Appeared in "Proceedings of Supercomputing '96"

Network Performance Modeling for PVM Clusters †

Mark J. Clement, Michael R. Steed
Computer Science Department
Brigham Young University
Provo, Utah 84602-6576

(801) 378-7608
clement@cs.byu.edu

Phyllis E. Crandall
Computer Science & Engineering
University of Connecticut
191 Auditorium Rd., U-155
Storrs, CT 06269-3155
(860) 486-2672
crandall@cse.uconn.edu

November 4, 1996

Abstract

The advantages of workstation clusters as a parallel computing platform include a superior price–performance ratio, availability, scalability, and ease of incremental growth. However, the performance of traditional LAN technologies such as Ethernet and FDDI rings are insufficient for many parallel applications.

This paper describes APACHE (Automated Pvm Application CHaracterization Environment) [16], an automated analysis system that uses an application-independent model for predicting the impact of ATM on the execution time of iterative parallel applications. APACHE has been used to predict the performance of several core applications that form the basis for many real scientific and engineering problems. We present a comparison of the performance predicted by APACHE with observed execution times to demonstrate the accuracy of our model. Finally, we present a method for a simple cost–benefit analysis that can be used to determine whether an investment in ATM equipment is justified for a particular workstation cluster environment.

Keywords: APACHE, Performance Model, ATM, Workstation Cluster

1 Introduction

Reasonable cost, increasing processor speeds, and improvements in communication network technology make clusters of workstations an attractive alternative to traditional parallel machines for high performance computing [2]. The popularity of this computational paradigm is evident from its substantial coverage in the current research literature and the proliferation of programming environments for workstation parallel processing, including the Parallel Virtual Machine (PVM) [10].

A number of network technologies are available for cluster computing. Ethernet connections are common, but are characterized by relatively low bandwidth, contention for the transmission medium, the possibility of transmission **collisions**, and the necessity of collision resolution. FDDI rings offer increased bandwidth and collision elimination, but retain the contention features of bus technology. The recent availability of Asynchronous Transfer Mode (ATM) switches presents a much different interprocessor communication model for the cluster

† This research was supported by Sprint Corporation.

environment. Not only is medium contention diminished or eradicated, but transmission speeds in the Gigabit range are becoming feasible [13, 14].

Other researchers who have compared the performance of Ethernet and ATM in PVM clusters report significant performance advantages for ATM [7, 12, 19]. However, not all PVM applications will benefit from ATM technology. The impact of different network technologies on specific parallel applications can be determined with performance prediction tools [6], and accurate performance prediction is critical to making media selection decisions. Since the performance of a specific application using high-speed switching technology largely depends upon the characteristics of the application itself, estimating the expected improvement is necessary to determine the efficacy of upgrading cluster connections. In this paper we present APACHE, an automated method for making this determination and illustrate its use in evaluating ATM for PVM applications.

The APACHE (Automated Pvm Application CHaracterization Environment) performance prediction system uses both analytical methods and empirically determined data to generate a performance model specific to a particular iterative application. APACHE automatically instruments PVM source code and experimentally determines the computational requirements of the application, its branching behavior, and the number and size of interprocessor communications. The gathered data are then used by the analytical model to determine the interaction of the application characteristics with the target network environment, and an estimate of execution time for the application on the specified platform is generated. When APACHE is used to predict the performance of a single application over different network configurations, the data produced may be used to determine the relative performance advantage of one platform over another.

To validate the accuracy of APACHE, the system has been applied to a collection of core applications that are often found at the heart of more complex scientific and engineering problems. We have selected Jacobi iteration, numerical integration, matrix multiplication, and parallel sorting as our test suite. APACHE assumes a Single Program Multiple Data (SPMD) model of computation, and communication patterns where all processors participate equally at each communication point. These assumptions are satisfied in numerous applications and, when they are met, the performance predicted by APACHE closely mirrors observed execution time.

