Quantitative Performance Modeling of Scientific Computations and Creating Locality in Numerical Algorithms

by

Sivan Avraham Toledo

B.Sc., Tel-Aviv University (1991)
M.Sc., Tel-Aviv University (1991)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1995

Author . . . .
Department of Electrical Engineering and Computer Science
May 25, 1995

Certified by . . . .
Charles E. Leiserson
Professor
Thesis Supervisor

Accepted by . . . .
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Quantitative Performance Modeling of Scientific Computations
and
Creating Locality in Numerical Algorithms

by

Sivan Avraham Toledo

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 1995, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

How do you determine the running time of a program without actually running it? How do you design an efficient out-of-core iterative algorithm? These are the two questions answered in this thesis.

The first part of the thesis demonstrates that the performance of programs can be predicted accurately, automatically, and rapidly using a method called benchmapping. The key aspects benchmapping are: automatic creation of detailed performance models, prediction of the performance of runtime system calls using these models, and automatic decomposition of a data-parallel program into a sequence of runtime system calls. The feasibility and utility of benchmapping are established using two performance-prediction systems called PERFSIM and BENCHCVL. Empirical studies show that PERFSIM's relative prediction errors are within 21% and that BENCHCVL's relative prediction errors are almost always within 33%.

The second part of the thesis presents methods for creating locality in numerical algorithms. Designers of computers, compilers, and runtime systems strive to create designs that exploit the temporal locality of reference found in some programs. Unfortunately, many iterative numerical algorithms lack temporal locality. Executions of such algorithms on current high-performance computers are characterized by saturation of some communication channel (such as a bus or an I/O channel) whereas the CPU is idle most of the time.

The thesis demonstrates that a new method for creating locality, called the blocking covers method, can improve the performance of iterative algorithms including multigrid, conjugate gradient, and implicit time stepping. The thesis proves that the method reduces the amount of input-output operations in these algorithms and demonstrates that the method reduces the solution time on workstations by up to a factor of 5.

The thesis also describes a parallel linear equation solver which is based on a method called local densification. The method increases the amount of dependencies that can be handled by individual processors but not the amount of dependencies that generate interprocessor communication. An implementation of the resulting algorithm is up to 2.5 times faster than conventional algorithms.

Thesis Supervisor: Charles E. Leiserson
Title: Professor

לזכר אבי, ג׳קי טולדו, 1936 - 1981.

To the memory of my father, Jacky Toledo, 1936–1981.

Acknowledgments

I had the privilege of being a student of Charles E. Leiserson during my graduate career at MIT. Charles taught me a great deal about computer science, about research, about teaching and advising, and about life in general. I greatly appreciate his time, efforts, and his advice. His help was particularly important during the research on benchmaps.

I thank Charles Leiserson and Satish Rao for allowing me to include the results of our joint research on blocking covers in Chapters 7 and 8.

I thank Alan Edelman and Jacob K. White, the readers of the thesis, for their help with my thesis. I learned a great deal about numerical algorithms from both of them, especially by attending courses they gave.

Tzu-Yi Chen, an undergraduate student, worked with me for the last two years as a participant in the Undergraduate Research Opportunities Program at MIT. She helped conduct performance studies on the CM-5, and did most of the design and implementation of the out-of-core Krylov-subspace methods.

Ken Jones, formerly Thinking Machines' application engineer at MIT, offered tremendous amounts of enthusiasm and support during my work on PerfSim. Rich Shapiro of Thinking Machines helped me solve a few problems in the implementation of PerfSim. Jonathan Hardwick from the SCANDAL group at CMU answered many questions about NESL and CVL, and fixed bugs when I ran into them.

Thanks to Scott Blomquist, our system administrator, for keeping the machines and software I used up and running.

I thank all those who gave me programs to use as test suites for both PerfSim and BenchCvl, Richard Brewer of BU, Alan Edelman of MIT, Taras Ivanenko of MIT, Girija Narlikar of CMU, Danesh Tafti of NCSA, and Dafna Talmor of CMU.

My work at the Lab for Computer Science has been an enjoyable experience thanks to the companionship of many, in particular my office mate Kavita Bala with whom I spent many hours in stimulating discussions.

Finally, I would like to thank my wife Orith for many enlightening discussions, for reading numerous drafts of papers and thesis chapters, and for being a wonderful friend during these years.

My research was supported in part by the Advanced Research Projects Agency under Grants N00014–91–J–1698 and N00014–94–1–0985.

Contents

Chapter 1

Introduction

How do you determine the running time of a program without actually running it? How do you design an efficient out-of-core iterative algorithm? These are the two questions answered in this thesis. The first part of the thesis presents a solution to the first question. The second part of the thesis presents techniques that create locality of reference in iterative numerical algorithms and thus improve their performance.

1.1 Performance Prediction

There are numerous situations in which is it necessary or desirable to know the performance of a program on a given computer system without actually running and timing it. Computer acquisition and configuration decisions, for example, are often based on performance predictions. Estimating the performance of a program on a future architecture can provide feedback during computer design. Optimizing compilers can use estimates of the performance of several ways of compiling a program to select the best one. Unfortunately, traditional methods for predicting performance are inadequate for many modern computer architectures.

Performance prediction based on a description of the characteristics of the hardware fail because of two reasons. First, most hardware descriptions describe complex computer systems with just a few numbers: number of floating-point operations per second, memory bandwidth, etc. Such descriptions leave important aspects of the system unspecified and therefore, accurate performance prediction is often impossible. Second, predicting performance with hardware descriptions is a manual process. Many application of performance prediction require or can benefit from automatic performance prediction.

Performance prediction based on the results of standardized benchmarks suffers from the same problems. Even large benchmark suites do not contain enough programs to capture all the complexities of current computer systems. Predicting performance of a specific program based on benchmark results requires comparing the program to the programs in the benchmark suite. The comparison and subsequent performance prediction is a manual and often subjective process. In addition, many benchmarks do not scale up the problems they solve to match the increasing performance of computer systems. Consequently, benchmarks often fail to exercise important components of the system. For example, the code and data of some popular benchmarks fits within the cache of current microprocessors, and thus, these benchmarks cannot reliably describe the performance of the main memory system.

Simulators of computer systems overcome the accuracy and automation problems associated with hardware characteristics, but they are too slow for many applications. Predicting the performance of a program with a simulator is often several orders of magnitude slower than running the program. Such speed is unacceptable for applications such as program optimization.

The first part of the thesis presents a performance-prediction method, called benchmapping, which is accurate, automated, and fast. Accuracy is achieved by modeling computer systems with great detail. The modeling software learns most of the details automatically from experiments performed on the system. Automation is achieved using software that decomposes programs into components whose performance can be accurately predicted. The software then composes the performance of the entire program from the performance of its components. Speed is achieved by modeling the performance of a high-level interface, namely data-parallel runtime systems. Modeling a high-level runtime system limits the overhead of performance prediction, because the number of program components whose performance must be modeled is small. In addition, it is often possible to predict the performance of programs without executing the runtime system calls. This strategy speeds up performance prediction even further, making performance prediction faster than program execution.

Automation and and speed render benchmaps suitable for predicting the performance of programs in applications such as program tuning, acquisition and configuration decision making, performance maintenance, hardware design, and compile-time and runtime optimization.

Part I of this thesis, which encompasses Chapters 2 to 6, presents the benchmapping methodology and its applications. Chapter 2 describes the challenges of performance prediction and how benchmapping meets these challenges. Benchmapping uses performance models that are called benchmaps. Chapter 3 describes how benchmaps are created. Chapter 4 describes two benchmapping tools called PERFSIM and BENCHCVL. Chapter 5 focuses on accuracy evaluation of benchmaps and shows that the benchmaps that are used in this thesis meet reasonable accuracy expectations. Part I of the thesis concludes with a discussion of the applications of benchmapping and of the reasons benchmapping is superior to traditional performance prediction methodologies.

1.2 Creating Locality in Numerical Algorithms

Designers of computers, compilers, and runtime systems strive to create designs that exploit the temporal locality of reference found in programs. Unfortunately, many iterative numerical algorithms lack temporal locality. Executions of such algorithms on current high-performance computers are characterized by saturation of some communication channel (such as a bus or I/O channel) whereas the CPU is idle most of the time.

The main contribution of Part II of the thesis, which encompasses Chapters 7 to 11, is a collection of the first efficient out-of-core implementations for a variety of iterative algorithms. Chapter 7 surveys the methods that are used in Part II to create temporal locality of reference. The chapter explains the essence of each of the techniques, and shows that one well-known method, called the covering method, cannot be applied to most of today's sophisticated iterative methods. Out-of-core numerical methods have a long and fascinating history, which is also surveyed in Chapter 7.

Chapters 8 to 10 describe out-of-core iterative numerical algorithms. Chapter 8 presents out-of-core linear relaxation algorithms for multigrid-type matrices. (Given a representation of a matrix A, a vector x, and an integer T, linear relaxation algorithms compute $A^T x$.) Linear relaxation algorithms with multigrid-type matrices are used in implicit time-stepping simulations in which a multigrid algorithm is used as the implicit solver, as well as in Krylov-subspace solutions of integral equations where a fast-multipole-method algorithm is used as a matrix-vector multiplication subroutine. Chapter 9 presents out-of-core linear relaxation algorithms for matrices that arise from implicit time-stepping simulations in one spatial dimension. Chapter 10 presents a method for implementing out-of-core Krylov-subspace algorithms, such as conjugate gradient.

Part II also presents a method for increasing the locality of reference and hence the performance of parallel preconditioners. Chapter 11 describes a a novel ordering scheme for two-dimensional meshes. The ordering scheme is used in a preconditioner that accelerates the solutions of certain important systems of linear equations on the Connection Machine CM-5 by a factor of 1.5 to 2.5.

Part I

Quantitative Performance Modeling of Scientific Computations

Chapter 2

Performance Modeling and Prediction

2.1 The Challenges

The first part of the thesis describes a methodology for determining the performance of a program automatically, accurately, and quickly without actually running the program. The feasibility and utility of this methodology, called **benchmapping**, are demonstrated in Chapters 4 and 5 using two benchmapping systems called PERFSIM and BENCHCVL. PERFSIM is a profiler for data-parallel Fortran programs. It runs on a workstation and produces the profile of the execution of a program on the Connection Machine CM-5 [110] quicker than the profile can be produced by running the program on a CM-5. BENCHCVL predicts the running time of data-parallel programs written in the NESL language [17] on several computer systems. Applications of benchmapping, including program profiling and tuning, making acquisition and configuration decisions, performance maintenance, hardware design, and compile-time and runtime optimization, are described in Section 6.1.

Performance prediction is a challenging problem. Essentially, any performance-prediction strategy decomposes a program into components, predicts the performance of components using a specification of the performance of the computer system, and then composes the performance of the entire program from the predicted performance of its components. (Section 2.2 defines precisely what we mean by performance and performance prediction.) Such strategies must meet three challenges:

1. Choosing the level at which the program is decomposed. We call the family of operations into which a program is decomposed a **decomposition interface**. Source statements form one decomposition interface, and machine instructions form another, for example. The goal of the decomposition is to decompose the program into a small number of components whose performance can be predicted.

2. Generating the sequence of operations that the program performs when executed. In other words, it is the dynamic behavior of the program that is decomposed, not the static structure of the program.

3. Creating a performance specification of the computer system. The specification must enable accurate and reliable prediction of the performance of program components.

The traditional forms of performance specification and prediction fail to meet these challenges. Traditionally, performance is specified with hardware descriptions, benchmark suites, and simulators. Hardware descriptions specify some of the characteristics of the hardware of the computer system. Popular characteristics include the number of floating-point operations per second, the number of processors and registers, the bandwidth of the memory system and the bandwidth of communication networks. Benchmark suites, typically consisting of between one and about fifteen standard programs, specify the measured performance of the programs in the suite. Simulators specify the performance of computer systems by implementing the systems in software that can run on other machines. Simulators are often instrumented in a way that allows them to accurately predict performance while they run a program.

Several factors contribute to the failure of traditional performance specification methodologies to meet the challenges of performance prediction. Both hardware descriptions and benchmarks specify the performance a complex system with just a few numbers. The limited amount of data is likely to leave important aspects of the computer system unspecified, and thus to render accurate performance prediction impossible. There is no standard way to decompose a program into components whose performance can be predicted by hardware descriptions or by benchmarks. Consequently, performance prediction with both hardware descriptions and with benchmarks is done manually, and it is difficult even for experts. Finally, simulators decompose programs into a large number of components, typically individual machine instructions. Consequently, even though performance prediction with simulators can be both automated and accurate, it is also slow. For many applications, simulators are simply too slow. Section 6.2 describes these issues in more detail.

This thesis demonstrates that a new methodology called **benchmapping** can meet the challenges of performance prediction and produce systems that predict performance automatically, accurately, and quickly. The three challenges are met in the following ways.

1. The program is decomposed into a sequence of data-parallel runtime system calls. Section 2.3 explains why I have chosen to decompose programs into a sequence of runtime-system calls and why runtime systems that support the data-parallel programming model are particularly well suited for benchmapping.

2. The decomposition is generated by by executing the control structure of the program. I have developed a mechanism, called **multiplexors**, that allows the control thread to execute even if runtime-system calls are not serviced. This strategy eliminates the need for complex static analysis of the program, while still allowing performance to be predicted more rapidly than the program can be executed. Section 2.4 describes this design decision.

3. Performance models of the subroutines in the runtime system enable automatic prediction of the performance of program components. We shall call a collection of performance models for the subroutines in a runtime system a **benchmap**. Benchmaps are detailed enough to capture the complexities of current computer systems. Most of the details, however, are gathered automatically from experiments performed on the system. Section 2.5 explains how benchmaps specify performance and how they are created.

2.2 Performance and Performance Prediction

This section explains what performance models predict and on what information they based their predictions. The quantities that performance models predict are called performance measures, and the quantities on which the predictions are based are called performance determinants. The section begins with a discussion of the difficulties of defining and measuring performance. We then discuss issues related to performance determinants, and in particular how to predict performance when the value of some determinants cannot be ascertained.

Performance Measures

Performance measures are quantities associated with an execution of a program on some computer. Performance measures include CPU time, maximum and average response time to external events, maximum memory allocated, amount of input/output activities, etc. We distinguish between **internal** and **external** performance measures. An internal performance measure quantifies the resources used by the program, such as CPU time. An external performance measure quantifies aspects of the interaction of the program with its environment, such as the maximum and average response time to external events. Since this thesis focuses on scientific applications, many of which are noninteractive, we focus on internal performance measures, and in particular, on running time.

Defining performance measures precisely is difficult, but measuring these quantities is even harder. Let us consider an example. If a program uses large data structures that cause the process running the program to page fault, we would like to charge the process for the context-switch time caused by the faults, and perhaps also for the burst of cache misses that are likely to follow the context switch. But if a process is interrupted many times because other processes are competing for the CPU, we do not want to charge the process for the context-switch overhead. Most systems cannot distinguish between these two cases, and the process is or is not charged for the context-switch overhead irrespective of which process is “responsible” for the switches. Time spent serving cache misses are always charged to the running process, even if the misses occur because of a context switch. Many systems do not even have cache miss counters and therefore cannot even determine how much of the CPU time was spent waiting for cache misses to be served.

Some performance measures, such as CPU time, are rarely defined exactly or measured directly. Instead, other related quantities are measured and reported by a variety of hardware and software mechanisms such as hardware timers and operating-system timers. We refer the reader to Malony [81], who explores the issue of performance observability in detail. In spite of these difficulties, in this thesis I try to pick one of the reported measures which correlates well with the CPU time and predict this measure using performance models. This pragmatic approach seems to work well in practice, but it does not constitute a replacement for good definition of performance measures and good mechanisms for measuring them.

Performance Determinants

The running time of a program or a subroutine, on a given input, depends on factors that we shall call **performance determinants**. There are **external** performance determinants, such as

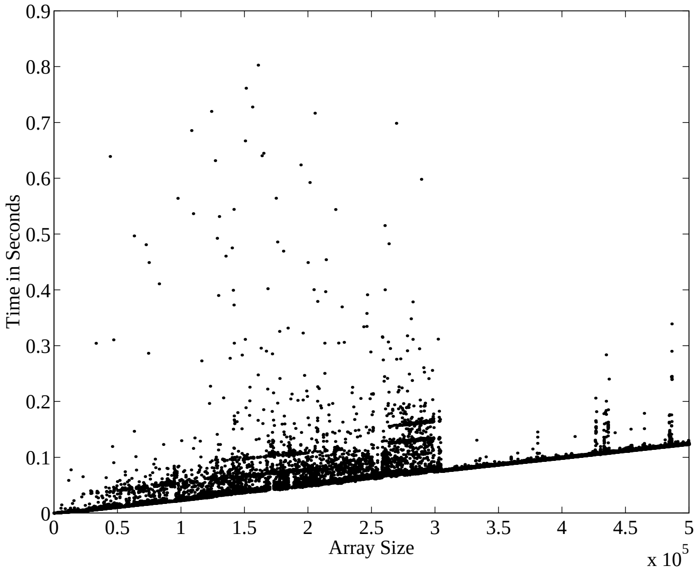


Figure 2.1 A scatter plot of the time it takes to copy an array of integers on an SGI Indigo2 workstation. Each dot represents one execution of an array copy function, which is timed using a builtin cycle counter. Five executions are plotted per array size, which are 100 elements apart. The executions were performed in a loop from small to large arrays, so real time advances from left to right in the graph. The distribution is largely one sided, with concentrations along lines parallel to the smallest running times. These lines seem to be caused by the timesharing quantum. The noisy and quiet regions represent different loads on the machine.

the load on the system and the level of fragmentation of the swap file, and there are **internal** performance determinants that describe the input, such as its size, content, and location in memory. The goal of a performance model is to predict the performance of a program or a subroutine given its performance determinants, or more often, a subset of its performance determinants.

For example, we might want to predict the running time of a matrix multiplication subroutine given the dimensions of the two input matrices. The running time might also depend on the location of the input and output in memory, the values of the elements of the matrices, and the load on the machine. If we ignore these factors, we cannot predict the running time exactly. In such cases, we try to predict performance as a range of possible running times.

Missing Performance Determinants

Predicting performance with missing performance determinants means specifying a range of possible performance measures. Different values for the missing determinants yield different performance measures within the predicted range. But performance outside the predicted range is a prediction error. External performance determinants are always missing from the models described in this thesis as well as from most models described by other researchers. Internal performance determinants are also sometimes missing from my models, especially contents of arrays.

Vague definitions of performance and imperfect ways of measuring performance may cause different running times to be reported in different executions of a single program on a single input data set, even if the system and the program have no randomized components. This effect is of course not random noise, but rather reflects different states of the computer system when the program is executed. In other words, the effect is caused by missing external performance determinants. For example, running a program on a heavily loaded timesharing system might seem to require more resources than running it on a lightly loaded system, if some of the context switching overheads are charged to the program's CPU time (which is almost always the case). Figure 2.1 shows the running times of a function copying an array on a workstation. The variability of the running time changes due to different loads on the system. Figure 2.2 verifies that light and heavy loads result in different running time distributions, even when the operating-system timer attempts to filter out the effects of context switches.

My viewpoint is that such effects must not be treated as random. A random phenomenon must have an underlying probability space, which is missing here. A lightly loaded system is not more probable or less probable than a heavily loaded system. I therefore believe that if the magnitude of such effects is large, the performance should be specified as a range of possible outcomes rather than a "mean" and a "standard deviation" or a "confidence interval". Such estimates of statistical parameters should only be used to describe the performance of randomized systems. Most computer systems exhibit large deviations in performance which are not random, and some smaller random deviations.

Some programs may exhibit wildly different running times on slightly different arguments, or on arguments that reside in different locations in memory, due to conflicts in memory systems. For example, Figure 2.3 shows that the time its takes to copy an N-elements array on an SGI Indigo2 workstation is about $193N$ ns, except when the size of the array is a multiple of 16 Kbytes, in which case the time rises to about $339N$ ns. If the size is a multiple of 1 Mbytes, the time rises further to about $2870N$ ns. In all cases, the array is copied to an adjacent location in memory, (such as the next matrix column in Fortran), which may cause conflicts in the two levels of direct-mapped cache that this workstation uses. Similar behaviors can result from other cache organizations and from interleaved memory banks. These effects are not random. The description of the performance of programs with such behaviors should include lower and upper bounds, possibly with a characterization of what performance to expect in which cases. Statistical estimates of the distribution of the running time are again not recommended, because of the lack of an underlying probability space. The assumption that all array sizes are equally likely for example, seems unattractive to me, and in any case must be specified if used.

2.3 Choosing a Decomposition Interface

The first step in performance prediction is to decompose the dynamic behavior of a program into components. The choice of a decomposition interface involves two decisions. We must decide whether the decomposition is done before or after compilation. Most programs use a hierarchy of interfaces. We must therefore choose one of the interfaces as the decomposition interface. This sections explains why I decided to decompose programs after compilation and why I decided to use a specific class of runtime system interfaces, data-parallel runtime systems, as the decomposition interface.

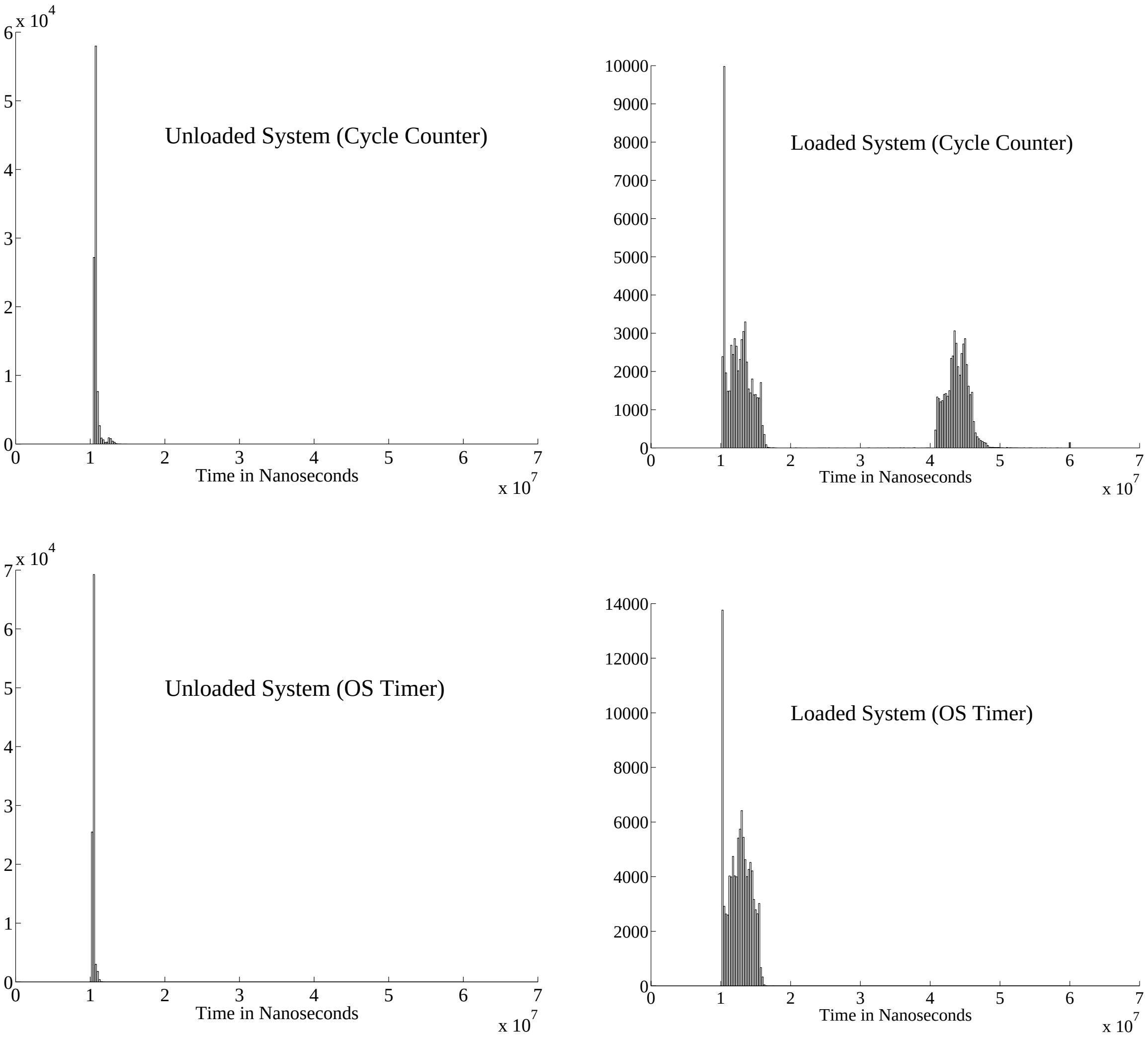


Figure 2.2 Histograms of the running times of 100,000 executions of an array copy function on an SGI Indigo2 workstation on arrays of size 100 K integers. The experiments were performed on an unloaded system (right), and a loaded system (left). The top histograms present the running time reported by a cycle counter which is oblivious to context switches, and the bottom histograms present the running time reported by the operating system counter. The size of the bins in the histograms is 250 μs. The distribution is not symmetric about the mean even in the unloaded system. The operating system timer filters out some of the effects of the context switches, but not all of them: on the loaded system, the noise has a large positive mean, and a large variance.

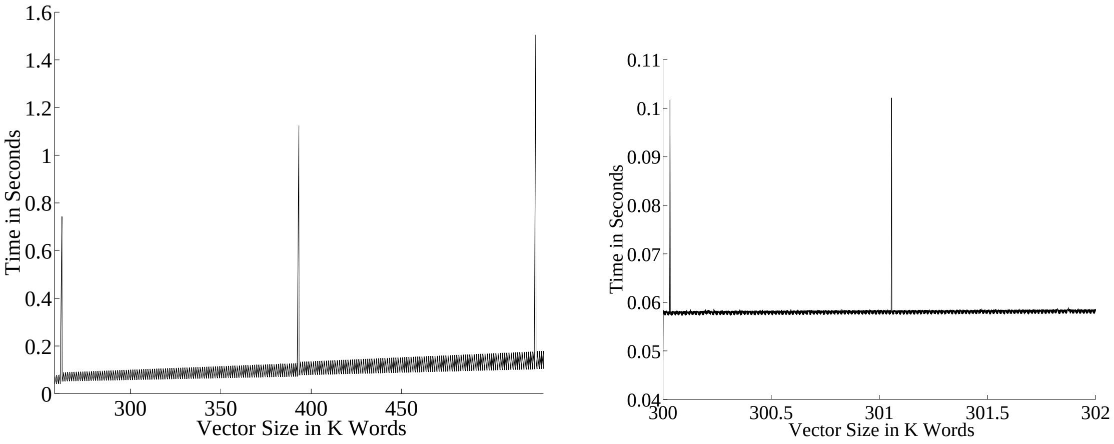


Figure 2.3 The time it takes an SGI Indigo2 to copy an array to an adjacent location in memory. The graph on the left shows that cache conflicts caused the direct mapped, physically indexed offchip cache degrade performance by a factor of about 14. The graph on the right presents one region of array sizes in detail, and shows that cache conflicts caused the direct mapped, virtually indexed onchip cache degrade performance by a factor of about 1.75. The fact that conflicts in virtual memory cause misses in a physically indexed cache seems to be due to the virtual-to-physical memory mapping policy of the operating system.

Decomposing Source and Object Codes

I believe that it is currently impossible to base accurate performance predictions on a decomposition of the source program, and that therefore, performance modeling must be based on a decomposition of the compiled program. The decomposition can be performed by the compiler or by a separate software tool. A compiled program consists of assembly language object code, sometimes with calls to runtime libraries. Simulators decompose a program into a sequence of machine instructions. Simulators can produce accurate performance predictions, but they are slow due to the large number of instructions that a typical program executes. Our efforts to rapidly predict performance lead us to search for a higher-level interface. As we shall see in the next subsection, data-parallel runtime systems provide such an interface. This section provides the evidence that source-based predictions cannot be accurate.

Some researchers propose to model programs in terms of the performance of source code constructs rather than in terms of runtime systems. Modeling the program in terms of the source is an attractive idea, because if it can be done accurately, programmers can do the same thing more-or-less intuitively. Early experiments with this technique were successful. For example, Bentley [14, pages 99–101] used models developed by Mary Shaw to predict the performance of several Pascal programs. The models were based on source code constructs, and they predicted the running time of Bentley's programs to within a 26% relative error, which we consider reasonably accurate. The programs were compiled with a nonoptimizing compiler and executed on a DEC PDP-KL10 computer, a model introduced in 1975. Bentley notes, however, that it is difficult to use such models with highly optimizing compilers.

Current literature seems to support Bentley's comment. MacDonald [79] attempts to predict the running time of the Lawrence Livermore Loop Kernels, a set of Fortran loops, using models

```
1       real a,b,x(n)
2       do i = 1,100
3         a = sum(x)
4         x = x/a
5         a = a*b
6       end do
```

Figure 2.4 A data-parallel Fortran program fragment.

similar to Bentley's. While on some machines his models are quite accurate, they err by almost a factor of 3 on an SGI workstation using an R3000 microprocessor and an optimizing compiler. These results support my claim that performance models should not attempt to model source code constructs.

I believe that optimizing compilers and architectural features such as pipelines, superscalar execution, multiple caches, and parallel processing, render accurate source-based predictions virtually impossible. Therefore, in this thesis programs are decomposed into sequences of runtime-system calls. In one benchmapping system, PERFSIM, program are decomposed into sequences of operations that include both runtime-system calls and invocations of compiler generated loops. Both the runtime system and these loops are already compiled and their performance can be modeled more accurately than the performance of source-code constructs.

Modeling Data-Parallel Runtime Systems

Runtime systems that support the **data-parallel** programming model [16] are well suited for benchmapping. Benchmapping a runtime system requires that the performance of its subroutines can be predicted from available performance determinants, and that the relation between the performance of a whole program and the performance of the runtime subroutines it calls be known. This section shows that the running time of data-parallel programs is essentially the sum of the running time of the vector operations in the program, and that it is possible to predict the running time of vector operations.

Data-parallel programs express most of the computation as a series of transformation of vectors, or arrays. Data-parallel Fortran programs, for example, 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 2.4, 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. Data-parallel NESL programs [17] also consists of sequential control and vector operations, but scalars are treated as vectors of length 1 and not as distinct data structures.

The running time of many data-parallel programs is dominated by the running time of vector operations. Many data-parallel languages, including CM Fortran [109] and NESL, do not exploit control-flow parallelism and parallelism in scalar operations. Since most programmers want their data-parallel programs to run well on parallel computers with a large number of processors, they write programs in which most of the work is expressed in vector operations that are parallelized. The sum of the running time of vector operations is therefore a good approximation of the running time of many data-parallel programs. If this sum is not a good approximation, then the program is not likely to run well on a larger number of processors.

The running time of most vector operations can be accurately predicted from a small number of scalar performance determinants. The performance determinants of vector operations include the scalar arguments of the operation and the layout of vector arguments in the memory of the computer. The running time of some vector operations is also affected by the contents of vector arguments. Predicting running time independently of the contents of vectors limit the accuracy of the prediction. Nevertheless, Chapter 5 argues that the impact of ignoring the content of vectors does not severely limit the usability of benchmaps-based tools.

One important reason the running time of vector operations can be accurately predicted is that their memory access patterns are often regular and predictable. The number of cache misses is therefore also regular and predictable. Some vector operations, such as matrix multiplication, can have a very high cache hit rate. Other vector operations have very low cache hit rates when the arguments are large vectors that do not fit within caches. For example, Chen [30] found that it is possible to predict the performance of vector operations on a CM-5 with data caches when the vectors were larger than the size of the cache, because the cache was too small to exploit the temporal locality in the program.

Runtime systems that support low-level programming models, such as **message passing** and **active messages** [122], are more difficult to model than runtime systems supporting the data-parallel model. The number of cache misses is hard to predict in low-level runtime systems, and the running times of individual runtime system calls are not independent. For example, the running time of a `receive` operation in a message-passing program depends on when the corresponding `send` operation was executed.

2.4 Decomposing a Program into Components

The performance-modeling systems described in this thesis generate the sequence of operations in the program by executing the control structure in the program. Executing the control structure eliminates the need for a complex static analysis of the program. In most data-parallel programs most of the running time is spent in vector operations, and most of the memory stores vectors. Since we are only interested in the sequence of operations the program performs and not in the output of the program, we avoid performing vector operations whenever possible. Not constructing vectors and not operating on vectors allow the program to execute very rapidly even on small machines with limited memory, such as workstations. This section explains how the program can execute when vector operations are not performed. Sections 4.5 and 4.8 describe the details of the decomposition of a program into a sequence of runtime-system calls in PERFSIM and in BENCHCVL.

The control structure of many scientific program can be executed correctly even when vector operations in the program are not performed. The control structure in some scientific is completely independent of values stored in vectors, and therefore program can execute even if vector operations are not performed. More often a few control flow decisions in the program depend on values stored in vectors. Convergence tests in iterative numerical algorithms are a typical example. Section 4.3 describes a mechanism that allows a programmer to annotate the program to remove such data dependencies in performance prediction runs. The annotations, called **multiplexors**, specify how to replace a data-dependent control flow decision with a data-independent one. For example, a `while` loop is annotated with a loop count that allows the loop to execute even when vector operations are not performed. The small number of data-dependent control flow decisions in typical scientific programs means that the extra effort required to annotate the program is often insignificant.

Fahringer and Zima [43] have taken a similar approach, but they rely on a profiling run to derive loop counts automatically. Although their approach eliminates the need for explicit annotations in the program, the profiling run slows down performance prediction considerably, especially since in their system the profiling run is performed on a sequential machine.

2.5 How Benchmaps Specify Performance

A benchmap specifies the performance of a computer system using a collection of performance models, one for each subroutine in the runtime system. Our goal is to describe the performance of the system with models that are accurate and therefore detailed. But we do not want to specify all the details by hand. The performance models that are used in this thesis have two parts. One part describes the hardware resources in the system. It is specified by hand. A second part describes the effective performance of each resource in each subroutine in the runtime system. This part can be generated automatically.

We describe the hardware in whatever level of detail necessary for accurate performance predictions. The structure of the hardware models of PERFSIM and BENCHCVL is described in Sections 4.2 and 4.7, respectively. The structure of PERFSIM's benchmap is specific to the architecture of the Connection Machine CM-5. The structure of BENCHCVL's benchmap is tailored for distributed-memory parallel computers with a hierarchy of cache memories in every processor, but it is not specific to any particular machine. We expect that accurately modeling new architectures will require benchmaps with new structures, because new architectures are likely to have new resources that must be modeled.

The process of estimating the performance of each resource for each subroutine, on the other hand, can be automated and needs little or no improvement to handle new architectures. The estimates of the performance of resources, for example the bandwidth of the memory system or the size of caches, are based on results of experiments performed on the system. The process of performing the experiments and estimating the parameters of a benchmap is described in Chapter 3.

Chapter 3

Creating Benchmaps

Benchmaps can be generated automatically using simple techniques and available software tools. Benchmaps are created in two phases. In the first phase, the performance of subroutines is measured on many different inputs. We call this phase **surveying**. In the second phase, the measured performance is used to create models that predict the performance of subroutines on all inputs. We call the creation of benchmaps from the survey data **cartography**. Section 3.1 describes how a software system for benchmapping can be organized. Section 3.2 describes an efficient and comprehensive strategy for surveying performance. Sections 3.3 through 3.5 describe robust techniques cartography. Section 3.6 describes a software tool called CARTOGRAPHER that implements all the modeling techniques described in this chapter.

3.1 A Software Organization for Benchmapping

Many of the applications of benchmaps require a software organization that supports automatic generation of benchmaps and automatic performance prediction using the benchmaps. The automatic generation of a benchmap enables the rapid modeling of large runtime systems, simplifies the maintenance of the benchmap over the lifetime of the runtime system, and ensures the reliability of the benchmap. Automatic performance prediction using the benchmap allows the modeling software to be used by users and other programs, rather than by performance analysts alone. The performance modeling systems described in this thesis utilize such an organization, which is described in this chapter.

The organization of my modeling systems, illustrated in Figure 3.1, consists of an instrumented interface to the runtime system, a program, which we shall call a **surveyor**, that measures the performance of the runtime system, and a mechanism to transform performance measurements into a benchmap. We refer to the transformation of survey data into a benchmap by performance cartography.

The surveyor performs a set of **experiments** on the runtime system. The results of the experiments serve as reference points where performance is known. Cartography uses the reference points to construct a benchmap of the runtime system. The surveyor must generate enough reference points to create a benchmap within a reasonable amount of time. Section 3.2 describes a surveying strategy capable of generating enough reference points to model data-parallel runtime systems within a short amount of time.

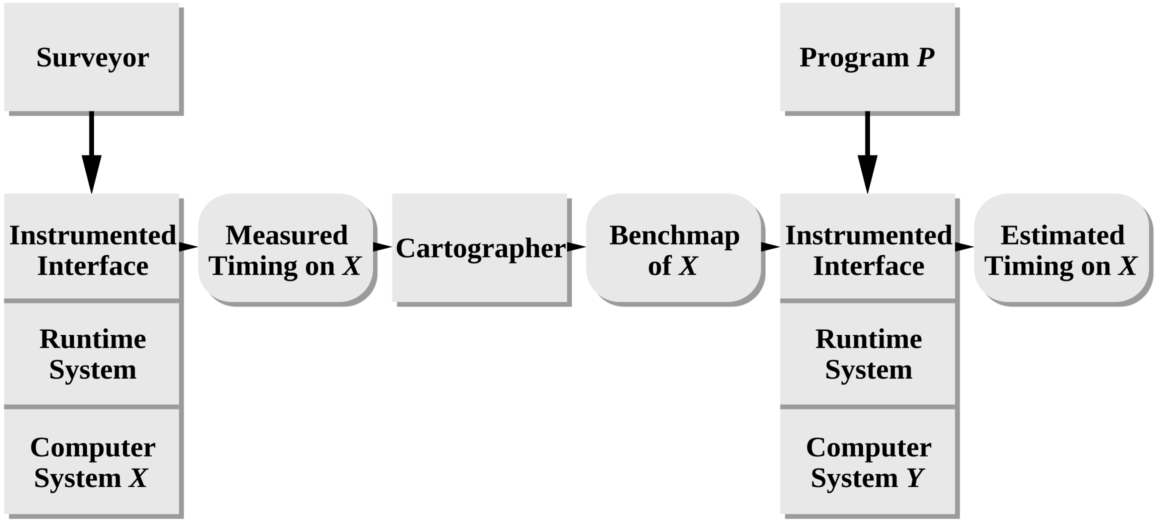


Figure 3.1 A software organization for benchmapping systems.

The instrumented interface to the runtime system records the results of performance experiments and predicts performance of calls to the runtime system using a benchmap. Each experiment is recorded as a set of performance determinants and a set of performance measures. The instrumentation in the interface therefore consists of a performance measuring mechanism and a mechanism to ascertain the value of performance determinants. The interface can use a benchmap to map the performance determinants associated with a call to the runtime system to the predicted performance of the call.

