the cache for the whole data array. Otherwise there is no cache miss penalty. This applies to both on-chip cache and virtual memory. We use the Heaviside step function to represent this relationship where

$$Heaviside(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{otherwise} \end{cases}$$

When the Heaviside function is differentiated it results in the Dirac delta function.

$$Dirac(x) = \frac{d}{dx}(Heaviside) = \begin{cases} \infty & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases}$$

This differentiability property will be used in later analysis to derive the cost optimal point for cache sizes. As a result of this linearization step, expressions for the number of cache misses can be derived.

Interprocessor communication can have a significant impact on the performance of a parallel application. We have found that modeling the number of message startup times necessary for a communication and the number of bytes transmitted results in an accurate estimation of the total communication cost. For each communication type, we model the number of messages which must be initiated to perform the transfer. Neighbor communications require a single message while broadcasts using a binomial tree algorithm require time logarithmic in the number of processors. The complexity of the other communication patterns for Dataparallel C has been thoroughly investigated previously [19] and is included in our model of the application. The expressions for message startup times and for the bandwidth must be altered to model different topologies, but will be constant for architectures with similar communication networks.

As a result of the architectural linearization phase, expressions for operation counts are computed as a function of the problem size and the number of processors.

3.3 Linear Parameter Model

Given counts of each operation ($\mathcal{X}$ values) and the cost for that operation (β values) on a given system, total execution time can be predicted for the application being modeled. For a given problem size and number of processors the execution time

$$t = \mathcal{X}_{Ops}\beta_{Ops} + \mathcal{X}_{VP}\beta_{VP} + \mathcal{X}_{L1}\beta_{L1} + \mathcal{X}_{L2}\beta_{L2} + \mathcal{X}_{St}\beta_{St} + \mathcal{X}_{Bw}\beta_{Bw} + e$$

where e is the error, or difference between the predicted and actual time. The variable β_{Ops} is the time in nanoseconds to perform one CPU operation and $\mathcal{X}_{Ops}$ the number of instructions. β_{VP} is the overhead associated with setting up a virtual processor emulation loop and $\mathcal{X}_{VP}$ is the number of loops entered during the execution. The values β_{L1} and β_{L2} represent the penalty for a miss in level one and two cache respectively with $\mathcal{X}_{L1}$ and $\mathcal{X}_{L2}$ the number of cache misses in each cache. The variable β_{St} is the message startup cost with $\mathcal{X}_{St}$ the number of messages sent. Bandwidth considerations are dealt with through β_{Bw} as the time required to transmit one byte through a connection to a processing element and $\mathcal{X}_{Bw}$ and the number of bytes transmitted. The predicted execution time $\hat{t} = t - e$.

If we consider a sample of n observations with $\mathcal{X}$ values from applications with different problem sizes and numbers of processors, then in vector notation, we have:

$$\begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_n \end{bmatrix} = \begin{bmatrix} \mathcal{X}_{Ops_1} & \mathcal{X}_{VP_1} & \mathcal{X}_{L1_1} & \mathcal{X}_{L2_1} & \mathcal{X}_{St_1} & \mathcal{X}_{Bw_1} \\ \mathcal{X}_{Ops_2} & \mathcal{X}_{VP_2} & \mathcal{X}_{L1_2} & \mathcal{X}_{L2_2} & \mathcal{X}_{St_2} & \mathcal{X}_{Bw_2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathcal{X}_{Ops_n} & \mathcal{X}_{VP_n} & \mathcal{X}_{L1_n} & \mathcal{X}_{L2_n} & \mathcal{X}_{St_n} & \mathcal{X}_{Bw_n} \end{bmatrix} \begin{bmatrix} \beta_{Ops} \\ \beta_{VP} \\ \beta_{L1} \\ \beta_{L2} \\ \beta_{St} \\ \beta_{Bw} \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{bmatrix}$$

or

$$\boldsymbol{t} = \boldsymbol{\mathcal{X}}\boldsymbol{\beta} + \boldsymbol{e}$$

where

$$\begin{aligned} \boldsymbol{t} &= \text{a column vector of } n \text{ observed values of } t = \{t_1, t_2, \ldots, t_n\} \\ \boldsymbol{\mathcal{X}} &= \text{an } n \text{ row column matrix containing linearized algorithmic characteristics} \\ \boldsymbol{\beta} &= \text{a column vector containing the cost in seconds for each operation} \\ \boldsymbol{e} &= \text{a column vector of the errors between predicted and experimental values} \end{aligned}$$

Values for the $\boldsymbol{\beta}$ vector can be obtained from system manufacturers or benchmark programs. The matrix formulation of the model also makes it possible to use multivariate techniques to obtain β values given a statistically significant number of experimental runs with different problem sizes and numbers of processors. These statistical techniques will be explored in the next section.

The dynamic performance prediction methodology presented here derives detailed symbolic equations accounting for major components in execution time. These equations can be used to analyze performance for scalable applications as the problem size and architecture varies.

4 Statistical Approximation of System Parameters

Multivariate statistics refers to a group of inferential techniques that have been developed to handle situations where sets of variables are involved as predictors of performance [22]. In classical scientific experiments, an effort is made to eliminate all but one causal factor through experimental control. The variables in our analytical model are difficult to isolate; hence, more complex methods are needed to estimate the value of model coefficients. Statistical software packages such as S-PLUS [23] allow large quantities of multivariate data to be analyzed with relative ease [24].

4.1 Advantages

Using statistical techniques to estimate the coefficients (β values) for the dynamic performance prediction model has several advantages:

- Statistical packages provide standard error values for each of the prediction variables. These values allow us to specify a confidence interval as well as an expected value for predicted performance on a target architecture.
- The model can be fit in an automated and structured way using real applications similar to the expected load for a parallel system.
- Standard information available from statistical software packages assesses the correlation of the model to experimental data. This information allows us to tune the model in order to reduce prediction error.

Multivariate statistical analysis must have access to a large number of samples in order to fit the model. Scalable applications are good candidates for this environment because multiple samples can be obtained from a single program using different problem sizes. Our results indicate that a reasonably accurate fit can be obtained from a limited number of applications. Statistical methods for finding coefficients may also be applicable to non-scalable applications if a larger number of sample applications are available.

4.2 Assumptions

The performance prediction model developed in this research was designed to create a linear model with respect to the important system characteristics we have identified. The following assumptions have been made in order to apply statistical techniques to this model:

- Both the predictor variables and the model errors are statistically independent. As the number of mathematical operations performed by a parallel application increases, there will not necessarily be a corresponding increase in the number of cache misses or communications performed by the application. The independence of these variables is important to the application of linear regression methods. This assumption is approximately true within runs of a single application as the problem size is varied. The assertion that the $\mathcal{X}$ values are independent is even stronger when multiple applications are included in the set of programs used to fit the model.

