AN ABSTRACT OF THE DISSERTATION OF

Mark J. Clement for the degree of Doctor of Philosophy in Computer Science presented on July 25, 1994.

Title: Analytical Performance Prediction of Data-Parallel Programs

Redacted for privacy

Abstract approved: ___

Michael J. Quinn

A combination of static and dynamic performance prediction techniques can be used effectively to address the problems of performance debugging, architectural improvement, machine selection and compiler optimization for data-parallel programs on multicomputers. This research develops two analytical models which account for the effects of CPU execution time, cache behavior and message passing overhead for multicomputers. These factors have been shown to be significant in determining performance on distributed memory systems. The first model performs static analysis on Dataparallel C source code in order to determine critical algorithmic parameters which can be used to predict performance when the problem size is fixed. The second model uses information obtained at compile time along with a single instrumentation run to generate a symbolic equation for execution time. This model can be used to predict performance even when the problem size varies. By leveraging technology from the Maple symbolic manipulation system [15] and the S-PLUS [89] statistical package we implement a system which presents users with critical performance information necessary for performance debugging, architectural enhancement, and procurement of parallel systems. The usability of these results is improved by 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.

Analytical Performance Prediction of Data-Parallel Programs

by

Mark J. Clement

A DISSERTATION
submitted to
Oregon State University

in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

Completed July 25, 1994
Commencement June 1995

Doctor of Philosophy dissertation of Mark J. Clement presented on July 25, 1994

APPROVED:

Redacted for privacy

Major Professor, representing Computer Science

Redacted for privacy

Chair of Computer Science Department

Redacted for privacy

Dean of Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Redacted for privacy

Mark J. Clement, Author

ACKNOWLEDGMENTS

Much of the work presented in this thesis originated from stimulating discussions with Dr. Michael Quinn. His encouragement and direction have been invaluable in formulating and polishing the ideas which I wanted to pursue.

Thanks are also due to Intel Corporation and the staff of its Supercomputer Systems Division. Their insights into industrial requirements for performance prediction motivated the design and specifications of this research. Special thanks go to Brent Baxter whose direction during a summer intern experience gave me exposure to several relevant issues in this field. Without the financial support provided by Intel Foundation, Sequent Computer Systems and the Oregon Advanced Computing Institute, I would not have been able to complete this research in a timely manner.

Comments from other researchers in the field have also helped me to focus on relevant issues. Discussions with Mark Crovella from the University of Rochester have helped to crystallize my understanding of different approaches used in this field.

The support of my wife Laura and my children Kendell, Nathan, Spencer, Sarah and Rebecca has been greatly appreciated. They have born the brunt of many long nights and Saturdays spent in the office instead of on the baseball field. Laura has also contributed significantly to the quality of this work during the proofreading process.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Computational experiments have played a key role in making recent advances in several scientific and engineering disciplines [11, 16, 17, 97]. Several implementations of "Grand Challenge" problems require far more processing power than any sequential processor can deliver [23] . 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 through 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 [4, 7, 48]. The causes of this low level of efficiency can be traced to unbalanced hardware systems and inefficient algorithmic implementations.

1.1 Problem Statement

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 not able 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. Extensive work has been performed to determine theoretical bounds on the performance of various parallel architectures [1, 28, 56, 77, 79, 85]. Similar work has explored routing algorithms for these diverse architectures [8, 53, 64]. Each of these implementations claims to be optimal under certain assumptions about the nature of parallel applications. Consensus in the microprocessor industry on the benefits of RISC processors came only after exhaustive analysis of instruction traces of real sequential programs. Similarly, progress in achieving a convergence for parallel architectures will require attention to the characteristics of production parallel codes.

- **Machine Selection**

  The high performance computing environment has changed significantly with

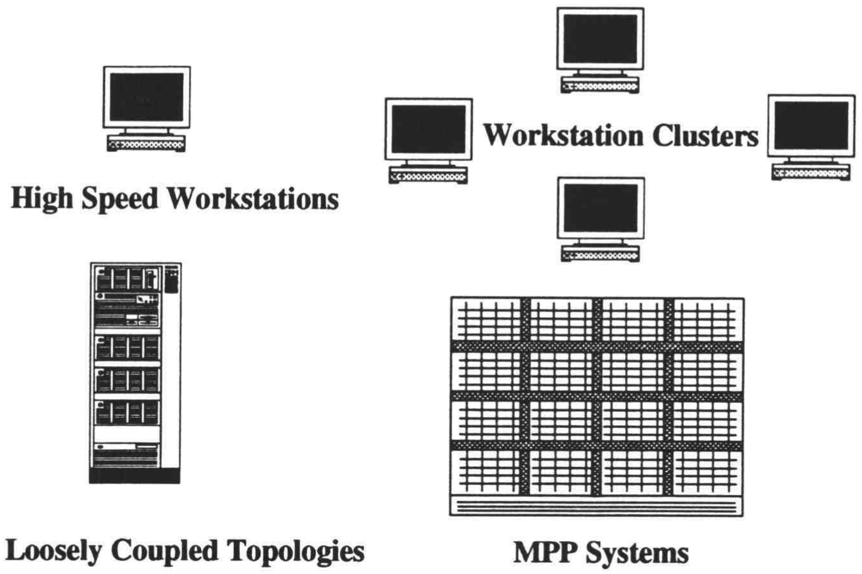


Figure 1.1. Diverse architectures available to a single researcher.

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. Figure 1.1 illustrates the machines that may be available to a single researcher. Selecting the appropriate computing platform to achieve efficient execution can be a difficult task. In order to make efficient use of a parallel machine, the user must also determine the optimal number of processors to use to maximize efficiency [37]. There is little information to guide the user of a parallel machine into making the correct selection.

- **Compiler Optimizations**

  Compilers for high level parallel languages are often not able to choose optimizations which will result in the best performance for an application. When optimizations are performed on sequential code, the compiler can be fairly certain that the changes will improve performance. In order for parallel compilers to make intelligent decisions, they must be provided with detailed information regarding the performance tradeoffs for alternate transformations [3, 27, 93].

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 [10]. 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 [43]. 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 [76, 87]. Performance prediction tools have been identified as an important technology in achieving the solution to grand challenge problems [23].

Dan Reed, a member of the performance analysis team for the National Consortium for High Performance Computing (NCHPC), has said:

> Performance analysis has both short-term and long-term goals. Typical short-term goals are to help write and tune programs that run fast for current problems on current computer systems, and to establish procurement criteria. One long-term goal is to produce programs that will also perform well in the future (e.g., for larger or longer problems on computer systems with more processors or different computation/communication tradeoffs). A complementary long-term goal is

to help design computer systems that will effectively support future applications.

All of these goals require determining the performance characteristics of application programs and computer systems. However, currently available tools and techniques provide only a subset of the capabilities needed to meet these goals. These tools are generally restricted to empirical measurements of a particular combination of code, system, and sample problems. Little or no support is provided for developing predictive models of application performance, or even for acquiring the information to construct such models. Additional work is needed to characterize massively parallel systems and extract application requirements, as well as to construct and validate models [81].

High level parallel languages are essential in making parallel processors feasible for large programming projects. They allow the program to be written in a machine independent manner, and abstract away the complexity of explicit message passing. David Kuck, a noted authority on computer architecture, has said:

HPCCI has distributed memory MPPs as its cornerstone architectural component .... Since the 1960's, the generally acceptable computer language level has risen from the machine level to the point where PC users now have problem-solving environments that do not require users to be programmers at all, but instead users may express themselves in terms of their own disciplines. Parallel processing cannot succeed by attempting to reverse this historical market force. ... if parallel processing is to emerge from its current niche market and become a practical technology it is essential that architectures be improved [58].

The Dataparallel C programming language provides a SIMD model of parallel programming with explicit parallel extensions to the C language [46]. Performance tools which are linked to a compiler for a high level parallel language have access to algorithmic information which can improve the accuracy of their analysis. Any methodology which addresses the performance problems for multicomputers should be tailored to applications written in high level languages.

1.2 Research Contributions

This research develops two predictive analytical models which are used to provide solutions to the problems we have described. We use these models to implement two performance prediction systems in order to show that the models are feasible. The first model analyzes application source code in order to estimate speedup for static programs on different hardware systems. The second model uses an instrumented run of the application, along with source code analysis, to provide detailed predictions of execution time for scalable applications. Both models use algorithmic information extracted from the source code of existing Dataparallel C applications. The concepts developed here can also be extended to other data-parallel languages.

Static applications solve problems with a fixed number of data items. The number of computations performed also remains fairly constant as the number of processors varies. For many of these problems compile time analysis of the source code is sufficient to predict the speedup attainable by a given parallel architecture [41]. Absolute execution time is difficult to estimate with static analysis, but the relative number of communications and computations can provide fairly accurate estimates of speedup values. Our research into static performance prediction concentrates on the relative speedup attained on a fixed problem size as the principle performance metric. Static analysis can provide rapid feedback to an optimizing compiler on the relative benefit of alternate transformations. It can also be used to predict performance for static applications with too many computations to admit an instrumentation run.

When additional processing elements are added to a computation, scalable applications can increase the problem size in order to maintain efficient execution. Speedup cannot be used as a performance metric for these applications since the number of computations will change as the number of processors vary. The performance prediction methodology used for scalable applications uses algorithmic information derived from the source code of a high level parallel language and an instrumentation run of the application with a small problem size. This information is combined with an analytical hardware model in order to derive an exact equation for execution time as the number of processors and the problem size vary. Since this modeling technique uses run-time data, it can be viewed as a dynamic performance prediction method. Through maintaining a symbolic representation of the performance equation, algebraic manipulation packages such as Maple [15] can be used to analyze and visualize the performance of an application. The equations for execution time can be differentiated in order to determine sensitivity to changes in variables and to determine cost optimal points for hardware architectures.

Through focusing on algorithmic variables in the dynamic model, performance debugging can be performed in order to create applications which will be efficient on a target architecture and more portable across architectures of interest to the user. Given a set of performance data from important applications, parallel system architects can determine which modifications to current designs will result in the largest improvements in performance for real programs. When parameters for actual parallel machine are specified, the dynamic model can assist in selecting the appropriate architecture and number of processors for an application.

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. Through utilizing multivariate statistical techniques, this research can specify a confidence interval for predictions. It is my thesis that a combination of static and dynamic performance prediction techniques can be used effectively to address the problems of performance debugging, architectural improvement, machine selection and compiler optimization for data-parallel programs on multicomputers.

1.3 Dissertation Organization

In Chapter 2 previous work in performance prediction is discussed and contrasted with the research presented here. The static analytical model is described in Chapter 3 and results are illustrated for several applications. Chapter 4 describes the implementation of the dynamic performance model and specifies the importance of the scalable applications. The statistical techniques developed in Chapter 5 are shown to be useful in creating confidence intervals for predicted values. Several applications are analyzed in Chapter 6 in order to show the utility of dynamic performance prediction. The accuracy of the model is also analyzed and a method is devised for estimating cost optimal values for system parameters. Finally, in Chapter 7 the results are summarized and future directions for this research are outlined. Table 1.1 and Table 1.2 explain the notation which we will use in the remainder of the thesis.

Static Model	
P,p	Number of processors
VP	Virtual Processor
N	Problem Size
$S(p)$	Speedup on p processors
$\mathcal{U}(p)$	Utilization of p processors $(S(p)/p)$
T_1	Execution time on a single processor
T_s	Execution time of sequential code
T_p	Execution time of parallel code
$\sigma(p, \tau)$	Communication overhead function
τ	Communication network topology
C_ϕ	Cost for an operation
T_ϕ	Time to execute a floating point instruction
$\mathcal{X}_\lambda$	Number of VP emulation loops
T_λ	Time spent in VP emulation loops
$\mathcal{X}_s$	Number of operations in sequential code
$\mathcal{X}_p$	Number of operations in parallel code
T_c	Message startup time
$\mathcal{X}_c$	Number of communications
$\mathcal{C}_c$	Normalized message startup cost
$\mathcal{X}_m$	Number of uncached memory accesses
$\mathcal{C}_m$	Normalized cache miss cost

Table 1.1. Notation used for static model.

Dynamic Model	
$\hat{t}$	Predicted execution time
t	Experimental execution time
e	Error
T_{seq}	Time spent in sequential execution
T_{par}	Time spent in parallel execution
$SS(p)$	Scaled Speedup
$E(p)$	Efficiency ($E(p) = SS(p)/p$
$Count_{seq}$	Number of sequential operations
$Count_{par}$	Number of parallel operations
$\mathcal{X}_{Ops}$	Number of operations
β_{Ops}	Time in *n*secs for CPU operation
$Count_{VPloops}, \mathcal{X}_{VP}$	Number of VP emulation loops
β_{VP}	Time in *n*secs for VP emulation loop
$Count_{L1Miss}, \mathcal{X}_{L1}$	Number of first level cache misses
β_{L1}	Time in *n*secs for first level cache miss
$Count_{L2Miss}, \mathcal{X}_{L2}$	Number of second level cache misses
β_{L2}	Time in *n*secs for second level cache miss
$Count_{CStart}, \mathcal{X}_{St}$	Number of message startup times
β_{St}	Message startup time in *n*secs
$Count_{CBand}, \mathcal{X}_{Bw}$	Number of bytes transmitted
β_{Bw}	Time in *n*secs to transmit one byte of data
σ	Standard deviation of errors
C.O.V.	Coefficient of Variation
$\mathcal{M}$	MFLOPS of performance attained
$\mathcal{C}$	Cost in dollars
Φ	Price-performance metric ($\frac{\mathcal{M}}{\mathcal{C}}$)

Table 1.2. Notation used for dynamic model.

Chapter 2
Related Work

Performance analysis is a key step in the design and selection of computer hardware and software. Many of the techniques which have traditionally been used in optimizing sequential machines have been applied to multicomputers. Performance prediction plays a much more important role with these parallel machines because of the large variance in performance between different applications on a single system and different hardware platforms for the same software. A simple profile of where time is being spent in a parallel application may give little insight into where improvements should be made in order to improve overall performance. The multidimensional aspect of performance analysis on multicomputers imposes certain constraints on the types of tools which can be effectively used in this environment. This chapter will examine four characteristics of performance modeling tools which are of pivotal importance with reference to MPP systems. The research performed in this dissertation focuses on two important problems which have not been adequately addressed by other methodologies.

In the remainder of the chapter we will differentiate our work from other research in this field through analyzing the different design decisions which must be made in developing a performance model. Previous work in performance prediction can be categorized in the following ways:

- **Queueing Theoretic Models vs. Real Applications**

  Sequential analysis has traditionally relied heavily on representing workloads with Markov models using random distributions. This modeling technique has been extended to parallel systems with limited success. Other analysis methodologies use real applications to build a performance model. These methods either examine the source code of real programs, require the programmer to write the code in an alternate modeling language or use sampling data from an instrumented run in order to adequately characterize the application.

- **Simulation vs. Analytical Methods**

  Approaches involving simulations attempt to emulate the behavior of the hardware as an application is executing. These methods are generally computationally intensive and often are able to examine more detailed aspects of the execution. Analytical models attempt to derive equations for the behavior of a system. Variables in these equations are then modified to predict performance for other environments.

- **Automatic vs. Manual Systems**

  Ease of use is an important consideration in developing performance analysis techniques. Some methodologies require the user to rewrite applications in new modeling languages. Manual techniques such as this will not be used even if they promise superior results.

- **Static vs. Dynamic Analysis**

  Methodologies which rely totally on compile-time information for their characterization of an application can be termed "Static". Other modeling techniques which require instrumentation runs are "Dynamic" in nature. Static analysis is preferable from a usability viewpoint since results can be predicted immediately and there is no need for access to an actual machine. Dynamic information, however, can significantly improve the accuracy of the prediction.

2.1 Queueing Theoretic Models vs. Real Applications

Queueing theory is a key modeling technique for the performance analysis of batch systems and computer networks. It has also been applied to the parallel environment. Queueing theory is used to estimate the time that jobs spend in various queues in the system [52]. These times can be used to predict the mean response time for a task. The workload and execution time are modeled using random variables with distributions which can be determined from studying existing computing environments. The principle drawbacks to queuing theoretic models are accuracy and usability. Predicted results may have little or no correlation to experimental data for a real application. Detailed analysis of the time spent in various blocks of an application is also impossible so tools based on these models cannot be used for performance debugging. The believability of results derived from using statistical workloads is also low. Users are more likely to trust predicted results from simulated executions of actual programs than they are to believe queuing theoretic models [52]. For this reason models which use real applications in their analysis process hold a big advantage. The models we develop in this research are based on actual programs. Although it is more difficult to extend these models to average system performance, they are more accurate and can be used for performance debugging, which is one of the key goals of this research. Developing queueing theoretic models is also difficult for most programmers and violates the ease of use criterion which we will examine later.

Several different approaches have been taken in developing queuing theoretic models. Akyildiz developed a hierarchical system which separated the global process model from process communications. The work attempts to model applications which communicate with each other using "SEND" and "WAIT" operations [2]. Results for utilization, mean response time and communication overhead are validated through a simulation of a system. Specific applications are not modeled and no attempt is made to compare predicted results with non-simulated experimental data. Similar work has also been performed by Sotz [88].

Mac [66] uses Markov models to predict performance for series parallel task programs. Predictions are within 10% of simulated results but the workload is all synthetic. Kapelnikov et al. have used an amalgamation of queuing theoretic models of physical system characteristics combined with graph models of actual applications in order to analyze the performance of parallel programs with looping constructs [55]. This technique incorporates actual application code to alleviate some of the drawbacks of queuing theoretic approaches, but the authors still conclude that the accuracy of the system is acceptable for preliminary evaluation of various system designs, but that further detailed analysis is necessary to draw specific conclusions. Research by Dimpsey and Iyer [29] attempts to use statistics from a cluster of real applications as input to their model.

The cost optimal analysis performed by Born et al. assumes that messages are generated so that the time between successive messages is an exponential random variable [9]. This model is attractive in terms of its ability to compare and contrast various network topologies, but it is limited in its accuracy and ability to predict execution time for a given application. Kleinrock has also investigated models to determine the optimal numbers of processors to use given specified arrival rates [57].

