To appear in the *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, 1995.

PERFSIM: A Tool for Automatic Performance Analysis of Data-Parallel Fortran Programs

SIVAN TOLEDO*

Laboratory for Computer Science
Massachusetts Institute of Technology

Abstract

This paper presents PERFSIM, *a tool for automatic performance analysis of CM Fortran programs running on the Connection Machine CM-5.* PERFSIM *executes the scalar part of the program, including all of its control structure, but estimates the running time of all vector operations, including both communication and computation. Our empirical studies show that the overall estimates are accurate to within a relative error of* $\pm 13\%$*, the estimates for vector computations are accurate to within* $\pm 21\%$*, and the estimates for vector communication are accurate to within* $\pm 13\%$*. These relative errors are comparable to deviations in the running time of programs running on the CM-5. By executing the cheap to execute but hard to analyze sequential control thread, and analyzing the easy to analyze but expensive to execute vector operations,* PERFSIM *produces accurate running time estimates efficiently. In particular,* PERFSIM *can execute on a workstation and generate in a few seconds performance data that would take several minutes or more to generate on a CM-5 using conventional performance monitors.*

1 Introduction

Scientists and engineers writing data-parallel Fortran applications for parallel supercomputers exert considerable efforts tuning their programs. The tuning process involves iteratively modifying the program, running it on realistic data sizes, collecting performance data, and analyzing the data to find further improvements. Tuning a scientific program for large data sizes uses substantial supercomputer time when the actual results of the computation are of no interest. Moreover, it is just plain slow.

This paper proposes a solution to this problem. We have written a tool, called PERFSIM, that analyzes CM Fortran[1] [9] programs running on the Connection Machine CM-5 [10]. By combining execution of some parts of the program and analysis of other parts, PERFSIM can execute a program on a workstation and in seconds, generate performance data that would take several minutes or more to generate by running the program on an actual CM-5[2]. Our empirical studies, described in Table 1, indicate that PERFSIM's overall estimates are accurate to within a relative error of $\pm 13\%$, the estimates for vector computations are accurate to within a relative error of $\pm 21\%$, and the estimates for vector communication are accurate to within $\pm 13\%$. Moreover, in all the test cases in which more than one version of a program was involved, PERFSIM correctly predicted the more efficient version. Section 8 examines the accuracy of PERFSIM more closely, and shows that these relative errors are comparable to deviations in the running time of programs running on the CM-5.

Data-parallel Fortran programs are composed of three classes of operations:

- sequential control and operations on scalars,
- computations on vectors, and
- communication operations involving vectors.

Operations on vectors are specified using vector notation and are interpreted as operations that are applied in parallel to all the vectors' elements. In the program fragment shown in Figure 1, the `do` loop is a sequential control structure, the summation of the vector `x` in line 3 is a vector communication operation, the update of the vector `x` in line 4 is a vector computation operation. and the scalar multiplication in line 5 is a scalar operation. PERFSIM executes the sequential control thread of the program and all operations on scalars, but instead of executing the vector operations, PERFSIM *estimates* the running time of all vector operations. Thus, PERFSIM executes the part of the program which is hard to analyze but cheap

*This research was supported in part by the Advanced Research Projects Agency under Grants N00014–91–J–1698 and N00014–94–1–0985, and by Project SCOUT under ARPA contract MDA972–92–J–1032.

[1] The current version of PERFSIM works with CM Fortran version 2.1 beta 0.1 with a run-time system dated 94/04/07.

[2] Due to a minor technical problem, it is currently not possible to execute PERFSIM on a workstation. It executes on a CM-5 partition manager without ever using the CM-5.

program name	author and affiliation	source lines	nodal code blocks	data size	relative errors				
					overall	nodal code	shift	reduce	permute
SOR-I	Alan Edelman, MIT	77	5	1024×1024	-10%	-17%	-0.3%	—	-2%
				2048×2048	-13%	-18%	-7%	—	-3%
SOR-II	Alan Edelman, MIT	92	4	1024×1024	-6%	-6%	$+11\%$	—	-2%
				2048×2048	-6%	-6%	-0.4%	—	-3%
QCD-KERNEL	Richard Brower, BU	132	20	$3^2 \times 4 \times 8^4$	-2%	-4%	$+5\%$	—	-3%
				$3^2 \times 4 \times 16^4$	-9%	-12%	-9%	—	-8%
CG3D	Sivan Toledo, MIT	205	35	$192 \times 192 \times 192$	$+2\%$	-15%	$+12\%$	$+2\%$	—
MG3D	NAS/Sivan Toledo, MIT	461	76	$130 \times 130 \times 130$	-2%	-14%	$+13\%$	—	$+6\%$
TURB	Danesh Tafti, NCSA	1003	25	$128 \times 128 \times 128$	$+4\%$	$+2\%$	$+7\%$	—	—
QCD-I	Souzhou Huang and Ruben Levi, MIT	1174	90	$3 \times 4 \times 8^4$	-3%	-1%	$+2\%$	—	-4%
				$3 \times 4 \times 16^4$	-5%	-18%	-1%	—	-2%
QCD-II	Taras Ivanenko, MIT	1225	90	$3 \times 4 \times 8^4$	-6%	-7%	$+2\%$	—	-6%
				$3 \times 4 \times 16^4$	-3%	-21%	-1%	—	-3%

Table 1: The programs used to assess PERFSIM's accuracy, their authors, sizes, the number of blocks of vector computations, called nodal code blocks, and the relative accuracy of estimates of the running time of both computation and communication operations when executed with various data sizes on a 32-node CM-5. The table shows the overall relative error, the error on vector computations, and the errors for shift operations, global reductions (such as summations), and permutations. Missing entries in the table, designated by a —, indicate that the vector operation is an insignificant cost in the program.

to execute, and analyzes the part of the program which is expensive and slow to execute but relatively easy to analyze.