- The $\mathcal{X}$ matrix is able to characterize important performance indicators equally well from application to application. Given an application $\mathcal{A}$ which we would like to make predictions for and a set of applications $\mathcal{S}$ used in fitting the model where $\mathcal{A} \notin \mathcal{S}$ we assume that the $\mathcal{X}$ matrix represents algorithmic characteristics equally well for $\mathcal{A}$ as for the other applications in $\mathcal{S}$. This assumption can be made in a particularly strong way among programs addressing a single problem. A supercomputing site may have several researchers developing fluid dynamics applications and a model fit to existing fluid flow programs will generalize well to other applications in this class. Our experience has shown that as long as $\mathcal{S}$ contains a large number of samples, the fit is quite good for programs not in the fitting set.

- The true relationship between the predicted time and model variables is linear and the algorithmic measurements are accurate. Inaccuracies in $\mathcal{X}$ values can degrade the validity of regression results. Our results indicate that accurate operation counts can be obtained through an instrumented run of a scaled down version of an application.

- Errors are normally distributed with zero mean and a constant standard deviation. The shape of the quantile-quantile curve can be used to determine the correlation of residuals to a normal distribution. Our analysis of quantile plots for experimental data shows that this is a reasonable assumption.

4.3 Statistical Model

Multiple linear regression techniques model a numeric response variable, y, by a linear combination of p predictor variables x_j for $j = 1, \ldots, p$. The predicted values are the sum of coefficients β_j multiplied by the corresponding x_j.

$$y = \beta_1 x_1 + \cdots + \beta_p x_p$$

Linear *least-squares* models (LSQ) estimate the coefficients to minimize the squared sum of errors between predicted and experimental values. If the response and predictors corresponding to the ith of n observations are $y_i, x_{i1}, x_{i2}, \cdots, x_{ip}$, then the fitting criterion chooses the β_j to minimize $\sum_{i=1}^{n}(y_i - (\sum_{j=1}^{p} \beta_j x_{ij}))^2$ [24].

One side effect of using the LSQ criterion is that outliers (experimental values with a large error) tend to have a big effect on the derivation of β values. From one perspective, this is reasonable because the seriousness of an error in prediction is more than linear as the magnitude of the error increases. Although we would like to have all predicted values within a certain percentage error of the experimental value, it is more important to estimate values correctly when the execution time is in thousands of seconds than it is with millisecond run times. In order to minimize the effects of erroneous measurement we have manually removed outliers which were suspect. The technique of "ridge regression" could also be used which would allow some bias in the estimated β values in exchange for a potentially large decrease in variability in the presence of "wild" observations [22].

Our dynamic performance model uses counts of critical operations generated from the linearization module as predictor variables for multivariate analysis. The β values generated by the statistical package are estimates of the actual values for system parameters in a

System Parameter	Value (μsec)	Std. Err.	Significance
β_{Ops}	0.6001	0.0044	0.00000000
β_{VP}	15.2648	5.1386	0.00000000
β_{St}	367.8870	757.9475	0.02351422
β_{Bw}	2.3690	0.5341	0.00004550

Table 1: Output of statistical parameters for the nCUBE 3200 multicomputer.

particular parallel architecture. A number of real data-parallel applications are run on an existing parallel machine in order to fit the model. Since the algorithmic characteristics have been abstracted into simple operation counts, the statistical package can make predictions, with confidence intervals, for other parallel applications on the selected platform.

4.4 Experimental Results

The statistical model was fit using actual runs on the iPSC/860, nCUBE 3200 and Meiko CS-2 multicomputers. The S-PLUS software package was then used to predict performance for two fluid dynamics modeling applications not included in the fit process. The results indicate that β values derived from sample applications can generalize to future programs analyzed by this performance prediction methodology.

Our initial experiments were run with the nCUBE 3200 multicomputer. The model was fit using three applications (Gaussian elimination, matrix multiplication and a Shallow Water Model) with several different problem sizes for a total of 58 experimental runs. The Shallow Water Model solves the system of shallow water equations using a finite difference method. The resulting β values are shown in Table 1.

The coefficient of determination (Multiple R-squared term) for this regression is 0.9978 indicating that over 99% of the total variation in the response is explained by the fitted values [24]. Values in Table 1 are given in microseconds. Given the system clock cycle time of 125$n$sec, the β_{Op} value indicates that it takes approximately 5 clock ticks for an average operation. The β_{VP} value suggests that it takes approximately 120 cycles to set up a virtual processor emulation loop. The startup time of 368μsec is similar to that found in previous research on the nCUBE [25].

The standard error column in Table 1 is an estimate of how much the regression coefficient

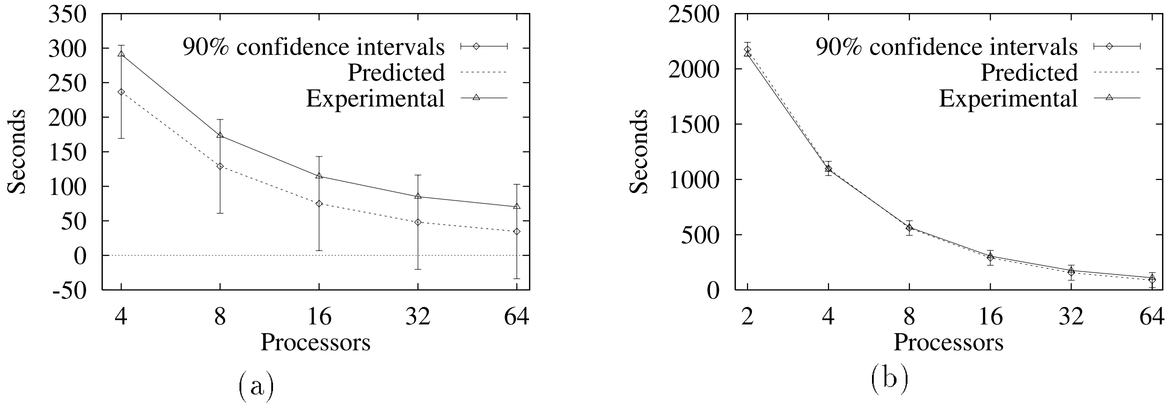


Figure 4: Predicted and experimental execution time in seconds for the Ocean Circulation Model on the nCUBE 3200 multicomputer with (a) 128 and (b) 640 segments in the east-west direction.