Queuing theoretic and statistical models of programs do not allow users to make critical performance debugging and hardware selection decisions. Table 2.1 summarizes the characteristics of these models. Because of our requirements of ease of use and accuracy for a specific application, these models are not sufficient.

2.2 Simulation vs. Analytical Methods

The principle advantage of simulation based approaches is their accuracy. Given sufficient computational resources an emulation of each machine instruction could

	Queueing theoretic models	Models derived from real applications
Ease of Use	Difficult	Automated
Accuracy	Low	High
Ability to Generalize	Simple	Difficult

Table 2.1. Characteristics of queuing theoretic models vs. models derived from real applications.

potentially give perfect prediction of performance for a selected application. The principle drawback of simulation methods is the time and resources necessary to run the simulation. A CPU intensive simulation must be performed for each new machine configuration and problem size. This makes the exploration of relationships between different system parameters difficult to perform. Analytical models can arrive at predicted performance values much more quickly. If the model is represented as an equation (as it is with the models described in this research), mathematical manipulations can be performed to determine the sensitivity of an algorithm to machine characteristics. This rapid prediction characteristic of analytical models makes them attractive for the problem of performance debugging. A programmer will not be likely to try many alternate implementations if the performance predictions take hours to complete.

Mehra et al. have conducted research into minimizing the time necessary to perform a simulation [68]. They use trace files to generate a model which can be simulated in order to generate predicted performance for varying problem size and numbers of processors. Simulation efficiency can also be improved through using execution driven simulation to replace basic blocks with an increment of the total number of instructions executed [24].

	Analytical Models	Simulations
Time Required	Low	High
Ease of Use	High	Low
Accuracy	Low	High

Table 2.2. Characteristics of analytical models and simulation systems.

The characteristics of Simulations vs. Analytical models are summarized in Table 2.2. Although analytical models achieve lower levels of accuracy, their ease of use and rapid results generation makes them the only viable choice for a performance model. Through specifying the confidence level for a prediction, lower accuracy levels can be tolerated for analytical models.

2.3 Automatic vs. Manual Systems

Several performance modeling systems require a rewrite of application code in a modeling language. This requirement of manual intervention can discourage use of tools which rely on these models. Tools which rely on information which can be acquired from application source code, trace files, or an instrumented run of the program are much more likely to be used in improving performance.

Morris et al. have developed a simulation data flow language for modeling distributed and parallel systems. The language includes a graphical interface as well as a text representation. In addition to being cumbersome, the language appears to be limited to small configurations of processors and does not scale to MPP systems. The LaRCS language developed by Lo et al. is used to describe data flow in an application [65]. The system then builds a task graph which is used to map a problem to a physical processor array.

Performance prediction is important enough that some researchers have suggested incorporating performance analysis into the early stages of the software engineering process. Wabnig and Haring have developed a system where programs are specified using directed task graphs [44]. The PAPS (Parallel Program Performance Prediction Toolset) is then used to generate a timed Petri net for the application [94]. A Petri net simulator is then used to generate a PICL (Portable Instrumented Communication Library) file. Using this system the programmer is able to focus on performance early in the design cycle and project results for various architectures [95].

Saghi et al. have developed a system to select the best mode of parallelism to use for an application [83]. The user specifies the complexity of each part of an application by hand and the system determines which parallel implementation will result in the highest performance on a particular hardware architecture. Since algorithmic complexity must be specified by hand this technique results in excessive overhead for the programmer.

By having the programmer input an augmented task graph representation of an application, the methodology proposed by Menasce predicts the performance of message passing programs with looping structures [71].

2.4 Static vs. Dynamic Analysis

Of particular importance are prediction techniques which incorporate information from the source code or from an instrumentation run of a target application. These methods achieve higher accuracy and allow a programmer to view the effects of modifications to an algorithm and implementation. Dynamic performance prediction techniques use sample runs on a target architecture to determine the amount of time spent in communications and computations. Static methods gather information strictly from the application source code. Isoefficiency metrics can be viewed as a static performance prediction method but the overhead necessary to have an analyst derive the isoefficiency function limits their usability.

Previous work in reconstructing instruction traces has examined the minimal amount of data necessary to represent control constructs in sequential applications [61]. A similar analysis of parallel applications indicates that pure static analysis is not sufficient to characterize the execution time of an application. Although static methods may be able to predict speedup values for many programs, predicting the number of operations to be executed requires knowledge of the flow of control which cannot be predicted at compile time.

2.4.1 Dynamic Methods

The PPPT, a Parameter based Performance Prediction Tool, developed by Fahringer is a dynamic technique used to parallelize and optimize code in the Vienna Fortran compilation system [31]. An instrumented run of an existing Fortran 77 program is used to locate sections of the code where a majority of the time is spent. The instrumentation run also is used to determine program unknowns, such as loop iteration counts and branch probabilities. The prediction system is not applied to scalable applications and the authors have noted that performance prediction for programs with variable problem size is an open problem. Similar work has also been performed by Chapman [14].

Through examining system statistics at run time, Crovella et al. determine the cause of poor performance in an application [25]. This information is used in performance debugging of an application. More recent work has focussed on using a large number of instrumented runs of a sample application in order to fit equations to the various overhead functions for parallel programs [26]. These equations can then be used to predict performance as the problem size is varied.

Dynamic performance prediction techniques have the following characteristics:

- Strengths
  - The analysis procedure for these techniques is automated and requires little effort from the programmer.
  - High levels of accuracy can be attained for the problem size used during instrumentation run since constants are preserved in expressions for the number of operations performed.
- Weaknesses
  - These methods generally discard symbolic information about loop iteration variables and are limited in their ability to accurately model the behavior of complex loop constructs.
  - Curve fitting is inexact and time consuming. Multiple runs of an application on the target architecture for multiple algorithm implementations require excessive use of scarce computational resources. The time required may also discourage programmers from experimenting with multiple implementations in order to optimize an application.

2.4.2 Isoefficiency

The isoefficiency metric has been developed to describe the rate at which the problem size must be increased in order to maintain constant efficiency as the number of processing elements grows [36, 59, 60]. In order to develop the isoefficiency function, an algorithm must be analyzed to determine asymptotic equations for the number of computations and communications performed in a parallel application. Applications with lower order isoefficiency functions will generally be able to make more effective use of increasing numbers of processors [38, 39]. Other researchers have developed metrics similar to isoefficiency which describe the rate at which the problem size must be scaled in order to achieve acceptable parallel performance [73].

Isoefficiency analysis can be characterized in the following way:

- Strengths
  - Complex iteration constructs can be accurately modeled since the programmer has access to structure of the source code or algorithm.
  - The performance prediction process can occur extremely quickly since sample runs of the application are not necessary.
- Weaknesses
  - Since an asymptotic analysis is performed, the constants are lost as equations for operation counts are derived. Execution time cannot be predicted unless more detailed analysis is performed. Although isoefficiency may provide a good theoretical measure of an application, estimating the performance on a finite number of processors with a finite problem size requires additional details.
  - Analysis of algorithms is an extremely labor intensive activity. Most programmers are unwilling to perform the work necessary to determine the complexity of each part of their code.

The symbolic performance prediction techniques described in Chapter 4 combine the strengths of dynamic methods and isoefficiency. A single instrumentation run is performed on the program with a scaled down problem size in order to build a call graph. This call graph is combined with symbolic information about the iteration variables in enclosing loops obtained through static analysis. An exact expression is then generated for execution time as a function of system parameters and the problem size [21]. 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.

Previous research into deriving order notation bounds on the complexity of computations and communications for important applications [82] could be used as input into our model in order to investigate applications where there is no available source code implementation.

2.4.3 Static Methods

Balasundaram et al. [5] have developed a static performance estimator based on a training set approach. Their analysis focuses on matching sections of source code with templates for which performance has previously been computed. Compiling a sufficient number of templates to match a given program is a difficult task and has been shown to be impractical by other researchers [30].

The MetaMP language described by Otto et al. was developed for use as an intermediate language between HPF and message passing level code [75]. The Tiny loop restructuring tool [96] can be used to predict the performance for MetaMP programs. Tiny symbolically analyzes code to count the frequency of floating point operations, memory accesses, stride-1 inner loops, non-stride 1 inner loops and invariant in inner loops. An attempt is then made to analyze which loop structure will result in the highest performance. Although the tool is able to analyze the effects of loop restructuring it does not address the problem of speedup calculations for an entire application.

While researching automatic data partitioning techniques, Gupta and Banerjee have also investigated performance prediction [40, 41]. They use constraints derived statically from the source code in order to generate a model. Although the user is currently required to input loop bounds and true ratios for conditional statements, the authors feel that compilers should be able to generate most of this information. Pattern matching at the statement level is used to derive

	Isoefficiency	Static	Dynamic
Ease of Use	Difficult	Automated	Automated
Accuracy	Low	Medium	High

Table 2.3. Characteristics of isoefficiency, static and dynamic models.

constraints on data distributions. These constraints combined with goodness measures, which approximate the communication penalty if the constraints are not satisfied, are used to predict the execution time for a program. The work is based on Paraphrase-2 [78] and focuses on data distributions for Fortran 77 code. The authors state that static compile time analysis should be feasible for applications with a regular computational structure and static dependence patterns that can be determined at compile time.

Although static analysis cannot be used for predicting execution time, it can be useful in guiding a compiler through estimating the relative speedup values attainable by alternate optimizations of a program [20, 18]. The model described in Chapter 3 implements a static analysis technique designed to differentiate between various implementations of a program. It can also be used for applications with fixed problem size where a sample execution run would take too much time to be practical.

Figure 2.3 summarizes the characteristics of static and dynamic models. Static methods are necessary in order to aid in compiler optimizations and for performance prediction of fixed problem size applications. A single instrumentation run with a small problem size, combined with static information can be used to predict performance for scalable applications.

The static model described in Chapter 3 differs from related work in that it is a first attempt at performing totally static prediction for applications across diverse hardware platforms. The dynamic model described in Chapter 4 represents new work in accurate performance prediction for scalable applications. These models are tailored to meet the requirements for performance debugging, architectural improvement, hardware selection and compiler optimization. Table 2.4 specifies which characteristics are necessary in order to meet these requirements. By using analytical models which automatically incorporate information from real applications, all of these criteria can be met. This work is the first to our knowledge to apply the regression techniques described in Chapter 5. The full equation for execution time generated by the symbolic performance model allows us to perform mathematical manipulations to determine the sensitivity of an application to various system parameters. This sensitivity and cost optimal analysis presented in Chapter 6 also originated with this research.

	Models based on real programs vs. Theoretic Models	Analytical Models vs. Simulations	Automatic model generation vs. Manual	Static vs. Dynamic
Performance Debugging	Real Programs	Analytical	Automatic	Don't Care
Architectural Improvement	Real Programs	Don't Care	Automatic	Don't Care
Selection	Real Programs	Don't Care	Automatic	Don't Care
Compiler Optimization	Real Programs	Analytical	Don't Care	Static

Table 2.4. The targeted uses of performance information impose specific constraints on the types of models which can be employed in generating predictions. The rows of the table show the four uses of performance prediction information. The columns indicate different ways of categorizing models. A given use may require a specific characteristic in given category, or it may have a don't care condition.

Chapter 3
Static Performance Prediction

Historically there have been two significant barriers to the development of accurate performance models for multicomputers. Programmers who are familiar with the structure of parallel applications often do not have the motivation or hardware background to build accurate models of parallel architectures. As a result, they usually do not have adequate performance feedback during the development of a parallel application. Secondly, architects who understand the underlying systems have not had access to algorithmic information from real applications to use as input into their models. Consequently, engineering models typically use statistical inputs or simplified program "kernels" instead of data collected from the execution of non-trivial parallel programs. As a result there has not been a consistent improvement in the balance between the subsystems of multicomputers [22, 49, 92]. The static performance prediction technique described here provides a vital bridge between parallel system designers and programmers. It allows programmers to evaluate algorithmic choices on different parallel architectures, and it allows system designers to understand the effects of varying system parameters using real application data.

For our purposes, multicomputers will be defined as parallel computers where the processors do not have shared memory [80]. This definition includes everything from clusters of workstations on Ethernet to machines with much faster and more complex interconnection schemes. These systems promise to provide solutions to many problems that require more computational resources than are available on conventional sequential processors.

We will use the term "Architectural Scaling" to denote the process of writing an application for platforms ranging from a cluster of workstations to a MPP system. A similar term, "Distributed Program Development" [76], has been coined to describe the process of developing an application on a machine when the program is intended to run on another specific machine. Architectural scaling is the process of developing an application which will run on several different parallel architectures. This process relies heavily on the ability to predict the performance as system parameters change and can benefit from the results of static analysis.

A single processor family will often be used from the desktop to the MPP machine. In this environment, programmers will want feedback on the likely performance of their codes as they move them between different computing platforms. This process of architectural scaling can consume significant computational and human resources if performance prediction is not available. System designers in this environment will need performance prediction information to know where to target their MPP machines to provide the highest increase in performance over clusters of workstations.

3.1 Model Specifications

Performance prediction information is useful only if it can be obtained at a reasonable cost in terms of programmer effort and computational resources. This model has been tailored for the following uses:

- During program development, it is important for the programmer to determine the effect that changes to the source code will have on the performance of the algorithm. If this performance prediction is difficult or time consuming, the programmer will not be likely to try very many different implementations of an algorithm. For this reason, our performance model will rely primarily on algorithmic information extracted from the source code.

Where multiple parallel architectures are available, performance prediction information can also allow the user of an application to choose the architecture and number of processors which will result in the most efficient use of computational resources. Because our model can make predictions of MPP performance from data collected on smaller systems, it offloads performance tuning tasks from large production systems. This is particularly valuable when a smaller configuration of a multicomputer is locally available and the larger system is a shared national resource.

- Many parallel applications fail to achieve good performance on multicomputers because the systems are unbalanced. If the message passing time is too high compared to the time for a computation, then fine grain applications will never achieve good performance. System designers have little information about how balanced the communication and computation must be to perform acceptably on a target set of applications. Since this model does not require sample runs on the target architecture, system designers can use it to determine the effects of varying system parameters on a variety of parallel applications.

3.2 Performance Metrics

There are several ways of evaluating performance in a parallel environment. On a sequential machine, the principle performance goal is to minimize the execution time of an application. In a parallel environment it is also important to make efficient use of the available processors. If the parallel machine is not able to execute the algorithm significantly faster than a single node, then there is no reason to buy a parallel machine; it would be more cost effective for the algorithm to be run on a single processor. Speedup is often used as a measure of how efficient an architecture is at executing a parallel algorithm. Although opinions are divided on exactly how speedup should be measured, there is some evidence that a majority of computationally oriented mathematical researchers feel that speedup should be used as the primary measure of parallel performance [6]. The speedup achieved by an algorithm running on p processors can be defined as:

$$Speedup(p) = \frac{\begin{array}{c}\textit{Time to solve a problem using the best}\\ \textit{sequential algorithm on one processor}\end{array}}{\begin{array}{c}\textit{Time to solve the same problem with}\\ \textit{the parallel algorithm on p processors}\end{array}}$$

It is often extremely difficult to determine the execution time of the best sequential algorithm for a given problem. For this reason we will use parallelizability, or relative speedup, as the primary performance measurement in this research. Relative speedup is defined as:

$$S(p) = \frac{\begin{array}{c}\textit{Time to solve a problem using the parallel}\\ \textit{algorithm on one processor}\end{array}}{\begin{array}{c}\textit{Time to solve the same problem with}\\ \textit{the parallel algorithm on p processors}\end{array}}$$

3.3 Model Development

Previous research has identified factors which are important predictors of performance. Several trends in modern distributed memory parallel systems permit us to make simplifying assumptions which lead to a more understandable model. We use these assumptions to develop the preliminary parallel model described here.

The time to execute the program on a single processor $T_1 = T_s + T_p$ where T_s is the inherently sequential part of the algorithm and T_p is the parallelizable part of the algorithm. Several researchers [32, 73] have suggested the following formula to model the parallel execution time of an algorithm:

$$\tau(p) = T_s + \frac{T_p}{p} + \sigma(p, \tau)$$

where $\sigma(p, \tau)$ is a function which estimates the communication overhead given the topology τ. We will use an enhanced version of this formula.

3.3.1 Compiler Effects

The Dataparallel C compiler generates a standard C program as its output. The native C compiler then compiles the C code into an executable. The quality of the native C compiler can have a big effect on the number of machine instructions generated for each logical operation specified in the program. A constant for each compiler C_ϕ can be determined through benchmarks or through extrapolating from results of the same compiler on other architectures. This compiler factor will be used to create a better estimate of the number of instructions executed in an application.

3.3.2 Parallel Overhead

We have found that it is important to include a term for parallel overhead introduced by the emulation of virtual processors in Dataparallel C. Depending upon the choice of global or local variables, different optimizations are possible which result in variable overhead for virtual processor (VP) emulation. The number of times the compiler must set up a VP emulation loop ($\mathcal{X}_\lambda$) can be used to estimate the parallelization overhead in the computation. The time spent in parallel overhead, T_λ will be accounted for in our model by the term $C_\phi \mathcal{X}_\lambda$.

The generalized form of the relative speedup for p processors can be expressed as:

$$S(p) = \frac{T_1}{\tau(p)} = \frac{T_s + T_p}{T_s + \frac{T_p}{p} + \sigma(p, \tau) + T_\lambda}$$

This reduces to Amdahl's law when $\sigma(p, \tau) = 0$ and $T_\lambda = 0$.

3.3.3 Trends in Floating Point Performance

In the past, floating point arithmetic was so much more time consuming than integer arithmetic that integer instructions were ignored in calculations of algorithmic complexity. The current generation of microprocessors exhibit floating performance that is equal to or greater than the integer performance. Many of the microprocessors used as compute nodes in multicomputers can execute two floating point operations (an add and multiply) in the same time that an integer instruction can execute. Several of the major multicomputer vendors are using this class of microprocessors for computational nodes. The Intel Paragon uses the i860 processor which has this feature [51]. The IBM POWERparallel machine (SP1) uses RS/6000 technology which also has comparable times for floating point and integer instructions.

Since floating point and integer instructions take close to the same amount of time in these machines, it is possible for the model to estimate the number of computations through examining the parse tree generated by the compiler and counting the number of operations in sequential code ($\mathcal{X}_s$) and in parallel code ($\mathcal{X}_p$).