We faced six important problems in the implementation of PERFSIM. The rest of the paper describes these problems and our solutions. The most challenging problem of estimating the running time of vector communication operations is described in Section 2. Section 3 describes the estimation of the running time of vector computations. Section 4 describes how the execution of the scalar part of a program is integrated with an estimation of the vector part. Since in a program's execution the control flow may cause some operations not to be executed, the problem of data dependent execution order must be addressed. We describe our solution to this problem in Section 5. We implemented a data visualization tool that enables the programmer to analyze the large amount of performance data generated by PERFSIM, which is described in Section 6. The last, but certainly not least challenge, was to use PERFSIM's performance models to present the user with information beyond the running time of different parts of the program. Section 7 describes our efforts in this direction, and in particular the ability of PERFSIM to extrapolate the running time from executions on small data sizes to executions on larger data sizes. We conclude the paper in Section 9 with a discussion of our results and of related work.

```
      real a,b,x(n)
      do i = 1,100
        a = sum(x)
        x = x/a
        a = a*b
      end do
```

Figure 1: A data-parallel Fortran program fragment.

2 Analyzing Communication Operations

This section describes our solution to the problem of estimating the running time of communication operations. There are several classes of communication operations in data-parallel languages [3] such as CM Fortran. Arrays can be permuted. They can be shifted along one of their axes. They can also be reduced to scalars as in summations. Finally, scans, or parallel prefix operations, can be computed. While scans and reductions perform some computation as well as communication, we still classify them as communication operations for the purpose of automatic performance analysis.

Our strategy for estimating the running time of a communication operation consists of four phases. In the first phase we identify all the communication resources in the architecture. Each resource corresponds to one type of communication channel in which data is transferred from one memory location to another. In the second phase we quantify the bandwidth and latency of each communication resource, and the degree to which different resources can be utilized in parallel. In the third phase we determine the number of words that must be transferred on each of the communication channels, which we call *transfer sizes*. Combining the transfer sizes with the capacity of each resource and the amount of parallelism to produce a running time estimate is the fourth and last phase. We now describe how each phase is carried out and how we determined the parameters for PERFSIM's performance models.

Identifying communication resources. The identification of the communication resources in a given architecture needs to be performed only once. The CM-5 contains five different communication channels on the CM-5, one of which contains several subdivisions. Figure 2 illustrates the following communication channels in the CM-5.

- Data transfer from one location to another in the same memory bank. These data transfers are performed by loading the data to a vector unit attached to the memory bank and storing it back to another location.
- Data transfer from one memory bank to another memory bank attached to the same vector unit chip (two memory banks are attached to each vector unit chip).
- Data transfer from one memory bank to another attached to another vector unit chip on the same node.
- Data transfer from one node to another using the CM-5's data network [7].
- Data transfer from one node to another using the CM-5's control network. The control network supports specialized communication patterns such as reductions and broadcasts.

The topology of the data network is a 4-ary tree, in which links at different levels of the tree have different capacities. In particular, the capacity of links doubles at every level as we approach the root in the first 3 levels of the tree, but quadruples from level 4 through the root, as shown in Figure 2. Since the capacity of each level in the tree is different, each level of the data network must be modeled separately.

Quantifying the bandwidth, latency, and parallelism of communication channels. The characteristics of the communication channels are functions of the implementation of the architecture in hardware and the software's use of that hardware. We have found that software has a large influence on the effective bandwidth of communication channels on the CM-5. For example, we have found a pair of subroutines in the CM run-time system that differ by more than a factor of 4 in the effective bandwidth they achieve on some communication channels, even when the communication patterns they use are essentially the same. Therefore, we decided not to assign a fixed bandwidth to each communication channel, but rather, to assign the bandwidth of most communication channels on a subroutine by subroutine basis. In our experience, software affects most the performance of communication channels within nodes, and therefore the bandwidth of the communication channels within each node, including the NI, is assigned per channel and per subroutine, but the assumed bandwidth of communication channels in the data network is the same for all subroutines (but of coarse different for different channels). The estimation process for these parameters is described later in this section.

The latency of the communication channels contributes to a constant term in the running time of subroutines. Instead of quantifying the latency of each communication channel, we estimate the constant term in the running time of each subroutine.