The cartographer uses reference points of known performance to create models that can predict the performance of calls in other points in the performance-determinants space. The cartography technology used by my modeling systems is described in Sections 3.3 though 3.6.

Other researchers have also found that automatic surveying, cartography, and performance prediction enable applications of performance modeling. Brewer [22, 23] propose a system that automates all three. Chen [28] and Chen and Patterson [29] propose a highly advanced surveying technique for modeling I/O performance. Their surveyor automatically locates the most important regions in the performance-determinants space and performs most of the experiments in those regions.

3.2 Surveying Performance

Surveying establishes **reference points** in the performance-determinants space in which performance is measured and recorded. The sampling strategy must establish enough reference points so that accurate models can be created, but not so many that surveying becomes impractical. We clearly cannot run subroutines on every possible input, for example. Since performance often cannot be measured directly, reference points must be indirectly derived from measurements. For example, a coarse timer requires that the running time of a subroutine be derived by timing multiple invocations of the subroutine, rather than just one.

I propose to survey the performance of data-parallel runtime systems using a geometric distribution of vector sizes. A geometric distribution of experiments performs a constant number of experiments on vector sizes between N and $2N$. The geometric distribution performs many experiments on small vector sizes and many, but not too many, on large vector sizes. Since input sizes can range from one to a million or even a billion, and since “interesting” behaviors occur at different scales, it is important to perform experiments on many scales. For example, on a Sun SPARCstation 10 the on-chip data cache size is 16 Kbytes, the off-chip cache size is 1 Mbytes, the main memory size is 32 Mbytes, and the virtual memory size is several hundred Mbytes. To model performance accurately, we need many experiments on each of these scales.

In contrast, a uniform distribution that performs αN experiments on vector sizes between N and $2N$ for some constant α performs too many experiments on large vector sizes or too few on small sizes.

The distribution of experiments should include both vector sizes that are powers of 2 and random vector sizes that are not powers of 2. Performing experiments on both powers of 2 and random sizes is likely to uncover conflicts in memory systems. Some memory systems, such as ones with direct mapped caches and interleaved memory banks, perform poorly on vector sizes and axes sizes that are powers of 2, while other memory systems, such as ones with set associative caches, perform better on powers of 2 than on random sizes.

A large number of reference points is more likely to uncover an incorrect model structure than a small number. Consider for example a linear cost model $x_1 + x_2 N$ for the time it takes to add two vectors of size N on a workstation. We can estimate the constants x_1 and x_2 very accurately from only two groups of experiments, one on a very small N and another on a very large N. However, we can never learn from these two input sizes that the real behavior is not linear, because of caches for example. A larger set of experiments may enable us to refute the conjecture that the behavior is linear. Furthermore, the refutation can be found automatically if the parameter estimation algorithm signals the operator that it failed to find a good model.

What distributions of values should be used in the experiments? Ideally, we should use a variety of distributions which spans the range of possible execution times. Models which are upper or lower bounded by the experiments then describe well the range of possible running times. For example, using both identity and random permutations in experimentation with a permutation routing library function might cover the entire range of possible running times of the function.

But on some systems, routing some specific permutations may take much longer than routing a random permutation, so on these systems our upper bound may be too optimistic. A similar example I encountered involves integer multiplication on SuperSPARC microprocessors. The number of cycles this operation requires turned out to be highly dependent on the values being multiplied. Since I did not expect this, I experimented with only one set of values, and the models turned out to be inaccurate. Finding a set of distributions which spans the range of execution times is important, but may be elusive in practice. I believe that evaluating of the models using a large test suite before they are used in production, may serve as a partial solution to this problem.

3.3 Linear Models and the Singular Value Decomposition

A **linear model** describes performance as a linear function of a set of functions of performance determinants called **basis functions**. Linear models are the basic building blocks in my benchmapping systems. Let $v = (v_1, \ldots, v_k)$ be the performance determinants of a runtime system subroutine. For example, the performance determinants for the summation of an N-vector distributed on P processors might be N and P. A linear models maps the performance determinants to a set of m basis functions $a = (a_1, \ldots, a_m)$ using a map F, such that $a = F(v)$. In the summation example, the basis functions might be 1, N/P, and $\log P$, which represent, respectively, a fixed cost, the cost of summing the vector section on each processor, and the cost of summing the partial sums. The linear models describes the execution time of the function as a linear combination of the basis functions,

$$\begin{aligned} \text{execution time} &= x_1a_1 + x_2a_2 + \cdots + x_m x_m \\ & \quad x_1F_1(v) + x_2F_2(v) + \cdots + x_mF_m(v)\ . \end{aligned}$$

A model then, is a set of basis functions F and a choice of costs $x = (x_1, \ldots, x_m)$. Consider n experiments in which the values of the basis functions in the ith experiment are $a_{i,1}, \ldots, a_{i,m}$, and the running time of the ith experiment is b_i. We denote the ith **reference point** by $[a_{i,1}, \ldots, a_{i,m}, b_i]$ and set of all reference points by $[A, b]$. The vector of predicted running times of the experiments given a model x is Ax, and the prediction errors are $(Ax - b)$. We normally try to find a value for x that minimizes the error vector $(Ax - b)$ in some norm.

The basis functions we use to model the performance of a library function correspond to possible costs the function might incur. If some of the costs are not incurred, the matrix A may be rank deficient. Consider, for example, a function that permutes a vector. Given an input vector s, a destination d, and a permutation vector π (containing a permutation of $1, 2, \ldots, N$), the function permutes the input so that upon return, $d[\pi[i]] = s[i]$. A linear model describing the time it takes to permute a vector of size N on P processors might be $x_1 + x_2(N/P) + x_3(N(P-1)/P)$. The first term represents a fixed cost, the second the cost to handle each one of the N/P elements on each processor, and the third the cost of sending and receiving messages. The expected number of messages sent and received by a processor is roughly $N(P-1)/P$, when π is a random permutation. On uniprocessor architectures no messages are required, so the matrix A containing the basis functions for the experiments has a zero column. If we rearrange the cost model as $x_1 + x_2(N/P) + x_3(N(P-1)/P) = x_1 + \tilde{x}_2(N/P) + x_3N$, where $\tilde{x}_2 = x_2 - x_3$, the matrix A has two equal columns. In general, our models have basis functions which are **inactive** on some architectures, by which we mean that they have no variation or that they linearly depend on other basis functions.

It is numerically difficult to search for a linear model x for a set of reference points $[A, b]$ when the matrix A is rank deficient. Many numerical optimization techniques that can find a value for x that minimizes the error vector $(Ax - b)$ in some norm break down when A is rank deficient. We use the Singular Value Decomposition (SVD) to handle rank-deficient matrices arising from inactive basis functions.

The SVD of an n-by-m matrix A is

$$A = U\Sigma V^{\mathrm{T}}$$

where U is an n-by-m orthonormal matrix (that is, has m orthonormal columns), Σ is an m-by-m diagonal matrix with nonnegative entries $\sigma_1, \ldots, \sigma_m$ such that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_m$, and V is an m-by-m unitary matrix. When A has rank r, exactly r entries in Σ are nonzero. Since V is nonsingular, the matrices A and $AV = U\Sigma$ span the same subspace. If A has rank

r, then A has the same span as $A\tilde{V} = \tilde{U}\tilde{\Sigma}$, where $\tilde{V} = V(:,1\!:\!r)$ (that is, the first r columns of V), $\tilde{U} = U(:,1\!:\!r)$, and $\tilde{\Sigma} = \Sigma(1\!:\!r,1\!:\!r)$. The matrices $A\tilde{V}$ and $\tilde{U}\tilde{\Sigma}$ have full rank r (since they are n-by-r) and the same span as A. A linear model $\tilde{x}$ for these matrices can be translated into a linear model x for A using the transformation $x = \tilde{V}\tilde{x}$.

3.4 Criteria for Choosing Models

What should be the criterion that determines whether one performance model is more accurate than another? Given a model for a subroutine, the **prediction error** in an experiment (an invocation of the subroutine) is the difference between the predicted performance output by the model and the actual performance. The **relative error** is the prediction error divided by the actual performance. For example, if the actual running time of an experiment is 10 seconds and the predicted running time is 9 seconds, then the prediction error is -1 second and the relative error is -10%. This section shows that choosing the model that minimizes the maximum relative error over all the reference points is both feasible and desirable.

Minimizing absolute rather than relative errors often results in models with large relative errors on experiments with short running times (or small values for other performance measures), especially if the set of reference points contain experiments with a wide range of running times. We should therefore prefer models that minimize relative errors. Minimizing absolute and relative errors in linear models is computationally equivalent. For a set of experiments with basis functions A and actual performance b, the absolute errors are $(Ax - b)$ and the relative errors are $B^{-1}Ax - B^{-1}b$, where B is a diagonal matrix with entries $B_{i,i} = b_i$. Hence, scaling the ith row of A and b by $1/b_i$ yields a set of reference points for which the error vector yields the relative errors for the original experiments.

The linear model x that minimizes the maximum prediction error (or the maximum relative error) is the solution of the linear-programming problem

$$\begin{array}{ll} \text{minimize}_{x,\epsilon} & \epsilon \\ \text{subject to} & Ax - b \le \epsilon \\ & Ax - b \ge -\epsilon \, , \end{array}$$

which has dimension $m + 1$, variables $\epsilon, x_1, \ldots, x_m$, and $2n$ constraints [65]. The problem is feasible and bounded, which can be seen by setting $x = 0$ and $\epsilon = \max_i |b_i|$. Any solution automatically satisfies $\epsilon \ge 0$. Linear-programming problems can be solved with simplex and interior point algorithms, which are implemented in a variety of software packages and libraries.

Minimizing the sum of the squares of the errors is a popular optimization criterion for linear models, sometimes known as the least-squares criterion. The popularity of least squares stems from efficient algorithms for finding least-squares models and from theoretical advantages the criterion has in statistical modeling [13]. The singular value decomposition $U\Sigma V^{\mathrm{T}}$ of A can be used to solve least-squares problems, even rank-deficient ones. It can be shown [37] that the solution x of the least-squares problem is $x = \tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^{\mathrm{T}}b$, where r is the rank of A, $\tilde{V} = V(:,1\!:\!r)$, $\tilde{U} = U(:,1\!:\!r)$, and $\tilde{\Sigma} = \Sigma(1\!:\!r,1\!:\!r)$. In practice, we round small σ_i's down to zero to determine the rank of A.

The disadvantage of the least-squares and similar averaging criteria is that they may ignore large errors on small subsets of reference point. Suppose that the running time of a subroutine is N seconds on inputs of size N, except if N is a multiple of 1000, in which case the running time is $5N$ seconds. If we use experiments on every N between one and one million, the least-squares model will be very close to N, because this models errs on only 1 reference point in a 1000. If we use N in steps of 500, however, the least-squares model will be close to $2.5N$. A minimal maximum error model eliminates this sensitivity because it will choose a $2.5N$ model with both sets of reference points.

The theoretical advantages of the least-squares criterion disappear when the statistical assumptions that underlie it are not satisfied [13]. Unfortunately, in performance modeling these assumptions are often violated. For example, the distribution of running times is often one-sided and not symmetric, as shown in Chapter 2.

We nonetheless use the least-squares criterion as well as the minimum maximal error criterion, because the least-squares criterion is computationally cheaper to solve. But we always assess the accuracy of models using the maximum error.

3.5 Bounding Performance

Since prediction errors are inevitable, the most reliable models are those that predict performance as a range that is guaranteed to include the actual performance. A model cannot normally predict performance exactly, because a few performance determinants are often unknown. Some determinants may be completely unknown, such as certain aspects of the state of the computer system (the identity of resident virtual pages, for example). Other determinants may only be approximated, such as the amount of data that needs to be transferred across a communication channel to route a permutation on a distributed-memory parallel computer. The performance of a subroutine is therefore best described by two models, one which always slightly overestimates performance and another which always slightly underestimates performance.

It is possible to find the linear model that minimize the maximum error or the sum of square errors among all models that upper or lower bound the set of reference points. The upper or lower bounded least-squares problem is a quadratic programming problem. That is, it is a quadratic optimization problem with linear inequality constraints. The optimization problem is convex if A has full rank. Otherwise, we use the SVD to transform A into a full rank matrix with fewer columns. The formulation of the upper bound problem is

$$\begin{array}{ll} \text{minimize}_x & x^{\mathrm{T}} A^{\mathrm{T}} A x - 2 x^{\mathrm{T}} A^{\mathrm{T}} b + b^{\mathrm{T}} b \\ \text{subject to} & Ax \geq b\ . \end{array}$$

The bounded maximum error minimization is a linear programming problem. It is almost identical to the unconstrained problem, except that the errors are constrained to be one sided rather than lie in the interval $[-\epsilon, \epsilon]$. The upper bound problem becomes

$$\begin{array}{ll} \text{minimize}_{x,\epsilon} & \epsilon \\ \text{subject to} & Ax - b \leq \epsilon \\ & Ax - b \geq 0\ . \end{array}$$

Figure 3.2 shows an upper bounded, lower bounded, and unconstrained least-squares models for a range of input sizes of some function.

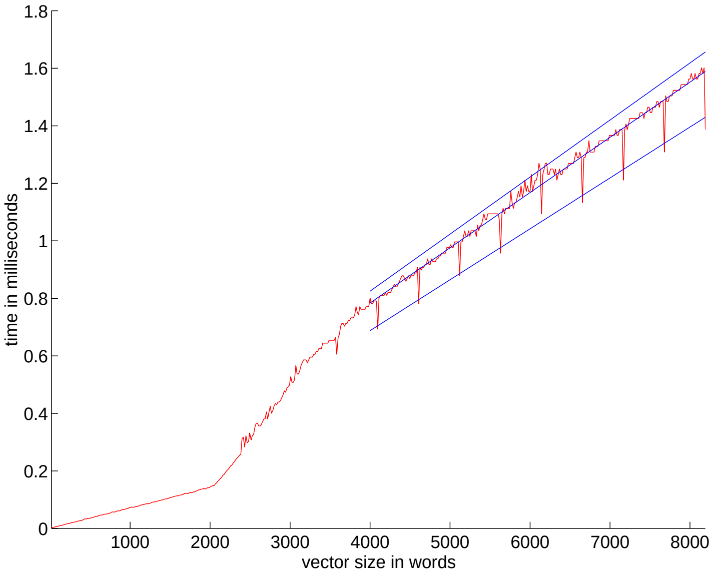


Figure 3.2 The execution time of a function copying a vector of integers to another location in memory on a Sun SPARCstation 10 workstation. The source and destinations are adjacent in memory, and the dips in the execution time are apparently due to a set associative offchip cache. The graph also shows three least-squares models of one region of input sizes, an upper bounded, a lower bounded, and an unconstrained models. The bounded models miss some of the experiments due to the fact that the algorithm used to solve the quadratic programming problems is an approximation algorithm.

3.6 Performance Mapping with CARTOGRAPHER

This section describe CARTOGRAPHER, a software tool for performance cartography that implements most of the techniques described in this chapter. CARTOGRAPHER's parameter estimation algorithms include algorithms that find bounded and unconstrained linear models that minimize the maximum error or the sum of square errors. CARTOGRAPHER experiment manipulation facilities include algorithms to scale reference points so that linear models minimize relative rather than absolute errors, as well as an SVD algorithm that allows parameter estimation algorithms to handle rank-deficient problems.

CARTOGRAPHER has an open architecture that allows parameter estimation modules to be added. The open architecture enables benchmaps system to use the best available parameter estimation algorithms using a uniform interface, and to add special-purpose parameter estimation algorithms. I have implemented two general-purpose parameter estimation modules, one for interfacing to MATLAB [108], an interactive mathematical software package, and another for interfacing to LOQO [121], an linear and quadratic optimization package. The BENCHCVL system adds one special-purpose parameter estimation module for modeling data caches, which is described in Chapter 4.

The MATLAB interface uses an interprocess communication library supplied with MATLAB, which starts a MATLAB process, can send and receive matrices to and from this process, and can execute MATLAB commands and programs. We use it to compute the SVD of matrices, to scale and solve least-squares problems, and to solve convex quadratic programming problems using an iterative algorithm, published in [90, pages 373–375]. The MATLAB module is also used for visualizing performance and models.

The LOQO interface provides access to a linear and convex quadratic programming package written by Richard J. Vanderbei, which uses a state-of-the-art interior-point algorithm. The interface communicates with LOQO by writing and reading appropriately formatted files.

I am considering adding at a later stage a LAPACK [2] interface module, which would add portability and some independence from external packages, and a dense Simplex algorithm, which would use either MATLAB or LAPACK to perform the required linear algebra.

Chapter 4

Performance Modeling Systems: PERFSIM and BENCHCVL

By describing two concrete benchmapping systems, this chapter shows how benchmaps are constructed and used. The systems are PERFSIM[1], a profiler for data-parallel Fortran programs, and BENCHCVL, a portable performance prediction system for portable data parallel NESL programs.

4.1 The Goals and Structure of PERFSIM

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.

PERFSIM is a benchmapping system that accelerates the profiling process by estimating the running time of most of the expensive operations in a program, while refraining from actually performing them. PERFSIM analyzes CM Fortran[2] [109] programs running on the Connection Machine CM-5 [110]. By combining execution of the control structure and scalar operations in a program with analysis of vector operations, PERFSIM can execute a program on a workstation and in seconds and generate performance data that would take several minutes or more to generate by running the program on an actual CM-5[3]. Our empirical studies, described in Table 4.1, 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 $\pm 21\%$,

[1] A paper describing PERFSIM was presented in the *5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, Virginia, February 1995.

[2] The current version of PERFSIM works with CM Fortran version 2.1 beta 0.1 with a runtime system dated 94/04/07.

[3] Due to a minor technical problem with the initialization of the CM runtime system, it is currently not possible to execute PERFSIM on an ordinary workstation. It executes on a CM-5 partition manager, which is essentially a Sun workstation, but it never uses 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 4.1 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 indicate that the vector operation is an insignificant cost in the program.

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. Chapter 5 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.

4.2 PERFSIM's Benchmap: Modeling a Runtime System and Compiled Code

The structure of the models in PERFSIM's benchmap are based on a detailed model of the CM-5's hardware, which is described in Figure 4.1. This structure enables very accurate modeling at the expense of portability to other hardware platforms. The models fall into two categories: models of runtime-system subroutines, and models of compiler generated subroutines. The structure of the models of runtime-system subroutines is based on the organization of the CM-5's communication channels, and the models' parameters are estimated using a performance survey. Models of compiler generated subroutines, on the other hand, are based on a model of the pipeline in the vector units installed on CM-5's nodes.

Modeling Runtime Subroutines

Most of the subroutines in the CM Fortran runtime system implement vector communication operations. 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, (parallel prefix operations), can be computed. While scans and reductions perform computation as well as communication, I still

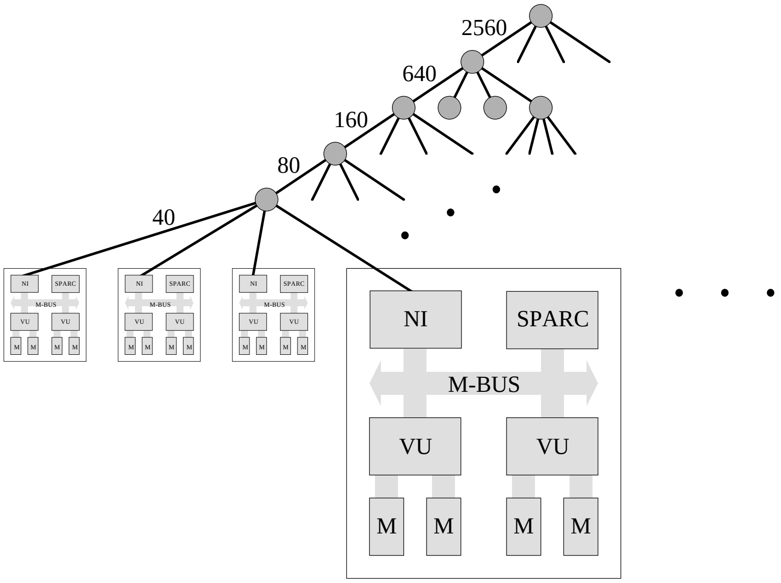


Figure 4.1 A schematic diagram of the CM-5. Each node contains a network interface (NI), a SPARC microprocessor, two vector unit chips (VU's), and four memory banks (M's), two attached to each VU chip. The SPARC microprocessor, the VU's, and the NI, are connected by a memory bus (M-BUS). The nodes are 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.

classify them as communication operations in PERFSIM's benchmap.

My 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 determine 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 separate communication channels, one of which contains several subdivisions. Figure 4.1 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 [73].

- Data transfer from one node to another using the CM-5's control network [73]. 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 4.1. 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 that 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 the performance of communication channels within nodes most, and therefore the bandwidth of the communication channels within each node, including the NI, is assigned per channel and per subroutine. The assumed bandwidth of communication channels in the data network, however, is the same for all subroutines (but of course 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.

I have found that while all the channels of the same type in the CM-5 operate in parallel, different types of channels are 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 the 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. Consequently, the running time of a communication operation 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. Three parameters describe the bandwidth in terms of the amortized transfer time for one data item on each of the three communication channels other than the control and data networks, one parameter describes the time it takes the processor to send and receive data to and from the data network (but not including the travel time in the network), and the fifth parameter describes the constant overhead term in the running time, which includes the latency of the control network if it is used.

I 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. I preferred this method over a automatic cartography because at the time my automatic cartography software was not robust enough to accurately estimate the five parameters PERFSIM uses in each model.

The bandwidth of the communication channels in the data network was given to me by Charles E. Leiserson.

Modeling Compiler Generated Loops

The CM Fortran compiler translates computations on arrays to **nodal code blocks**, also known as **nodal loops**, 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 runtime communication operation that transforms the geometry of one of the arrays, and then a nodal code block is invoked.)

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.

I 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 takes to execute, which depends on 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" are created in the VU pipeline. A bubble occur when a sequence of two consecutive VU instructions 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 [107].

4.3 Handling Data Dependent Decisions

Multiplexors form the core of PERFSIM's support for handling data dependencies. Data dependencies pose a problem for PERFSIM's simulation strategy, because PERFSIM does not actually execute vector operations. Consequently, if a control flow decision depends on a vector value, PERFSIM does not know how to make the decision. A multiplexor allows the user to specify a data-independent decision for use during simulation, while continuing to provide the data-dependent decision during actual runtime. This section shows how the multiplexor mechanism in PERFSIM can be implemented in a reasonably transparent and efficient fashion.

Multiplexors 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 multiplexor 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 multiplexors 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. The `loop_mux` works in PERFSIM's executions by keeping a hash table with the addresses of loops and the number of times each loop has been 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.

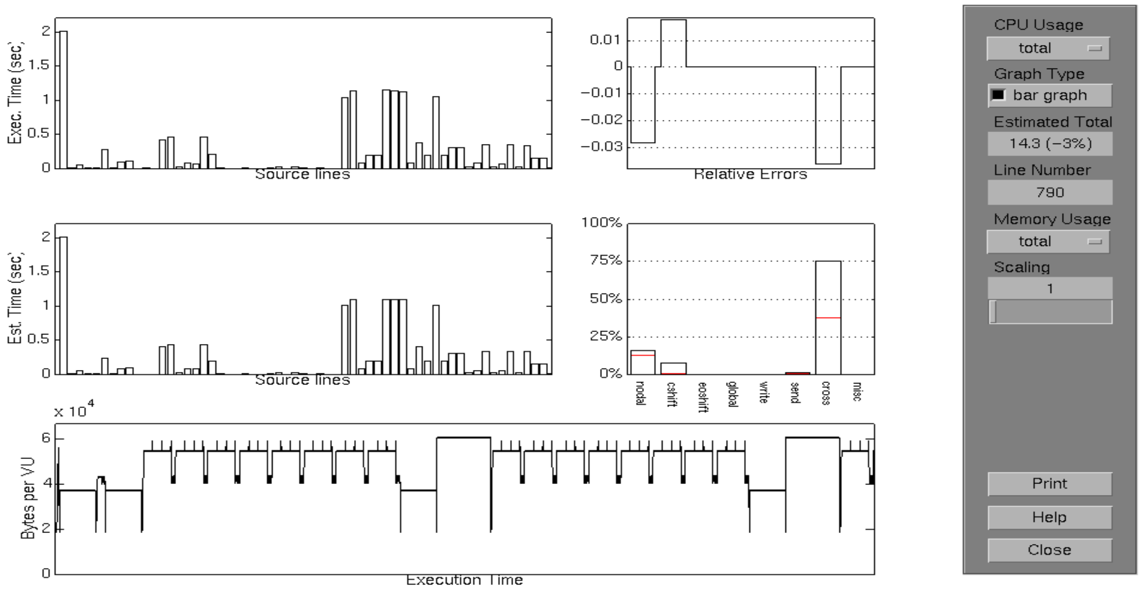
4.4 Using the Benchmap of the CM-5: Data Visualization and Performance Extrapolation

PERFSIM uses the benchmap of the CM-5 to visually present profiling information and to extrapolate performance. Like most current profilers, PERFSIM provides a data visualization facility, which is described in this section, to enable the programmer to analyze and use the large amount of data gathered in a typical execution. Since PERFSIM uses analytic models for the running time of vector operations, it can provide more information than ordinary performance monitoring tools that rely on measurements of actual vector operations. This section shows that PERFSIM can accurately extrapolate the running time of a program from executions on small data sizes to executions on large data sizes.

Data Visualization

The data visualization tool, which is implemented as a MATLAB application with a graphical user interface, displays three graphs (see Figure 4.2). 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



CPU Usage
total

Graph Type
■ bar graph

Estimated Total
14.3 (−3%)

Line Number
790

Memory Usage
total

Scaling
1

Figure 4.2 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 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.

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, suppose that the first phase of a program involves sorting its input (sorting involves permutation routing). Then, PERFSIM may not estimate the running time of this phase accurately, since the actual running time depends on the input data, but PERFSIM's estimates do not. Since PERFSIM breaks down the running time by source line and by operation types, the user can use PERFSIM for tuning the other parts of the program.

Performance Extrapolation

PERFSIM can 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, from two executions on small arrays of size 4096 and 8192, PERFSIM generated the graphs, shown in Figure 5.2, describing the estimated running times of three programs.

Let me first explain what I mean by performance extrapolation. Let us 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 extrapolating performance is quicker than running PERFSIM several times.

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 on two different data sizes, which we shall denote by λ_1 and λ_2. During each of the two experiments PERFSIM records which run-time subroutines were called on what geometries, how many times, and most importantly, how much data was transferred on each of the communication channels. By comparing the two experiments, 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 were sent out and received in each CM-5 node in one experiment, and that in the second experiment, 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 correctly handles 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, we need 3 different experiments: 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 required to extrapolate the running time. This approach may introduce array size aliasing, however, in which two groups of arrays that grow at different rates have exactly the same size in one of the two executions. Suppose, for example, that a program allocates some arrays of fixed size 32 and others whose sizes 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 occurs, the number of calls do not match and extrapolation does not take place.

4.5 PERFSIM's Software Organization

This section describe PERFSIM's software organization. The CM-5, as well as all the other Connection Machines and certain other parallel supercomputers, uses dedicated hardware to execute the scalar part of the program. On the CM-5 this hardware, called the **partition manager** or **host**, is a Sun workstation equipped with a network interface. As shown in Figure 4.3, the CM Fortran compiler generates a SPARC assembly language file containing the scalar part of the program, as well as a set of nodal loops, or nodal code blocks. Any vector operation is translated into one or more function calls to the CM runtime system [107]. These functions invoke subroutines on the CM-5's nodes and transfer data between the partition manager and the nodes. The subroutine invoked on the nodes can be either a subroutine in the CM runtime system or a nodal code block containing a nodal loop.

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, as shown in Figure 4.4. 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 actually 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.

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 as are passed to the CM run-time system in normal executions on the CM-5.

4.6 The Goals and Structure of BENCHCVL

BENCHCVL is a benchmapping system that advances the state of the art in performance modeling by automatically generating models for an entire runtime system. To automatically create benchmaps for several computer systems requires a runtime system that is implemented on several computers. Since PERFSIM models the performance of the CM-Fortran runtime system, which is implemented only on the Connection Machines CM-2 and CM-5, I have chosen to implement a new system that models the performance of CVL [18], the runtime system supporting the NESL programming language [17]. Runtime subroutines in CVL operate on one-dimensional vectors. Implementations of CVL exist for workstations, Cray vector computers, Connection Machines CM-2 and CM-5, Maspar computers, and for other parallel and distributed platforms that support the Message Passing Interface (MPI).

Using BENCHCVL is simple. The user names, in a configuration file, the benchmap to be used for performance prediction. When the user runs a program, BENCHCVL uses the benchmap to simulate the timer of the benchmapped system. All accesses to the runtime-system timer

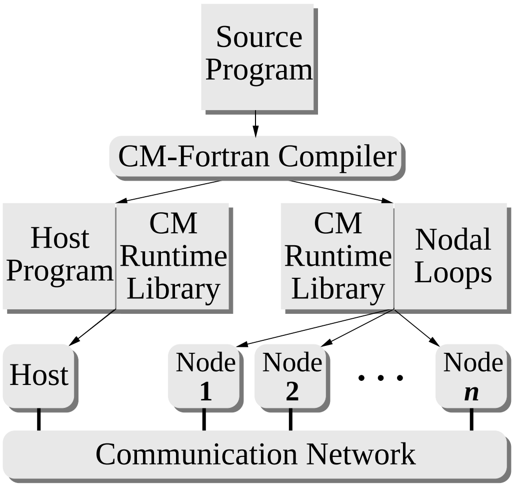


Figure 4.3 The CM Fortran compiler generates two object code files, one is a host program which contains the control structure and scalar operations in the program, and the other is a set of nodal code blocks, or nodal loops. All the vectors operations are invoked by the host program by calling the CM runtime system. The runtime system broadcasts a request to the nodes to perform the required operation, which may be an invocation of a nodal code block or an invocation of a runtime system subroutine.

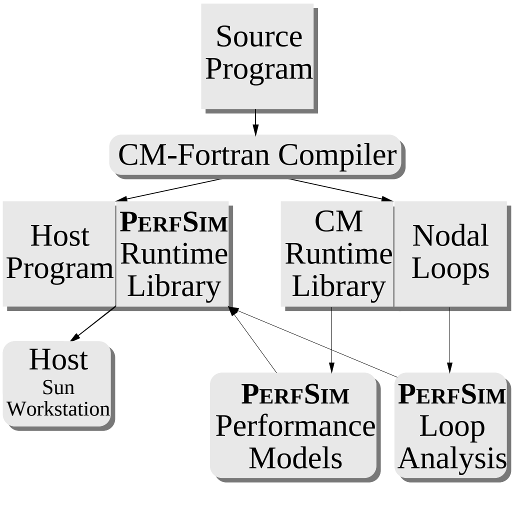


Figure 4.4 PERFSIM replaces the CM runtime system with another with the same interface. PERFSIM's runtime system never invokes operations on the nodes, and therefore the program runs on the host alone. PERFSIM's runtime system estimates the time runtime system calls would have taken on a CM-5 using performance models of the CM runtime system, and the results of an analysis of the nodal loops.

	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 4.2 The actual and predicted performance ratios 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. The table shows that BENCHCVL can accurately compare the performance of computer systems on specific programs.

by the user's program or by a profiler return the estimated time on the benchmapped system, rather than the actual time on the system that executes the program. Hence, existing programs and profilers can use BENCHCVL's predictions transparently simply by using the builtin timing facilities in CVL.

Two characteristics of NESL and CVL limit both the accuracy of BENCHCVL's models and the performance of the runtime system itself. One characteristic is the fact that NESL does not attempt to exploit locality in distributed architectures, and therefore CVL does not provide special classes of permutation operations such as shifts. Therefore most data movements in NESL are implemented by a general permutation routing mechanism whose performance depends on the contents of arrays. The second characteristic is the implementations of multidimensional arrays using segmented vectors. The arbitrary lengths of segments cause another dependence of performance on the contents of vectors, here the lengths of segments. Since BENCHCVL's models do not depend on the contents of vectors, their accuracy is somewhat limited by this design choice in CVL.

In comparison, the CM runtime system provides optimized classes of permutation operations that are easy to model, and in multidimensional arrays all segments (i.e. columns of matrices) have the same length. Thus, BENCHCVL demonstrates that benchmaps for multiple platforms can be automatically created, but PERFSIM demonstrates better the potential accuracy of benchmaps of high performance runtime systems.

4.7 BENCHCVL's Benchmaps: Coping with Caches

BENCHCVL's benchmap is based on a model of parallel computers in which a collection of nodes is connected by a communication channel. Each node has a processing unit and a local memory hierarchy, as shown in Figure 4.5. 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.

BENCHCVL's benchmaps model temporal locality in data caches using piecewise-linear models. A **piecewise-linear model** decomposes the space of performance determinants into regions and predicts performance in each region using a linear model. Consider, for example,

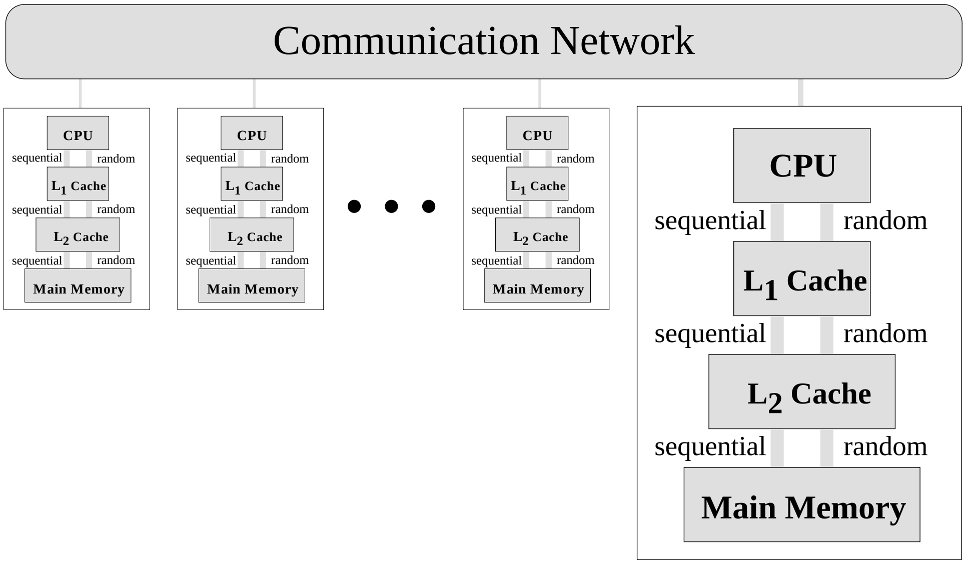


Figure 4.5 A schematic diagram of the model that underlies BENCHCVL's benchmap. The model describes the computer as a collection of nodes connected by a communication channel, where each node contains a processing unit and up to three levels of local memory. The benchmap describes the cost of interprocessor communication across the communication network, the sizes of caches, and the cost of random and sequential data transfers between levels in the local memory hierarchy.

the running time a subroutine that computes the inner product of two vectors. A piecewise-linear model can decompose the space of vector sizes into small sizes, where the input vectors fit within the cache, and large vector sizes, where the input vectors must be stored in main memory.

The CVL runtime library implements operations entire and segmented one-dimensional vectors. A segmented vector is partitioned into segments of arbitrary lengths. Vector operations in CVL include elementwise 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 upon 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.

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.
- 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 nonsegmented 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 diameter of the communication network, the third the number of elements each processor owns, and the last the expected number of elements each processor has to send and receive from other processors. The number of messages used by the model is an approximation for the expected number for 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 sequential accesses where spatial locality is guaranteed and for random accesses. 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 to 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.

The temporal locality of data accesses is modeled using piecewise-linear models. BENCHCVL assumes that vectors that fit within the cache are indeed in the cache. A piecewise-linear model use separate linear models to describe the performance of a subroutine when argument vectors fit in 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 4.6 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

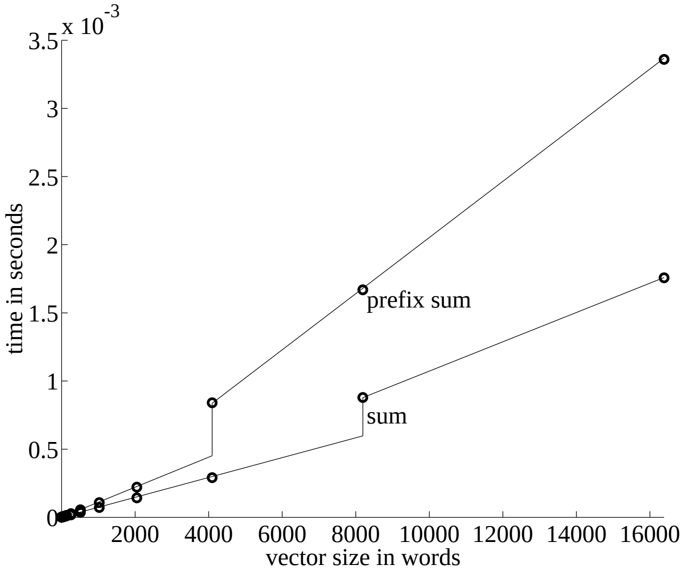


Figure 4.6 The execution times of a summation function and a prefix sum function, represented by circles, on a Sun SPARCstation 10 workstation. The lines represent the models estimated from these data points. The running time of the two functions is clearly not linear, and the breakpoints occur on different input sizes, because one function handles only one vector, while the other handles two.

enable BENCHCVL to search for only one breakpoint in the running time, and ensures a robust estimation of cache sizes.

BENCHCVL's algorithms for finding piecewise-linear models are implemented as a CARTOGRAPHER module. The module can use any linear estimation algorithms that are available in CARTOGRAPHER.

4.8 BENCHCVL's Software Organization

Portability of NESL programs is achieved by porting the CVL runtime system [19]. Data-parallel NESL programs are translated by the compiler into a representation called VCODE, for vector code. The VCODE representation is interpreted by a VCODE interpreter. The VCODE interpreter uses the CVL runtime system to perform operations on vectors, the only data structures in NESL. Only the CVL runtime system needs to be implemented when porting NESL to a new platform—the other components of the system are identical on all platforms. (In addition, there exists a VCODE compiler that does not call CVL, which I did not use.)

The BENCHCVL system replaces all the CVL calls in the VCODE interpreter with calls to an instrumented interface to CVL, as illustrated in Figure 4.7. Subroutines in BENCHCVL's interface are capable of timing CVL subroutines and estimating their running time using a benchmap. The instrumented interface is portable and makes no assumptions about any particular CVL implementation. As described in Section 4.6, BENCHCVL can replace the interface to the native system timer provided by CVL by an identical interface to a simulated timer. The simulated timer is advanced by the estimated running time of subroutines used by the calling program, typically the VCODE interpreter. The instrumented interface and CVL itself are also used by the BENCHCVL's surveyor program that generates performance reference

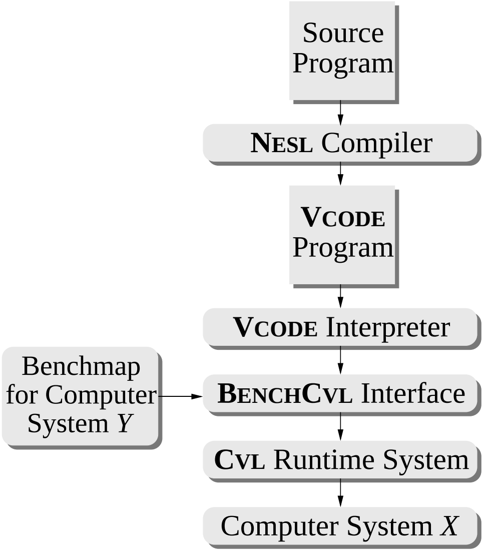


Figure 4.7 NESL programs are translated into VCODE programs. The VCODE interpreter executes vector instructions by calling the CVL runtime system. In BENCHCVL, the VCODE interpreter calls the BENCHCVL instrumented interface. BENCHCVL's interface calls CVL and can measure or estimate the running time of CVL subroutines. The estimates are based on a benchmap.

points for benchmapping.