β will vary from sample to sample. If multiple samples of the same size were taken from the same population and used to calculate the regression equation, this would be an estimate of how much the regression coefficient would vary from sample to sample [26]. The large standard error value for the message startup cost indicates that a more detailed model needs to be investigated for this parameter. By analyzing standard error values, the quality of the dynamic performance prediction model can be improved. The "Significance" column describes the result of an $\mathcal{F}$-test to determine the probability that the variance accounted for by the coefficient could come from a $\mathcal{F}$ distribution. Small values indicate that the variable is important in explaining variance. All of the entries in this column indicate that the coefficients are highly significant in accounting for variance in the experimental data.

Figure 4 illustrates predicted output for the Ocean Circulation Model on the nCUBE 3200. The program models wind-driven circulation in a density-stratified ocean [19]. The problem scales up by increasing the number of segments modeled in the east-west direction. The vertical bars join the upper and lower twice-standard-error points, meant to represent approximately 90% confidence intervals for the mean response. Communication costs make up a higher fraction of total execution time for the smaller problem in Figure 4. Since the standard error value for communications is higher than that for computations, the confidence interval for the 128 segment problem is wider.

Figure 5 examines predicted output for the Shallow Water Model application on the

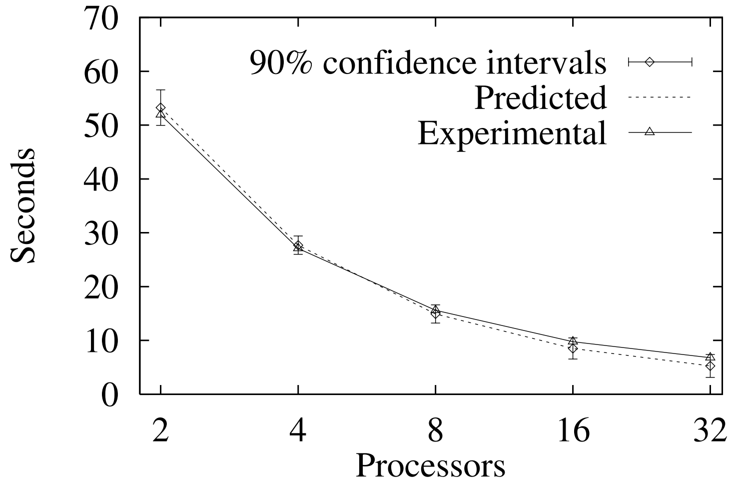
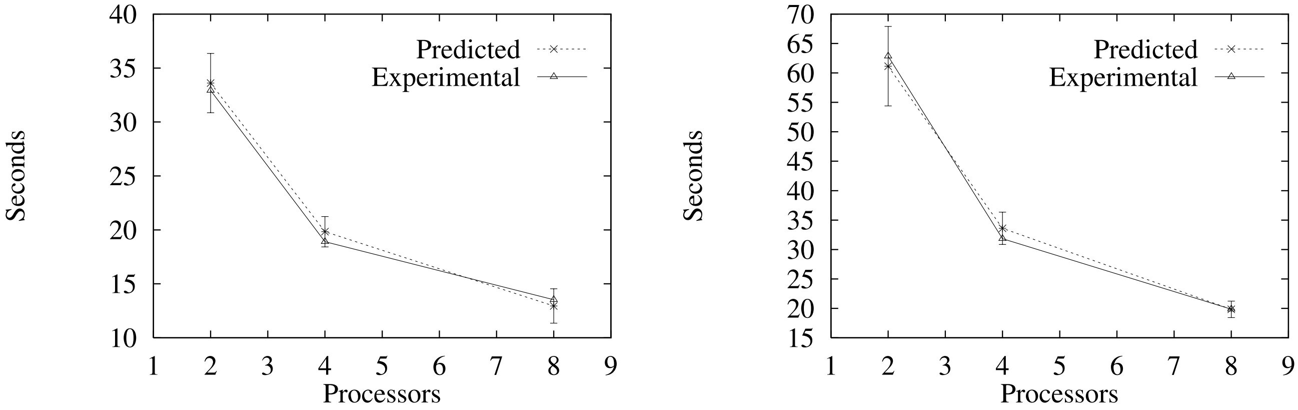


Figure 5: Predicted and experimental values for the Shallow Water Model with a 64×64 grid and 1200 iterations for an iPSC/860 multicomputer.

iPSC/860. The National Center for Atmospheric Research has developed this application for use in benchmarking the performance of MPP systems. The program solves a set of nonlinear shallow water equations in two horizontal dimensions [19]. For this experiment we rely exclusively on previous experimental data for execution times in order to fit the model. The instrumentation run for the applications was performed on a single processor workstation and yet accurate results were still obtained.

The Meiko CS-2 multicomputer consists of SPARC processors connected in an Omega network configuration [27]. Each node is equipped with two vector processors to improve floating point performance. A copy of the multi-user Solaris operating system executes on each node increasing the variability of successive runs of the same program on the machine. Figure 6 shows the results obtained for the Ocean Circulation Model with two different problem sizes. The experimental data has a much larger number of outliers than were found on the other two machines. Some of this variability can be attributed to the multi-user nature of the machine. The current implementation of Dataparallel C on the Meiko relies on libraries written for the iPSC/860 which use the NX message passing interface provided on the Meiko. This extra level of software indirection may also account for some of the inaccuracies in the model. Future work will focus on adapting this modeling technique to networks of workstations where high levels of variability exist in the message passing latency. This work on the Meiko is a first step in that direction.



(a) (b)

Figure 6: Predicted and experimental execution time in seconds for the Ocean Circulation Model on the Meiko CS-2 multicomputer with (a) 640 and (b) 1280 segments in the east-west direction. The error bars are 90% confidence intervals for the predicted values.

If designers and users of parallel systems are to benefit from the results of performance prediction, they must be able to determine how accurate the predictions are. The statistical methods described here allow confidence intervals to be placed on predictions in order to fulfill this requirement. The statistical information can be obtained in an automated fashion using statistical software packages such as S-PLUS [23]. Additional information from the statistical analysis can also guide the development of more accurate models.

5 Evaluation of Dynamic Modeling Techniques

Several applications were analyzed during the course of this research in order to validate dynamic performance prediction techniques. They represent a good cross section of the data-parallel programs which are used for scientific research including matrix multiplication, a linear system solver and finite difference solutions to differential equations.

For our error analysis we have used the classical error computation method:

$$Error = \frac{Predicted - Actual}{Actual}$$

A total of 246 executions of the validation suite applications were performed using different problem sizes and numbers of processors. The error contours shown in Figure 7 indicates that over 90% of the experimental runs achieve less than 40% error for all three machines.