3.3.4 Communication Overhead

The $\sigma(p, \tau)$ term incorporates overhead caused by communication between processors during the computation. In the general case, it accounts for effects caused by the topology dependent distance between processors, link bandwidth and message startup time for communications. For this analysis, we will assume that the machine uses cut-through or wormhole routing. With these circuit-switched routing schemes, the transfer time between any two nodes is fairly similar. Most modern parallel computers employ some form of circuit-switched technology to avoid the delay associated with store and forward routing. This simplifies the $\sigma(p, \tau)$ term by allowing us to ignore distance considerations when estimating the communication cost for an operation.

One of the most significant contributors to communication overhead in the current generation of multicomputers is the message startup time (T_c). We will define message startup time as the total time between the call to the communication library and when data begins to be transmitted across the interface. This startup cost includes time spent in the communication library and system call overhead as well as the inherent time for the hardware to begin transmitting. As multicomputers have matured, they have added multitasking operating systems and more stringent error checking which has increased the overhead associated with starting a communication. Several researchers have noted that startup cost is the predominant factor in determining the total cost of communication [49, 92]. For this reason we will assume that overhead induced by limitation in actual bandwidth on the communication channels and link congestion are actually second order effects, and we will not consider them in our model. This makes the model much simpler, since it does not have to deal with message length, but can just account for the number of messages exchanged. Some applications which transmit large data sets will also see the the network bandwidth as a first order effect, but for problems with neighbor communications it can often safely be ignored. Since the Dataparallel C compiler inserts explicit message passing calls into the instruction stream, the number of communications $\mathcal{X}_c$ can be determined from the source code. We will define the normalized message startup cost C_c as the ratio T_c/T_ϕ (where T_ϕ is the time to execute a floating point instruction). The model will estimate the total number of cycles spent in communication to be $\mathcal{X}_c C_c$.

3.3.5 Memory Effects

Several researchers have noted that the memory hierarchy can have a significant impact on the performance achieved by a parallel program [35, 98]. References to parallel variables in Dataparallel C are grouped into arrays of structures with an element for each virtual processor. When a program enters parallel code through the *domain* statement, we will assume that the entire array containing parallel variables used there will be accessed. This assumption holds for the problems we will be studying in this research and is approximately true for many other irregular computational problems. Because of the predictable sequential access of these arrays, it is possible to predict the number of memory accesses which will occur. These uncached accesses will generally be limited to parallel code and have a significant effect on the performance of popular multicomputer processors including the iPSC/860 [72]. The number of uncached memory accesses ($\mathcal{X}_m$) and the number of cycles necessary to access an uncached memory location (C_m) account for the time spent waiting for cache lines to fill.

3.3.6 Applying the Assumptions

Using information extracted from the source code, the compiler can estimate $\mathcal{X}_s$ and $\mathcal{X}_p$, the number of operations in the sequential and parallelizable portions of the code. Let $T_s = C_\phi \mathcal{X}_s T_\phi$ and $T_p = (C_\phi \mathcal{X}_p + C_m \mathcal{X}_m) T_\phi$.

Our model of $\sigma(p, \tau)$ involves only the startup cost T_c and the topology. We can express $\sigma'(p, \tau) = \sigma(p, \tau)/T_\phi$ in terms of the normalized startup time. For a broadcast communication on a hypercube topology $\sigma'(p, \tau) = \mathcal{X}_c C_c (1 + \log(p))$. Neighbor communications would result in $\sigma'(p, \tau) = \mathcal{X}_c C_c$.

Using the dominant effects we have described here,

$$\begin{aligned} T_s &= (C_\phi(\mathcal{X}_s + \mathcal{X}_p) + C_m \mathcal{X}_m) T_\phi \\ T_p &= C_\phi \mathcal{X}_s T_\phi + \frac{(C_\phi \mathcal{X}_p + C_m \mathcal{X}_m) T_\phi}{p} + T_\phi \sigma'(p, \tau) + T_\phi C_\phi \mathcal{X}_\lambda \end{aligned}$$

With speedup

$$S(p) = T_s / T_p$$

$$= \frac{C_\phi(\mathcal{X}_s + \mathcal{X}_p) + C_m\mathcal{X}_m}{C_\phi\mathcal{X}_s + \frac{C_\phi\mathcal{X}_p + C_m\mathcal{X}_m}{p} + \sigma'(p,\tau) + C_\phi\mathcal{X}_\lambda}$$

the T_ϕ terms drop out and we are left with a speedup equation dependent only on the variables which are available to our prediction tool.

The terms $\mathcal{X}_s$, $\mathcal{X}_p$, $\mathcal{X}_m$, $\mathcal{X}_\lambda$ and $\mathcal{X}_c$ can all be determined from the internal parse tree generated by the compiler. The term C_ϕ can be determined through a sample program or through experience with the compiler on other processors. C_ϕ describes the efficiency of the compiler in generating optimized code. The terms C_c and C_m can be obtained from system specifications or through benchmark programs.

Given an equation for $S(p)$, we can derive expressions describing the levels of balance which are required for an efficient parallel architecture. For our analysis we will assume that an efficient execution of an algorithm will require a relative speedup of at least $p/2$ where p processors are used. This results in processor utilization of 50%.

Theorem 3.1 *In order to achieve a utilization of 50% on grid problems with neighbor communications, the following inequality must hold:*

$$C_c < \frac{C_\phi(\mathcal{X}_s + \mathcal{X}_p) + C_m\mathcal{X}_m - (p-1)C_\phi\mathcal{X}_s - pC_\phi\mathcal{X}_\lambda}{p\mathcal{X}_c}$$

Proof:

Given $S(p)$, we can determine the utilization

$$\mathcal{U}(p) = \frac{S(p)}{p}$$

$$= \frac{C_\phi(\mathcal{X}_s + \mathcal{X}_p) + C_m\mathcal{X}_m}{pC_\phi\mathcal{X}_s + C_\phi\mathcal{X}_p + C_m\mathcal{X}_m + p\sigma'(p,\tau) + pC_\phi\mathcal{X}_\lambda}$$

$$= \frac{C_\phi(\mathcal{X}_s + \mathcal{X}_p) + C_m\mathcal{X}_m}{(C_\phi(\mathcal{X}_s + \mathcal{X}_p) + C_m\mathcal{X}_m) + (p-1)C_\phi\mathcal{X}_s + p\sigma'(p,\tau) + pC_\phi\mathcal{X}_\lambda}$$

$$= \frac{1}{1 + \frac{(p-1)C_\phi \mathcal{X}_s + p\sigma'(p,\tau) + pC_\phi \mathcal{X}_\lambda}{C_\phi(\mathcal{X}_s + \mathcal{X}_p) + C_m \mathcal{X}_m}}$$

In order to achieve a utilization greater than 50%

$$\frac{(p-1)C_\phi \mathcal{X}_s + p\mathcal{X}_c C_c + pC_\phi \mathcal{X}_\lambda}{C_\phi(\mathcal{X}_s + \mathcal{X}_p) + C_m \mathcal{X}_m} < 1$$

This can only occur when

$$C_c \; < \; \frac{C_\phi(\mathcal{X}_s + \mathcal{X}_p) + C_m \mathcal{X}_m - (p-1)C_\phi \mathcal{X}_s - pC_\phi \mathcal{X}_\lambda}{p\mathcal{X}_c}$$

□

Several intuitive relationships can be observed as special cases of Theorem 3.1. If the message startup cost, C_c, is to have a positive value, then:

$$C_\phi(\mathcal{X}_s + \mathcal{X}_p) + C_m \mathcal{X}_m > (p-1)C_\phi \mathcal{X}_s - pC_\phi \mathcal{X}_\lambda \qquad (3.1)$$

If we assume that the memory component of the execution time ($C_m \mathcal{X}_m$) and the loop overhead ($pC_\phi \mathcal{X}_\lambda$) are not dominant, then Equation 3.1 reduces to $\mathcal{X}_p > p\mathcal{X}_s$. The parallel component of a computation must be more than p times as large as the sequential component if $\mathcal{U}(p)$ is to be greater than 50%.

Assuming memory accesses and loop overhead are insignificant and that the parallel component of the execution time dominates the sequential time

$$C_c < \frac{C_\phi \mathcal{X}_p}{p\mathcal{X}_c}$$

Essentially, this expression indicates that the grain size must be larger than the cost of a communication.

We can also observe that higher communication costs can be tolerated when a slower memory subsystem is used or when a poor compiler causes increased values of C_ϕ. Theorem 3.1 also indicates the sensitivity of parallel computations to VP emulation loop overhead.

3.4 Experimental Results

The results of several experiments are presented here to illustrate the utility of the analytical model in predicting performance. One set of experiments was performed to determine if the tool could accurately predict the performance effects of changing the implementation of an algorithm on a fixed target machine. This kind of performance prediction information would be used by a programmer or compiler to optimize a program. Other experiments were performed to demonstrate that the tool could predict performance on different target machines for several different algorithms. This kind of prediction information would be useful to system designers in determining the effects of changing system parameters.

3.4.1 Source Code Variation

One of the challenges of programming in Dataparallel C is determining the parallel type to use for different variables. Dataparallel C has a notion of global (mono) variables which are kept consistent across all of the physical processors and local (poly) variables which may be different for every virtual processor. There is a complex set of rules for determining which parallel type to use for loop variables or array index variables to produce the best performance [46]. In some cases, the choice depends on the target architecture to be used by the application. If the compiler were able to predict the performance characteristics for each of the choices, it could automatically select the correct types and relieve the programmer of the task of variable type selection.

Matrix multiplication is often used as a benchmark on parallel machines. Several versions of the matrix multiplication algorithm have been implemented in Dataparallel C. As a test of the model we changed two of the loop indices from parallel local variables to parallel global variables. The experiment was performed on the Intel iWarp array. The iWarp is connected in a mesh topology, uses wormhole

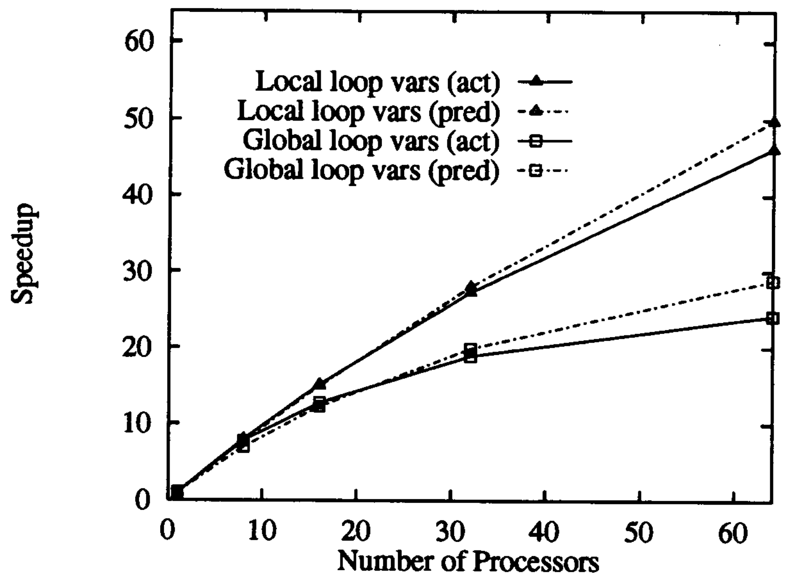


Figure 3.1. Experimental and predicted values results for 256 × 256 matrix multiplication on the iWarp array.

routing, has a message startup latency of 470 cycles and has similar floating point and integer execution times. The prediction tool was able to accurately predict the performance of the original version and the new version called "matrix2+" . The results are shown in Figure 3.1.

3.4.2 Experimental Results on Different Target Machines

A second set of experiments was performed using two target architectures that exhibit the features we described in our model development. The experiment was performed on the Intel iWarp array and on an iPSC/860. The iPSC/860 uses the Intel i860 processor, is connected in a hypercube topology, uses wormhole routing, has a message startup latency of 5280 cycles and has similar floating point and integer execution times. Experiments were performed using several standard Dataparallel C applications and were reported previously [20]. The experimental

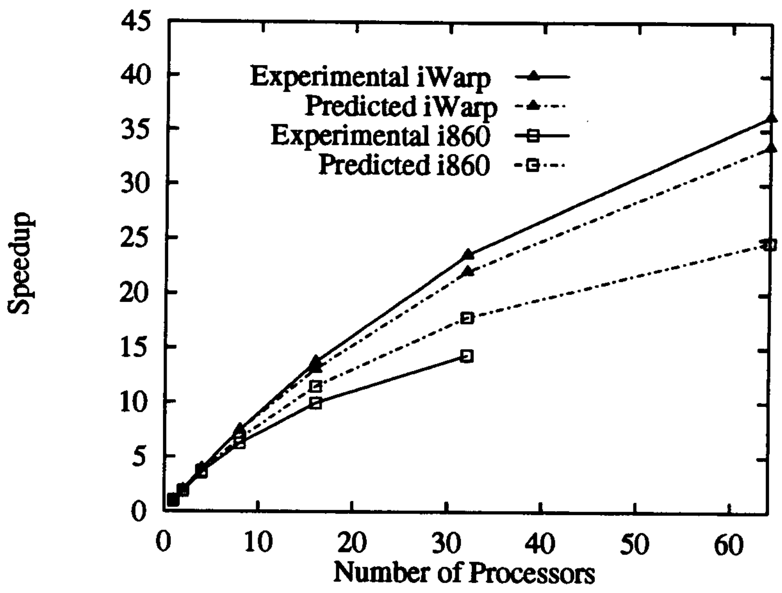


Figure 3.2. Experimental and predicted results for the shallow water atmospheric model.

results indicate that the analytical model is successful in predicting performance in the cases we have examined. Given an accurate model, important analysis can occur with respect to a specific application.

The shallow water atmospheric application was developed by the National Center for Atmospheric Research for benchmarking the performance of parallel processors [46]. The program solves a system of shallow water equations on a rectangular grid using a finite differences method. The model uses a two dimensional array of data elements that communicate with their nearest neighbors. The performance prediction tool is able to approximate the actual performance fairly accurately. More significantly, the tool was able to differentiate clearly between the performance to be expected on the two machines. The results are shown in Figure 3.2.

Performance information from an analytical model can allow a system architect to observe the effects of changing specific system parameters. In Figure 3.3 the message startup cost is varied for the shallow water atmospheric model to show the effect this parameter has on speedup. Figure 3.4 shows how cache miss penalty and message startup cost interact in predicted speedup results. Machines with large cache miss penalties will achieve greater speedup values for a given message startup cost because their effective grain size is larger. The performance in Mflops, however, degrades on machines with large values of C_m as is shown in Figure 3.5. Although machines with extremely low latency memory and network subsystems may promise higher performance, a more inexpensive solution may be acceptable for many users. If we assume an exponential increase in cost, corresponding to improvements in memory and network speed, a plot of performance per dollar can be generated as is shown in Figure 3.6. Given this information, manufacturers can position new machines near the cost optimal points for important applications. Users can also select parallel architectures which will be most economical for their particular needs. Results from this analytical model can be of significant use to systems architects and MPP users.

Ocean Circulation Model

This program simulates ocean circulation using a linearized, two-layer channel model [46]. This application also uses nearest neighbor communication; however, in this case the two machines achieve nearly identical speedup results. This is due to a combination of differences in grain size and the number of accesses to uncached memory between the Ocean Circulation Model and the shallow water model. It would be difficult for a programmer to guess that the two programs would perform with such disparity from perusing the source code. Again, the performance prediction tool was able estimate the speedup attained by the application. The results are shown in Figure 3.7 and Figure 3.8.

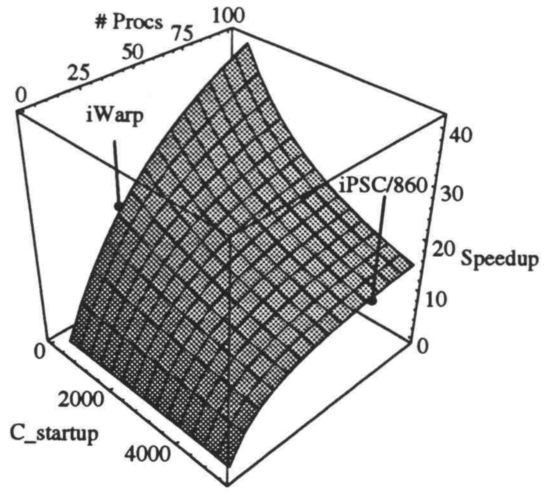


Figure 3.3. Speedup results for the shallow water atmospheric model with variable message startup cost. C_c is the message startup time divided by the time for an arithmetic operation. A C_c value of 470 corresponds to the iWarp processor. The value 5000 approximates the iPSC/860.

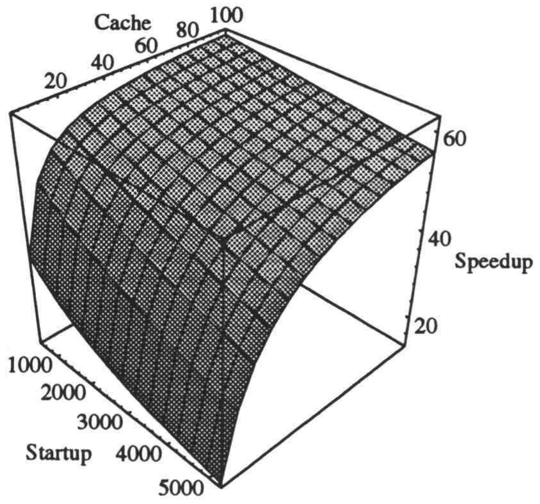


Figure 3.4. Speedup results for the shallow water atmospheric model for 64 processors with variable message startup cost (Startup) and cache miss penalty (Cache).