This paper is organized as follows. Section 2 presents the performance models used in the APACHE system. In Section 3 we present the experimental validation of APACHE on our application test suite. Section 4 describes our method for network media selection for PVM clusters, including a simple cost–benefit analysis of ATM. Finally, Section 5 contains concluding remarks.

The research presented here was performed with PVM version 3.3.8 on HP 715/80 workstations, each equipped with a 10 Mbps Ethernet connection and a 155 Mbps ATM connection to a Fore ASX-200 ATM switch.

2 APACHE

The APACHE system represents the performance of a PVM application using an application-specific model, which is based on general computation and communication models and application information that is automatically derived from the source code.

2.1 Computation Model

The computation model represents the fraction of program execution time that is spent in computation. 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 [5, 18] to obtain values and confidence intervals for each C_i on hardware platforms of interest.

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.

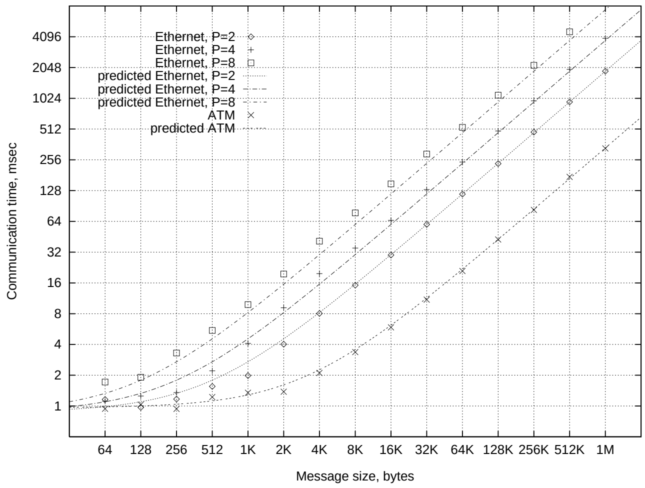


Figure 1: Predicted and actual times for PVM communications over Ethernet and ATM. Ethernet communication times are shown for different numbers of processors. Actual communication times for ATM are shown for 2 processors; higher numbers of processors resulted in similar times.

Unlike other performance prediction systems that assume a uniform cost for all operations, 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.

2.2 Communication Model

The communication model represents the portion of program execution time spent in communication. Other researchers who have modeled network performance report that a simple communication model that accounts for message startup time and network bandwidth gives adequate results [17]. According to this model, the predicted communication time T_{comm} for a given program execution can be expressed as

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

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 in a workstation cluster connected with ATM.

With shared-medium networks such as Ethernet, contention 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. Figure 1 compares the communication times predicted by Equations 2 and 3 with actual communication times for different message sizes and numbers of processors. All data points are for $N_c = 1$.

2.3 PVM Application Model

Adding together equations 1 and 3 provides a general equation for predicted execution time.

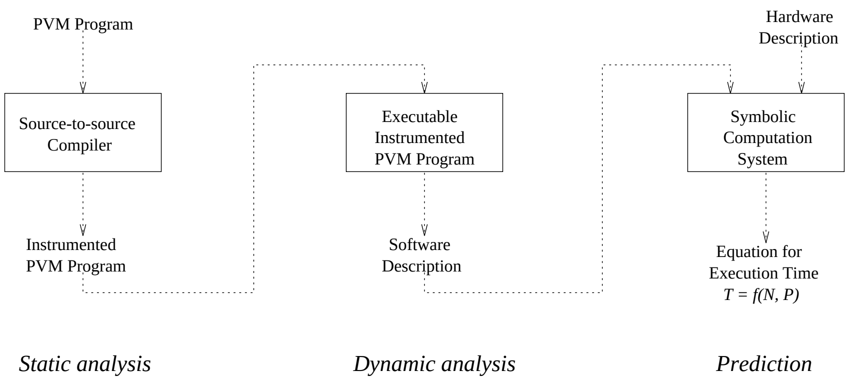


Figure 2: Construction of an application model with the APACHE system

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

In this equation, $N_{op(i)}$, N_c, and B_i vary with the application, the problem size N, and number of processors P. A PVM application model consists of equations for these parameters expressed as functions of N and P.

