Performance Prediction with Benchmaps*

Sivan Toledo
IBM T.J. Watson Research Center

Abstract

Benchmapping is a performance prediction method for data-parallel programs that is based on modeling the performance of runtime systems. This paper describes a benchmapping system, called BENCHCVL*, that predicts the running time of data-parallel programs written in the NESL language on several computer systems.* BENCHCVL *predicts performance using a set of more than 200 parameterized models. The models quantify the cost of moving data between processors, as well as the cost of moving data within the local memory hierarchy of each processor. The parameters for the models are automatically estimated from measurements of the execution times of runtime system calls on each computer system.*

1 Introduction

Benchmapping is a performance prediction method for data-parallel programs that is based on the following paradigm. The runtime system of a data-parallel programming language is modified so that it can automatically predict the running time of every runtime system call. When a program that uses the runtime systems runs, the modified runtime system generates a running-time profile of the execution. The performance predictions that the runtime system generates are based on a detailed model, called a **benchmap**, of the performance of a given computer system. The running-time profile is then viewed by the programmer using an interactive performance visualization tool.

The main benefit of the benchmapping method is that the benchmap can model the performance of a computer system other than the one the program is running on. For example, when the user runs a program on a workstation, the runtime system can generate the performance profile of the program on a parallel computer if the runtime system has a benchmap of the parallel computer.

A previous paper [14] described a benchmapping system called PERFSIM that demonstrated the feasibility and usefulness of the method. PERFSIM used a benchmap of the Connection Machine CM-5 to predict the running time of CM-Fortran programs. PERFSIM was able to predict the performance of compiler-generated subroutines as well as of runtime system subroutines. The main issues that PERFSIM addressed were accuracy and usability. Accuracy was achieved by using a benchmap that modeled the hardware of the CM-5 in considerable detail. The system was useful because it predicted the performance of some programs on a workstation without any programmer intervention, and it could predict the performance of many other programs after the progmmer inserted a few annotations into the program.

This paper addresses two more issues in the benchmapping method. First, the paper shows that one parameterized benchmap can model the performance of several computer systems. The parameters are different for each computer system, but the basic structure of the benchmap is the same for all of them. Second, the paper shows that the parameters in the benchmap of a computer system can be automatically estimated by a program. This program measures the execution time of calls to the runtime system and uses the running times of these experiments to estimate the benchmap's parameters.

Addressing these issues requires a data-parallel programming system that has been ported to several computer systems. We chose NESL [5], an experimental data-parallel programming language developed at CMU. The NESL programming system uses a runtime system called CVL [6] to execute NESL programs. CVL has been ported to several computers. Our performance prediction system, called BENCHCVL, predicts the performance of NESL programs on several computer systems by modeling the performance of CVL on each of them.

Several related systems have been described in the literature. They are briefly described in the next paragraph. All of them support the claim that performance prediction is feasible and useful. We present two contributions to this discourse. First, we provide evidence that performance prediction is feasible in relatively realistic settings. For example, we show that it is possible to model data caches using non-linear models and that it is possible to automatically estimate parameters for benchmaps that include more than 200 individual models of runtime system subroutines. In addition, we showed in a previous paper that it is possible to create models for a production runtime system without access to its source code. Our second contribution is the benchmapping method itself, which we believe provides several advantages over other performance prediction methods. A benchmapping system allows users to

*This research was performed while Sivan Toledo was a graduate student at the MIT Laboratory for Computer Science, and was supported in part by ARPA under Grant N00014–94–1–0985.

predict the performance of a program on a given machine without requiring them to run the program on that machine even once. Benchmapping systems do not require a static analysis of the program because the control thread of the program is executed when its performance is being predicted.