Chapter 5

Assessment of Performance Models

Three factors may compromise the accuracy of performance models: missing performance determinants, models with inappropriate structure (such as missing basis functions), and insufficient reference points to estimate parameters in models. The accuracy of models should therefore be evaluated before they are used, to determine whether their accuracy meets our expectations. Expectations from models should depend on the context. In some cases very accurate models are required, and in other cases rough models suffice. This chapter demonstrates, in context, that PERFSIM and BENCHCVL are accurate.

5.1 Accuracy in Context

How accurate does a benchmap need to be? We cannot expect performance models to predict performance exactly. Which prediction errors are acceptable depends on the quantitative context in which the predictions are used. Consider for example two implementations of an algorithm, one of which runs on a given input in 1 second, whereas the other takes in 2 seconds to run. Performance models whose predictions are within 25% of the actual running time can correctly determine the faster subroutine for the given input, while models whose predictions are within 50% might err. In this case, the difference in running times provides a context in which models can be evaluated. This section describes the quantitative contexts in which the accuracy of PERFSIM and the accuracy of BENCHCVL are evaluated.

PERFSIM is a profiler for CM Fortran programs, so the yardstick to which it is compared is Prism, the CM-5's native profiler [111]. Prism uses the CM-5's timers to profile programs. I have found that the CM-5's timers can have large deviations when timing the same program several times, even though they are supposed to be aware of time sharing.

Table 5.1 shows that the deviations in running times reported by the CM-5's timer can be larger than PERFSIM's estimation errors. On large input sizes, where performance is most important, PERFSIMis especially accurate because its models were tuned for large input sizes, whereas deviations in reported timings are very large.

Finding a yardstick for for assessing BENCHCVL is more difficult, because benchmarks are interpreted by humans. As we have seen in Chapter 2 the performance of multiple programs in one benchmark, such as the NAS Parallel Benchmark, can differ by more than a factor of 7. (Here performance is taken as running time compared to the running time on a reference

		Elapsed Time		Max Dev	Est Error
Prog	Size	CM-5	PERFSIM	CM-5	PERFSIM
CG3D	192^3	338	8	44%	-19%
	16^3	17	8	-5%	0%
QCD	36×16^3	596	20	1%	-4%
	36×4^3	36	20	19%	-30%

Table 5.1 A comparison of the speed and accuracy of PERFSIM versus actual performance measurements on the CM-5. The table reports the maximum relative deviation from the mean running time among 10 executions of two programs on the CM-5, and PERFSIM's relative estimation error. The deviation from the mean running time in timings reported by the CM-5's elapsed timers can be larger than PERFSIM's estimation errors. The table also shows that profiling the programs with PERFSIM is much faster than running them on the CM-5. Time is reported in seconds.

machine). It is not clear how accurate can a prediction derived from these benchmarks be.

5.2 The Role of Test Suites

This section presents guidelines for the use of test suites for accuracy evaluations, and explains why it is difficult in practice to adhere to these guidelines. The accuracy of models is evaluated using **test suites**. The programs in the test suite should not be used to estimate parameters in models. Indeed, the programs in the test suites used to evaluate the accuracy of benchmaps in this thesis were not used to estimate parameters in models. Ideally, these programs should not be known at all to the person or team implementing the models. The size of the test suite should be large enough to ensure a reasonable level of confidence in the evaluation. This section examines the issue in more detail and explains why these simple objectives are often difficult to achieve in practice.

The most important objective of accuracy evaluation is to eliminate feedback between the test suite and the models. If test programs were plentiful, we could set aside a number of them for the final evaluation, and use other groups for intermediate evaluations of benchmaps. After each evaluation, we would improve the models or add reference points and use the next group to evaluate the models. After the final evaluation, no further modifications are allowed. If the models are inaccurate at this point, they are either not used at all, or delivered to users with an appropriate warning.

Having one set of evaluation programs for all models on all systems is desirable, because it allows us to compare the accuracy of models relative to one another. But if the models must evolve when new architectures arise, then using the same evaluation programs introduces feedback into the evaluation process, because the models are modified after the behavior (in terms of the subroutine calls performed) of the evaluation programs is known.

Unfortunately, it is difficult to eliminate feedback in the benchmap evaluation process because the number of programs available for testing is often small. Users are often reluctant to give away their programs and their input data sets, even for performance evaluation purposes, since they may encapsulate important research results or they may have considerable financial

Authors and Reference	Progs	Nature and Author of Programs
Atapattu and Gannon [5]	6	Lawrence Livermore Loop Kernels.
Balasundaram et al [9]	1	2D red-black relaxation, by the authors.
Brewer [22, 23]	3	2D stencil, 2 sorting algorithms, all by the author.
Crovella and LeBlanc [34]	2	Subgraph isomorphism, 2D FFT, latter by Jaspal Subhlok.
Fahringer and Zima [43]	"many"	Includes Jacobi relaxation, matrix multiplication, Gauss-Jordan, LU, authors not specified.
Formella et al [47]	3	Dense conjugate gradient and 1D and 2D stencils, by the authors.
MacDonald [79]	6	Lawrence Livermore Loop Kernels.

Table 5.2 The number of test programs used to evaluate performance models in various papers. In some of the papers programs were investigated in detail and executed on many input sizes. For comparison, I used 8 programs to assess the accuracy of PERFSIM, one of which was written by me.

value. For example, the implementations of the NAS Parallel Benchmarks could have been good test programs, but vendors do not release them.

Many programs must run in some context which includes input data, other programs, and user interaction. It may be difficult to duplicate a program's environment, which leads to the elimination of many programs from the test suite.

Limited use of a programming language may be another limiting factor. For the BENCHCVL system, were we use the example programs distributed with the NESL language, plus two other programs, the number of available programs is particularly small.

Table 5.2 illustrates how hard it is to get a large number of test programs by listing the number of test programs used in a number of research papers describing performance models.

5.3 PERFSIM's Accuracy

PERFSIM is accurate. As an illustrative example, Figure 5.2 compares the actual and estimated running times of three implementations of a simple algorithm for various problem sizes on a log-log scale. The figure clearly shows that the estimates are accurate enough to order the relative performance of the three subroutines for nearly all problem sizes. The three implementations, described in Figure 5.1, implement a one-dimensional red-black relaxation algorithm. The implementations differ in how they represent the red (even) and black (odd) points in the domain. The three representations are: (1) logical masks, (2) array sections, and (3) separate arrays to store the red and black parts of the domain.

Since PERFSIM's benchmap is based on the organization of the CM-5's hardware, incorrect model structure is not a major concern. Because the main objective of PERFSIM is to predict the performance of programs accurately on large input sizes, the sufficiency of the training set is not a major concern either. Thus, we focus on the impact of missing performance determinants on PERFSIM's accuracy. The empirical results in Table 4.1 in Chapter 4 indicate that that PERFSIM's models are accurate and that the impact of unknown performance determinants on

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 5.1 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.

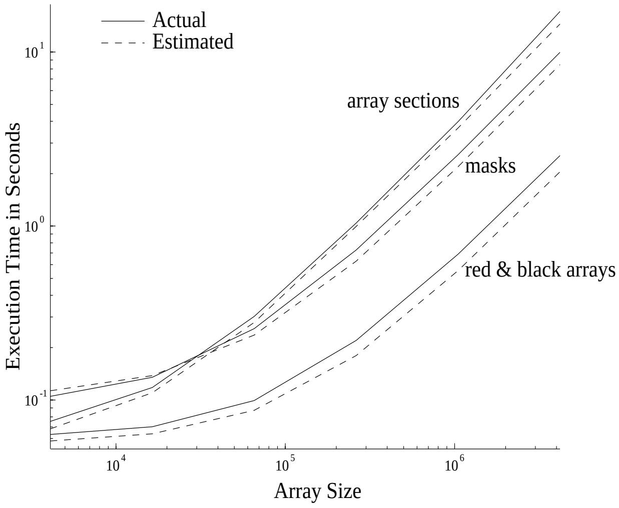


Figure 5.2 The the actual and estimated running times of three red-black relaxation subroutines as a function of the domain size. PerfSim can generate the entire graphs describing the estimated running times from two executions on arrays of size 4096 and 8192, both well to the left of the crossover point.

PERFSIM's accuracy does not significantly limit its application.

Four categories of missing performance determinants, however, compromise PERFSIM's accuracy. We now describe these categories, and explain which of these determinants should be used in future benchmaps and how.

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 $n = 16$ million 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 situations. 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 fails 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 so that the compiler can replace the call to the irregular-communication subroutine by a call to a regular-communication subroutine, whose performance is always better.

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. I could not implement PERFSIM in this manner since I 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 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 1 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 0.196 seconds, a slowdown of more than 4.8% over the 0.184 seconds it took to shift the array by 1. The slowdown is due to a bug, because there is no reason that two operations that are completely symmetric would run in different amounts of time.

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 often manifest themselves as discrepancies between estimated and actual running times, as discussed in Chapter 2.

5.4 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 model 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: Sun SPARCstation 10, Sun SPARCstation $1+$, a 32-node Connection Machine CM-5, and a Cray C90 (CVL uses only one processor on a 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. Tables 5.3, 5.4, 5.5, and 5.6 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 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 5.7 shows that the predictions can be effectively used to compare the performance of computer systems on specific programs. Such comparisons are more meaningful to the users of the programs being compared than comparisons based on benchmark programs.

The relative errors of the models on the reference points 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 involv-

Program	Input	Actual		Predicted
		Total Time	CVL Time	CVL Time
Geometric Separator	airfoil, 12K vertices	5.49	1.67	1.42
Spectral Separator	airfoil, 12K vertices	74.68	64.50	76.56
Convex Hull	100K points	9.06	6.90	8.48
Conjugate Gradient	78K nonzeros	88.07	83.41	110.81
Barnes Hut	4K points	269.48	175.65	193.59

Table 5.3 Measured and predicted running times of NESL programs on a Sun SPARCstation 10.

Program	Input	Actual		Predicted
		Total Time	CVL Time	CVL Time
Geometric Separator	airfoil, 12K vertices	15.88	5.52	6.71
Spectral Separator	airfoil, 12K vertices	268.06	270.81	361.07
Convex Hull	100K points	35.42	29.12	37.19
Conjugate Gradient	78K nonzeros	339.50	326.20	451.98
Barnes Hut	4K points	917.96	652.08	655.05

Table 5.4 Measured and predicted running times of NESL programs on a Sun SPARCstation 1+.

Program	Input	Actual		Predicted
		Total Time	CVL Time	CVL Time
Spectral Separator	airfoil, 12K vertices	28.93	19.92	17.03
Conjugate Gradient	78K nonzeros	15.09	6.07	7.75

Table 5.5 Measured and predicted running times of NESL programs on a 32-node time-shared CM-5. The CM-5 implementation of CVL could not run the other test programs due to bugs.

Program	Input	Actual		Predicted
		Total Time	CVL Time	CVL Time
Convex Hull	100K points	0.703	0.134	0.109
Conjugate Gradient	78K nonzeros	5.38	2.15	1.86

Table 5.6 Measured and predicted running times of NESL programs on a Cray C90. One test program could not be executed because of a problem in the C90 implementation of CVL and two other because of an apparent bug in BenchCvl.

	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 5.7 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. This is a reproduction of Table 4.2.

ing 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 very large, because the models did the sequential bottleneck in the processor which owns the C array. This problem is easily fixed, but it may be typical of latent bugs 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 the reference points probably include some outliers whose timing 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 these conflicts cannot be ignored.)

The BenchCvl system 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.

Chapter 6

Performance Prediction: Methods and Applications

Now that we have explained what is benchmapping and demonstrated that it is a feasible performance-prediction methodology, it is time to readdress two questions: what are the applications of benchmapping and in what ways is it superior to other performance prediction methodologies. We shall see that benchmapping has applications beyond those described in Chapter 4. We shall also see that traditional performance specification and prediction methodologies are not suitable for most of these applications.

6.1 Applications of Benchmaps

By examining my own experiences with benchmapping systems, as well as other research on performance modeling, this section demonstrates that benchmapping is an enabling technology for a variety of applications. We focus on six application areas: program profiling and tuning, acquisition decisions, performance maintenance, hardware design, compiler optimization, and runtime optimization. Benchmapping contributes to these application areas by supplying decision-making processes with quantitative performance estimates.

Profiling and Tuning

PERFSIM is a profiler for CM Fortran programs that runs on a workstation and estimates the profile of the execution of a program on the Connection Machine CM-5 [110] quicker than the profile can be produced by measuring the program on a CM-5. PERFSIM is fast because it does not actually perform vector operations, only estimates their running time. Chapter 4 describes PERFSIM and its benchmap of the CM-5's data-parallel runtime system. PERFSIM's estimation errors are comparable to deviations in the measured running time of programs on the CM-5. Hence, PERFSIM is about as reliable as the native CM-5 profiler that uses timers to profile programs. Chapter 5 presented a detailed study of PERFSIM's accuracy that supports this conclusion.

Modeling rather than measuring performance can bring several benefits to a profiler. If estimating the running time of an operation is faster than performing and timing it, then a profiler like PERFSIM that does not perform all the operations in a program can be significantly faster than a timing-based profiler. When timers are noisy or have low resolution, models can be more accurate and provide more details than timing-based approaches. Finally, models provide insights into the cost structure of operations, insights that can enable activities such as performance extrapolation.

Several other research groups have also proposed to use performance models in interactive program tuning tools. Atapattu and Gannon [5] describe an interactive performance prediction tool for 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. In comparison, performance prediction in PERFSIM is based mostly on a model of the runtime system, and static analysis is avoided by executing the control structure of the program. Crovella and LeBlanc [34] 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. To use their tool, a user must run the program several time on the target machine. To use PERFSIM the user does not have to run the program on the CM-5.

System Acquisition and Configuration

Benchmaps are effective tools for comparing the performance of several computer systems or system configurations. Benchmaps can compare the performance of several systems by predicting the performance of programs that prospective buyers want to use. Comparisons based on predicting the performance of users programs are more relevant for acquisition decisions than comparisons based on standardized benchmarks.

BENCHCVL predicts the running time of data-parallel programs written in the NESL language. The NESL language is implemented on a wide range of serial, parallel, and vector computers, and the BENCHCVL system can automatically generate a benchmap for any NESL platform. Given the benchmap of a NESL platform, NESL programs running on another platform can predict their running time on the benchmapped platform. Users compare computer systems with the BENCHCVLsystem by predicting the performance of their own programs on any system on which the NESL programming language is implemented.

This approach to acquisition and configuration decision making is already used by the industry in the area of real-time embedded computer systems. A firm called *JRS Research Laboratories Inc.* from Orange, California, offers a design automation environment for embedded systems, which includes performance models of hardware and software components and allows designers to estimate the performance of system designs using various hardware platforms and configurations. It was not possible to understand from the information supplied about this product how it works.

Chen [28] and Chen and Patterson [29] propose a benchmarking system for I/O performance. Their system creates simple models of I/O performance using an automatic performance surveying technique. The models can be used by a human performance analyst to predict the I/O performance of application programs.

Acquisition and configuration decisions concerning time-shared resources, such as CPU's, communication networks, and disks, often depend on external performance measures, such as response time. External performance measures have been modeled extensively using *queuing models*. Such models were developed for the early timesharing systems of the early sixties [38, 101], and they are still in use today [60, 63, 72].

Performance Maintenance

The performance of software is often compromised by **performance bugs**, which are flaws that have no effect on the correctness of the software, but have an adverse effect on performance. Performance maintenance is the activity in which performance bugs are uncovered and fixed. A key component in performance maintenance is a set of explicit **performance expectations**. Performance bugs are uncovered when the performance of the software does not live up to our expectations, or then the expectations are low compared to the capabilities of the hardware. Perl [94] and Perl and Weihl [93] propose a system for checking actual performance against expectations, but they do not propose an automated way for generating the expectations.

Benchmaps are capable of generating expectations. The predicted performance of a subroutine is an expectation. If benchmaps can automatically predict performance, then they can automatically generate expectations for performance maintenance. When the cost structure of the models in a benchmap has some physical interpretation, the models can inspected by hand to verify that they are in line with the hardware's capabilities. The manual inspection may be tedious, but it is certainly less tedious than inspecting the source code and searching for performance bugs.

Performance-testing methodologies should include generation of performance expectations as well as an execution of a test suite of programs. During the execution of the test suit, the performance of the software under test is compared to the expectations. If the discrepancy is larger than some threshold, the discrepancy is reported, and a search for a bug can be undertaken. Benchmaps can automate the testing process by generating an expectation for the performance of every runtime-subroutine call in the test suite and comparing the expectation with the actual performance.

The two modeling tools described in this thesis uncovered several performance bugs in runtime systems using this methodology, even though the original intended use was not performance maintenance. But no matter what the purpose of the model is, every comparison of prediction and measurement is an opportunity to uncover performance bugs.

For example, by comparing PERFSIM's models for the `cshift` and the `eoshift` subroutines in one version of the CM-5 runtime system[1], we discovered that even though the subroutines are similar in functionality, the `eoshift` subroutine was about 4 times slower, which was due to a performance bug. Large discrepancies between predicted and measured performance of subroutines in the BENCHCVL system led us to discover a serious performance bug in the operating system[2] of the SGI Indigo2 workstation. This performance problem, whose manifestation is shown in Figure 2.3, is caused by mapping contiguous blocks of virtual memory to contiguous physical blocks. This mapping policy causes conflicts in virtual memory to translate into cache misses in the off-chip physically indexed cache.

[1] The version of the CM-5 runtime system dated 8/31/93.

[2] IRIX Release 5.3 IP22

Hardware Design

High performance is one of the main goals of computer design. Designing computers requires that the performance of various alternatives be predicted. Hennessy and Patterson [57, page 70] write that "because accurately predicting performance is so difficult, the folklore of computer design is filled with suggested shortcuts."

Modeling techniques described in this thesis may be able to perform such predictions, without taking shortcuts. In particular, this thesis describes models of runtime systems which can predict the running time of application programs. When the architect and runtime system designer can translate the performance of the hardware to the performance of the runtime system services, tools such as PERFSIM and BENCHCVL can predict the running time of applications. Such tools can accelerate the design process by providing a fast alternative to simulators. While simulators are still clearly necessary for hardware design, it is possible that some of the performance predictions can be performed using higher-level modeling tools which are much faster.

The main difference between benchmapping an existing system and benchmapping a computer system that is being designed is that parameters in the benchmap cannot be estimated from experiments. The benchmap's parameters should be estimated in such cases either by manual analysis of the design or using experiments on a simulator.

Compiler Optimizations

Most compiler optimizations are in essence choices between several ways of performing a set of operations. the optimizer tries to choose the fastest way. A great deal of research [27, 67, 75] has been done for example in the area of optimizing data distributions, alignment, and redistributions in data parallel languages such as High Performance Fortran [68, 59]. While some of that research uses fairly crude cost models, some of the research on performance modeling for high performance computers was done specifically for guiding compiler optimizations. In particular, Balasundaram et al. [9] describe a rather limited study of this idea, while Fahringer and Zima [43] describe a comprehensive solution within the framework of the Vienna Fortran Compilation System.

Runtime Optimizations

When a runtime system provides services which take a long time to complete, or services which are likely to be called repeatedly with similar arguments, it makes sense for the runtime system to spend some time finding the most economical way of providing the service. If the decision is made quickly relative to the time the service is likely to take, the decision making process does not impact the total service time. Even if the decision making is expensive, it can sometimes be amortized over many calls with similar arguments, because the same decision is made.

Brewer [22, 23] proposes a runtime system with built-in performance models that chooses the best implementation of a service, such as sorting, from several available implementations. The CM-Fortran [109] runtime system uses a similar idea: the most appropriate implementation of a subroutine such as `cshift` is chosen from several alternatives. The decision, which can

	EP	MG	CG	FT	IS	LU	SP	BT
CM-5E/CM-5	.04	4.6		2.3	7.2	2.6	1.9	2.5
T3D/C90	5.3	3.1	.85	3.3	1.2	2.7	3.3	2.9

Table 6.1 The ratios between the performance of the Connection Machine CM-5 and the Connection Machine CM-5E (both with 128 processors), and between performance of a 64-processors Cray T3D and a single-processor Cray C90, as reported by the NAS Parallel Benchmarks, Class A [8]. (Performance on the CM-5 is not reported for the CG benchmark.) The variation among the ratios makes it difficult to assess the performance of another application, even if it is a fluid dynamics application.

take significantly more time to make than the execution time of the `cshift` subroutine itself, is cached, and used on subsequent calls with the same arguments.

6.2 Comparison with Other Performance Specification Methodologies

The benchmapping methodology has three advantages over traditional performance specification methodologies: it is automated, accurate, and fast.

Traditionally, the performance of computers has been specified by **hardware characteristics**. Clock rate, number of functional units, cache size, and network bandwidth are some of the favorites. Computers are complex systems with many interacting parts. If the characteristics of even a single part are not specified by the vendor, determining performance becomes difficult. For example, a low-bandwidth interface between processors and a communication network may limit the effective bandwidth of that network. When the bandwidth of network is specified but the bandwidth of the interface is not, it becomes difficult to assess performance. Even if the performance of every component is fully specified, which is almost never the case, interaction between components may limit performance in ways that are difficult to predict.

A computer benchmark is a program or algorithm whose execution time on various computers is reported by the vendors or by third parties. In some cases, the benchmark is a fixed program, but in other cases, only an algorithm is given and the benchmarker may implement the algorithm in the most efficient way subject to reasonable restrictions. These two classes of benchmarks exist because some users would like to run their programs unchanged on new computers, whereas others are willing to tune their codes for a new platform. The hope is that by measuring the execution times of programs which are similar to the programs the prospective user wants to run, the execution time of the user's programs can be estimated without programming, debugging, and tuning the entire application or application suite.

Unfortunately, the user's program may be dissimilar to the benchmark programs. Even when the user's program is similar to a combination of some of the benchmark programs, determining its running time may be difficult. Table 6.1 describes performance ratios between pairs of computer systems (e.g., a CM-5 and a CM-5E), as reported by the NAS Parallel Benchmarks [8], a set of algorithmic benchmarks designed specifically to assess the performance of fluid dynamics applications. The ratios vary considerably from one problem to another. Consequently, a user who has a CM-5 application, for example, will find it difficult to predict the performance of the application on the CM-5E. The difficulty remains even for users of fluid dynamics applications. Weicker [123] describes some of the expertise required to relate the results of several popular benchmarks to one's own programs.

In short, performance prediction with hardware descriptions and benchmarks is difficult because of two reasons. Both specify the performance of a complex system with just a few numbers. The specification is therefore often incomplete. Consequently, accurately predicting the performance of programs using these specification methods is at best difficult. There is no standard way to decompose a program into components whose performance can be accurately predicted with either hardware descriptions or benchmarks. As a result, performance prediction with benchmarks and hardware characteristics is not automated.

Benchmaps predict performance faster than simulators. Almost every new computer designed is implemented in software before it is implemented in hardware (see [66] for one popular account). The software implementation is called a **simulator**. It consists of a detailed model of the hardware design. The level of detail cause most simulators to be slow, often around 10,000 times or more slower than the hardware implementation. They can potentially predict performance accurately; the simulator of the MIT Alewife machine for example, whose main goals were not accurate performance predictions, predicts performance to within 10% [26].

Because benchmaps model the performance of a high-level interface, the runtime system, they can predict performance much faster than simulators. Their speed makes benchmaps suitable for applications that simulators are too slow for, such as acquisition decision making and profiling. Benchmaps and simulators can be used together. A simulator can be used to survey the performance of a computer that is not yet implemented, and the benchmap generated from the survey can predict the performance of programs.

Part II

Creating Locality in Numerical Algorithms

Chapter 7

Locality in Iterative Numerical Algorithms

7.1 Introduction

This chapter sketches three algorithmic techniques that increase the temporal locality in many numerical algorithms and improve their performance. Chapters 8 through 10 use one of the techniques to reduce the number of I/O operations performed by three classes of algorithms when their data structures do not fit within main memory. Implementations of this technique on workstations, presented in Chapters 9 and 10, outperform conventional implementations by factors of up to 5. A second technique is used in Chapter 11 to accelerate the solution of certain systems of linear equations on the CM-5 parallel computer by a factor of 1.5–2.5.

Many numerical algorithms are **iterative**, in the sense that they consist of repeated updates to a state vector x of size n,

$$x^{(t)} = F^{(t)}(x^{(t-1)}) \; . \tag{7.1}$$

Two important applications of iterative algorithms are discrete time-stepping in simulations of physical processes, where $x^{(t)}$ represents the state of a system at a certain time, and solution of systems of equations by iterative improvement, where the sequence $\left\langle x^{(0)}, x^{(1)}, \ldots, x^{(t)}, \ldots \right\rangle$ converges to a solution of the system. This part of my thesis is mainly concerned with numerical algorithms whose goal is to compute, given an initial state $x^{(0)}$ and a number T, the final state $x^{(T)}$. In this chapter we focus on update operators $F^{(t)}$ that require relatively little computation, say linear in the size of x.

Most computers have hierarchical memory systems, where a small portion of the data is stored in fast and expensive memory called **primary** memory, and the rest is stored in a larger, slower, and less expensive memory, called **secondary** memory. Data is transferred between secondary and primary memory using a communication channel whose bandwidth is often much smaller than the bandwidth between the processing unit and the primary memory. Many systems have more than two levels of memory. Some parallel computers have distributed memory, where memory is attached to individual processors, and the processors communicate using some communication medium. In such systems, the bandwidth between a processor and its local memory is much larger than the bandwidth between a processor and memory attached to other processors.

Since we are normally interested only in the final state vector $x^{(T)}$, we can reuse the state vector storage. Besides the space required for representing $F^{(t)}$, the total amount of storage required for the entire T-step computation is therefore $\Theta(n)$ to store the state vector. A computer with ample primary memory can perform an iterative computation by simply updating the state vector according to Equation (7.1).

In this thesis we ignore the costs associated with reading or computing the operators $F^{(t)}$, which are often minor. Furthermore, in this chapter we assume for simplicity that the amount of work required to apply $F^{(t)}$ is $\Theta(n)$. This assumption is relaxed in subsequent chapters.

If the computation does not fit within the primary memory of the computer, however, a so-called "**out-of-core**" method must be used. An out-of-core algorithm tries to be computationally efficient but in addition attempts to move as little data as possible between the computer's primary memory and secondary memories. The naive method, which is inefficient, repeatedly applies Equation (7.1), computing the entire state vector for one time step before proceeding to the next. This strategy causes $\Omega(Tn)$ words to be transferred (**I/O**'s) for a T-step computation, if $n \geq 2M$, where M is the size of primary memory. In other words, each word transferred from secondary to primary memory is used a constant number of times before it is written back or erased. This limits the level of utilization of the processor, since data is transferred between primary and secondary memories at a much lower rate than the rate at which the processor operates on data in primary memory.

The key to improving the processor's utilization, and hence reducing the solution time, is to increase the **temporal locality** in the algorithm, or the number of times a datum is used before it is erased from primary memory. Section 7.2 presents an overview of the results in this part of the thesis. Section 7.3 discusses the issues of numerical stability and performance of out-of-core algorithms. Section 7.4 describes three techniques for increasing the temporal locality in iterative numerical algorithms, and Section 7.5 presents a lower bound on the number of I/O's performed by one of the techniques. We end this chapter in Section 7.6 with a historical perspective on out-of-core numerical methods.

7.2 An Overview of the Results

The thesis presents out-of-core methods for two classes of algorithms: linear relaxation algorithms and Krylov-subspace algorithms. Given a matrix A, an n-vector x, and a positive integer T, linear relaxation algorithms compute the sequence $\langle x, Ax, A^2x, \ldots, A^Tx\rangle$. Such algorithms are used for solving linear systems of equations by relaxation and for simulating time-dependent physical processes. The matrix A is typically sparse and usually has some special structure that can be exploited by the linear relaxation algorithm. Krylov-subspace algorithms solve systems of linear equations and eigenproblems. These algorithms use the input matrix A in only one way: multiplying vectors by the matrix (and sometimes by its traspose as well). They are thus suitable for instances in which the representation of A is only a matrix-vector multiplication subroutine.

A well known technique can be used to implement out-of-core linear relaxation when the sparcity structure of A corresponds to a low dimensional mesh (regular or irregular). The method will be demonstrated in Section 7.4 on regular two-dimensional meshes. Chapter 8 describes out-of-core linear relaxation algorithms for multigrid matrices A. The sparsity structure of such matrices largely corresponds to that of a low dimensional mesh, but it has a small number of nonlocal connections in the mesh. Matrices arising from the Fast Multipole Method (FMM) have a similar structure and can be handled by the algorithm as well. Relaxation algorithms with such matrices arise in implicit time-stepping simulations in which a multigrid algorithm is used as the implicit solver (see [20]), as well as in Krylov-subspace solutions of integral equations where an FMM algorithm is used as a matrix-vector multiplication subroutine [98]. Chapter 9 presents an out-of-core linear relaxation algorithm for matrices A which are a product $A = T_1^{-1}T_2$ of a tridiagonal matrix T_2 and the inverse of another tridiagonal matrix T_1. Relaxation algorithms with such matrices are used in implicit time-stepping schemes for simulating time-dependent physical processes in one spatial domain (see, for example, [91] or [104]).

Chapter 10 presents a method for implementing out-of-core Krylov-subspace algorithms. The method assumes that an efficient out-of-core linear relaxation algorithm is available for generating Krylov subspaces. Any of the algorithms mentioned in the previous paragraph can be used to generate Krylov subspaces. Given an out-of-core linear relaxation subroutine, the method implements Krylov-subspace algorithms efficiently out-of-core. This method can thus be used to implement out-of-core Krylov-subspace algorithms for sparse matrices arising from meshes, from multigrid and FMM methods.

The algorithm presented in Chapter 11 presents an efficient parallel preconditioner for elliptic problems discretized on a two-dimensional mesh. It is not an out-of-core algorithm, but it does create locality, as explained in Section 7.4.

7.3 Practical Considerations: Numerical Stability and Performance

I claim that the out-of-core algorithms presented in this part of the thesis solve problems faster than conventional algorithms. To demonstrate that they indeed solve the problems that they were designed to solve, I show that they are equivalent to conventional algorithms in exact arithmetic and numerically stable in floating point arithmetic. To demonstrate that they are faster, I prove that my algorithms have better asymptotic behavior than conventional algorithms, and I show that they run several times faster on workstations. They run faster because they perform much less I/O and not much more work than conventional algorithms.

Some of the out-of-core algorithms described in this thesis are equivalent to the naive algorithm in exact arithmetic, but are not equivalent in floating-point arithmetic. In time-stepping algorithms that extrapolate a state vector over time, it is very important not to introduce excessive noise in the form of rounding errors, because once present, there is no way to get rid of the errors. Hence, in Chapter 9 where we describe a time-stepping algorithm, we compare the deviation of the results returned by our algorithm from the results returned by the naive algorithm. In iterative solvers, it is not important for the iterates produced by the out-of-core algorithm to be very close to the iterations produced by the naive algorithm. What is important is for the iterates to approach the solution vector at roughly the same rate. Therefore, in Chapter 10, where we describe our-of-core Krylov-subspace solvers, we assess their stability by examining their convergence rates.

The asymptotic performance of out-of-core algorithms is usually specified by how much they reduce the number of I/O's, compared with the naive algorithm, as a function of the primary memory size. For example, we describe in Chapter 9 an out-of-core algorithm that performs a constant factor more work than the naive algorithm, but performs only $\Theta(1/\sqrt{M})$ I/O operations per unit of work, a factor of $\Theta(\sqrt{M})$ less than the naive algorithm. But whether an algorithm delivers good enough performance in practice depends not only on how much faster it is than a naive algorithm executed out of core, but also on how much *slower* it is than a good algorithm executed in core. Many users can choose not to use an out-of-core algorithm at all. Some can avoid solving the problem at hand, while others can solve their problem on a machine with a larger primary memory or buy more memory.

It is not usually necessary to increase locality in an algorithm that spends only 10% of its running time performing I/O, since the reduction in the running time can be at most 10%. Increasing the locality in such an algorithm may even increase its overall running time, since many of the techniques that increase locality also increase the amount of work the algorithm performs.

Changes in computer technology may render certain out-of-core algorithms, which today do not seem practical, more attractive. Two such technological trends are the increasing capacities of memories at all levels of the memory hierarchy, and the increasing disparity between bandwidths of communication channels at different levels of the memory hierarchy. For example, the bandwidth between the register file and the on-chip cache of microprocessors grows faster than the bandwidth between the on-chip cache and main memory, which grows still faster than the bandwidth of the I/O channel to which disks are connected.

A larger primary memory means larger reductions in the number of I/O's required per unit of work in out-of-core algorithms, as will become clear by the analysis of the algorithms. Relatively slower secondary memories mean that the naive algorithms perform poorer when executed out-of-core, whereas specialized algorithms that perform less I/O are impacted less. On the other hand, larger primary memories also mean that problems that cannot be solved in core today may fit into primary memory in the future (but people seem to want to solve ever larger problems), and slower secondary memories mean that any algorithm which performs I/O, even specialized and highly tuned ones, run slower with lower processor utilization.

7.4 Creating Locality

This section presents three algorithmic techniques for creating locality in iterative numerical algorithms so that they perform better in out-of-core executions. Which of the three techniques applies to a given problem depends on the structure of the update operator $F^{(t)}$. The first technique, which dates back to at least 1963 [95], uses a **cover** of the graph associated with $F^{(t)}$. The technique works well when $F^{(t)}$ is a sparse, local operator in which $x_v^{(t)}$ depends on only a few elements of $x^{(t-1)}$, which we call the "neighbors" of v. Another technique, which we call **local densification**, applies to roughly the same class of update operators. It works by replacing $F^{(t)}$ by another update operator $G^{(t)}$, which requires the same amount of I/O per iteration as $F^{(t)}$, but which yields the solution to the numerical problem in fewer iterations. The technique also has important applications on computers with distributed memory. Although the basic tools used in this technique are well known, the insight that these tools can be used to improve the locality in algorithms only dates back to the 1980's [100]. In 1993 Leiserson, Rao, and the author introduced a third technique, based on **blocking covers**, which may be applicable when the action of the update operator is sparse but global.

Creating Locality Using Covers

The covering technique for executing algorithms out-of-core is based on a simple idea. We load a subset of the initial state vector $x^{(0)}$ into primary memory. If the state of some variables in primary memory can be updated without referencing variables which are not in primary memory, we do so. If we can now update some of them again only using references to elements of $x^{(1)}$ which are in primary memory, we do so. We continue for some τ steps, then write back to secondary memory the elements of $x^{(\tau)}$ we have computed, and load another subset of $x^{(0)}$. The question is, which elements of $x^{(0)}$ should be loaded? This section answers this question for a simple update operator, and discusses some general aspects of this technique.

We demonstrate the covering technique using linear relaxation on a $\sqrt{n}$-by-$\sqrt{n}$ mesh that uses only $\Theta(Tn/\sqrt{M})$ I/O's, where the primary memory has size M. That is, the state vector x represents values on the vertices of a mesh, and each iteration involves updating every mesh point by a linear combination of its neighbors. The idea is illustrated in Figure 7.1. We load into primary memory the initial state of a k-by-k submesh S, where $k \leq \sqrt{M}$ is a value to be determined. With this information in primary memory, we can compute the state after one step of relaxation for all vertices in S except those on S's boundary ∂S. We can then compute the state after two steps of relaxation for vertices in $S - \partial S$, except for vertices in $\partial(S - \partial S)$. After τ steps, we have a $(k - 2\tau)$-by-$(k - 2\tau)$ submesh S' at the center of S such that every vertex $i \in S'$ has state $x_i^{(\tau)}$. We then write the state of S' out to secondary memory. By tiling the $\sqrt{n}$-by-$\sqrt{n}$ mesh with $(k - 2\tau)$-by-$(k - 2\tau)$ submeshes, we can compute τ steps of linear relaxation using $\Theta(k^2\tau \cdot n/(k - 2\tau)^2)$ work, since there are $n/(k - 2\tau)^2$ submeshes in the tiling, each requiring $\Theta(k^2\tau)$ work. By choosing $k = \sqrt{M}$ and $\tau = \sqrt{M}/4$, the total time required for τ steps is $\Theta(4n\tau) = \Theta(n\tau)$. The number of I/O's for τ steps is $\Theta(k^2 \cdot n/(k - 2\tau)^2) = \Theta(4n) = \Theta(n)$. By repeating this strategy, we can compute T steps with proportional efficiency, saving a factor of $\Theta(\sqrt{M})$ I/O's over the naive method and using only a small constant factor more work, which results from redundant calculations. Hong and Kung [62] extend this result to save a factor of $\Theta(M^{1/d}/d)$ I/O's for d-dimensional meshes.

We now describe a sufficient condition for the covering technique to work when all the iterations use the same update operator F. We associate each state variable x_v with a vertex v in a directed graph G, and we connect vertex u to vertex v with an edge in the graph if $x_v^{(t)}$ depends on $x_u^{(t-1)}$. We need to cover the graph G with a family of subgraphs $\langle G_1, \ldots, G_k \rangle$ such that for each vertex v, some G_i contains all of v's neighbors within distance τ. The out-of-core algorithm can then perform τ iterations using $\Theta(Tm)$ work and $\Theta(m)$ I/O's, where m is the sum of the sizes of the subgraphs in the cover. In many cases, such as relaxation on multidimensional meshes, m is not much larger than n, and this technique leads to algorithms that perform substantially the same amount of work as the naive algorithm, but greatly reduce the amount of I/O's required.