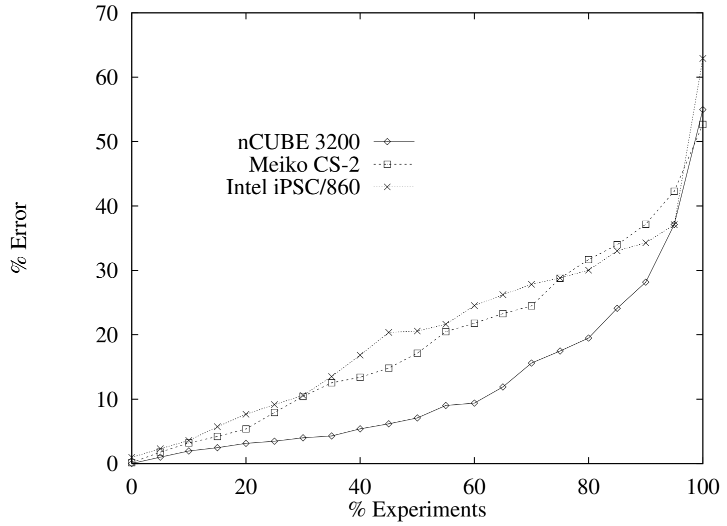


Figure 7: Percent error for experimental runs of applications in the validation suite.

System	Number of Experiments	σ	$\bar{t}$	C.O.V.	Average Error
nCUBE 3200	115	8.87 sec	105 sec	0.084	12%
Meiko CS-2	63	0.55 sec	6.36 sec	0.087	20%
iPSC/860	46	5.4 sec	12.7 sec	0.425	23%

Table 2: Experimental error values.

The standard deviation of errors

$$\sigma = \sqrt{\frac{SSE}{n-3}}$$

where the sum of squared errors

$$SSE = \sum_{i=1}^{n} e_i^2$$

can be examined to determine the quality of the model. The weighted average value of σ is 5.4 seconds. The weighted average of the mean $\bar{t}$ is 53.7 seconds. The ratio of standard deviation to the mean, or the coefficient of variation (C.O.V.) can also be used as a unitless measure of error. For the nCUBE and Meiko the standard deviation was less than 9% of the mean experimental value. One problem with the standard deviation of error is that it tends to be influenced disproportionately by the relative error of long experimental runs because of the squared nature of SSE. The average error of all validation runs indicates that the model is able to predict results at an acceptable level for the uses we have targeted. A summary of the accuracy analysis is shown in Table 2.

Given a validated model the performance of applications and architectures can be analyzed in detail. Our validation testing indicates that an accurate performance model can be obtained with minimal user intervention through leveraging existing software tools for symbolic computations and statistical analysis. The next section will describe how this prediction methodology can be used to achieve detailed performance analysis.

5.1 Analysis Using Performance Prediction Results

Many aspects of parallel performance have been examined from a purely theoretical perspective. Additional insights can be obtained from a detailed analysis of performance using an analytical model derived from an actual application. The following examples show how this performance prediction system has been used to analyze performance as the problem size and architecture are scaled. We also examine cost optimal metrics which can indicate algorithm-architecture compatibility.

A fundamental use of performance prediction results is in the area of performance debugging. Figure 8 illustrates the percentage of time spent in broadcasting the pivot row for the Gaussian elimination application on an nCUBE 3200. For extremely small problem sizes, the execution time is dominated by message startup costs for the communications. Since all communications in the application grow linearly with P, the percentage of time spent in this basic block is constant. As the problem size is scaled up the fraction of time grows logarithmically with P due to the number of messages required for the binomial tree broadcast algorithm. For larger problem sizes the communication ratio grows linearly with the number of processors.

The ability to predict the sensitivity of an algorithm to changes in system parameters is critical to determining its portability. As architects are able to predict the sensitivity of applications to changes in system parameters the efficiency of parallel hardware should increase. Since the result of this performance prediction system is a symbolic expression for execution time, the equation can be differentiated with respect to critical system parameters in order to view the effect modifications will have on performance as the problem size is scaled up. In Figure 9, the execution time was differentiated with respect to message startup cost where

$$Sensitivity = \frac{dt}{d\beta_{St}}$$

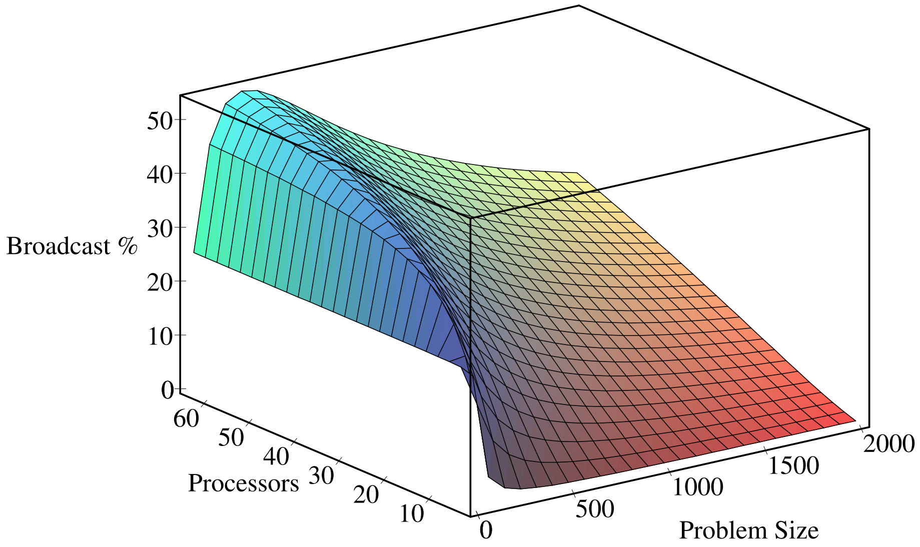


Figure 8: Percentage of execution time spent in broadcasting the pivot row for Gaussian elimination on an nCUBE 3200.

The vertical scale shows the increased execution time in seconds for each increase of 10μsec in message startup cost. The sensitivity increases linearly with problem size since a constantly growing number of communications must be performed as problem size increases. The sensitivity grows logarithmically as the number of processors increases due to the complexity of the binomial tree broadcast algorithm. Similar analysis can be performed to determine the effect of other system parameters.

5.2 Cost Optimal Analysis

Recent advances in the speed of workstations have motivated several systems vendors to introduce workstation clusters as parallel computing platforms. These systems are attractive because they deliver high levels of potential processing power for a reduced dollar cost. Through mathematical manipulations of the basic execution time equations produced by the symbolic performance prediction system, a cost optimal architecture for a given application can be determined.