We have found that while all the channels of the same type in the CM-5 operate in parallel, different types of channels are

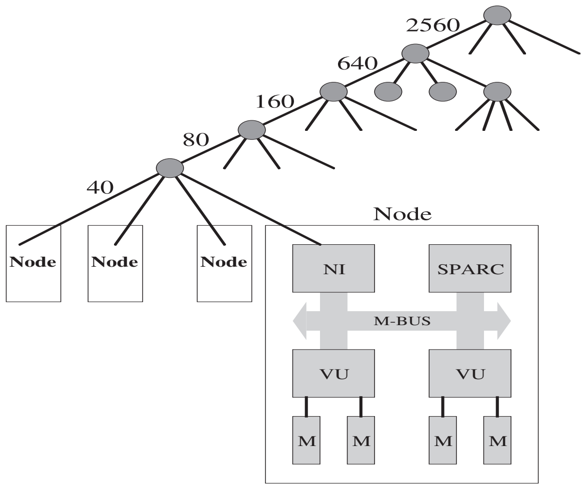

Figure 2: A schematic diagram of the CM-5. Each node contains a network interface (NI), a SPARC microprocessor, two vector unit chips (VU's), and four memory banks (M's), two attached to each VU chip. The SPARC microprocessor, the VU's, and the NI, are connected by a memory bus (M-BUS). The nodes are linked by a data network and a control network, both of which have a tree structure. The shaded circles represent routing nodes in the data network. In this diagram only part of the data network is shown, with the numbers near the links designating the capacities in megabytes per second in each direction.

almost never operated in parallel. For example, all the vector units in the machine compute in parallel, but this operation does not happen concurrently with data transfer on the data network.

Therefore, the total time it takes to perform a communication operation is the sum of the time it takes to transfer the required amount of data on each of the communication channels, plus a constant overhead term representing the latency of all the channels. If, for example, new software permitted all the communication channels to be used in parallel, the running time estimate would be the maximum rather than the sum of transfer times on the different channels.

Determining the resource requirements of subroutines. The number of data items that must be transferred on each communication channel depends on the functionality of the subroutine, on the value of scalar and array arguments, and on the layout of array arguments in the machine. PERFSIM uses knowledge about the subroutine functionality, the scalar arguments, and the geometry of array arguments to determine the number of data items that must be transferred on each communication channel. A *geometry* is an object that describes the size and layout of arrays in the CM run-time system. Geometries are scalar objects, and thus, they are allocated and operated upon in PERFSIM's executions. The value of an array however, cannot be determined in PERFSIM's executions, and therefore the running time of communication operations whose running time depends on the value of array arguments, especially permutation operations (`sends`), cannot be estimated accurately.

Most the variance in the running time of permutations on the CM-5 stems from the different loads that different permutations place on the data network. We have chosen to assume that the load is light, and that the data network is not a bottleneck in permutation routing. Even in cases in which PERFSIM fails to provide accurate predictions, the predictions are still useful, since the estimation inaccuracy often remains consistent over program modifications. For example, PERFSIM always predicts correctly that the operation `a = b(pi)`, where `pi` is a permutation array, is much more efficient than `a(pi) = b`, even though the overall accuracy might not be good.

Parameter Estimation for PERFSIM's models. PERFSIM has a database of five model parameters for each subroutine. Four of the parameters describe the bandwidth in terms of the amortized transfer time for one data item on each of the four communication channels other than the control and data networks, and the fifth describes the constant overhead term in the running time, which includes the latency of the control network if it is used.

We estimated these parameters manually, starting from rough values based on the technical literature, and refining them by measuring the running time of subroutines on specific array layouts. We preferred this method over a least squares fit (with or without scaling), because it proved hard to design experiments for robust estimation of the five parameters PERFSIM uses.

The bandwidth of the communication channels in the data network was taken from Thinking Machines' technical data.

3 Analyzing Computation Operations

This section describes how PERFSIM estimates the running time of computations on arrays. The CM Fortran compiler translates computations on arrays to *nodal code blocks*, which are subroutines written in SPARC and vector unit assembly language. These subroutines operate on one or several arrays with the same geometry. (A program statement specifying a computation on two arrays with different geometries translates into a communication operation that transforms the geometry of one of the arrays, and then a nodal code block is invoked.)

Our approach is simple. PERFSIM analyzes the compiler generated code block and determines the amortized number of cycles it takes to operate on one element in each array. During execution this number is multiplied by the number of array elements that need to be operated upon by each vector unit, since all the vector units work in parallel. A fixed overhead that models the invocation and initialization of the code block is added. The main difficulty is estimating the number of cycles it takes to operate on one element.

Each code block contains a main loop in which array elements are processed in groups of 8 by the vector unit. The code blocks may be quite long and involve many loads and stores of arguments, results, and partial results. Within the main loop, a run-time library called CMTRA may be called. CMTRA functions take their arguments in vector registers and compute trigonometric functions, exponentials and so on. The results are also returned in vector registers. Finally, the main loop contains SPARC code to increment the arguments' addresses and to decrement the number of elements to be operated upon by 8.

We have written a program that analyzes the code blocks. The program finds the main loop and analyzes each instruction within the main loop. The program estimates how many cycles the instruction will take to execute, which depends whether it is a SPARC instruction or a VU instruction, which VU instruction (in particular, divisions, square roots, and stores of single words take longer than other instructions), and whether or not bubbles will be created in the VU pipeline. Bubbles depend on sequences of two consecutive VU instructions that cannot be performed back to back without stalling the pipeline. The program also incorporates estimates for the running time of CMTRA calls, which is documented in the Connection Machine literature.

