Performance Prediction and Network Media Selection for PVM Clusters

A Thesis

Presented to the

Department of Computer Science

Brigham Young University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Michael R. Steed

April 1996

Performance Prediction and Network Media Selection for PVM Clusters

Michael R. Steed

Department of Computer Science

M.S. Degree, April 1996

ABSTRACT

Workstation clusters are becoming more popular as a parallel computing platform. Several programming libraries, including Parallel Virtual Machine (PVM), allow workstation clusters to be programmed as a single parallel computer.

The performance of parallel programs is often affected by complex interactions between hardware and software. Performance prediction tools can help characterize the performance of parallel programs for different problem sizes, numbers of processors, and hardware configurations. This research investigates the feasability of developing a performance prediction system for PVM applications running on workstation clusters. We have developed the APACHE (Automated Pvm Application CHaracterization Environment) performance prediction system for PVM programs. We review the system implementation and present some experimental results.

The performance of workstation clusters in parallel processing has traditionally been limited by the communication network. New networking technologies such as Asynchronous Transfer Mode (ATM) promise to relieve the performance bottleneck of workstation clusters. However, not all parallel applications will be helped by these new technologies. One objective of the current research is to investigate the use of information obtained from a performance prediction system as a basis for evaluating different network media for use with a particular set of parallel applications.

COMMITTEE APPROVAL:

Mark J. Clement, Committee Chairman

J. Kelly Flanagan, Committee Member

Tony Martinez, Committee Member

Tony Martinez, Graduate Coordinator

This thesis by Michael R. Steed is accepted in its present form by the Department of Computer Science of Brigham Young University as satisfying the thesis requirement for the degree of Master of Science.

____
Mark J. Clement, Committee Chairman

____
J. Kelly Flanagan, Committee Member

____
Tony Martinez, Committee Member

____
Date

____
Tony Martinez, Graduate Coordinator

Acknowledgements

I would like to acknowledge the help of my advisor, Dr. Mark Clement, who has always made himself available to discuss ideas and provide feedback. His enthusiasm and insights have been the basis for my interest in this research. Without his guidance and encouragement, this work would have been much more difficult and the result would have been of much lower quality. I would also like to acknowledge the valuable suggestions of the other members of my Thesis Committee, Kelly Flanagan and Tony Martinez.

This research would not have been possible without the support of the Brigham Young University Computer Science Department. I am grateful for the opportunity to have studied in the unique environment provided by the University, and I am indebted to the Department and the University for the facilities they provided. I am grateful for my association with the professors and students in the graduate program, who have helped me become a better student and a better human being.

Finally, I would like to thank my wife Caroline for her continued support and encouragement while I attended school. She has always been quick to encourage me in my studies. She has given up much of her time to allow me to finish graduate school. She has made the sacrifices that are part of life on a “student budget”. I dedicate this work to her.

Contents

Chapter 1 Workstation Cluster Parallel Computing

This chapter presents some of the issues involved in writing programs for parallel computers and introduces workstation clusters as a parallel computing platform. Section 1.1 explains what parallel computers are and reviews the common parallel programming paradigms and methods. Section 1.2 introduces two approaches to tuning the performance of parallel programs. Section 1.3 reviews the advantages and disadvantages of workstation clusters for parallel computing applications. Section 1.4 states the objectives of the current research. Finally, Section 1.5 reviews prior work related to this research.

1.1 Parallel Computers and Programs

In general, a *parallel computer* is a computer that has multiple processors connected by a communication network and is capable of using all its processors simultaneously to solve a single problem [50]. Increased performance is the primary reason for the development of parallel computers, and many argue that parallelism is the future of high-performance computing [45, 50]. Although parallel computers have the computational power necessary to solve many "grand challenge" problems [40, 50], it can be quite difficult to write programs that perform well on parallel machines.

1.1.1 Parallel Programming Paradigms

To solve a problem on a parallel computer, the programmer divides the problem into pieces which are then distributed across the processors of the machine. The problem may be divided according to its functions (*control* or *task parallelism*) or its data (*data parallelism*). For example, consider an image processing problem that requires a sequence of operations to be performed on an image. A control parallel approach might be to have each processor perform one operation on the entire image, and then pass the result to the next processor in pipeline fashion. A data parallel approach to this problem might be to assign a part of the image to each processor, have each processor perform the entire sequence of operations on its image fragment, and then combine the results from each processor to construct the final image. Many problems can be solved with a combination of control and data parallelism [50].

Data-parallel algorithms have several advantages over their control-parallel counterparts:

- Data-parallel algorithms are more common. Although data parallelism is not appropriate for all applications, it has been estimated that 90% of scientific and engineering problems are amenable to data-parallel solutions [28].

- Data-parallel algorithms are easier to design and debug. Control-parallel algorithms tend to suffer from time-related errors such as deadlocks and data incoherence [49]. Data-parallel algorithms, however, are logically synchronous and inherently deterministic.

- Data-parallel algorithms are better able to scale to large numbers of processors than control-parallel algorithms [50]. Since data-parallel algorithms generally work for a range of problem sizes, more processors can be efficiently utilized by solving larger problems. In contrast, the number of processors usable by control-parallel algorithms is limited by the problem decomposition.

Related to the data-parallel/control-parallel issue is Flynn's [27] classification of multiprocessor computer architectures as SIMD (single instruction stream, multiple data stream) or MIMD (multiple instruction stream, multiple data stream) [49]. In both SIMD and MIMD machines, each processor operates on its own local data. In a SIMD machine, all the processors execute the same operations simultaneously, proceeding in lock-step fashion under the direction of a single control unit. In contrast, each processor of a MIMD machine can execute a unique program.

By their nature, SIMD machines can exploit data parallelism but not control parallelism, while MIMD machines are able to exploit both control and data parallelism [50]. However, MIMD machines are usually programmed in data-parallel style [37]. Data-parallel programs written for MIMD machines are classified as SPMD (single program, multiple data streams). In SPMD-style programs each processor executes the same code in a “loosely synchronous” fashion—processors execute blocks of code asynchronously, and synchronize when they exchange data with other processors. This is in contrast with SIMD processors, which are “tightly synchronous”, operating in lock-step fashion.

1.1.2 Writing Parallel Programs

There are several ways to create programs for parallel computers.

- Write a program in a sequential language, such as Fortran or C, and let a parallelizing compiler transform the program for execution on a parallel machine. Although this approach allows the programmer to use familiar sequential languages and tools, automatic parallelization is limited to code fragments in which the compiler can detect potential parallelism (e.g., a set of nested **`for`** loops) [2]. Some transformations do not work well without the help of compiler pragmas or programmer interaction [33]. Also, some common programming constructs, such as **`while`** loops, are not amenable to straightforward transformations [51]. Current parallelizing compilers are effective on only a small fraction of existing scientific codes written in sequential languages [48].

- Use a data-parallel language. Data-parallel languages provide high-level constructs to express parallelism, thus freeing the programmer from worrying about the details of communication and process synchronization. Several researchers and hardware vendors have developed data-parallel languages, usually by adding data-parallel constructs to sequential languages such as C, Fortran, and Lisp [8, 30, 58]; see [34] for a description of several data-parallel languages.

- Use a sequential language augmented with message-passing constructs. Several programming libraries are available which support this approach, including Express [26], LINDA [12], MPI [41], p4 [10], PICL [31], and PVM [32]. Most of these libraries run on a variety of platforms, allowing programs to run with little or no modification on different machines. The primary disadvantage of the explicit message-passing approach is that it requires the programmer to manage the communication and synchronization aspects of programs. According to [49], this burden makes it difficult to write correct parallel programs. However, SPMD-style programs exhibit simple communication and process interaction patterns, making the message-passing approach quite tractable. In addition, this approach allows the programmer considerable freedom in optimizing program performance. Thus, sequential languages and message-passing libraries are widely used to write SPMD-style parallel programs [20, 32, 62].

The current research focuses on performance aspects of SPMD programs written to use the PVM message-passing library.

1.2 Performance Tuning

The performance of parallel programs is often affected by complex interactions between algorithm implementations, input data, and machine characteristics. Performance tuning, the process of optimizing a program so that it performs well on a parallel computer, can be a tedious and expensive process. This is cited as one reason that parallel computers do not enjoy widespread use [13, 16].

Considerable research has been done to develop tools to facilitate performance tuning. Two general approaches to performance tuning are performance debugging and performance prediction.

1.2.1 Performance Debugging

Performance debugging involves recording critical aspects of program behavior during execution (e.g., message traffic, the time spent by each processor in computation and communication, etc.) and then using this data in some type of "post-mortem" analysis. The instrumentation data is typically used as input to a visualization system which presents graphical views of program behavior. The programmer can use this visual information to identify the causes of performance problems and modify the program accordingly. Several instrumentation and visualization tools are available for parallel systems [35, 43, 53]; see [13] for a comparison of several such tools.

1.2.2 Performance Prediction

Performance debugging tools can help programmers understand the behavior of a particular program implementation on a particular machine. However, these tools are not well suited to determining the effects of system parameters on program performance. For example, using a performance debugging tool, it would be difficult to determine that the performance of one program is limited by message latency, while the performance of another program is bound by the speed of the processors in the parallel machine. Performance prediction is better suited to answering these types of questions.

Performance prediction attemps to model program performance as a function of the hardware and software characteristics of a system [16, 18, 57]. By changing these characteristics in the model, program execution time can be predicted for a variety of platforms and configurations.

1.3 Workstation Clusters for Parallel Computing

A *workstation cluster* is a group of interconnected workstations that can be used as a single computing resource [48]. Several advantages of workstation clusters as a parallel processing platform are often cited [7, 11, 48].

- Availability. If a node in a cluster fails, the work that it was doing can potentially be reassigned to the other nodes.

- Superior price–performance ratio. Clusters generally consist of machines that have very good performance for their price. The price–performance ratio of of workstations and personal computers has increased steadily in recent years and is expected to do so in the future [48].

- Ease of incremental growth. More workstations can be added to a cluster to increase its computational power while maintaining investments in existing hardware.

- Scalability. Theoretically, there is no limit to the number of workstations that can be added to a cluster. In practice, however, the communication network becomes a performance bottleneck in large workstation clusters [48].

The limiting factor in the performance of workstation clusters as a parallel computing platform has traditionally been the communication network. Workstations are typically connected with Ethernet, which often performs poorly under the load of parallel applications due to its high message latency and limited bandwidth. Emerging high-speed communication technologies, such as Asynchronous Transfer Mode (ATM) [46], promise to relieve the communication bottleneck of workstation clusters. However, not all parallel applications will be helped significantly by high-speed networks, particularly if message latency is not reduced significantly. There is a need for tools that allow programmers to predict the impact of different interconnection networks on their applications. One goal of the current research is to investigate the feasability of developing a performance prediction system for parallel programs running on workstation clusters.

There are several programming libraries that support parallel processing on workstation clusters. One of the most popular of these is Parallel Virtual Machine (PVM) [32], which is freely available and runs on a wide variety of platforms. We call a workstation cluster running PVM a *PVM cluster*. Although several performance debugging tools have been developed for PVM programs (see Section 1.5.2), to our knowledge none are capable of performance prediction.

1.4 Thesis Statement

The objective of the current research was to investigate the following propositions:

1. *It is feasible to build a performance prediction system for PVM applications running on workstation clusters.* We have developed the APACHE (Automated Pvm Application CHaracterization Environment) performance prediction system. The APACHE system works for PVM programs written in C that conform to the SPMD programming model (see Section 1.1.1). Chapter 2 describes the performance models on which the APACHE system is based. Chapter 3 describes the implementation of APACHE. Chapter 4 presents the

experimental results obtained with the performance prediction system. Finally, Appendix A contains the APACHE system documentation.

2. *Information obtained from a performance prediction system can be used as a basis for selecting the most appropriate network medium for a PVM cluster running a particular set of applications.* Section 4.3 describes how information obtained with the APACHE system can be used for network media selection for PVM clusters.

It is important to note that the APACHE system is intended as a "proof of concept". We made several simplifying assumptions in the design of the system that facilitate performance prediction but limit the applicability of the system; however, the system could easily be modified or extended to overcome these limitations (see Section 3.1).

1.5 Related Work

Prior work related to the current research consists of performance prediction systems for various platforms, and performance debugging tools for PVM applications.

1.5.1 Performance Prediction Systems

Several researchers have developed performance prediction systems [4, 16, 18, 21, 24, 36, 47, 59]. Much of the work that has been done in performance prediction is applicable only to a particular hardware platform or is capable of examining only isolated performance factors (e.g., number of processors or problem size but not both) [4, 24, 36, 47].

Some hardware-specific prediction systems use a "training set" approach. In this method, a sequence of operations (a training set) is performed on the machine of interest to capture relevant performance characteristics, which are then stored in a database for use by the prediction tool. Problems with this approach include the large effort required to collect, maintain, and verify training sets, and the fact that measurements taken during execution of the training set often differ from actual performance, due to interactions between register allocation, cache, and CPU pipeline status [25].

Other prediction tools use a "template" approach, in which a pre-constructed performance model (a template) is chosen and then parameterized by dynamic measurements. Problems with this approach include the difficulty of developing templates and tight restrictions on the form of programs (i.e., programmers must express applications in terms of templates) [18].

Clement [16] describes a technique for predicting the performance of data-parallel programs written in the C* data-parallel language. Static analysis is used to build a call graph of the program, and dynamic analysis determines the time spent in communication and computation. This information is combined with an analytical model of a hardware platform to produce an equation for total execution time. Using the Maple [15] symbolic computation system, which facilitates the manipulation of equations and mathematical expressions, this equation can be cast into a variety of forms to predict performance and reveal an application's sensitivity to various system parameters, such as message startup cost, network bandwidth, cache size, and so forth. This system is restricted to programs written in C* (and perhaps other data-parallel languages), and is not directly applicable to programs written to use explicit message passing.

The current research differs from the work done by Clement in several ways. First, the current work is concerned with PVM programs written in C. Because the work done by Clement is based on a compiler for the C* language, it is applicable only to applications written in that language. Second, the current research focuses on the development of a highly automated system. The system developed by Clement requires that some of the instrumentation code be modified or inserted manually. Third, the current work is not based on the uniform cost assumption (see Sections 2.1 and 4.2.3). The approach taken in the current research allows us to examine the validity of this common simplifying assumption.

1.5.2 Performance Debugging Tools for PVM Applications

Several existing performance debugging tools work with PVM programs. Recent versions of PVM include instrumented libraries for recording trace data, which the XPVM graphical interface [38] can use to provide runtime and post-mortem visualization of program execution. Because the trace data is stored in the Self-Defining Data Format (SDDF) used by the Pablo performance tools [52], it can be used with any visualization system that accepts SDDF files.

The Paradyn [42] performance tools, which are designed for runtime performance monitoring and bottleneck detection, have been ported to the PVM platform. Other debuggers are available, including Aims [61], Xab [5], and Xmdb [19].

Some graphical programming environments are available for PVM, including HeNCE [6] and VPE [44]. In addition to visual programming facilities, these environments also provide some debugging capabilities.

Some performance problems are more easily solved with performance debugging tools (such as those listed above) than with performance prediction tools. For example, a parallel debugger, such as Paradyn, would be helpful for detecting deadlocks in a parallel algorithm, while a performance prediction tool would be better at assessing the impact of different network media on an application's performance. To our knowledge, there are no tools capable of performance prediction of PVM programs.

Chapter 2 Performance Models

To predict the performance of a PVM application running on a workstation cluster, the APACHE system employs models that represent the performance characteristics of the workstations, the communication network, and the application. This chapter describes the models that were developed as part of APACHE. Section 2.1 describes the model that represents computation costs. Section 2.2 describes the models used to represent communication costs over Ethernet and ATM networks. Finally, Section 2.3 explains the construction of a PVM application model.

2.1 Workstation Computation Model

The computation model represents the portion of program execution time that is spent in computation. We have deliberately kept the model simple. For example, although we attempted to model cache and virtual memory effects as was done in [16], we found that these effects were quite difficult to predict consistently, particularly in a non-dedicated cluster environment. Chapter 3 reviews the implementation details of the APACHE system.

To model computation costs, we assign a cost factor C_i, measured in execution time, to each of the N_{ops} different operations that may be performed during program execution. For example, floating-point multiplication has a different cost factor than floating-point division.

We use simple benchmark programs and linear regression techniques [17, 60] to obtain values and confidence intervals for each C_i on workstation platforms of interest. Table 2.1 shows the means and standard deviations for some of the factors obtained from an

	Floating point		Integer	
Operation	$\overline{C_i}$	σ_{C_i}	$\overline{C_i}$	σ_{C_i}
Addition	0.0429	0.00014	0.0353	0.00028
Multiplication	0.0595	0.00039	0.2341	0.00027
Division	0.2107	0.00027	0.5064	0.00026

Table 2.1. Costs for some operations on an HP workstation (μsec).