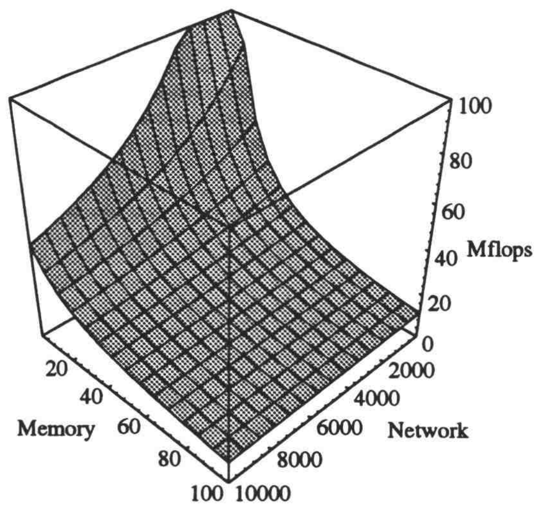


Figure 3.5. Mflops results for the shallow water atmospheric model for 64 processors and variable message startup cost and cache miss penalty. A processor speed of 10Mflops is assumed for each processing element.

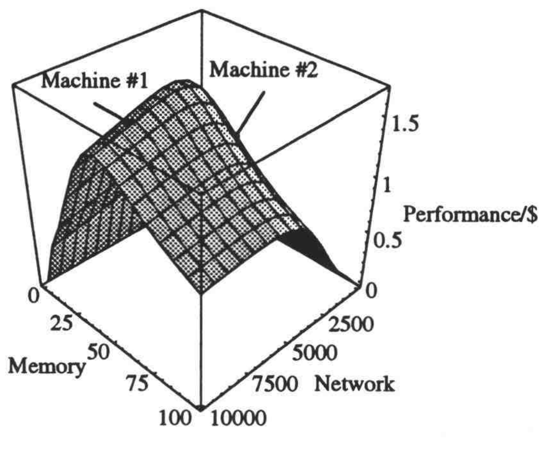


Figure 3.6. Predicting Mflops/dollar for the Shallow Water Model

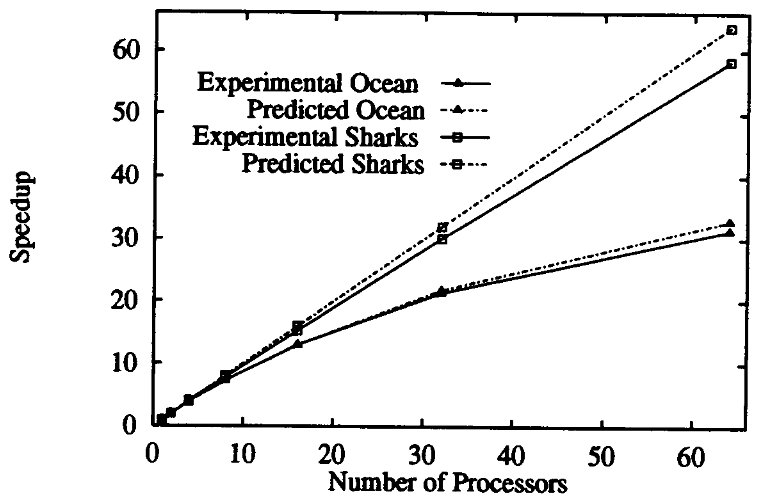


Figure 3.7. Experimental and predicted results for the Ocean Circulation Model and Sharks World on the Intel iWarp multicomputer.

Sharks World

Sharks World is included as an example of an application with few communications. The program simulates sharks and fish on a toroidal world [46]. As expected, both machines are able to achieve near linear speedup on this application. The predicted and actual results are shown in Figure 3.7 and Figure 3.8.

3.4.3 External Sorting

External sorting is a disk intensive problem. N records must be read and written to disk while $O(Nlog(N))$ comparison operations are performed. Additionally, in the worst case, all N records will have to pass through the bisectional bandwidth of the interconnection network. If the disk speed is not well balanced with the processor and network speed, the time spent reading and writing data will often dominate [63]. There are few guidelines as to how well balanced these system parameters should be in order to achieve acceptable performance. Preliminary

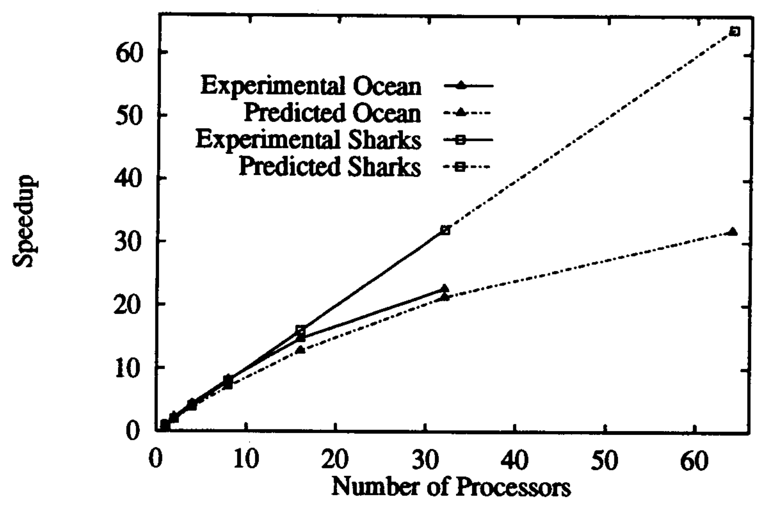


Figure 3.8. Experimental and predicted results for the Ocean Circulation Model and Sharks World on the Intel iPSC/860 multicomputer.

work has been done to predict the effects of architectural features on the sorting problem. We have developed a new parallel sorting algorithm that maximizes the overlap between the disk, network and CPU subsystems on a parallel node [19]. This algorithm hides much of the imbalance of a particular subsystem behind the operations of the others. Because of this overlap, we are able to draw some conclusions about the minimum levels of system balance that are necessary for any sorting algorithm. A model was built using this algorithm to examine the sorting problem on parallel architectures.

Figure 3.9 shows the speedup values predicted on 100 processors with skew equal to 1.5 as the network and disk speeds vary. Skew is defined as the ratio of the maximum number of records which are written to disk by any processor to the average number of records written by a node on the machine. The system parameters of several actual machine architectures have been labeled to show the estimated performance of the overlapped sorting algorithm on actual machines. All of the machine plots were made assuming SCSI disks on each node with a

3Mbyte/sec transfer speed. Figure 3.10 shows a close-up view of the region of maximum speedup. The Intel Paragon is shown to be on the plateau of maximum performance, but it is near the point where severe performance degradation would occur if network transmission time were increased. The network on the Paragon was assumed to have 200Mbyte/sec links configured in a mesh [51]. The IBM SP1 is plotted assuming a 6Mbyte/sec omega network [62]. The Meiko CS-2 has a logarithmic network where bisectional bandwidth increases linearly with the number of processors. The Meiko plot assumes that each node has a 50Mbyte/second link [70]. The SP1 appears to have higher disk transmission times because of the faster CPU speed. Both the network and disk axes are given in terms of the CPU speed. The Ethernet plot in Figure 4 pertains to a network of Sun SPARCstation 1 workstations with SCSI disks connected with Ethernet. It is obvious that the network bisectional bandwidth must be significantly increased before a network of workstations can be used with the overlapped sorting algorithm.

In order to validate our analytical sorting model, we used algorithmic information from the *Parallel Sorting by Regular Sampling* (PSRS) [86] algorithm in our model. We then compared our speedup predictions to those documented by the authors of the PSRS paper [86]. Our results were within 10 percent of the empirical results for the iPSC/2 and the iPSC/860 multicomputers. The authors claimed 37% lower performance than we predicted for the network of 8 Sun 3/80 workstations. It was impossible to determine the exact test environment used with the workstations and we attribute the additional error to our assumptions. Table 3.1 shows the predicted and actual results for the PSRS algorithm. In Figure 3.11 we show the impact of message startup time on the PSRS algorithm. Before the last merging phase of the algorithm can proceed, p communications must be initiated by each processor. As the number of processors becomes large, this message startup time degrades the speedup attained. With the analytical model we have proposed, algorithmic limitations can be detected without using critical resources

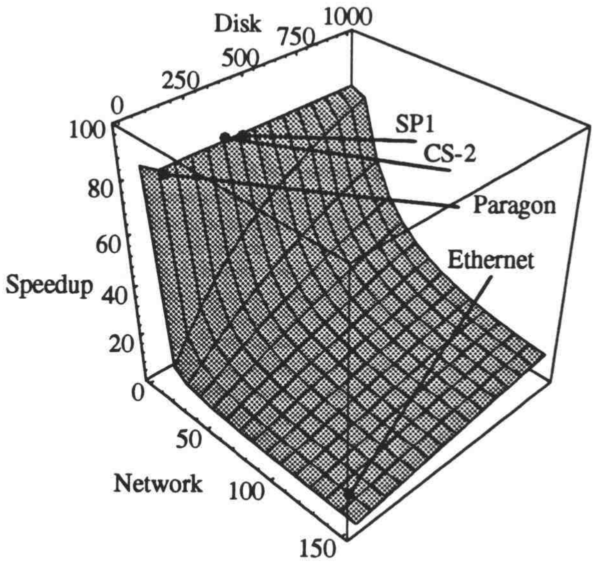


Figure 3.9. Results of an analytical model developed to investigate the impact on the sorting algorithm of varying system parameters. The Disk and Network axes are given in number of comparison times to transfer a tuple. Skew is assumed to be 1.5. Points are plotted for the Intel Paragon, IBM SP1, Meiko CS-2 and a network of Sun SPARCstation 1 workstations connected by Ethernet.

on MPP systems. The majority of time spent in performance debugging should be spent working with a performance prediction tool in a workstation environment.

Proper system balance is important to achieving acceptable speedup on the sorting problem. Modeling algorithmic behavior can help system designers know what effect their design decisions will have on the efficiency of an algorithm family.

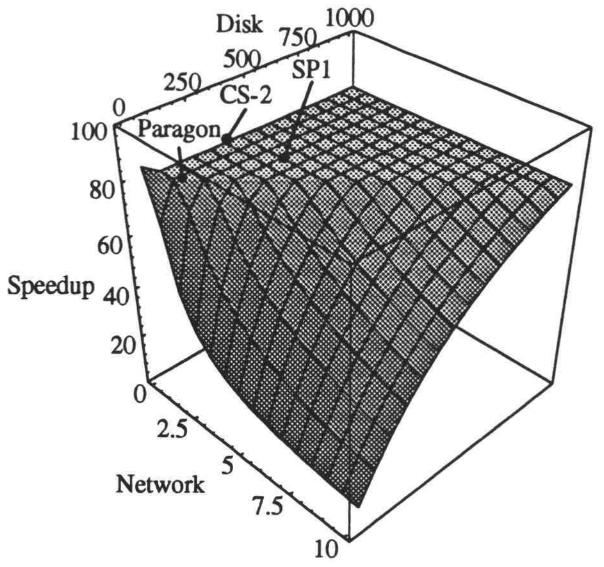


Figure 3.10. Closer view of the maximum speedup plane of the analytical model. The Disk and Network axis are given in number of comparison times to transfer a tuple. Skew is assumed to be 1.5. Points are plotted for the Intel Paragon, IBM SP1 and Meiko CS-2.

			Actual	Predicted	
Machine	Nodes	Items sorted	Speedup	Speedup	Error
iPSC/2-386	32	4000000	27.5	25.0	9.08%
iPSC/860	64	8000000	38.0	34.0	10.6%
Network	8	500000	4.0	5.5	37.1%

Table 3.1. Actual and predicted speedups for the PSRS algorithm. Actual speedup results are from Shi [86].

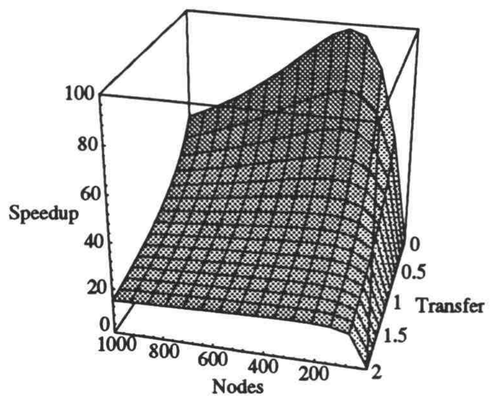


Figure 3.11. Analytical results for the PSRS algorithm as the number of processors becomes large. The transfer time is given in number of comparison times to transfer an integer. The message startup time is set at 5000 comparison times. This corresponds to the startup time for the iPSC/860.

3.5 Summary

Static performance prediction plays an important role in improving the efficiency of multicomputer environments. It can be used to provide feedback to optimizing compilers on the relative benefits of alternate program transformations. Static models are also important in characterizing the performance of fixed problem size applications where an instrumentation would take an excessive amount of time.

We have shown how this methodology can be used in analyzing the performance of several application classes. The accuracy of static analysis is also shown to be quite high on some applications. For other applications with irregular structure, an instrumentation run of the application run is necessary in order to achieve acceptable accuracy.

Chapter 4
Dynamic Performance Prediction

When additional processing elements are added to a computation, scalable applications can increase the problem size in order to maintain efficient execution. Speedup cannot be used as a performance metric for these applications since the number of computations will likely change as the number of processors vary. The performance prediction methodology used for scalable applications uses algorithmic information derived from the source code of a high level parallel language and an instrumentation run of the application with a small problem size. This information is combined with an analytical hardware model in order to predict execution time as the number of processors and the problem size vary. The techniques described here are termed dynamic because of their reliance on run-time instrumentation data. They also employ static information generated by the Dataparallel C compiler. By maintaining a symbolic representation of the computational model additional mathematical characterizations can be made through the use of algebraic manipulation packages such as Maple [15].

This model can be used to improve the architecture of future multicomputers through quantifying the relative benefits to real applications of enhancing different subsystems. The performance prediction system also can be used by programmers to improve the performance of scalable applications by pinpointing performance bottlenecks and allowing the developer to evaluate performance as the architecture and problem size vary. Some applications, however, will not be able to make effective use of a given hardware platform regardless of how many performance improvements are made. The prediction tools developed here can be used to determine which architectures will yield maximal performance for a set of target applications. This information can be used to make wise procurement decisions and to avoid the frustration of using a machine with a poor match to a problem space. The automated performance modeling techniques described here simplify the multi-dimensional task of algorithmic and architectural analysis.

In this chapter we first specify the class of scalable parallel applications which are targeted by this research. The dynamic performance prediction model and its implementation are then described.

4.1 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 [33]. Many of these programs can be scaled up by varying the number and size of data elements. The increased complexity induced by varying problem size makes automated performance prediction even more important for these programs.

Scalable algorithms are able to utilize increasing numbers of processing elements efficiently. Let T_{seq} be the number of sequential operations and T_{par} be the number of parallel operations in an application. The total execution time on p processors can be expressed as:

$$T(p) = T_{seq} + \frac{T_{par}}{p}$$

The maximum attainable relative speedup for a non-scalable application on p processors is:

$$\begin{aligned} S(p) &\leq T(1)/T(p) \\ &\leq \frac{T_{seq} + T_{par}}{T_{seq} + \frac{T_{par}}{p}} \\ &\leq \frac{T_{seq} + T_{par}}{T_{seq}} \qquad \text{as } p \rightarrow \infty \end{aligned}$$

If f is defined as the fraction of operations which are sequential in nature, then

$$f = \frac{T_{seq}}{T_{seq} + T_{par}}$$

and

$$S(p) <= \frac{1}{f}$$

This limitation on speedup for applications with a constant number of computations is commonly referred to as Amdahl's law [80].

As the number of processors increases, the accuracy of scalable applications increases while the elapsed time may remain constant. For example, a weather prediction program may require results in less than 24 hours, with the number of computations being scaled to achieve that execution time. Figure 4.1 illustrates the use of the Jacobi relaxation algorithm in solving for the steady state temperature distribution across a steel plate [80]. The plate is divided into square elemental regions, and the temperature is assumed to be constant across each region. A virtual processor is associated with each region. During an iteration of the algorithm, every virtual processor finds the average of the temperatures of each of its four adjacent regions in order to compute a next state temperature value. When this algorithm is mapped to a parallel machine, a single physical processor executes a virtual processor emulation loop to compute values for many regions. As the number of available processors increases, the problem size can be scaled

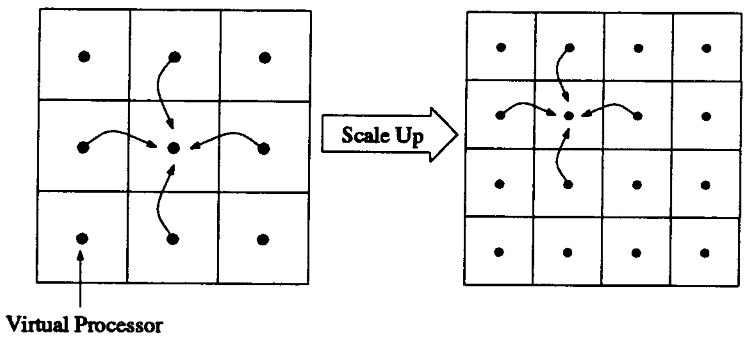


Figure 4.1. Illustration of problem size scaling for the Jacobi algorithm.

up, creating additional rows with smaller elemental regions. The resulting steady state distribution will be more accurate because the temperature is assumed to be constant across a smaller area.

Scaled speedup on p processors, denoted $SS(p)$, has been used to determine the efficiency of scalable applications [42, 59].

$$\begin{aligned} SS(p) &\leq \frac{T_{seq} + pT_{par}}{T_{seq} + \frac{pT_{par}}{p}} \\ &\approx p \quad \text{when } T_{par} \gg T_{seq} \end{aligned}$$

If we define efficiency on p processors, denoted $E(p)$, as the ratio $E(p) = SS(p)/p$, then efficiency for scalable applications can be as high as 100% on MPP systems. The ability of these applications to increase the number of computations as additional processors become available gives them an important advantage over programs with fixed problem sizes. The problem size can be scaled linearly with the number of processors or with the size of memory available as new processing nodes are added to the computation [74].

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 [50] 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 reduce trace volume by an order of magnitude [61] are still unacceptable when viewed in light of the current trend towards machines with rapidly increasing CPU speeds and relatively constant disk bandwidth. Simulation techniques would require in excess of 1000 hours for this sample program. This lower bound on simulation time exceeds the maximum acceptable overhead for performance debugging. With multiple MPP systems becoming available to a single researcher, performance prediction is also an important tool in matching the appropriate hardware platform to a specific application.

Accurate performance prediction information can be used by programmers for performance debugging, by compilers to guide optimizations, and by system architects to determine optimal hardware configurations for scalable applications. 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.