Setting γ to 1.0 effectively provides the communication equation 2 for ATM connections. By adjusting the value of γ equation 4 may be used for a variety of network connections.

To generate an application model, we run the APACHE source-to-source compiler on the application source code to produce a call graph and an instrumented version of the program. The instrumented program is run to record aspects of program behavior such as the number of times each block is executed and the number and size of PVM communications. The results of compile-time and runtime analysis are combined to obtain raw values for $N_{op(i)}$, N_c, and B_i for each block in the call graph. Given the values of N and P used during the instrumented run, simplified fits are performed to express $N_{op(i)}$, N_c, and B_i as 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, which are then used in Equation 4. We use the Maple [4] symbolic computation system to facilitate the generation and manipulation of the equations.

The diagram in Figure 2 summarizes 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 consists of 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.

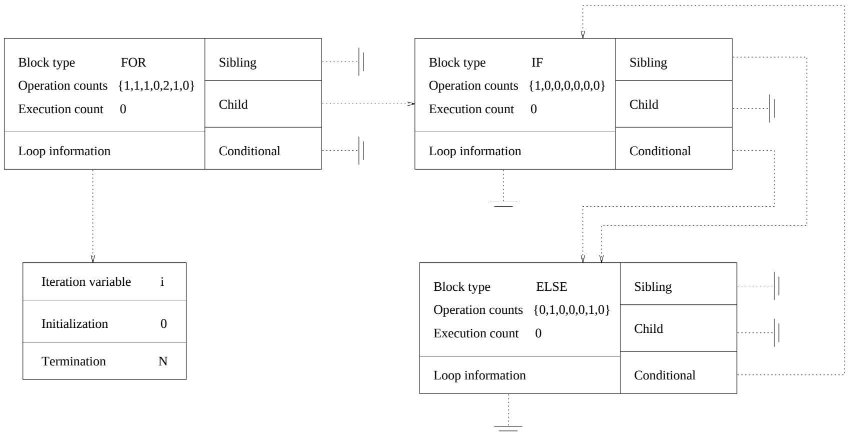
A **call graph**, as used in the APACHE system, is a graph representing the structure of a program [1]. The nodes of the graph represent the basic blocks of the program (i.e., sections of code such that control may pass into the block only at the first statement in the block and leave the block only 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. 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.

- **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. These include increment, addition, multiplication, division, modulo, relational, and dereference operators.

- **Execution count**. This field counts the number of times the block has been executed. It is used only during dynamic analysis.




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

Figure 3: Simple call graph generated by the APACHE source-to-source compiler, and the code fragment it represents.

- **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. The instrumented application is then run with a small problem size in order to determine true ratios for branches and to scale loop iteration counts to the problem size.

In order to generate final equations for execution time, 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).

Figure 4 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 2.

```
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,
   'iter'=1..1) + 1.000000 * sum(0, 'iter'=1..1) + sum(0 + 0.333333 *
   sum(0, 'iter'=1..1), 'slaveno'='1'..'P/2') + sum(0, 'i'='1'..'N/P')
   + sum(0, 'i'='1'..'N/P') + sum(0, 'i'='1'..'N/P'), 'iter'=1..1) );
```

Figure 4: Maple equation for the number of messages transmitted by the matrix multiplication program. Maple simplifies this equation to $N_{message}$:= P.

	Floating Point		Integer	
Operation	Mean Cost	Std Dev	Mean Cost	Std Dev
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 1: The times in μseconds on our testbed for each of the operations traced by APACHE.

3 Model Validation

We implemented several parallel algorithms using PVM to test the APACHE performance prediction system. In this section we present an overview of the algorithms and discuss their computation and communication complexity. Graphs of the predicted execution times across a varying number of processors and problem sizes are given for each application, along with the percent error between predicted and observed results.

Table 1 gives the times, in μseconds, for each of the operations on our testbed, and Table 2 shows the communication time components measured on our system.