4 Integrating Execution and Estimation

In this section we start to describe the integration of the execution of the scalar part of the program with the estimation of the running time of vector operations. This section focuses on the mechanism PERFSIM employs to achieve this integration, and the next section focuses on the problem of executing the program correctly in the absence of vector data. For now, we assume that all the scalar data is valid.

The CM-5, as well as all the other Connection Machines and some other parallel supercomputers, uses dedicated hardware to execute the scalar part of the program. On the CM-5 this hardware, called the *partition manager*, is a Sun workstation equipped with a network interface. The CM Fortran compiler generates a SPARC assembly language file containing the scalar part of the program. Any vector operation is translated into one or more function calls to the CM *run-time system*. These functions invoke subroutines on the CM-5 nodes and transfer data between the partition manager and the nodes.

PERFSIM modifies the assembly language code generated by the compiler for the partition manager . Each call to the CM run-time system is replaced by a call to a corresponding function in PERFSIM's run-time system. For the most part, PERFSIM's run-time functions use their arguments to estimate the running time of the corresponding CM run-time function, and do not operate on vectors. Memory allocation and deallocation subroutines in the CM run-time library are replaced by subroutines that do not allocate memory on the nodes, but record the amount of allocated memory for later analysis.

Modifying the assembly language code generated by the compiler rather than the Fortran source code has two advantages. First, some language constructs are transformed by the compiler to function calls, so it is not possible to intercept all the function calls by modifying the Fortran source. For example, the expression `a = b(p)` where `a`, `b` and `p` are arrays, is transformed into several function calls to subroutines that route the elements of `b` according to the permutation `p`. Second, by transforming the compiler output rather than the compiler input, we are guaranteed to profile the program with all compiler optimizations in place.

The only functions in the CM run-time system that are called during PERFSIM's executions are those that allocate geometries. These functions do not access the CM-5's nodes, and calling them ensures that PERFSIM's subroutines receive as arguments exactly the same geometries passed to the CM run-time system in normal executions on the CM-5.

5 Handling Data Dependent Decisions

Since in PERFSIM's executions arrays do not exist, decisions that depend on values stored in distributed arrays must be replaced by data independent decisions. This section describes *multiplexers*, which form the core of PERFSIM's support for handling data dependencies. Users can slightly modify their programs using multiplexers, to create a program with only one source and one binary, which is equivalent to the original program in normal CM-5 executions, but has no data dependencies when executed under PERFSIM.

Multiplexers are functions that return either their first argument or their second, depending upon whether PERFSIM executes the program or a CM-5 executes the program. For example, the expression `integer_mux(100,n)` returns the constant `100` in PERFSIM executions and the value of `n` in normal executions. To assist in modifying data dependent `while` loops, PERFSIM provides a generalized multiplexer called `loop_mux`. A data dependent loop such as `do while sum(A)` $\geq$ `0` in the original program is replaced by `do while loop_mux(500,sum(A)` $\geq$ `0)`. The transformed loop is equivalent to the original in normal CM-5 executions, but equivalent to `do i=1,500` in PERFSIM's executions (when the loop index i is a temporary variable used nowhere else).

All the multiplexers work by looking up a special variable which is set at the beginning of the execution and which indicates whether this is a normal execution on a CM-5 or a PERFSIM's execution. `loop_mux` works in PERFSIM's executions by keeping a hash table with the addresses of loops and the number of times each loop was already executed. The table is initially empty. When `loop_mux` is called, it checks whether the particular loop which generated the call is in the table. If it is, `loop_mux` increments the loop counter in the table. If the counter does not exceed the argument of the `loop_mux`, then `loop_mux` returns `true`. If the counter exceeds the argument, the entry is removed from the table, and `false` is returned. If the loop is not found in the table, it is inserted with a counter of 1 and `true` is returned. The removal of loops from the table ensures correct behavior for loops that are reached more than once in the program, e.g. nested loops.

To assist the user in identifying the data-dependent control flow decisions in the program, PERFSIM can execute in a special mode in which decisions that depend on floating-point distributed arrays cause a trap with an appropriate error message. This feature of PERFSIM uses NaN's which cause a floating-point exception when used as arguments of compare instructions. Unfortunately, there is no comparable mechanism for tagging integers and logical data, so PERFSIM cannot provide similar support for other than floating-point data.

6 Data Visualization

A performance monitoring tool must provide a data visualization facility to enable the programmer to analyze and use the large amount of data gathered in a typical execution. This section describes PERFSIM's data visualization facility.

The data visualization tool, which is implemented as a MATLAB application with a graphical user interface, displays three graphs (see Figure 3). The most important graph in the display shows the estimated CPU time usage for each source line. The total time is displayed by default, but the time associated with different classes of computation and communication operations can be displayed as well. By clicking on a bar in the display, the user can determine the source line number associated with that bar. Another graph is a histogram showing the fraction of the total time spent in different classes of communication and computation operations. The data visualization tool also displays the memory allocation as a function of execution time. Stack, heap, and total allocation can be displayed. This display helps the user locate spots in the program where large amounts of memory are allocated and where, therefore, the program is prone to running out of memory. It also enables the user to determine the size of compiler generated temporary arrays, which are always allocated on the heap.