HP 715/80 workstation. Note that some integer operations take significantly longer than their floating-point counterparts. Table A.2 in Appendix A contains a complete list of the operations we consider in the APACHE computation model as well as values obtained from HP workstations.

Given the cost factors C_i, we express the predicted computation time T_{comp} spent during program execution as

$$T_{comp} = \sum_{i=1}^{N_{ops}} C_i N_{op(i)} \tag{1}$$

where $N_{op(i)}$ is the number of times operation i is performed during program execution.

Other performance prediction systems are based on the *uniform cost assumption*, which is that all floating point operations have the same cost (measured in execution time) [50]. In contrast, the APACHE system accounts for the actual cost of each type of operation. The difference in execution time for a division compared to a multiplication, for instance, can be significant, particularly when a large fraction of the total execution time is spent in computation. One of the goals of the current research is to examine the validity of the uniform cost assumption (see Section 4.2.3).

2.2 Communication Models

The communication models represent the portion of program execution time spent in communication. We have developed models to represent the performance of PVM communications over Ethernet and ATM networks.

In SPMD-style programs, every processor runs the same program and operates on different data elements (see Section 1.1.1). Thus, SPMD programs usually exhibit regular communication patterns. For example, in our implementation of matrix multiplication (see Section 4.1.3), each processor communicates with its neighbor once per iteration. Thus, to simplify the communication model, we assume that in a configuration with P processors, the communications involve all P processors simultaneously, and that the machine load due to communications is balanced across the processors. There are some common communication patterns that violate this assumption, such as broadcast (one processor sending to $P - 1$ processors) and reduction (essentially $P - 1$ processors sending to one processor). However, Balasundaram [3] reports that a large percentage of real applications exhibit this type of regular, synchronous communication. Section 4.2.1.4 describes the experimental results obtained with a program that uses broadcasts and reductions.

Other researchers who have modeled the performance of interprocessor communication report that a simple model that accounts for message latency and network bandwidth gives adequate results [59]. According to this model, the predicted communication time T_{comm} spent during program execution can be expressed as

$$T_{comm} = \sum_{i=1}^{N_c}(\alpha + \frac{B_i}{\beta}) \qquad (2)$$

where N_c is the total number of communications per processor, α is the message latency, β is the network bandwidth, and B_i is the size of message i. We use this equation to model PVM communications between workstations connected with ATM.

With shared-medium networks such as Ethernet, contention for bandwidth can significantly affect network throughput. Therefore, to model the performance of PVM communications over Ethernet, we add a contention factor, γ, to Equation (2):

$$T_{comm} = \sum_{i=1}^{N_c} (\alpha + \frac{\gamma B_i}{\beta}) \tag{3}$$

According to our experiments, a contention factor of $\gamma = P$, where P is the number of processors, provides a good approximation of Ethernet contention, assuming that all P processors are communicating simultaneously.

We tested the validity of Equations (2) and (3) in the following way. We assumed that the P processors involved in a communication formed a logical ring. Each processor sent a 64-byte message to its right neighbor (and received a 64-byte message from its left neighbor), and then sent a 64-byte message to its left neighbor (and received a 64-byte message from its right neighbor). The time spent by each processor for the two sends and receives was divided by two to obtain the time for a single 64-byte communication (one send and one receive). This was repeated ten times for a total of ten samples per processor. Means and variances were computed using the times from all P processors. This process was repeated for messages of 128 bytes, 256 bytes, and so on.

Figure 2.1 shows the communication times predicted by Equations (2) and (3) and actual communication times for different message sizes and numbers of processors. (Times for ATM are shown for two processors; higher numbers of processors resulted in similar times.) The individual points in the figure show the mean communication times; we chose to omit the error bars from these points to make the figure less cluttered. The actual communication times exhibited high variance for four or more processors sending small messages, and relatively

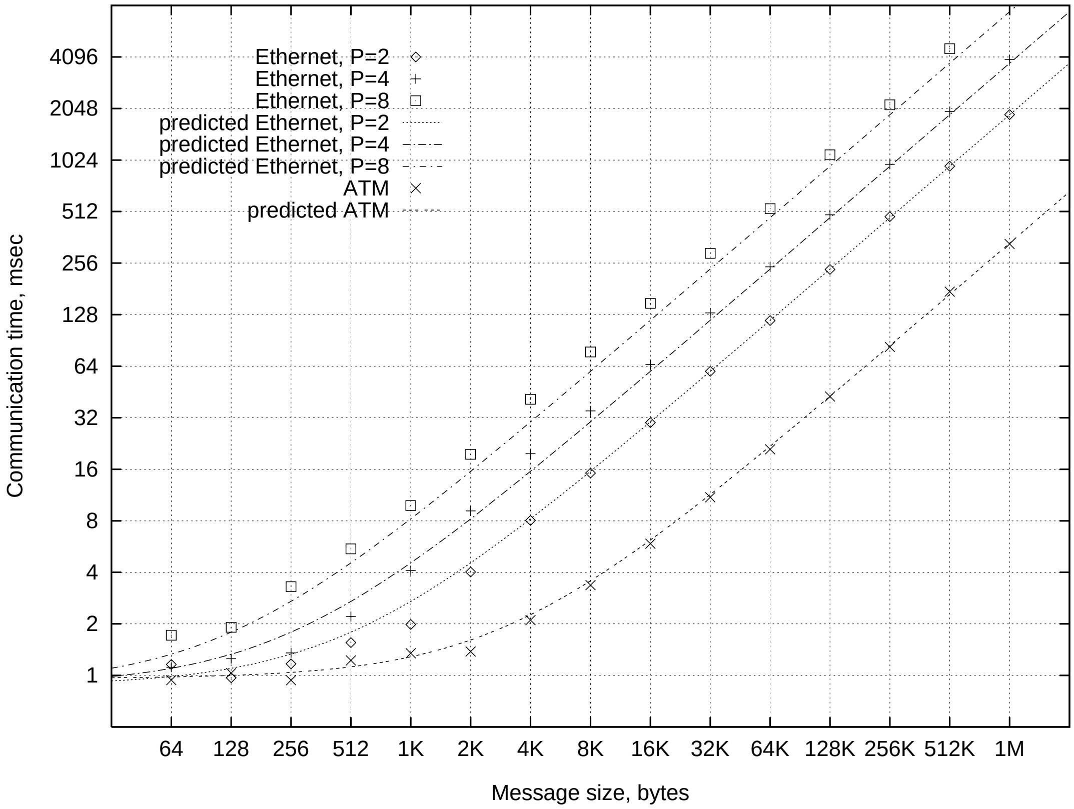


Figure 2.1. Predicted and actual times for PVM communications over Ethernet and ATM. Ethernet communication times are shown for different numbers of processors. Actual communication times for ATM are shown for 2 processors; higher numbers of processors resulted in similar times.

low variance for any number of processors sending large messages. One possible explanation for this behavior is variation in message startup time (i.e., the time from the invocation of the PVM communication routine until the message begins to be sent over the physical medium). Message startup time constitutes a greater percentage of the total communication time for small messages than for large messages. Variation in message startup time could produce the observed higher variance in communication times for small messages.

2.3 PVM Application Model

Equations (1) and (3) represent the computation and communication performance of a PVM cluster connected with Ethernet. Adding the two equations gives us an equation for predicted execution time:

$$T_{total} = T_{comp} + T_{comm} = \sum_{i=1}^{N_{ops}} C_i N_{op(i)} + \sum_{i=1}^{N_c} (\alpha + \frac{\gamma B_i}{\beta}) \qquad (4)$$

In this equation, $N_{op(i)}$, N_c, and B_i vary with the application and with the problem size and the number of processors used during program execution. A PVM application model consists of equations for these factors expressed as functions of the problem size, N, and the number of processors, P. (We use the term *problem size* to describe the application-dependent parameter that determines the size of the data set to be operated upon. For example, our matrix multiplication program, when run with a problem size N, multiplies two $N \times N$ matrices (see Section 4.1). In contrast, the problem size for our numerical integration program determines the number of subintervals that will be used in approximating the area under a curve. Throughout this document, we use N to represent the problem size and P to represent the number of processors used during program execution.)

To construct a PVM application model, we take the following approach. Given the application source code, we construct a call graph of the program at the C source code level [1]. Included in each block of the call graph is information regarding the operations that are performed in that block, including calls to the PVM communication routines. In addition to building the call graph, we produce an instrumented source code version of the original application. When the instrumented program is executed, the instrumentation code records the number of times each block is executed and the size of each PVM communication.

The information gathered by the instrumentation code at runtime is combined with the information recorded at compile time (i.e., the call graph) to obtain raw values for $N_{op(i)}$, N_c, and B_i for each block in the call graph. Given the problem size N and the number of processors P used during the instrumented run, simplified fits are performed to express $N_{op(i)}$, N_c, and B_i as

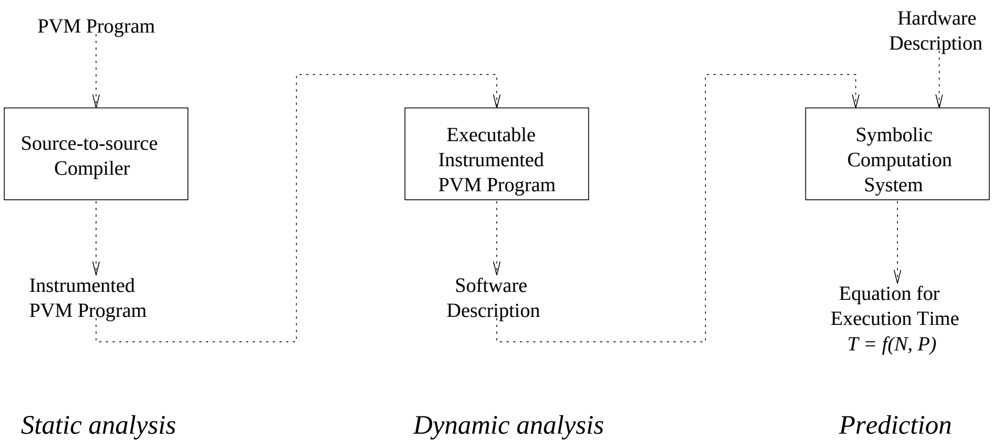


Figure 2.2. Building a PVM application model with APACHE.

functions of N and P. Combining the contributions of each block in the call graph, we obtain the final equations for $N_{op(i)}$, N_c, and B_i. These equations constitute the PVM application model.

The diagram in Figure 2.2 shows a high-level view of the process of using the APACHE system to obtain an equation for the execution time of an application. The "Software Description" referred to in the figure is the PVM application model (i.e., the equations for $N_{op(i)}$, N_c, and B_i), and the "Hardware Description" consists of the remaining (hardware-dependent) factors in Equation (4).

Chapter 3 discusses the implementation of the source-to-source compiler as well as our use of the Maple symbolic computation system [15].

Chapter 3 System Implementation

This chapter describes the implementation of the APACHE performance prediction system. Section 3.1 presents the goals we had and assumptions we made in the design of the system. Section 3.2 provides a logical view of the structure of the APACHE system. Section 3.3 describes the implementation of the source-to-source compiler, and Section 3.4 describes the prediction system implemented in the Maple language.

3.1 Design Considerations

Our goals in the design of the APACHE system included the following:

- Automation. The system should require little or no program modification or other manual intervention. There are situations where the APACHE system requires minor program modifications, usually in the form of compiler pragmas (see Section 3.3.3.2 and Section A.1.6 in Appendix A).

- Ease of use. The system should permit easy exploration of the effects of various system parameters, including problem size, number of processors, and network and processor characteristics. This is accomplished with the Maple symbolic computation system (see Section 3.4).

- Accuracy. A performance prediction system must be accurate to be useful. Some characteristics common to workstation cluster environments, such as non-exclusive use of computation and communication resources, make accurate predictions more difficult. However, based on our experiments, we believe that a prediction system for such environments can provide accurate performance prediction information (see Section 4.2).

- Versatility. The system should work for several types of computers and networks. The research presented here is based on the APACHE system as it was implemented on HP workstations connected with Ethernet and ATM, but it would not be difficult to extend the system to work with other types of workstations and communication media.

The performance of parallel programs is often affected by complex interactions between several factors. Because it can be quite difficult to accurately predict the performance of a parallel program, we have made several simplifying assumptions in the design of the APACHE system:

1. The system will be used only with applications that conform to the SPMD programming model (see Section 1.1.1). A minor qualification is needed on this point. Many PVM programs are written using a "master–slave" model of computation, which is a variation on the SPMD model [32]. In master–slave programming, one processor executes the *master* program, which is responsible for administrative functions such as spawning other processes, distributing data, coordinating the efforts of the other processors, and collecting the results when the computation is done. The other processors run the *slave* program, which does the "useful" computation. Since not every processor runs the same program, this is not true SPMD programming, but it is close enough for purposes of performance prediction. Since performance measures usually disregard the time spent performing administrative functions, the APACHE system can be used for programs that conform to the master–slave model; the slave program is simply treated as if it were a true SPMD program and the master program is disregarded.

   This first simplifying assumption is not a serious limitation. Fox [28] reports that 90 percent of scientific problems are amenable to SPMD solutions. Also, Geist [32] indicates that most PVM programs conform to a master–slave model of computation.

2. The nodes and network connections of the parallel machine are homogeneous. Although one of the most important features of PVM is its ability to combine heterogeneous computers and network connections into a single virtual parallel machine, this heterogeneity increases the difficulty of performance prediction. Therefore, we assume that all workstations have the same architectures and processor speeds, the same amount of cache and main memory, the same communication and I/O facilities, the same operating system, and so on. However, the results of the current research should be applicable to clusters composed of heterogeneous workstations that have similar performance characteristics. Clusters with a higher degree of heterogeneity could be accommodated by obtaining values for the parameters of the computation and communication models for each class of machine in the cluster, and then modifying the application performance model according to the distribution of PVM processes on the virtual machine.

3. Each processor in the system will execute a single PVM process, and the cluster is dedicated to executing the PVM application. PVM allows each node in the virtual machine to execute several processes simultaneously. However, the performance impact of running more than one task per processor is difficult to predict, due to contention for shared resources and interactions with the other processors in the system. For the same reason, we assume that the workstation cluster is "dedicated", i.e., no other user processes are executing while the PVM application is running. Performance prediction for non-dedicated clusters could be performed by modifying the performance models to account for the extra computation and communication load on the nodes of the parallel machine.

4. The PVM applications processed by the APACHE system must consist of a single function

(i.e., `main()`). This limitation is due not to the complexity of performance prediction, but to the difficulty of interprocedural analysis [54]. Although the APACHE system does not report an error when it encounters a function call, the portion of execution time spent executing subroutines is not reflected in the application performance model.

Other researchers who have developed performance prediction systems commonly assume that the parameter outcome for procedure call instantiations are independent of the call site (i.e., that procedures are always called with the same parameters). This assumption is made to prevent more expensive analysis techniques such as simulation [24]. Under this same assumption, the APACHE system could be modified to account for procedure calls by extending the routines that build the call graph and generate the performance equations. This is one possibility for future work (see Section 5.2).

5. Compiler optimizations are disabled. Because the APACHE computation model generates application performance models based strictly on C source code, the system cannot account for the operations performed during code optimization. Furthermore, the system is based on the assumption that, without the optimizer enabled, there is a close correspondence between the operations that appear in the C source code and the instructions that comprise the compiled object code. Our experiments indicate that this assumption is valid in most cases (see Section 4.2). One opportunity for future work is to integrate the performance prediction system with the compiler so that it can account for the operations performed during code generation and optimization (see Section 5.2).

3.2 Logical System Structure

The APACHE system can be viewed as consisting of three logical parts or phases: static

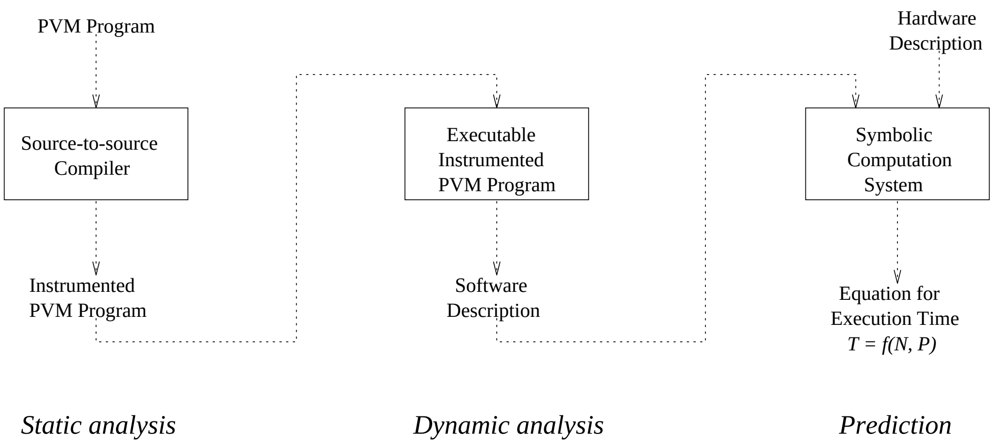


Figure 3.1. Logical structure of the APACHE system.

analysis, dynamic analysis, and prediction (see Figure 3.1).