3.0.1 Jacobi Algorithm

The Jacobi algorithm is an iterative method for solving the linear system $Ax = B$ for the unknown vector x[15], and it is an almost ideal fit for the assumptions made by APACHE. Both the computational and communication loads are well balanced among the processors. The algorithm uses estimates for each element of x obtained during iteration t to generate new estimates for iteration $t + 1$:

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 2: The communication overhead components.

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

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 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 in spite of the large number of communications.

Figure 5 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 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 is a straightforward relationship between the C source code and the compiled object code in terms of the computations that are performed.

3.0.2 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 $1 <= i <= N$, and let x_0 denote the beginning of the first interval. According to Simpson's Rule [20]:

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

Our `int` program implements parallel numerical integration by assigning 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" [9]. 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.

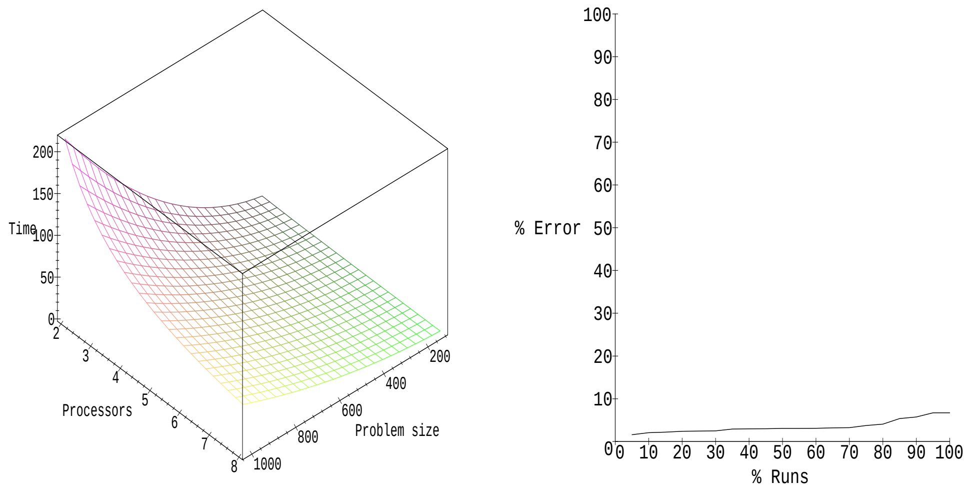


Figure 5: Predicted execution time and percent error of actual times for `jacobi`.

Figure 6 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 6 shows that all the observed times exhibit nearly the same amount of error, independent of the problem size. The APACHE system currently has no mechanism for automatically distinguishing between `float` and `double` variables in the source program. We explicitly indicated to the system that `double` values were to be used for the cost factors in the computation model. Incorporating automatic type detection is an area of future work. The percent error at all points is less than 2%, indicating a high degree of accuracy in computational component of the model for applications where load balance is equal among the processors. 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.

3.0.3 Matrix Multiplication

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

The predicted times for `matmult` are quite accurate, with about 90 percent of the runs having less than 15 percent error (see Figure 7). Since the largest prediction errors for this program result from overestimating execution time for runs with small grain size, we feel the prediction error is due largely to cache effects.

Because of the structure of the program, `matmult` was a good candidate for further analysis in order to