The technique does not cause any numerical instabilities. It performs exactly the same operations on the same values as the naive algorithm, except that some operations are performed more than once, and that the order in which independent operations are executed may be different. It therefore has no effect whatsoever on the numerical results, and its applicability

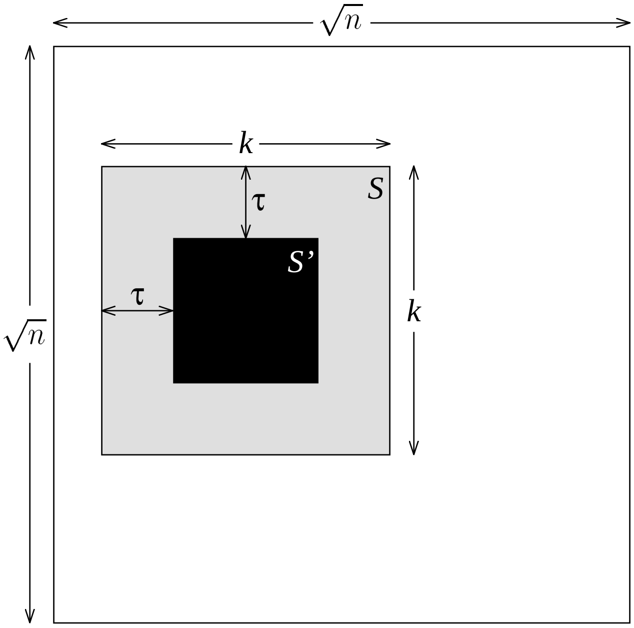


Figure 7.1 Performing an out-of-core linear relaxation computation on a $\sqrt{n}$-by-$\sqrt{n}$ mesh. The k-by-k mesh S is loaded into primary memory, τ relaxation steps are performed, and the smaller mesh S' is stored to secondary memory. The submeshes that are loaded into main memory are overlapping.

depends only on the sparsity pattern of the update operator, rather than on any special properties the update operator might have, such as linearity.

Unfortunately, as we show in Section 7.5, the technique does not apply to many common and important update operators.

Creating Locality Using Local Densification

The local densification technique tries to replace the update operator $F^{(t)}$ by another operator $G^{(t)}$ that makes more memory references to data which are already in primary memory, and which yields the solution vector in fewer iterations. Consider for example the point Jacobi relaxation algorithm for solving a linear system of equations $Ax = b$. In iteration t, we relax each "point" x_i with respect to the other points in $x^{(t-1)}$. In other words, for each i, we make the equation[1] $a_{i,*}x^{(t-1)} = b_i$ consistent by replacing $x_i^{(t-1)}$ by $x_i^{(t)}$. In matrix terms, we have $A = D + P$ where D is diagonal, and each iteration consists of updating $x^{(t)} = D^{-1}(b - Px^{(t-1)})$. Rather than relax each variable x_i separately, we can relax blocks of equations. We split $A = B + Q$ where B is block diagonal, and the iteration becomes $x^{(t)} = B^{-1}(b - Qx^{(t-1)})$. Each iteration is computationally more expensive, because we must now factor entire diagonal blocks rather than scalars, but if we choose the blocks so that entire diagonal blocks fit into primary memory, each iteration requires about the same number of I/O's as the naive point Jacobi algorithm. Since the block Jacobi is likely to converge faster, the total number of I/O's required to solve the linear system is smaller for block Jacobi than for

[1] Given a matrix A, we denote the (i, j) element of A by $a_{i,j}$, the ith row by $a_{i,*}$, and the jth column by $a_{*,j}$.

point Jacobi. In other words we have replaced the operator $F(x) = D^{-1}(b - Px)$ by a more locally dense operator $G(x) = B^{-1}(b - Qx)$.

Other instances of the local densification technique include domain decomposition methods, and polynomial preconditioners, where local densification is combined with the covering technique.

In Chapter 11 we present an application of the local densification technique for distributed memory parallel computers, where a block of state variables is stored in each processor's local memory. The algorithm is a modification of a red-black relaxation scheme, in which the update operator contains very few dependencies between state variables. The modification adds many more dependencies, but only within blocks. The modified algorithm converges faster, and requires the same amount of interprocessor communication per iteration and the same amount of time per iteration as the red-black algorithm.

Creating Locality Using Blocking Covers

In some cases, $F^{(t)}$ is a global operator in which every $x_u^{(t)}$ depends on each $x_v^{(t-1)}$, typically through some intermediate variables, such as the sum of all the $x_v^{(t-1)}$'s. The next section shows that in such cases, no good cover exists, and therefore the covering technique cannot be used. Chapter 8 describes an alternative based on **blocking covers**. The idea is to try to identify a small set of state or intermediate variables that carry a great deal of global information. These variables are called **blockers**. We then perform τ iterations in two phases. In the first phase, we determine the state of the blockers in each of the τ iterations. In the second phase, we use the precomputed state of the blockers to perform the τ iterations on the rest of the state variables. During both phases, no information is carried through the blockers, which enables us to find a good cover for the rest of the state vector. This cover, together with the specification of the blockers, is called a blocking cover. Chapter 8 describes out-of-core linear relaxation algorithms using blocking covers, and in particular, out-of-core multigrid algorithms. Chapter 9 describes out-of-core algorithms for implicit time-stepping in one dimension, again using blockers.

Chapter 10 describes an application of the blockers idea to Krylov-subspace methods. Krylov-subspace methods apply three types of operations to the state vectors: they multiply the state vectors by a matrix A, they compute inner products of state vectors, and they compute linear combination of state vectors (Krylov-subspace methods usually manipulate a few state vectors $x^{(t)}$, $r^{(t)}$, $p^{(t)}$, etc., rather than just one). We treat the inner products, which carry global information from one iteration to the next, as blockers. This allows us to use the covering technique for the other operations, namely the matrix multiplications and the linear combinations. The algorithm can be easily expressed and understood as a basis change in the Krylov-subspace method.

While some of our out-of-core algorithms based on blocking covers achieve good I/O speedups both in theory and in practice, for others, most notably multigrid algorithms, we have not been able to show good speedups in practice. One reason is that all the algorithms based on blocking covers perform more work than the corresponding in-core algorithms, typically by a factor of 2 or slightly more, because they simulate the in-core algorithms twice: once to determine the state of blockers, and again to compute the the final state vector. The other reason is that the asymptotic reduction in I/O's for 2-dimensional multigrid algorithms is $\Theta(M^{1/5})$,

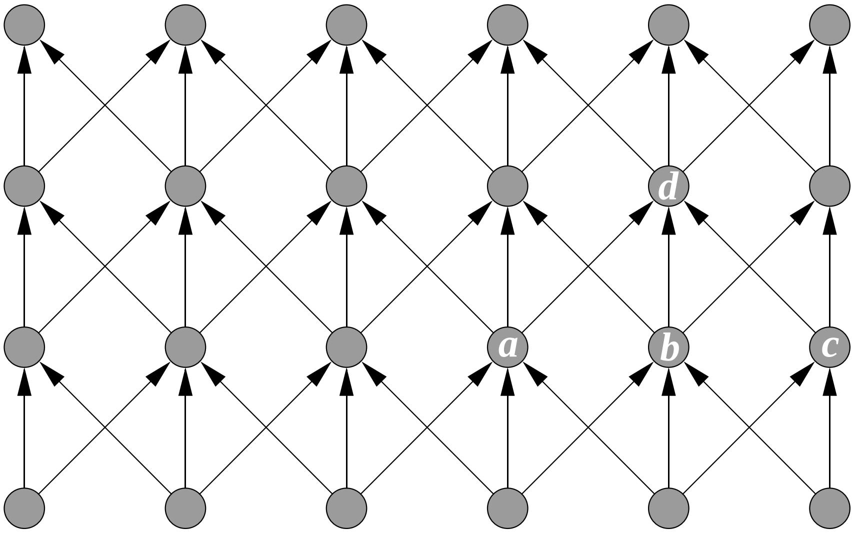


Figure 7.2 A dag that represents three iterations in a one-dimensional linear relaxation algorithm. In each iteration, the state of every variable is updated by a weighted average of its own state and the state of its two neighbors. The red-blue pebble game allows nodes to be computed more than once. The nodes a, b, and c are the predecessors of node d and must reside in primary memory whenever node d is computed.

so a very large primary memory size M is required to achieve a reduction by a factor of 30 say.

Another factor that limits our ability to reduce I/O's in some of our out-of-core Krylov-subspace methods is numerical instability, which is explained and studied in Chapter 10. Numerical experiences with other methods, however, such as with the out-of-core implicit time-stepping schemes, indicate that they are stable.

7.5 A Lower Bound

The limited applicability of the covering technique motivated most of the research reported in the second part of the thesis. Many iterative algorithms, such as multigrid algorithms, implicit time-stepping algorithms, and Krylov-subspace algorithms have a global update operator. This section proves that the covering technique cannot substentially reduce the number of I/O's, compared to the naive method, in algorithms with a global update operator. This lower bound is due to Leiserson, Rao and Toledo [74].

Hong and Kung [62] devised a formal model for studying the I/O requirements of out-of-core algorithms which use the covering techniques, called the **red-blue pebble game**. The model assumes that an algorithm is given as a directed acyclic graph (**dag**) in which nodes represent intermediate values in the computation, as illustrated in Figure 7.2. The only constraint in this model is that all predecessors of a node must reside in primary memory when the state of the node is computed. Other than this constraint, the red-blue pebble game allows arbitrary scheduling of the dag. In a linear relaxation computation, for example, each node in the dag corresponds to a state variable, and its predecessors are the state variables of its neighbors at the previous time step. The arbitrary scheduling allowed in the red-blue pebble game can be effective in reducing I/O using covers, as outlined in Section 7.4 for multidimensional meshes. It has also been applied to various other problems (see [1] for examples.)

A large class of numerical methods, including multigrid algorithms, Krylov-subspace methods, and implicit time-stepping schemes, have a common information flow structure. The methods iteratively update a state vector. The dag associated with the methods contains a set of nodes corresponding to the state vector variables for each iteration of the algorithm, plus some intermediate variables. The state of a variable at the end of iteration t is an immediate predecessor in the dag of the state of the variable at the end of $t+1$. In addition, the states of *all* variables at the end of iteration t are indirect predecessors of every variable at the end of iteration $t+1$. We now show that under the assumptions of the red-blue pebble game, the reduction in I/O for these methods is limited to $O(1)$. In other words, *no* asymptotic saving is possible in this model.

Theorem 7.1 *Let D be the dag corresponding to a T-step iterative computation with an n-node state vector $x^{(t)}$ in which the state $x_v^{(t+1)}$ of a node v after iteration $t+1$ depends directly on $x_v^{(t)}$ and indirectly on the $x_u^{(t)}$ for all state variables u. Then any algorithm that satisfies the assumptions of the red-blue pebble game requires at least $T(n-M)$ I/O's to simulate the dag D on a computer with M words of primary memory.*

Proof: The proof is illustrated in Figure 7.3. The red-blue pebble game allows redundant computations, and therefore the state $x_v^{(t)}$ of a vertex v after time step t may be computed more then once during the course of the execution of the algorithm. Let $Time_v^{(t)}$ be the first instant during the execution of the algorithm in which $x_v^{(t)}$ is computed. We denote by $Time^{(t)}$ the first instant in which the state of any vertex after time step t is computed; which is to say

$$Time^{(t)} = \min_{v \in V} \{ Time_v^{(t)} \} .$$

The state $x_v^{(t+1)}$ of each vertex v after iteration $t+1$ depends on the state of all vertices after iteration t. Therefore, we deduce that $Time^{(0)} < Time^{(1)} < \cdots < Time^{(T)}$ and that algorithm must compute the state of all vertices after iteration t between $Time^{(t)}$ and $Time^{(t+1)}$.

Let $C_u^{(t)}$ be the path $x_u^{(0)} \rightarrow x_u^{(1)} \rightarrow \cdots \rightarrow x_u^{(t)}$ in the dag. In Figure 7.3, the path $C_2^{(3)}$ is represented by a shaded area. If $x_u^{(t)}$ is computed between two time points $Time^{(t)}$ and $Time^{(t+1)}$, we know that either a vertex in $C_u^{(t)}$ was in memory at $Time^{(t)}$ or one I/O was performed between $Time^{(t)}$ and $Time^{(t+1)}$ in order to bring some vertex in $C_u^{(t)}$ into primary memory.

The vertex sets $C_u^{(t)}$ and $C_w^{(t)}$ are disjoint for $u \neq w$. Since primary memory at time $Time^{(t)}$ can contain at most M vertices, one vertex from at least $n-M$ chains $C_u^{(t)}$ must be brought from secondary memory between $Time^{(t)}$ and $Time^{(t+1)}$. Summing over all iterations, we conclude that the algorithm must perform at least $T(n-M)$ I/O's. □

For iterative algorithms in which the work per iteration is $\Theta(n)$, the performance of the naive out-of-core algorithm matches the asymptotic performance of any algorithm which satisfies the red-blue pebble game assumptions, if $M < 2n$.

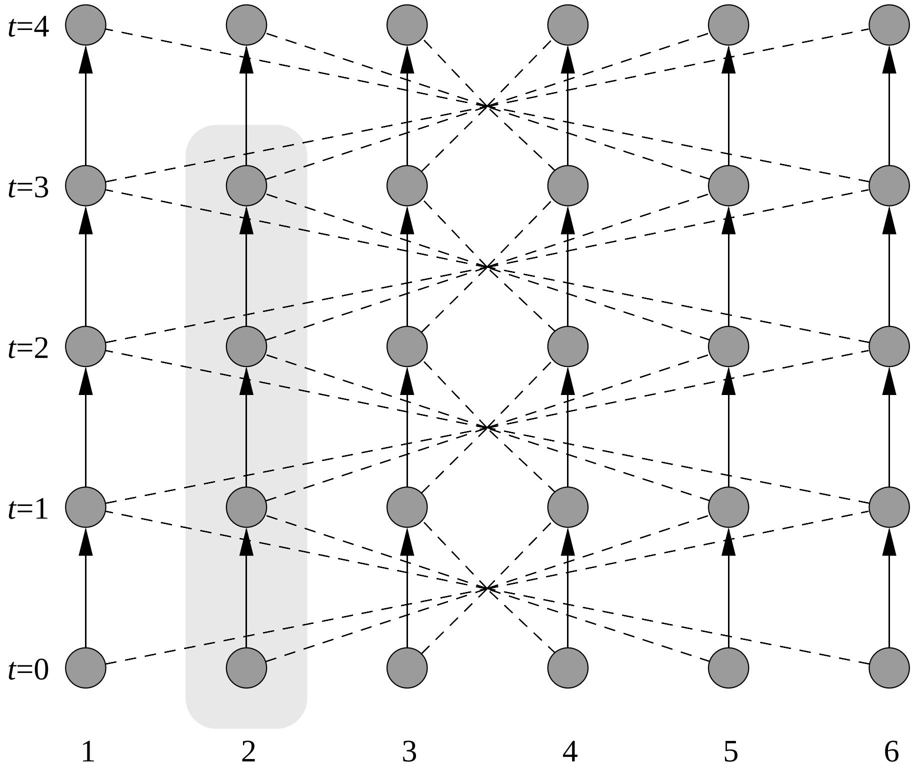


Figure 7.3 An illustration of the proof of the lower bound. The circles represent state variables $x_u^{(t)}$, the arrows represent edges in the dag, and the dashed lines represent dependencies through some intermediate variables which are not shown. The shaded area represents $C_2^{(3)}$. Between the time a variable in $x^{(3)}$ is first computed, and the time a variable in $x^{(4)}$ is first computed, at least one variable in $C_u^{(3)}$ must be in primary memory for each $u = 1, \ldots, 6$.

Corollary 7.2 *Let D be the dag corresponding to a T-step iterative computation with an n-node state vector $x^{(t)}$ in which the state $x_v^{(t+1)}$ of a node v after iteration $t+1$ depends directly on $x_v^{(t)}$ and indirectly on the $x_u^{(t)}$ for all state variables u. Then any algorithm that satisfies the assumptions of the red-blue pebble game requires $\Omega(Tn)$ I/O's to simulate the dag D on a computer with $M < 2n$ words of primary memory.* □

7.6 A Historical Survey of Out-of-Core Numerical Methods

Fifty years of computing have produced many storage technologies, including vacum tubes, transistors, and integrated circuits, electrostatic storage, delay lines, magnetic cores, magnetic tapes, drums, and disks. But through all these generations of storage technologies one fact remains unchanged: fast memory is more expensive than slow memory. This fact led computer architects to design hierarchical memory systems. Hierarchical memory systems gave rise to out-of-core algorithms that can solve problems that do not fit within the fast but small primary memory. This sections surveys the history of out-of-core numerical algorithms.

Ladd and Sheldon [71] of IBM described in 1952 a naive out-of-core iterative solver for the Thomas-Fermi-Dirac equation of quantum mechanics. Their solver ran on an IBM 701, using the primary electostatic memory to hold a small portion of the state vector, 4 magnetic tapes to hold the entire state vector, and a magnetic drum to hold the coefficients of the equations. Even though their solver read the entire state vector and all the coefficients in every iteration, it was an efficient out-of-core solver, using only 30% of the running time of each iteration for I/O operations.

A program called PDQ-5, written at the Bettis Atomic Power Laboratory, used a cover to implement an out-of-core solver for two-dimensional few-group time-independent neutron-diffution problems arising in nuclear reactor design [95]. The program was written in the early 1960's by W. R. Cadwell, L. A. Hageman, and C. J. Pfeifer. It ran on a Philco-2000 using magnetic tapes to store the state vector. The program used a line successive overrelaxation (SOR) method, which is a block relaxation method for two-dimensional problems, in which in each iteration every line of state variables in the domain is relaxed with respect to its two neighboring lines. Once the first two lines in the domain are relaxed, the first line can be relaxed again. This out-of-core method is therefore a one dimensional covering method that treats entire lines as the elementary objects in the cover.

At about that time, more efficient iterative methods became populer. Many of these methods do not lend themselves to simple and efficient out-of-core implementations, because they propagate information quickly throughout the state vector. In a 1964 survey paper on computers in nuclear reactor design [35], Cuthill remarked that alternating directions iterative methods converge faster, but are more difficult to implement efficiently out-of-core, than line SOR methods. The growing use of symmetric successive overrelaxation, conjugate gradient and Chebychev acceleration, alternating directions, and later multigrid algorithms, which are all difficult to implement out-of-core, led to declining use of out-of-core iterative solvers.

Another factor contributed to a decline in research on out-of-core iterative methods. Some of the designers and users of iterative algorithms were buying computers with very large primary memories that could store their entire problems, so they did not need out-of-core algorithms. Designers of nuclear weapons and designers of nuclear reactors were probably the most intesive users of iterative methods. Much of the research on in iterative techniques was done in that context. Nuclear engineers had enough resources to buy expensive computers with large primary storage. Designers of nuclear weapons in particular, had a significant impact on the evolution of scientific supercomputers [80]. For example, the Atomic Energy Commision (AEC) financed the development of the Univac LARC through Livermore National Laboratory and of the IBM STRETCH through Los Alamos (delivered in 1960 and 1961, respectively). Later the AEC supported the development of the Control Data Corporation CDC 6600, delivered in 1964. [99]. Once machines with very large memories became available, users at the National Laboratories became reluctant to use out-of-core iterative algorithms:

> Given that no one wants data moving between main memory and peripheral storage with every iteration of a model, memory size was an issue even with the million-word memory of the Cray 1. [80, based on a 1989 interview with George A. Michael from Livermore]

Nevertheless, there has been some recent progress in out-of-core iterative methods. In 1993, Leiserson, Rao, and the author [74] introduced the notion of blocking covers, which they used to design out-of-core multigrid algorithms. Mandel [82] described an out-of-core Krylov-subspace algorithm with a domain-decomposition preconditioner for solving finite-elements problems. Fischer and Freund proposed out-of-core Krylov-subspace methods based on polynomial preconditioning [45] and on an inner-product free Krylov-subspace method [44]. Both methods perform a small number of conjugate-gradient iterations to approximate the spectrum of the matrix. This approximation is used to construct a family of polynomials which is used in a polynomial preconditioner in one method, and in an inner-product free Krylov-subspace algorithm in another method. Both methods compute far fewer inner products than popular Krylov-subspace algorithms are are therefore easier to implement out-of-core if a good out-of-core linear relaxation algorithm is available. Finally, this thesis presents out-of-core iterative algorithms that use the blocking-covers technique.

Research on out-of-core algorithms for dense linear algbra computations yielded more lasting results, partially because the problem is easier. Some linear algebra operations have a high ratio of arithmetic operations to data, and many of them lend themselves to efficient out-of-core implementations. For example, most factorization and multiplication algorithms for n-by-n matrices perform $\Theta(n^3)$ arithmetic operation, whereas the size of their data structures is only $\Theta(n^2)$. The key idea that leads to efficient out-of-core implementations of these algorithms is that of block algorithms. For example, an algorithm for multiplying two n-by-n matrices stored in secondary storage can compute the product in blocks of size $\sqrt{M}$-by-$\sqrt{M}$ where $M/3$ is the size of primery memory. Computing each block requires $M\sqrt{M}$ multiply-adds, but less than M I/O's. The high ratio of computation to I/O leads to good performance on most machines.

The earliest reference I could find on out-of-core dense linear algebra is due to Barron and Swinnerton-Dyerm [12], who implemented a factorization algorithm for the EDSAC 2 using magnetic tapes to store the matrix. McKellar and Coffman [83] were the first to propose block-oriented storage of dense matrices and to show that this storage scheme leads to efficient out-of-core algorithms for matrix multiplication. They also investigated row major layouts and the performance of these layouts in matrix factorization. They did not produce working software. Fischer and Probert [46] analyzed the out-of-core implementation of Strassen's matrix multiplication algorithm and showed that the number of I/O's can be smaller than the number of I/O's in the convensional algorithm analyzed by McKellar and Coffman.

Moler [88] described a column-oriented factorization subroutine that works well in paging environments when the matrix is stored in a column-major order, as in Fortran. He claims that this approach is better than the block storage approach of McKellar and Coffman because of difficulties in pivoting and because of software engineering issues associated with block storage of matrices. Du Cruz, Nugent, Reid and Taylor [39] describe the block column factorization algorithm that they developped for the NAG library and its performance on virtual memory machines. Stabrowski [106] described an out-of-core factorization algorithms that use disk I/O explicitely, rather than throught the use of virtual memory. Geers and Klees [49] describe a dense out-of-core solver implemented on Siemens vector supercomputers. Their solver is motivated by applications of the boundary-elements method in geodetics. Grimes and Simon [55] describe an out-of-core algorithm for dense symmetric generalized eigenvalue problems, motivated by quantum mechanical bandstructure computations. Their algorithm works by reducing the matrix into a band matrix using block Householder transformation. They assume that the band matrix fits within primary memory. They report on the performance of the algorithm on a Cray X-MP with a solid state storage device (SSD) as secondary storage. Grimes [54] descirbe an out-of-core implementation of a dense factorization algorithm for a Cray X-MP with an SSD.

Direct solvers for sparse linear systems of equations have also been successfully implemented out-of-core, mostly in the context of finite-elements analysis. One factor leading to this success is that the amount of work in many of these algorithms is much larger than the size of the data structures they use. Consider, for example, the factorization of a sparse, symmetric positive definite matrix arising from a two-dimensional discretization of a self-adjoint elliptic partial differential equation on a $\sqrt{n}$-by-$\sqrt{n}$ mesh. The most efficient algorithms factorize the matrix, which has $\Theta(n)$ nonzeros, into factors with $\Theta(n \log n)$ nonzeros, using $\Theta(n\sqrt{n})$ work [52].

Early out-of-core direct solvers for sparse linear systems were primarily implementations of banded solvers. A very early reference on sparse elimination is due to Riesel [97] who, in 1956, implemented a sparse elimination algorithm on the BESK computer at the SAAB Aircraft Company in Sweden. The solver was used to solve linear systems with 214 variables, which did not fit within the BESK's 8192-word magnetic drum. Cantin [25] described in 1971 an out-of-core solver for banded linear systems, which used disks as secondary storage. In 1974, both Mondkar and Powell [89] and Wilson, Bathe and Doherty [124] described band solvers using tapes for secondary storage. Wimberly [125] implemented out-of-core solvers for finite element analysis. He implemented both direct and iterative (Gauss-Seidel) solvers, and compared their performance using a software simulation of paging algorithms. Crotty [33] described an out-of-core solver for matrices arising from the boundary-elements method in linear elasticity. The solver handles zero blocks in the coefficient matrix efficiently.

Recent out-of-core sparse solvers often use variants of nested dissection and frontal methods, rather than band methods. For example, a the PROPHLEX finite-elements software package by *Computational Mechanics Company* includes an out-of-core frontal solver[2]. George and Rashwan [51] described an out-of-core sparse factorization algorithm based on incomplete nested dissection. Liu [76, 77] described out-of-core implementations of the multifrontal method for sparse factorization (see also [78]). Bjørstad [15] described the out-of-core sparse factorization used in SESAM, a structural analysis software package. The solver is a block Cholesky algorithm that handes nonzero blocks as dense submatrices. The key idea behind many of these out-of-core algorithms is the switching from a sparse representation to a dense representation on small submatrices. Liu attributes this idea to Gustavson [77, page 311]. George, Heath, and Plemmons [53] describe an out-of-core sparse QR factorization algorithm for solving large least-squares problems. They motivate the algorithm with several applications, including geodetic surverying, finite elements, and earthquake studies. Their factorization algorithm uses Givens rotations and reordering of the variables to reduce the amount of I/O required by conventional algorithms.

Eisenstat, Schultz and Sherman [41] describe an interesting algorithm that uses a technique that is also used in this thesis. They noticed that the triangular Cholesky factors of a matrix typically require much more storage than the original matrix, and hence an out-of-core algorithm is often needed even when the matrix fits within main memory. They reduce the amount of storage required by the solver by discarding and recomputing entries in the Cholesky factor, and

[2] See `http://www.comco.com`.

hence eliminate the need for an out-of-core algorithm in some cases. Their algorithm performs more work than conventional solvers, but it runs without paging on machines with a small primary storage. One disadvantage of the algorithm is that it solves the input linear system but it does not produce the factorization of the coeficient matrix. The out-of-core Krylov-subspace algorithms described in Chapter 10 also discard and recompute information, which leads to efficient out-of-core algorithms that outperform conventional algorithms on machines with small primary memories.

Out-of-core algorithms for the Fast Fourier Transform (FFT) have been recently surveryed by Bailey [7] and by Van Loan [119]. The ratio of computation to data in the FFT is smaller than in the other problems surveyed in this section: computing the FFT of an n-vector requires only $\Theta(n \log n)$ work. Fortunately, there are some good out-of-core FFT algorithms that perform only a few passes over the data. Soon after the introduction of the FFT algorithm by Cooley and Tukey, Gentleman and Sande [50] presented an FFT algorithm which is suitable for out-of-core applications. The algorithm works by arranging the input n vector in a two-dimensional array, performing an FFT on every row, scaling the array, transposing the array and performing an FFT on every row again (every column of the original array. Assuming that primary memory can hold at least one row and one column of the array, the algorithm requires only two passes over the data and an out-of-core matrix transposition. The algorithm did not gain much acceptance, but it was rediscovered and implemented on several machines. See [7] for a survey and a description of an implementation on a Cray X-MP with an SSD as a secondary memory. Other, less efficient, out-of-core FFT algorithms were proposed by Singleton [103] and Brenner [21]. Eklundh [42] noticed that an efficient out-of-core matrix transposition algorithm is a key to out-of-core two-dimensional FFT algorithms, and suggested one. Others have also proposed algorithms for matrix transposition, for example [3, 102, 117].

Although outside the scope of this thesis, it is worth mentioning that out-of-core implementations are the standard in mathematical programming, and in linear programming in particular. Hoffman tells about solution of linear programming problems in the National Bureau of Standards between 1951 and 1956 [61]. The SEAC computer [99] was used to run simplex code which in every iteration read a simplex tableau from one tape and wrote the next tableau on another tape. Orchard-Hays, who is credited by Dantzig as the most important early implementor of the simplex algorithm [36], described in 1968 out-of-core considerations as being essential to linear programming algorithms [92].

Chapter 8

Efficient Out-of-Core Algorithms for Linear Relaxation Using Blocking Covers

8.1 Introduction

This chapter presents a method for increasing the locality in linear relaxation algorithms, in particular multigrid algorithms, using the blocking covers technique. The method is applied here to algorithms in which the simpler covering technique cannot be applied. Although the algorithmic details of this chapter differ from the details in Chapters 9 and 10 that also use blocking covers, the essence of the technique is the same: the blocking cover uses linear transformations to turn an algorithm for which no good cover exists into an algorithm for which a good cover does exist. Once an algorithm is so transformed, the plain covering technique is applied.

Many numerical problems can be solved by linear relaxation. A typical linear relaxation computation operates on a directed graph $G = (V, E)$ in which each vertex $v \in V$ contains a numerical state variable x_v which is iteratively updated. On step t of a linear relaxation computation, each state variable is updated by a weighted linear combination of its neighbors:

$$x_v^{(t)} = \sum_{(u,v) \in E} A_{uv}^{(t)} x_u^{(t-1)} , \tag{8.1}$$

where $A_{uv}^{(t)}$ is a predetermined **relaxation weight** of the edge (u, v). We can view each iteration as a matrix-vector multiplication $x^{(t)} = A^{(t)} x^{(t-1)}$, where $x^{(t)} = \langle x_1^{(t)}, x_2^{(t)}, \ldots, x_{|V|}^{(t)} \rangle^{\mathrm{T}}$ is the state vector for the tth step, and $A^{(t)} = (A_{uv}^{(t)})$ is the **relaxation matrix** for the tth step. We assume $A_{uv}^{(t)} = 0$ if $(u, v) \notin E$. The goal of the linear relaxation is to compute a final state vector $x^{(T)}$ given an initial vector $x^{(0)}$, a scheme for computing $A_{uv}^{(t)}$ on each step t, and a total number T of steps. Examples of linear relaxation computations include Jacobi relaxation, Gauss-Seidel relaxation, multigrid computations, and many variants of these methods [24]. (Iterative processes of the form $y^{(t)} = M^{(t)} y^{(t-1)} + b$ can be transformed to an iteration of the form $x^{(t)} = A^{(t)} x^{(t-1)}$ using a straightforward linear transformation.)

[1] A preliminary version of this chapter, which is joint work with Charles E. Leiserson and Satish Rao, was presented in the *34th Symposium on Foundations of Computer Science*, Palo Alto, California, November 1993, and accepted for publication in the *Journal of Computer and System Sciences*.

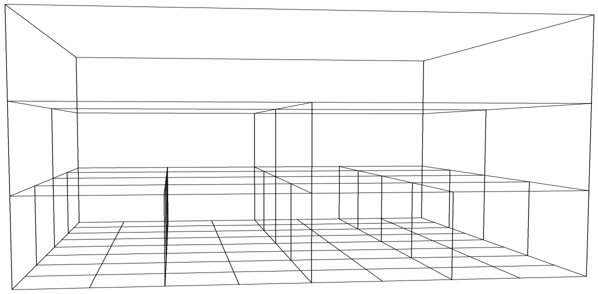


Figure 8.1 A 9-by-9 multigrid graph. The graph has levels 0, the bottommost, through 3, the topmost.

A computer with ample primary memory can perform a linear relaxation computation by simply updating the state vector according to Equation (8.1). Since we are normally interested only in the final state vector $x^{(T)}$, we can reuse the state-vector storage. If we assume that the scheme for generating the nonzero entries of the relaxation matrices $A^{(t)}$ is not a significant cost (for example, all relaxation matrices may be identical), then the time to perform each iteration on an ordinary, serial computer is $O(E)^2$. (It can be less, since if a row v of the relaxation matrix at some step t is 0 everywhere except for a 1 on the diagonal, then no computation is required to compute $x_v^{(t)}$.) Besides the space required for the relaxation weights, the total amount of storage required for the entire T-step computation is $\Theta(V)$ to store the state vector.

Linear relaxation on grids naturally arises from the problem of solving sparse linear systems of equations, arising from the discretization of partial differential equations. We have shown in Chapter 7 that if the sparsity structure of the relaxation matrix corresponds to a low-dimensional grid, substantial reduction in I/O's can be achieved through the covering technique. For this class of problems, however, it has been found that more rapid convergence can often be obtained by performing a linear relaxation computation on a multigrid graph [24]. A multigrid graph is a hierarchy of progressively coarser meshes, as is shown in Figure 8.1. The kth level is a $\sqrt{n}/2^k$-by-$\sqrt{n}/2^k$ mesh, for $k = 0, 1, \ldots, (\lg n)/2$, whose (i, j) vertex is connected to the $(2i, 2j)$ vertex on the $(k-1)$st mesh.

A typical multigrid application is that of solving a time-dependent partial differential equation. The computation consists of many repeated **cycles**, in which the relaxation proceeds level by level from the finest mesh to the coarsest and back down. A naive implementation of $T \geq \lg n$ cycles of the computation takes $\Theta(Tn)$ time, even though there are $\Theta(\lg n)$ levels, since the number of vertices on each level decreases geometrically as the grids become coarser. For a computer with M memory running T cycles of a $\sqrt{n}$-by-$\sqrt{n}$ multigrid algorithm, where $n \geq 2M$, the number of I/O's required for T cycles is $\Theta(Tn)$ as well.

Can the number of I/O's be reduced for this multigrid computation? Unfortunately, the answer provided by Theorem 7.1 is no, even if redundant computations are allowed. The naive algorithm is optimal. The problem is essentially that information propagates quickly in the multigrid because of its small diameter.

Nevertheless, we shall see in Section 8.5 that we can actually save a factor of $M^{1/5}$ in

[2] Inside asymptotic notation (such as O-notation or Θ-notation), the symbol V denotes $|V|$ and the symbol E denotes $|E|$.

I/O's. The key idea is to artificially restrict information from passing through some vertices by treating their state variables symbolically. Because the relaxations are linear, we can maintain dependences among the symbolic variables efficiently as a matrix. This technique is general and is particularly suited to graphs whose connections are locally dense and globally sparse.

The remainder of this chapter is organized as follows. In Section 8.2 we formally introduce the notion of blocking covers and discuss the relation between state variables in a linear relaxation computation and a blocking cover. In Section 8.3 we present our method. The details of the method are presented in Section 8.4. The application of our basic result to multigrid relaxation is presented in Section 8.5. In Section 8.6 we describe algorithms for finding good blocking covers for planar and simplicial graphs, which yield I/O-efficient relaxation algorithms for these classes of graphs.

8.2 Blocking Covers

This section introduces the definition of a blocking cover, as well as several other definitions and notations that we shall use extensively in subsequent sections. We conclude the section with an important identity describing how state variables depend on one another.

We can abstract the method of Hong and Kung described in Section 8.1 using the notion of graph covers. Given a directed graph $G = (V, E)$, a vertex $v \in V$, and a constant $\tau \geq 0$, we first define $N^{(\tau)}(v)$ to be the set of vertices in V such that $u \in N^{(\tau)}(v)$ implies there is a path of length at most τ from u to v. A **τ-neighborhood-cover** [6] of G is a sequence of subgraphs $\mathcal{G} = \langle G_1 = (V_1, E_1), \ldots, G_k = (V_k, E_k)\rangle$ such that for all $v \in V$, there exists a $G_i \in \mathcal{G}$ for which $N^{(\tau)}(v) \subseteq V_i$. Hong and Kung's method can reduce the I/O requirements by a factor of τ over the naive method if the graph has a τ-neighborhood-cover with $O(E/M)$ subgraphs, each of which has $O(M)$ edges, where M is the size of primary memory. Although a vertex can belong to more than one subgraph in the cover, there is one subgraph that it considers to be its "home," in the sense that the subgraph contains all of its neighbors within distance τ. When performing a linear relaxation on G for τ time steps, therefore, the state of v depends only on other vertices in v's home. Thus, in a linear relaxation computation, we can successively bring each subgraph in the cover into primary memory and relax it for τ steps without worrying about the influence of any other subgraph for those τ steps.

The problem with Hong and Kung's method is that certain graphs, such as multigrid graphs and other low-diameter graphs, cannot be covered efficiently with small, high-diameter subgraphs. Our strategy to handle such a graph is to "remove" certain vertices so that the remaining graph has a good cover. Specifically, we select a subset $B \subseteq V$ of vertices to form a **blocking set**. We call the vertices in the blocking set **blocking vertices** or **blockers**. We define the τ-neighborhood of v with respect to a blocking set $B \subseteq V$ to be $N_B^{(\tau)}(v) = \{u \in V : \exists$ a path $u \to u_1 \to \cdots \to u_t \to v$, where $u_i \in V - B$ for $i = 1, 2, \ldots, t < \tau\}$. Thus, the τ-neighborhood of v with respect to B consists of vertices that can be reached with paths of length at most τ whose internal vertices do not belong to B.

We can now define the notion of a blocking cover of a graph.

Definition Let $G = (V, E)$ be a directed graph. A (τ, r, M)**-blocking-cover** of G is a pair $(\mathcal{G}, \mathcal{B})$, where $\mathcal{G} = \langle G_1 = (V_1, E_1), \ldots, G_k = (V_k, E_k)\rangle$ is a sequence of subgraphs of G and $\mathcal{B} = \langle B_1, \ldots, B_k\rangle$ is a sequence of subsets of V such that

BC1. for all $i = 1, \ldots, k$, we have $M/2 \leq |E_i| \leq M$;

BC2. for all $i = 1, \ldots, k$, we have $|B_i| \leq r$;

BC3. $\sum_{i=1}^{k} |E_i| = O(E)$;

BC4. for all $v \in V$, there exists a $G_i \in \mathcal{G}$ such that $N_{B_i}^{(\tau)}(v) \subseteq V_i$.

For each $v \in V$, we define **home(v)** to be an arbitrary one of the G_i that satisfies BC4.

Our basic algorithm for linear relaxation on a graph $G = (V, E)$ depends on having a (τ, r, M)-blocking-cover of G such that $r^2\tau^2 \leq M$. In the description and analysis of the basic algorithm, we shall assume for simplicity that each step of the computation uses the same relaxation matrix A. We shall call such a computation a **simple linear relaxation computation**. We shall relax this simplifying assumption in Section 8.5.

In a simple linear relaxation computation on a graph $G = (V, E)$ with a relaxation matrix A, the state vector $x^{(t)}$ at time t satisfies

$$x^{(t)} = A^t x^{(0)} \ .$$

That is, the computation amounts to powering the matrix.[3] We shall generally be interested in the effect that one state variable $x_u^{(s)}$ has on another $x_v^{(t)}$. Define the weight $w(p)$ of a length-r path $p = v_0 \to v_1 \to \cdots \to v_r$ in G to be

$$w(p) = \prod_{k=1}^{r} A_{v_{k-1},v_k} \ . \tag{8.2}$$

For two vertices $u, v \in V$, we define

$$w(r; u, v) = \sum_{p \in \mathcal{P}(r)} w(p) \ ,$$

where $\mathcal{P}(r) = \{p \in G : p \text{ is a length-}r \text{ path from } u \text{ to } v\}$. (We define $w(r; u, v) = 0$ if no length-r path exists between u and v.) Using this notation, we have

$$x_v^{(t)} = \sum_{u \in V} w(t; u, v) x_u^{(0)} \ . \tag{8.3}$$

If $B_i \subseteq V$ is a blocking set, then we define

$$w_{B_i}(r; u, v) = \sum_{p \in \mathcal{P}_{B_i}(r)} w(p) \ ,$$

where $\mathcal{P}_{B_i}(r) = \{p \in G : p \text{ is a length-}r \text{ path from } u \text{ to } v \text{ whose intermediate vertices belong to } V - B_i\}$.