The primary functions of the *static analysis* phase are (1) to extract information from the source code of a PVM application about program structure; and (2) to produce an instrumented version of the PVM application for use in the dynamic analysis phase.

The aims of *dynamic analysis* are (1) to run the instrumented PVM application to collect runtime information about the program, including the number of times each block is executed and the number and size of PVM communications; and (2) to express the metrics collected during execution as functions of the problem size N and the number of processors P used during the instrumented run. Section 3.3.2 discusses the implementation of static analysis phase, and Section 3.3.3 describes the implementation of the dynamic analysis phase.

The purpose of the *prediction* phase of the APACHE system is to support performance prediction tasks by manipulating the hardware and software models produced during static and dynamic analysis. Section 3.4 discusses the implementation of the prediction phase. Section 4.2 provides examples of the types of performance prediction activities supported by APACHE.

The primary tools comprising the APACHE system are a source-to-source compiler, which implements the static and dynamic analysis phases, and a set of procedures, implemented in the programming language of the Maple symbolic computation system, which support performance prediction activities. In the remainder of this chapter we review the implementation of the three phases of the APACHE system in the context of these tools.

3.3 Source-to-Source Compiler

This section presents the implementation details of the source-to-source compiler, which implements the static and dynamic analysis phases of the APACHE system.

3.3.1 General Implementation Issues

The APACHE source-to-source compiler is based on a lexical analyzer and a parser for ANSI C [55]. These were generated using the GNU **`flex`** and **`bison`** utilities from freely available specification files.[1]

Some modifications were made to the specification for the lexical analyzer. The most significant change was to add support for the C **`typedef`** construct. Although the lexical analyzer specified by the ANSI standard does not distinguish between identifiers and user-defined type names, the need to do so arises when parsing C programs. Therefore, we extended the lexical analyzer with functions to recognize user-defined type names. To simplify the implementation, we assume that all user-defined types are visible from the point where they are defined through the end of the file. This differs slightly from the ANSI standard, which allows **`typedefs`** with block scope [55].

The other major change to the specification of the lexical analyzer was to add support for C

[1]Available via anonymous ftp from **`ftp.uu.net`**, file **`/usenet/net.sources/ansi.c.grammar.Z`**

preprocessor directives. Specifically, the modified lexical analyzer processes compiler pragmas specific to the APACHE system (see Section 3.3.3.2 and Section A.1.6 in Appendix A), and echos the standard C preprocessor directives so they can be included in the instrumented program.

The changes to the parser specification consisted of extensions for producing the call graph and the instrumentation code. The structure and use of the call graph is discussed in the next section.

3.3.2 Static Analysis

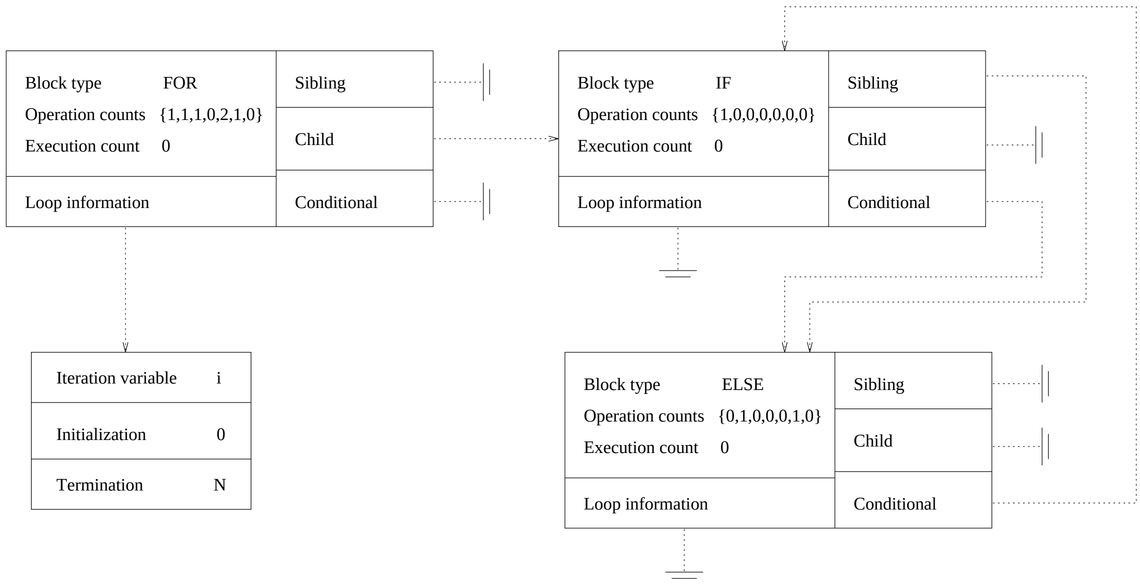
The static analysis functions of the source-to-source compiler are (1) to build a call graph of the PVM application at the source-code level; and (2) to produce a version of the PVM application containing appropriate instrumentation code for use during dynamic analysis.

3.3.2.1 Call Graph

A *call graph*, as used in the APACHE system, is a graph representing the structure of a program [1]. The nodes of the graph represent the basic blocks of the program (i.e., sections of code such that control may pass into the block only at the first statement in the block and leave the block only after the last statement). An edge between two nodes indicates some relationship between the basic blocks, usually that control can pass from one block to the other during program execution.

A simple call graph, and the code fragment it represents, appear in Figure 3.2. The fields in the nodes of the call graph are:

- Block type. A block can have one of five types: **`IF`**, **`ELSE`**, **`FOR`**, **`WHILE`**, or **`DO`**. Each type of block is handled somewhat differently during static and dynamic analysis. For example, only **`FOR`** blocks need pointers to loop information structures.




```
for (i = 0; i < N; i++) {
   if (A[2 * i + j] == 0)
      k++;
   else
      A[j + k] = 1;
}
```

Figure 3.2. Simple call graph generated by the APACHE source-to-source compiler, and the code fragment it represents.

- Operation counts. This array indicates the number of times each type of operation appears within the block. Counts are maintained for each of the operations included in the APACHE computation model (see Table A.2 in Appendix A) and for the number and size of PVM communications.

- Execution count. This field counts the number of times the block has been executed. It is used only during dynamic analysis (see Section 3.3.3).

- Loop information. This field, used only in **FOR** blocks, contains a pointer to a loop information structure indicating the iteration variable, the loop initialization expression, and the loop termination condition.

- Sibling. This field contains a pointer to the next block at the same level as the current block. A null value indicates that there is no such block.
- Child. This field contains a pointer to the first block nested within the current block. The pointer may be null.
- Conditional. This field is used by **`IF`** and **`ELSE`** blocks to link corresponding conditional blocks. This information is used during dynamic analysis to determine branch ratios of **`if-else`** constructs.

As the application source code is parsed, the source-to-source compiler builds the call graph and populates the fields describing each basic block. The call graph information is written to a file and eventually becomes part of the instrumented PVM program.

3.3.2.2 Instrumentation Code

In addition to building the call graph, the source-to-source compiler produces an instrumented version of the PVM application. As the program source code is parsed, the original source code is combined with instrumentation code and written to a file. The instrumentation code consists of statements to count the number of times each basic block is executed, interfaces to the PVM communication routines that record the size of each message transmitted, and additional library routines used during dynamic analysis. As part of the instrumentation process, the **`main()`** function of the original program is renamed to **`main_instrumented()`**. A replacement **`main()`** is added to the program which calls **`main_instrumented()`** as well as other library routines used during dynamic analysis.

3.3.3 Dynamic Analysis

The primary tasks involved in dynamic analysis are the collection of runtime data and the generation of equations that predict the behavior of the program for different problem sizes and numbers of processors.

3.3.3.1 Collection of Runtime Data

While the instrumented PVM program is executing, instrumentation code records the number of times each block is executed and size of each PVM communication. Although the data collection process is fairly straightforward, there is one subtle issue that arises regarding the size of PVM communications. It is possible for a program statement that calls a PVM communication routine to send a message of a different size each time the statement is executed. However, Balasundaram [3] reports that the communications in a large percentage of real applications involve regular, predictable message sizes. Therefore, we assume that each program statement that calls a PVM communication routine uses the same message size in all invocations. The APACHE system could easily be modified to handle irregular sized messages, for example, by averaging the message sizes over all the invocations during the instrumented run.

3.3.3.2 Generation of Equations

When control returns from the **`main_instrumented()`** function of the instrumented program, the fields of the call graph have been completely populated (see Section 3.3.2.1). Two things remain to be done to complete the dynamic analysis phase. First, the execution counts of each block need to reflect their "per-iteration" values. This can be illustrated by considering the following instrumented code fragment:

```
for (i = 0; i < 2 * N; i++) {
   bb14.count++;                        /* instrumentation */
   for (j = 0; j < N; j++) {
      bb15.count++;                     /* instrumentation */
```

```
            A[i][j] = i * j;
        }
    }
```

The lines marked **instrumentation** are typical of those inserted during static analysis to record block execution counts. The structures **bb14** and **bb15** are nodes in the call graph, and the **count** field contains the execution count of each block.

Suppose that this code is executed with $N = 128$. The outer block will be executed $2N = 256$ times, and the inner block will be executed $2N^2 = 32768$ times. Ultimately, we need to express the execution counts of these blocks as functions of the problem size N and the number of processors P. Before we can do this, these counts must reflect their "per-iteration" values. For our example, the per-iteration count of the inner block is $\frac{32768}{256} = 128$, which is equal to N. In other words, the inner block is executed 128 times for each execution of the outer block. Thus, to change absolute block execution counts into per-iteration counts, we traverse the call graph of the program and divide the execution count of each block by the count of its surrounding block. The execution count of the outermost block in any program (i.e., the body of the **main()** function) will not change since that block is executed only once.

Conditional constructs are handled in a slightly different manner. For example, consider this code fragment:

```
1.      for (i = 0; i < N; i++) {
2.         bb26.count++;                       /* instrumentation */
3.         if (B[i] == 0) {
4.            bb27.count++;                    /* instrumentation */
5.            A[i] = 2 * j;
6.         }
7.         else {
8.            bb28.count++;                    /* instrumentation */
9.            A[i] = j ^ 2;
10.        }
11.     }
```

The conditional construct on lines 3–10 constitutes two basic blocks, each with its own execution count. Since it is guaranteed that the sum of the execution counts for the two blocks will equal N, we compute the branch ratio from the instrumented run for the two blocks and assume that this ratio will remain constant for different values of N. Although this assumption is generally valid, in some cases it is not. For example, if the test on line 3 is designed to succeed for exactly one iteration of the enclosing loop, the branch ratio will change for different values of N. In such cases, APACHE will predict block execution counts inaccurately. However, we believe that this type of inaccuracy is infrequent enough that we can safely ignore it. Fahringer [24] reports that branch ratios usually remain constant for different problem sizes.

The C **`switch`** construct is more difficult to instrument correctly than the **`if-else`** construct, since it can have an arbitrary number of accompanying **`case`** clauses; there may or may not be a **`default`** clause; and there is no guarantee regarding the number of branches that will be followed each time the **`switch`** is executed. For these reasons, and because an equivalent construct can be implemented with **`if-else`** if necessary, the **`switch`** statement is not supported by the APACHE system.

Once block execution counts are expressed in "per-iteration" form, the final part of dynamic analysis is to generate a set of equations describing the predicted behavior of the program. This includes equations for the number of messages and the number of bytes transmitted, and equations for each of the operations considered in the computation model (see Section 2.1 and Table A.2 in Appendix A). These equations, which conform to the syntax of the Maple symbolic computation system [15], are expressed as functions of N, which represents the problem size, and P, which represents the number of processors.

To generate these equations, the call graph is traversed and the contribution of each block is accounted for. For blocks that are part of conditional constructs, the branch ratio computed earlier is included in the equation. For blocks that form the body of a loop structure (**`for`**,

while, or **do**), the execution count may need to be expressed as a function of N or P (or both). The following cases are examined in order until one is found to hold (the first condition applies to all types of loops, and the last three apply only to **for** loops):

1. A **#pragma complexity** directive was given inside the block (see below). If a complexity pragma is given, the actual execution count is ignored. This pragma is required for **while** and **do** loops.

2. The loop termination condition involves the iteration variable of an enclosing loop. This condition is checked by examining the loop information extracted from the program source code and stored in a loop information structure in the call graph (see Section 3.3.2.1).

3. The loop termination condition involves a constant expression. This is also determined from information stored in a loop information structure.

4. The execution count is a simple function of the problem size, the number of processors, or both. This case is checked by comparing the execution count with a set of predefined functions of N and P evaluated with the problem size and number of processors used during the instrumentation run.

If none of these conditions hold (e.g., because a loop iterates an unusual number of times), an error is reported and the dynamic analysis process is interrupted. In such cases, the time complexity of the block can be specified manually by inserting a compiler pragma of the form

#pragma complexity $f(N, P, \ldots)$

somewhere inside the block. Here $f(N, P, \ldots)$ is a Maple expression potentially involving N, P, and iteration variables of enclosing loops, that describes the number of times the block is executed. (Section A.1.6 in Appendix A documents all the pragmas supported by the APACHE system.)

An important assumption made by the APACHE system is that the only change to the iteration variable of a **`for`** loop is a single increment (or decrement) per iteration, and that control does not pass outside of the loop prematurely (e.g., with a **`break`** statement). Loops that violate this assumption should contain a complexity pragma.

Figure 3.3 shows one of the "raw" equations generated by an instrumented matrix multiplication program. Similar equations are generated for each of the operations in the computation model and for the number of messages and the number of bytes transmitted. These equations constitute the "Software Description" indicated in Figure 3.1 on page 22.

3.4 Prediction Utilities

The prediction portion of the APACHE system consists of several procedures, implemented in the programming language of the Maple symbolic computation system [15], designed to facilitate performance prediction. The Maple programming language supports the manipulation of arbitrarily complex symbolic expressions and equations and the construction of 2-D and 3-D plots and animations, which makes it well suited to performance prediction applications.

Descriptions of the main procedures in the prediction portion of the APACHE system follow:

- The **`predict`** procedure initializes the prediction system for use with a particular combintation of workstations, communication network, and PVM application.
- The **`dissect`** procedure shows the predicted contribution to program execution time of message latency, message bandwidth requirements, and the different operations in the

```
N_message := ( sum(0 + sum(1 + sum(0 + sum(0 + sum(0, 'k'='0+1'..'N'),
    'j'='1'..  'N/P'), 'i'='1'..'N/P'), 'iter'='1'..'P') + sum(0,
    'i'='1'..'N/P') + sum(0, 'i'= '1'..'N/P') + 0.000000 * sum(0,
    'foo'=1..1) + 1.000000 * sum(0, 'foo'=1..1) + sum(0 + 0.333333 *
    sum(0, 'foo'=1..1), 'slaveno'='1'..'P/2') + sum(0, 'i'='1'..'N/P')
    + sum(0, 'i'='1'..'N/P') + sum(0, 'i'='1'..'N/P'), 'foo'=1..1) );
```

Figure 3.3. Maple equation for the number of messages transmitted by the matrix multiplication program (see Section 4.1.3). Maple simplifies this equation to **`N_message := P`**.

APACHE computation model.

- The **`errs`** procedure compares predicted and actual execution times. This procedure reads a file containing a set of tuples of the form $[N, P, Time]$, indicating actual execution times for given values of N and P, and compares the actual times with predicted times for the same values of N and P. The procedure also plots a profile of the prediction errors.

- The **`grainsize`** and **`probsize`** procedures predict the grain size of an application for a given problem size and number of processors, and predict the problem size necessary to achieve a certain grain size with a given number of processors. These procedures were developed for use in network media selection (see Section 4.3).

Detailed 2-D and 3-D plots showing predicted program performance characteristics can be produced with simple Maple commands. Also, because the Maple language is interpreted and has a simple syntax, the prediction portion of the APACHE system is easily modified and extended.