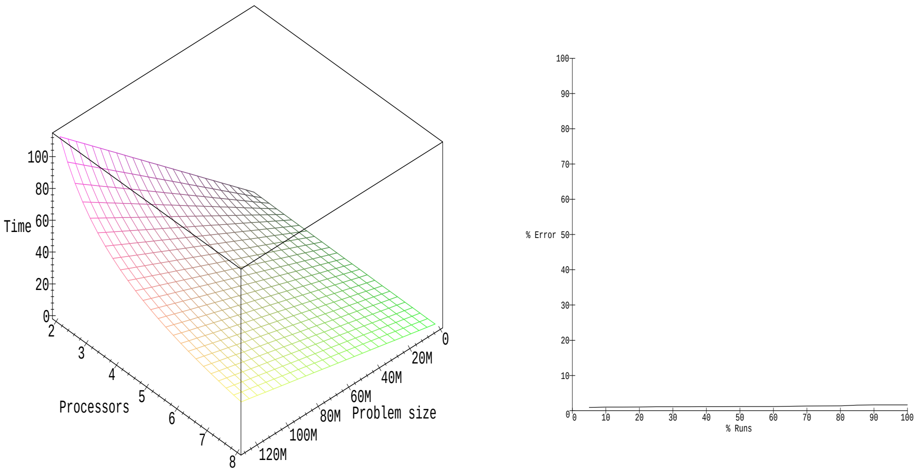


Figure 6: Predicted execution time and percent error of actual times for **int**.

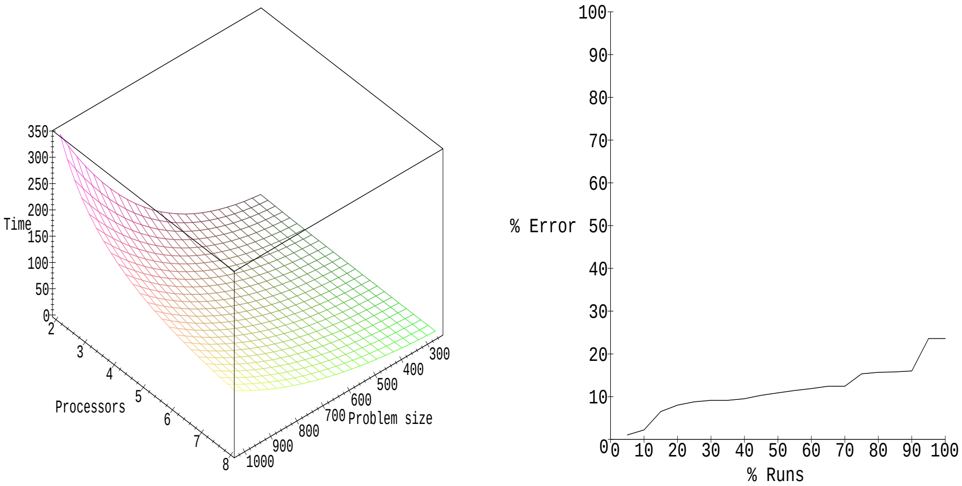


Figure 7: Predicted execution time and percent error of actual times for **matmult**.

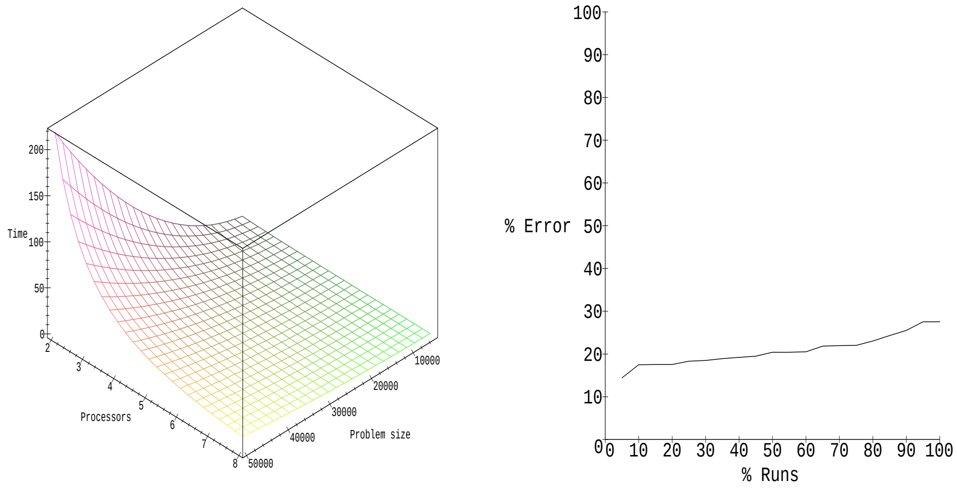


Figure 8: Predicted execution time and percent error of actual times for `sort`.