Brewer [7, 8] demonstrated that a single set of parameterized models can predict the performance of the services of a small runtime-system that he wrote and that runs on multiple architectures. The number models in his system, however, is quite small, and they do not model data caches. Balasundaram et al. [3] propose a performance modeling system designed to guide compiler optimizations. Fahringer and Zima [11] describe a more comprehensive system for guiding compiler optimization, which is a part of the Vienna Fortran Compilation System. They use a simple model with a few parameters to describe a computer system and claim that the predictions are accurate. Atapattu and Gannon [2] describe an interactive performance prediction tool for single-threaded Fortran programs running on a bus-based shared memory parallel computer. Performance prediction in their tool is based on a static analysis of the assembly language code of the compiled program. Crovella and LeBlanc [10] describe an interactive performance tuning tool that tries to fit the behavior of the program to a model taken from a library of performance models. Their tool fits a model to the performance of the program based on several executions of the program. To use their tool, a user must run the program several times on the target machine. To use PERFSIM or BENCHCVL the user does not have to run the program on the target architecture at all. It is necessary to run experiments on the target architecture in order to estimate parameters in the benchmap, but this is not done by the user of the benchmapping system. Some researchers have tried to predict the running time of a program based on source-code constructs rather than on constructs in the compiled program [4, 13]. In both of these papers the prediction was done by hand. In all the other cases, including ours, a software system predicts performance automatically.

The rest of the paper is organized as follows. Section 2 describes the models that BENCHCVL uses. Section 3 explains how our system estimates parameters in benchmaps. Section 4 examines the accuracy of BENCHCVL. We conclude the paper with a discussion of our results in Section 5.

2 Portable Benchmaps in BENCHCVL

This section describes the models that BENCHCVL uses to predict the running time of calls to the CVL runtime system. All the models are based on an abstract models of a parallel computer. Each model in the benchmap maps a small set of quantities that we shall call **performance determinants**, such as the length of a vector, to a **performance measure**, for example, the running time of a vector summation subroutine.

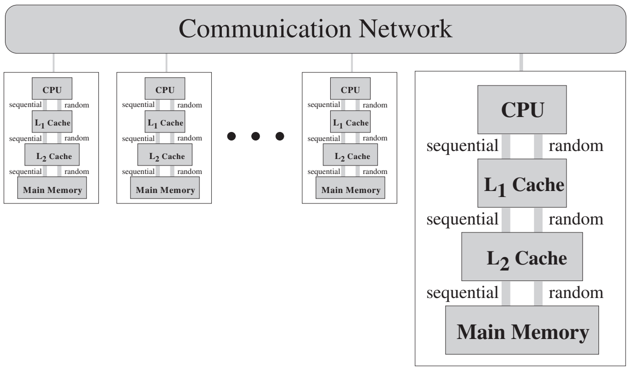

Figure 1 A schematic diagram of the model that underlies BENCHCVL's benchmap. The model describes the computer as a collection of nodes connected by a communication channel, where each node contains a processing unit and up to three levels of local memory. The benchmap describes the cost of interprocessor communication across the communication network, the sizes of caches, and the cost of random and sequential data transfers between levels of the local memory hierarchy.

We begin the discussion with a brief description of CVL, and then turn to a description of the models themselves.

The CVL runtime library [6], which is the runtime system for NESL programs, implements operations on entire and segmented one-dimensional vectors. A segmented vector is partitioned into segments of arbitrary lengths. Vector operations in CVL include element-wise operations, such as adding two vectors, scans, or parallel prefix operations, reductions of vectors to scalars, such as summations, vector permutations, ranking (sorting) vectors, and packing of sparse vectors. On parallel computers, every processor "owns" a section of each vector and is responsible for operating on that section.

The performance determinants that BENCHCVL uses to predict the running time of CVL subroutines are the number of elements per processor in each argument vector and the number of segments in argument vectors. The content of vectors is not used to predict performance. The length of individual segments is not used either, because the number of segments can be arbitrarily large. Using the contents of vectors could degenerate benchmapping into a simulation and slow down the prediction process.

BENCHCVL's benchmap is based on a model of parallel computers in which a collection of nodes are connected by a communication channel. Each node has a processing unit and a local memory hierarchy, as shown in Figure 1. The benchmap describes the cost of interprocessor communication across the communication network, the sizes of caches, and the cost of random and sequential data transfers between levels in the local memory hierarchy. Sequential access to data creates better spatial locality of reference than random access and therefore the performance of sequential accesses is often better than that of random accesses.