Consider a subgraph G_i in the cover and its corresponding blocking set B_i. The following lemma shows that in order to know the value of a state variable $x_v^{(t)}$ where $home(v) = G_i$, it suffices to know the initial values of all state variables for vertices in G_i at time 0 and also to know the values of the **blocker variables**—state variables for blockers—in B_i at all times less than t.

[3] Since we do not want to destroy the sparsity of A, and we wish our technique to generalize, we do not take advantage of techniques such as repeated squaring.

Lemma 8.1 *Let $G = (V, E)$ be a directed graph, and let $(\mathcal{G}, \mathcal{B})$ be a (τ, r, M)-blocking-cover of G. Then, for any simple linear relaxation computation on G, we have for all $v \in V$ and for any $t \leq \tau$*

$$\begin{aligned} x_v^{(t)} &= \sum_{u \in V_i} w_{B_i}(t; u, v) x_u^{(0)} \\ &\quad + \sum_{u \in B_i} \sum_{s=1}^{t-1} w_{B_i}(s; u, v) x_u^{(t-s)} \ , \end{aligned} \tag{8.4}$$

where $G_i = home(v)$.

Proof: We have $\mathcal{P}(r) = \mathcal{P}_{B_i}(r) + \mathcal{P}_{\overline{B}_i}(r)$ where $\mathcal{P}_{\overline{B}_i}(r) = \{p \in G : p$ is a length-r path from u to v with at least one intermediate vertex which belongs to $B_i\}$. We also define

$$w_{\overline{B}_i}(r; u, v) = \sum_{p \in \mathcal{P}_{\overline{B}_i}(r)} w(p) \ ,$$

By Equation (8.3) and the notation above we have

$$\begin{aligned} x_v^{(t)} &= \sum_{u \in V} w(t; u, v) x_u^{(0)} \\ &= \sum_{u \in V} w_{B_i}(t; u, v) x_u^{(0)} + \sum_{u \in V} w_{\overline{B}_i}(t; u, v) x_u^{(0)} \ . \end{aligned} \tag{8.5}$$

We now prove that the first sum in Equation (8.5) equals the first sum in Equation (8.4) and that the second sum in Equation (8.5) equals the second sum in Equation (8.4). That the first summations are equal follows from condition **BC4** in the definition of blocking covers, which imply that if $home(v) = G_i$, $u \notin V_i$, and $t \leq \tau$, then $w_{B_i}(t; u, v) = 0$.

We use induction on t to prove that

$$\sum_{u \in B_i} \sum_{s=1}^{t-1} w_{B_i}(s; u, v) x_u^{(t-s)} = \sum_{z \in V} w_{\overline{B}_i}(t; z, v) x_z^{(0)} \ .$$

For $t = 0$ the equation holds since both summations are empty. Assume that the equation holds for all $s > 0$ and for all $v \in V$ such that $home(v) = G_i$. We split the blocking influence $w_{\overline{B}_i}(t; z, v)$ according to the last blocker on each path,

$$w_{\overline{B}_i}(t; z, v) = \sum_{u \in B_i} w_{\overline{B}_{i,u}}(t; z, v)$$

where $w_{\overline{B}_{i,u}}(t; z, v)$ is the sum of path weights over all length-t paths from z to v in which the last vertex in B_i is u. Splitting the paths from z to v at u we get

$$w_{\overline{B}_{i,u}}(t; z, v) = \sum_{s=1}^{t-1} w_{B_i}(s; u, v) w(t - s; z, u) \ .$$

We now have

$$\sum_{z \in V} w_{\overline{B}_i}(t; z, v) x_z^{(0)} \quad = \quad \sum_{z \in V} \sum_{u \in B_i} w_{\overline{B}_{i,u}}(t; z, v) x_z^{(0)}$$

$$
\begin{aligned}
&= \sum_{u \in B_i} \sum_{z \in V} w_{\overline{B}_i,u}(t; z, v) x_z^{(0)} \\
&= \sum_{u \in B_i} \sum_{z \in V} \sum_{s=1}^{t-1} w_{B_i}(t-s; u, v) w(s; z, u) x_z^{(0)} \\
&= \sum_{u \in B_i} \sum_{s=1}^{t-1} w_{B_i}(s; u, v) \sum_{z \in V} w(t-s; z, u) x_z^{(0)} \\
&= \sum_{u \in B_i} \sum_{s=1}^{t-1} w_{B_i}(t-s; u, v) x_u^{(t-s)} \ ,
\end{aligned}
$$

where the last equality follows by the inductive assumption. □

8.3 Simple Linear Simulation

In this section, we present our I/O-efficient algorithm to perform a simple linear relaxation computation on any graph with a (τ, r, M)-blocking-cover $(\mathcal{G}, \mathcal{B})$. We call it a "simulation" algorithm, because it has the same effect as executing a simple linear relaxation algorithm, but it does not perform the computation in the same way. The simulation algorithm is not a new numerical algorithm. Rather, it is a new way to *implement* a numerical algorithm. Since we only present a new implementation strategy, convergence properties are maintained. Given a numerical algorithm, if both a conventional implementation and our implementation are executed on an ideal computer with no rounding errors, the output is exactly the same. In this section, we give an overview of the simulation algorithm and analyze its performance.

The goal of the simulation algorithm is to compute the state vector $x^{(T)}$ in Equation (8.3) given an initial state vector $x^{(0)}$ and a number T of steps. The algorithm has four phases, numbered 0 through 3. Phase 0 is executed once as a precomputation step. It computes the coefficients $w_{B_i}(s; u, v)$ in the second summation of Equation (8.4) that express the influence of one blocker variable on another. Phases 1–3 advance the state vector by τ steps each time they are executed, and these steps are then iterated until the state vector has been advanced by a total of T steps. Phase 1 computes the first summation in Equation (8.4) for the blocker variables, which in combination with the coefficients computed in Phase 0 yields a triangular linear system of equations on the blockers. Phase 2 solves these equations for the blockers using back substitution. Finally, Phase 3 extends the solution for the blocker variables to all the state variables. If $r^2\tau^2 \leq M$, each iteration of Phases 1–3 performs $O(\tau E)$ work, performs $O(E)$ I/O's, and advances the state vector by τ steps, as compared with the naive algorithm, which would perform $O(\tau E)$ I/O's for the same effect. Phase 0, the precomputation phase of the algorithm, requires $O(r\tau E)$ work and $O(E)$ I/O's.

We now describe each phase in more detail.

The goal of Phase 0 is to compute the coefficients $w_{B_i}(s; u, v)$ in the second summation of Equation (8.4) for all $s = 1, 2, \ldots, \tau - 1$, for all $u \in B_i$, and for all $v \in B$ where $B = \bigcup_{B_i \in \mathcal{B}} B_i$ and $G_i = \text{home}(v)$. The coefficient $w_{B_i}(s; u, v)$ represents the influence that the value of u at time $\tau - s$ has on the value of v at time τ. The influence (coefficient) of blocker u on another in the same time step ($s = 0$) is 0, unless the other vertex is in fact u, in which case the influence is 1. Inductively, suppose that the state variable for each vertex v contains $w_{B_i}(s; u, v)$. To compute the $w_{B_i}(s+1;u,v)$, we set the blocker variables to 0 and run one step of linear relaxation on G_i. The value for $w_{B_i}(s+1;u,v)$ is produced in the state variable for v. To compute up to $s=\tau-1$, this computation is therefore nothing more than a linear relaxation computation in which the blockers are zeroed at every step. Intuitively, this method works because any individual coefficient can be obtained from Equation (8.4) by setting all the state variables in both summations to 0, except for that state variable in the second summation which is multiplied by the desired coefficient, which we set to 1. During Phase 0, any coefficient that represents an influence of a blocker on a blocker whose home is G_i is saved for use in Phase 2.

In Phase 1 we focus on the first summation in Equation (8.4). Phase 1 computes the sums $\sum_{u\in V_i} w_{B_i}(t;u,v)x_u^{(0)}$ for all $t\leq\tau$ and for all $v\in B$, where $G_i=\text{home}(v)$. These values represent the contribution of the initial state on v without taking into account contributions of paths that come from or pass through blockers in v's home. For a given subgraph G_i in the blocking cover for G, the first summation is simply a linear relaxation on the subgraph of G_i induced by $V_i - B_i$. Thus, we can compute this summation for all blocker variables whose home is G_i by a linear relaxation on G_i as follows. We initialize the state variables according to $x^{(0)}$, and then for each subsequent step, we use the value 0 whenever the computation requires the value of a blocker variable.

Phase 2 solves for the blocker variables. If we inspect Equation (8.4), we observe that since we have computed the value of the first summation and the coefficients of the variables in the second summation, the equations become a linear system in the blocker variables. Furthermore, we observe that the system is triangular, in that each $x_v^{(i)}$ depends only on various $x_u^{(j)}$ where $j<i$. Consequently, we can use the back substitution method [32, Section 31.4] to determine the values for all the blocker variables.

Phase 3 computes the state variables for the nonblocker vertices by performing linear relaxations in each subgraph as follows. For a subgraph G_i, we set the initial state according to $x^{(0)}$ and perform τ steps of linear relaxation, where at step i blocker variable $x_u^{(i)}$ is set to the value computed for it in Phase 2. We can show that the state variables for each node whose home is in G_i assume the same values as if they were assigned according to a linear relaxation of G with the initial state $x^{(0)}$ by using induction and the fact that each blocker variable assumes the proper value.

In Section 8.4 we prove that given a graph $G=(V,E)$ with a (τ,r,M)-blocking-cover such that $r^2\tau^2\leq M$, a computer with $O(M)$ words of primary memory can perform $T\geq\tau$ steps of a simple linear relaxation on G using at most $O(TE)$ work and $O(TE/\tau)$ I/O's. The precomputation phase (which does not depend on the initial state) requires $O(r\tau E)$ work and $O(E)$ I/O's.

8.4 The Algorithm in Detail

This section contains the detail of our basic algorithm, which was outlined in Section 8.3. We begin by describing the data structures used by our algorithm and then present the details of each of the four phases of the algorithm. We give pseudocode for the phases and lemmas that imply their correctness.

Data Structures

The main data structure that the algorithm uses is a table S which during the algorithm contains information about vertices in one of the subgraphs G_i in the blocking cover of G with respect to the blocking set B_i. Each row $S[j]$ of S contains several fields of information about one vertex. The field $S[j].\mathit{Name}$ contains the vertex index in G, the boolean field $S[j].\mathit{IsInB}$ denotes whether the vertex belongs to $B = \bigcup_{B_i \in \mathcal{B}} B_i$, the boolean field $S[j].\mathit{IsBlocker}$ denotes whether the vertex belongs to B_i, and the boolean field $S[j].\mathit{IsHome}$ denotes whether the home of the vertex is G_i. The field $S[j].\mathit{Adj}$ is an adjacency list of the neighbors of the vertex in G_i (incoming edges only). Each entry in the adjacency list is the index of a neighbor in S together with the relaxation weight of the edge that connects them. The two last fields are numeric fields $S[j].x$ and $S[j].y$. The field $S[j].x$ holds the initial state of the vertex, and after the algorithm terminates the field $S[j].y$ holds the state after time step τ if $S[j].\mathit{IsHome}$ is set.

A data structure S_i is stored in secondary memory for each subgraph G_i in the cover. In addition to S_i we store in secondary memory a table H_i for each subgraph G_i. For every vertex v whose home is G_i, the table lists all the subgraphs G_i in the blocking cover containing v and the index of v in S_i. These tables enable us to disperse the value of $x_v^{(\tau)}$ from the home of v to all the other G_i which contain v. The size of each S_i is $6|V_i| + 2|E_i|$. The total amount of secondary storage required to store the blocking cover is $O(E)$.

We also store in secondary memory two 2-dimensional numeric tables WX and X of size τ-by-$|B|$, and one 3-dimensional numeric table W of size τ-by-r-by-$|B|$. The table W is used to store the coefficients $w_{B_i}(s; u, v)$ for all $s < \tau$, for all $u \in B_i$, and for all $v \in B$, where $G_i = \text{home}(v)$. The table WX is used to store the sum $\sum_{u \in V_i} w_{B_i}(t; u, v) x_u^{(0)}$ for all $t \leq \tau$, and all $v \in B$ where $G_i = \text{home}(v)$. The table X is used to store the values $x_v^{(t)}$ for all $t \leq \tau$ and all $v \in B$.

Phase 0

The pseudocode below describes the Phase 0 of the algorithm. The influence of one blocker on all other blockers in a subgraph is computed by the procedure BlockersInfluence. The procedure PhaseZero loads one subgraph at a time into primary memory, and then calls BlockersInfluence at most r times. Before each call exactly one vertex whose $S[j].\mathit{IsBlocker}$ field is set is chosen, its $S[j].x$ field is set to 1 and all the other x fields are set to 0. The index i of the blocker whose influence is computed is passed to BlockersInfluence.

```
PhaseZero()
for $i \leftarrow 1$ to $k$
    do load $S_i$ into primary memory
       for $b \leftarrow 1$ to $M$
           do if $S[b].\mathit{IsBlocker}$
                 then for $l \leftarrow 1$ to $M$ do $S[l].x \leftarrow 0$
                      $S[b].x \leftarrow 1$
                      BlockersInfluence($b$)
```

```
BLOCKERSINFLUENCE($b$)
1  for $s \leftarrow 1$ to $\tau - 1$
2      do for $j \leftarrow 1$ to $M$
3             do $S[j].y \leftarrow \sum_{(l,a) \in S[j].Adj} a \cdot S[l].x$
4         for $j \leftarrow 1$ to $M$
5             do if $S[j].IsBlocker$
6                   then $S[j].x \leftarrow 0$
7                   else $S[j].x \leftarrow S[j].y$
8                if $S[j].IsInB$ and $S[j].IsHome$
9                   then write $S[j].y$ to $W[s, S[b].Name, S[j].Name]$
```

Lemma 8.2 *After Phase 0 ends, for each $v \in B$, $u \in B_i$ and $s < \tau$, we have*

$$W[s, u, v] = w_{B_i}(s; u, v) \; ,$$

where $G_i =$ home(v).

Proof: We denote the state vectors in phase 0 by $e^{(t)}$ instead of $x^{(t)}$ to indicate that the initial state is a unit vector with 1 for one blocker and 0 for all the other vertices.

We prove by induction on s that lines 1–7 of BLOCKERSINFLUENCE perform linear relaxation on G with all outgoing edges from blockers in B_i removed, on all the vertices v for which $N_{B_i}^{(s)}(v) \subseteq V_i$ and on all the blockers in B_i. The claim is true for $s = 0$ because before the first iteration the state $S[j].x$ of every vertex is the initial state set by PHASEZERO. Assume that the claim is true for $s < \tau - 1$. In the next iteration $S[j].y$ is assigned the weighted linear combination of all of its neighbors in G_i. If the vertex v is a blocker, its state is zeroed in line 6 and the claim holds. If the vertex is not a blocker but $N_{B_i}^{(s+1)}(v) \subseteq V_i$, then all its neighbors are in G_i, and each neighbor u is either a blocker or $N_{B_i}^{(s)}(u) \subseteq V_i$. In either case, the y field is assigned the weighted linear combination of vertex states which are correct by induction, so its own state is correct.

The initial state is 0 for all vertices except for one blocker $u = S[b].Name$ whose initial state is $e_u^{(0)} = 1$. By Equation (8.3) and condition **BC4** in the definition of blocking-covers we have for all $s < \tau$ and $v \in V$ such that $G_i = home(v)$

$$\begin{aligned} e_v^{(s)} &= \sum_{z \in V} w_{B_i}(s; z, v) e_z^{(0)} \\ &= w_{B_i}(s; u, v) \; . \end{aligned}$$

The value $e_v^{(s)} = w_{B_i}(s; u, v)$ is written to $W[s, u, v]$ in line 9 for all $v \in B$ such that $G_i = home(v)$. □

Let us analyze the amount of work and the number of I/O's required in Phase 0. BLOCKERSINFLUENCE is called at most rk times where k is the number of subgraphs in the cover. In each call, the amount of work done is $O(\tau M)$ so the total amount of work is $O(rkM\tau) = O(r\tau E)$. The total number of I/O's is $O(E)$ to load all the S_i into primary memory, and $|B|r\tau \leq kr^2\tau = O(E)$ to store the table W (since $W[s, *, v]$ is a sparse vector).

Phase 1

Phase 1 is simpler than Phase 0. The procedure INITIALSTATEINFLUENCE is similar to procedure BLOCKERSINFLUENCE in Phase 0, but the table WX is written to secondary memory instead of the table W. The procedure PHASEONE loads one subgraph at a time and calls INITIALSTATEINFLUENCE once, with the initial state loaded from secondary memory.

```
PHASEONE()
for $i \leftarrow 1$ to $k$
    do load $S_i$ into primary memory
       INITIALSTATEINFLUENCE()
```

```
INITIALSTATEINFLUENCE()
1  for $s \leftarrow 1$ to $\tau$
2      do for $j \leftarrow 1$ to $M$
3             do $S[j].y \leftarrow \sum_{(l,a) \in S[j].Adj} a \cdot S[l].x$
4         for $j \leftarrow 1$ to $M$
5             do if $S[j].IsBlocker$
6                   then $S[j].x \leftarrow 0$
7                   else $S[j].x \leftarrow S[j].y$
8                if $S[j].IsInB$ and $S[j].IsHome$
9                   then write $S[j].y$ to $WX[s, S[j].Name]$
```

Lemma 8.3 *After Phase 1 ends, for each $v \in B$ and $s \leq \tau$, we have*

$$WX[s, v] = \sum_{u \in V_i} w_{B_i}(s; u, v) x_u^{(0)} ,$$

where $G_i = home(v)$.

Proof: Lines 1–7 of INITIALSTATEINFLUENCE simulate linear relaxation on G with all outgoing edges from blockers in B_i removed. The proof of this claim is identical to the proof of Lemma 8.2, with the initial state being the given initial state $x^{(0)}$. Therefore we have for all $s \leq \tau$ and $v \in V$ such that $G_i = home(v)$

$$x_v^{(s)} = \sum_{z \in V} w_{B_i}(s; z, v) x_z^{(0)} . \tag{8.6}$$

This value is written to $WX[s, v]$ in line 9 for all $v \in B$ such that $G_i = home(v)$. □

The total amount of work in Phase 1 is $O(k\tau M) = O(\tau E)$. The number of I/O's is $O(E)$ to load all the subgraphs, and $|B|\tau$ to store the table WX.

Phase 2

Phase 2 solves the lower triangular system of linear equations defined by Lemma 8.1 for every $v \in B$ and all $t \leq \tau$. Entries in the tables W, WX and X are written and read from secondary memory as needed.

```
PhaseTwo()
for $t \leftarrow 1$ to $\tau$
    do for $v \leftarrow 1$ to $|B|$
          do Let $G_i$ be the home of $v$
             $X[t,v] \leftarrow WX[t,v] + \sum_{\substack{1\le s<t\\ u\in B_i}} W[s,u,v]X[t-s,u]$
```

Lemma 8.4 *After Phase 2 ends we have for each $v \in B$ and $t \leq \tau$*

$$X[t,v] = x_v^{(t)} \ .$$

Proof: The result follows immediately from Lemma 8.1 and the previous two lemmas. □

Since the number of terms in each of the $T|B|$ sums is at most rT, the total amount of work and I/O's is $O(r|B|\tau^2) = O(kr^2\tau^2) = O(E)$.

Phase 3

The structure of Phase 3 is similar to the structure of Phase 1. The main difference between the two phases is that in Phase 1 a zero was substituted for the state of a blocker during the simulation, whereas in the procedure RelaxG the correct value of the state of blockers is loaded from the table X in secondary memory. The procedure PhaseThree loads each subgraph and its initial state to primary memory, calls RelaG , and then stores back the subgraph with the correct state in the y field.

```
PhaseThree()
for $i \leftarrow 1$ to $k$
    do load $S_i$ into primary memory
       RelaG ()
       store $S_i$ back to secondary memory
```

```
RelaG ()
for $s \leftarrow 1$ to $\tau$
    do for $j \leftarrow 1$ to $M$
          do $S[j].y \leftarrow \sum_{(l,a)\in S[j].Adj} a \cdot S[l].x$
       for $j \leftarrow 1$ to $M$
          do if $S[j].IsBlocker$
                then read $X[S[j].name, s]$ into $S[j].x$
                else $S[j].x \leftarrow S[j].y$
for $j \leftarrow 1$ to $M$
    do if $S[j].IsHome$
          then $S[j].y \leftarrow S[j].x$
```

Lemma 8.5 *After Phase 3 ends, for every $v \in V$ whose home is G_i, the y field in the entry of v in S_i is $x_v^{(\tau)}$.*

Proof: We prove by induction on s that lines 1–7 of RELAXG simulate linear relaxation on G on all the vertices v for which $N_{B_i}^{(s)}(v) \subseteq V_i$ and on all the blockers in B_i. The claim is true for $s = 0$ because before the first iteration the state $S[j].x$ of every vertex is the initial state loaded from secondary memory. Assume that the claim is true for $s < \tau$. In the next iteration $S[j].y$ is assigned the weighted linear combination of all of its neighbors in G_i. If the vertex v is a blocker, its state is loaded from the table X in line 6 and the claim holds. If the vertex is not a blocker but $N_{B_i}^{(s+1)}(v) \subseteq V_i$, then all its neighbors are in G_i, and each neighbor u is either a blocker or $N_{B_i}^{(s)}(u) \subseteq V_i$. In either case, the y field is assigned the weighted linear combination of vertex states which are correct by induction, so its own state is correct.

The lemma follows from the inductive claim, since if $G_i = home(v)$ then $N^{(\tau)}(v) \subseteq V_i$ and therefore its y field is assigned $x_v^{(\tau)}$. □

In Phase 3 each subgraph is loaded into primary memory and RELAXG is called. The total amount of work is $O(k\tau M) = O(\tau E)$ and the total number of I/O's is $O(E + V) = O(E)$.

Summary

The following theorem summarizes the performance of our algorithm.

Theorem 8.6 (Simple Linear Simulation) *Given a graph $G = (V, E)$ with a (τ, r, M)-blocking-cover such that $r^2\tau^2 \leq M$, a computer with $O(M)$ words of primary memory can perform $T \geq \tau$ steps of a simple linear relaxation on G using at most $O(TE)$ work and $O(TE/\tau)$ I/O's. A precomputation phase (which does not depend on the initial state) requires $O(r\tau E)$ work and $O(E)$ I/O's.*

Proof: The correctness of the algorithm follows from Lemma 8.5. The bounds on work and I/O's follow from the performance analysis following the description of each phase. □

The Simple Linear Simulation Theorem applies directly in many situations, but in some special cases which are common in practice, we can improve the performance of our method. In Section 8.5 we will exploit two such improvements to obtain better I/O speedups.

8.5 Multigrid Computations

In this section we present the application of our method to multigrid relaxation algorithms. We show that a two-dimensional multigrid graph (shown previously in Figure 8.1) has a $(\Theta(M^{1/6}), \Theta(M^{1/3}), M)$-blocking-cover, and hence, we can implement a relaxation on the multigrid graph using a factor of $\Theta(M^{1/6})$ fewer I/O's than the naive method. We improve this result to $\Theta(M^{1/5})$ for multigrid computations such as certain elliptic solvers that use only one level of the multigrid graph at a time and have a regular structure of relaxation weights.

Lemma 8.7 *For any $\tau \leq \sqrt{M}$, a 2-dimensional multigrid graph G has a (τ, r, M)-blocking-cover, where $r = O(\tau^2)$.*

Proof: Consider a cover of a multigrid graph in which every $G_i = (V_i, E_i)$ consists of a k-by-k submesh at the bottommost level together with all the vertices above it in the multigrid graph, and the blocking set $B_i \subset V_i$ consists of all the vertices in levels $\ell + 1$ and above. Let each subgraph G_i be the home of all vertices in the inner $(k - \tau 2^{\ell+1})$-by-$(k - \tau 2^{\ell+1})$ bottommost submesh of G_i and all the vertices above them. The number of vertices in B_i is

$$\begin{aligned} r &= \sum_{i=\ell+1}^{(\lg n)/2} \left(\frac{k}{2^i}\right)^2 \\ &< \left(\frac{k}{2^{\ell+1}}\right)^2 \sum_{i=0}^{\infty} 4^{-i} \\ &= \frac{4}{3}\left(\frac{k}{2^{\ell+1}}\right)^2 \end{aligned}$$

and the number of edges in G_i is

$$\begin{aligned} |E_i| &= (2 + 1/4)|V_i| \\ &< \frac{9}{4} \cdot \frac{4}{3} k^2 \\ &= 3k^2 , \end{aligned}$$

since there are at most two edges for each vertex in a mesh, and in the multigrid, $1/4$ of the $O((4/3)k^2)$ vertices have edges connecting to a higher level mesh. Setting $k - \tau 2^{\ell+1} = k/2$, we obtain $k = 4\tau 2^{\ell}$ and $r < (4/3)(2\tau)^2 = (16/3)\tau^2$. Setting $\ell = \frac{1}{2}\lg(M/\tau^2)$, we obtain $|E_i| < 48M$. □

Combining Theorem 8.6 and Lemma 8.7, we obtain the following result.

Corollary 8.8 *A computer with $\Theta(M)$ words of primary memory can perform $T = \Omega(M^{1/6})$ steps of a simple linear relaxation on a $\sqrt{n}$-by-$\sqrt{n}$ multigrid graph using $O(Tn)$ work and $O(Tn/M^{1/6})$ I/O's. A precomputation phase requires $O(M^{1/2}n)$ work and $O(n)$ I/O's.*

Proof: Set $\tau = M^{1/6}$ in Lemma 8.7, and apply Theorem 8.6. □

As a practical matter, linear relaxation on a multigrid graph is not simple: it does not use the same relaxation matrix at each step. Moreover, for many applications, a given step of the relaxation is performed only on a single level of the multigrid or on two adjacent levels.

For example, one generic way to solve a discretized version of a parabolic 2-dimensional heat equation in the square domain $[0,1]^2$, as well as a wide variety of other time-dependent systems of partial differential equations, such as the Navier-Stokes equations, is to use discrete time steps, and in each time step to solve an elliptic problem on the domain. In the heat equation example, for instance, the elliptic problem is

$$\frac{\partial^2 u(x,y,t_i)}{\partial x^2} + \frac{\partial^2 u(x,y,t_i)}{\partial y^2} = \frac{u(x,y,t_i) - u(x,y,t_{i-1})}{t_i - t_{i-1}} .$$

In a common implementation of this strategy, the elliptic solver is a multigrid algorithm, in which case the entire solver can be described as a linear relaxation algorithm on a multigrid graph [24]. This algorithm consists of a number of **cycles**, where each cycle consists of $\Theta(\lg n)$ steps in which the computation proceeds level-by-level up the multigrid and then back down. Since the size of any given level of the multigrid is a constant factor smaller than the level beneath it, the $\Theta(\lg n)$ steps in one cycle of the algorithm execute a total of $\Theta(n)$ work and update each state variable only a constant number of times. Thus, a naive implementation of T cycles of the elliptic solver requires $O(nT)$ work and $O(nT)$ I/O's.

We can use the basic idea behind the simple linear simulation algorithm to provide a more I/O-efficient algorithm. We present the algorithm in the context of a multigrid graph which is used to solve an equation with constant coefficient, but the same algorithm works in other special cases.

Definition We say that a multigrid graph has *regular edge weights* if for every level, the edge weights in the interior of the grid are all identical, the edge weights in the interior of every face (or edge) of the grid are all identical, and if the edge weights going from one level to another are all identical in the interior and all identical along each face.

Theorem 8.9 *A computer with $\Theta(M)$ words of primary memory can perform $T = \Omega(M^{1/5})$ multigrid cycles on a $\sqrt{n}$-by-$\sqrt{n}$ multigrid graph with regular edge weights using $O(nT)$ work, and $O(nT/M^{1/5})$ I/O's. A precomputation step requires $O(M^{8/5})$ work and $O(M)$ I/O's.*

Proof: The algorithm generally follows the simple linear relaxation algorithm. We outline the differences.

The linear simulation algorithm uses a (τ, r, M)-blocking-cover as described in the proof of Lemma 8.7, but we now choose $\tau = M^{1/5}$ and $r = \Theta(M^{2/5})$. Because the relaxation algorithm is not simple, the paths defined by Equation (8.2) must respect the weights defined by the various relaxation matrices. In a single cycle, however, there are only a constant number of relevant state variables for a single vertex. Moreover, the phases can skip over steps corresponding to updates at level $\frac{3}{10}\lg M + 1$ and above, since only blocker variables occupy these levels. Most of these changes are technical in nature, and whereas the bookkeeping is more complicated, we can simulate one cycle of the elliptic solver with asymptotically the same number of variables as the simple linear simulation algorithm uses to simulate one step of a simple linear relaxation problem on the lowest level of the multigrid.

The real improvement in the simulation, however, comes from exploiting the regular structure of the blocking cover of the multigrid graph. The cover has three types of subgraphs: interior ones, boundary ones, and corner ones. All subgraphs of the same type have isomorphic graph structure, the same relaxation weights on isomorphic edges, and an isomorphic set of blockers. Thus, in Phase 0 of the simulation algorithm, we only need to compute the influence on blockers in one representative subgraph of each type. We store these coefficients in primary memory for the entire algorithm, and hence, in Phase 2, we need not perform any I/O's to load them in. Phases 1 and 3 are essentially the same as for simple linear simulation.

The change to Phase 2 is what allows us to weaken the constraint $r^2\tau^2 \leq M$ from Theorem 8.6 and replace it by $r^2\tau \leq M$, which arises since the total work $(r\tau)^2E/M$ in Phase 2 must not exceed $E\tau$ if we wish to do the same work as the naive algorithm. Because all 3 types of subgraphs must fit into primary memory at the same time, the constraint $3r^2\tau \leq M$ also arises. Maximizing τ under the constraints of Lemma 8.7 yields the choice $\tau = M^{1/5}$. The work in Phase 2 is $O((r\tau)^2E/M) = O(M^{1/5}n)$, rather than $O(n)$ as it would be without exploiting the regularity of subgraphs. The number of I/O's in Phase 2 is $O(r\tau)E/M) = O(n)$ in order to input the constants computed in Phase 1 corresponding to the first summation in Equation (8.4). The work in Phases 1 and 3 is $O(\tau E) = O(M^{1/5}n)$, and the number of I/O's is $O(r\tau E/M) = O(n)$. The amount of work in Phase 0 becomes $O(3r\tau M) = O(M^{8/5})$, and the number of I/O's for this precomputation phase is $O(M)$. □

We mention two extensions of this theorem. The 3-dimensional multigrid graph has a (τ, r, M)-blocking-cover, where $r = O(\tau^3)$, which yields an I/O savings of a factor of $\tau = \Theta(M^{1/7})$ over the naive algorithm when τ is maximized subject to the constraint $r^2\tau \leq M$. For the 2-dimensional problem, one can exploit the similarity of the coefficients computed by Phase 0 to save a factor of as much as $\Theta(M^{1/3})$ in I/O's over the naive method, but at the expense of doing asymptotically more work.

8.6 Finding Blocking Covers

In this section, we describe how to find blocking covers for graphs that arise naturally in finite-element computations for physical space. Consequently, I/O efficient linear relaxation schemes exist for these classes of graphs. Specifically, we focus on planar graphs to model computations in two dimensions and d-dimensional simplicial graphs of bounded aspect ratio to model computations in higher dimensions. Planar graphs are those that can be drawn in the plane so that no edges cross. Simplicial graphs arise from dividing d-dimensional space into polyhedra whose aspect ratio is bounded and where the sizes of polyhedra are **locally similar**: the volume of a polyhedron is no larger than twice (or some constant times) the volume of any neighboring polyhedron. Linear relaxation algorithms on such graphs can be used to solve differential equations on various d-dimensional structures [86, 87].

We begin by defining simplicial graphs formally using definitions from [86].

Definition A k**-dimensional simplex**, or k**-simplex**, is the convex hull of $k+1$ affinely independent points in $\Re^d$. A simplicial complex is a collection of simplices closed under subsimplex and intersection. A k**-complex** K is a simplicial complex such that for every k'-simplex in K, we have $k' \leq k$.

A 3-complex is a collection of cells (3-simplexes), faces (2-simplices), edges (1-simplices), and vertices(0-simplices). A d**-dimensional simplicial graph** is the collection of edges (1-simplices) and vertices (0-simplices) in a k-complex in d-dimensions. The **diameter** of a k-complex is the maximum distance between any pair of points in the complex, and the **aspect ratio** is the ratio of the diameter to the kth root of the volume. A simplicial graph of aspect ratio α is a simplicial graph that comes from a k-complex with every k-simplex having aspect ratio at most α.

We now state the main theorems of this section.

Theorem 8.10 *A computer with $\Theta(M)$ words of primary memory can perform $T \geq \tau$ steps of simple linear relaxation on any n-vertex planar graph using $O(nT)$ work and $O(nT/\tau)$ I/O's, where $\tau = O(M^{1/4}/\sqrt{\lg n})$. A precomputation phase requires $O(\tau^2 n \lg n)$ work and $O(n)$ I/O's. Computing the blocking cover requires $O(n \lg n)$ work and I/O's.* □

Theorem 8.11 *A computer with $\Theta(M)$ words of primary memory can perform $T \geq \tau$ steps of simple linear relaxation on any n-vertex d-dimensional simplicial graph of constant aspect ratio using $O(nT)$ work and $O(nT/\tau)$ I/O's, where $\tau \leq M^{\Omega(1/d)}/\lg n$. A precomputation step requires $O(\tau^{\Omega(d)} n \lg^{\Omega(d)} n)$ work and $O(n)$ I/O's. Computing the blocking cover requires $O(n^2/\tau + nM)$ work and I/O's.* □

These theorems follow from the fact that good blocking covers can be found for planar and simplicial graphs by extending the techniques of [64] and [96]. We proceed by stating the definition of a cut cover from [64], and then we relate cut covers to blocking covers. We describe recent results from [64] and [96] that describe how to find good cut covers, and thus, how to find good blocking covers for planar and simplicial graphs.

Given a subgraph $G_i = (V_i, E_i)$ of a graph $G = (V, E)$ with vertex and edge weights $w : V \cup E \rightarrow \{0, 1\}$, we define the weight of G_i as $w(G_i) = \sum_{v \in V_i} w(v) + \sum_{e \in E_i} w(e)$. The following definitions are slight modifications of definitions in [64].

Definition A **balanced** (τ, r, ϵ)**-cut-cover** of a graph $G = (V, E)$ with vertex and edge weights $w : V \cup E \rightarrow \{0, 1\}$ is a triplet (C, G_1, G_2), where $C \subseteq V$ is called a **cut set** and $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are subgraphs of G, such that

CC1. $|C| \leq r$;

CC2. $w(G_1) + w(G_2) \leq (1 + \epsilon) w(G)$;

CC3. $\max(w(G_1), w(G_2)) \leq 2w(G)/3$;

CC4. $\forall v \in V$, either $N_C^{(\tau)}(v) \subseteq V_1$ or $N_C^{(\tau)}(v) \subseteq V_2$.

Definition A **two-color** (τ, r, ϵ)**-cut-cover** for a graph $G = (V, E)$ with two weight functions $w_1, w_2 : V \cup E \rightarrow \{0, 1\}$ is a triplet (C, G_1, G_2) which constitutes a balanced (τ, r, ϵ)-cut-cover for G for both weight functions.

The following theorem relates cut covers to blocking covers.

Theorem 8.12 *If every subgraph of a graph, $G = (V, E)$, has a two-color $(\tau, r, O(1/\lg E))$-cut-cover for any two weight functions, then the graph has a $(\tau, 3r, M)$-blocking-cover.*

Proof: We find a blocking cover by recursively taking two-color cut-covers of subgraphs of G with respect to two weight functions. One weight function w_E assigns weight 1 to each edge in the graph and weight 0 to each vertex. The second weight function w_B assigns 1 to any node that was designated to be a blocker at a higher recursive level and assigns 0 to any other node or edge. That is, we find a two-color $(\tau, r, \epsilon = O(1/\lg E))$-cut-cover (B, G_1, G_2) on the current subgraph, G, and then recurse on each of G_1 and G_2, where w_B and w_E for G_i is inherited from G, except that the new w_B assigns 1 to any element of B in G_i.

We now argue that the set of subgraphs generated at level $\log_{3/2} |E|/M$ of the recursive decomposition is a $(\tau, 3r, M)$-blocking cover of G. The set of subgraphs forms a τ-cover since a (τ, r, ϵ) cut-cover is a τ-cover and successively taking τ-covers yields a τ-cover of the first graph. The number of blockers in any subgraph can be bounded by $3r$ as follows. Assume that at some recursive level, the current subgraph G contains $3r$ blockers from higher recursive levels. Then the number of blockers that G_1 or G_2 contains is less than $(2/3)(3r) + |B| \leq 3r$ by the definition of two-color cut-cover. After $\log_{3/2} |E|/M$ recursive levels, the largest subgraph has at most M edges, since the number of edges in a subgraph is reduced by at least $2/3$ at each recursive level. Finally, the total number of edges in the set of subgraphs is at most

$(1+\epsilon)^{\log_{3/2}(|E|/M)}|E| \le e^{\epsilon \log_{3/2}(|E|/M)}|E| = O(E)$, since the total number of edges does not increase by more than $(1+\epsilon)$ at each recursive level. □

Kaklamanis, Krizanc, and Rao [64] have shown that for every integer ℓ, every n-vertex planar graph has a two-color $(\tau, O(\ell), \tau/\ell)$-cut-cover which can be found in $O(n)$ time. Moreover, Plotkin, Rao, and Smith [96] have recently shown that for every ℓ, every n-vertex d-dimensional simplicial graph of constant aspect ratio has a two-color $O(\tau, O(\ell^{O(d)} \lg n), \tau/\ell)$-cut-cover that can be found in $O(n^2/\ell)$ time.[4] These results can be combined with Theorem 8.12 to yield the following corollaries.

Corollary 8.13 *For every $r > 0$, every n-vertex planar graph has a (τ, r, M)-blocking cover, where $\tau = O(r/\lg n)$.* □

Corollary 8.14 *For every $r > 0$, every n-vertex d-dimensional simplicial graph with constant aspect ratio has a (τ, r, M)-blocking cover, where $\tau = O(r^{\Theta(1/d)}/\lg^{1+\Theta(1/d)} n)$.* □

Corollary 8.13 and Corollary 8.14 can be combined with Theorem 8.6 to prove Theorem 8.10 and Theorem 8.11.

8.7 Discussion

We have presented out-of-core algorithms for linear-relaxation, in particular multigrid algorithms. The asymptotic I/O speedup over a naive 2-dimensional multigrid algorithm is $M^{1/5}$, which is smaller than the asymptotic speedup of most of the covering algorithms and of the other out-of-core algorithms presented in this thesis. An analysis of the constants did not convince me that the algorithm would outperform conventional multigrid algorithms, so I did not implement the algorithm. I still believe that the method has practical value, and that situations in which using it is advantageous will arise in the future.

[4] In fact, they can find cut-covers in any graph that excludes a *shallow* minor. That is, they consider graphs that do not contain K_h as a minor, where the set of vertices in the minor that correspond to a vertex of K_h has small diameter. Our results also hold for this class of graphs.