Prediction is the last stage in the logical structure of the APACHE system (see Figure 3.1 on page 22). Combining the cost factors that describe the system hardware with the set of equations generated during dynamic analysis, we obtain an equation for predicted program execution time. As implemented, the APACHE system uses twelve equations to describe the behavior of a program. Even for simple programs, these equations can be quite large (Figure

3.3 shows one of these equations for matrix multiplication). One of the benefits of using a symbolic computation system is the automatic simplification of these equations. For example, Maple reduces the twelve equations for the matrix multiplication program to the following:

$$Time := 0.1000000000\ 10^{-12}\ (0.8714664715\ 10^{10}\ P^2 + 0.1714325\ 10^7\ N\ P + 0.6259559\ 10^7\ N^3 + 0.3474991\ 10^7\ N^2 + 0.35888000\ 10^8\ P^2\ N^2)\ /\ P$$

In the next chapter, we present some experimental results obtained with the APACHE performance prediction system.

Chapter 4 Experimental Results

This chapter presents the experimental results obtained with the APACHE performance prediction system. Section 4.1 describes the algorithms used in the experiments. Section 4.2 evaluates the accuracy of the predictions made by the APACHE system and illustrates some of the prediction activities that the system supports. Finally, Section 4.3 describes how information obtained with the APACHE system can be used to help determine the most appropriate network medium for a given application.

4.1 Algorithms

We implemented several parallel algorithms using PVM to test the performance prediction system. In this section, we present an overview of the algorithms and discuss the complexity of their computation and communication.

4.1.1 Jacobi Algorithm

The Jacobi algorithm is an iterative method for solving the linear system **Ax** = **b** for the unknown vector **x** [50]. The algorithm uses estimates for each element of **x** obtained during iteration t to generate new estimates for iteration $t + 1$:

$$x_i(t+1) = \frac{1}{a_{i,i}}(b_i - \sum_{i \neq j} a_{i,j}\, x_j(t))$$

The Jacobi algorithm can be used to solve the two-dimensional steady state temperature distribution problem over an $N \times N$ grid. In this problem, the cells of the grid are initialized with initial temperature values, some of which are fixed (i.e., as heat sources or heat sinks). During each iteration of the algorithm, a new temperature value for each cell is computed with the formula

$$x'_{i,j} = \frac{x_{i-1,j} + x_{i,j-1} + x_{i+1,j} + x_{i,j+1}}{4}$$

The values on the right side of the equation represent the temperature of the four-neighbors of cell (i, j) during iteration t, and the value on the left is the new temperature for that cell for iteration $t + 1$.

In our implementation, called **`jacobi`**, we parallelize the Jacobi algorithm by mapping $\frac{N}{P}$ contiguous rows of the grid onto each processor. During each iteration, processors exchange values of neighboring rows. Generally, the Jacobi algorithm iterates until the maximum change across the cells of the grid is smaller than some value ε. However, we found that the number of iterations required to reach a steady state is not a simple function of the problem size, but depends on the initial temperature distribution as well. Therefore, our implementation iterates a fixed number of times independent of the problem size.

The computational complexity of **`jacobi`** is $O(N^2)$, while the number of bytes transmitted is $O(N)$. Since our implementation iterates a fixed number of times, the number of communications is constant. For large problem sizes, **`jacobi`** exhibits relatively good performance on a workstation cluster.

4.1.2 Numerical Integration

Numerical integration approximates the area under the curve $f(x)$ within a certain interval. To approximate $\int_a^b f(x)\,dx$, we divide the interval $[a, b]$ into N subintervals. Let x_i denote the end of the ith interval, for $1 \le i \le N$, and let x_0 denote the beginning of the first interval. According to Simpson's Rule [63]:

$$\int_a^b f(x)\,dx \approx \frac{b-a}{6N}\left[f(x_0) + f(x_N) + \sum_{i=1}^{N-1}\left(2f(x_i) + 4f\left(\frac{x_i + x_{i+1}}{2}\right)\right)\right]$$

Our **`int`** program implements parallel numerical integration by assigning $\frac{N}{P}$ subintervals to each processor. Each processor simply applies Simpson's Rule to its set of subintervals. For the experiments presented here, we use numerical integration to approximate the value of π:

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

Since the only communications in the algorithm are to distribute the subintervals and collect the sums from each processor after the computation is complete, this algorithm can be classified as "embarrassingly parallel" [29]. However, because the algorithm's computational complexity is only $O(N)$ (with a relatively small constant of proportionality, depending on the integrand), the time required to distribute the subintervals and collect the results is often larger than the time required to solve the problem on a single processor.

4.1.3 Matrix Multiplication

Our implementation of parallel matrix multiplication, called **`matmult`**, computes $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, where $\mathbf{A}$ and $\mathbf{B}$ are $N \times N$ matrices. To parallelize the computation, we distribute $\frac{N}{P}$ contiguous rows of $\mathbf{A}$ and $\frac{N}{P}$ contiguous columns of $\mathbf{B}$ to each processor. Each processor computes the appropriate subblock of $\mathbf{C}$ and then passes its columns of $\mathbf{B}$ to the neighboring processor. After P iterations, the computation is complete, with the resulting $\mathbf{C}$ matrix distributed across the processors.

The matrix multiplication problem is a good candidate for parallelization on a workstation cluster because, for sufficiently large problem sizes, it involves a large amount of computation and relatively few communications. The computational complexity of **`matmult`** is $O(N^3)$, the total number of bytes transmitted is $O(N^2)$ and the number of communications is $O(P)$.

4.1.4 Traveling Salesman Problem

The traveling salesperson problem is to find, given a weighted graph, a minimum-weight cycle containing every vertex exactly once [23]. Because this problem is NP-complete, heuristics are often used to obtain an approximation to the optimal solution. Our implementation, which uses the *farthest insertion* heuristic, proceeds as follows.

We are given a set of N cities $\{C_0, C_1, \ldots, C_{N-1}\}$ and a symmetric matrix containing the distance between every pair of cities. Initially no cities are marked. We arbitrarily select a city (say C_0) and build a tour (i.e., a cycle) from the city to itself. The algorithm then iterates $N-1$ times, with the iterations labeled 1 through $N-1$. At the start of iteration i there are i marked cities (i.e., i cities in the partially completed tour) and $N-i$ unmarked cities (i.e., cities not yet in the tour).

Every city C_j keeps track of its distance from the partially completed tour, which is the distance from C_j to the city in the tour closest to C_j. During each iteration we insert the unmarked city C_k that is farthest from the tour. To add C_k to the tour, we replace the edge between two cities C_u and C_v in the tour with an edge from C_u to C_k and an edge from C_k to C_v. We choose the insertion point such that the increase in the length of the tour is minimized. After $N-1$ iterations, all the cities are in the tour.

Our parallel implementation of the algorithm, called **`tsp`**, uses one master process and P slave processes. We assign a group of $\frac{N}{P}$ cities to each slave process. Initially, the master process spawns the slave processes and distributes the cities. Each slave process chooses one of its cities as a candidate to be inserted into the tour, and communicates its choice to the master process. The master process chooses one of the P candidate cities and communicates the choice back to the slave processes. A similar process is followed for determining the insertion point for the city. All the communications in the algorithm involve small, fixed-size messages.

4.1.5 Parallel Sorting

Our **sort** program implements a bucket-based sorting algorithm similar to the one described in [22]. The algorithm assumes that N keys are to be sorted by P processors $P_1, \ldots, P_P$, where $N >> P$, and that the keys are initially distributed among the processors. Furthermore, we assume that the values of the keys are uniformly distributed across some known interval.

In the first phase of the algorithm, each processor P_i assigns its keys to buckets based on $P-1$ splitter values $S_1, \ldots, S_{P-1}$. The splitter values are chosen such that no element in bucket i is larger than any element in bucket $i+1$, for $1 \le i \le P-1$. Since the range of keys is known beforehand, splitter values can be determined without examining the data. Alternatively, appropriate splitter values can be chosen by selecting a sample of the keys, sorting them sequentially, and choosing $P-1$ values equally spaced apart in the sorted list [39].

In the second phase of the algorithm, the processors exchange buckets such that processor P_1 has all the buckets containing values less than S_1, processor P_2 has all the buckets containing values between S_1 and S_2, and so on. In the final phase of the algorithm, each processor performs a sequential sort on its set of buckets. The final sorted list is produced by concatenating the results from all the processors in order.

The first phase of **sort**, which divides the keys into buckets, is $O(\frac{N}{P})$. The last phase of the program, the sequential quicksort, is $O(\frac{N}{P} \log \frac{N}{P})$. Thus, the complexity of the computation in **sort** is $O(\frac{N}{P} \log \frac{N}{P})$. The communication, which takes place during the second phase of the program, consists of $P-1$ communications, each involving $O(\frac{N}{P})$ bytes.

4.2 Performance Prediction

In this section we evaluate the accuracy of predictions made by the APACHE system for the programs described above. In addition, we illustrate some of the prediction activities that can be done with the APACHE system and present the results of our experiments regarding the uniform cost assumption.

4.2.1 Predicting Execution Time

The APACHE system represents program execution time as a function of the problem size, N, and the number of processors, P. However, program execution time cannot be predicted exactly due to factors such as other processes executing in the background and system interrupts. This is particularly true of multiuser, multitasking systems such as Unix workstations. In addition, errors arise in the equation for predicted execution time when the assumptions of the underlying models are violated.

In this section, we compare actual execution times of the programs described in Section 4.1 with times predicted by the APACHE system. We obtained actual execution times for each program using a range of problem sizes and numbers of processors. In cases where the error in predicted times is large, we attempt to identify the sources of the prediction errors. The times shown in all the graphs in this section are in seconds.

4.2.1.1 Jacobi Algorithm

Figure 4.1 shows the predicted execution time and the prediction error for the `jacobi` program. The right graph in the figure shows the percent prediction error for an increasing percentage of the experimental runs. For example, according to the graph, half of the times observed in actual runs are within about three percent of the predicted times, and all of the observed times

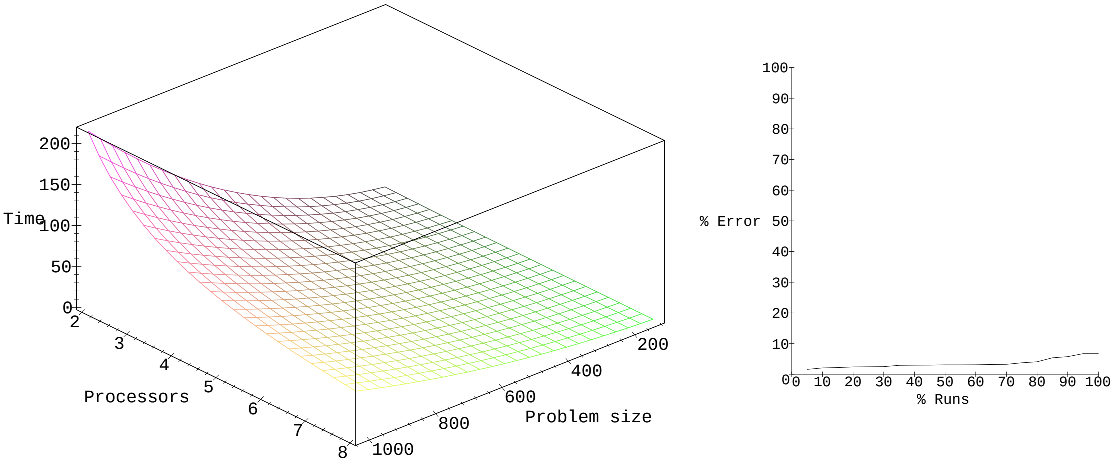


Figure 4.1. Predicted execution time and percent error of actual times for **`jacobi`**.

are within approximately seven percent of the predicted times. The predictions for **`jacobi`** are the most accurate of the five programs we tested. The APACHE communication model accurately reflects the communications in the **`jacobi`** program. Also, there seems to be a straightforward relationship between the C source code and the compiled object code in terms of the computations that are performed.

4.2.1.2 Numerical Integration

Figure 4.2 shows the predicted execution time and the percent error of actual times for the **`int`** program. For **`int`**, as with the other programs, we ignore the cost of spawning processes and distributing the initial data among the processors. All that remains to be done in the **`int`** program is to perform the computation and send a small, fixed-size message to the master process. Because there is only a single, small communication, most of the measured execution time is spent in computation. Thus, the **`int`** program provides a test case for the accuracy of the computation model.

The graph of percent error of actual times in Figure 4.2 shows that all the observed times exhibit nearly the same amount of error, independent of the problem size (our predictions

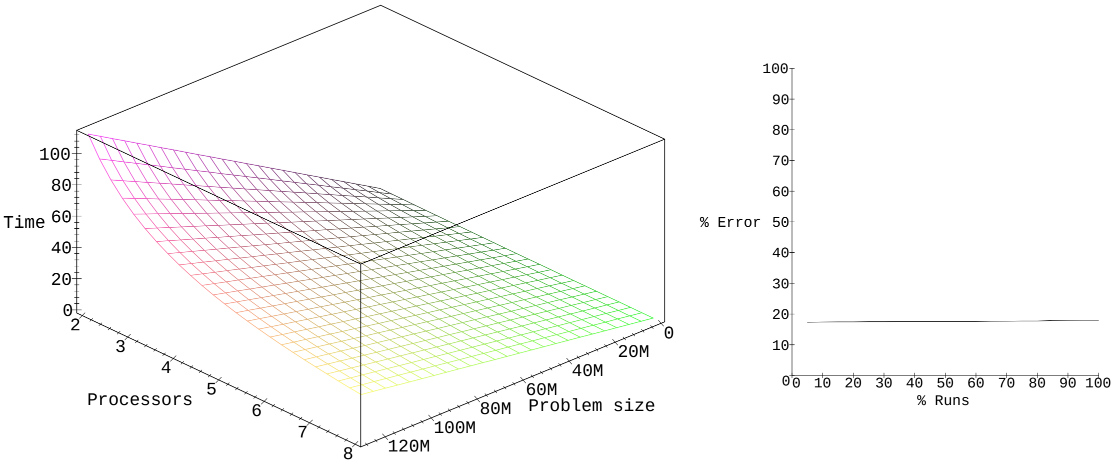


Figure 4.2. Predicted execution time and percent error of actual times for **`int`**.

consistently underestimated execution time). The primary reason for the prediction error is that the **`int`** program uses the **`double`** data type, while the values we used for the cost factors in the computation model were obtained with benchmark programs using **`float`**s. The error curve is nearly flat because the actual computation time is a very regular function of the problem size. Since the total execution time of **`int`** is dominated by computation, the prediction error for the total execution time was consistent for different problem sizes.

4.2.1.3 Matrix Multiplication

The predicted times for **`matmult`** are relatively good, with about 90 percent of the runs having less than 15 percent error (see Figure 4.3). Since the largest prediction errors for this program result from overestimating execution time for runs with small grain size, the prediction error may be due largely to cache effects.

Because of the structure of the program, **`matmult`** is a good candidate for further analysis in order to isolate the prediction error. The program iterates only P times, and during each iteration a large amount of computation is done, followed by a single, relatively large communication. Thus, calls to timing routines can be inserted to measure computation

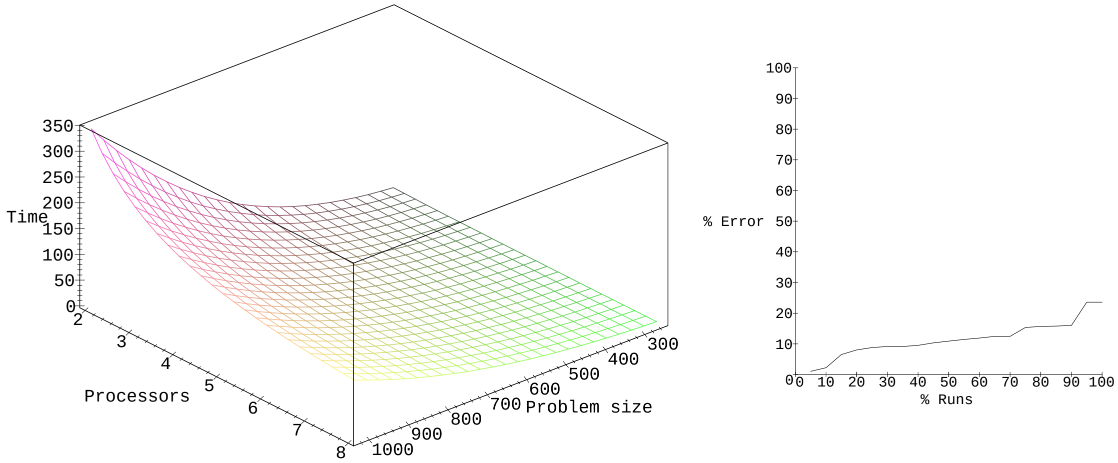


Figure 4.3. Predicted execution time and percent error of actual times for **`matmult`**.

and communication time separately with minimal perturbation to program performance. Performing this type of analysis confirms that the predicted communication time is quite accurate, while the computation time is overestimated for small grain sizes. Thus, we conclude that cache effects are the primary source of prediction error for **`matmult`**.