During the first iteration of an iterative tuning process for a program, the user executes the program on an actual CM-5. PERFSIM generates for such executions both running time estimates and measurements of the running time. The data analysis tool displays the estimates and the actual running times in two different graphs, and the relative estimation errors for each class of operations. The user can use this information to assess the estimation accuracy of PERFSIM on that particular program. If the overall accuracy is good, the user proceeds to use PERFSIM on a workstation to optimize the program. If the accuracy is not good, the user uses PERFSIM to optimize the parts of the program in which the estimates were good, but he or she must use executions on the CM-5 to optimize the part of the program that was not estimated accurately. The main source of inaccurate estimates in PERFSIM is permuting arrays, since the running time depends on the permutation, which is a vector that does not exist in PERFSIM's executions. For example, when the first phase of a program is sorting its input (sorting involves permutation routing), the running time of this phase is not estimated accurately, since the actual running time depends on the input data, but PERFSIM's estimates do not. The fact that PERFSIM breaks down the running time by source line and by operation types, however, allows the user to use PERFSIM even if the estimates for some lines or some operation types are not accurate.

7 Performance Extrapolation

The fact that PERFSIM uses analytic models for the running time of vector operations enables it to provide much more information than ordinary performance monitoring tools that rely on measurements of actual vector operations. In particular, PERFSIM can break down the fraction of the time spent in a class of vector operations into components corresponding to time spent using specific communication and computation resources. PERFSIM can also accurately extrapolate the total running time and the fraction associated with each class of vector operations from executions on small data sizes to executions on large data sizes. For example, PERFSIM can generate the entire graphs describing the estimated running times of three programs, shown in Figure 4, from two executions on small arrays of size 4096 and 8192. The implementation of the first feature is quite simple, so in this section we focus on the second, performance data extrapolation.

Let us first explain what we mean by performance extrapolation. We assume that the program is scalable, that is, its data size can grow as a function of some parameters. In typical numerical modeling, these parameters might govern the discretization of space or the size of the domain being modeled by the program. When the data size grows, two things might happen. First, the program may require more iterations to reach convergence or termination. This phenomenon is problem- and algorithm-specific, and we do not attempt to model it. The second phenomenon is that the relative costs of different operations change. An operation that takes most of the execution time on small data sizes, say computations in nodal code blocks, may be replaced by another, say permutation routing, which takes most of the execution time on large data sizes. Since the running time of programs under PERFSIM is independent of the data size, it is possible to use PERFSIM to estimate the running time and its breakdown on large data sizes. But it is also instructive to see the gradual change from one data size to a larger one, and this is exactly why we have incorporated extrapolation into PERFSIM.

PERFSIM extrapolates the running time and its components based on one parameter governing the data size. We chose to use only one parameter in order to enable effective presentation of the extrapolation results. To perform extrapolation, the user executes the program under PERFSIM twice on two different data sizes. Let us denote the two data sizes λ_1 and λ_2. During each of the two executions PERFSIM records which run-time subroutines were called on what geometries, how many times, and most importantly, how much data had to be transferred on each of the communication channels. By comparing the two executions, PERFSIM determines the rate at which each transfer size grows, and is able to correctly extrapolate each component of the running time. For example, let us suppose that in a particular vector communication operation n_1 words had to be sent out and received in each