Chapter 9

An Efficient Out-of-Core Algorithm for Implicit Time-Stepping Schemes in One Dimension

9.1 Introduction

This chapter describes an efficient out-of-core method for performing linear implicit time-stepping computations, which uses the blocking-covers technique described in Chapter 7. The update operator in implicit time-stepping schemes updates the state vector by solving a linear system of equations, often by invoking a direct LU solver. In Section 9.7 we prove that the covering technique cannot be applied to implicit operators when a direct linear equations solver implements the update operator. This chapter shows, however, that a method based on blockers can be applied to implicit time-stepping schemes in one dimension. The method is efficient both in theory and in practice. Section 9.4 proves that the number of I/O's performed by the method is asymptotically optimal for a wide range of problems, and Section 9.5 demonstrates that my implementation of the method can outperform the naive method by more than a factor of 5.

Our method applies to iterative schemes of the form

$$Ax^{(t)} = Bx^{(t-1)} + b^{(t)} \ , \tag{9.1}$$

where A and B are n-by-n tridiagonal matrices and $b^{(t)}$ is a vector whose only nonzeros are its first and last elements. Given A, B, the initial condition $x^{(0)}$, and the boundary conditions $b_1^{(t)}$, $b_n^{(t)}$, for $t = 1, \ldots, T$, we would like to compute $x^{(T)}$. We refer to n as the **size** of the time-stepping scheme.

Equation (9.1) can express a wide range of implicit time-stepping schemes for time-dependent partial differential equations in one spatial dimension, including backward-time-center-space and Crank-Nicolson (see, for example, [91] or [104]) with both Dirichlet and Neumann boundary conditions.

A computer with a large primary memory can implement the scheme (9.1) by invoking a tridiagonal solver in every step. Since we are only interested in the state $x^{(T)}$ after T iterations, the total amount of memory used is $\Theta(n)$ words. The amount of work involved in the computation is $\Theta(nT)$, assuming the use of an efficient variant of Gaussian Elimination as a tridiagonal solver (this variant is sometimes known as Thomas's algorithm [115]).

If the computation does not fit into the primary memory of the computer an out-of-core method must be used. A naive way to implement an out-of-core algorithm is to use the in-core algorithm, which invokes a tridiagonal solver in every iteration, and perform I/O as needed using an automatic paging algorithm. Theorem 9.6 in Section 9.7 proves, however, that any attempt to organize a computation in which any direct solver is applied T times to a state vector of size n on a computer with a primary memory of size M requires at least $T(n-M)$ I/O's. Even for $n = 2M$, the number of I/O's is $\Omega(Tn)$ I/O's, which is the same as the amount of work required by the algorithm. A naive implementation, in which every word used by the CPU is transferred between memories, requires $O(Tn)$ I/O's.

In this chapter I propose an algorithm, which I call the **blocked** out-of-core algorithm, which requires much less I/O. My algorithm can beat the lower bound since it is based on the idea of **blockers**. The algorithm does not invoke a tridiagonal solver on problems of size n, but only on much smaller problems. The main contribution of this chapter is a method for decomposing a problem of size n into k decoupled problems of size $\lfloor (n+1)/k \rfloor - 1$. The domain decomposition scheme is described in Section 9.2, and the decoupling of the subdomains is described in Section 9.3. Our algorithm and analysis of its performance are presented in Section 9.4.

I have implemented the algorithm[1], measured its performance on workstations, and found its performance superior to that of the naive out-of-core algorithm. On a workstation, our implementation of the out-of-core algorithm outperforms the naive algorithm by more than a factor of 5 when the problem does not fit into primary memory. When the problem does fit, the out-of-core algorithm is slower by only a factor of 2.3–3.1. The comparison is described in detail in Section 9.5. We have also used our implementation to investigate the numerical stability of our algorithm, and have found it to be quite stable and accurate, even on very large problems. Our numerical experiments are described in Section 9.6.

9.2 Domain Decomposition

The fundamental operations in the blocked out-of-core algorithm are the decomposition of the state vector into intervals and the derivation of a local update operator for each interval. The structure of these local update operators is similar to the structure of the global update operator.

We decompose the domain into k intervals, which are separated by **blocker** points. That is, every two adjacent intervals are separated by one point which belongs to neither of them. The first $k-1$ intervals have the same length, $m = \lfloor (n+1)/k \rfloor - 1$, but the last interval may be shorter if k does not divide $n+1$. The domain decomposition is illustrated in Figure 9.1.

Given an interval $I = [i, \ldots, j]$, where $i \le j$, we define the **reduced system** with respect to I to be Equation (9.1) restricted to rows i through j, as illustrated in Figure 9.2:

$$A_I x_I^{(t)} = B_I x_I^{(t-1)} + b_I^{(t)} \ . \tag{9.2}$$

[1] The program is available by anonymous FTP from `ftp://theory.lcs.mit.edu/pub/sivan/outofcore.c`, or through the world-wide-web in `http://theory.lcs.mit.edu/~sivan/papers.html`.

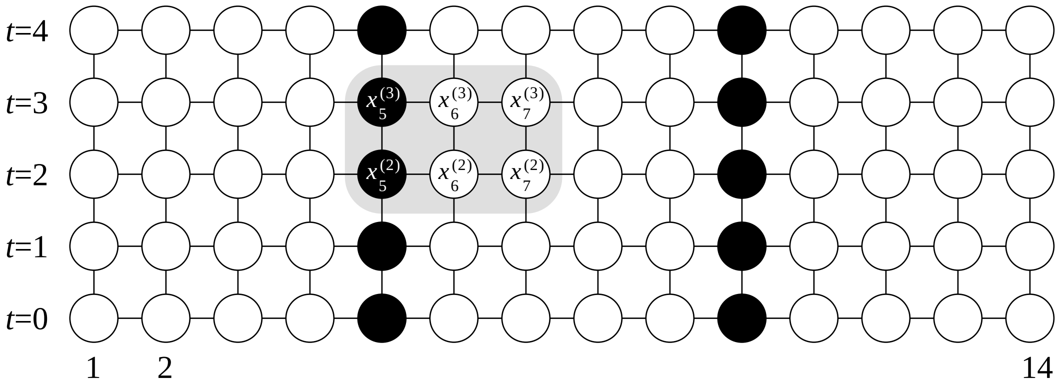


Figure 9.1 A domain of size $n = 14$ is decomposed into $k = 3$ intervals of size $m = 4$ and two blockers, which are represented as black circles. The shaded area surrounds the variables $x_5^{(3)}$, $x_6^{(3)}$, $x_7^{(3)}$, $x_5^{(2)}$, $x_6^{(2)}$, and $x_7^{(2)}$, which are related by one equation in the system $Ax^{(3)} = Bx^{(2)} + b^{(3)}$.

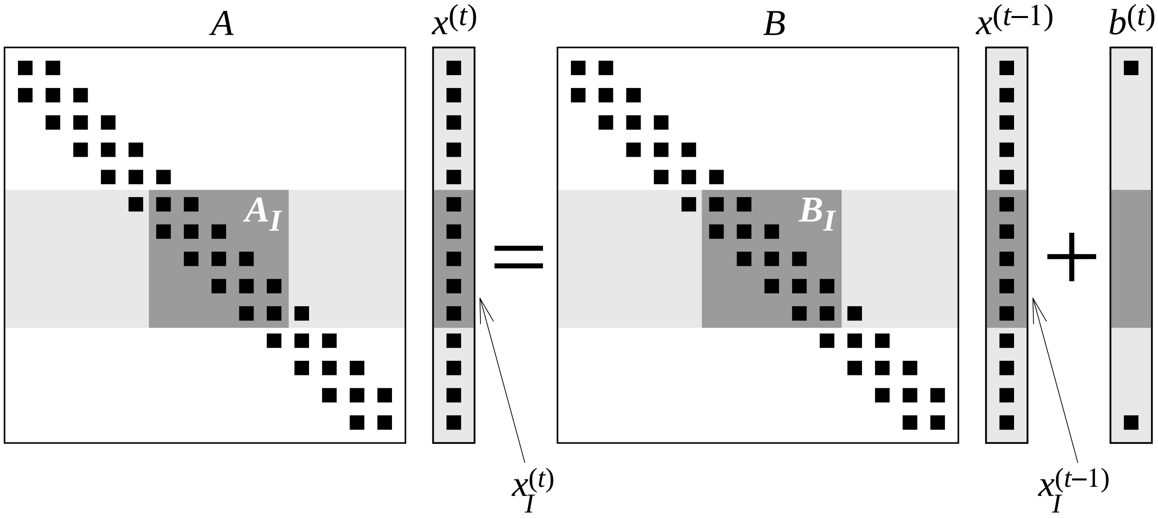


Figure 9.2 The reduced system with respect to $I = [6, \ldots, 10]$ is identical to 6 through 10 in the original system, which are lightly shaded in the figure. The matrices A_I and B_I and the vectors $x_I^{(t)}$ and $x_I^{(t-1)}$ are heavily shaded. The vector $b_I^{(t)}$ is defined so that the reduced system is indeed identical to rows 6 through 10 in the original system.

In Equation (9.2), A_I and B_I are the square diagonal blocks consisting of rows and columns i through j in A and B, and $x_I^{(t)} = (x_i^{(t)}, \ldots, x_j^{(t)})^{\mathrm{T}}$. The vector $b_I^{(t)}$, which is again a vector of zeros except for its first and last entries, is defined in a way which ensures that Equation (9.2) is indeed identical to rows i through j of Equation (9.1). The first entry of $b_I^{(t)}$ is defined as

$$(b_I^{(t)})_1 = \begin{cases} b_1^{(t)} & \text{if } i = 1, \\ -A_{i,i-1}x_{i-1}^{(t)} + B_{i,i-1}x_{i-1}^{(t-1)} & \text{otherwise,} \end{cases} \tag{9.3}$$

and the last entry is defined as

$$(b_I^{(t)})_{j-i+1} = \begin{cases} b_n^{(t)} & \text{if } j = n, \\ -A_{j,j+1}x_{j+1}^{(t)} + B_{j,j+1}x_{j+1}^{(t-1)} & \text{otherwise.} \end{cases} \tag{9.4}$$

If $i = j$, the one element of $b_I^{(t)}$ is defined as the sum of Equations (9.3) and (9.4):

$$(b_I^{(t)})_1 = \begin{cases} b_1^{(t)} - A_{1,2}x_2^{(t)} + B_{1,2}x_2^{(t-1)} & \text{if } i = j = 1, \\ -A_{i,i-1}x_{i-1}^{(t)} + B_{i,i-1}x_{i-1}^{(t-1)} & \\ \quad -A_{j,j+1}x_{j+1}^{(t)} + B_{j,j+1}x_{j+1}^{(t-1)} & \text{if } 1 < i = j < n, \\ b_n^{(t)} - A_{n,n-1}x_{n-1}^{(t)} + B_{n,n-1}x_{n-1}^{(t-1)} & \text{if } i = j = n. \end{cases} \tag{9.5}$$

9.3 Blocker Equations

Our algorithm works by eliminating from the iterative scheme the variables representing points within the intervals. This elimination results in an iterative scheme consisting of blocker points only. We call the equations in these systems of equations the **blocker equations**. This section describes the derivation of the blocker equations.

We begin by representing the state of points within an interval as a function of the state of the blockers that delimit the interval.

Definition Given a point i, its **state sequence** is the vector $x_i = (x_i^{(0)}, x_i^{(1)}, \ldots, x_i^{(t)})^{\mathrm{T}}$.

We express the state sequence of a point p within an interval $I = [i, \ldots, j]$ as a function of the state sequences of the blockers that delimit the interval:

$$x_p = c_p + L_p x_{i-1} + R_p x_{j+1} \; . \tag{9.6}$$

If $i = 1$, then x_{i-1} is taken to be a vector of zeros, and if $j = n$ then x_{j+1} is taken to be a vector of zeros. The vector $c_p = (c_p^{(0)}, \ldots, c_p^{(T)})^{\mathrm{T}}$ is a $(T+1)$-vector, and the L_p and R_p are $(T+1)$-by-$(T+1)$ lower triangular matrices. We denote the (t, r) element of L_p (R_p) by $L_p^{(t,r)}$ ($R_p^{(t,r)}$), where the indices t and r take values from 0 through T. We can define $c_p^{(0)} = x_p^{(0)}$, and we can define the first rows of the matrices as $L_p^{(0,\bullet)} = R_p^{(0,\bullet)} = (0, \ldots, 0)$.

The derivation of the expressions for the rest of the elements of c_p, L_p, and R_p is based on the following lemma.

Lemma 9.1 *The state vector $x^{(t)}$ after t applications of Equation (9.1) satisfies*

$$\begin{aligned} x^{(t)} &= \left(A^{-1}B\right)^t x^{(0)} + \left(A^{-1}B\right)^{t-1} A^{-1}b^{(1)} + \cdots \\ &\quad + \left(A^{-1}B\right) A^{-1}b^{(t-1)} + A^{-1}b^{(t)} \; . \end{aligned} \tag{9.7}$$

Proof: By induction. We have

$$Au^{(1)} = Bx^{(0)} + b^{(1)} \; ,$$

so the basis of the induction holds:

$$u^{(1)} = \left(A^{-1}B\right) x^{(0)} + A^{-1}b^{(1)} \; .$$

Assume that the lemma holds for $t - 1$,

$$\begin{aligned} x^{(t-1)} &= \left(A^{-1}B\right)^{t-1} x^{(0)} + \left(A^{-1}B\right)^{t-2} A^{-1}b^{(1)} + \cdots \\ &\quad + \left(A^{-1}B\right) A^{-1}b^{(t-2)} + A^{-1}b^{(t-1)} \; . \end{aligned}$$

Multiplying Equation (9.1) by A^{-1}, we obtain

$$\begin{aligned} x^{(t)} &= A^{-1}Bx^{(t-1)} + A^{-1}b^{(t)} \\ &= \left(A^{-1}B\right)^t x^{(0)} + \left(A^{-1}B\right)^{t-1} A^{-1}b^{(1)} + \cdots \\ &\quad + \left(A^{-1}B\right) A^{-1}b^{(t-1)} + A^{-1}b^{(t)} \,. \end{aligned}$$

□

To derive the expressions for c_p, L_p, and R_p, we use Lemma 9.1 applied to a reduced system for the interval $I = [i, \ldots, j]$,

$$\begin{aligned} x_I^{(t)} &= \left(A_I^{-1}B_I\right)^t x_I^{(0)} + \left(A_I^{-1}B_I\right)^{t-1} A_I^{-1}b_I^{(1)} + \cdots \\ &\quad + \left(A_I^{-1}B_I\right) A_I^{-1}b_I^{(t-1)} + A_I^{-1}b_I^{(t)} \,. \end{aligned} \tag{9.8}$$

To simplify the notation, we define

$$C_I^{(t)} = \left(A_I^{-1}B_I\right)^t A_I^{-1} \,.$$

Let p be the qth point in the interval $I = [i, \ldots, j]$, that is, $p = i + q - 1$. Equating the right-hand side of row q of Equation (9.8) with the right hand side of the $(t+1)$st row of Equation (9.6), both of which are equal to $x_p^{(t)}$, yields

$$c_p^{(t)} = \begin{cases} \left[\left(A_I^{-1}B_I\right)^t x_I^{(0)}\right]_q & \text{if } i > 1, \\ \left[\left(A_I^{-1}B_I\right)^t x_I^{(0)}\right]_q + \sum_{r=1}^{t} \left[C_I^{(r)}\right]_{q,1} b_1^{(r)} & \text{if } i = 1, \\ \left[\left(A_I^{-1}B_I\right)^t x_I^{(0)}\right]_q + \sum_{r=1}^{t} \left[C_I^{(r)}\right]_{q,j-i+1} b_n^{(r)} & \text{if } j = n; \end{cases} \tag{9.9}$$

$$L_p^{(t,r)} = \begin{cases} 0 & \text{if } t < r \text{ or } i = 1, \\ \left[C_I^{(t-1)}\right]_{q,1} B_{i,i-1} & \text{if } i > 1 \text{ and } t > r = 0, \\ \left[C_I^{(t-r-1)}\right]_{q,1} B_{i,i-1} - \left[C_I^{(t-r)}\right]_{q,1} A_{i,i-1} & \text{if } i > 1 \text{ and } t > r > 0, \\ -\left[C_I^{(0)}\right]_{q,1} A_{i,i-1} & \text{if } i > 1 \text{ and } t = r > 0; \end{cases} \tag{9.10}$$

$$R_p^{(t,r)} = \begin{cases} 0 & \text{if } t < r \text{ or } j = n, \\ \left[C_I^{(t-1)}\right]_{q,j-i+1} B_{j,j+1} & \text{if } j < n \text{ and } t > r = 0, \\ \left[C_I^{(t-r-1)}\right]_{q,j-i+1} B_{j,j+1} & \\ \quad -\left[C_I^{(t-r)}\right]_{q,j-i+1} A_{j,j+1} & \text{if } j < n \text{ and } t > r > 0, \\ -\left[C_I^{(0)}\right]_{q,j-i+1} A_{j,j+1} & \text{if } j < n \text{ and } t = r > 0. \end{cases} \tag{9.11}$$

We now turn to the derivation of the blocker equations themselves, which constitute a sequence of T tridiagonal linear systems of size $(k-1)$-by-$(k-1)$. Consider an equation relating a blocker b, the blocker b_l to its left, and the blocker b_r to its right. (If b is the first or last blocker, we take $x_{b_l}^{(t)} = 0$ or $x_{b_r}^{(t)} = 0$, respectively.) The reduced system for the blocker b is

$$\begin{aligned}
A_{b,b} x_b^{(t)} &= B_{b,b} x_b^{(t-1)} + b_{[b \ldots b]}^{(t)} \\
&= B_{b,b} x_b^{(t-1)} + \\
&\quad (-A_{b,b-1} x_{b-1}^{(t)} + B_{b,b-1} x_{b-1}^{(t-1)}) + \\
&\quad (-A_{b,b+1} x_{b+1}^{(t)} + B_{b,b-1} x_{b+1}^{(t-1)}) \,.
\end{aligned} \tag{9.12}$$

Using Equation (9.6), the expressions for $x_{b-1}^{(t)}$ and $x_{b-1}^{(t)}$ are

$$\begin{aligned}
x_{b-1}^{(t)} &= c_{b-1}^{(t)} + L_{b-1}^{(t,\bullet)} x_{b_l} + R_{b-1}^{(t,\bullet)} x_b \\
&= c_{b-1}^{(t)} + \\
&\quad L_{b-1}^{(t,t)} x_{b_l}^{(t)} + \sum_{r=0}^{t-1} L_{b-1}^{(t,r)} x_{b_l}^{(r)} + \\
&\quad R_{b-1}^{(t,t)} x_b^{(t)} + \sum_{r=0}^{t-1} R_{b-1}^{(t,r)} x_b^{(r)}
\end{aligned}$$

and

$$\begin{aligned}
x_{b+1}^{(t)} &= c_{b+1}^{(t)} + L_{b+1}^{(t,\bullet)} x_b + R_{b+1}^{(t,\bullet)} x_{b_r} \\
&= c_{b+1}^{(t)} + \\
&\quad L_{b+1}^{(t,t)} x_b^{(t)} + \sum_{r=0}^{t-1} L_{b+1}^{(t,r)} x_b^{(r)} + \\
&\quad R_{b+1}^{(t,t)} x_{b_r}^{(t)} + \sum_{r=0}^{t-1} R_{b+1}^{(t,r)} x_{b_r}^{(r)} \,.
\end{aligned}$$

Substituting the expressions for $x_{b-1}^{(t)}$ and $x_{b-1}^{(t)}$ in Equation (9.12), we get

$$\begin{aligned}
A_{b,b-1} L_{b-1}^{(t,t)} x_{b_l}^{(t)} &+ (A_{b,b} + A_{b,b-1} R_{b-1}^{(t,t)} + A_{b,b+1} L_{b+1}^{(t,t)}) x_b^{(t)} + A_{b,b+1} R_{b+1}^{(t,t)} x_{b_r}^{(t)} \\
&= B_{b,b} x_b^{(t-1)} \\
&\quad + B_{b,b-1} (c_{b-1}^{(t-1)} + \sum_{r=0}^{t-1} L_{b-1}^{(t-1,r)} x_{b_l}^{(r)} + \sum_{r=0}^{t-1} R_{b-1}^{(t-1,r)} x_b^{(r)}) \\
&\quad + B_{b,b+1} (c_{b+1}^{(t-1)} + \sum_{r=0}^{t-1} L_{b+1}^{(t-1,r)} x_b^{(r)} + \sum_{r=0}^{t-1} R_{b+1}^{(t-1,r)} x_{b_r}^{(r)}) \\
&\quad - A_{b,b-1} (c_{b-1}^{(t)} + \sum_{r=0}^{t-1} L_{b-1}^{(t,r)} x_{b_l}^{(r)} + \sum_{r=0}^{t-1} R_{b-1}^{(t,r)} x_b^{(r)}) \\
&\quad - A_{b,b+1} (c_{b+1}^{(t)} + \sum_{r=0}^{t-1} L_{b+1}^{(t,r)} x_b^{(r)} + \sum_{r=0}^{t-1} R_{b+1}^{(t,r)} x_{b_r}^{(r)}) \,.
\end{aligned}$$

The collection of these equations for all blockers forms a tridiagonal linear system whose variables are the state of all blockers after time step t, and whose right hand side can be numerically evaluated if the state of all blockers at *previous* time steps is known.

9.4 The Algorithm

In this section we describe our out-of-core algorithm. The algorithm works in three phases. In the first phase, the algorithm computes certain quantities which are needed for computing the blocker equations in the second phase. In Phase 2, the blocker equations are formed and solved, which decouples the intervals from one another. Once the state sequences for the blockers are known, Phase 3 solves the reduced system for each interval to produce the algorithm's output, $x^{(t)}$. The three phases of the algorithm are described in detail and analyzed bellow.

Phase 1

Phase 1 computes the quantities

$$\left(A_I^{-1}B_I\right)^t x_I^{(0)} \ ,$$

$$\left[C_I^{(t-1)}\right]_{1,1} , \left[C_I^{(t-1)}\right]_{1,j-i+1} , \left[C_I^{(t-1)}\right]_{j-i+1,1} \text{ and } \left[C_I^{(t-1)}\right]_{j-i+1,j-i+1}$$

for each interval $I = [i, \dots, j]$ and for $t = 0, \dots, T$.

These quantities are computed by repeated multiplication by B_I and A_I^{-1}, using a tridiagonal solver to multiply by A_I^{-1}. To compute $\left(A_I^{-1}B_I\right)^t x_I^{(0)}$, we start the iteration with $x_I^{(0)}$, and to compute the other four quantities, we start the iteration with the first and last unit vectors in order to compute the first and last columns of $C_I^{(t-1)}$.

The amount of work involved is $3T$ tridiagonal solves using A_I and $3T-2$ multiplications by B_I, which yields $\Theta(mT)$ work per interval and $\Theta(nT)$ work for the whole phase. We assume that for each interval, the whole computation can be carried out *in core*, which requires $\Theta(m)$ words of memory. The number of I/O's is $\Theta(n)$ to load all the A_I's, B_I's and x_I's and $\Theta(kT)$ to store the output of Phase 1.

Phase 2

The goal of Phase 2 is to construct a sequence of T tridiagonal systems of blocker equations of size $(k-1)$-by-$(k-1)$. The solution of the tth system consists of the state of all blockers after time step t. The systems are constructed and solved in succession for $t = 1, \dots, T$.

Forming every equation in the tth time step requires computing $2t+1$ entries of L_{b+1}, L_{b-1}, R_{b+1} and R_{b-1} as well as 2 entries of c_{b-1} and c_{b+1}. Using the output of Phase 1, we can compute the entries of L_{b+1}, L_{b-1}, R_{b+1} and R_{b-1} in constant time, according to Equations (9.10), and (9.11). Computing the entries of c_{b+1} and c_{b+1} requires constant time except for the first and last blockers, where it takes $\Theta(t)$ time using Equation (9.9).

Phase 2 uses two data structures. One is a table that holds the results of Phase 1, and the other is a table that holds the state sequences of all blockers. The size of both data structures is $\Theta(kT)$ words, and each is accessed $\Theta(kT^2)$ times during Phase 2. These data structures can be stored either in primary memory, in which case Phase 2 requires $\Theta(kT)$ words of primary memory and performs $\Theta(kT)$ I/O's, or they can be stored in secondary memory, in which case the phase requires $\Theta(k)$ words of primary memory but performs $\Theta(kT)$ I/O's.

The amount of work in Phase 2 is therefore $\Theta(kT^2)$ to form T systems of blocker equations, and $\Theta(kT)$ to solve the T tridiagonal systems of size $(k-1)$-by-$(k-1)$.

Phase 3

In Phase 3 we compute the state of all points after time step T. The computation is carried out interval by interval. In each interval, we solve the reduced system T times using a tridiagonal solver. Since the state sequence of all blockers is now stored in secondary memory, we can compute numerical values for $b_I^{(t)}$, which essentially decouples the intervals from one another.

The amount of work required in this phase is $\Theta(nT)$ to perform T time steps on each interval. The amount of I/O's is $O(n)$ to load the initial conditions $x_I^{(0)}$ and store the solutions $x_I^{(T)}$, and $\Theta(kT)$ to load the state sequences of blocker which serve as boundary conditions for the intervals.

Summary

We now analyze the total amount of work and I/O required by the algorithm. Phases 1 and 3 perform $\Theta(nT)$ work and $\Theta(n+kT)$ I/O's, and both of them require $\Omega(m)$ primary memory. Phase 2 performs $\Theta(kT+kT^2)$ work. The data structures used in Phase 2 can be stored in either primary or secondary memory, and the choice results in two different tradeoffs between primary memory size and the number of I/O's. Storing the data structures in primary memory requires $\Theta(kT)$ primary memory and I/O's, and storing them in secondary memory reduces the primary memory requirement to $\Theta(k)$ but increasing the number of I/O's to $\Theta(kT^2)$. The following two theorems correspond to the two possibilities.

Theorem 9.2 *A computer with $\Omega(M)$ words of primary memory can perform $T \leq \sqrt{M}$ steps of the implicit time-stepping scheme (9.1) of size $n = O(M^2)$ using $\Theta(n)$ I/O's and $\Theta(nT)$ work.*

Proof: We set $m = M$ so that each interval in Phases 1 and 3 can be computed upon in core. We keep the data structures of Phase 2 in secondary memory. The primary memory requirement of Phase 2 is satisfied, since

$$\begin{aligned} k &= \Theta\left(\frac{n}{M}\right) \\ &\leq \Theta(M)\,. \end{aligned}$$

The amount of I/O performed in Phase 2 is

$$\begin{aligned} O(kT^2) &\leq O(kM) \\ &= O(n)\,. \end{aligned}$$

□

Theorem 9.3 *A computer with $\Omega(M)$ words of primary memory can perform $T \leq \min(M, M^2/n)$ steps of the implicit time-stepping scheme (9.1) of size n using $\Theta(n)$ I/O's and $\Theta(nT)$ work.*

Proof: We set $m = M$ so that each interval in Phases 1 and 3 can be computed upon in core. We keep the data structures of Phase 2 in primary memory. The primary memory requirement of Phase 2 is satisfied,

$$\begin{aligned} kT &\leq \Theta\left(\frac{n}{M}\frac{M^2}{n}\right) \\ &\leq \Theta(M) \,. \end{aligned}$$

The amount of I/O performed in Phase 2 is

$$\begin{aligned} O(kT^2) &\leq O(\frac{n}{M}MT) \\ &= O(nT) \,. \end{aligned}$$

□

Recursive Algorithms

On very large problems, insisting that $m = M$ leads to large data structures in Phase 2 that cause a large number of I/O's. Since Phases 1 and 3 essentially involve performing implicit time-stepping schemes of size m on the intervals, we can employ one of the two algorithms discussed above to execute them. This approach leads to efficient algorithms that use intervals of size $m > M$ and small data structures in Phase 2.

Storing the data structures of Phase 2 in secondary memory and using the algorithm of Theorem 9.2 to perform Phases 1 and 3, we obtain

Theorem 9.4 *A computer with $\Omega(M)$ words of primary memory can perform $T \leq \sqrt{M}$ steps of the implicit time-stepping scheme (9.1) of size n such that $n = O(M^3)$ using $\Theta(n)$ I/O's and $\Theta(nT)$ work.*

Proof: We set $m = M^2$ so that each interval in Phases 1 and 3 can be computed upon using the algorithm of Theorem 9.2. These two phases require $O(mT)$ work and $O(m)$ I/O's per interval, or $O(n)$ work and $O(nT)$ I/O's overall. We keep the data structures of Phase 2 in secondary memory. The primary memory requirement of Phase 2 is satisfied,

$$\begin{aligned} k &= \Theta\left(\frac{n}{m}\right) \\ &\leq \Theta\left(\frac{M^3}{M^2}\right) \\ &= \Theta(M) \,. \end{aligned}$$

The amount of I/O performed in Phase 2 is

$$\begin{aligned} O(kT^2) &\leq O(kM) \\ &= O(\frac{n}{m}M) \end{aligned}$$

Platform	n	Naive Algorithm	Blocked Algorithm
SPARCstation 1+	25K	42.1	98.1
	50K	41.9	98.6
	75K	266.0	108.6
	100K	331.7	123.6
	150K	374.3	123.0
	200K	415.7	125.7
SPARCstation 10	100K	12.3	20.8
	200K	12.7	20.9
	300K	16.7	21.5
	400K	163.1	29.5
	500K	145.2	33.6

Table 9.1 The running times of the naive implementation and my out-of-core blocked algorithm. The table reports the running times per point per iteration in microseconds for various domain sizes n, on a Sun SPARCstation 1+ workstation and on a Fun SPARCstation 10 workstation. The numbers reported are averages of three executions of 25 iterations. The variation between execution times of the same experiment was typically less than 5%. The out-of-core algorithm used intervals of size 10000.

$$= \quad O(\frac{n}{M}) \, .$$

□

One can employ k levels of recursion and hence turn the constraint $n = O(M^3)$ in the theorem into a more relaxed constraint $n = O(M^{2+k})$:

Corollary 9.5 *For any integer $k \geq 1$, a computer with $\Omega(M)$ words of primary memory can perform $T \leq \sqrt{M}$ steps of the implicit time-stepping scheme (9.1) of size n such that $n = O(M^{2+k})$ using $\Theta(n)$ I/O's and $\Theta(nT)$ work.* □

The constants hidden in the big-Θ notations grow with k, so it advisable to use this approach only when the hypotheses of Theorems 9.2 and 9.3 cannot be satisfied.

9.5 Performance

My implementation of the blocked out-of-core algorithm outperforms the naive out-of-core algorithm by a factor of up to 5. On small problems that fit within the primary memory of a workstation, the naive algorithm is 2.3–3.1 times faster than the blocked algorithm. On large problems that do not fit within primary memory, the naive algorithm is slowed down by a factor of up to 11, whereas the blocked algorithm is slowed down very little, as shown in Table 9.1

Experiments were performed on two models of Sun workstations, and with C implementation of the two algorithm using 64-bit floating-point arithmetic. I used both a Sun SPARCstation 10 with 32M bytes of main memory and a 40Mhz SuperSPARC processor, and an older Sun SPARCstation 1+ with 8M bytes of main memory and a Sun 4/60 processor. The program was compiled by the GNU C compiler with a –02 optimization option. Experiments were performed on a system with only one user remotely logged on. The naive algorithm simply

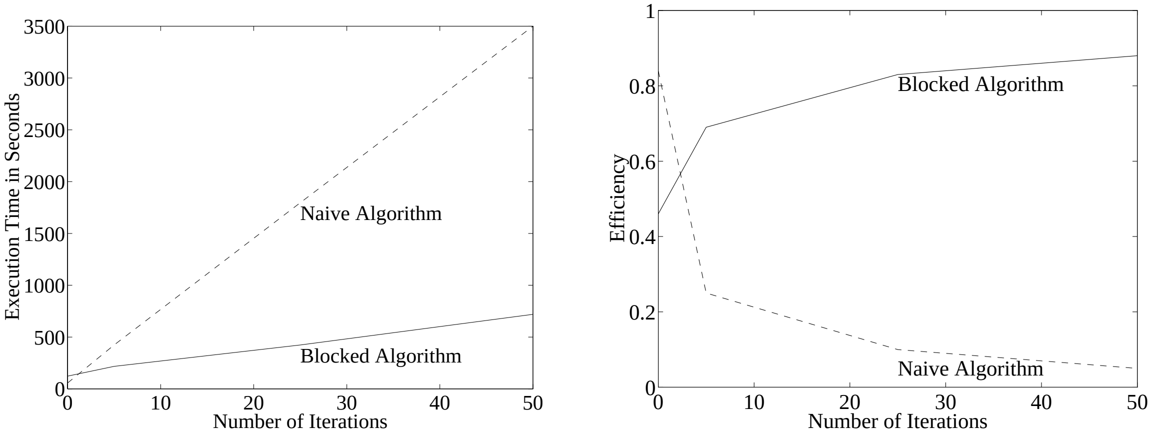


Figure 9.3 The performance of both the naive and the out-of-core algorithms for different numbers of iterations on a SPARCstation 10, on a problem of size $500,000$. The graph on the left shows the running time of the algorithms, which includes both the computing time and the time for performing the I/O. The graph on the right shows the efficiency of the algorithms, which is the ratio of the computing time to the overall running time.

invokes an efficient tridiagonal solver on the entire one-dimensional domain in every iteration. In the blocked algorithm, Phases 1 and 3 are carried out using a tridiagonal solver, rather than by recursion. Both rely on the automatic paging algorithm in the Unix to perform I/O to a local disk, except of course for the input and output of the programs, which is performed explicitly.

The naive algorithm performs well when its data structures fit within primary memory, but its performance is poor when its data structures must be stored on disk. Table 9.1 shows that for large problems its running time per point per iteration, on both platforms, is 10–11 times larger than its running time for small problems. On a SPARCstation 1+, the slowdown is by a factor of more than 7. Figure 9.3 shows that the reason for the slowdown is loss of efficiency. On large problems, the naive algorithm spends less than 10% of its running time computing, and more than 90% performing I/O.

The blocked algorithm performs more work than the naive algorithm, but it performs much less I/O on large problems. It is 2.3–3.1 times slower than then naive algorithm on small problems, when neither algorithm performs any I/O. When invoked to perform 25 iterations on large problems that must be stored on disk, however, it slows down by only a factor of less than 1.2. Figure 9.3 shows that the fraction of its running time spent performing I/O is diminishing as the number of iterations grow. When invoked to perform 50 iterations on large problems, it spends less than 15% of its running time performing I/O, and is more than 5 times faster than the naive algorithm.

The out-of-core blocked algorithm can be restarted every $T = 25$ or $T = 50$ iterations with little gain in efficiency and with potential gains in numerical stability. Figure 9.3 shows that the efficiency of the algorithm is 83% with with $T = 25$ and 88% with $T = 50$. Invoking the algorithm with values of T higher than 25 can only reduce its running time by 17%. The running time can even increase, because the hypotheses of Theorems 9.2 and 9.3 may be violated. On the other hand, as we shall see in the Section 9.6, the numerical accuracy of the algorithm may degrade with T. Restarting the algorithm every 25 iterations or so therefore seems numerically more stable than invoking the algorithm with a high value of T. Finally, the application may require or benefit from saving the state vector $x^{(t)}$ periodically.

9.6 Numerical Stability

The blocked out-of-core algorithm is equivalent to the naive algorithm is exact arithmetic, but in floating-point arithmetic their outputs slowly diverge as the number of iterations grow. Still, on two model problems the outputs agree to within 8 decimal digits even after 25 iterations. This level of accuracy probably suffices for most applications, especially considering that even the output of the naive algorithm typically diverges from the exact solution of the iterative scheme, depending on the condition number of the matrices A and B.

The model problems used in the experiments were two implicit time-stepping schemes for solving the one-dimensional heat equation. The two schemes were used to solve the equation

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}$$

in the interval $(0,1)$, with zero Dirichlet boundary conditions

$$u(0,t) = u(1,t) = 0$$

for all t, and with the initial condition

$$x^{(0)} = \begin{cases} 2x & \text{if } x < 1/2, \\ 1-2x & \text{otherwise.} \end{cases}$$

One time-stepping scheme that approximates the solution of the differential equation is the backward-time-center-space discretization, which leads to a system of equations of the form

$$\frac{\Delta t}{\Delta x^2} x_{i-1}^{(t)} + \left(-2\frac{\Delta t}{\Delta x^2} - 1\right) x_i^{(t)} + \frac{\Delta t}{\Delta x^2} x_{i+1}^{(t)} = -x_i^{(t-1)} \ .$$

The other scheme is the Crank-Nicolson discretization, which leads to a system of equations of the form

$$\begin{aligned} &\frac{\Delta t}{2\Delta x^2} x_{i-1}^{(t)} + \left(-\frac{\Delta t}{\Delta x^2} - 1\right) x_i^{(t)} + \frac{\Delta t}{2\Delta x^2} x_{i+1}^{(t)} = \\ &\qquad \frac{\Delta t}{2\Delta x^2} x_{i-1}^{(t-1)} + \left(\frac{\Delta t}{\Delta x^2} - 1\right) x_i^{(t-1)} + \frac{\Delta t}{2\Delta x^2} x_{i+1}^{(t-1)} \ . \end{aligned}$$

In both cases we have used $\Delta x = \Delta t = 1/(n+1)$.

The numerical experiments used the 64-bit floating-point arithmetic implementation of the algorithms in C, described in Section 9.5, and were performed on Sun SPARCstation workstations.

The experiments measured the difference between the outputs of the two algorithms in two norms: the relative L_2 distance and the maximum pointwise distance. The relative L_2 distance between the output $u^{(T)}$ of the naive algorithm and the output $v^{(T)}$ of the blocked algorithm is

$$\frac{||u^{(T)} - v^{(T)}||_2}{||u^{(T)}||_2} \ .$$

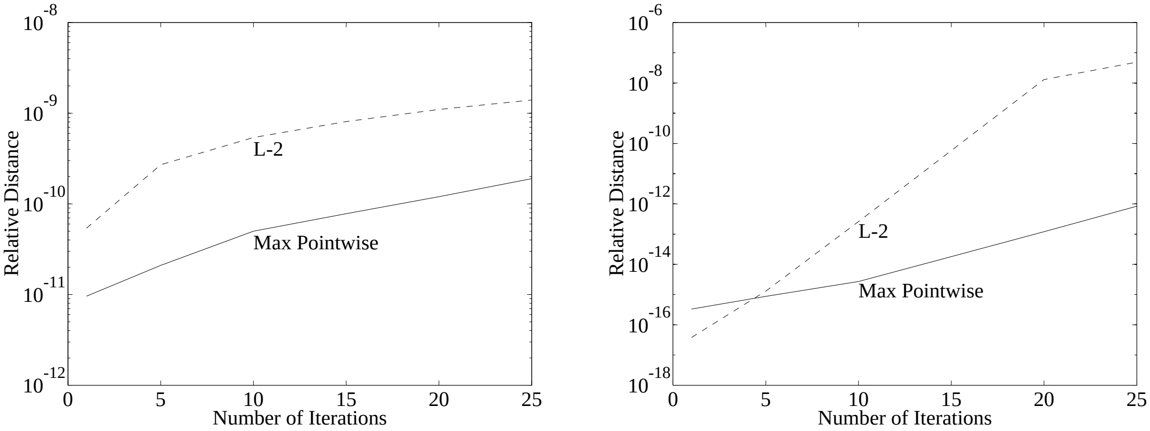


Figure 9.4 The distance of the outputs of the out-of-core and the naive algorithms, relative to the size of the output of the naive algorithm, on model problems of size $n = 500,000$. The blocked out-of-core algorithm used intervals of size $m = 1,000$. Both the L_2 distance and the maximum pointwise distance are shown for the backward-time-center-space scheme on the left and for the Crank-Nicolson scheme on the right.