Although the APACHE computation model assumes a fixed cost for each type of operation, in practice the time required to perform these operations varies with interactions between the processor cache, register allocation, and pipeline status. This weakness of the APACHE computation model suggests one area for further research (see Section 5.2).

The version of **`matmult`** used in these experiments uses dynamically allocated two-dimensional arrays. We developed another version of the program that is identical to the original except that it uses statically allocated arrays. The predicted execution time for both programs is the same, since they both use the same code to perform the actual matrix multiplication. However, the static array version of **`matmult`** is significantly slower than the dynamic array version and, therefore, the prediction error is much higher (see Figure 4.4). Analysis reveals that the computation time for the static array program is consistently

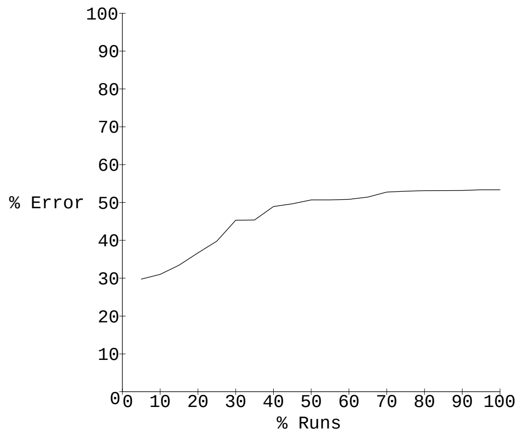


Figure 4.4. Prediction error of the static array version of **`matmult`**.

underestimated by about 50%. Turning on the compiler optimizer decreases the execution time of both programs significantly, but the dynamic array version of **`matmult`** still runs significantly faster.

Since the only difference between the programs is the declaration of the arrays, it seems intuitive that the performance disparity is due to differences in the cost of array accesses. However, computing the address of an element of a static two-dimensional array should involve two additions and two multiplications (see [1]). The arrays in the dynamic version of **`matmult`** are composed of N blocks of memory, each containing N values. The first index of the array selects a block of memory (with a multiplication and an addition), and the second index selects an element within that block (also with a multiplication and an addition). Thus, array accesses in both programs should involve the same cost—two multiplications and two additions. Examining the assembly output of the compiler confirms that there are substantial differences in the final code generated for the two programs. We conclude that, at least for the compiler we used, slight differences in data structures can significantly affect the form and performance of the executable program produced by the compiler.

A more general conclusion to be drawn from our experiments with the **`matmult`** program is that compiler output is a significant factor in the accuracy of performance predictions. Another opportunity for future work is to integrate the performance prediction system with a compiler, enabling it to account for the operations performed during code generation and optimization (see Section 5.2).

4.2.1.4 Traveling Salesman

The **tsp** program exhibits high prediction error, as shown in Figure 4.5. One reason for the large error is that **tsp** violates the assumptions of the APACHE communication model. The model assumes that communications involve all P processors, and that the processors communicate in a way such that the load due to communication is distributed equally across the processors. However, the communications in the **tsp** program consist of *broadcasts* and *reductions*. During each iteration, the P slave processes choose a city to insert into the tour, and send their candidate city to the master process (a reduction). The master process chooses the city to be inserted and broadcasts its choice to the slave processes. A similar process is followed to determine the insertion point for the city.

According to the equation generated by APACHE, the **tsp** program performs $2(N-1)$ communications for a problem size of N, which is correct according to the source code. However, all the messages sent by the slave processes are directed to the single master process, which must receive and process the messages sequentially. Thus, communication time is increased by about a factor of P. The increased communication time can be modeled by changing the equation for the number of communications to $P \times 2(N-1)$. The modified equation results in the predicted time and error profile shown in Figure 4.6. Although the prediction error is smaller than before, it is still significant. Further analysis reveals that over 90 percent of the execution time of **tsp** is spent in communication. Thus, we attribute the remaining prediction error to the imprecise modeling of the communication behavior.

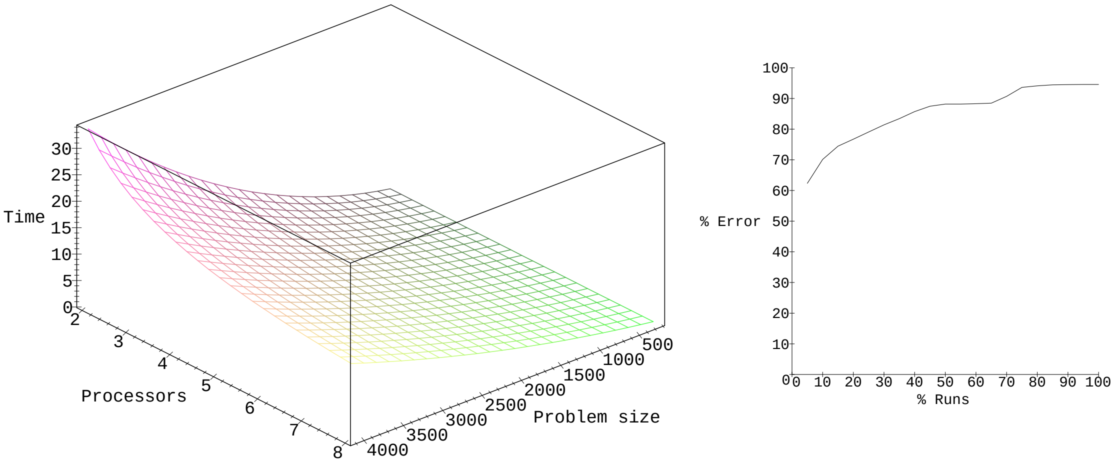


Figure 4.5. Predicted execution time and percent error of actual times for **`tsp`**.

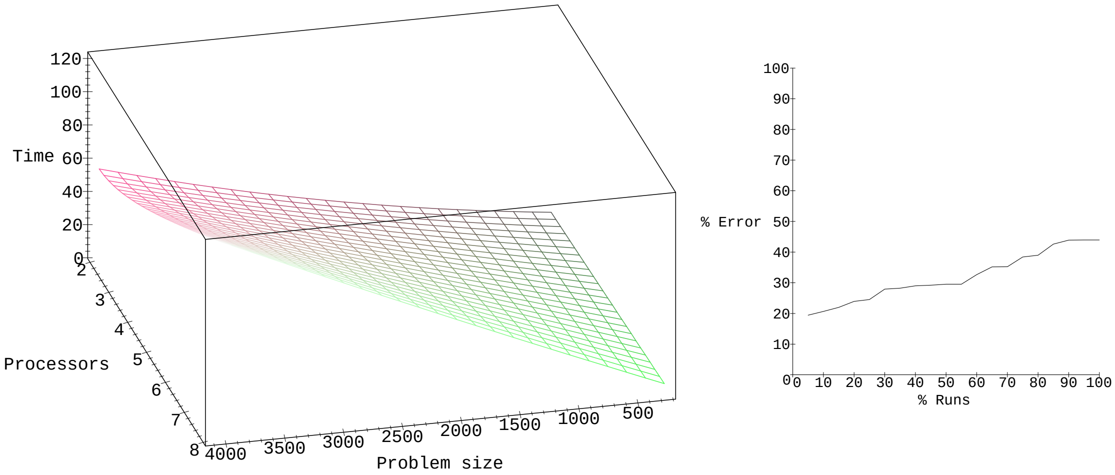


Figure 4.6. Predicted execution time and percent error of actual times for **`tsp`** after modifiying the equation for the number of communications.

4.2.1.5 Parallel Sorting

The APACHE system produced fair predictions of the performance of the **`sort`** program (see Figure 4.7). We found that the predicted communication time was quite accurate, while the predicted computation time was usually underestimated by about 15 to 20 percent. According to the performance model generated by the APACHE system, the predicted computation time

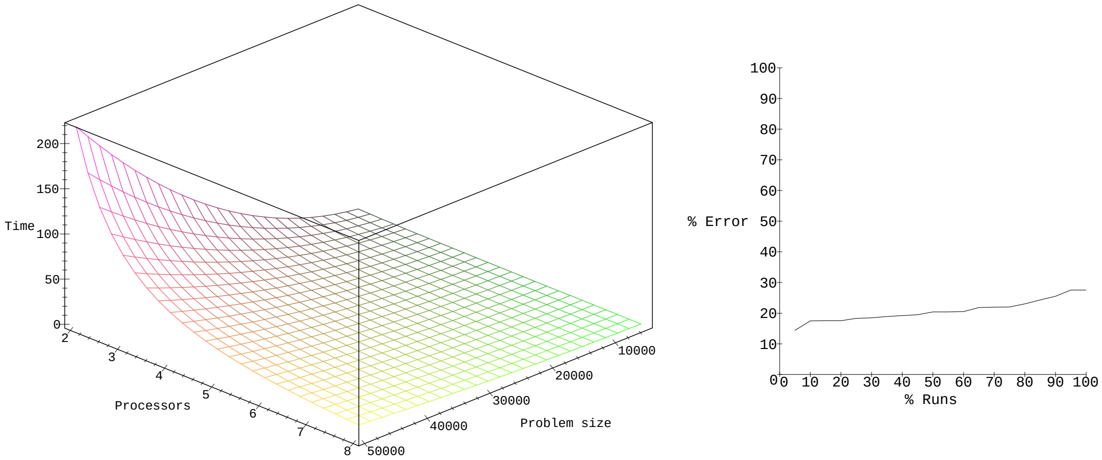


Figure 4.7. Predicted execution time and percent error of actual times for **`sort`**.

for the program is divided among relational operations (i.e., comparisons), array dereferences, and increments. Since the predicted number of occurrences for all three operations are very similar, it is difficult to pinpoint the primary source of prediction error. In other words, by slightly increasing the cost factor of *any* of the three operations that dominate computation time in the performance model, the percent error in the performance model can be reduced to less than nine percent for all the observed times. We can be confident that the prediction error is in the computation model, but we cannot easily identify a more specific source for the error.

4.2.1.6 Summary

The five programs we tested exhibit a variety of behaviors. For example, the execution time of **`matmult`** is divided somewhat evenly between computation and communication. In contrast, **`jacobi`** and **`int`** spend most of their time in computation, and the execution time of **`tsp`** is dominated by communication. The **`sort`** program spends most of its time comparing values and moving data. The **`int`** and **`tsp`** send small fixed-size messages, while **`matmult`** communicates $O(N^3)$ bytes.

In most cases, the APACHE system was able to make reasonably accurate predictions of program performance. Cases where predictions were less accurate indicate directions for future work in this area. Two factors that appear to affect the accuracy of the predictions made by the APACHE system are compiler output and CPU architectural effects. We believe that these factors can be compensated for by improvements or extensions to the system (see Section 5.2).

4.2.2 Other Performance Prediction Activities

The equations generated by the APACHE system can be manipulated in a variety of ways to examine different aspects of program performance. In this section we present some sample examples of the types of predictions that can be done with these equations.

Figure 4.8 shows the predicted percentage of time spent in communication and computation in the **`matmult`** program as the problem size and the number of processors vary. This type of analysis is useful for determining an appropriate combination of problem size and number of processors to avoid communication bottlenecks.

With the equations generated by the APACHE system, we can predict the impact of running programs on different system configurations. For example, by eliminating the contention factor from the equation for Ethernet communication time, we obtain a performance model for switched Ethernet. Figure 4.9 shows the predicted savings in execution time (in seconds) for the **`matmult`** program of switched Ethernet over traditional Ethernet. For example, with eight processors and a problem size of 1000, we predict that **`matmult`** will run 25 seconds faster over switched Ethernet. As the figure depicts, the greatest predicted time savings are with large numbers of processors and large problem sizes, where contention significantly affects network performance. For small problem sizes and small numbers of processors, switched Ethernet does not provide a significant performance advantage for

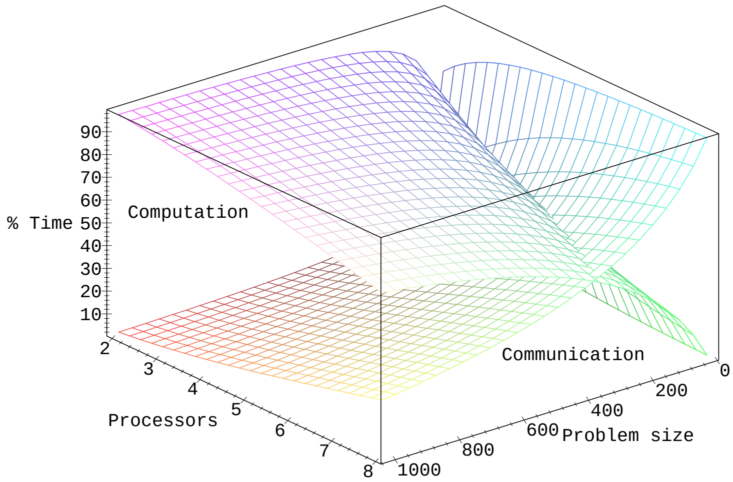


Figure 4.8. Predicted percent time spent in communication and computation for **`matmult`**.

`matmult`.

Another use for the equations generated by the APACHE system is to compare the predicted performance of different algorithms. For example, in our implementation of the Jacobi algorithm, we distributed the data in contiguous rows. An alternative implementation is to distributed blocks of data. According to [50], which of these algorithms performs better depends on the problem size and the number of processors. Many problems can be solved with a number of different parallel algorithms. The APACHE system can be used to predict which of several algorithms will perform better on a given hardware configuration and with a given problem size.

4.2.3 Uniform Cost Assumption

Performance prediction systems are often based on the *uniform cost assumption*, which is that all floating point operations have the same cost (measured in execution time). Our experiments with simple benchmark programs indicate that this assumption is not valid for the hardware platforms we used (see Section 2.1 and Table A.2 in Appendix A).

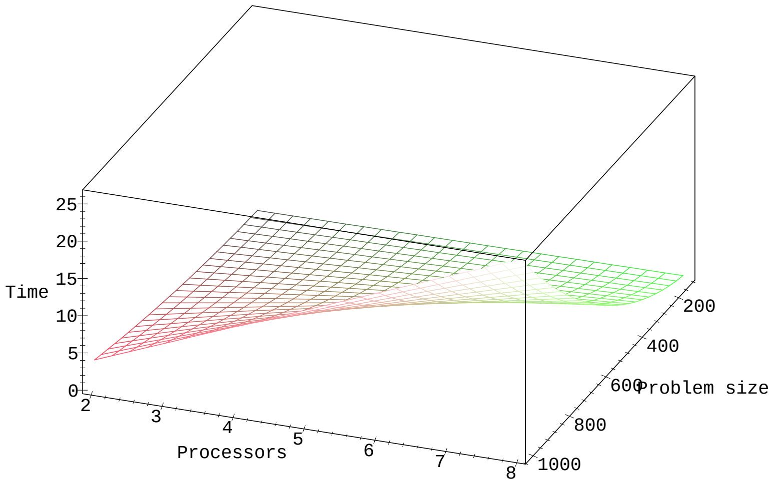


Figure 4.9. Predicted savings in execution time (in seconds) of **`matmult`** on switched Ethernet over traditional Ethernet.

We tested the uniform cost assumption by setting all the cost factors in the computation model equal to the same value and comparing actual execution times with predicted times. We attempted to find the value for the cost factors that produced the most accurate predictions (eventually settling on 0.08 μsec).

We found that the impact of the uniform cost assumption on the accuracy of performance predictions was different for each of the five applications we tested. The average prediction error of the **`jacobi`** and **`tsp`** increased by less than one percent. According to the performance model generated by the APACHE system, most of the computation time in **`jacobi`** is spent doing additions and dereferences, two operations whose original cost is similar to the cost we used under the uniform cost assumption (see Table A.2 in Appendix A). Thus, the predicted computation time for **`jacobi`** was minimally affected. The impact on the prediction error for **`tsp`** was insignificant simply because computation time is a very small fraction of the program's total execution time.

The average prediction error for the **`matmult`** program increased by about two percent under the uniform cost assumption. The performance model for the program predicts that most of the computation time is spent in array dereferencing. Since the original cost of array dereferencing is similar to the cost used under the uniform cost assumption, there was only a small increase in the average prediction error.