4.2 Dynamic Model Development

Many of the goals of this research were 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 4.2 shows a block diagram of the dynamic performance prediction system. Our implementation focuses on analysis for the Dataparallel C programming language [46]. 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 passing. This is another argument for high level languages as opposed to the message passing added to an

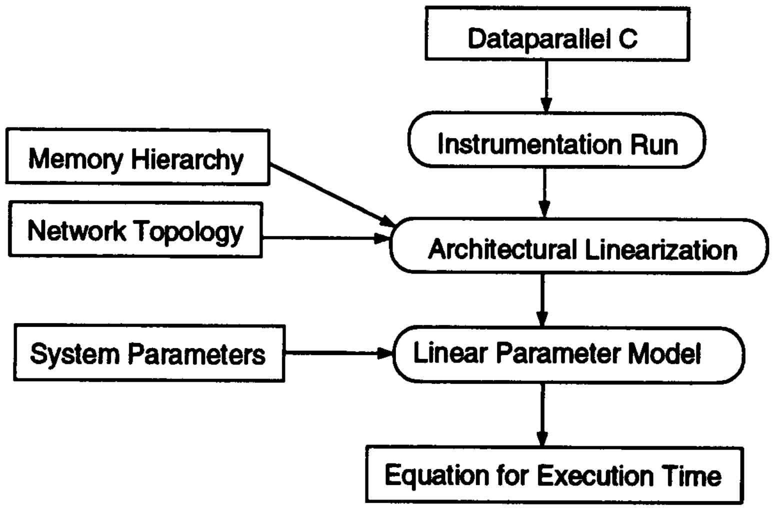


Figure 4.2. Block diagram for dynamic performance prediction system

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.

We use a dense linear system solver as an example problem to illustrate the steps taken in developing an analytical model for an application. Gaussian elimination with partial pivoting and back substitution is used as the algorithm for our sample application because of its complex iteration structure. Previous work in performance prediction using a sampling method failed to achieve accurate results for this application [20]. The program solves the linear system $AX = b$ when the matrix A is a dense array. Gaussian elimination reduces the A matrix to an upper triangular system and then performs back substitution to compute the final X values [80].

In the parallel implementation of this algorithm, two-dimensional data are distributed by rows to all processors. For each column of the A matrix, the row with the largest value in that column is used to reduce the remaining rows in the system. Pseudo code for the elimination phase of the algorithm is shown below. Performance prediction using sampling techniques is difficult for this problem because the initialization value for the iteration variable of the innermost loop depends on the iteration variable for the outermost loop.

```
for(i = 0; i < N; i++) {
  pivot_row = find_max_row(i);
  broadcast(pivot_row);
  for(active virtual processors) {
    for(k = i; k < N; k++) {
      data[k] = data[k] - pivot_row[k]*data[i]/pivot_row[i];
    }
  }
}
```

4.2.1 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 [67]. This time could then be used to improve the operation count for the program. The following examples show instrumentation code for major control constructs.

- Shape Declarations

```
static struct cg_shape_desc cg_shape1 = {1, 17, N, N};
```

- Iteration constructs

```
    for (i = 0; i < N; i++)
      {
static struct cg_loop_desc loop_tmp1 = {"i","0","N","i++"};
static struct cg_desc cg_tmp1 = {&cg_root,__LINE__,CG_LOOP,
  2,0,0,&loop_tmp1};
cg_count(&cg_tmp1);
```

- Conditional Code

```
    if (DPC_temp_0[DPC_vpi]) {
static struct cg_desc cg_tmp6 = {&cg_tmp5,__LINE__,CG_COND,4};
cg_count(&cg_tmp6);
```

- Virtual Processor Emulation

```
static struct cg_shape_inst cg_inst1 = {&cg_shape1,
  N*sizeof(double),0};
static struct cg_shape_inst cg_inst2 = {&cg_shape1,
  3*sizeof(int),&cg_inst1};
static struct cg_desc cg_tmp2 = {&cg_tmp1,__LINE__,
  CG_VPLOOP,2,&cg_inst2,0};
cg_count(&cg_tmp2);
      for (DPC_vpi = 0; DPC_vpi < DPC_num_vp_system; DPC_vpi++)
```

- Communications

```
static struct cg_comm_desc cg_com1 = {CG_REDUCE, &cg_inst2};
static struct cg_desc cg_tmp3 = {&cg_tmp1,
  __LINE__,CG_COMM,0,0,&cg_com1};
cg_count(&cg_tmp3);
```

Dataparallel C 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 through 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. A sample call graph is shown in Figure 4.3. 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:

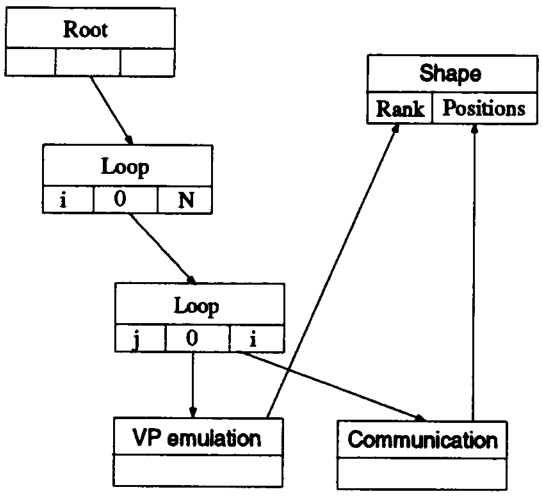


Figure 4.3. Call graph data structure.

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. The variables $Count_{seq}$ and $Count_{par}$ account for the number of sequential and parallel operations performed in the application. $Count_{VPloops}$ represents the number of virtual processor emulation loops that are executed. Cache behavior is evaluated through $Count_{L1Miss}$ and $Count_{L2Miss}$ which indicate the number of cache misses in the first and second level caches respectively. Message passing overhead is dealt with through enumerating the total number of message startup times ($Count_{CStart}$) and the number of bytes ($Count_{CBand}$) which are transmitted through the network.

$Count_{seq} :=$

$(sum(2 + (7) * 1.00, i = 0..N - 1) + sum(2 + (1) * 1.00,$

$i = 0..N - 1));$

$Count_{par} :=$

$(sum(N * 1 * (6 + (20) * 0.49) + N * 1 * (6), i = 0..N - 1) +$

$sum((N * 1 * (5 + (sum(64, k = i..N + 1) + 4) * 0.50)) * 1.00 +$

$N * 1 * (6 + (4) * 0.01) + N * 1 * (4) + (N * 1) + N * 1 * (2),$

$i = 0..N - 1));$

$Count_{VPloops} :=$

$(sum((N*1)+(N*1), i=0..N-1)+sum(((N*1))*1.00+$
$(N*1)+(N*1)+(N*1), i=0..N-1));$

$Count_{L1Miss} :=$

$(sum((L1Cache(((8*N)+0)*N*1))+(L1Cache(((8*N)+12)*$
$N*1)), i=0..N-1)+sum(((L1Cache(((8*N)+12)*N*1)))$
$*1.00+(L1Cache(((8*N)+12)*N*1))+(L1Cache(((8*N)+12)*$
$N*1))+(L1Cache(((8*N)+12)*N*1)), i=0..N-1));$

$Count_{L2Miss} :=$

$(sum((L2Cache(((8*N)+0)*N*1))+(L2Cache(((8*N)+12)*$
$N*1)), i=0..N-1)+sum(((L2Cache(((8*N)+12)*N*$
$1)))*1.00+(L2Cache(((8*N)+12)*N*1))+(L2Cache(((8*N)+$
$12)*N*1))+(L2Cache(((8*N)+12)*N*1)), i=0..N-1));$

$Count_{CStart} :=$

$(sum((ComStart(BC)+ComStart(BC))*1.00, i=0..N-1)+$
$sum(ComStart(BC)+ComStart(RE), i=0..N-1));$

$Count_{CBand} :=$

$(sum((ComBW(BC,12,N*1)+ComBW(BC,12,N*1))*1.00,$
$i=0..N-1)+sum(ComBW(BC,N*8,N*1)+$
$ComBW(RE,12,N*1), i=0..N-1));$

The summation operator represents iteration constructs. Fractional numeric values are derived from true ratios for conditional code. The problem size is denoted by the variable "N". The Maple [15] symbolic computation system can then be used to reduce the output to expressions which are functions of the fundamental architectural features of a particular parallel implementation. The following equations represent counts of arithmetic operations ($\mathcal{X}_{Ops} = Count_{seq} + Count_{par}/P$), virtual processor emulation loops ($\mathcal{X}_{VP}$), on-chip cache misses ($\mathcal{X}_{L1}$), page faults ($\mathcal{X}_{L2}$), message startup times ($\mathcal{X}_{St}$), and number of bytes transferred through the communication network($\mathcal{X}_{Bw}$). In addition to the problem size, the equations also are dependent on the number of processors (P), the expressions for level one and level two cache misses ($L1Cache(), L2Cache()$), the number of message startup times for each communication library invocation ($ComStart()$) and the number of bytes in each message ($ComBW()$). 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.

$$\begin{aligned}
\mathcal{X}_{Ops} &= 12.0N + \frac{121.8400000N^2 + 16.0N^3}{P} \\
\mathcal{X}_{VP} &= 6.0N^2 \\
\mathcal{X}_{L1} &= N\left(L1Cache(8\,N^2) + L1Cache((8\,N+12)\,N)\right) + \\
&\quad 4.0NL1Cache((8\,N+12)\,N) \\
\mathcal{X}_{L2} &= N\left(L2Cache(8\,N^2) + L2Cache((8\,N+12)\,N)\right) + \\
&\quad 4.0NL2Cache((8\,N+12)\,N) \\
\mathcal{X}_{St} &= 2.0N\,Comstart(BC) + N\,(Comstart(BC) + Comstart(RE)) \\
\mathcal{X}_{Bw} &= 2.0N\,ComBW(BC, 12, N) + \\
&\quad N\,(ComBW(BC, 8\,N, N) + ComBW(RE, 12, N))
\end{aligned}$$

4.2.2 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.

Several researchers have noted that the memory hierarchy can have a significant impact on the performance achieved by a parallel program [35, 98]. Previous research has indicated that a combination of local analytical models and empirical observations can characterize the performance of the memory system [34]. 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.

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

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 (*L1Cache* and *L2Cache*). The "*size*" parameter passed to *L1Cache* and *L2Cache* is generated by a compile time calculation of the total size of data items accessed from within a virtual processor emulation loop.

$$
\begin{aligned}
L1Size &:= 1024 * 4; \qquad \text{\# On-Chip Cache} \\
L1Cache &:= proc(size) \\
&\quad Heaviside(((size/P)/L1Size) - 1) * (size/P)/L1Size; end; \\
L2Size &:= 1024 * 1024 * 32; \qquad \text{\# Physical Memory} \\
L2Cache &:= proc(size) \\
&\quad Heaviside(((size/P)/L2Size) - 1) * (size/P)/L2Size; end;
\end{aligned}
$$

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 [46] 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. The values given below are intended to approximate a hypercube topology. A different expression is generated for each communication pattern. The $log[2](P)$ terms are Maple notation for $log_2(P)$.

$ComStart(NR)$	:=	1;	#Neighbor Read
$ComStart(NW)$	:=	2;	#Neighbor Write
$ComStart(BC)$	:=	$log[2](P)$;	#Broadcast
$ComStart(RW)$	:=	$log[2](P)$;	#Random Write
$ComStart(RR)$	:=	$2 * P$;	#Random Read
$ComStart(RE)$	:=	$log[2](P)$;	#Reduction
$ComStart(MR)$	:=	$P * log[2](P)$;	#Multireduce
$ComStart(PO)$	:=	1;	#Point to Point

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. The following output for Gaussian elimination can be combined with the cost in seconds for each of these operations in order to derive an equation for total execution time for the application.

$$
\begin{aligned}
\mathcal{X}_{Ops} &= 12.0N + \frac{121.8400000N^2 + 16.0N^3}{P} \\
\mathcal{X}_{VP} &= 6.0N^2 \\
\mathcal{X}_{L1} &= N\left(\frac{Heaviside\left(\frac{N^2}{512\,P} - 1\right)N^2}{512\,P}\right. \\
&\quad \left. + \frac{Heaviside\left(\frac{(8\,N+12)N}{4096\,P} - 1\right)(8\,N+12)\,N}{4096\,P}\right) \\
&\quad + 0.00097\frac{N^2\,Heaviside\left(\frac{(8\,N+12)N}{4096\,P} - 1\right)(8\,N+12)}{P} \\
\mathcal{X}_{L2} &= N\left(\frac{Heaviside\left(\frac{N^2}{4194304\,P} - 1\right)N^2}{4194304\,P} + \right. \\
&\quad \left. \frac{Heaviside\left(\frac{(8\,N+12)N}{33554432\,P} - 1\right)(8\,N+12)\,N}{33554432\,P}\right) + \\
&\quad .119\,10^{-6}\frac{N^2\,Heaviside\left(\frac{(8\,N+12)N}{33554432\,P} - 1\right)(8\,N+12)}{P}
\end{aligned}
$$

$$\begin{aligned}\mathcal{X}_{St} &= 2.0N + \frac{2\,N\ln(P)}{\ln(2)} \\ \mathcal{X}_{Bw} &= \frac{24.0N\ln(P)}{\ln(2)} + N\left(\frac{8\,N\ln(P)}{\ln(2)} + \frac{12\,\ln(P)}{\ln(2)}\right)\end{aligned}$$

4.2.3 Linear Parameter Model

Given counts for each operation and the cost for that operation 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, β_{VP} is the overhead associated with setting up a virtual processor emulation loop, β_{L1} and β_{L2} represent the penalty for a miss in level one and two cache respectively. The variable β_{St} is the message startup cost and β_{Bw} is the time required to transmit one byte through the bandwidth of a connection to a processing element. 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

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

where

$\boldsymbol{t}$ = a column vector of n observed values of $t = \{t_1, t_2, \ldots, t_n\}$

$\boldsymbol{\mathcal{X}}$ = an n row column matrix containing linearized algorithmic characteristics

$\boldsymbol{\beta}$ = a column vector containing the cost in seconds for each operation

$\boldsymbol{e}$ = a column vector of the errors between predicted and experimental values

The value of predicted execution time for the Gaussian elimination sample program is

$$\begin{aligned}
\hat{t} = {} & 10^{-9}\left(12.0N + \frac{121.84N^2 + 16.0N^3}{P}\right)\beta_{Ops} + 0.6\,10^{-8}\,N^2\beta_{VP} + \\
& \left(N\left(\mathit{Heaviside}\left(\frac{N^2}{512\,P} - 1\right)\frac{N^2}{512\,P} + \right.\right. \\
& \left.\mathit{Heaviside}\left(\frac{(8\,N+12)\,N}{4096\,P} - 1\right)(8\,N+12)\,N\frac{1}{4096\,P}\right) + \\
& \left.0.00097N^2\mathit{Heaviside}\left(\frac{(8\,N+12)\,N}{4096\,P} - 1\right)\left(\frac{8\,N+12}{P}\right)\right)\beta_{L1}10^{-9} + \\
& \left(N\left(\mathit{Heaviside}\left(\frac{N^2}{4194304\,P} - 1\right)\frac{N^2}{4194304\,P} + \right.\right. \\
& \left.\mathit{Heaviside}\left(\frac{(8\,N+12)\,N}{33554432\,P} - 1\right)(8\,N+12)\frac{N}{3.3\,10^7\,P}\right) + \\
& \left.0.12\,10^{-7}\,N^2\mathit{Heaviside}\left(\frac{(8\,N+12)\,N}{33554432\,P} - 1\right)\left(\frac{8\,N+12}{P}\right)\right)\beta_{L2}10^{-9} + \\
& \frac{0.4\,10^{-8}\,N\ln(P)\beta_{St}}{\ln(2)} + \\
& \left(\frac{24.0N\ln(P)}{\ln(2)} + N\left(\frac{8\,N\ln(P)}{\ln(2)} + \frac{12\,\ln(P)}{\ln(2)}\right)\right)\beta_{Bw}10^{-9}
\end{aligned}$$

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 chapter.

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.

Chapter 5
Statistical Analysis of Machine 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 [45]. 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 [89] allow large quantities of multivariate data to be analyzed with relative ease [13].

5.1 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.

5.2 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. The quantile plot of Figure 5.1 indicates that the residuals for the Meiko CS-2 have slightly longer tails than a normal distribution [52, 12].

We use a "Systematic Sampling" approach to selecting experimental values for the fitting process [84]. This approach was suggested in a study performed through the Statistics Department at Oregon State University. The results of this study are provided in Appendix A.

5.3 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.

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

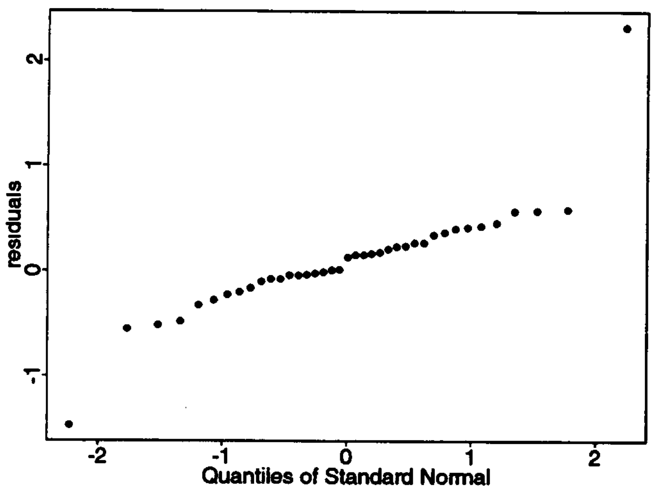


Figure 5.1. Quantile plot of model errors for experimental data from the Meiko CS-2 vs. a normal distribution of residuals.

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 $\sum_{i=1}^{n}(y_i - (\sum_{j=1}^{p} \beta_j x_{ij}))^2$ [13].

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 [45].

Our analytical 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 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.

5.4 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 5.1.

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 [13]. Values in Table 5.1 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.

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 5.1. Output of statistical parameters for the nCUBE 3200 multicomputer.

The startup time of 368μsec is similar to that found in previous research on the nCUBE [22].