As in most performance models, the basic building block of BENCHCVL's benchmap is the linear model. A linear model is an affine function that maps a set of **basis functions**, such as the length of an input vector, to a **performance measure**, for example, the running time of the subroutine. The basis functions can be performance determinants or functions of performance determinants. Consider the running time of a subroutine for sorting a vector of length n, for example. If the running time of the subroutine is proportional to n, a linear model with a single basis function n can predict the its running time. If, on the other hand, the growth of the running time is proportional to $n \log n$, then the model must use $n \log n$ as the basis function. The performance determinant in both cases is n, but the basis functions are different.

BENCHCVL's models include basis functions that represent five cost categories:

- A fixed cost that represents the subroutine call overhead.
- A cost proportional to the diameter of the interprocessor communication network. Currently, all the models use a $\log(P)$ term to represent this cost, where P is the number of processors.
- Costs for sequentially operating on the section of arrays owned by one processor.
- Costs for random accesses to sections of arrays owned by one processor.
- Costs for interprocessor transfers of array elements.

We distinguish between sequential access and random access to vectors because when vectors do not fit in data caches, sequential access cause a cache miss at most once every cache line size (ignoring conflicts), whereas random accesses can generate a miss on almost every access.

The model for permuting a non-segmented vector of size N into another vector of size N on a computer with P processors, for example, has the following structure:

$$x_1 + x_2 \log(P) + x_3 \frac{N}{P} + x_4 \frac{N}{P} \frac{P-1}{P} \ .$$

The first term represents the fixed cost, the second the latency, which is proportional to the diameter of the communication network, the third the number of elements that each processor owns, and the last the expected number of elements each processor must send and receive from other processors. The number of messages used by the model is an approximation for the expected number in a random permutation.

BENCHCVL models both the temporal locality and spatial locality in data accesses. The spatial locality in vector operations is modeled by using separate terms for for random accesses and for sequential accesses, where spatial locality is guaranteed. The cost of random accesses is represented by two basis functions, one that represents the cost of a cache miss times the probability of a miss, and another that represents the cost of a cache hit times the probability of a cache hit. The probability of a cache hit depends on the size of the cache relative to the size of the vector owned by a processor (assuming a cache for every processor). For example, the model for permuting the elements of a vector of size N_1 into a vector of size N_2 includes the basis functions

$$\frac{N_1}{P} \frac{1}{N_2/P}$$

and

$$\frac{N_1}{P} \left(1 - \frac{1}{N_2/P} \right) \ .$$

The number of random accesses in the operation is N_1/P. The probability of a cache hit in one of those accesses, when N_2/P is larger than the size α of the cache, is approximated by $\alpha/(N_2/P)$, the fraction of the target vector's elements that can reside in the cache. The parameter α is left unspecified in the basis function, and is estimated by the parameter estimation algorithm.

BENCHCVL's benchmap models temporal locality in data caches using piecewise-linear models. A **piecewise-linear model** decomposes the space of performance determinants, such as array sizes, into regions and predicts performance in each region using a linear model. Consider, for example, the running time of a subroutine that computes the inner product of two vectors. A piecewise-linear model can decompose the space of vector sizes into a region of small sizes, where the input vectors fit within the cache, and a region of large vector sizes, where the input vectors must be stored in main memory.

BENCHCVL assumes that vectors that fit within the cache are indeed in the cache. A piecewise-linear model uses separate linear models to describe the performance of a subroutine when argument vectors fit within the cache and when they do not. BENCHCVL does not specify the size of caches. Rather, an automatic parameter estimation algorithm estimates the size of the cache based on apparent "knees" in the running time, which we call **breakpoints**. Figure 2 shows two models with breakpoints generated by BENCHCVL's parameter estimation module.

Architectures with more than one level of caches are modeled with piecewise-linear models with more than one breakpoint and more than two regions. In such cases, the benchmapper specifies the ratios between the sizes of caches in the system. The specification of these ratios enables BENCHCVL to search for only one breakpoint in the running time, and ensures a robust estimation of cache sizes.