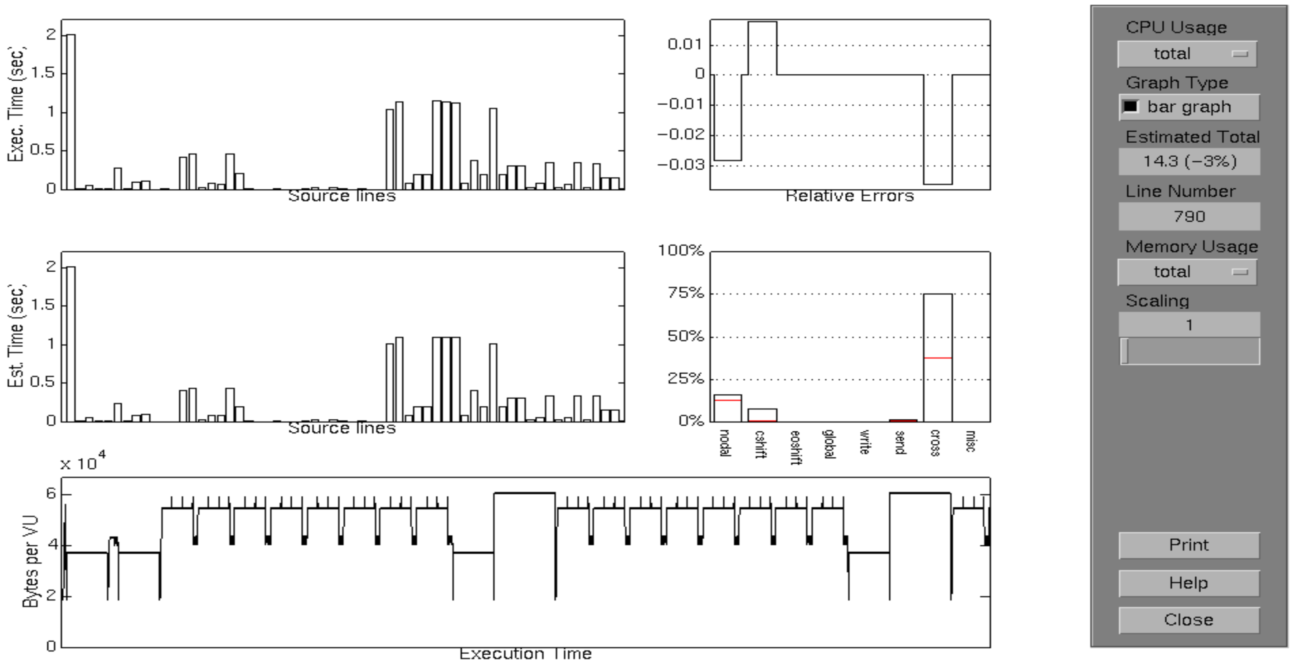

Figure 3: Performance data displayed by the data visualization tool, with its user interface controls on the right. The upper left graph shows the total estimated CPU time per source line. The user can determine the source line number associated with a particular bar by clicking on it. This graph can be modified by the user to show the time associated with any class of operations instead of the total time. The histogram on the upper right corner shows the relative amounts of time different classes of computation and communication operations are estimated to use. The horizontal line near the bottom of each bar designates the fraction of the time which is associated with various overheads. The graph at the bottom shows the total amount of memory allocation as a function of execution time. The user can zoom in and out to examine specific portions of the executions, and can request that the allocation on the heap and on the stack be displayed as well.

CM-5 node in one execution, and that in the second execution the same operation required sending and receiving n_2 words. PERFSIM determines that when the data size grows by a factor of λ_1/λ_2, the number of words sent in and out of each node in that particular operation grows by a factor of n_1/n_2, and this is the factor used to scale the data transfer time. This method handles correctly situations in which different arrays in the program grow at different rates, surface-to-volume effects in many communication operations (i.e. different transfer sizes in a single operation grow at different rates), and nonlinear behaviors in operations such as matrix multiplication.

To extrapolate the running time as a function of more than one parameter, say both λ and η, 3 different executions would be necessary, one serving as a base case, another with the same η but a different λ, and one with the same λ as one of the previous ones but a different η. The pair of executions with the same η is used to determine the dependency of transfer sizes on λ, and the two executions with the same λ are used to determine the dependency on η. In general, $k+1$ executions are necessary for extrapolation based on k independent parameters.

Storing the transfer sizes associated with every single vector operation in an execution generates a tremendous amount of data. We have implemented a data compression scheme, which is based on aggregating all the invocations of a given run-time system subroutine which gave rise to the same transfer sizes. This approach reduces both the amount of data that must be generated during an execution, and the time it takes to extrapolate the running time. However, this approach may introduce array size aliasing in which two groups of arrays which grow at different rates, have exactly the same size in one of the two executions. Suppose for example that a program allocates both arrays of fixed size 32 and arrays whose size depend on a parameter λ. In executions with $\lambda = 32$, PERFSIM does not distinguish between operations on fixed size arrays and variable size arrays. To avoid incorrect extrapolation, PERFSIM compares the number of subroutine calls for each combination of transfer sizes, and performs extrapolation only if the number of calls match. When array size aliasing occur, the number of calls do not match and extrapolation does not take place.

8 Estimation Accuracy

In this section we discuss the accuracy of PERFSIM. We show that PERFSIM's accuracy is of the same order of magnitude as the standard deviation of the running time of programs on the CM-5, that the estimates are accurate enough to be useful, and we trace and document the sources of estimation inaccuracy. We begin however with a discussion of the consistency of running time measurements, which are key to conventional performance visualization.

We ran some of the benchmark programs 10 times each, on a time-shared 32-node CM-5, in order to determine how consistent running time measurements are. The following table documents for each program and array size the average running time μ of all vector operations, and the standard deviation σ and the maximum deviation $d_{\max}$ as a fraction of μ. These deviations are of the same order of magnitude as PERFSIM's estimation errors reported in Table 1.

Prog.	Array Size	μ	σ/μ	$d_{\max}/\mu$
CG3D	$16 \times 16 \times 16$	0.975	8.4%	19%
CG3D	$128 \times 128 \times 128$	40.7	1.8%	3.4%
MG3D	$32 \times 32 \times 32$	5.46	8.7%	−18%

Inconsistencies in running time measurements, especially on small arrays, form a barrier for accurately tuning programs with conventional profilers that rely on running time measurements of single executions.

Before turning to the details of our empirical studies, we describe an illustrative example involving the estimation of the running time of three different implementations of the same subroutine, a simple one dimensional red black relaxation described in Figure 5. We have coded the subroutine using logical masks to represent the red (or even) and black (odd) points in the domain, using array sections, and using separate arrays to store the red and black parts of the domain. Figure 4 compares the actual and estimated running times for various problem sizes, on a log-log scale. The figure clearly shows that the estimates are accurate enough to decide on the relative performance of the three subroutines for nearly all problem sizes.