The average prediction error for the **`int`** program increased by about four percent. However, examining the predicted distribution of the computation time among the different operations reveals that the impact could have been much greater. The original performance model predicts that 22% of the computation time is occupied by addition, 20% by multiplication, and 48% by division operations. Under the uniform cost assumption, the predicted distribution of computation times becomes 43% for addition, 29% for multiplication, and 19% for division. Although the average prediction error increases only slightly, the prediction models are quite different. Since the computations performed by the **`int`** program are determined largely by the function being approximated, it would not be difficult to find a function for which the uniform cost assumption would result in very high prediction errors. We conclude that the small observed impact on the prediction error for the **`int`** program is an artifact of the function we chose to approximate (see Section 4.1.2).

We found that the average prediction error for the **`sort`** program increased by about six percent under the uniform cost assumption. The computation time in **`sort`** is dominated by relational, dereference, and increment operations, all of which have costs similar to the one used under the uniform cost assumption. The cost of these operations was slightly lower under the uniform cost assumption, resulting in smaller predicted execution times. Since the original performance model for **`sort`** already underestimated the actual execution time, the lower predicted execution time resulted in higher average prediction error.

In summary, the impact of the uniform cost assumption on the accuracy of performance predictions varies according to the application. Generally, this impact depends on the particular operations that the program performs and on the actual costs of those operations on the hardware platform in question. Naturally, predictions for programs that spend most of their execution time in communication will be minimally affected. We conclude from our experiments that the impact of the uniform cost assumption on prediction accuracy can be significant. Further research could be done to create profiles of the operations performed in typical parallel applications in order to better understand when the impact of the uniform cost assumption is significant.

4.3 Network Media Selection

Because the APACHE system represents program performance with equations, the effects of changes in system parameters can be predicted by changing the factors in these equations. We use this capability to predict the performance of PVM applications on workstation clusters connected with Asynchronous Transfer Mode (ATM) [56]. A similar process could be used to evaluate other network technologies such as FDDI or switched Ethernet.

4.3.1 Performance Comparison

The primary advantages of ATM over Ethernet as a communication medium for parallel processing are its high bandwidth and switched topology. However, according to our experiments, the latency of PVM communications is slightly higher over ATM than over Ethernet. Other researchers have reported similar findings [14]. Thus, the performance advantage of ATM is most apparent for communications involving large messages and large numbers of processors, when the performance of Ethernet begins to suffer because of bandwidth restrictions and packet collisions. This effect is depicted in Figure 4.10. The solid lines in the figure show the percent improvement of ATM over Ethernet in the time required for a single communication involving varying message sizes and numbers of processors. We define "percent improvement" as

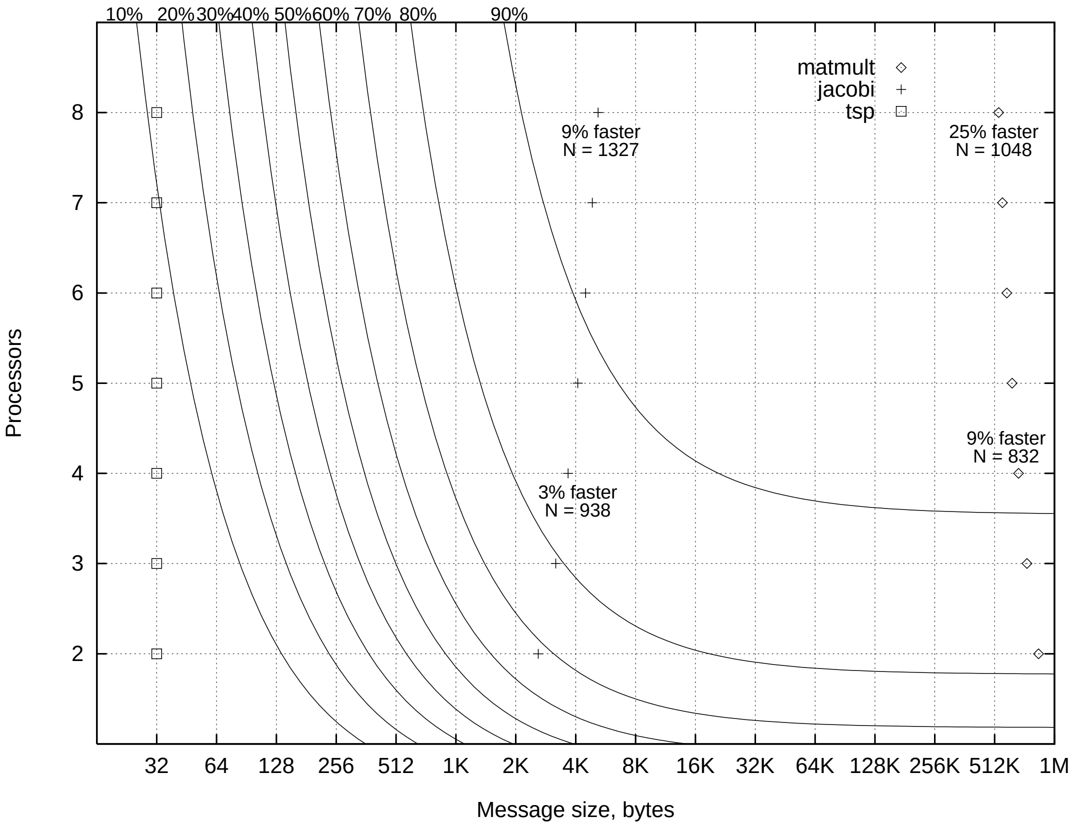


Figure 4.10. Predicted improvement in communication time of ATM over Ethernet. Each solid line represents a 10% improvement in communication time. Individual points show the predicted improvement in communication time for three PVM applications. Some points are labeled indicating the improvement in total execution time and the problem size.

$$\frac{T_{Ether} - T_{ATM}}{T_{Ether}} \times 100$$

where T_{ATM} and T_{Ether} are derived from Equations (2) and (3), respectively.

As Figure 4.10 depicts, the performance of ATM is comparable to that of Ethernet for small messages and small numbers of processors. However, as the number of processors increases, the performance advantage of ATM becomes significant even for small messages. For large messages, where bandwidth is the dominant factor in communication time, ATM provides much better performance than Ethernet.

The three sets of points in Figure 4.10 show the predicted communication behavior of different PVM applications: **`matmult`**, **`jacobi`**, and **`tsp`** (see Section 4.1). To obtain these points, we use the APACHE system to construct models of the applications as described in Section 2.3. We then choose a fixed computation time T_{comp} and use the Maple system to solve Equation (1) for N, resulting in an expression for scaleup as a function of P. (We define *scaleup* as a function returning the problem size required to maintain a fixed computation time for different numbers of processors. For the points shown in Figure 4.10, we set $T_{comp} = 10$ seconds.) With the expression for scaleup we compute the problem size N necessary to keep the grain size constant for a given number of processors. We use this value of N to determine the average message size from Equation (2). Thus each set of points represents the average improvement in communication time for an application with a fixed grain size and different numbers of processors. Since communication time comprises a different fraction of the total execution time for each application, we have indicated for some of the points the problem size and the predicted improvement in *total* execution time. For example, we predict that **`matmult`** will run 25% faster over ATM than over Ethernet with a problem size of 1048 on 8 processors.

For all three PVM applications depicted in Figure 4.10, we must use larger problem sizes to maintain a constant grain size as the number of processors increases. It is interesting to note that, with a fixed grainsize, the average message size for the **`jacobi`** program increases with the number of processors, while the average message size for **`matmult`** decreases as the number of processors increases. Since the number of bytes sent by each processor in **`jacobi`** is $O(N)$ (independent of the number of processors), the average message size increases for a fixed grain size on an increasing number of processors. For **`matmult`**, the number of bytes transmitted by each processor is $O(\frac{N^2}{P})$. Since the computational complexity of the program is $O(N^3)$, a small increase in the problem size is sufficient to maintain a fixed grain size for additional processors. The small increase in the problem size results in a small increase in the *total* number of bytes transmitted, but the communications are distributed among more processors, resulting in smaller messages for each processor.

As shown in Figure 4.10, the **matmult** program, which communicates large amounts of data, exhibits the greatest predicted performance improvement with ATM. Although the communications in **jacobi** are relatively small, ATM results in a 60% to 90% predicted improvement in communication time. However, communication time constitutes a small fraction of the total execution time for **jacobi**, so the predicted improvement in total execution time is rather small. The **tsp** program communicates only small, fixed-size messages, so ATM provides little or no improvement in communication time and the total execution time is minimally affected.

The impact of ATM on program execution time can be examined in a more application-independent way. If we assume that our program iterates N_{iter} times and spends T_{comp} time in computation and T_{comm} time in communication during each iteration, we have the following equation for predicted execution time:

$$T_{total} = N_{iter}(T_{comp} + T_{comm}) \tag{5}$$

This model is appropriate for many applications that iterate a fixed number of times or until a steady state is achieved.

From Equation (5), execution time for ATM or Ethernet clusters can be predicted by replacing T_{comm} with an expression derived from Equation (2) or (3). To predict the improvement in execution time of ATM over Ethernet, we have

$$Improvement = \frac{T_{Ether} - T_{ATM}}{T_{Ether}} = \frac{N_{iter}(T_{comp} + T_e) - N_{iter}(T_{comp} + T_a)}{N_{iter}(T_{comp} + T_e)} = \frac{T_e - T_a}{T_{comp} + T_e} \tag{6}$$

where T_a and T_e are the communication times for ATM and Ethernet for each iteration. For a given T_{comp}, we can use this equation to compute the improvement in total execution time as a function of message size and the number of processors. Figure 4.11 shows surfaces

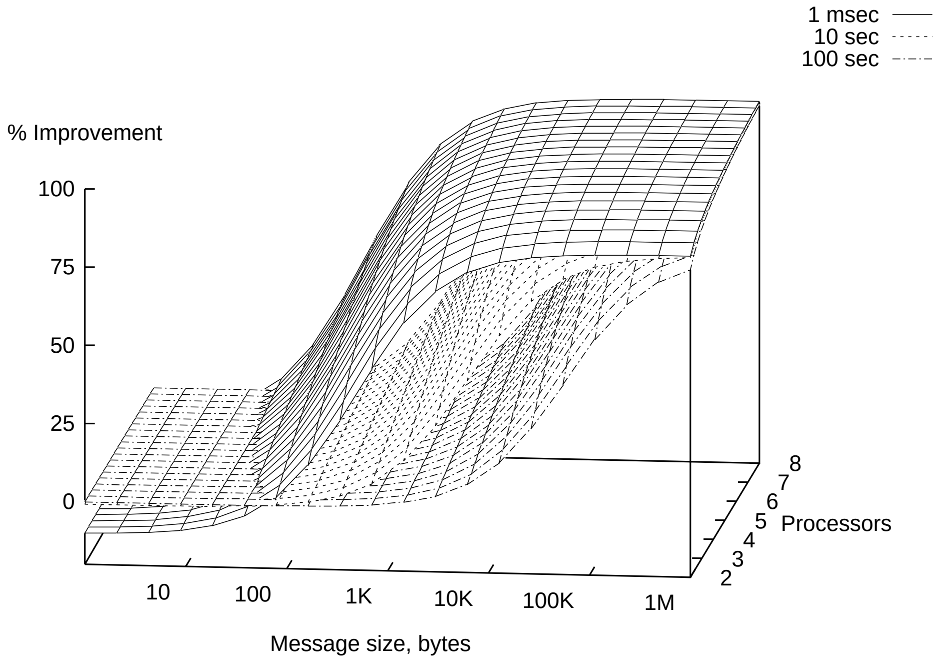


Figure 4.11. Predicted effect of ATM on program execution time as message size and the number of processors vary (see Equation (6)). Results for three values of T_{comp} are shown.

generated by Equation (6) for three values of T_{comp}. For $T_{comp} = 1$ msec, ATM results significant performance gains for messages larger than about 1Kbyte. For small messages, ATM actually hinders performance since PVM communications exhibit slightly higher message latency over ATM than over Ethernet. Very large messages are required for performance gains for T_{comp} = 10 and 100 seconds, presumably since communication time comprises a smaller fraction of the total execution time.

Figure 4.12 shows the surface for T_{comp} = 1 msec, with contour lines indicating improvement in execution time in 10% increments. The contour lines help clarify the interaction between message size and the number of processors.

4.3.2 Cost–Benefit Analysis

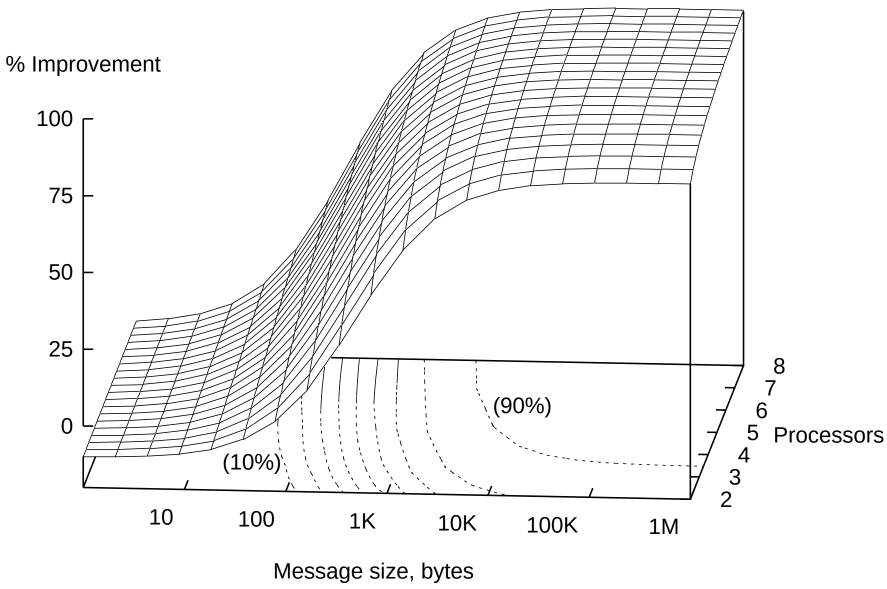


Figure 4.12. Predicted effect of ATM on program execution time as message size and the number of processors vary, $T_{comp} = 1\,\text{msec}$ (see Equation (6)). Contour lines mark improvement in 10% increments.

If we have a parallel application (or set of applications) that runs on P workstations connected with Ethernet and we know the potential performance gains of ATM for our system, we can perform a cost–benefit analysis to determine whether an investment in ATM equipment for our application is justified.

For purposes of the analysis, we assume that a workstation has cost C_w and is employed for a period of time T_w. If we amortize the cost of P workstations over their period of use, the value of CPU time on P workstations is $\frac{P\,C_w}{T_w}$. If our application currently takes T_{app} time to execute on P workstations, and it will be run N_{app} times during the workstations' period of use, the cumulative cost of executing the application during the lifetime of the workstations is

$$T_{app}\,N_{app} \times \frac{P\,C_w}{T_w}$$

If our application is predicted to execute $(100 \times I_{atm})$ percent faster over ATM, we can express the value of the performance improvement as

$$I_{atm}\ T_{app}\ N_{app} \times \frac{P\ C_w}{T_w}$$

Assuming that an ATM switch has cost C_{sw} and an ATM network interface card has cost C_{nic}, then we may say that the investment in ATM is justified for our application if

$$C_{sw} + P\ C_{nic} \ \le\ I_{atm}\ T_{app}\ N_{app} \times \frac{P\ C_w}{T_w} \tag{7}$$

For example, given the following values

$C_w = \$15,000$	$T_w = 2$ years $\approx 63.12 \times 10^6$ sec	$P = 8$
$T_{app} = 1$ hour $= 3600$ sec	$N_{app} = 8$ times/day $\times 2$ years ≈ 7300 times	$I_{atm} = 0.4$
$C_{sw} = \$25,000$	$C_{nic} = \$1500$	

Equation (7) becomes

$$25000 + 8(1500) \le 0.4(3600)(7300) \times \frac{8\,(15000)}{63.12 \times 10^6} \ \Rightarrow\ 37000 \le 19985$$

indicating that the cost of ATM would not be completely offset by the potential performance gain. We may wish to solve Equation (7) for I_{atm} to determine how much improvement we need to "break even" within time T_w:

$$I_{atm} \ \ge\ \frac{T_w(C_{sw} + P\ C_{nic})}{T_{app}\ N_{app}\ P\ C_w}$$

For our example, we need

$$I_{atm} \geq \frac{63.12 \times 10^{6}(25000 + 8(1500))}{3600(7300)(8)(15000)} = 0.74$$

The performance benefits of ATM can be leveraged by running an application with a larger problem size instead of running it for a shorter time. If we have T_{app} and I_{atm} as functions involving N (which can be obtained using the APACHE system), we could solve Equation (7) for N to determine the largest size problem that could be solved in a fixed amount of time. The benefit of ATM may then be measured as an increase in accuracy or productivity, depending on the nature of the problem we are solving.