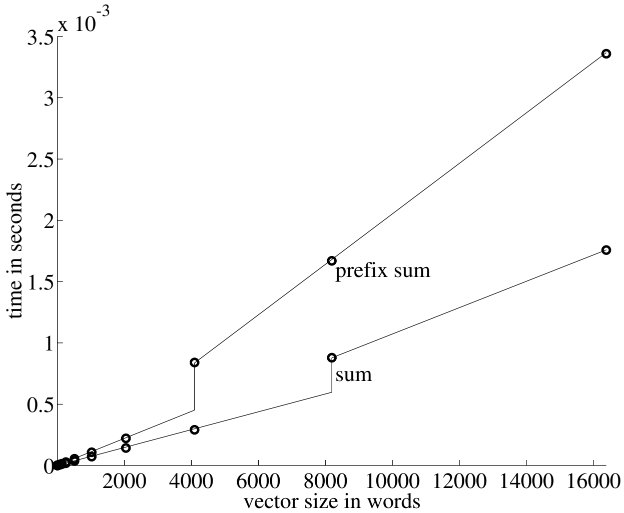

Figure 2 The execution times of a summation function and a prefix sum function, represented by circles, on a Sun SPARCstation 10 workstation. The lines represent the models estimated from these data points. The running time of the two functions is clearly not linear, and the breakpoints occur on different input sizes, because one function handles only one vector, while the other handles two.

3 Estimating the Parameters of A Benchmap

The models in a benchmap are parameterized. The parameters are chosen so as to minimize the estimation error in some norm. These parameters can be estimated automatically from the results of experiments using a variety of algorithms. We have written a program called CARTOGRAPHER, which is now in its second version, that conducts experiments and estimates parameters in a benchmap.

Although most of the parameter estimation techniques that CARTOGRAPHER uses are well known, some of them deserve special attention. We have already mentioned the capability to estimate parameters in a piecewise-linear model that predicts the effects of data caches. Other interesting techniques include the use of the singular value decomposition (SVD), the minimization of the maximum error, and the creation of models that bound performance from above or below. This section presents an overview of these techniques. For a complete description of the techniques, see [15].

Linear models that predict performance on multiple architectures must have terms that correspond to factors that determine performance on any of the architectures. For example, a model might have a term that describes the cost of sending messages on parallel computers. On uniprocessor workstations no messages are sent, so the term is zero in all the experiments that are used to estimate parameters. The zero terms cause a phenomenon known as **rank deficiency**, which creates numerical problems in some popular least-squares algorithms. We cannot eliminate this term from the models since it is required for other architectures. Instead, we use the SVD algorithm for linear least-squares, which can cope with rank deficiency.

Several norms can be used to assess the choice of parameters in a model. One popular norm is the least-squares norm, or the sum of the squares of the errors (in performance modeling the relative error is used, rather than the absolute error). In performance modeling, minimization of least squares sometimes leads to poor models, because it can practically disregard a few experiments. When there are no outliers, that is, when all the experiment are valid, minimizing the maximum relative error is a better strategy. This optimization problem is a linear programming problem, and algorithms for its solution are readily available.

No model can always predict performance exactly. Therefore, the prediction should consist of a range of possible outcomes. The standard estimates of confidence intervals are based on several statistical assumptions, most of which are often invalid in performance modeling, such as normality and independence. (For quantitative data supporting this claim, see [15].) Instead, we use models that bound performance from above and below. From a given set of experiments, CARTOGRAPHER can choose parameters for a linear model that minimize the norm of the error under the constraint that the resulting model under- or over-estimates all the experiments. If all the data collected represent actual performance, this procedure yields two models that is likely to bound the performance of future runs. The mathematical formulation of these parameter estimation criteria leads to quadratic or linear programming problems, depending on the error norm chosen. Again, readily available optimization packages can solve these problems.

4 BENCHCVL's Accuracy

This section exhibits, by examining BENCHCVL's accuracy, the feasibility of automatic performance modeling on multiple architectures. Since the same models are supposed to predict performance on multiple architectures, the discussion focuses on the structure of models on different architectures.

BENCHCVL's models were developed on a Sun SPARCstation 10. Subsequently, four computer systems were automatically surveyed and modeled with CARTOGRAPHER: a Sun SPARCstation 10, a Sun SPARCstation 1+, a 32-node Connection Machine CM-5, and a Cray C90 (CVL uses only one processor on Cray vector computers). A test suite composed of the larger programs in the NESL distribution plus one other large program was used to evaluate BENCHCVL. Table 1 show the actual and predicted running times in seconds of the programs in the test suite. Reported running times are averages of at least 3 executions. Bugs in the two CM-5 implementations of CVL prevent some of the test program from running. A bug in the CVL implementation and a bug in BENCHCVL prevented some of the test programs from running on the C90.