We now turn to the details of our empirical studies. Table 1 contains results of experiments we conducted to determine PERFSIM's estimation accuracy for communication and computation operations. None of the programs reported here were used to tune PERFSIM's model. All the errors reported in this table and the rest of the paper are relative errors. We define the relative error E_r of an operation whose actual running time is T_{actual} and whose estimated running time is T_{est} to be

$$E_r = \frac{T_{est} - T_{actual}}{T_{actual}} .$$

The relative error is therefore between -1 and $+\infty$, and is positive for overestimates and negative for underestimates.

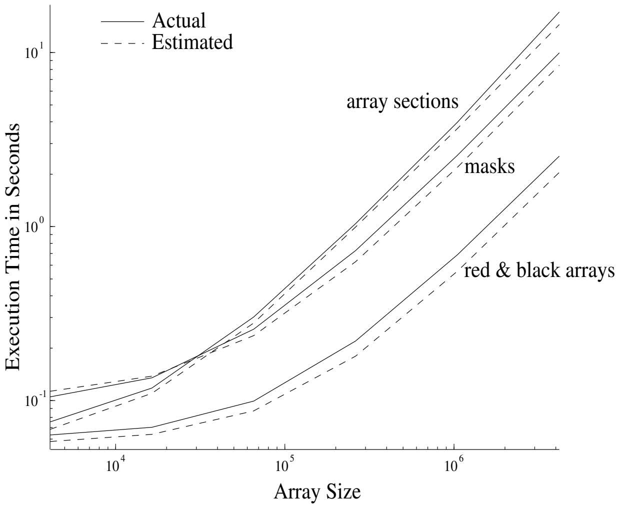

Figure 4: The log of the actual and estimated running times of three red black relaxation subroutines as a function of the log of the domain size. PERFSIM can generate the entire graphs describing the estimated running times from two executions on arrays of size 4096 and 8192, both well to the left of the crossover point.

We end this section with a list of the main sources of inaccuracy in PERFSIM's estimates. Each item in the list describes one source of inaccuracy, possible remedies, and the impact of the inaccuracies on PERFSIM's use.

Several different subroutines underlying one run-time system call. Some of the subroutines in the CM run-time system are actually driver subroutines that may call several different subroutines depending on the layout of the arrays passed.

A simple remedy would be to intercept run-time system calls made to actual subroutines rather than to driver subroutines. We could not implement PERFSIM in such a way since we had no access to the sources of the run-time system, but this approach is certainly possible when sources are available.

Uneven sequential overheads. Some calls to the CM run-time system take much longer than other calls with identical arguments, due to sequential overheads incurred only on some calls. In the case of driver subroutines mentioned above, the overhead is probably caused by the decision making process required to determine which actual subroutine to use. This

Data declarations	`double precision x(n),b(n)`
	`double precision xred(n/2),xblk(n/2),bred(n/2),bblk(n/2)`
	`logical          red(n),blk(n)`

Red Black Relaxation Using Logical Masks	`where (red) x = 0.5 * (eoshift(x,1,1)+eoshift(x,1,-1)+b)`
Red Black Relaxation Using Separate Black and Red Arrays	`xred = 0.5 * (xblk+eoshift(xblk,1,-1)+bred)`
Red Black Relaxation Using Array Sections	`x(3:n-1:2) = 0.5 * (x(2:n-2:2)+x(4:n:2)+b(3:n-1:2))` `x(1)       = 0.5 * (x(2)+b(2))`

Figure 5: The table shows the code for updating the red points of the domain in three different implementations of a red black relaxation subroutine, and the data declarations for the subroutine.

information is cached and used on subsequent calls with the same arguments. For example, out of 50 calls with the same arguments to the `cshift` subroutine, the first call took 1.3ms where as the other 49 all took between 0.15ms and 0.19ms.

Here again, intercepting run-time system calls to the subroutines that actually perform the data movement rather than to driver subroutines which may have large sequential overheads on some calls would fix the problems. In other words, the sequential overheads, like the rest of the sequential control, should be executed rather than estimated.

Performance bugs. Some subroutines consistently exhibit different performance on different arguments even though the transfer sizes are exactly the same. For example, while executing the program CG3D on arrays of size 32^3, we have found that shifting an array along the first axis by distance of one takes a different amount of time depending on the direction of the shift. Over 5 different executions, each of which consisted of 500 iterations, shifting by 1 always took 0.184 seconds (for all 500 shifts combined), whereas shifting the same array along the same axis by -1 took between 0.193 and 196 seconds, a slowdown of more than 4.8%.

There is way and no reason to *model* performance bugs. As much as possible, they should be eliminated. Our experience has shown that PERFSIM is valuable for finding performance bugs, since such bugs clearly manifest themselves as discrepancies between estimated and actual running times.

Data dependencies. Some communication operations, permutations in particular, can take different amounts of time depending on the contents of the arrays passed as arguments. Let us take as an example two cyclic permutations of a one dimensional array `x` with $n = 16M$ double word elements on a 32-node CM-5. Cyclically shifting the array by 1 using a permutation operation `x = x(p)` takes 0.216s, while shifting it by $n/2$, such that every word is transferred through the root of the data network tree, takes 2.89s. Much of the difference is not due to the inability of the hardware to route the second permutation quickly, but due to software overheads; the array can be cyclically shifted by $n/2$ in 0.937s using the `cshift` subroutine.