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 were inserted to measure computation 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, cache effects are the primary source of prediction error for `matmult`.

3.0.4 Parallel Sorting

Our `sort` program implements a bucket-based sorting algorithm similar to the one described in [8]. The algorithm assumes that N keys are to be sorted by P processors $P_1, \ldots, P_P$, where $N \gg P$, 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 $1 \leq i \leq P-1$. 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 [11].

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 $O(N/P)$. 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.

The APACHE system produced fair predictions of the performance of the `sort` program (see Figure 8). 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 for the program is divided among relational operations (i.e., comparisons), array dereferences, and increments. Since the predicted number of occurrences for all three

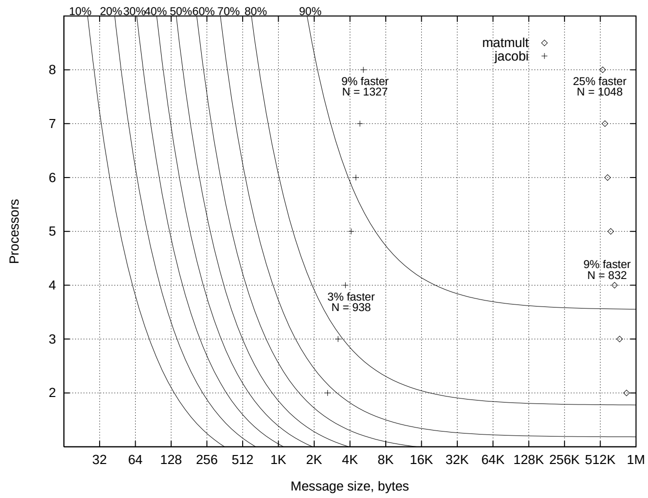


Figure 9: Predicted improvement in communication time of ATM over Ethernet. Each solid line represents a 10% improvement in communication time. Individual points show the predicted improvement in communication time for three PVM applications. Some points are labeled indicating the improvement in total execution time and the problem size.

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 further investigation is needed to more specifically identify the source of the error.

In most cases, the APACHE system was able to make acceptably accurate predictions of program performance. Cases where predictions were less satisfactory indicate directions for future work. Two factors that ostensibly affect the accuracy of the predictions made by the APACHE system are compiler output and CPU architectural effects. We believe that refinements or extensions to the system will be able to compensate for these factors.

4 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 ATM. A similar process could be used to evaluate other network technologies such as FDDI or switched Ethernet.

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 [3]. 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 9. 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

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

where T_{ATM} and T_{Ether} are derived from Equation 2 and Equation 3.

As Figure 9 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 two sets of points in Figure 9 show the predicted communication behavior of different PVM applications: `matmult` and `jacobi`(see Section 3). To obtain these points, we use the APACHE system to construct models of the applications as described in Section 2. 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 9, we set $T_{comp} = 10$ seconds.) With the expression for scaleup we compute the problem size N necessary to keep the grain size (the number of computations between communications) 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 9, 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 grain size, 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 as shown in Equation 2 (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 9, 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 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 Equation 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 10 shows surfaces 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.

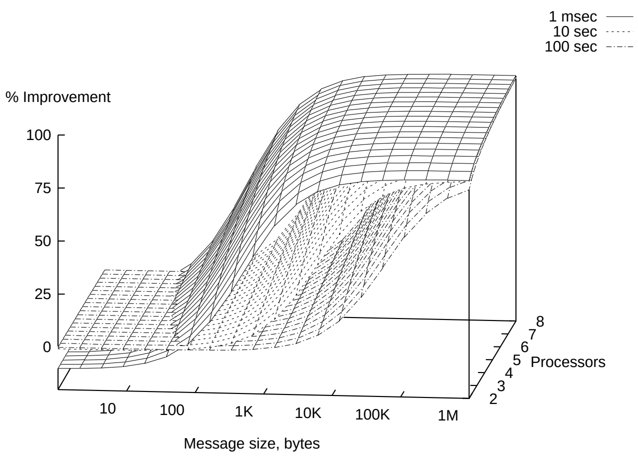


Figure 10: Predicted effect of ATM on program execution time as message size and the number of processors vary (see Equation 6). Results for three values of T_{comp} are shown.