For our analysis we have concentrated on the dollar cost of the CPU, on-chip cache and the

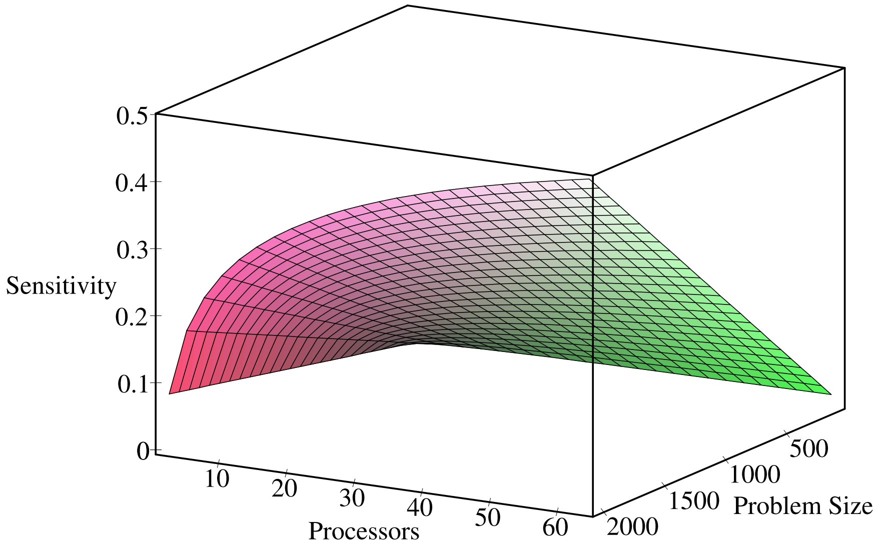


Figure 9: Sensitivity to changes in message startup cost for Gaussian elimination as a function of problem size and number of processors.

interconnection network. A more detailed analysis could be performed by users familiar with the cost structure available to their organization. With Ops_{seq} and Ops_{par} as the number of sequential and parallel operations performed in an application we define a price-performance metric (Φ) as

$$\Phi = \frac{\mathcal{M}}{\mathcal{C}}$$

where performance in MFLOPS

$$\mathcal{M} = \frac{Ops_{seq} + Ops_{par}}{\hat{t}} / 10^6$$

and cost in dollars

$$\mathcal{C} = COST_{CPU} + COST_{L1} + COST_{net}$$

In order to determine the dollar cost for communication networks we obtained quotes for high speed network connections using switched Ethernet, FDDI and ATM technology. Through connecting workstations to switched hubs, many of the bandwidth limitations caused by contention can be eliminated. We then fit a curve to these points and derive an expression for dollar cost as a function of network bandwidth

$$COST_{net} = \$800 + \$30,000 * (2^{(1000/\beta_B W)})$$

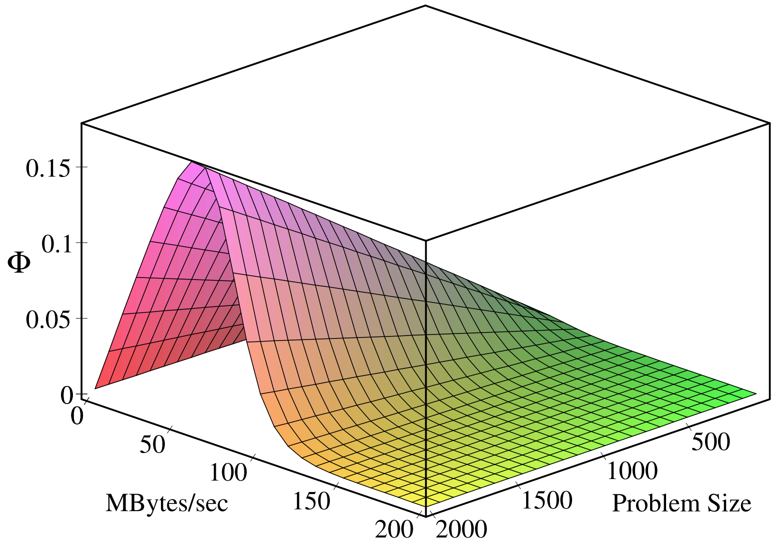


Figure 10: Predicted Φ for Gaussian elimination on 8 workstations as network bandwidth varies.

Figure 10 plots Φ as bandwidth and problem size vary. Our analytical results indicate that a network bandwidth of 50Mbyte/sec is near the cost optimal point for Gaussian elimination on a network of 8 contemporary workstations.

The value of $COST_{CPU}$ was computed in a similar manner by fitting a curve to the dollar costs of several commercial microprocessors

$$COST_{CPU} = \$50 + \$100 * 2^{100/\beta_{Ops}}$$

The exponential nature of the dollar cost functions makes sense from a practical standpoint. As the time to execute an instruction approaches zero, the dollar cost for the CPU approaches infinity. Figure 11 shows the price-performance plot for the Shallow Water Model when considering both CPU speed and network bandwidth.

In analyzing cost optimal cache configurations we will focus on the cache size instead of the β_{L1} value or the time necessary for a level one cache miss. The β_{L1} value may be dependent on the processor architecture, the disk subsystem and even the operating system and does not have a definite relationship to the dollar cost of the cache. As the size of the on-chip cache increases, the cost of the cache increases approximately as the cube of the cache