The maximum pointwise distance between the outputs is

$$\max_i \frac{|u_i^{(T)} - v_i^{(T)}|}{|u_i^{(T)}|} .$$

The results of the experiments, reported in Figure 9.4, indicate that the output of the blocked out-of-core algorithm diverges quite slowly from the output of the naive algorithm. Even after 25 iterations, the outputs agree to within 8 decimal digits on a problem of size $n = 500,000$. I believe that the results indicate that the blocked algorithm is stable enough for most applications, especially if restarted every 25 iterations or so.

9.7 Analysis of the Naive Algorithm

The naive algorithm must perform at least $T(n - M)$ I/O's in the course of T iterations of an implicit time-stepping scheme. My implementation of the naive algorithm is therefore asymptotically optimal, and hence, the comparison between the naive algorithm and the blocked algorithm in Section 9.5 is valid. We define the naive algorithm as one that invokes a direct linear solver in every iteration. In this section we prove that essentially any iterative algorithm that satisfies the assumptions of the red-blue pebble game and invokes a direct solver in every iteration must perform at least $T(n - M)$ I/O's in the course of T iterations. The proof is similar to the proof of Theorem 7.1.

Direct solvers factor the matrix A into $LU = A$, where L is a lower triangular matrix and U is an upper triangular matrix. If A is tridiagonal, then L and U are bidiagonal. The naive method performs an implicit iterative scheme

$$Ax^{(t)} = x^{(t-1)} ,$$

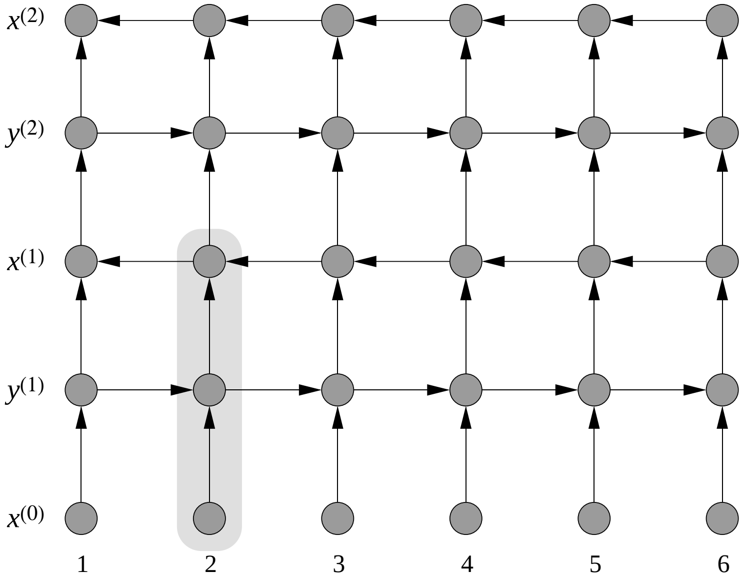


Figure 9.5 A dag that represents two iterations in a one-dimensional implicit time-stepping scheme. In every iteration the state vector $x^{(t)}$ is computed from the previous state vector using a direct linear solver $LUx^{(t)} = x^{(t-1)}$ using an intermediate vector $y^{(t)}$. The matrices L and U are lower and upper bidiagonal, respectively. The shaded area represents the path $C_2^{(1)}$, defined in the proof of Theorem 9.6.

which is even simpler than Equation (9.1) by factoring A and performing two substitution operations in every iteration:

1. $Ly^{(t)} = x^{(t-1)}$, and
2. $Ux^{(t)} = y^{(t)}$.

Figure 9.5 shows the dag associated with the naive algorithm when A is tridiagonal.

We now prove a lower bound on the number of I/O's required by such algorithms the assumptions of the red-blue pebble game. The red-blue pebble game is described in Chapter 7.

Theorem 9.6 *Let D be the dag corresponding to a T-step iterative computation with an n-node state vector $x^{(t)}$ in which the update operator is a direct LU solver. Furthermore, assume that $A^{-1} = U^{-1}L^{-1}$ contains no zeros. Then any algorithm that satisfies the assumptions of the red-blue pebble game requires at least $T(n - M)$ I/O's to simulate the dag D on a computer with M words of primary memory.*

Proof: The red-blue pebble game allows redundant computations, and therefore the state $x_i^{(t)}$ of a vertex i after time step t may be computed more then once during the course of the execution of the algorithm. Let $Time_i^{(t)}$ be the first instant during the execution of the algorithm in which $x_i^{(t)}$ is computed. We denote by $Time^{(t)}$ the first instant in which the state of any vertex after time step t is computed; which is to say

$$Time^{(t)} = \min_{i\in[1,\ldots,n]} \{ Time_i^{(t)} \} .$$

Since $x^{(t+1)} = A^{-1}x^{(t)}$ and since A^{-1} has no zeros, the state $x_i^{(t+1)}$ of each vertex i after iteration $t + 1$ depends on the state of all vertices after iteration t. Therefore, we deduce that $Time^{(0)} < Time^{(1)} < \cdots < Time^{(T)}$ and that algorithm must compute the state of all vertices after iteration t between $Time^{(t)}$ and $Time^{(t+1)}$.

Since

$$y_i^{(t)} = \frac{1}{l_{i,i}} x_i^{(t-1)} - \sum_{j=1}^{i-1} \frac{l_{i,j}}{l_{i,i}} y_j^{(t-1)}$$

and

$$x_i^{(t)} = \frac{1}{u_{i,i}} y_i^{(t)} - \sum_{j=i+1}^{n} \frac{u_{i,j}}{u_{i,i}} x_j^{(t)} \ ,$$

$x_i^{(t-1)}$ is a direct predecessor of $y_i^{(t)}$ in the dag, and $y_i^{(t)}$ is a direct predecessor of $x_i^{(t)}$.

Let $C_i^{(t)}$ be the path $x_i^{(0)} \rightarrow y_i^{(1)} \rightarrow x_i^{(1)} \rightarrow \cdots y_i^{(t)} \rightarrow x_i^{(t)}$ in the dag. In Figure 9.5, the path $C_2^{(1)}$ is represented by a shaded area. If $x_i^{(t)}$ is computed between two time points $Time^{(t)}$ and $Time^{(t+1)}$, then we know that either a vertex in $C_i^{(t)}$ was in memory at $Time^{(t)}$ or one I/O was performed between $Time^{(t)}$ and $Time^{(t+1)}$ in order to bring some vertex in $C_i^{(t)}$ into primary memory.

The vertex sets $C_i^{(t)}$ and $C_j^{(t)}$ are disjoint for $i \neq j$. Since primary memory at time $Time^{(t)}$ can contain at most M vertices, one vertex from at least $n - M$ chains $C_i^{(t)}$ must be brought from secondary memory between $Time^{(t)}$ and $Time^{(t+1)}$. Summing over all iterations, we conclude that the algorithm must perform at least $T(n - M)$ I/O's. □

Theorem 9.6 applies to any iterative scheme of the form $LUx^{(t)} = x^{t-1}$ that uses substitution to solve the two triangular systems, including implicit time stepping-schemes as well as SSOR relaxation algorithms. Hong and Kung proved a slightly weaker bound for a dag identical to the dag in Figure 9.5 [62] (they call it the snake-like graph), but they did not relate this dag to an iterative invocation of a direct linear solver. By decomposing the problem into decoupled problems of smaller size, our blocked out-of-core algorithm beats this lower bound and outperforms any algorithm based on invoking a direct solver in every iteration.

9.8 Discussion

This chapter presented an efficient out-of-core algorithm for iterative, implicit time-stepping schemes arising from the discretization of one-dimensional partial differential equations. We have implemented the algorithm and measured its performance on a workstation. Our implementation performs about 2.3–3.1 times more work than the naive algorithm (for $T = 25$), but requires much less I/O when the entire problem does not fit into primary memory, leading to speedups of more than 5 over the naive algorithm. Our floating-point implementation proved to be accurate and stable in numerical experiments, even on very large problems.

Our implementation relies on an automatic paging algorithm to perform I/O, rather than perform the I/O explicitly. This design choice simplified the implementation and allows the implementation to be used in situations where explicit I/O is not supported (e.g. cache memories), while still providing excellent performance.

Our algorithm can be modified to handle non constant matrices A and B, that is, a different pair of matrices in every iteration. Apart from the I/O or computation necessary to generate the matrices in every iteration, the asymptotic performance of the algorithm remains the same. Such an algorithm makes sense mostly in cases in which generating the matrices does not require much I/O. The algorithm can also be modified to handle a constant forcing term.

Chapter 10

Efficient Out-of-Core Krylov-subspace Methods

10.1 Introduction

This chapter uses the blocking covers technique to derive efficient out-of-core implementations for a class of popular iterative methods, Krylov-subspace methods. This special case of the blocking covers technique yields simple algorithms that outperform on workstations the naive algorithms by more than a factor of 3. The implementation and performance of my algorithms are described in Sections 10.5 and 10.6. My empirical studies, described in Section 10.4, show that the method is numerically stable.

Krylov-subspace methods are a class of iterative numerical methods for solving linear equations. Given an n-by-n matrix A and an n-vector b, Krylov-subspace methods compute a sequence of approximations $\{x^{(t)}\}$ to the solution x of the linear system of equations $Ax = b$, such that $x^{(t)} \in \text{span}\{b, Ab, A^2b, \ldots, A^tb\}$. The subspace $\mathcal{K}_t(A, b) = \text{span}\{b, Ab, A^2b, \ldots, A^tb\}$ is called a **Krylov** subspace. Krylov-subspace methods exist for eigenproblems as well. For recent surveys of Krylov-subspace methods, see [11, 48].

Most Krylov-subspace methods involve several types of operations in every iteration, namely multiplication of a vector by the matrix A, vector operations (additions and multiplication by scalars), and inner products. The iteration may also include preconditioning, which consists of solving a linear system of equations $Mz = r$. This chapter does not consider preconditioned Krylov-subspace methods. Almost all Krylov-subspace methods maintain a number of n-vectors in addition to $x^{(t)}$. Many reuse the space used to store these vectors, so the space required is only a small multiple of n words.

Computing inner products in every iteration cause Krylov-subspace algorithms to satisfy the hypothesis of Theorem 7.1. In the most popular Krylov-subspace method, conjugate gradient, two inner products are computed in every iteration, and their location in the iteration necessitates reading the data twice in every iteration into primary memory. Several papers [10, 85, 100] propose algorithms that reorganize the conjugate gradient algorithm so that the data can be read into primary memory only once in every iteration. But since both the original and the modified conjugate gradient algorithm satisfy the hypothesis of Theorem 7.1, they require $\Omega(n)$ I/O's per iteration when $n > 2M$, where M is the size of primary memory. The matrix-vector multiplication step may require additional I/O's.

Chronopoulos and Gear [31] proposed the first multistep Krylov-subspace algorithms, which works in **phases**, each of which is mathematically equivalent to τ conjugate gradient iterations. In their method, called the s-step method (we use τ instead of s to denote the number of iterations in a phase), the 2τ inner products required for a phase are computed together at the beginning of the phase. Unfortunately, their method maintains two matrices of size n-by-τ which must be updated in every phase. Therefore, if these matrices do not fit into primary memory, the s-step method requires $\Omega(\tau n)$ I/O's per phase, which is asymptotically the same amount of I/O required by single-step methods. Their method has several further drawbacks. First, their algorithms perform asymptotically more work per iteration than single-step algorithms. For example, their τ-step conjugate-gradient algorithm requires $\tau + 1$ multiplications by A and $\Theta(\tau^2 n + \tau^3)$ work, compared to τ multiplications by A and $\Theta(\tau n)$ work required by the single-step algorithm. Second, their algorithms may suffer from numeric instabilities since the basis for the Krylov-subspace they compute is $[b, Ab, A^2 b, \ldots, A^\tau b]$. In floating point, this basis is not stable since as τ grows, the vector $A^\tau b$ tends to an eigenvector associated with the largest eigenvalue of A. This instability further limits the value of τ that can be used in the method. In particular, Chronopoulos and Gear use $\tau = 5$. Finally, the derivation of their algorithms follows the derivations of the corresponding single-step algorithms, and is therefore quite involved, which is a disadvantage given the growing number of modern Krylov-subspace methods.

We propose a new method for deriving multistep Krylov-subspace algorithms that are simpler and more efficient, both in terms of I/O and in terms of work. We also present a way to stabilize the method, at the expense of performing more work. Our method produces algorithms which are more efficient than the ones proposed by Chronopoulos and Gear. For example, our conjugate gradient algorithm requires reading the data once, 4τ multiplications by A, and $\Theta(\tau n + \tau^2 \log \tau)$ work, per each τ iterations. Since in our method the work per iteration is almost independent of τ (when n is much larger than τ), its performance improves monotonically with τ. Our method is based on a simple basis change in the algorithm. Consequently, the method is easy to apply to virtually any single-step Krylov-subspace method. For Krylov-subspace algorithms with long recurrences, such as GMRES(τ), our method provides the first implementation that requires less than τn words of storage.

Out-of-core Krylov subspace methods require that the matrix-vector subroutines used have special properties, which are described in Section 10.2. Section 10.3 describes our multistep Krylov-subspace methods, and illustrates them by describing in detail our out-of-core conjugate gradient algorithm. The numerical stability problem raised by the algorithm, along with a remedy, are studied in Section 10.4. Sections 10.5 and 10.6 describe our implementation of out-of-core conjugate gradient algorithms and their performance. We conclude this chapter with a discussion of our method in Section 10.7.

10.2 Matrix Representations for Krylov-subspace Methods

This section specifies the characteristics of matrix-vector multiplication mechanisms required by out-of-core Krylov-subspace methods. All Krylov-subspace algorithms require a mechanism for multiplying a vector by the matrix A, which is used to construct a basis for the Krylov subspace. Out-of-core Krylov-subspace algorithms require that the mechanism for multiplying a vector by A has certain special characteristics.

In particular, we need to multiply a *section* of a vector by A repeatedly, so we start with two definitions that describe partitions of vectors into sections.

Definition A **k-partition** of $\{1, 2, \ldots, n\}$ is a sequence of index sets $\langle P_1, P_2, \ldots, P_k\rangle$ such that

$$P_j \subseteq \{1,2,\ldots,n\} \ ,$$

$$\bigcup_{j=0}^{k} P_j = \{1,2,\ldots,n\} \ ,$$

and

$$P_i \cap P_j = \emptyset \text{ for } i \neq j \ .$$

Definition A **k-cover** of $\{1, 2, \ldots, n\}$ is a sequence of index sets $\langle P_1, P_2, \ldots, P_k\rangle$ and a sequence $\langle C_1, C_2, \ldots, C_k\rangle$ such that

$$P_j \subseteq C_j \subseteq \{1,2,\ldots,n\} \ ,$$

$$\bigcup_{j=0}^{k} P_j = \bigcup_{j=0}^{k} C_j = \{1,2,\ldots,n\} \ ,$$

and

$$P_i \cap P_j = \emptyset \text{ for } i \neq j \ .$$

We now turn to the specific characteristics of the matrix-vector multiplication mechanism. We start with a simple mechanism which is powerful enough for many out-of-core algorithms, and then move to a more sophisticated mechanism.

Definition A **(τ, M)-partitioned linear relaxation** algorithm for an n-by-n matrix A is a software module $\mathcal{A}$ that provides the following two services on a computer with $O(M)$ words of primary memory. Service 1 must be called before Service 2.

1. Given two integers n and τ, $\mathcal{A}$ computes a k-partition of $\{1, 2, \ldots, n\}$.

2. Given an n-vector x in secondary memory, $\mathcal{A}$ can be called $k(\tau + 1)$ times, returning in primary memory the sequence $\{x_{P_i}, (Ax)_{P_i}, (A^2x)_{P_i}, \ldots, (A^\tau x)_{P_i}\}$ for any $i \in \{1, \ldots, k\}$.

We denote the total amounts of work and I/O's required by the algorithm by work$(\mathcal{A}, n, \tau)$ and io$(\mathcal{A}, n, \tau)$, respectively.

For example, Chapter 8 presents a (τ, M)-partitioned linear relaxation algorithm for matrices whose underlying graph is a two-dimensional multigrid graph, using $\Theta(n)$ I/O's and $\Theta(\tau n)$ work, for $\tau = \Omega(M^{1/5})$.

Many partitioned matrix-vector multiplication algorithms have the additional feature that if, together with the parameters n and τ, they are also given a family of polynomials $\phi_0, \phi_1, \ldots, \phi_\tau$, where ϕ_t is a degree-t polynomial, then they return $\{(\phi_0(A)x)_{P_i}, (\phi_1(A)x)_{P_i}, \ldots, (\phi_\tau(A)x)_{P_i}\}$ instead of x multiplied by powers of A. In some cases the polynomials are not known in advance, however, and in such cases we need a more flexible mechanism, which the next definition describes.

Definition A (τ, M)**-covered linear relaxation** algorithm for an n-by-n matrix A is a software module $\mathcal{A}$ that provides the following two services on a computer with $O(M)$ words of primary memory. Service 1 must be called before Service 2.

1. Given two integers n and τ, $\mathcal{A}$ computes a k-cover $\langle P_1, P_2, \ldots, P_k\rangle$, $\langle C_1, C_2, \ldots, C_k\rangle$, of $\{1, 2, \ldots, n\}$ such that $\max_{i=0}^{k} |C_i| \le M/(\tau + 1)$.

2. Given a vector section $x^{(0)}_{C_i}$ in primary memory, $\mathcal{A}$ can be called up to τ times. For the tth call, $\mathcal{A}$ is given a degree-t polynomial ϕ_t, and a block of memory of size $\max_{i=0}^{k} |C_i|$. The tth call then returns in that block $(\phi_t(A)x)_{P_i}$. Following the last call, the blocks can be reclaimed.

We denote the amounts of work required by the algorithm by work$(\mathcal{A}, n, \tau)$, and define io$(\mathcal{A}, n, \tau) = \sum_{i=1}^{k} |C_i|$.

The blocks of primary memory are necessary for the algorithm to store previous iterates. The polynomial $\phi_t(A)$ can be given by its coefficients, or better yet, as a linear combination of $\phi_1(A), \ldots, \phi_{t-1}(A)$ and $A\phi_{t-1}(A)$. For example, Hong and Kung [62] analyze (τ, M)-covered matrix-vector multiplication algorithms for matrices whose underlying graphs are two-dimensional meshes, using $\Theta(n)$ I/O's and $\Theta(\tau n)$ work, for $\tau = \Omega(M^{1/2})$. A graph theoretic construction known as a τ-neighborhood-cover [6] can be used to construct covered matrix-vector multiplication algorithms.

10.3 Change of Basis in Krylov-subspace Methods

This section describes our multistep Krylov-subspace algorithms. Applying a simple linear transformation to Krylov-subspace algorithms enables us to beat the lower bound of Theorem 7.1 that normally applies to these algorithms. We describe in detail one algorithm, a multistep conjugate gradient algorithm, which is an algorithm for finding an exact, or more typically an approximate, solution to a linear system of equations $Ax = b$, where A is symmetric positive definite. Other algorithms can be derived using the same basis transformation.

The pseudocode below describes the conjugate gradient algorithm, which is due to Hestenes and Stiefel [58]. The loop structure has been modified slightly from the original algorithm to make the first iteration of the algorithm identical to the rest. The algorithm is called with an initial guess x, the initial residual $r = b - Ax$, and an initial search direction $p = r$. Invoking the algorithm once with $\tau = T$ is equivalent to invoking the algorithm k times, each time using $\tau = T/k$ and the last values of x, r, p, and $r^{\mathrm{T}}r$ computed (except in the first invocation where the initial values are described above). All vectors in our codes are passed by reference.

```
ConjugateGradient($A, x, r, p, r^{\mathrm{T}}r, \tau$)
for $t \leftarrow 1$ to $\tau$
    do Compute and store $Ap$
       Compute $p^{\mathrm{T}}Ap$
       $\alpha \leftarrow r^{\mathrm{T}}r / p^{\mathrm{T}}Ap$
       $x \leftarrow x + \alpha p$
       $\rho \leftarrow r^{\mathrm{T}}r$
       $r \leftarrow r - \alpha Ap$
       Compute $r^{\mathrm{T}}r$
       $\beta \leftarrow r^{\mathrm{T}}r / \rho$
       $p \leftarrow r + \beta p$
```

In the following pseudocode due to [10, 85, 100], the inner products have been brought closer together, to enable efficient out-of-core implementations. To overcome numerical stability problems, this implementation of the conjugate-gradient algorithm maintains a fourth vector, denoted by Ap, and computes an additional inner product in every iteration. This algorithm computes an additional inner product for numerical stability reasons, and it maintains a fourth vector Ap. A slightly different variant can be found in [31]. To begin, the algorithm is called with an initial guess x and with $p = r = b - Ax$.

```
ClusteredDotProdsCG($A, x, r, p, Ap, p^{\mathrm{T}}Ap, (Ap)^{\mathrm{T}}(Ap), r^{\mathrm{T}}r, \tau$)
for $t \leftarrow 1$ to $\tau$
    do $\alpha \leftarrow r^{\mathrm{T}}r / p^{\mathrm{T}}Ap$
       $\beta \leftarrow \left(\alpha^2 (Ap)^{\mathrm{T}}(Ap) - r^{\mathrm{T}}r\right) / r^{\mathrm{T}}r$
       $x \leftarrow x + \alpha p$
       $r \leftarrow r - \alpha Ap$
       $p \leftarrow r + \beta p$
       Compute and store $Ap$
       Compute $p^{\mathrm{T}}Ap$, $(Ap)^{\mathrm{T}}(Ap)$, $r^{\mathrm{T}}r$
```

The following theorem states the performance of the ClusteredDotProdsCG algorithm. In practice, x, r, and Ap are loaded and stored exactly once per iteration, and p is stored once and loaded once in a way that allow it to be multiplied by A (which usually means that more than n I/O's are needed). The number of vector operations, both inner products and vector updates, is 6.

Theorem 10.1 *Given a $(1, M)$-partitioned matrix-vector multiplication algorithm $\mathcal{A}$ for an n-by-n matrix A, a computer with $O(M)$ words of primary memory can perform T iterations of algorithm* ClusteredDotProdsCG *using*

$$\Theta(Tn) + T \cdot \text{work}(\mathcal{A}, n, 1)$$

work and

$$7Tn + T \cdot \text{io}(\mathcal{A}, n, 1)$$

I/O's. □

Unfortunately, the amount of I/O required per iteration of the CLUSTEREDDOTPRODSCG cannot be asymptotically reduced using scheduling techniques, even if A is the identity matrix. From Theorem 7.1 it follows that as long as the algorithm maintains a state vector of size n, and that every element of the state vector depends directly on the same element in the previous iteration and on all the elements in the previous iteration through a summation, the number of I/O's per iteration is at least $n - M$.

Using a more powerful technique, we do improve the asymptotic performance of CONJUGATEGRADIENT. In particular, we perform a linear transformation on the algorithm. Let us denote the initial values of x, r, and p passed to the algorithm by x_0, r_0, and p_0. One can prove by induction that during the algorithm,

$$x \in x_0 + \text{span}\{r_0, p_0, Ar_0, Ap_0, \ldots, A^{\tau} r_0, A^{\tau} p_0\} \ ,$$

$$r \in \text{span}\{r_0, p_0, Ar_0, Ap_0, \ldots, A^{\tau} r_0, A^{\tau} p_0\} \ ,$$

$$p \in \text{span}\{r_0, p_0, Ar_0, Ap_0, \ldots, A^{\tau} r_0, A^{\tau} p_0\} \ .$$

We therefore make a basis change in the algorithm and represent r, p, and the change in x in the **Krylov basis** $B = [r_0, p_0, Ar_0, Ap_0, \ldots, A^{\tau} r_0, A^{\tau} p_0]$. In particular, rather than compute on x, r, and p, we perform equivalent computations on the vectors w_x, w_r, and w_p, such that $x = x_0 + Bw_x$, $r = Bw_r$, and $p = Bw_p$. In the new basis, multiplication by scalars and addition of vectors remain unchanged. For two vectors y, z in the standard basis, the inner product $y^{\mathrm{T}} z$ is computed using the identity $y^{\mathrm{T}} z = w_y^{\mathrm{T}} B^{\mathrm{T}} B w_z$. Multiplication of a vector by a A is replaced by shifting the vector down by two places and filling the two uppermost positions with zeros. In other words, the representation of A in the basis B is

$$\begin{pmatrix} 0 & & & & & \\ 0 & 0 & & & & \\ 1 & 0 & 0 & & & \\ 0 & 1 & 0 & 0 & & \\ & & \ddots & & \ddots & \\ & & 0 & 1 & 0 & 0 \end{pmatrix}$$

It follows that if we precompute the $(2\tau + 2)$-by-$(2\tau + 2)$ matrix $B^{\mathrm{T}} B$, which we denote by Q, we can perform the conjugate gradient in the Krylov basis.

```
KrlovBasisConjugateGradient ($A, x, r, p, r^{\mathrm{T}}r$)
 1  Compute $Q = B^{\mathrm{T}}B$, where $B = [r, p, Ar, Ap, \ldots, A^{\tau}r, A^{\tau}p]$
 2  $w_r \leftarrow [1, 0, 0, 0, \ldots, 0, 0]^{\mathrm{T}}$
 3  $w_p \leftarrow [0, 1, 0, 0, \ldots, 0, 0]^{\mathrm{T}}$
 4  $w_x \leftarrow [0, 0, 0, 0, \ldots, 0, 0]^{\mathrm{T}}$
 5  for $t \leftarrow 1$ to $\tau$
 6      do $w_{Ap} \leftarrow$ shift-down-by-2($w_p$)
 7         $p^{\mathrm{T}}Ap \leftarrow w_p^{\mathrm{T}}Qw_{Ap}$
 8         $\alpha \leftarrow r^{\mathrm{T}}r/p^{\mathrm{T}}Ap$
 9         $w_x \leftarrow w_x + \alpha w_p$
10         $w_r \leftarrow w_r - \alpha w_{Ap}$
11         $r^{\mathrm{T}}r \leftarrow w_r^{\mathrm{T}}Qw_r$
12         $\beta \leftarrow r^{\mathrm{T}}r/(r^{(t)})^{\mathrm{T}}r^{(t)}$
13         $w_p \leftarrow w_r + \beta w_p$
14  $r \leftarrow Bw_r$
15  $p \leftarrow Bw_p$
16  $x \leftarrow x + Bw_x$
```

The following theorem describes the characteristics of a simple implementation of the algorithm.

Theorem 10.2 *Given a (τ, M)-partitioned matrix-vector multiplication algorithm $\mathcal{A}$ for an n-by-n matrix A, a computer with $O(M)$ words of primary memory, where $M = \Omega(\tau)$, can perform T iterations of algorithm* KrylovBasisConjugateGradient *using*

$$\Theta(Tn + T\tau\log\tau) + (2T/\tau)\left(\text{work}(\mathcal{A}, n, 2\tau) + \text{work}(\mathcal{A}, n, \tau)\right)$$

work and

$$(6T/\tau)n + (2T/\tau)\left(\text{io}(\mathcal{A}, n, 2\tau) + \text{io}(\mathcal{A}, n, \tau)\right)$$

I/O's.

Proof: The correctness of the algorithm follows from the discussion above. The bounds in the theorem are achieved by calling the algorithm T/τ times. The main difficulty in the algorithm is the implementation of line 1 and lines 14–16.

To compute Q, we invoke the partitioned matrix-vector multiplication algorithm on each of the k sets in the partition. For each set P_i, we first generate $\{(Ar)_{P_i}, (A^2r)_{P_i}, \ldots, (A^{2\tau}r)_{P_i}\}$ and $\{(Ap)_{P_i}, (A^2p)_{P_i}, \ldots, (A^{2\tau}p)_{P_i}\}$. For $m = 0, \ldots, 2\tau$, we multiply $r_{P_i}^{\mathrm{T}}$ by $(A^m r)_{P_i}$ and by $(A^m p)_{P_i}$, as well as $p_{P_i}^{\mathrm{T}}$ by $(A^m p)_{P_i}$. Once we have done so for each set in the partition, we sum up the partial sums to form $r^{\mathrm{T}}A^m r$, $r^{\mathrm{T}}A^m p$, and $p^{\mathrm{T}}A^m p$, for $m = 0, \ldots, 2\tau$. The symmetric matrix Q is then formed using the identity

$$Q_{i,j} = \begin{cases} r^{\mathrm{T}}A^{k+l-2}r & i = 2k, j = 2l \\ r^{\mathrm{T}}A^{k+l-2}p & i = 2k, j = 2l+1 \\ p^{\mathrm{T}}A^{k+l-2}r = r^{\mathrm{T}}A^{k+l-2}p & i = 2k+1, j = 2l \\ p^{\mathrm{T}}A^{k+l-2}p & i = 2k+1, j = 2l+1\ . \end{cases}$$

Therefore, computing Q requires $\Theta(\tau n)$ work for computing the $3(2\tau + 2)$ inner products of size n, in addition to $2 \cdot \text{work}(\mathcal{A}, n, 2\tau)$ work, and $2 \cdot \text{io}(\mathcal{A}, n, 2\tau)$ I/O's.

Lines 14, 15, and 16 are performed in a similar manner by calling the matrix multiplication algorithm on each set in the partition, to compute

$$r_{P_i} \leftarrow [r_{P_i}, p_{P_i}, (Ar)_{P_i}, (Ap)_{P_i}, \ldots, (A^\tau r)_{P_i}, (A^\tau p)_{P_i}] w_r \ .$$

The sections x_{P_i} and p_{P_i} are computed in the same way. The amount of work required for each of the three lines is $\Theta(\tau n)$ to update the vectors x, r, and p, and $2\text{work}(\mathcal{A}, n, \tau)$ for recreating B. The amount of I/O required is $2\text{io}(\mathcal{A}, n, \tau)$ for creating B, and $3n$ for saving the final values of x, r, and p.

A final detail involves the multiplication of Q by w_{Ap} in line 7 and by w_r in line 11. A general dense matrix-vector multiplication algorithm costs $O(\tau^2)$ work per invocation, since Q is $(2\tau + 2)$-by-$(2\tau + 2)$. The cost can be reduced by using the Fast Fourier Transform algorithm. The matrix B can be decomposed into two matrices B_{even} and B_{odd} containing the even and odd columns of B, respectively (that is , "p" columns and "r" columns). We denote $w_y = w_{y,even} + w_{y,odd}$, so that $y = Bw_y = B_{even} w_{y,even} + B_{odd} w_{y,odd}$. Therefore

$$\begin{aligned} z^{\mathrm{T}} y \ = \ & w_{z,even}^{\mathrm{T}} B_{even}^{\mathrm{T}} B_{even} w_{y,even} \\ & + w_{z,odd}^{\mathrm{T}} B_{odd}^{\mathrm{T}} B_{even} w_{y,even} \\ & + w_{z,even}^{\mathrm{T}} B_{even}^{\mathrm{T}} B_{odd} w_{y,odd} \\ & + w_{z,odd}^{\mathrm{T}} B_{odd}^{\mathrm{T}} B_{odd} w_{y,odd} \ . \end{aligned}$$

Since A is symmetric, the three matrices $B_{even}^{\mathrm{T}} B_{even}$, $B_{even}^{\mathrm{T}} B_{odd}$, and $B_{odd}^{\mathrm{T}} B_{odd}$ represent convolutions, and therefore multiplying any of them by a vector can be done in $\Theta(\tau \log \tau)$ time using the FFT algorithm (see [32] for details). Using this technique, it is never necessary to represent Q as a dense matrix.

All the primary memory the algorithm needs is $\Theta(\tau)$ memory to store a compact representation of Q as three $(2\tau + 2)$-vectors, and the space to store a section of the vectors x, r, and p, which is guaranteed to exist since the partitioned matrix-vector multiplication algorithm returns its output in primary memory. □

In practice a naive $\Theta(\tau^2)$ matrix-vector multiplication algorithm may be a good choice for lines 7 and 11.

Two small and independent changes in the implementation details can significantly improve the performance of the algorithm. The first is a modification to the construction of Q.

Theorem 10.3 *Given a $(\tau, M/\tau)$-partitioned matrix-vector multiplication algorithm $\mathcal{A}$ for an n-by-n matrix A, a computer with $O(M)$ words of primary memory, such that $M = \Omega(\tau^2)$ can perform T iterations of algorithm* KRYLOVBASISCONJUGATEGRADIENT *using at most*

$$O(Tn + T\tau \log \tau) + (2T/\tau)\left(\text{work}(\mathcal{A}, n, \tau) + \text{work}(\mathcal{A}, n, \tau)\right)$$

work and

$$(6T/\tau)n + (2T/\tau)\left(\text{io}(\mathcal{A}, n, \tau) + \text{io}(\mathcal{A}, n, \tau)\right)$$

I/O's.

Proof: Although the proof of Theorem 10.2 calls for using the partitioned-matrix multiplication algorithm to multiply two vectors by A 2τ times, it is possible to multiply the two vectors r and p by A only τ times. Instead of multiplying say r^{T} by $A^m r$ for $m = 0, \ldots, 2\tau$, we multiply both r^{T} by $A^m r$ for $m = 0, \ldots, \tau$, and $(A^\tau r)^{\mathrm{T}}$ by $A^m r$ for $m = 1, \ldots, \tau$. This approach requires, however, that we store all the iterates $r_{P_i}, p_{P_i}, (Ar)_{P_i}, (Ap)_{P_i}, \ldots, (A^\tau r)_{P_i}, (A^\tau p)_{P_i}$ in primary memory. □

A second change reduces the amount of I/O required by reducing the number of calls to the matrix-multiplication algorithm, at the expense of multiplying by A more times in each invocation.

Theorem 10.4 *Given a (τ, M)-covered matrix-vector multiplication algorithm $\mathcal{A}$ for an n-by-n matrix A, a computer with $O(M)$ words of primary memory, such that $M = \Omega(\tau)$ can perform T iterations of algorithm* KrylovBasisConjugateGradient *using at most*

$$O(Tn + T\tau \log \tau) + (2T/\tau)\text{work}(\mathcal{A}, n, 3\tau)$$

work and

$$(6T/\tau)n + (2T/\tau)\text{io}(\mathcal{A}, n, 3\tau))$$

I/O's.

Proof: When the algorithm is used to perform $T > \tau$ iterations of conjugate gradient, it is called repeatedly, so that line 1 of one call is executed immediately after lines 14, 15, and 16 of the previous call. Both iterate over the sets in the partition generated by $\mathcal{A}$, and therefore it is possible to fuse the two loops so that the matrix multiplication algorithm is called only once on each set. Each call to the matrix multiplication algorithm multiplies by A 3τ times: the first τ to generate the sections of r, p, and x (corresponding to lines 14–16) and the last 2τ to construct Q for the next iteration. This modification requires that the matrix-multiplication algorithm be able to use linear combinations of previous iterates to generate the next one. In other words, we need a covered matrix-multiplication algorithm, or at least a partitioned algorithm that allows us to use polynomials in A rather than powers of A, where the polynomials are known in advance. □

These two modification to the algorithm can be combined, yielding the following theorem.

Theorem 10.5 *Given a $(\tau, M/\tau)$-covered matrix-vector multiplication algorithm for an n-by-n matrix A, A computer with $O(M)$ words of primary memory, such that $M = \Omega(\tau^2)$ can perform T iterations of algorithm* KrylovBasisConjugateGradient *using at most*

$$O(Tn + T\tau \log \tau) + (2T/\tau)\text{work}(\mathcal{A}, n, 2\tau)$$

work and

$$(6T/\tau)n + (2T/\tau)\text{io}(\mathcal{A}, n, 2\tau)$$

I/O's. □

10.4 Numerical Stability

The KrylovBasisConjugateGradient algorithm, like Chronopoulos and Gear's algorithms, suffers from numerical instabilities when τ is large. The reason for the instability is that as τ grows, the vector $A^\tau x$ tends to an eigenvector of A associated with its largest eigenvalue, which causes both B and Q to be ill-conditioned. This section shows that this source of instability does not pose a significant problem when τ is small, and it proposes a technique to stabilize the algorithm.

Figures 10.1 and 10.2 show the relative distances between the 2-norms of the residuals $r = b - Ax$ in the ConjugateGradient and KrylovBasisConjugateGradient algorithms. The distances are relative to the norm of the residual in ConjugateGradient. The experiments used values of τ between 5 and 20 (except the last phase which may be shorter), with matrices A which represent a one- or two-dimensional Poisson problem and random right-hand sides b. The figures show that the error in the residual of the KrylovBasisConjugateGradient algorithm grows exponentially as a function of the phase length τ. The error is quite small however up to about $\tau = 9$. This behavior persists until about $\tau = 10$ or 11. Beyond that point, the output KrylovBasisConjugateGradient algorithm behaves erratically. The error is not strongly influenced by the order of the matrix.

The value $\tau = 5$ used by Chronopoulos and Gear therefore seems quite conservative, and values up to $\tau = 9$ are certainly usable for some problems. Figure 10.3 compares the 2-norms of the residual in the conventional conjugate gradient and the residual in the Krylov-basis conjugate gradient with $\tau = 9$ for a large number of iterations, on a one-dimensional Poisson problem of size 128. The results agree well up to the last few iterations, which indicates that the Krylov-basis method is stable. When the residual gets very small towards the end of the iterative process, however, it might be advisable to switch to a smaller value of τ or to the conventional conjugate-gradient algorithm.

In fact, the algorithm can be stabilized while using very high values of τ up to $\tau = O(\sqrt{M})$. The idea is to replace B in KrylovBasisConjugateGradient with a more stable basis spanning the same Krylov subspace. We use

$$B = [\phi_0(A)r_0, \psi_0(A)p_0, \phi_1(A)r_0, \psi_1(A)p_0, \ldots, \phi_\tau(A)r_0, \psi_\tau(A)p_0] ,$$

where $\{\phi_t\}_{t=0}^{\tau}$ and $\{\psi_t\}_{t=0}^{\tau}$ are families of polynomials, such that ϕ_t and ψ_t are degree t polynomials.

In theory, such a method is susceptible to a breakdown, although We have not experienced any breakdowns during our numerical experiments. A polynomial ϕ_t (or ψ_t) with a root coinciding with an eigenvalue of A causes a problem if r_0 (or p_0) is an eigenvector associated with that eigenvalue, since in that case B has some zero columns and it does not span the Krylov subspace.