While PERFSIM cannot predict the exact running time of data dependent operations, we do not believe this poses a significant problem to PERFSIM's users. Irregular, data dependent communication operations are used in two cases. The first case is when the program implements an algorithm in which irregular communication is used, such as computations on irregular meshes. In this case the communication operations cannot be removed or significantly changed by the programmer, and most of the tuning is done on the rest of the program, which PERFSIM estimates accurately. The second case is when the communication pattern is not data dependent, but the compiler failed to find a way to implement it using calls to communication subroutines for regular communication such as shifts. Here the programmer is better off modifying the program until the call to the irregular-communication subroutine is replaced by a call to a regular-communication subroutine, whose performance is always better, so the potential inaccuracy is not very significant.

9 Discussion

This paper describes PERFSIM, a performance analysis tool for data-parallel Fortran programs. We have demonstrated that the PERFSIM's modeling of both run-time subroutines and compiler generated code blocks is accurate enough to be useful. The accuracy is comparable to the consistency of running-time measurements on the CM-5. In test cases in which more than one version of a program was involved, PERFSIM always predicted correctly which version is more efficient. This behavior is probably the single most important feature of any performance monitor.

We believe that a tool such as PERFSIM ought to be supported by the vendor of the compiler and run-time system. While we have demonstrated that one can obtain accurate models without access to the source code of the subroutines, the source is essential when one subroutine uses different algorithms for different array layouts.

Performance modeling using analytic models is becoming an important paradigm for analyzing the performance of applications. PERFSIM's main contributions to this paradigm are the following.

- PERFSIM combines execution of the hard-to-analyze control thread with automatic analysis of vector operations. The fact that PERFSIM does not attempt to analyze the control thread keeps it simple. This approach is made possible by the separation of the control structure and the vector operations in the data-parallel programming model. Other performance analysis tools, especially those modeling Fortran 77 programs, are typically much more complex. Fahringer and Zima's PPPT [5] for example, which analyzes the entire program including its control structure, comprises of 25,000 lines of code, about ten times the number of source lines in PERFSIM.

- PERFSIM uses a robust performance model, based on the hardware architecture and the system software's performance. Much of the related research in performance modeling was done using simple machine models that proved sufficient on machines such as the Intel iPSC/860 [5] or the NCUBE [2]. Our experience shows that a rather complex machine model is necessary for accurately predicting program performance on current parallel computers.

- PERFSIM is an automatic profiling tool geared toward program tuning. While some researchers have used performance modeling for profiling (Atapattu and Gannon [1] developed an interactive performance prediction tool for the Alliant FX/8), most have used it to guide data layouts decisions (Balasundaram et al. [2], Fahringer and Zima [5]), and for automatic selection of subroutine out of a set of subroutines with the same functionality (e.g. sorting) [4]. Performance models have also been used for manual, rather than automatic, performance prediction [6, 8, 11].

Acknowledgments

Many people helped in this project, and I would like to thank them. Charles E. Leiserson suggested that I implement PERFSIM and offered many suggestions about its operation, its user interface, and about this paper. Charles and I analyzed the performance of one class of algorithms on the CM-5 using a spreadsheet program, and my experience from that analysis helped me a lot. Ken Jones, Thinking Machines Corporation's Applications Engineer at MIT, helped me realize the usefulness of PERFSIM. Tzu-Yi Chen helped conducting some of the experiments that preceded and led to the development of PERFSIM. Rich Shapiro of Thinking Machines helped me figure out the array sizes in nodal code blocks. Thanks to Richard Brewer of BU, Taras Ivanenko and Alan Edelman of MIT, and Danesh Tafti of NCSA for contributing code for me to analyze. Thanks to the anonymous referees for their helpful comments.

References

[1] D. Atapattu and D. Gannon. Building analytical models into an interactive performance prediction tool. In *Supercomputing '89*, pages 521–530, 1989.

[2] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide partitioning decisions. In *Proceedings of the 3rd ACM Symposium on Principles and Practice of Parallel Programming*, pages 213–223, 1991.

[3] G. E. Blelloch. *Scan primitives and parallel vector models*. PhD thesis, MIT, Oct. 1989.

[4] E. A. Brewer. Portable high-performance supercomputing: High-level architecture-dependent optimization. PhD thesis proposal, MIT, Jan. 1994.

[5] T. Fahringer and H. P. Zima. A static parameter based performance prediction tool for parallel programs. In *Proceedings of the 7th ACM International Conference on Supercomputing*, July 1993.

[6] A. R. Hainline, S. R. Thompson, and L. L. Halcomb. Vector performance estimation for CRAY X-MP/Y-MP supercomputers. *Journal of Supercomputing*, 6:49–70, 1992.

[7] C. E. Leiserson et al. The network architecture of the Connection Machine CM-5. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, 1992.

[8] C. E. Leiserson and S. Toledo. Performance estimates for iterative relaxation algorithms on an enhanced CM-5. Manuscript, Jan. 1993.

[9] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, Dec. 1992.

[10] Thinking Machines Corporation, Cambridge, MA. *The Connection Machine CM-5 Technical Summary*, Jan. 1992.

[11] S. R. Thompson, A. R. Hainline, and L. L. Halcomb. Vector performance estimation for CRAY X-MP/Y-MP supercomputers, part 2. *Journal of Supercomputing*, 7:437–467, 1993.