The standard error column in Table 5.1 is an estimate of how much the regression coefficient 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 [54]. The large standard error value for the message startup cost indicates that a more detailed model needs to be investigated for this parameter. Through 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 5.2 illustrates predicted output for the Ocean Circulation Model on the nCUBE 3200. The program models wind-driven circulation in a density-stratified ocean [46]. The problem scales up by increasing the number of segments modeled in the east-west direction. The vertical bars join the upper and lower

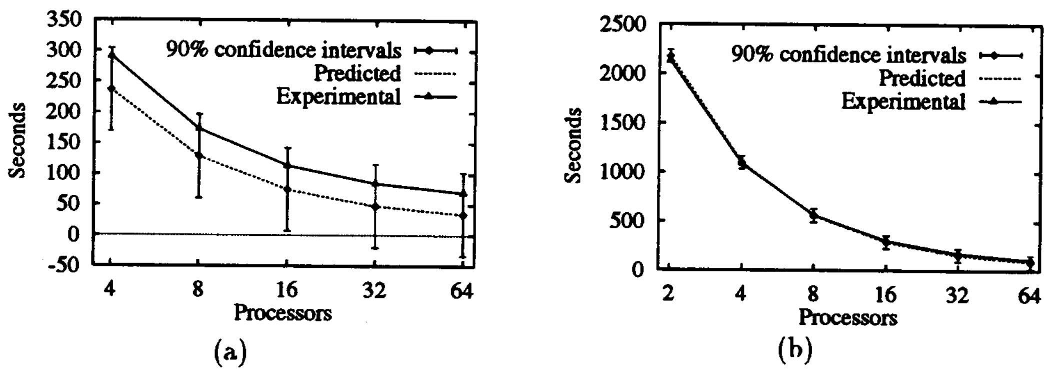


Figure 5.2. Predicted and experimental execution time in seconds for the Ocean Circulation Model on the nCUBE 3200 multicomputer with (a) 128 and (b) 640 segments in the east-west direction.

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 5.2. Since the standard error value for communications is higher than that for computations, the confidence interval for the 128 segment problem is wider.

Figure 5.3 examines predicted output for the Shallow Water Model application on the 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 [46]. 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 [70]. Each node is equipped with two vector processors to improve floating point performance. A copy of the multi-user Solaris

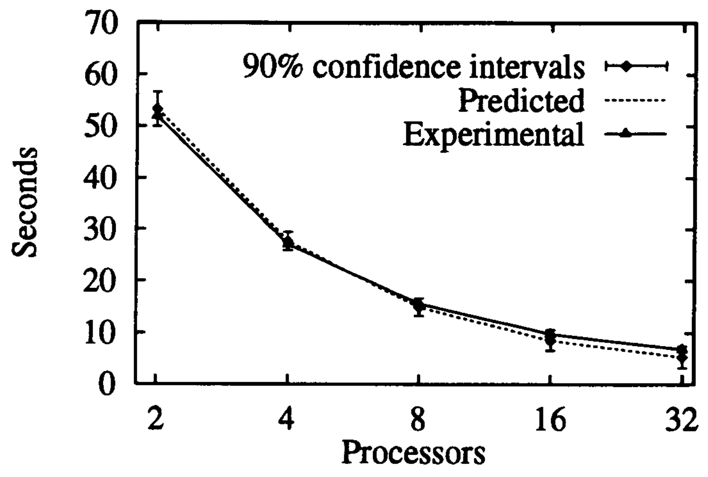


Figure 5.3. Predicted and experimental values for the Shallow Water Model with a 64 × 64 grid and 1200 iterations for an iPSC/860 multicomputer.

operating system executes on each node increasing the variability of successive runs of the same program on the machine. Figure 5.4 and Figure 5.5 show 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.

Through analyzing the statistical results, we have discovered several areas where our model needs to be improved.

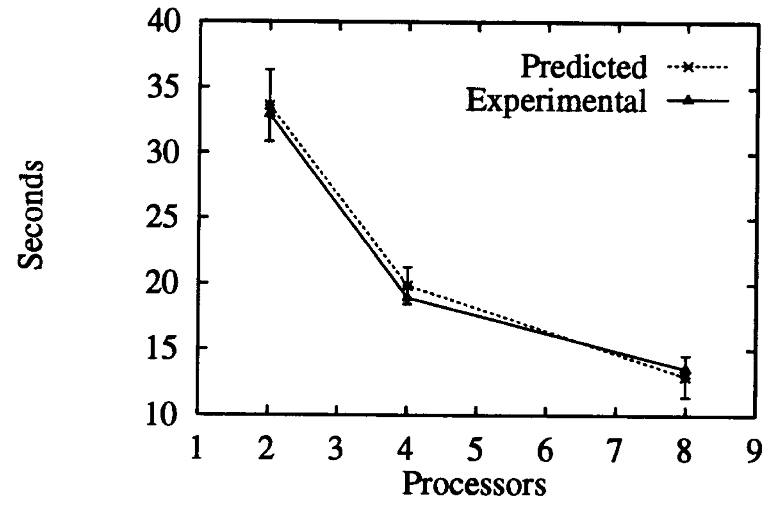


Figure 5.4. Predicted and experimental execution time in seconds for the Ocean Circulation Model on the Meiko CS-2 multicomputer with 640 segments in the east-west direction. The error bars are 90% confidence intervals for the predicted values.

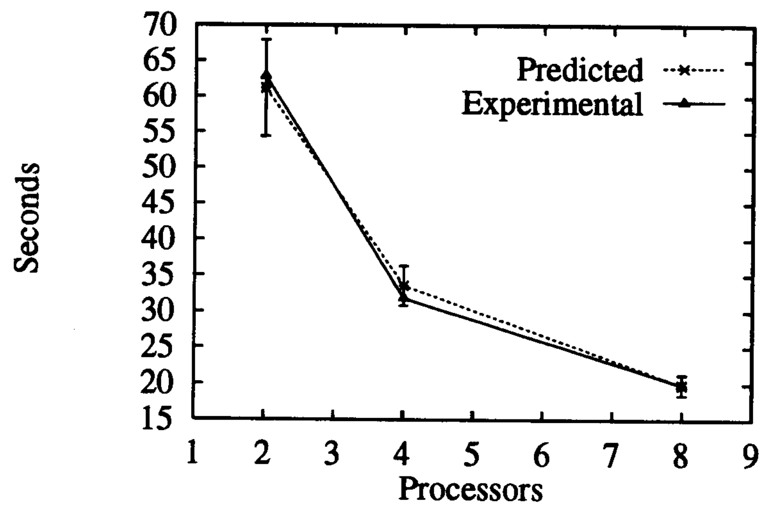


Figure 5.5. Predicted and experimental execution time in seconds for the Ocean Circulation Model on the Meiko CS-2 multicomputer with 1280 segments in the east-west direction. The error bars are 90% confidence intervals for the predicted values.

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 [89]. Additional information from the statistical analysis can also guide the development of more accurate models.

Chapter 6
Evaluation of Dynamic Modeling Techniques

Dynamic modeling techniques can be used to meet the needs of performance debugging, architectural enhancement and machine selection. In this chapter we analyze the accuracy of this model with several applications when run on the iPSC/860, Meiko CS-2 and nCUBE 3200 multicomputers with varying problem sizes and numbers of processors. Examples are also given to show the utility of this approach in providing performance information. A methodology for deriving expressions for cost optimal points is also developed.

6.1 Accuracy Analysis

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 6.1 indicates that over 90% of the experimental runs achieve less than 40% error for all three machines.

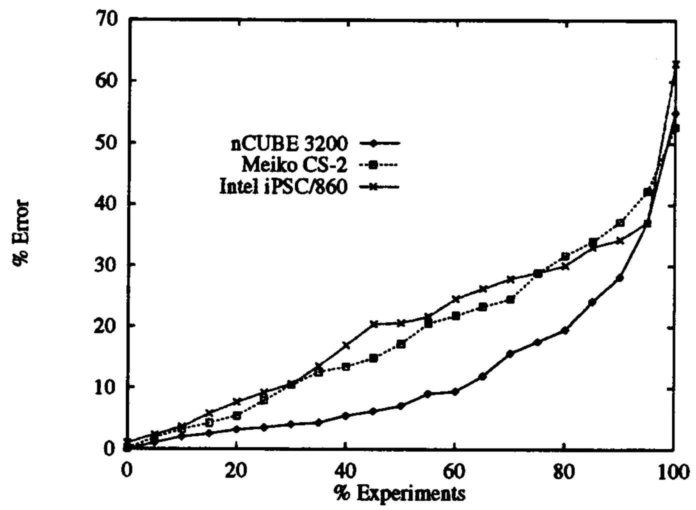


Figure 6.1. Percent error for experimental runs of applications in the validation suite.

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 6.1.

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 6.1. Experimental error values.

Machine	β_{VP} Value in nsec	Standard Error
iPSC/860	-209069	215109
Meiko CS-2	152512	326691
nCUBE 3200	38194	219144

Table 6.2. Coefficient and standard error values for β_{VP} with constant virtual processor emulation loop overhead.

Through examining data from the statistical package, improvements to the accuracy of the model can been made. A previous version of the model accounted for the overhead of a virtual processor emulation loops with a constant value. The standard error values for the β_{VP} coefficients shown in Table 6.2 was unexpectedly large. After analyzing the experimental data, we hypothesized that the overhead for a VP loop could be accounted for more accurately by a term which scaled with the number of virtual processors being operated on in the loop. This change in the model resulted in the increased accuracy shown in Figure 6.2.

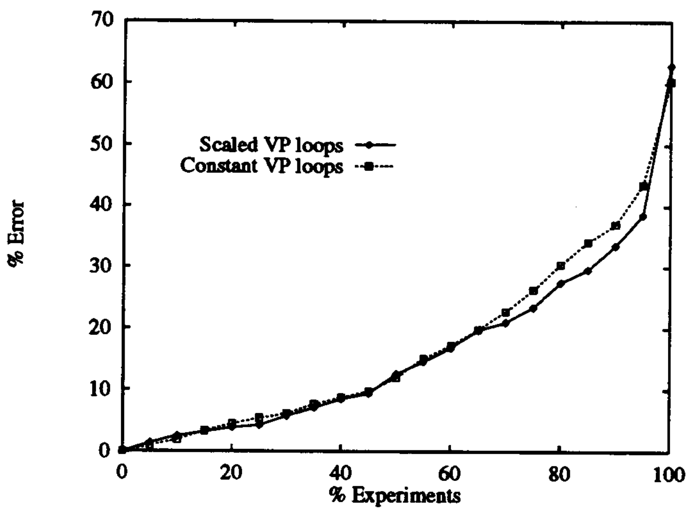


Figure 6.2. Percent error for constant virtual processor emulation loop overhead contrasted with the scaled model.

6.1.1 Layer

The Layer application implements a simple two-layer ocean circulation model with periodic boundary conditions on the north and south edges and wraparound boundary conditions on the east and west edges. This model is amenable to data-parallel treatment and naturally fits a two dimensional toroidal grid representation. The model is simplified by assuming that there are no intervening continents. The program iterates through 7500 iterations of updating local values based on the values of the nearest neighbors in the grid. In the Dataparallel C version of the program, each data point in the grid is treated as a virtual processor, regardless of the actual number of physical processors involved. The program can be configured to output graphical data to show changes in the data points that model the circulation over time. Figure 6.3 shows the error contour for this application.

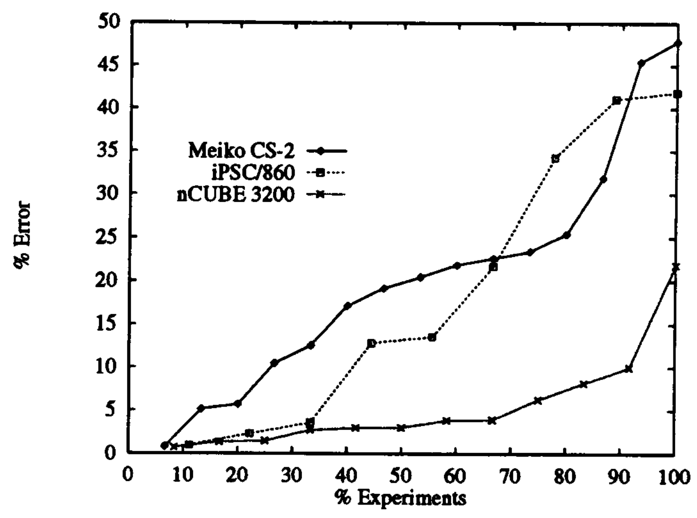


Figure 6.3. Percent error for experimental runs of the Layer application on the iPSC/860, Meiko CS-2 and nCUBE 3200.

6.1.2 Shallow

The shallow water equations were developed at the Laboratoire de Meteorologie Dynamique du C.N.R.S., Paris in order to investigate different finite-difference schemes. The National Center for National Center for Atmospheric Research, Colorado uses a model based on this work in benchmarking the performance of MPP systems.

The equations explain the flow of a two dimensional slightly compressible inviscid fluid:

$$\begin{aligned} \frac{\partial \vec{V}}{\partial t} + \eta \vec{N} \wedge (P\vec{V}) + \vec{\nabla}(P + \frac{1}{2}\vec{V}\cdot\vec{V}) &= 0 \\ \frac{\partial \vec{P}}{\partial t} + \vec{\nabla}\cdot(P\vec{V}) &= 0 \end{aligned}$$

where $\vec{V}$ is the velocity, P the density of pressure, η the potential vorticity, rot $\vec{V}/P$, and $\vec{N}$ a unit normal to the plane. The slightly compressible case is enforced by a balance condition for the initial fields [69]:

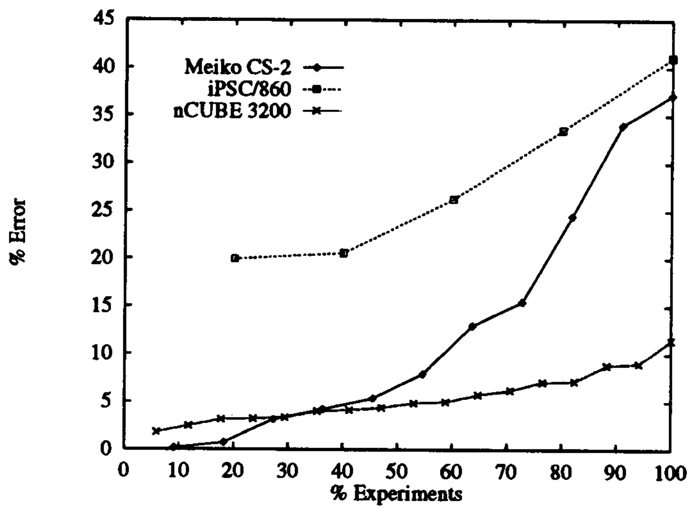


Figure 6.4. Percent error for experimental runs of the Shallow application on the iPSC/860, Meiko CS-2 and nCUBE 3200.

$$
\begin{aligned}
\vec{\nabla} \cdot \vec{V} &= 0 \\
\frac{\partial \vec{\nabla} \cdot \vec{V}}{\partial t} &= 0
\end{aligned}
$$

The program solves a set of nonlinear shallow water equations in two horizontal dimensions using a finite difference method [46]. It assumes periodic boundary conditions and uses a leap frog time differencing scheme so that the fluid flow is confined to the surface of a torus. Figure 6.4 shows the error contour for this application.

6.1.3 Matrix

Matrix multiplication is an important element of many scientific applications. When multiplying two $N \times N$ matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ in order to yield the $N \times N$ matrix C, $\Theta(N^3)$ operations will be performed. Because the computational complexity grows more quickly than the $\Theta(N^2)$ number of communications which must

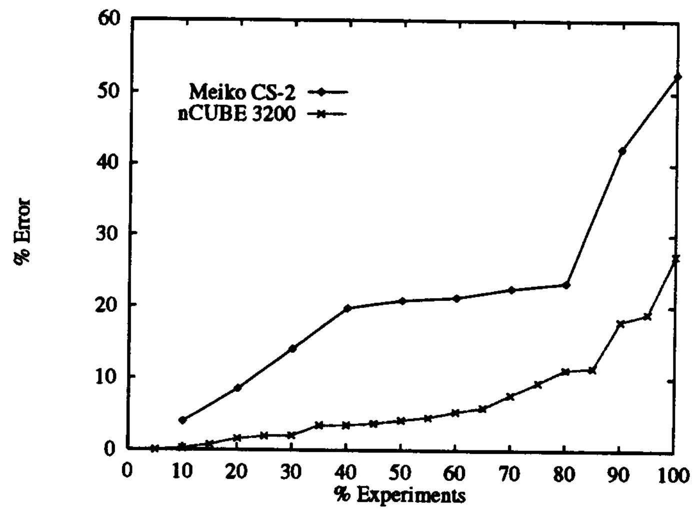


Figure 6.5. Percent error for experimental runs of of block matrix multiplication on the Meiko CS-2 and nCUBE 3200.

be performed, matrix multiplication can achieve high efficiency values on parallel machines if the problem size can be made sufficiently large. A block matrix multiplication was used for this validation application [80]. The algorithm breaks the A, B and C matrices into $\frac{N}{\sqrt{P}} \times \frac{N}{\sqrt{P}}$ blocks. One of these blocks is assigned to each processor. The algorithm memory locality and has been found to be highly efficient on processors with on-chip caches. Figure 6.5 shows the error contour for this application.