Figure 11 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.

4.1 Cost–Benefit Analysis

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 $\frac{PC_w}{T_w}$. 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

$$T_{app} N_{app} \times \frac{PC_w}{T_w}$$

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

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

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

$$C_{sw} + PC_{nic} \leq I_{atm} T_{app} N_{app} \times \frac{PC_w}{T_w} \tag{7}$$

For example, given the following values

$C_w = \$15{,}000$	$T_w = 2$ years $= 63.12 \times 10^6$ 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

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

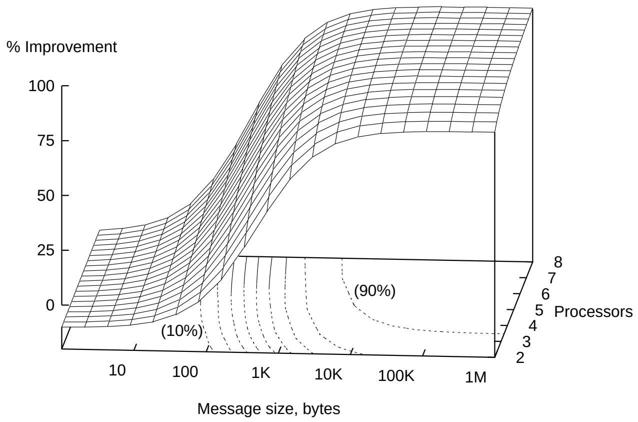


Figure 11: Predicted effect of ATM on program execution time as message size and the number of processors vary, $T_{comp} = 1$ msec (see Equation 6)). Contour lines mark improvement in 10increments.

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:

$$I_{atm} \geq \frac{T_w(C_{sw} + PC_{nic})}{T_{app}N_{app}PC_w}$$

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.

5 Conclusions

While high-speed switched networks offer a nearly contention-free transmission environment and Gigabit range transmission speeds, it is not necessarily the case that all PVM applications will benefit from using this newer technology. The computational characteristics and communication patterns of the application will determine whether or not a move to a different network platform will yield a sufficient performance improvement to offset the cost of upgrade.

The APACHE system allows quantification of expected performance improvement through analysis of application-specific characteristics and the features of the target network. The system automatically instruments the source code and empirically determines the computational features, communication patterns, and branching behavior of the application. The analytical model utilizes the gathered data to produce a performance estimate for the specified network. We have demonstrated that for certain classes of SPMD programs, APACHE provides accurate estimations of execution times.

The information provided by APACHE is crucial for making informed decisions regarding media selection for PVM applications and for developing cost–benefit analyses to aid in decisions regarding the efficacy of system upgrade.

Future work includes refining and extending the system to account for variation in the relationship of compiler output to source code, automatic determination of data types (e.g., floats versus doubles), and incorporating architectural considerations, such as cache behavior.

References

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

[2] G. Bell. Ultracomputers: A teraflop before its time. *Commun. ACM*, 35(8):26–47, August 1992.

[3] 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.

[4] 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.

[5] 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 3–6 1995.

[6] M. J. Clement and M. J. Quinn. Symbolic performance prediction of scalable parallel programs. In *Proceedings of the International Parallel Processing Symposium IPPS*, April 25–28 1995.

[7] 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.

[8] 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.

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

[10] 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.

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

[12] M. Lin, J. Hsieh, D. H. C. Du, J. P. Thomas, and J. A. MacDonald. Distributed network computing over local atm networks. *IEEE Journal on Selected Areas in Communications*, 13(4):733–748, 1995.

[13] A. Miller. From here to ATM. *IEEE Spectrum*, 31(6):20–24, June 1994.

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

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

[16] M. R. Steed and M. J. Clement. Performance prediction of PVM programs. In *Proceedings of the 10th International Parallel Processing Symposium*, April 15-19 1996.

[17] 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.

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

[19] 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.

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