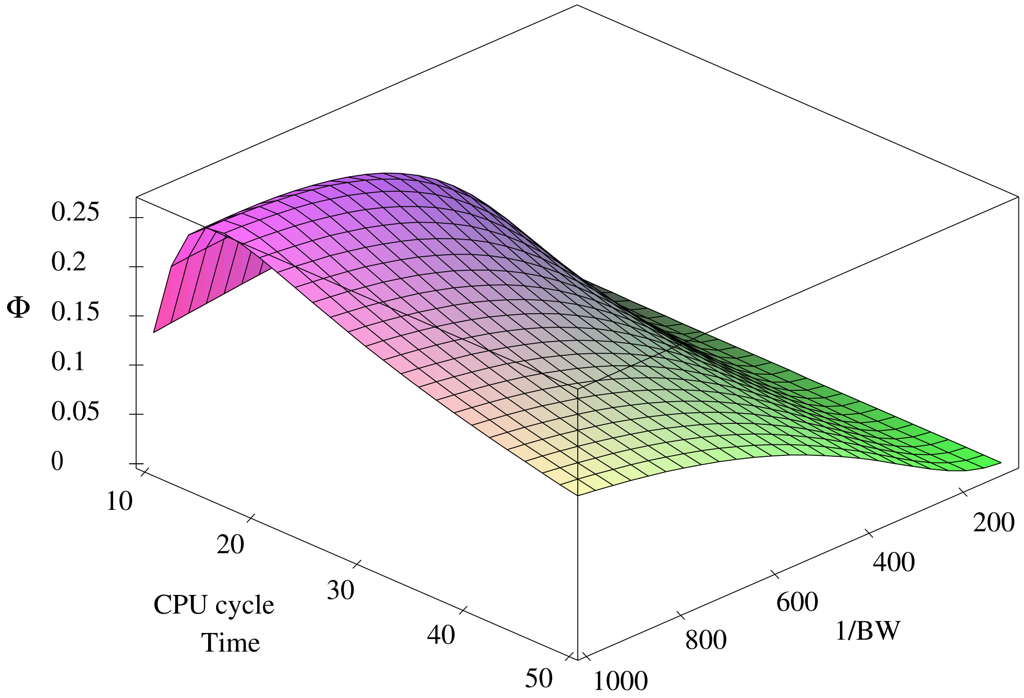


Figure 11: Predicted Φ for the Shallow Water Model as bandwidth and CPU speed vary.

size [28]. This occurs because of the increase in defects occurring in larger die sizes and the reduced number of dies which will fit on a wafer. We examined data from the PowerPC 601 chip in order to fit this general cost model to an actual implementation [29]. The PowerPC 601 has 32Kbytes of cache with a total area of $119mm^2$ and sells for approximately \$450. Since the cache consumes 31% of the die area we assume that the cost of this size of cache is approximately \$166. The resulting dollar cost for on-chip cache $COST_{L1}$ is

$$COST_{L1} = \$5 * 10^{-12} * L1Size^3$$

The price-performance curve for the Shallow Water Model as a function of level one cache size is shown in Figure 12. The steps in the graph occur when the cache size reaches a level where data in a virtual processor emulation loop all fits in cache. Once the cache size becomes large enough for all data to stay in cache additional increases in $L1Size$ will not result in additional performance increases. At this point the additional cost of larger cache reduces the Φ value.

The gradient of the price-performance equation $\nabla\Phi$ represents the slope of the function

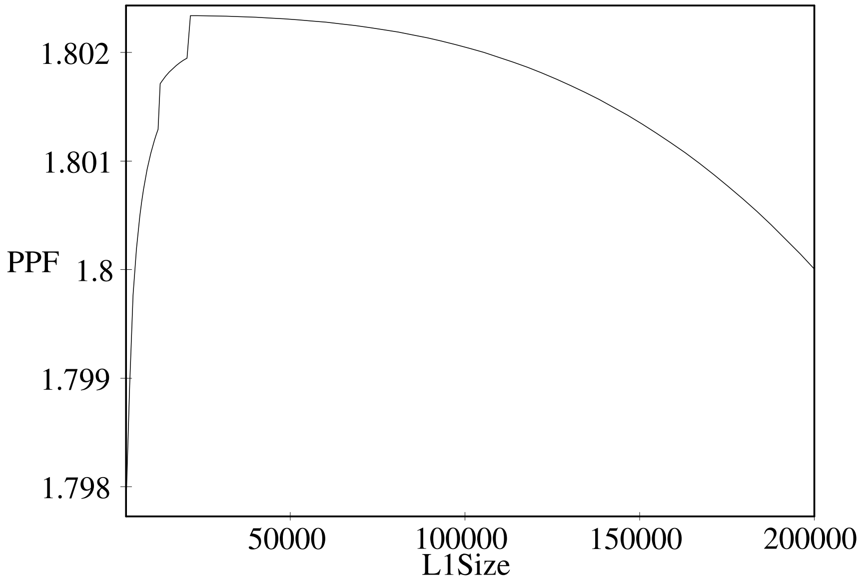


Figure 12: Price performance curve as cache size is varied for the Shallow Water Model.

$\Phi =$ in each dimension at all points in the graph.

$$\nabla\Phi = \begin{bmatrix} \frac{d\Phi}{d\beta_{Ops}} & \frac{d\Phi}{d\beta_{BW}} & \frac{d\Phi}{dL1Size} \end{bmatrix}$$

Given $\nabla\Phi$, system designers can determine the dimension where enhancements to system parameters will result in the most progress towards the cost optimal point for the system. By setting $\nabla\Phi = \mathbf{0}$ and solving for the cost sensitive variables, the maximum value can be determined if Φ has a single maximum. If the Φ surface is not unimodal the maximum point is found through an exhaustive search of practical values for the system parameters in question. The Maple system generates the gradient automatically and can solve the simultaneous equations for the cost optimal point for some Φ equations.

Theorem 5.1 *If $\mathcal{C}$ is twice differentiable with the second gradient universally non-negative and $\mathcal{M}$ is twice differentiable with the second gradient universally non-positive, then the function $\Phi = \mathcal{M}/\mathcal{C}$ is unimodal and there is a non-trivial maximum.*

Proof:

If $\nabla^2\mathcal{C}$ is non-negative then the surface of $\mathcal{C}$ is concave up with the slope increasing or remaining constant as system parameters are enhanced. The cost function for many subsystems will conform to this condition. As the CPU cycle time or the time to transmit a byte across the network decreases linearly, the slope of the increase in cost for the subsystem will be strictly increasing. The function $\mathcal{C}$ is also strictly increasing as system parameters are improved.

When $\nabla^2\mathcal{M}$ is non-positive then the surface of $\mathcal{M}$ is concave down with the slope decreasing as the speed of a subsystem is increased. As a particular β value is improved its contribution to the execution time will decrease with respect to other parameters. If the $\mathcal{M}$ surface is concave up and increases faster than the $\mathcal{C}$ function in any dimension then the optimal value for β is either zero or ∞. This trivial case is not attainable for real implementations. Regions of the Φ surface where $\nabla\mathcal{M}$ is negative can also be ignored since a cost optimal point will not exist in regions where the slope of performance is negative.

Given these assumptions, the following conditions can be specified for the Φ surface in the region where cost optimal conditions can occur.

$$\mathcal{C} > 0, \mathcal{C}' > 0, \mathcal{C}'' > 0$$

and

$$\mathcal{M} > 0, \mathcal{M}' > 0, \mathcal{M}'' < 0$$

Let $\mathbf{x_0}$ be the location of a local maximum where

$$\Phi'(\mathbf{x_0}) = \mathbf{0}$$

Assume that there is another local maximum at $\mathbf{x_2}$ where $\Phi(x_2) > \Phi(x_0)$. A local minimum must exist between $\mathbf{x_0}$ and $\mathbf{x_2}$ at $\mathbf{x_1}$ (see Figure 13).