6.1.4 Gauss

This program solves the linear system $AX = b$ when the matrix A is a dense array. Gaussian elimination reduces the A matrix to an upper triangular system and then performs back substitution to compute the final X values [80]. In the implementation of this algorithm, two-dimensional data are distributed by rows to all processors. For each column of the A matrix, the row with the largest value

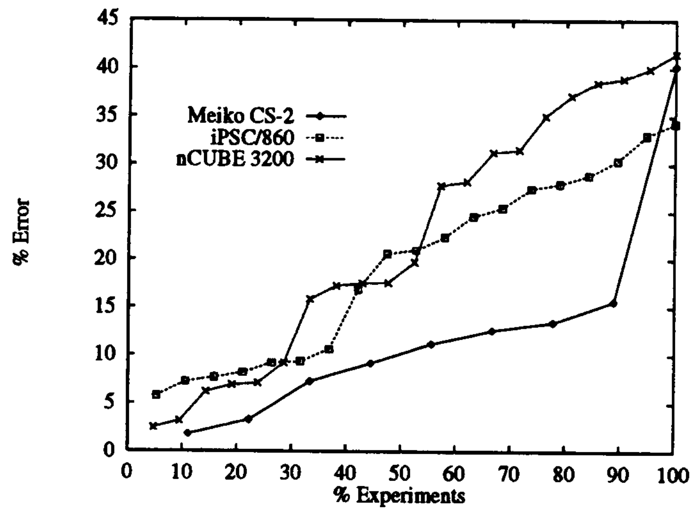


Figure 6.6. Percent error for experimental runs of Gaussian elimination on the iPSC/860, Meiko CS-2 and nCUBE 3200.

in that column is broadcast and used to reduce the remaining rows in the system. Figure 6.6 shows the error contour for this application.

6.1.5 Jacobi

The Jacobi application uses an iterative method to solve a system of linear equations. Iterative algorithms such as this are often used to solve the large, sparse linear equations generated from partial differential equations. This program uses Jacobi relaxation to solve for the steady state temperature distribution across a steel plate [80]. The plate is divided into square elemental regions, and the temperature is assumed to be constant across each region. A virtual processor is associated with each region. During an iteration of the algorithm, every virtual processor finds the average of the temperatures of each of its four adjacent regions in order to compute a next state temperature value. Figure 6.7 shows the error contour for this application.

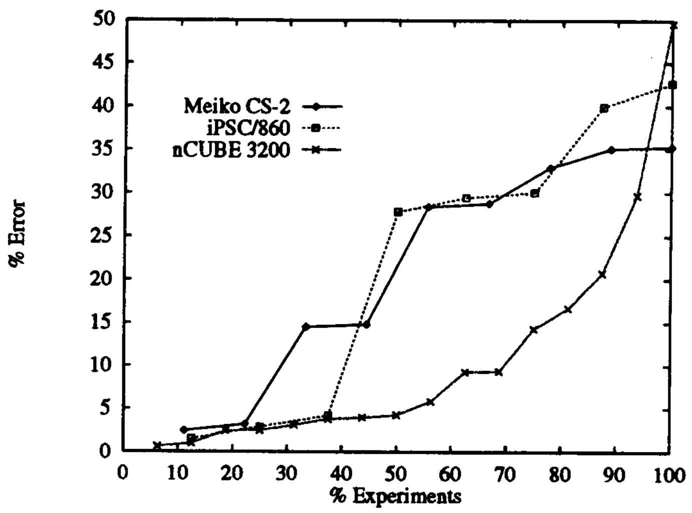


Figure 6.7. Percent error for experimental runs of the Jacobi application on the iPSC/860, Meiko CS-2 and nCUBE 3200.

6.1.6 Ocean

The Ocean application is derived from a model of wind-driven circulation in a stratified ocean [46]. The program employs finite element methods to implement a linearized, two-layer channel model using forward-backward schemes to calculate baroclinic and barotropic flows. Conventional ocean models have assumed that the ocean has a rigid "lid" which allows them to exclude the fast surface gravity waves and concentrate on general circulation. This program does not make this assumption since the resultant data dependencies would be difficult to accommodate on a massively parallel architecture. Instead, it uses the classical wave equation for displacement of the free surface. As a consequence, many small time steps must be simulated to insure computational stability. Figure 6.8 shows the error contour for this application.

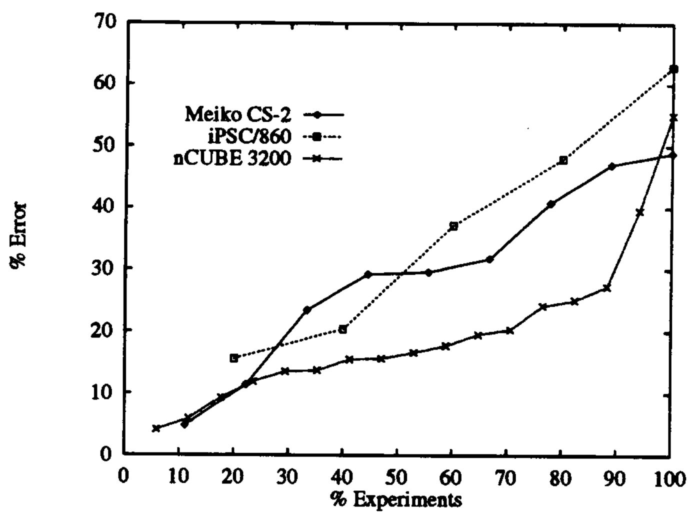


Figure 6.8. Percent error for experimental runs of the Ocean application on the iPSC/860, Meiko CS-2 and nCUBE 3200.

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.

6.2 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.

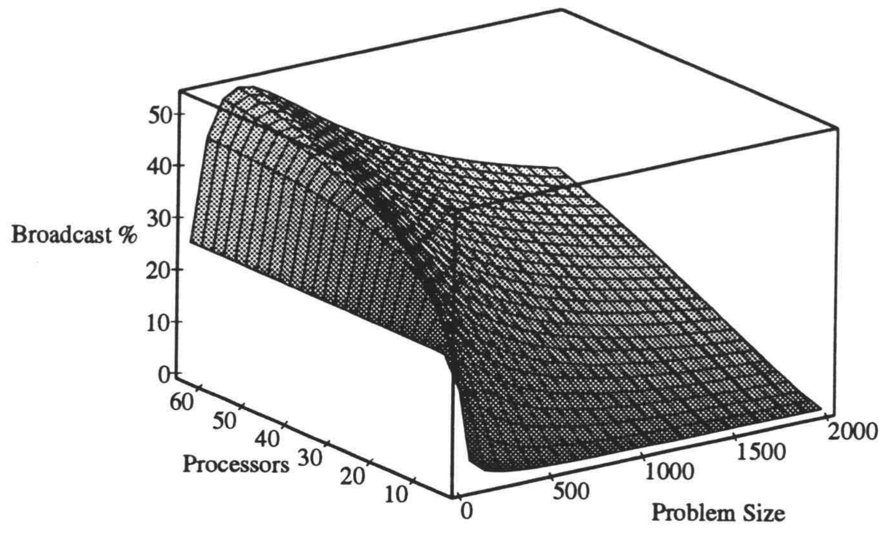


Figure 6.9. Percentage of execution time spent in broadcasting the pivot row for Gaussian elimination on an nCUBE 3200.

A fundamental use of performance prediction results is in the area of performance debugging. Figure 6.9 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 at an equal rate with P, the percentage of time spent in this basic block is constant. For medium sized problems 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.

Given our analytical model 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

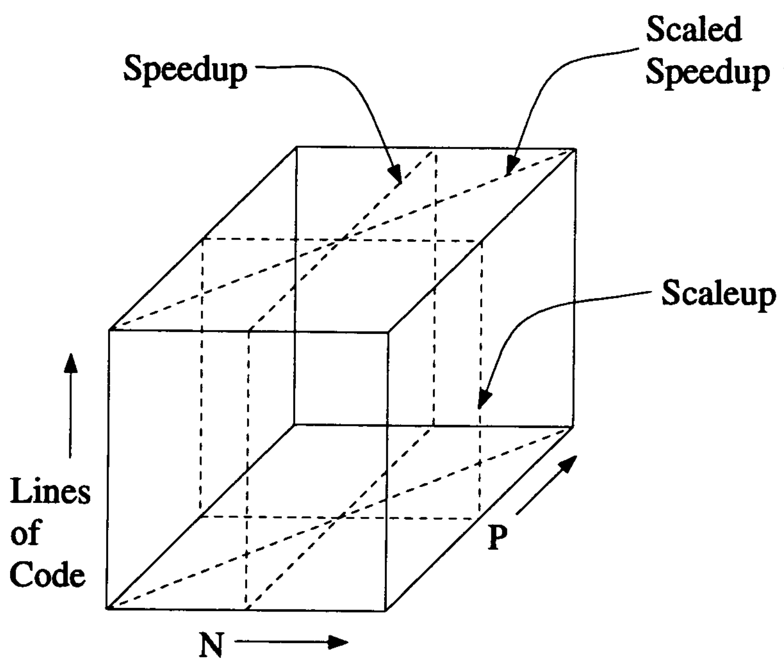
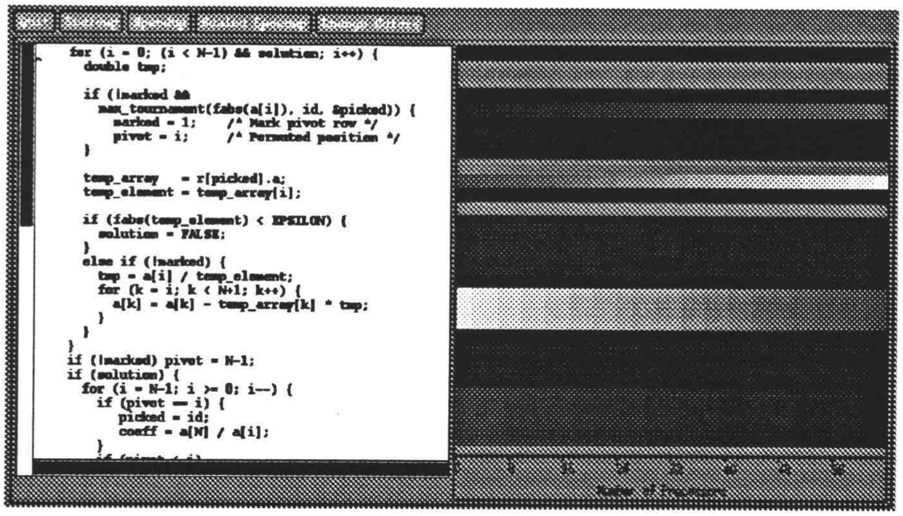


Figure 6.10. Views of basic block contributions to overall execution time under different scaling schemes.

metric can be examined by increasing the number of processors 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 6.10 summarizes the different views of an application under these different scaling schemes. Figure 6.11 shows the contributions of basic blocks in a Gaussian elimination application under speedup scaling.

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



```
for (i = 0; (i < N-1) && solution; i++) {
  double tmp;

  if (!marked &&
    max_tournament(fabs(a[i]), id, &picked)) {
      marked = 1;     /* Mark pivot row */
      pivot = i;      /* Permuted position */
  }

  temp_array   = r[picked].a;
  temp_element = temp_array[i];

  if (fabs(temp_element) < EPSILON) {
    solution = FALSE;
  }
  else if (!marked) {
    tmp = a[i] / temp_element;
    for (k = i; k < N+1; k++) {
      a[k] = a[k] - temp_array[k] * tmp;
    }
  }
}
if (!marked) pivot = N-1;
if (solution) {
  for (i = N-1; i >= 0; i--) {
    if (pivot == i) {
       picked = id;
       coeff = a[N] / a[i];
    }
```

Figure 6.11. Contributions of basic blocks in Gaussian elimination application with speedup scaling. Lighter colors indicate a higher percentage of time.

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 6.12, the execution time was differentiated with respect to message startup cost where

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

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.

Although theoretical comparisons of multicomputer topologies have been performed previously using asymptotic bounds, the additional detailed information

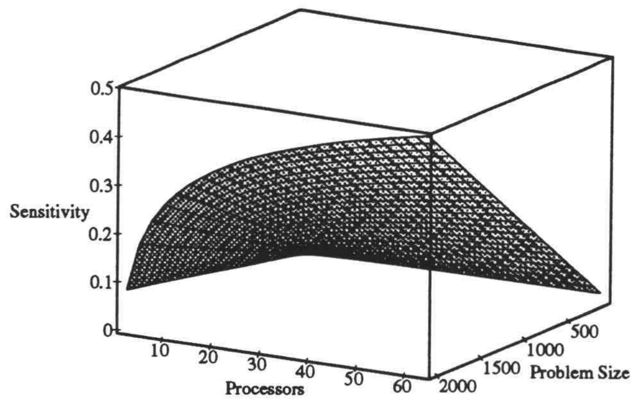


Figure 6.12. Sensitivity to changes in message startup cost for Gaussian elimination as a function of problem size and number of processors.

available from symbolic performance prediction allows us to compare network configurations for practical numbers of processors. Figure 6.13 compares performance for Gaussian elimination on the nCUBE 3200 and a network of SPARC workstations. For small numbers of processors the workstations are significantly faster than the multicomputer. As the number of processors and problem size increases, the bandwidth limitations of Ethernet reduce the performance of a workstation cluster and the nCUBE system becomes superior.

6.2.1 Cost Optimal Methods

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

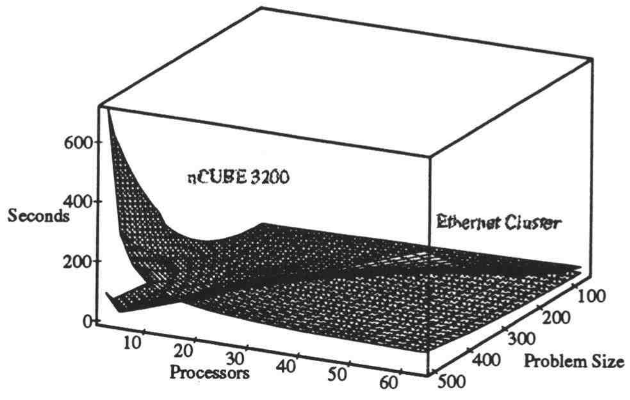


Figure 6.13. Execution time plots comparing the nCUBE 3200 with a network of workstations connected with Ethernet.

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 for the CPU, on-chip cache and the 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

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

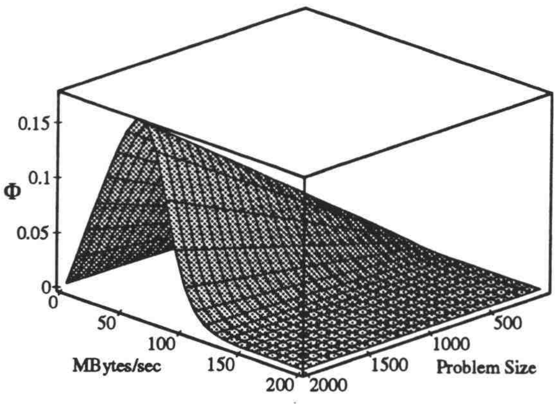


Figure 6.14. Predicted Φ for Gaussian elimination on 8 workstations as network bandwidth varies.

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 derived an expression for dollar cost as a function of network bandwidth

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

Figure 6.14 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 a network of 8 workstations.

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

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

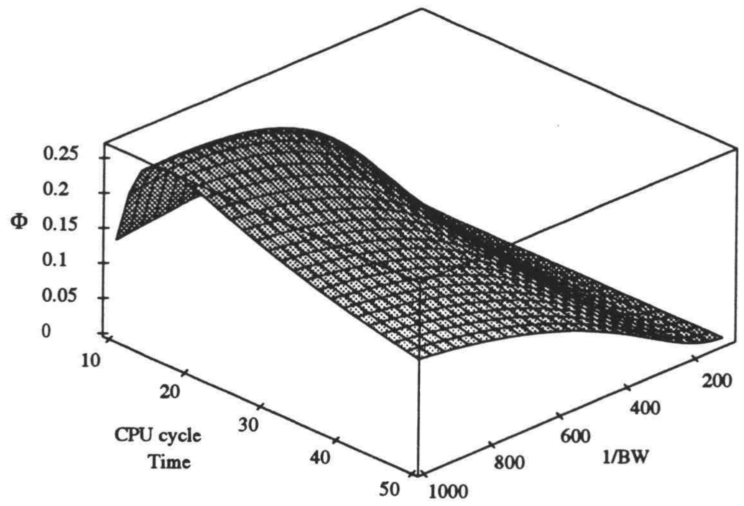


Figure 6.15. Predicted Φ for the Shallow Water Model as bandwidth and CPU speed vary.

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 6.15 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 size [47]. 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 [91]. The PowerPC 601 has 32Kbytes of cache with a total area of $119mm^2$ and sells for approximately

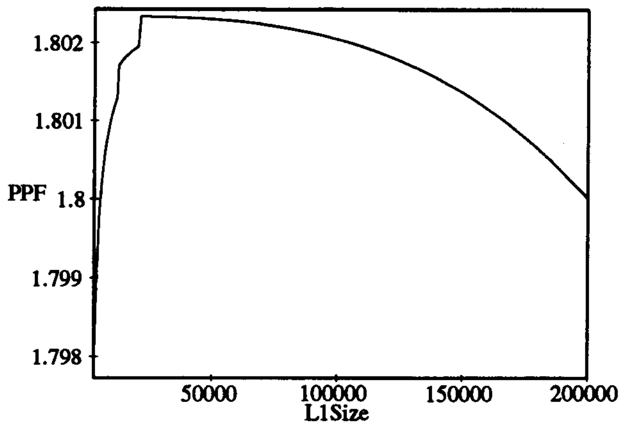


Figure 6.16. Price performance curve as cache size is varied for the Shallow Water Model.

\$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 6.16. 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 $\Phi =$ 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. Through 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 6.1 *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 6.17).

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.

$$\begin{aligned}
\Phi &= \frac{\mathcal{M}}{\mathcal{C}} \\
\Phi' &= \frac{\mathcal{M}'\mathcal{C} - \mathcal{M}\mathcal{C}'}{\mathcal{C}^2} \\
\Phi'' &= \frac{[(\mathcal{M}''\mathcal{C} + \mathcal{M}'\mathcal{C}') - (\mathcal{M}'\mathcal{C}' + \mathcal{M}\mathcal{C}'')]\mathcal{C}^2 - [2\mathcal{C}\mathcal{C}'(\mathcal{M}'\mathcal{C} - \mathcal{M}\mathcal{C}')]}{\mathcal{C}^4} \\
&= \frac{\mathcal{M}''\mathcal{C} - \mathcal{M}\mathcal{C}''}{\mathcal{C}^2} - \frac{2\mathcal{C}'}{\mathcal{C}}\left(\frac{\mathcal{M}'\mathcal{C} - \mathcal{M}\mathcal{C}'}{\mathcal{C}^2}\right) \\
&= \frac{\mathcal{M}''\mathcal{C} - \mathcal{M}\mathcal{C}''}{\mathcal{C}^2} - \frac{2\mathcal{C}'}{\mathcal{C}}(\Phi') \qquad (6.2)
\end{aligned}$$

The first term of Equation 6.2 is strictly negative since $\mathcal{M}'' < 0$ and $\mathcal{C}'' > 0$. If $\Phi''|_{x_0}^{x_2} > 0$ then the second term of Equation 6.2 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 6.3 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.