The tables show that BENCHCVL is accurate. The relative

Program	SPARCstation 10			SPARCstation 1+			CM-5			Cray C90		
	Actual Time		Pred.	Actual Time		Pred.	Actual Time		Pred.	Actual Time		Pred.
	Total	CVL	CVL	Total	CVL	CVL	Total	CVL	CVL	Total	CVL	CVL
Geometric Separator	5.49	1.67	1.42	15.88	5.52	6.71	N/A	N/A	N/A	N/A	N/A	N/A
Spectral Separator	74.68	64.50	76.56	268.06	270.81	361.07	28.93	19.92	17.03	N/A	N/A	N/A
Convex Hull	9.06	6.90	8.48	35.42	29.12	37.19	N/A	N/A	N/A	0.703	0.134	0.109
Conjugate Gradient	88.07	83.41	110.81	339.50	326.20	451.98	15.09	6.07	7.75	5.38	2.15	1.86
Barnes Hut	269.48	175.65	193.59	917.96	652.08	655.05	N/A	N/A	N/A	N/A	N/A	N/A

Table 1 Measured and predicted running times of NESL programs on four different computer systems. The NESL programming system executes programs by interpreting an intermediate representation of the program. The total time reported includes both the time spent in CVL subroutines and the time spent by the interpreter. BENCHCVL only models the time spent in CVL subroutines. We were unable to run some of the experiments that are marked N/A due to technical difficulties.

	Conjugate Gradient	Spectral Separator
Sun 10/CM-5 (Actual)	14	3.2
Sun 10/CM-5 (Predicted)	14	4.4
Sun 1+/CM-5 (Actual)	54	13
Sun 1+/CM-5 (Predicted)	58	21

Table 2 The actual and predicted speedups between Sun workstations and a CM-5, on two different programs. The ratios represent the running time on a workstation divided by the running time on the CM-5.

errors are 33% or better, except for one experiment in which the error is 39%. BENCHCVL's accuracy enables meaningful comparisons between computer systems. Table 2 shows that the predictions can be effectively used to compare the performance of computer systems on specific programs. Such comparisons are more meaningful to users of the programs being compared than comparisons based on the performance of benchmark programs.

The relative errors of the models on the experiments that were used to estimate parameters are generally small. On a Sun SPARCstation 10, the errors in most of the elementwise vector operations are 5% and smaller, with a few exceptions where the errors are up to 8%. The errors in reductions and scans (including segmented operations) are 7% and smaller, except for segmented operations involving integer multiplication and bitwise `and`, where the errors are up to 20%. The reason for the larger errors in these operations is that the running time of individual operations is value dependent, and therefore cannot be accurately modeled by value independent models. This is a good example of the importance of upper and lower bounds rather than a single model. The errors on data movement operations, such as permutations, gathers, and scatters were below 37%, with many of the operations being modeled to within 15% or less.

Results on the CM-5 were similar, except for large relative errors in the subroutines that transform ordinary C arrays to and from distributed vectors. This uncovered a latent "bug" in the models' structure: the models lacked a term to account for a large sequential bottleneck on the processor which owns the C array. This problem is easily fixed, but it may be typical of latent deficiencies in models, which are discovered only when a model fails on a certain architecture.

Relative errors on the C90 were larger than on Sun workstations and the CM-5. The most likely reason for the larger errors is that the C90 timers are oblivious to time sharing, so some of the timings probably include some outliers that include other users' time slices. The problem can be fixed by taking the minimum of several measurements as the actual time, rather than the average of the measurements.

Modeling the performance of CVL on a Silicon Graphics Indigo2 workstation revealed that cache conflicts make performance virtually unpredictable. BENCHCVL revealed that conflicts in the two levels of direct mapped caches degrade performance on certain vector sizes. Conflicts in the onchip virtually indexed cache degrade performance of operations such as vector copy by a factor of about 1.75, and conflicts in the offchip physically indexed cache degrade performance by a factor of about 14. Since BENCHCVL does not model memory-system conflicts, the models cannot predict performance on this machine with any degree of accuracy. Modeling conflicts in the physical address space is particularly difficult, because only the virtual addresses of vectors are known to the runtime system. (We have found however that conflicts of virtual addresses usually translates to conflicts of physical addresses on this machine, so conflicts in the virtual address space cannot be ignored.)