In the region between the two maxima $\Phi'|_{x_0}^{x_1} < 0$, $\Phi'|_{x_1}^{x_2} > 0$, and $\Phi''|_{x_0}^{x_2} > 0$ is concave up.

$$\begin{aligned}
\Phi &= \frac{\mathcal{M}}{\mathcal{C}} \\
\Phi' &= \frac{\mathcal{M}'\mathcal{C} - \mathcal{M}\mathcal{C}'}{\mathcal{C}^2} \\
\Phi'' &= \frac{[(\mathcal{M}''\mathcal{C} + \mathcal{M}'\mathcal{C}') - (\mathcal{M}'\mathcal{C}' + \mathcal{M}\mathcal{C}'')]\,\mathcal{C}^2 - [2\mathcal{C}\mathcal{C}'\,(\mathcal{M}'\mathcal{C} - \mathcal{M}\mathcal{C}')]}{\mathcal{C}^4}
\end{aligned}$$

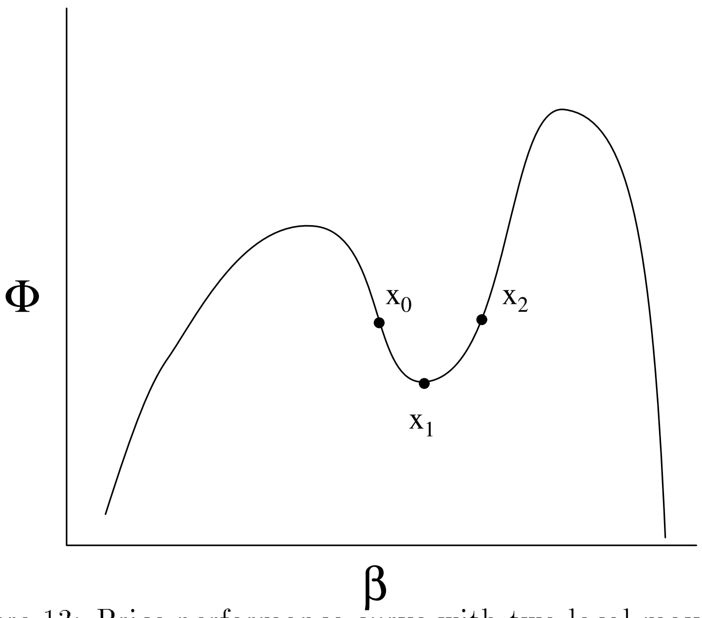


Figure 13: Price performance curve with two local maxima.

$$
\begin{aligned}
&= \frac{\mathcal{M}''\mathcal{C} - \mathcal{M}\mathcal{C}''}{\mathcal{C}^2} - \frac{2\mathcal{C}'}{\mathcal{C}}\left(\frac{\mathcal{M}'\mathcal{C} - \mathcal{M}\mathcal{C}'}{\mathcal{C}^2}\right) \\
&= \frac{\mathcal{M}''\mathcal{C} - \mathcal{M}\mathcal{C}''}{\mathcal{C}^2} - \frac{2\mathcal{C}'}{\mathcal{C}}(\Phi') \qquad (1)
\end{aligned}
$$

The first term of Equation 1 is strictly negative since $\mathcal{M}'' < 0$ and $\mathcal{C}'' > 0$. If $\Phi''|_{x_0}^{x_2} > 0$ then the second term of Equation 1 must be positive, which can only occur if $\Phi'|_{x_0}^{x_2} < 0$. This is a contradiction since by construction $\Phi'|_{x_1}^{x_2} > 0$.

□

Table 3 specifies cost optimal values for the applications we have studied in this research. The problem sizes were scaled for an execution time of 1000 seconds on 1000 processors. In order to compute these values, we developed a program which took as input the symbolic equation for Φ. A hill climbing technique was then used to calculate the maximum point in the Φ surface.

The Gaussian elimination application favors a higher bandwidth network with slightly slower CPU speed than the other applications. The applications using finite difference methods all have bandwidth requirements which could be satisfied with a switched Ethernet configuration if the switch could be extended to allow connections for 1000 processors. The overhead for message startup time in a TCP/IP environment is the limiting factor for

Application	Problem Size	β_{Ops} in $n$sec	$L1Size$ in bytes	β_{BW} in $n$sec/byte
Gauss	$3.9 * 10^4$	18	21310	851
Matrix	$4.3 * 10^5$	16	5090	1751
Jacobi	$1.8 * 10^5$	15	2390	20000
Layer	$2.4 * 10^8$	15	7640	20000
Ocean	$9.3 * 10^7$	15	3630	20000
Shallow	$2.8 * 10^4$	16	12030	3181

Table 3: Cost optimal values for system parameters for applications in validation suite.

these applications rather than bandwidth. If an expression were available which related message latency to the cost of a system, an optimal startup time could also be specified. The cache sizes specified suggest that some of these applications could make effective use of more than the 8 Kbytes of data cache available on many microprocessors.

Additional analysis could be performed for more complex network configurations using dollar cost estimates developed specifically for these topologies. Computer architects can use the symbolic equations for execution time combined with cost functions for critical system parameters in order to improve performance in the most effective way.

6 Conclusions

Performance prediction can be used in performance debugging of parallel applications and in making architectural improvements for multicomputer hardware and system software. Using predicted performance results for existing commercial MPP systems, purchasing agents can also make informed procurement decisions.

The dynamic performance prediction methodology developed here allows performance analysis to be performed on scalable parallel applications. This important class of programs imposes different requirements on an analysis system than traditional, constant size problems do. The simultaneous use of analytical and sampling techniques allows this technique to accomplish the goals of ease of use and accuracy which are important for performance prediction models.

If decisions are to be made on the basis of predicted behavior, the accuracy of the estimates must be specified. By using multivariate statistical analysis to determine critical system parameters, confidence intervals can be determined for predicted results. This statistical characterization can also assist in improving the accuracy of the model. By automatically determining the values for system parameters, the overall ease of use is also improved.

By preserving the symbolic equations which characterize the detailed behavior of each basic block in the application, this methodology can be used to mathematically evaluate cost optimal architectures for a given application. The estimated performance of an application can be viewed across continuous values of each hardware parameter, allowing quantitative measures of application portability to be performed.

Future work will focus on improving the accuracy of the model and using the results to explore different parallel architectures. We also plan to develop portability measures similar to the isospeed metric [30] which can be quantified by analyzing the sensitivity of applications to changes in system parameters. Several alternatives are also being investigated to incorporate the effects of load imbalance into the performance prediction system.