Chapter 5 Conclusions and Future Work

This chapter presents some conclusions from the research described in previous chapters, and suggests areas for future work. Section 5.1 summarizes the contributions of the current research and the issues that arise in performance prediction of PVM programs. Section 5.2 suggests some areas for further work.

5.1 Conclusions

5.1.1 Summary of Research

The objective of the current research was twofold: (1) to show that a performance prediction system can be developed for PVM programs running on workstation clusters; and (2) to show that information provided by the performance prediction system can be used as the basis for selecting the most appropriate network medium for a particular parallel application (or set of applications). In Chapters 3 and 4 we described the development of the APACHE performance prediction system and the experimental results obtained with the system. In Section 4.3, we described a methodology for network media selection for PVM clusters.

5.1.2 Contributions

The current research has shown the feasibility of developing a performance prediction system for PVM programs running on workstation clusters. To our knowledge, the APACHE performance prediction system is the only tool of its kind. Although there are several performance tools available for PVM programs, none are capable of performance prediction. In addition, our system improves upon the work of Clement [16] by incorporating a high degree of automation and a performance model which allowed us to test the validity of the uniform cost assumption (see below).

The current research also shows how information obtained with a performance prediction system can be used as the basis for network media selection for PVM clusters (see Section 4.3). Since this performance information can be determined in a number of other ways, the methodology we present for network media selection constitutes an independent contribution.

An important conclusion of the current work is that a performance prediction system needs to be highly integrated with the compiler in order to accurately predict computation time. An examination of compiler-generated assembly code confirms that the form of an executable program often bears little resemblance to the original source code, particularly if compiler optimizations are enabled. Also, slight changes to data structures can have a significant impact on the form and performance of the executable program generated by the compiler (see Section 4.2.1).

Another result of this research concerns the uniform cost assumption. We found that this common simplifying assumption can significantly decrease the accuracy of performance predictions (see Section 4.2.3). Timing results from our simple benchmark programs indicate that the uniform cost assumption is not valid for the hardware platforms we used (see Section 2.1 and Table A.2 in Appendix A). We concluded from our experiments with different PVM applications that the impact of this assumption on prediction accuracy varied with the application and the hardware platform.

5.1.3 Factors Affecting Prediction Accuracy

There were two major factors affecting the accuracy of the predictions that we obtained with the APACHE system. The first of these is compiler output. The APACHE system generates performance models based on C source code. If there is not a close correspondence between the source code and the machine instructions emitted by the compiler, the computation model will be inaccurate. Thus, although the APACHE system is designed to be portable, in reality the accuracy of its predictions is highly dependent on the compiler, and it may be that no compiler adequately meets the requirements of our method. Code optimizations commonly performed by compilers also significantly affect the form of the final executable code, particularly within loops. We disabled code optimizations for the experiments presented in Chapter 4.

A related issue affecting the accuracy of our predictions is the interaction between the executable code and CPU architecture. The APACHE computation model assigns a fixed cost to each operation. In reality, however, the time required to perform an operation is affected by interactions between the processor cache, register allocation, and pipeline status. It may be possible to account for cache effects by determining the cache miss rate for an application and adjusting the cost of memory access accordingly. However, it seems difficult to account for the other factors except by running the instrumented program on a simulator.

The second factor affecting the accuracy of our predictions was the simplifying assumption we made in the development of the communication model (see Section 2.2). We assume that communications involve all P processors simultaneously, and that the processors communicate in such a way that the load due to communication is balanced across all the processors. Some of the programs that we used to test the performance prediction system exhibited this type of communication, but others did not.

An additional factor that we did not deal with in the current research is algorithmic effects.

Some algorithms exhibit unpredictable behavior depending on the input data. For example, the Jacobi algorithm iterates a different number of times depending on the size, shape, and initial state of the grid. For our **jacobi** program, we compensated for this by modifying the algorithm to iterate a fixed number of times. Other well-known algorithms, such as quicksort, also exhibit irregular behavior. In the most general case, such irregular behavior precludes the accurate prediction of program performance. However, with some assumptions about the input data or appropriate algorithmic parameters, it should be possible to predict the performance of this type of algorithm.

5.2 Future Work

We see several areas in which the current work could be extended or improved. The first involves compiler issues. As described in Section 3.3, the APACHE system constructs performance models based on program source code. If there is not a close correspondence between the operations expressed in the original source code and the executable code emitted by the compiler, the computation model will be inaccurate. Although we disabled the code optimizations for the experiments presented in Chapter 4, we believe that a low correlation between program source code and final executable code is responsible for much of inaccuracy in our computation model.

One approach to solving this problem is to integrate the performance prediction system with an existing compiler, such as GNU's **gcc**, so that the operations performed during code generation and optimization can be reflected in the computation model. This level of integration with the compiler would also provide a better context for experiments with the uniform cost assumption, and might facilitate the consideration of data access patterns and cache effects. Finally, integration with the compiler would also allow provide the system with information regarding data types that could be used to further refine the prediction model (see

Section 4.2).

Another area for future compiler-related work is support for interprocedural analysis. The current APACHE system works only for PVM programs that consist of a single function. A common simplifying assumption in performance prediction systems is that procedures are always called with the same parameters [24]. Under this assumption, the APACHE system could easily be extended to account for procedure calls. Other alternatives for interprocedural analysis include expanding functions “inline” with parameter substitution, and supporting a compiler pragma indicating the time complexity of the function in terms of its parameters. A different approach would be required in order to account for time spent in operating system routines. No external functions were invoked by the programs used for the experiments presented in Chapter 4, with the exception of the PVM communication routines.

A second area for future work involves the identification of communication patterns. Although some of the programs we used in our experiments perform communications among neighboring processors, many parallel programs exhibit very different communication patterns. Communication patterns could be identified during dynamic analysis by recording the sender and receiver of each message and maintaining some type of data structure representing interprocessor communication.

A third area for future work is to use the information provided by the APACHE performance prediction system as the basis for a performance debugging tool. The system could be extended to provide a performance model for each basic block of a PVM program. This per-block information can be used in a debugging tool such as the one depicted in Figure 5.1.

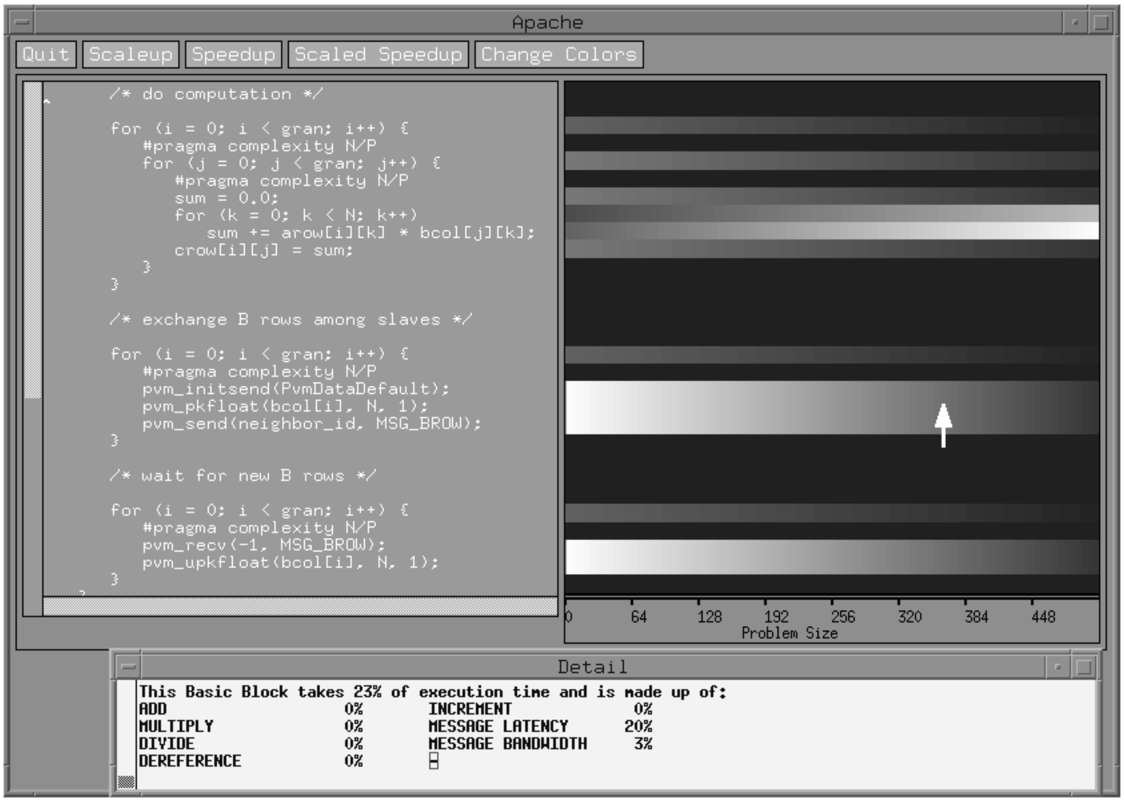


Figure 5.1. Interface to a performance debugging tool that could be developed with the APACHE system.

Appendix A APACHE System Documentation

This appendix contains the documentation for the APACHE system, which consists of the source-to-source compiler, covered in Section A.1, and the prediction system, covered in Section A.2.

A.1 Source-to-Source Compiler

This section documents the installation and use of the APACHE source-to-source compiler, including the generation of application performance models.

A.1.1 Installation

The APACHE source-to-source compiler consists of several files, all of which should be installed in the same directory. To install the compiler, one change needs to be made to the **`inst`** shell script. Near the beginning of the script, the **`INST_DIR`** variable should be set to the path where the compiler is installed. This same path should be included in the **`PATH`** environment variable of anyone who will run the compiler.

A Makefile is included with the compiler. Typing **`make`** in the directory where the compiler files are located should build everything. The GNU utilities **`flex`**, **`bison`**, and **`gcc`** must be available in one of the directories in the **`PATH`** environment variable for things to work properly.

A.1.2 PVM Application Requirements

Section 3.1 explains the limitations on the PVM applications that can be processed by the APACHE source-to-source compiler. We emphasize two related points. First, if the program conforms to the "master–slave" programming model, the slave program is the one that should be processed by the compiler. Second, remember that the program must consist of a single function (i.e., **`main()`**). Although the compiler will not complain if it encounters calls to other functions, the cost of executing the subroutines will not be reflected in the final equation for execution time.

Other program requirements concern the use of compiler pragmas (see Section A.1.6 for details on all the APACHE pragmas). First, there must be at least one matching pair of pragmas of the form **`#pragma instrument {on|off}`** to turn instrumentation on and off. One reason for the instrumentation pragmas is that it is common in performance evaluation to disregard the time spent spawning processes, distributing data, and so on before the "real" computation begins. The final equation for predicted execution time will reflect the part (or parts) of the program enclosed by instrumentation pragmas. In addition, the instrumentation pragmas are replaced by calls to timing routines in the "timed" version of the PVM application (see below). When the timed application is executed, it will produce a message indicating the execution time for each part of the program enclosed by instrumentation pragmas. Usually, one pair of instrumentation pragmas around a group of contiguous program statements is appropriate, but several pairs of these pragmas may be used.

Finally, the APACHE system requires that every **`while`** and **`do`** loop contain a pragma indicating the time complexity of the loop. The loop bounds of most **`for`** loops can be determined with static and dynamic analysis (see Sections 3.3.2 and 3.3.3). However, predicting the time complexity of **`while`** or **`do`** loops, which can have arbitrary termination conditions, reduces to the (provably impossible) halting problem [9].

A.1.3 Invocation

The source-to-source compiler is invoked by running **`inst`**. The usage is

```
inst [-p] file
```

where **`file`** is the name of the C source file that will be instrumented. The **`-p`** option runs the C preprocessor before the instrumentation process begins. This is useful for programs that use preprocessor macros. For example, consider this code fragment:

```
1.      #define F(x) (1.0 / (4.0 + (x)*(x)))
2.      ...
3.      for (i = 0; i < N; i++) {
4.         sum += F(i);
5.         ...
6.      }
```

If this code is instrumented without running the preprocessor (i.e., without the **`-p`** flag), the compiler will count a single addition on line 4. However, when the macro is expanded, line 4 actually contains two additions, a multiplication, and a division.

The **`-p`** option should not be used unless it is strictly necessary, since running the preprocessor often causes headaches related to files included with the **`#include`** directive (**`stdio.h`** seems to be the most troublesome). Commenting out the **`#include`** directive in the original source file is usually a sufficient workaround for such problems.

If all goes well, **`inst`** will display several messages indicating what it is doing. The compiler produces source files for instrumented and "timed" versions of the original program. The instrumented program contains the code that eventually produces the equations representing program performance (see Section 3.3). The timed version of the program is identical to the original except for the addition of two calls to timing routines. When the timed program is run, a message is sent to **`stdout`** indicating the total execution time. In addition to the two source files, the compiler produces a Makefile suitable for compiling the instrumented and timed programs, and then makes them. The source files for the instrumented and timed programs will be named **inst.prog.c** and **time.prog.c**, where **prog.c** is the name of the original source file. The executables will be named **inst.prog** and **time.prog**, and the Makefile will be named **Makefile.prog**.

A.1.4 Generating the Equations

The equations representing the performance of the PVM application are generated by running the instrumented program. The instrumented program must be run with two command line options:

```
inst.prog -N<probsize> -P<procs>
```

where **<probsize>** and **<procs>** represent the problem size and number of processors that will be used during the instrumentation run. These values are used to fit the block execution counts to functions of N and P (see Section 3.3.3). If the PVM application is written in SPMD fashion, these parameters can be passed to the instrumented program on the command line as shown. If the program is written in master–slave style, the master program will be executed from the command line, and the instrumented slave program will be spawned by the master program. In this case, two changes need to be made to the **pvm_spawn()** call in the master program. First, the master program must spawn the instrumented version of the slave program. Second, the **-N** and **-P** arguments must be passed to the instrumented slave program via the **argv** parameter. Any command line arguments expected by the original PVM program should appear *after* the **-N** and **-P** options. (In the master–slave model, this refers to command line arguments expected by the *slave* program, passed via the **argv** parameter to **pvm_spawn()**.)

If all goes well, each instantiation of the instrumented program will create the file

/tmp/eqns on the machine where it executes. This file contains Maple-understandable equations describing the predicted performance of the PVM program. The file created by one of the instantiations of the instrumented program should be copied to the directory where the prediction system can find it; this is usually the same directory where the application resides (see Section A.2 below).

Occasionally, the different **eqns** files produced during a single instrumented run will contain slightly different equations describing predicted program performance. Usually this is because of boundary conditions or exceptions in the input data. For example, in the **jacobi** program described in Section 4.1, the processors that are assigned a group of rows on the top or bottom of the grid communicate half as much data as the other processors. In such cases, the most "representative" set of equations should be used for the performance model.

If an instrumented program does not terminate, or if a **eqns** file exists but is empty, it is probably due to an error during dynamic analysis (see Section A.1.7 below).

A.1.5 Obtaining Actual Execution Times

The APACHE source-to-source compiler produces a "timed" version of the PVM application it processes which can be used to measure actual execution times. When the timed program is executed, it will produce a message indicating the execution time of each block enclosed by instrumentation pragmas (see Section A.1.3 above). The messages are directed to **stdout**, so they will show up wherever **stdout** from spawned tasks is sent (by default in the PVM log file, **/tmp/pvml.<uid>** on the machine running the master PVM daemon).

Usually the times reported by the different PVM tasks differ slightly. Occasionally, one or two times are significantly smaller than the others. This is seems to be due to varying loads on the processors of the virtual machine. When recording execution times to check the accuracy of predictions, we have chosen the smallest time in cases where it is significantly different from the others.

A.1.6 Pragmas

The APACHE source-to-source compiler supports the following pragmas:

- **`#pragma instrument {on|off}`**. This pragma is used to turn instrumentation on and off. The equation for execution time that is generated by an instrumented program reflects the cost of executing program statements enclosed within instrumentation pragmas. Therefore, a program that will be instrumented must contain at least one pair of instrumentation pragmas (one **`on`** and one **`off`**).
- **`#pragma complexity <exp>`**, where **`<exp>`** is a Maple-understandable expression, potentially involving N (which represents the problem size), P (which represents the number of processors), and the iteration variables of enclosing **`for`** loops. This pragma is used to express the time complexity of a loop. A complexity pragma must appear somewhere within the body of every **`while`** and **`do`** loop. Occasionally **`for`** loops need a complexity pragma as well.
- **`#pragma datasize <exp>`**, where **`<exp>`** is a Maple-understandable expression, as in the complexity pragma, indicating the size of the data accessed by the program. Originally, this pragma was implemented to account for the impact of on-chip cache and virtual memory on execution time. Although the pragma is still supported by the source-to-source compiler, we found that there was usually enough error in the equation for computation time that it was pointless to try to account for cache and virtual memory effects.