BENCHCVL performed its job: it indicated that performance on this workstation is not predictable to within less than a factor of about 15, at least for NESL programs (we have duplicated this behavior in simple C programs as well). This is a valid input for decision makers who must assess the expected performance of machines before they are purchased.

5 Conclusions

Our experience, as well as other research cited in the Introduction, shows that performance prediction is both feasible and important for software development on parallel systems. What sets our work apart is the recognition that complex systems must be described by complex models (but most of the complexity is represented by parameters that can be automatically estimated) and in the insight that modeling the performance of data-parallel runtime systems allows for automatic, accurate, and very fast prediction of the performance of entire programs.

Limiting the scope of the benchmapping systems to data-parallel programs allows us to achieve accuracy and speed while still encompassing a large number of scientific codes. There is plenty of evidence that the data-parallel programming model is suitable for many scientific applications. Data-parallel programming includes not only traditional data-parallel languages such as High Performance Fortran [12], but also sequential and parallel programs that extensively use numerical libraries such as the BLAS, LAPACK [1], and ScaLAPACK [9]. Benchmaps that model the performance of these numerical libraries should enable accurate prediction of the running time of such programs.

Neither PERFSIM nor BENCHCVL predicts performance exactly, but both are accurate enough to be used for program tuning. Some papers in the literature suggest that simple models are enough to accurately predict performance on current computer systems. We believe that such claims are overly optimistic and result from too little testing of the models with large-scale programs.

Consequently, we believe that the most important open question in performance prediction today is how to assess and verify the accuracy of performance models. Without the means to assure the accuracy of models it is difficult to put them in production use.

Acknowledgments

Thanks to Charles E. Leiserson for advising this work. Thanks to Guy Blelloch and Jonathan Hardwick for assistance with NESL and CVL. Thanks to the anonymous referees for several helpful comments.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. SIAM, Philadelphia, PA, 2nd edition, 1994.

[2] Daya Atapattu and Dennis Gannon. Building analytical models into an interactive performance prediction tool. In *Proceedings of Supercomputing '89*, pages 521–530, 1989.

[3] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A static performance estimator to guide partitioning decisions. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 213–223, 1991.

[4] John Louis Bentley. *Writing Efficient Programs*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[5] Guy E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[6] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Margaret Reid-Miller, Jay Sipelstein, and Marco Zagha. CVL: A C vector library. Technical Report CMU-CS-93-114, School of Computer Science, Carnegie Mellon University, February 1993.

[7] Eric A. Brewer. *Portable High-Performance Superconducting: High-Level Platform-Dependent Optimization*. PhD thesis, Massachusetts Institute of Technology, 1994.

[8] Eric A. Brewer. High-level optimization via automated statistical modeling. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 80–91, 1995.

[9] J. Choi, J. Dongarra, R. Pozo, and D. Walker. ScaLAPACK: A scalable linear algebra for distributed memory concurrent computers. In *Proceedings of the 4th Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127, 1992. Also available as University of Tennessee Technical Report CS-92-181.

[10] Mark E. Crovella and Thomas J. LeBlanc. Parallel performance prediction using lost cycles analysis. In *Proceedings of Supercomputing '94*, pages 600–609, Washington, D.C., November 1994.

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

[12] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA, 1994.

[13] Neil B. MacDonald. Predicting execution times of sequential scientific kernels. In Christoph W. Kessler, editor, *Automatic Parallelization*, pages 32–44. Vieweg, 1994.

[14] Sivan Toledo. PerfSim: A tool for automatic performance analysis of data parallel Fortran programs. In *5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, February 1995.

[15] Sivan A. Toledo. *Quantitative Performance Modeling of Scientific Computations and Creating Locality in Numerical Algorithms*. PhD thesis, Massachusetts Institute of Technology, 1995. Also available as Technical Report MIT-LCS-TR-656.