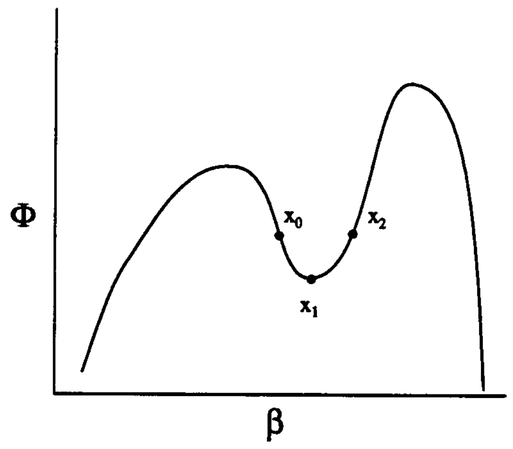


Figure 6.17. Price performance curve with two local maxima.

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 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 [9]. 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.

Application	Problem Size	β_{Ops} in nsec	$L1Size$ in bytes	β_{BW} in nsec/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 6.3. Cost optimal values for system parameters for applications in validation suite.

Chapter 7
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, users can optimize their selection of parallel platforms to use for production runs of scientific applications.

The static methods developed here can provide rapid feedback to an optimizing compiler on the relative benefit of alternate transformations. It can also be used to predict performance for static applications with too many computations to admit an instrumentation run.

The dynamic performance prediction techniques 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. Through 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.

Through preserving the symbolic equations which characterize the detailed behavior of each basic block in the application, the dynamic model 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.

7.1 Significance of Research

MPP system users, programmers and designers are all interested in performance analysis since their goal is to achieve the highest possible performance at the lowest cost. Performance prediction techniques can play a significant role in improving the efficiency of this environment. Historically, little effort has been placed on analysis methods which use actual applications. Instead, most of the research has concentrated on theoretical bounds as a means of characterizing the performance of systems. This work represents an important proof of concept, indicating the efficient analytical models can be based on real scientific programs. The resulting analysis information is shown to be important in solving important problems in this area.

Multivariate data analysis techniques simplify the performance prediction process by deriving system parameters from experimental runs of sample applications. The utility of prediction data is also increased as users are able to evaluate the confidence interval for estimated execution time. Improvements to the underlying analytical model can be made through focusing on predictor variables which have a large variance in the statistical analysis. This research is the first to investigate using statistical analysis techniques in this way. Multivariate statistical techniques improve the utility and accuracy of performance prediction models.

We have analyzed the needs of several groups requiring performance prediction information. This research is unique in being able to meet the needs of all these groups. 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.

7.2 Future Directions

Future work will focus on improving the accuracy of these modeling techniques and using them to solve other performance prediction problems. Through examining statistical data which evaluates the correlation of predicted and experimental performance, we plan to make improvements in the accuracy of the model. Additional analysis will also be performed to determine new model features which will allow performance prediction on a broader range of parallel programs and hardware architectures. We also plan to develop portability measures similar to the isospeed metric [90] which can be quantified through analyzing the sensitivity of applications to changes in system parameters. Finally, this modeling methodology will be integrated into a graphical performance tool.

The complexity of computations and communications for algorithms and the corresponding effects of topology and routing scheme for hardware platforms have been studied through analyzing asymptotic bounds. Through dynamic performance prediction, detailed complexity results can be obtained for specific algorithmic and architectural characteristics. We plan to conduct research comparing hypercubes, meshes and fat trees for several classes of real applications. This research will attempt to determine if the increased expense incurred in building more complex topologies is justified by the performance obtained.

The use of performance prediction systems by system architects and programmers should increase the efficiency of MPP systems in general and scalable applications in particular. Much work is left to be done as far as generalizing the model to other applications and architectures, but this approach has the potential to provide much needed information to the multicomputer user community. Performance prediction tools can aid multicomputer users and designers in increasing parallel efficiency on these machines. High efficiency parallel execution will be essential if “Grand Challenge” problems are to be solved on multicomputers.

BIBLIOGRAPHY

[1] S. Abraham and K. Padmanabhan. Performance of the direct binary n-cube network for multiprocessors. *IEEE Transactions in Computers*, 38(7):1000 - 1011, July 1989.

[2] I. F. Akyildiz. Performance analysis of a multiprocessor system model with process communication. *The Computer Journal*, 35(1), 1992.

[3] M. Annaratone and R. Ruhl. Balancing interprocessor communication and computation on torus-connected multicomputers running compiler-parallelized code. In *Proceedings SHPCC 92*, pages 358–365, March 1992.

[4] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. NAS parallel benchmark results. Technical Report RNR-93-016, Nasa Ames Research Center, October 1993.

[5] V. Balasunderam, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. *SIGPLAN Notices*, 26(7):213 - 223, July 1991.

[6] R. S. Barr and B. L. Hickman. Reporting computational experiments with parallel algorithms: Issues, measures, and experts' opinions. *ORSA Journal on Computing*, 5(1):2–18, Winter 1993.

[7] G. Bell. Scalable, parallel computers: Alternatives, issues and challenges. *International Journal of Parallel Programming*, 22(1):3–46, 1994.

[8] D. D. Bertsekas, C. Ozveren, G. D. Stamoulis, P. Tseng, and J. N. Tsitsiklis. Optimal communication algorithms for hypercubes. *Journal of Parallel and Distributed Computing*, 11:263–275, 1991.

[9] R. G. Born and J. R. Kenevan. Theoretical performance-based cost-effectiveness of multicomputers. *The Computer Journal*, 35(1):63–70, 1992.

[10] M. Calzrossa and G. Serazzi. Workload characterization: A survey. *Proceedings of the IEEE*, 81(8), August 1993.

[11] D. Case. Computer simulations of protein dynamics and thermodynamics. *Computer*, 26(10), October 1993.

[12] J. M. Chambers, W. S. Cleveland, B. Kleiner, and P. A. Tukey. *Graphical Methods for Data Analysis.* Wadsworth International Group, Belmont, California, 1983.

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

[14] B. M. Chapman, T. Fahringer, and H. P. Zima. Automatic support for data distribution on distributed memory multiprocessor systems. Technical Report TR 93-2, Institute for Software Technology and Parallel Systems, University of Vienna, 1993.

[15] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt. *Maple V Language Reference Manual.* Springer-Verlag, New York, 1991.

[16] A. Chen. No challenge too grand: Intel supercomputers in the mainstream. *Microcomputer Solutions*, November 1992.

[17] R. S. M. Christopher R. Johnson and M. A. Matheson. Computational medicine: Bioelectric field problems. *Computer*, 26(10), October 1993.

[18] M. J. Clement and M. J. Quinn. Architectural scaling and analytical performance prediction. Submitted to *The Seventh International Conference on Parallel and Distributed Computing Systems*, Las Vegas, Nevada, October 6-8, 1994.

[19] M. J. Clement and M. J. Quinn. Overlapping computations, communications and I/O in parallel sorting. Submitted to *Journal of Parallel and Distributed Computing*, October 1993.

[20] M. J. Clement and M. J. Quinn. Analytical performance prediction on multicomputers. In *Proceedings of Supercomputing '93*, pages 886–905, November 1993.

[21] M. J. Clement and M. J. Quinn. Symbolic performance prediction of scalable parallel programs. Technical Report 94-80-6, Department of Computer Science, Oregon State University, April 1994.

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

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

[24] R. G. Covingtion, S. Dwarkadas, J. R. Jump, J. B. Sinclair, and S. Madala. The efficient simulation of parallel computer systems. *International Journal in Computer Simulation*, 1:31–58, 1991.

[25] M. E. Crovella and T. J. LeBlanc. Performance debugging using parallel performance predicates. *Third ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1993.

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

[27] L. A. Crowl. Architectural adaptability in parallel programming. Technical Report 381, Department of Computer Science, University of Rochester, May 1991.

[28] W. J. Dally. Express cubes: Improving the performance of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 40(9):1016–1023, September 1991.

[29] R. T. Dimpsey and R. K. Iyer. Performance prediction and tuning on a multiprocessor. *Comput. Archit. News*, 19(3):190–199, May 1991.

[30] T. Fahringer. Evaluation of benchmark performance estimation for parallel Fortran programs on massively parallel SIMD and MIMD computers. In *Proceedings of the 1994 Euromicro Conference, Spain*, 1994.

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

[32] H. P. Flatt and K. Kennedy. Performance of parallel processors. *Parallel Computing*, 12:1 - 20, 1989.

[33] G. Fox. What have we learnt from using real parallel machines to solve real problems? Technical Report C3P-522, Caltech, December 1989.

[34] 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*, 16(2), February 1990.

[35] A. J. Goldberg and J. L. Hennessy. Mtool: An integrated system for performance debugging shared memory multiprocessor applications. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):28–40, January 1993.

[36] A. Y. Grama and V. Kumar. Scalability analysis of partitioning strategies for finite element graphs: A summary of results. In *Proceedings Supercomputing '92*, 1992.

[37] A. Gupta and V. Kumar. Performance properties of large scale parallel systems. *Journal of Parallel and Distributed Computing*, 19:234–244, 1993.

[38] A. Gupta and V. Kumar. The scalability of FFT on parallel computers. Technical Report TR-90-20, Computer Science Department, University of Minnesota, October 1992.

[39] A. Gupta and V. Kumar. The scalability of parallel algorithms for matrix multiplication. Technical Report TR-91-54, Computer Science Department, University of Minnesota, September 1992.

[40] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. In *Proceedings of the Sixth Distributed Memory Computing Conference*, pages 43–50, April 1991.

[41] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

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

[43] 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.

[44] 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.

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

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

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

[48] R. W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20:389 – 398, 1993.

[49] R. W. Hockney and E. A. Carmona. Comparison of communications on the intel iPSC/860 and touchstone delta. *Parallel Computing*, 18(9):1067 – 1072, 1992.

[50] 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.

[51] Intel Corporation. *Paragon OSF/1 C Compiler User's Guide*, January 1993.

[52] R. Jain. *The Art of Computer Systems Performance Analysis.* John Wiley and Sons, Inc., New York, 1991.

[53] S. L. Johnsson and C.-T. Ho. Optimum broadcasting and personalized communication in hypercubes. *SIAM J. Sci. Stat. Comput.*, 38(9):1249–1268, July 1988.

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

[55] A. Kapelnikov, R. R. Muntz, and M. D. Ercegovac. A methodology for performance analysis of parallel computations with looping constructs. *Journal of Parallel and Distributed Computing*, 14(2):105–120, February 1992.

[56] P. Kermani and L. Kleinrock. A tradeoff study of switching systems in computer communication networks. *IEEE Transactions on Computers*, C-29(12):1052–1060, December 1980.

[57] L. Kleinrock and J.-H. Huang. On parallel processing systems: Amdahl's law generalized and some results on optimal design. *IEEE Transactions on Software Engineering*, 18(5):434–447, May 1992.

[58] D. J. Kuck. What do users of parallel computer systems really need? *International Journal of Parallel Programming*, 22(1):99–127, 1994.

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

[60] V. Kumar and A. Gupta. Analyzing scalability of parallel algorithms and architectures. Technical Report TR-91-18, Computer Science Department, University of Minnesota, June 1991.

[61] J. R. Larus. Abstract execution: A technique for efficiently tracing programs. Technical Report TR912, University of Wisconsin Department of Computer Science, February 1990.

[62] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1992.

[63] C. Leopold. A fast sort using parallelism within memory. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pages 326–333, 1992.

[64] D. H. Linder and J. C. Harden. An adaptive and fault tolerant wormhole routing strategy for k-ary n-cubes. *IEEE Transactions in Computers*, 40(1):2–12, January 1991.

[65] V. Lo, S. Rajopadhye, M. A. Mohamed, S. Gupta, B. Nitzberg, J. A. Telle, and X. Zhong. LaRCS: A language for describing parallel computations for the purpose of mapping. Technical Report CIS-TR-90-16a, University of Oregon, 1990.

[66] V. W. Mak and S. F. Lundstrom. Predicting performance of parallel computations. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):257–270, July 1990.

[67] A. D. Malony. Event-based performance perturbation: A case study. *SIGPLAN Notices*, 26(7), July 1991.

[68] 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*, pages 187–192, Jan 1994.

[69] Meiko World Incorporated. *Weather Prediction with the Computing Surface*, 1992.

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

[71] D. Menasce, S. H. Hoh, and S. K. Tripathi. A methodology for the performance prediction of massively parallel applications. In *Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*, pages 250–257, December 1993.

[72] S. A. Moyer. Performance of the iPSC/860 node architecture. Technical Report IPC-TR-91-007, Institute for Parallel Computation, School of Engineering and Applied Science, University of Virginia, May 17, 1991.

[73] D. Muller-Wichards. Problem size scaling in the presence of parallel overhead. *Parallel Computing*, 17(12):1361 - 1376, December 1991.

[74] W. Oed. The Cray Research Massively Parallel Processor System. Technical report, Cray Research, 1993.

[75] S. W. Otto and M. Wolfe. The MetaMP approach to parallel programming. In *Proceedings of the Fourth IEEE Symposium on the Fronteers of Massively Parallel Computation*, October 1992.

[76] 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.

[77] J. H. Patel. Performance of processor-memory interconnections for multiprocessors. *IEEE Transactions on Computers*, C-30(10):771–780, October 1981.

[78] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing and scheduling programs on multiprocessors. In *Proceedings of the 1989 International Conference on Supercomputing*, August 1989.

[79] F. P. Preparata and J. Vuillemin. The cube-connected cycles: A versatile network for parallel computation. *Communications of the ACM*, 24(5):300–309, May 1981.

[80] M. J. Quinn. *Parallel Computing: Theory and Practice.* McGraw-Hill Book Company, New York, New York, 1994.

[81] D. Reed and R. Harrison. Performance characterization and evaluation. In *Proceedings of the Workshop and Conference on Grand Challenges Applications and Software Technology, GCW-0593, Pittsburgh, Pennsylvania*, pages 36–41, May 1993.

[82] E. Rothberg, J. P. Singh, and A. Gupta. Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. *Comput. Archit. News*, 21(2):14–25, May 1993.

[83] G. Saghi, H. J. Siegel, and J. L. Gray. Predicting performance and selecting modes of parallelism: A case study using cyclic reduction on three parallel machines. *Journal of Parallel and Distributed Computing*, 19:219–233, 1993.

[84] M. R. Sampford. *An Introduction to Sampling Theory.* Oliver and Boyd Ltd., Tweeddale Court, Edinburgh, 1962.

[85] C. L. Seitz. The cosmic cube. *Communications of the ACM*, 28:22–33, January 1985.

[86] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, April 1992.

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

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

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

[90] X.-H. Sun and D. T. Rover. Scalability of parallel algorithm-machine combinations. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):599–613, June 1994.

[91] T. Thompson. Power. *Byte*, August 1993.

[92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. Technical Report UCB/CSD 92/#675, Computer Science Division – EECS, University of California, Berkeley, CA 94720, March 1992.

[93] R. von Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. Technical Report 92-22, ICASE Final Report, June 1992.

[94] H. Wabnig and G. Haring. Petri net performance models of parallel systems - methodology and case study. In *Proceedings of PARLE'94 - Parallel Architectures and Languages Europe,Athens, Greece*, July 4-7 1994.

[95] H. Wabnig, G. Haring, D. Kranzlmuller, and J. Volkert. Communication pattern based performance prediction on the nCUBE 2 multiprocessor system. In *Proceedings of CONPAR 94 - VAPP VI, Linz, Austria*, September 1994.

[96] M. Wolfe. Experiences with data dependence and loop restructuring in the tiny research tool. Technical Report CS/E 90-016, Oregon Graduate Institute, 1990.

[97] P. R. Woodward. Interactive scientific visualization of fluid flow. *Computer*, 26(10), October 1993.

[98] X. Zhang. Performance measurement and modeling to evaluate various effects on a shared memory multiprocessor. *IEEE Trans. Software Engineering*, 17(1):87 – 93, January 1991.

APPENDIX

Appendix A
Statistical Consulting Study

The following recommendations were given to Mark Clement from the Department of Computer Science by Nobutaka Yagi from the Statistics Department in reference to the feasibility of using multiple regression techniques to estimate machine parameters for multicomputers.

Advice Given to Client

So far as the objective of the study is to specify the given linear model by estimating the basically unknown machine parameters (coefficients), the study statistically corresponds to a multiple regression analysis, where numbers of operations in each of particular preidentified sections in processing play a role of explanatory variables which would associate to the observed execution time. The biggest issues that I am concerned with in the study are the aspects of sampling and the analytical approach to a regression problem.

A.1 Sampling

A.1.1 Analysis

I think that the problem generated from a given program is the sampling unit of your study, and the sample population of the study would be all possible problems generated from the program. Therefore you will be able to make an inference on a program basis processed by a given architecture. One particular consideration is taken in terms of the size of your sample for multiple regression analysis. Details of this point will be discussed later but the major point is, generally speaking, that the size of sample, which is now specified as the number of observations, is to be at least several times larger than the number of parameters to be estimated in the model.

A.1.2 Recommendations

Your study would lead a best inference if you follow one of the sampling schemes based on probability sampling when you actually determine the problems to be used in your study. Primarily speaking, in probability sampling, each elementary unit (an individual unit of population being examined) has a known equal probability of being selected into a sample. The most simple technique to be concerned with here is the Simple Random Sample, in which you can sample as many units as you want by using, for instance, random numbers generated by a computer. One major disadvantage of Simple Random Sampling is that it is possible that you get a sample which only has many small sizes of problems. The alternative to avoid this situation and improve the validity of your estimates is achieved by using "Systematic Sampling". This technique is very advantageous when the sampling units are listed in order of a population characteristic of interest and can be used even if you really don't know the total number of the sampling units. For further information on Systematic Sampling, see any elementary sampling textbook. The analysis procedure of Systematic Sampling is the same as that of Simple Random Sampling.