Dynamic performance prediction has been shown to be effective in meeting the goals suggested for an analytical model. The use of this type of performance analysis system by system architects and programmers should increase the efficiency of MPP systems in general and scalable applications in particular.

References

[1] Committee on Physical, Mathematical, and Engineering Sciences Federal Coordinating Council for Science, Engineering, and Technology, *Grand Challenges 1993: High Performance Computing and Communications.* Washington, D.C.: National Science Foundation, 1993.

[2] G. Bell, "Scalable, parallel computers: Alternatives, issues and challenges," *International Journal of Parallel Programming*, vol. 22, no. 1, pp. 3–46, 1994.

[3] S. Hackstadt and A. Malony, "Next-generation parallel performance visualization: A prototyping environment for visualization development," in *Proceedings of Parallel Architectures and Languages Europe (PARLE), Athens, Greece*, July 1994.

[4] C. M. Pancake, "Why is there such a mis-match between user needs and tool products?." Keynote address, 1993 Workshop on Parallel Computing Systems, Keystone, Colorado., April 1993.

[5] P. H. Smith, "System software and tools for high performance computing environments." Report on the Findings of the Pasadena Workshop, April 14-16 1992.

[6] F. Sotz, "A method for performance prediction of parallel programs," in *Proceedings of CONPAR 90 - VAPP IV, Zurich, Switzerland*, September 1990.

[7] P. Mehra, M. Gower, and M. Bass, "Automated modeling of message-passing programs," in *Proc. Int'l. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 94), Durham, NC*, pp. 187–192, Jan 1994.

[8] G. Haring and A. Ferscha, "Performance oriented development of parallel software with capse," in *Proceedings of the Workshop on Environments and tools for Parallel Scientific Computing, Blackberry-Inn, Walland/Tennesee*, May 25-27 1994.

[9] T. Fahringer and H. P. Zima, "A static parameter based performance prediction tool for parallel programs," Tech. Rep. ACPC/TR 93-1, University of Vienna Department of Computer Science, January 1993.

[10] V. Balasunderam, G. Fox, K. Kennedy, and U. Kremer, "A static performance estimator to guide data partitioning decisions," *SIGPLAN Notices*, vol. 26, pp. 213 – 223, July 1991.

[11] M. Gupta and P. Banerjee, "Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 179–193, March 1992.

[12] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing.* Redwood City, California: Benjamin/Cummings Publishing Company, Inc., 1994.

[13] M. E. Crovella and T. J. LeBlanc, "The search for lost cycles: A new approach to parallel program performance evaluation," Tech. Rep. 479, Computer Science Department, University of Rochester, Dec. 1993.

[14] M. J. Clement and M. J. Quinn, "Analytical performance prediction on multicomputers," in *Proceedings of Supercomputing '93*, pp. 886–905, November 1993.

[15] G. Fox, "What have we learnt from using real parallel machines to solve real problems?," Tech. Rep. C3P-522, Caltech, December 1989.

[16] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, pp. 532–533, May 1988.

[17] J. K. Hollingsworth, B. P. Miller, and J. Cargille, "Dynamic program instrumentation for scalable performance tools," in *Scalable High Performance Computing Conference, Knoxville, Tennessee*, May 1994.

[18] J. R. Larus, "Abstract execution: A technique for efficiently tracing programs," Tech. Rep. TR912, University of Wisconsin Department of Computer Science, February 1990.

[19] P. J. Hatcher and M. J. Quinn, *Data-Parallel Programming on MIMD Computers.* Cambridge, Massachusetts: The MIT Press, 1991.

[20] A. D. Malony, "Event-based performance perturbation: A case study," *SIGPLAN Notices*, vol. 26, July 1991.

[21] K. Gallivan, D. Gannon, W. Jalby, and A. Malony, "Experimentally characterizing the behavior of multiprocessor memory systems: A case study," *IEEE Transactions on Software Engineering*, vol. 16, February 1990.

[22] R. J. Harris, *A Primer of Multivariate Statistics.* New York: Academic Press Inc., 1985.

[23] Statistical Sciences Inc., *S-PLUS Users Manual*, September 1991.

[24] J. M. Chambers and T. J. Hastie, *Statistical Models in S*. Pacific Grove, California: Wadsworth & Brooks/Cole Advanced Books & Software, 1992.

[25] M. J. Clement, M. J. Quinn, and B. Baxter, "Medium grain size applications on distributed memory multicomputers," Tech. Rep. 93-80-13, Department of Computer Science, Oregon State University, September 1993.

[26] J. Joseph F. Hair, R. E. Anderson, and R. L. Tatham, *Multivariate Data Analysis with Readings*. New York: Macmillan Publishing Company, 1987.

[27] Meiko World Incorporated, *Computing Surface 2 Overview Documentation Set*, 1993.

[28] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach*. San Mateo, California: Morgan Kaufmann Publishers, Inc., 1990.

[29] T. Thompson, "Power," *Byte*, August 1993.

[30] X.-H. Sun and D. T. Rover, "Scalability of parallel algorithm-machine combinations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 599–613, June 1994.

Index Terms — Performance prediction, scalability, data-parallel, performance debugging, multicomputer, modeling.

Preferred Address

Mark J. Clement
Computer Science Department
3361 TMCB
Brigham Young University
Provo, Utah 84602
clement@cs.byu.edu

Footnotes

Affiliation of Authors

M. Clement is with the Computer Science Department, Brigham Young University, Provo, Utah 84602, USA; e-mail: clement@cs.byu.edu.

M. Quinn is with the Department of Computer Science, Oregon State University, Corvallis, Oregon 97331-3202, USA; e-mail: quinn@research.cs.orst.edu.

This research was supported by Intel Foundation and NSF grants ASC-9208971 and CDA-9216172.

Biographies

MARK CLEMENT is an assistant professor of computer science at Brigham Young University. He received the B.S. and M.S. degrees in electrical engineering from Brigham Young University in 1985 and 1989 respectively. He was a principle software engineer at Digital Equipment Corporation before earning his PhD in computer science from Oregon State University in 1994. His research interests include parallel computer architecture, parallel systems modeling and parallel performance evaluation.

MICHAEL J. QUINN is an associate professor of computer science at Oregon State University, author of *Designing Efficient Algorithms for Parallel Computers* (McGraw-Hill, 1987) and *Parallel Computing: Theory and Practice* (McGraw-Hill, 1994), and coauthor of *Data-Parallel Programming on MIMD Computers* (MIT Press, 1991). He was a software engineer at Tektronix before earning his PhD in computer science in 1983 from Washington State University. From 1983 to 1989 he was an assistant professor of computer science at the University of New Hampshire.