Adding support for additional pragmas would be fairly straightforward.

A.1.7 Error Messages

Error handling by the source-to-source compiler is somewhat primitive. Most compile-time errors reported by **`inst`** are self-explanatory, but the compiler isn't very specific about where errors occur. For example, **`inst`** might report a **`while`** loop without a complexity pragma without indicating the line number where the loop begins.

The Makefile generated by the source-to-source compiler will be incorrect if the program being processed requires additional libraries (e.g., **`libm`**). In these cases, a link error will be reported, and the appropriate linker options must be added manually to the generated Makefile. Also, remember that the cost of executing subroutines is not included in the performance model generated by the APACHE system.

Dynamic analysis errors that occur during the instrumented run are reported to **`stderr`**. After an instrumented program is executed, the output from the spawned tasks should be checked for warnings and errors, especially if the program fails to terminate, or if the **`eqns`** file is not created or is empty (see Section A.1.4). Note that, by default, **`stderr`** from spawned PVM tasks is sent to the PVM log file.

A.2 Prediction System

This section documents the installation and usage of the prediction portion of the APACHE system.

A.2.1 Installation

The prediction portion of the APACHE consists of several files, all of which reside in the same directory. The important procedures are in the file named **`predict`**. Other files contain

Maple-understandable versions of the performance models on which the APACHE system is based, and cost factors for HP workstations and Ethernet and ATM networks.

Two modifications may need to be made to the **`predict`** file. At the top of the file, the variable **`PREDICT_DIR`** should be set to the path where the files for the prediction system are located. The **`PARAPPS_DIR`** should be set to the path containing subdirectories for the programs that will be processed by the prediction system.

A.2.2 Predicting Execution Time

Performance prediction activities are carried out in the Maple environment. To initialize the system, run **`xmaple`** in the directory where the APACHE performance prediction files are located, then run the command **`read(`predict`);`**. A usage message will appear for each of the procedures defined in the file. Short descriptions of the important procedures can be found in Section 3.4.

To prepare the system for predicting execution times, the user calls the **`predict`** routine to specify a type of workstation and communication network and the name of a program that has passed through static and dynamic analysis. For example, the system would be setup with for the **`jacobi`** program, HP workstations, and an Ethernet network with the command **`predict(hp, ether, matmult)`**. The **`predict`** routine performs the following setup tasks:

- Read values for the cost factors for the specified type of workstation. These factors correspond to the C_i values from the APACHE computation model (see Section 2.1). Values for these cost factors must be obtained from the workstations for which program performance will be predicted. The values that we have obtained for HP 715/80 workstations are in the file **`costs.hp`** and are reproduced in Table A.2 on page 75. If values for another type of workstation are to be used, they should be stored in a file named

costs.<wkst>, where **<wkst>** is the first parameter passed to the **predict** procedure.

- Read the communication costs for the specified type of network. These costs are the α, β, and (possibly) γ factors in the applicable communication models (see Section 2.2). The values that we have obtained values for Ethernet and ATM networks are in the files **costs.enet** and **costs.atm** and are reproduced in Table A.3 on page 75. To use values for another types of network, they should be stored in a file named **costs.<net>**, where **<net>** is the second parameter passed to the **predict** procedure.

- Read the equations predicting the behavior of the specified program. These are the equations generating during the dynamic analysis phase (see Section 3.3.3), and should be stored in a file named **eqns** in a subdirectory of the **PARAPPS_DIR** directory with the same name as the third parameter passed to the **predict** procedure.

The **predict** procedure initializes several global variables. The primary variables are shown in Table A.1. Other global variables are documented in the **predict** source code.

Plots similar to the ones in Section 4.2 can be produced with Maple's plotting commands. For example:

```
plot3d(Time, N=256..2048, P=2..8);
```

This command produces a 3-D plot showing the predicted execution time of the current application for problem sizes from 256 to 2048 and two to eight processors.

A.2.3 Comparing Predicted and Actual Times

Actual execution times should be stored in a file named **times.<net>**, where **<net>** indicates the network used during the timed runs, and is the same value passed as the second parameter to the **predict** procedure. The file should reside in same subdirectory of the **PARAPPS_DIR**

Variable Name	Meaning
`N`	Problem size
`P`	Number of processors
`Time`	Equation for execution time
`T_comp`	Equation for computation time
`T_comm`	Equation for communication time
`C_incdec`, etc.	Operation costs
`N_incdec`, etc.	Equations for operation counts

Table A.1. Important Maple variables in the APACHE performance prediction environment.

directory as the **`eqns`** file for the application (see the previous section). The execution times in the file are expressed as tuples of the form **`[N, P, Time]`**, indicating the problem size and the number of processors used during the run and the actual execution time in seconds.

To compare the actual execution times with predicted times, run the **`errs`** procedure. The procedure computes the prediction error for each of the actual times and plots a profile of the prediction errors for the entire set of times.

A.2.4 Model Parameters

The tables in this section contain the values we obtained for the parameters of the APACHE computation and communication models. Table A.2 shows the cost factors for the operations of the APACHE computation model, and the values we obtained for our HP workstations.

Table A.3 shows the values for the parameters of the APACHE communication models for Ethernet and ATM. To obtain these values, we timed PVM communications as described in Section 2.2. In our benchmark programs and the experiments presented in Chapter 4, we used PVM's **`PvmRouteDirect`** option to minimize message latency [32]. ATM times were measured using TCP/IP over ATM with a Fore ASX-200 ATM switch.

	Floating point		Integer	
Operation	$\overline{C_i}$	σ_{C_i}	$\overline{C_i}$	σ_{C_i}
Increment	—	—	0.0816	0.00010
Addition	0.0429	0.00014	0.0353	0.00028
Multiplication	0.0595	0.00039	0.2341	0.00027
Division	0.2107	0.00027	0.5064	0.00026
Modulo	—	—	1.0713	0.00047
Relational	—	—	0.0898	0.00006
Dereference	—	—	0.0880	0.00016

Table A.2. Operations and associated costs for HP 715/80 workstations.

Parameter	Meaning	Value (Ethernet)	Value (ATM)
α	Latency	871 μsec	960 μsec
β	Bandwidth	1.11 MBytes/sec	3.14 MBytes/sec
γ	Contention	P	—

Table A.3. Values for parameters of the communication models (see Section 2.2).

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[2] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys* **26** (4), 345–420 (1994).

[3] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An Interactive Environment for Data Partitioning and Distribution. In *Proceedings of the Fifth Distributed Memory Computing Conference*, April 1990.

[4] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A Static Performance Estimator to Guide Data Partitioning Decisions. In *Proceedings of PPoPP 91*, pages 213–223. ACM SIGPLAN Notices, 1991.

[5] A. L. Beguelin. Xab: A Tool for Monitoring PVM Programs. Tech. Rep. CMU-CS-93-164 (June 1993), Carnegie Mellon University, School of Computer Science.

[6] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and K. Moore. HeNCE: A Heterogeneous Network Computing Environment. Tech. Rep. CS-93-205 (August 1993), University of Tennessee.

[7] G. Bell. Ultracomputers: A Teraflop Before its Time. *Communications of the ACM* **35** (8), 26–47 (August 1992).

[8] W. S. Brainerd, C. H. Goldberg, and J. C. Adams. *Programmer's Guide to FORTRAN 90*. McGraw-Hill, 1990.

[9] J. Glenn Brookshear. *Theory of Computation: Formal Languages, Automata, and Complexity*. Benjamin/Cummings, 1989.

[10] R. Butler and E. Lusk. Monitors, Messages, and Clusters: The p4 Parallel Programming System. Tech. Rep. Preprint MCS-P362-0493 (1993), Argonne National Laboratory.

[11] C. H. Cap. Workstation Cluster Computation from the Perspective of the User. Tech. Rep. IFI-TR 94.06 (1994), Institut für Informatik, Universität Zürich.

[12] N. Carriero and D. Gelernter. LINDA in Context. *Communications of the ACM* **32** (4),

444–458 (1989).

[13] T. L. Casavant, J. A. Kohl, and Y. E. Papelis. Practical Use of Visualization for Parallel Systems. Tech. Rep. ECETR-920102 (1992), Department of Electrical and Computer Engineering, University of Iowa.

[14] S.-L. Chang, D. H. C. Du, J. Hsieh, M. Lin, and R. P. Tsang. Enhanced PVM Communications Over a High-Speed Local Area Network. In *Proceedings of the First International Workshop on High Speed Network Computing*, pages 37–46. IEEE Computer Society Press, April 1995.

[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] M. J. Clement. *Analytical Performance Prediction of Data-Parallel Programs*. Ph.D. thesis, Oregon State University, 1994.

[17] M. J. Clement and M. J. Quinn. Multivariate Statistical Techniques for Parallel Performance Prediction. In *Proceedings of the 28th Hawaii International Conference on System Sciences, HICSS-28*, January 1995.

[18] M. E. Crovella. *Performance Prediction and Tuning of Parallel Programs*. Ph.D. thesis, University of Rochester, 1994.

[19] S. K. Damodaran-Kamal and J. M. Francioni. Mdb: A Semantic Race Detection Tool for PVM. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, May 1994.

[20] P. W. Dowd, S. M. Srinidhi, E. Blade, and R. Claus. Issues in ATM Support of High Performance Geographically Distributed Computing. In *Proceedings of the First International Workshop on High Speed Network Computing*, pages 19–28. IEEE Computer Society Press, April 1995.

[21] A. C. Dusseau. Modeling Parallel Sorts with LogP on the CM-5. Tech. Rep. UCB//CSD-94-829 (1994), University of California, Berkeley.

[22] K. Efe, P. Uthayopas, V. Krishnamoorthy, and T. Dechsakulthorn. Parallel Algorithms for Workstation Clusters. In *Proceedings of 1995 ICPP Workshop on Challenges for Parallel Processing*, 1995.

[23] S. Even. *Graph Algorithms*. Computer Science Press, Potomac, Md., 1979.

[24] T. Fahringer and H. P. Zima. A Static Parameter Based Performance Prediction Tool for Parallel Programs. In *Proceedings of International Conference on Supercomputing*, pages 207–219. ACM SIGARCH, ACM Press, 1993.

[25] T. Fahringer and H. P. Zima. Evaluation of Benchmarking Performance Estimation for Parallel Fortran Programs on Massively Parallel SIMD and MIMD Computers. In *IEEE Proceedings of the 2nd Euromicro Workshop on Parallel and Distributed Processing, Malaga, Spain*, January 1994.

[26] J. Flower, A. Kolawa, and S. Bharadwaj. The Express Way to Distributed Processing. *Supercomputing Review*, 54–55 (May 1991).

[27] M. J. Flynn. Very High-Speed Computing Systems. *Proceedings of the IEEE*, 1901–1909 (December 1966).

[28] G. C. Fox. What Have We Learnt From Using Real Parallel Machines to Solve Real Problems?. Tech. Rep. C3P-522 (December 1989), California Institute of Technology.

[29] G. C. Fox, R. D. Williams, and P. C. Messina. *Parallel Computing Works!*. Morgan Kaufmann Publishers, 1994.

[30] J. Frankel. C* Language Reference Manual, Thinking Machines Corporation, 1991.

[31] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A Users' Guide to PICL: A Portable Instrumented Communication Library. Tech. Rep. ORNL/TM-11616 (August 1990), Oak Ridge National Laboratory.

[32] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Networked Computing*. MIT Press, 1994.

[33] M. Gokhale and W. Carlson. An Introduction to Compilation Issues for Parallel Machines. *The Journal of Supercomputing* **6**, 283–314 (1992).

[34] P. Hammarlund and B. Lisper. Data Parallel Programming: A Survey and a Proposal for a New Model. Tech. Rep. TRITA-IT-9308 (September 1993), Royal Institute of Technology, Department of Teleinformatics.

[35] M. T. Heath and J. A. Etheridge. Visualizing performance of parallel programs. Tech. Rep. ORNL/TM-11813 (May 1991), Oak Ridge National Laboratory.

[36] T. J. Hickey, J. Cohen, H. Hotta, and T. Petijean. Computer-Assisted Microanalysis of

Parallel Programs. *ACM Transactions on Programming Languages and Systems* **14** (1), 54–106 (1992).

[37] W. D. Hillis and G. L. Steele, Jr.. Data Parallel Algorithms. *Communications of the ACM* **29** (12), 1170–1183 (1986).

[38] J. A. Kohl and G. A. Geist. XPVM 1.0 User's Guide. Tech. Rep. 12981 (April 1995), Computer Science and Mathematics Division, Oak Ridge National Laboratory.

[39] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms*. Benjamin/Cummings, Redwood City, Calif., 1994.

[40] E. Levin. Grand challenges to computational science. *Communications of the ACM* **32** (12), 1456–1457 (1989).

[41] Message Passing Interface Forum. MPI: A Message-passing Interface Standard. Tech. Rep. CS-94-230 (April 1994), University of Tennessee, Computer Science Department.

[42] B. P. Miller, J. K. Hollingsworth, and M. D. Callaghan. The Paradyn Parallel Performance Tools and PVM. In *Environments and Tools for Parallel Scientific Computing*. SIAM Press, 1994.

[43] B. P. Miller, J. M. Cargille, R. B. Irvin, K. Kunchithapadam, M. D. Callaghan, J. K. Hollingsworth, K. L. Karavanic, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer* **28** (11) (1995).

[44] P. Newton and J. Dongarra. Overview of VPE: A Visual Environment for Message-Passing Parallel Programming. Tech. Rep. CS-94-261 (1994), University of Tennessee.

[45] C. M. Pancake. Performance Evaluation Tools for Parallel and Distributed Systems. *Computer* **28** (11), 16–19 (November 1995).

[46] C. Partridge. *Gigabit Networking*. Addison Wesley, Reading, Massachusetts, 1994.

[47] D. Pease, A. Ghafoor, I. Ahmad, D. Andrews, K. Foudil-Bey, T. Karpinski, M. Mikki, and M. Zerrouki. PAWS: A Performance Evaluation Tool for Parallel Computing Systems. *IEEE Computer*, 18–29 (January 1991).

[48] G. F. Pfister. *In Search of Clusters: The Coming Battle in Lowly Parallel Computing*. Prentice Hall PTR, 1995.

[49] M. J. Quinn and P. J. Hatcher. Data-Parallel Programming on Multicomputers. *IEEE Software* **7** (5), 69–76 (1990).

[50] M. J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, 1994.

[51] L. Rauchwerger and D. Padua. Parallelizing While Loops for Multiprocessor Systems. In *Proceedings of the 9th International Parallel Processing Symposium*, April 1995.

[52] D. A. Reed, R. D. Olson, R. A. Aydt, T. M. Madhyastha, T. Birkett, D. W. Jensen, B. A. A. Nazief, and B. K. Totty. Scalable Performance Environments for Parallel Systems. In *The Sixth Distributed Memory Computing Conference*, pages 562–569. IEEE Computer Society Press, April 1991.

[53] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society, 1993.

[54] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessor*. MIT Press, Cambridge, Mass., 1989.

[55] H. Schildt. *The Annotated ANSI C Standard: American National Standard for Programming Languages—C : ANSI/ISO 9899-1990*. Osborne McGraw-Hill, 1990.

[56] M. R. Steed, M. J. Clement, and P. E. Crandall. Network Media Selection for PVM Clusters *(Submitted)*. In *Proceedings ACM/SIGMETRICS Symposium on Parallel and Distributed Tools*, May 1996.

[57] M. R. Steed and M. J. Clement. Performance Prediction of PVM Programs *(To appear)*. In *Proceedings of the International Parallel Processing Symposium*, April 1996.

[58] Thinking Machines Corporation. Connection Machine: Programming in *Lisp. 6.1, 1991.

[59] H. Wabnig and G. Haring. Performance Prediction of Parallel Systems with Scalable Specifications—Methodology and Case Study. *Performance Evaluation Review* **22** (2–4), 46–62 (1995).

[60] S. Weisberg. *Applied Linear Regression*. John Wiley and Sons, New York, 1980.

[61] J. Yan. Performance Tuning with AIMS: An Automated Instrumentation and Monitoring System for Multicomputers. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, January 1994.

[62] H. Zhou and A. Geist. Faster Message Passing in PVM. In *Proceedings of the First International Workshop on High Speed Network Computing*, pages 67–72. IEEE Computer Society Press, April 1995.

[63] D. Zwillinger. *Handbook of Integration*. Jones and Bartlett, Boston, 1992.