Before we suggest how to chose the polynomials $\{\phi_t\}_{t=0}^{\tau}$ and $\{\psi_t\}_{t=0}^{\tau}$, we discuss the impact on the implementation and performance of the algorithm. In particular, the construction of B and $Q = B^{\mathrm{T}}B$ must be modified (i.e., the details of Line 1 in the pseudocode). The cost of computing B depends on how the polynomial families are specified. Defining the polynomials in terms of their coefficients does not help stabilize the algorithm, because then $A^\tau r_0$ and $A^\tau p_0$

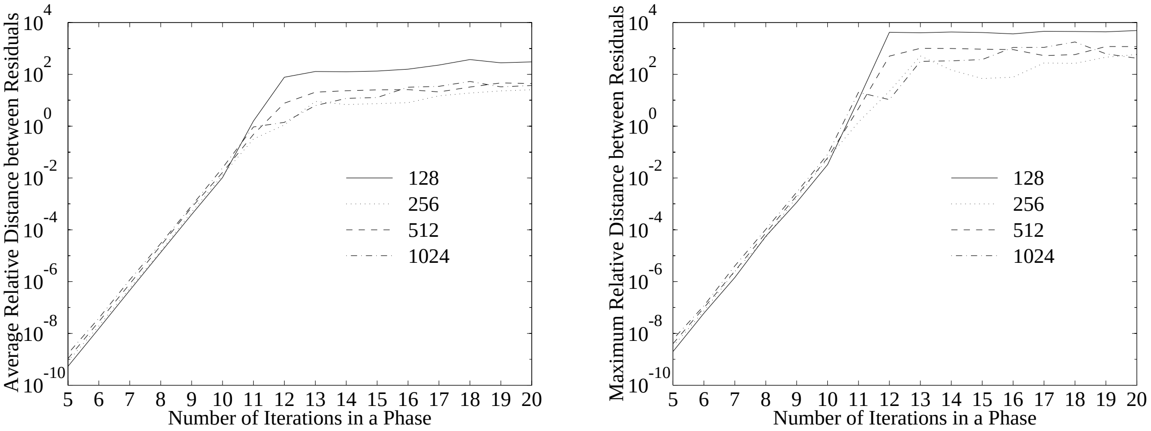


Figure 10.1 The distance, after 64 iterations, between the norms of the residuals in the conventional and the Krylov-basis conjugate-gradient algorithms, relative to the norm of the residual in the conventional algorithm. The graphs show the average distance (left) and the maximum distance (right) in groups of 100 experiments, with a one-dimensional Poisson matrix of several orders (represented by different line styles) and random right-hand sides.

must be computed. We therefore assume that the polynomials are given using a recurrence

$$\phi_t(A) = \alpha_{t,-1} A \phi_i(A) + \sum_{i=0}^{t-1} \alpha_{t,i} \phi_i(A) \; ,$$

where the $\alpha_{t,i}$'s are given. Computing B is straightforward using a covered matrix-vector multiplication algorithm using the recurrences on the columns of B. The computation requires $\Theta(\tau^2 n)$ work. If the recurrences are **short**, which means that all but a constant number of the $\alpha_{t,i}$'s are nonzero for each t, the work required for computing B is only $\Theta(\tau n)$ (two nonzero terms is a common case in families of orthogonal polynomials). The same amount of work, $\Theta(\tau n)$, is required for computing B in the case of **coupled** short recurrences such as

$$\phi_t = \alpha_t A \phi_{t-1} + \beta_t \psi_{t-1} \; ,$$

which is the form found in the conventional conjugate gradient algorithm. Computing Q by multiplying B^{T} by B requires $\Theta(\tau^2 n)$ work. In Section 10.3 we reduced this cost for the case $\phi_i(A) = \psi_i(A) = A^i$ using the facts that $A^{\mathrm{T}} = A$ and that $\phi_i \phi_j$ is a function of $i + j$.

One particular choice of polynomials that stabilize the algorithm well in numerical experiments is a family of polynomials in which the sections of $\phi_0(A)r_0, \phi_1(A)r_0, \ldots, \phi_\tau(A)r_0$ corresponding to the set P_1 in the cover are orthogonal. To compute the polynomials, we load the section of r_0 corresponding to C_1 into primary memory. We then repeatedly multiply the last iterate by A and orthogonalize it with respect to all the previous iterates. The coefficients $\alpha_{i,j}$ for $i = 2, \ldots, \tau$ and $j = -1, 0, 1, \ldots, i-1$ are saved and later used on all the other sets in the cover. In numerical experiments on one-dimensional Poisson problems with random right-hand sides, stability was excellent for all values of τ up to the size of P_1, which can be as large as $(1/2)\sqrt{M}$. This result is not surprising, since an orthogonal submatrix consisting of $|P_1|$ rows of a matrix and all its columns ensures that the columns of the original matrix are

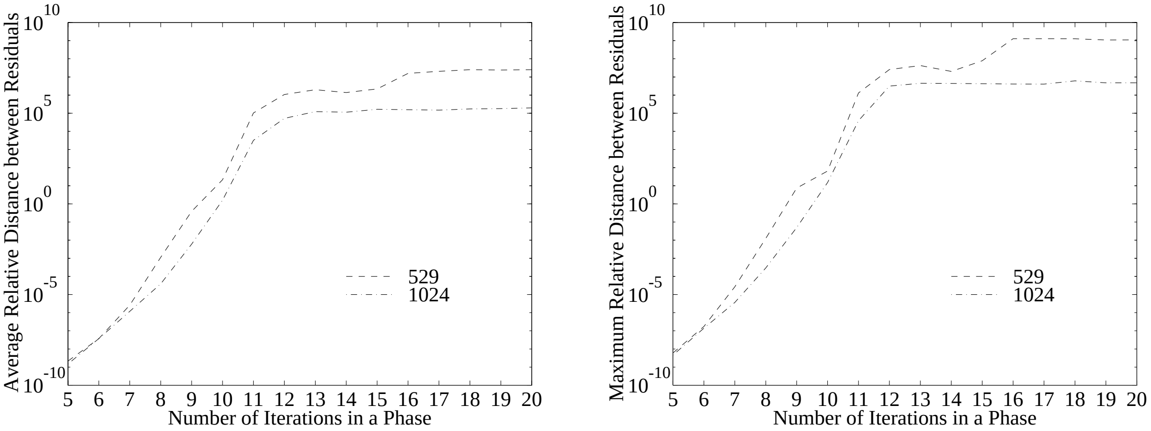


Figure 10.2 Relative distances between the norms of the residuals in the conventional and krylov-basis conjugate-gradient algorithms. The experiments reported in the graphs are identical to those reported in Figure 10.1, except that the matrix is a two-dimensional Poisson matrix.

independent. In our case, the even columns of B are independent and the odd columns of B are independent. Excluding the matrix-vector multiplications, the work required to compute both B and Q using this scheme is $\Theta(\tau^2 n)$, which is a factor of τ more than the work required by the conventional conjugate gradient algorithm. The amount of work can be reduced if we orthogonalize only once every m iterations.

10.5 Implementation

We have implemented the out-of-core conjugate gradient algorithms described in this section. The implementation was designed and coded by the Tzu-Yi Chen and the author. This section describes the implementations, and the next section shows that the performance of the Krylov-basis algorithm outperforms conventional conjugate gradient algorithms when executed out-of-core.

We have prototyped all the algorithms in MATLAB [108], an interpreted, interactive environment for numerical computations. The prototypes served to check the correctness of the algorithms (we have found one bug in a published algorithm), to study their numerical properties, and to verify that subsequent implementations are correct. A similar strategy was used in the implementation described in Chapter 9.

The algorithms were then implemented in C, so that their performance could be assessed. The main design objectives of the C implementation were high performance, and modularity. Our modular design separates the matrix-vector multiplication code from the out-of-core Krylov-subspace algorithm.

We have implemented several algorithms: the conventional conjugate gradient, the clustered single-step conjugate gradient, and several versions of the Krylov-basis conjugate gradient algorithm, including all the versions described in Section 10.3. We have also implemented a covered matrix-vector multiplication algorithm for tridiagonal matrices. The implementation

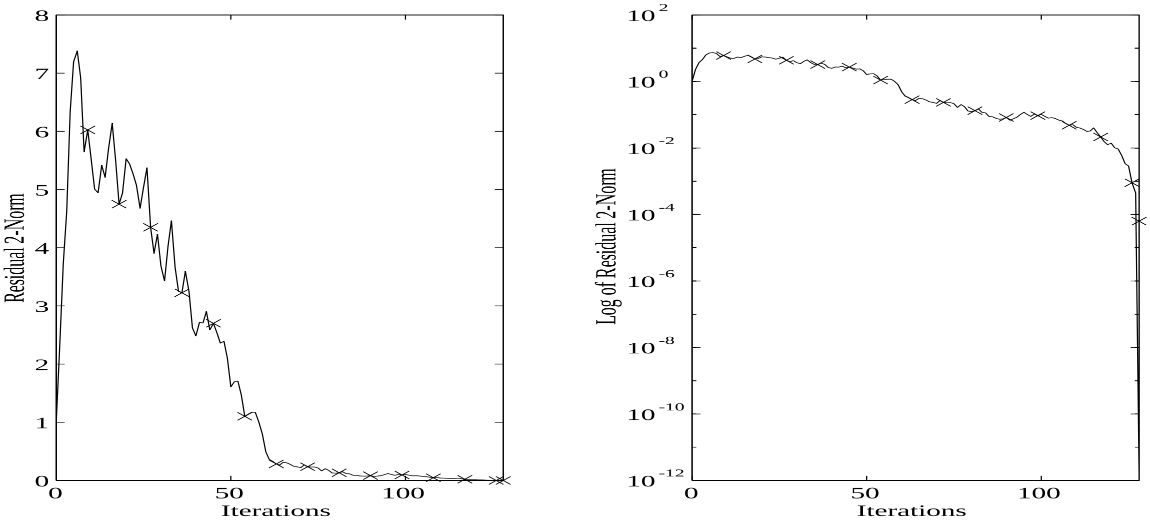


Figure 10.3 Convergence of conventional conjugate gradient (sold line) and the Krylov-basis conjugate gradient with $\tau = 9$ (crosses) on a one-dimensional Poisson problem of size 128. The 2-norm of the residuals is plotted in a linear scale on the left , and in a logarithmic scale on the right. The results agree well up to the last few iterations.

relies on the automatic paging system to perform the I/O. As we shall see in the next section, the scheme works well, it is simpler than using explicit I/O system calls, it eliminates copying to and from I/O buffers, and it allows the code to work with hierarchical memory systems in which no explicit I/O is possible, such as caches. The disadvantage of this scheme is that the algorithm blocks when I/O is needed, whereas using asynchronous I/O system calls would enable the implementation to overlap computation and I/O.

10.6 Performance

Table 10.1 shows that, On a workstation, our Krylov-basis algorithm with $\tau = 9$ outperforms the original conjugate-gradient algorithm by up to a factor of 3.5, when data structures do not fit within main memory. Our algorithm outperforms the ClusteredDotProdsCG algorithm by a factor of up to 3. When the data fits within main memory, our algorithm is about 2.6 times slower than the other two, because it performs more work. But by performing more work and less I/O, our algorithm is faster on large problems. The Krylov-basis algorithm is significantly faster then the other implementations even with $\tau = 5$.

The experiments were performed on a Sun SPARCstation 1+ workstation with a Sun 4/60 CPU, 8Mbytes of main memory, and a local disk for paging. The programs were compiled by the GNU C compiler with a –02 optimization option. On this machine, computing an inner product of two double-precision n-vectors in core on this machine takes about $2.2n\mu$s, whereas computing the inner product out of core takes about $61.1n\mu$s, reflecting an average bandwidth of about 15ms per 4Kbytes page.

	ConjugateGradient	ClusteredDotProdsCG	KrylovBasisCG		
n			$\tau = 5$	$\tau = 9$	$\tau = 15$
10K	23.2	22.8	60.7	58.9	58.7
50K	22.8	23.0	59.2	56.6	56.1
100K	25.9	197.5	77.1	67.0	61.0
250K	154.1	214.3	86.5	70.4	63.2
500K	245.8	211.4	85.6	70.3	62.8

Table 10.1 The running times of the three implementations of the conjugate-gradient algorithm described in this chapter. The table shows the running time per element per iteration in microseconds on a Sun SPARCstation 1+ workstation. The numbers represent averages of three runs, each consisting of 10 phases of τ iterations. The running time per iteration of ConjugateGradient and ClusteredDotProdsCG is independent of τ, and experiments used $\tau = 5$. The matrix A is an n-by-n tridiagonal matrix which was partitioned into sets of at most 2000 elements. The Krylov-basis algorithm used 4τ matrix-vector multiplication per phase.

10.7 Discussion

We have described the theory, implementation, and performance of new multistep, out-of-core Krylov-subspace methods. Our algorithms are based on explicit construction of a basis B of the Krylov subspace, on discarding and recomputing the same basis to save I/O's, and on transforming the algorithm into the Krylov subspace. The algorithms proposed by Chronopoulos and Gear, in comparison, do not discard and recompute the basis, so they perform more I/O's, and they do not completely transform the algorithm into the Krylov basis. As a consequence, their algorithms are less efficient than ours, and they are more difficult to derive. But they have the advantage that they multiply vectors by A much less. Finding a way to do the same in our framework could lead to significant speedups of our algorithms.

One technique that should increase the stable range of τ is to compute B in higher precision. The convergence rate of $A^\tau x$ to an eigenvector of A is linear, and so, roughly speaking, we lose a constant number of accurate bits in every iteration. Doubling the number of bits in the significand of the floating-point numbers used to represent $A^\tau x$ should enable us to more than double τ while retaining the same number of accurate bits. Since we never store B to secondary memory, but rather recompute it, there is no need to do any I/O in higher precision. To summarize, we should be able to trade an increase by about a factor of 2 in work per iteration in exchange for a similar decrease in I/O.

The local densification technique, described in Chapter 7, can also be used to improve the locality in out-of-core Krylov subspace methods. The local densification is implemented via polynomial preconditioning. The idea is to use a preconditioning step in every iteration, in which the residual r is multiplied by a given polynomial in A. Since the polynomial is known, preconditioning can be performed by a covered matrix-vector multiplication algorithm for A. In many cases preconditioning reduces the number of iterations, and since the amount of I/O is about the same as that of a single-step algorithm, the total number of I/O's decreases. The amount of work however often stays the same or increases, because the total number of matrix-vector multiplications often grows.

Several families of polynomials have been proposed for preconditioning. One traditional choice has been Chebychev polynomials. Constructing them requires good approximations to the extreme eigenvalues of A, but even exact eigenvalues can result in a dramatic increase in the amount of work required. The effect is problem dependent. Fischer and Freund [45] propose a different family of polynomials, whose construction requires additional spectral information, but they show how to gather in the information during the execution of the conjugate gradient algorithm. With polynomials of degree 10-15, they report that number of matrix multiplications remains about the same as the nonpreconditioned algorithm. They have not measured the performance of their algorithm.

One disadvantage of polynomial preconditioning, compared to our Krylov-basis algorithms, is that the effect on convergence is problem dependent and may lead to poor performance, whereas in our method there is no effect on convergence.

Chapter 11

Preconditioning with a Decoupled Rowwise Ordering on the CM-5

11.1 Introduction

This chapter describes the implementation and performance of a preconditioned conjugate gradient solver for discretized Poisson problems in 2 dimensions, which I have implemented on the Connection Machine CM-5 parallel supercomputer [110]. The preconditioner is an SSOR preconditioner with a novel ordering scheme for the variables, which I call **decoupled rowwise ordering**.

The solver is efficient. The implementation of a major component in the solver, the matrix-vector multiplication subroutine, is twice as fast as a naive implementation in CM Fortran, and one and a half times as fast as the implementation in the CM Scientific Subroutine Library. Preconditioning with the decoupled rowwise ordering reduces the solution time of important problems by a factor of 1.5 to 2.5, compared with no preconditioning and with red-black preconditioning. The software, which is described in Sections 11.3 and 11.4, is used in an ocean model being developed by John Marshall and his colleagues at the department of Earth, Atmospheric and Planetary Sciences at MIT.

Decoupled rowwise ordering, which is illustrated in Figure 11.1 and described in detail in Section 11.2, is tailored for block layouts of rectangular grids where every processor in a parallel computer is assigned one rectangular subgrid. The grid points on the boundary of every block are ordered using a red-black coloring to decouple the blocks, and interior grid points in every block are ordered row by row. This ordering scheme admits an efficient parallel implementation of the preconditioner on the CM-5 and other distributed memory parallel computers. The preconditioner runs in the same amount of time per iteration as the more conventional red-black preconditioner, but on many important problems it reduces the number of iterations more than the red-black preconditioner.

Decoupled rowwise ordering locally densifies the dependencies in the red-black ordering to accelerate convergence. The local densification technique, described in Chapter 7, is used here to increase the utilization of data paths within individual processors without increasing

[1] A preliminary version of this chapter was presented in the *7th SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, California, February 1995.

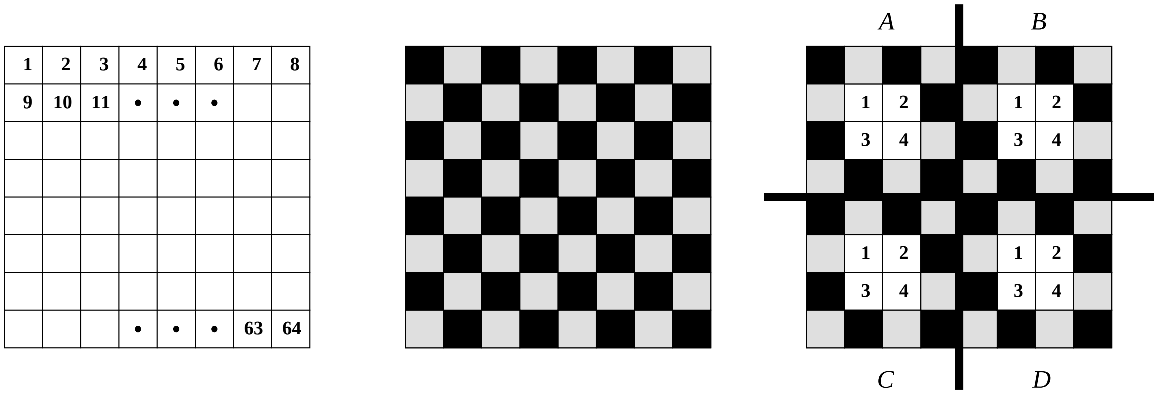


Figure 11.1 Orderings for an 8-by-8 grid. The rowwise ordering is depicted on the left, a red-black ordering in the center, and a decoupled rowwise with four blocks marked A, B, C, and D, on the right. The red points, which are lightly shaded in the figure, are ordered before the black points which are ordered before all the interior points. The interior points in each block, are ordered in rowwise order.

the load on interprocessor communication channels.

11.2 Ordering Schemes for Parallel Preconditioners

Several popular preconditioners require the solution of two sparse triangular systems, usually by substitution. Such preconditioners include Symmetric Successive Over Relaxation (SSOR) [120], Incomplete Cholesky preconditioners [84], and Modified Incomplete Cholesky preconditioners [56]. We focus on preconditioning linear systems of equations $Ax = b$, where A is derived from a 5-point stencil in two dimensions using a preconditioner of the form $M = \tilde{L}\tilde{D}\tilde{L}^{\mathrm{T}}$, where $\tilde{L}$ is a sparse lower triangular matrix, whose sparsity pattern is identical to the sparsity pattern in the lower triangular part of A, and where $\tilde{D}$ is a diagonal matrix. That is, no fill-in is allowed in incomplete factorizations. Most of our experiments use SSOR preconditioning in which

$$M = \frac{1}{2-\omega}\left(\frac{1}{\omega}D + L\right)\left(\frac{1}{\omega}D\right)^{-1}\left(\frac{1}{\omega}D + L\right)^{\mathrm{T}} ,$$

where D and L are the diagonal and strictly lower triangular parts of A. We determine the optimal acceleration parameter ω experimentally. We also compare SSOR preconditioning with preconditioning based on Incomplete and Modified Incomplete Cholesky factorizations.

The ordering of the variables in the domain affects both the convergence rate of the preconditioned problem and the implementation of the sparse triangular solvers. Duff and Meurant [40] show that the ordering of variables in incomplete Cholesky factorizations influences the spectrum of the preconditioned matrix and hence the convergence rate of iterative solvers. The ordering also influences the implementation of the sparse triangular solvers. When the sparsity pattern of the triangular factor L is identical to that of the 5-point stencil equations, the variable that represents point i in the domain depends on the solution of the triangular system at neighboring points that come before i in the ordering, or on neighbors that come after i in the substitution for L^{T}.

Discretizing a partial differential equation on a regular rectangular grid allows the solver to use a simple data structure to represent the solution. A d-dimensional rectangular mesh is mapped to a d-dimensional array of floating-point numbers in the natural way.

The most popular ordering scheme on sequential computers is the **rowwise**, or **natural**, ordering. In this ordering scheme the variables are ordered row-by-row, from left to right. The triangular solver for the rowwise ordering contains a significant amount of parallelism — one can solve a system on an n-by-n grid using n processors in $2n$ parallel steps, by proceeding along so-called “wavefronts” [118]. On the CM-5 and similar machines, however, it is hard to exploit this parallelism. In order to balance the computational load, a triangular solver for the rowwise ordering requires a data layout with a long and thin block on every processor, with the Fortran array rotated 45 degrees with respect to the physical domain. Long and thin blocks result in inefficient communication in the rest of the finite-differences computation.

Red-black ordering has been proposed as a way of increasing the parallelism in the triangular solver. The variables in the domain are colored red and black in a checkerboard pattern, so that neighboring points always have different colors. All the red points come before all the black points. The resulting triangular solver has a tremendous amount of parallelism — the system can be solved in 2 parallel steps using $n^2/2$ processors. The implementation in data parallel Fortran is simple. Unfortunately, the convergence rate obtained with this ordering is often worse than the convergence rate obtained with the natural ordering. The slower convergence rate of red-black incomplete factorization preconditioners was observed experimentally by Ashcraft and Grimes [4] and Duff and Meurant [40]. Kuo and Chan [70] proved that for problems in square domains with Dirichlet boundary conditions naturally ordered preconditioners converge asymptotically faster than red-black ordered preconditioners. Their results apply to conjugate gradient with SSOR, incomplete Cholesky, and modified incomplete Cholesky preconditioners, as well as to SSOR without conjugate gradient acceleration.

I propose a new ordering scheme, called decoupled rowwise ordering. Figure 11.1 depicts the decoupled rowwise ordering along with the rowwise and red-black orderings for an 8-by-8 grid. The ordering scheme is based on a decomposition of the grid into rectangular subgrids. The boundary points of every block are colored red and black in a checkerboard pattern. The red boundary points are placed first in the ordering, followed by the black boundary points, followed by the interior points. The interior points in every block are ordered in a rowwise order. The triangular solver solves first for the red points which depend on no other points, then for the black points which depend on their red neighbors, and then for the interior points. Solving for the boundary points, which can be done in 2 parallel steps, completely decouples the blocks, and each processor can proceed to solve for all the interior points in its block. The decomposition of the grid used by my software is simply the block layout of the array representing the grid in the parallel computer.

On test problems, the decoupled rowwise SSOR preconditioner accelerates the convergence more than the red-black SSOR preconditioner. Figure 11.2 shows the convergence rate of conjugate gradient without preconditioning, with red-black preconditioning, and with decoupled rowwise preconditioning, for a Poisson equation with Neumann boundary conditions and a random right-hand side. The preconditioners use an optimal over relaxation parameter ω. The optimal value of ω was determined experimentally for each problem and preconditioner. The number of iterations required to reduce the norm of the residual by a given factor is proportional to the length of the the grid side for the nonpreconditioned solver and for the red-black

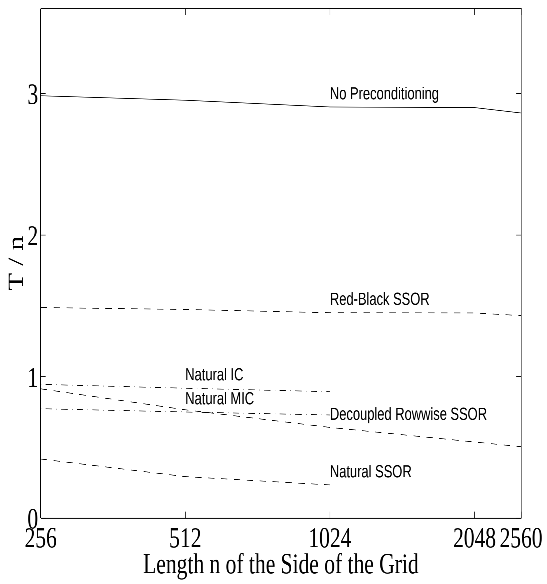


Figure 11.2 Convergence rates of conjugate gradient without preconditioning, with SSOR preconditioning using natural, red-black, and , decoupled rowwise orderings, as well as with incomplete Cholesky (IC) and modified incomplete Cholesky (MIC) using the natural ordering. The incomplete factorization preconditioner are unperturbed. The number of iterations T it took to reduce the 2-norm of the residual by a factor of 10^{-5} divided by the grid length n is plotted as a function of the grid length. The problem is a Poisson equation with Neumann boundary conditions in a square, with a random right-hand side. The decoupled rowwise preconditioner used 128 rectangular blocks.

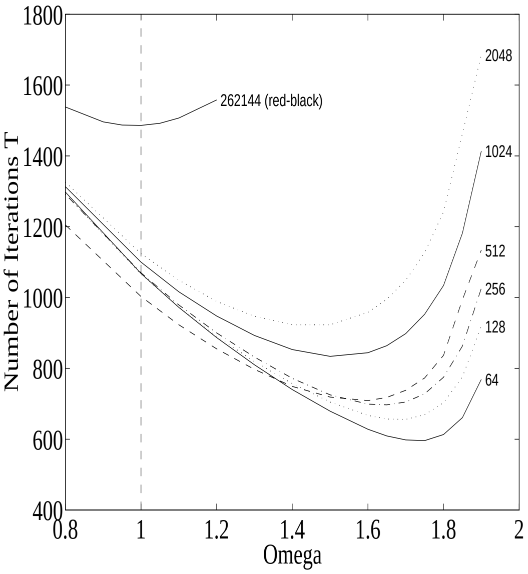
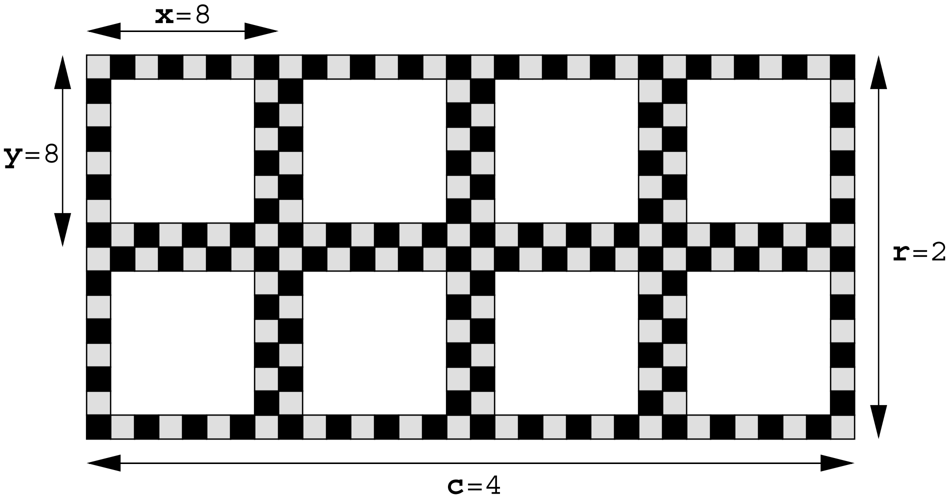


Figure 11.3 Convergence rates of conjugate gradient with decoupled rowwise preconditioning. The number of iterations T it took to reduce the 2-norm of the residual by a factor of 10^{-5} is plotted as a function of the over relaxation parameter ω for a number of block sizes. The number next to each graph is the number of blocks to which the square domain is decomposed. The problem is a Poisson equation with Neumann boundary conditions in a square discretized on a 1024-by-1024 grid, with a random right-hand side.

preconditioning. For the decoupled rowwise preconditioner, however, the number of iterations depends sublinearly on the grid-side length, and it is always smaller than the number of iterations required by the red-black preconditioner. The decoupled rowwise preconditioner reduces the number of iterations by a factor of 3.4–5.4 over the number required by the nonpreconditioned solver, whereas the red-black preconditioner reduces the number of iterations by a factor of roughly 2. The preconditioners behave in a similar manner on other test problems, such as problems in nonsquare domains with holes and problems with Dirichlet boundary conditions.

Figure 11.3 shows the convergence rate of the solver with the decoupled rowwise preconditioner as a function of the over relaxation parameter ω for a range of block sizes. For a given problem, orderings with few large blocks generally lead to faster convergence than orderings with many small blocks. Although this rule does not always hold, in the experiments it always holds for the optimal value of ω. Figure 11.3 also demonstrate that the optimal value of ω decreases as the number of blocks increases. In the limit, when the length of the side of each block is 1 or 2, decoupled rowwise degenerates into a red black ordering, in which the optimal



```
do i=2,x-1
  do j=2,y-1
    forall (n=0:c-1,m=0:r-1)
      z(n*x+i,m*y+j) = ...
```

Figure 11.4 The skeleton of an implementation of the decoupled-rowwise ordering in CM-Fortran or High Performance Fortran. The algorithm can be expressed in these languages, but the code the compiler generated was unacceptably slow.

value of ω is very close to 1 in the experiments. Kuo and Chan [70] proved, for certain model problems, that the optimal value of ω in the red-black SSOR method is indeed 1.

11.3 Implementation

My CM-5 implementation[2] of the conjugate-gradient solver and of the preconditioners combines high-level CM Fortran code with low-level assembly language code to simultaneously achieve high performance and reasonable code complexity. The software includes a conjugate-gradient solver, a matrix-vector multiplier, and preconditioners for 5-point stencils in 2 dimensions. The stencils use one variable coefficient and four constant coefficients. The sparse triangular solver can be used for both SSOR preconditioning and for preconditioning that is based on incomplete factorizations with no fill-in.

I implemented the solver on the CM-5 using a combination of CM Fortran [109] and CDPEAC [113, 114], the assembly language of the vector units with which the CM-5's nodes are equipped. The overall solver is written in Fortran. The nodal part of the sparse matrix-vector multiplier and the nodal parts of the two preconditioners are implemented in CDPEAC. The communication in both the matrix-vector multiplier and the preconditioners is done in Fortran, using shift operations on small "ghost" arrays into which the boundary grid points on every block

[2] The software is available by anonymous FTP from `ftp://theory.lcs.mit.edu/pub/sivan/rowwise-decoupled.tar.Z`, or through the world-wide-web in `http://theory.lcs.mit.edu/˜sivan/papers.html`.

	Busy Timer			Elapsed Timer		
$\sqrt{n}$	CM-Fortran	CMSSL	CDPEAC	CM-Fortran	CMSSL	CDPEAC
256	34.9	32.4	22.5	41.0	45.1	28.0
512	24.7	20.8	13.1	28.7	23.6	15.0
1024	21.4	17.2	10.3	22.8	18.5	10.7
2048	19.9	15.8	10.0	20.2	16.1	10.1

Table 11.1 The running time of three matrix-vector multiplier subroutines: a subroutine written in CM Fortran using `shift` operations, a subroutine that uses the CM Scientific Subroutine Library, and my implementation in CDPEAC and CM Fortran. The table reports the running times per point per iteration in nanoseconds for various $\sqrt{n}$-by-$\sqrt{n}$ grids. The running time reported are averages of three executions of 100 iterations each on a time-shared 32-node CM-5.

are copied. The copying from the grid to the ghost arrays is done in CDPEAC, however, since the CM Fortran compiler does not vectorize these operations well. The CDPEAC subroutine that computes the nodal part of the matrix-vector multiplier also computes the inner product of the vector before and after the multiplication, which is required in the conjugate gradient method. The integration of the inner product into the matrix-vector multiplier eliminates the need to load the two vectors into registers again. The red-black preconditioner uses strided load and store instructions to select red or black points. The CDPEAC subroutines consist of more than 1500 lines of code.

I wrote the solver partially in assembly language for three reasons. First, I wanted to obtain the highest quality solver I could, and I could not achieve this goal with either Fortran or CMSSL [112], the CM-5 Scientific Subroutine Library, as demonstrated in Section 11.4. Second, since the ordering scheme requires using DO loops within each block, or on each node, there is no way to efficiently implement the preconditioner in Fortran. The only way to express such a computation in CM Fortran or High Performance Fortran is using a `forall` loop nested within `do` loops, as shown in Figure 11.4. On the CM-5, this code does not compile into a tight loop on the nodes, but rather into a loop containing a broadcast in every iteration, whose performance is about two orders of magnitude worse than my CDPEAC implementation. Third, since the usefulness of a preconditioner depends on the ratio between the preconditioning time and the conjugate gradient iteration time, the only way to evaluate the preconditioner is using the best possible implementation for both the preconditioner and the conjugate gradient solver.

11.4 Performance

To put the performance of my solver in perspective, Table 11.1 compares the performance of my sparse matrix-vector multiplier subroutine with the performance of a subroutine written in CM Fortran and a subroutine from the Connection Machine Scientific Subroutine Library (CMSSL)[3] with the same functionality. (CMSSL only contains variable-coefficients stencil matrix-vector multiplier, which is the subroutine I use in the comparison.) The table shows that on large grids my multiplier takes half the time the CM Fortran multiplier takes, and two

[3] I have used CM Fortran version 2.1.1-2 Final and CMSSL version 3.1 Final.

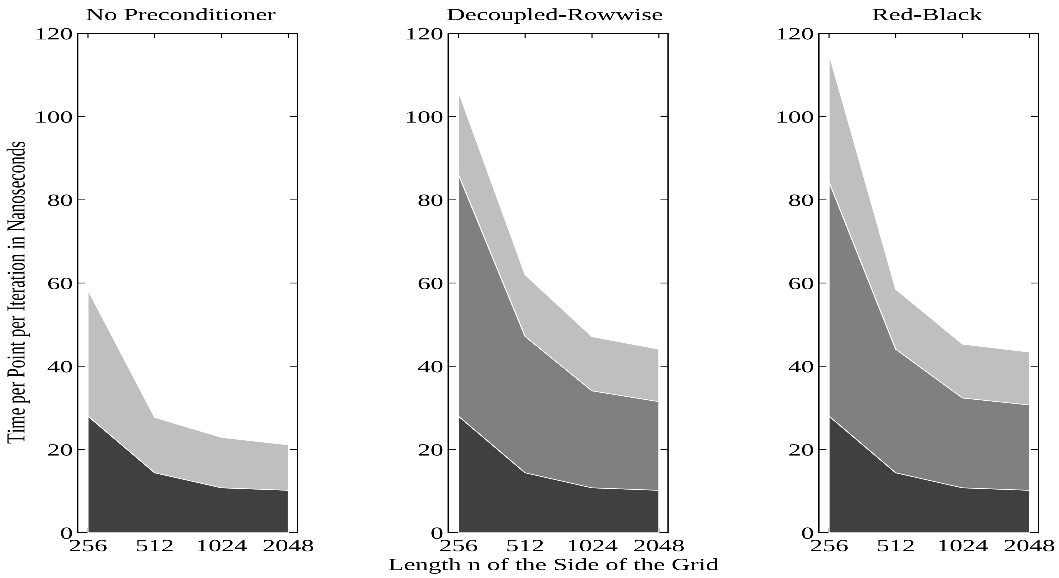


Figure 11.5 The running time of the solver per point per iteration as a function of the problem size, on a 32-node CM-5. The bottom, darkest section in each graph represents the matrix-vector multiplication time, the middle section represents the preconditioning time (if a preconditioner is used), and the top and lightest section represents the rest of the vector operations.

thirds the time the CMSSL multiplier takes.

Figure 11.5 shows the running time of the solver per point per iteration, measured on a 32-node CM-5. The graphs show the matrix-vector multiplication time, the preconditioning time (for both a red-black and a decoupled rowwise preconditioners), and the overall time, for problems sizes ranging from 256-by-256 up to 2048-by-2048. Without preconditioning the iteration time is divided roughly evenly between the matrix-vector multiplication and the rest of the vector operations in the solver, and both preconditioners roughly double the running time of each iteration.

11.5 Related Work

This section describes research related to the results reported in this chapter. In particular, we describe preconditioners whose structure is similar to that of the decoupled rowwise preconditioner, and preconditioners used in numerical ocean models.

Duff and Meurant [40] describe numerical experiments with an ordering scheme called **vdv1**. Vdv1 does not decouple the blocks in the same way that the decoupled rowwise ordering does, however, and is therefore less suitable for parallel implementations. I have recently learned that Tomacruz, Sanghavi, and Sangiovanni-Vincentelli [116] describe an ordering called **block-partitioned natural ordering**, which they use in an incomplete LU preconditioner for three-dimensional drift-diffusion device simulations. Their equations are nonlinear, and are solved using Newton's method, which requires solving a sparse asymmetric linear system in every iteration. These linear systems are solved using the preconditioned conjugate gradient squared method. The Block Partitioned Natural Ordering is similar to my decoupled rowwise ordering, except that the blocks are decoupled in a different way. Rather than color boundary points red and black, they eliminate some of the edges that cross processor boundary. They eliminate different edges in the forward and backward substitution processes. They have implemented their algorithm on the CM-5, although unlike my implementation, they do not use the CM-5's vector units, and thus their implementation is much slower. The simpler programming model allowed them to experiment with different amounts of fill-in within the blocks, which was difficult in our case.

Research groups that implement numerical ocean models often face the need to implement Poisson solvers for problems similar to the ones described in this chapter. The ocean model developed by a group of researchers at Los Alamos National Laboratory [105] solves two-dimensional Poisson problems discretized using a 9-point stencil (as opposed to a 5-point stencil used by the MIT model being developed by John Marshall, Chris Hill, and others). They chose to use sparse *explicit* preconditioners, in which M^{-1} is given explicitly, and therefore preconditioning amounts to a sparse matrix-vector multiplication. They have considered polynomial preconditioners, but have chosen to use a preconditioner M^{-1} which has exactly the same sparsity pattern as A and which minimizes the trace of $(I - M^{-1}A)(I - M^{-1}A)^{\mathrm{T}}$. They report good convergence acceleration using this preconditioner, even though the preconditioner is purely local and therefore cannot asymptotically reduce the number of iterations.

I believe that in the future, large two-dimensional Poisson problems arising from ocean models would require the use of more sophisticated preconditioners. Even with the decoupled rowwise preconditioner, large problems (e.g. 2560-by-2560) take thousands of iterations to converge. It seems to us that unless future ocean models use more efficient Poisson solvers, such as multigrid solvers or multigrid-preconditioned solvers, the fraction of the total simulation time devoted to solving elliptic equations will grow.

11.6 Conclusions

We have presented an efficient conjugate-gradient solver for 5-point difference equations in 2 dimensions, utilizing an SSOR preconditioner with a novel ordering scheme for the variables. Thanks to careful coding and the use of assembly language, the implementation is more efficient than CM Fortran and CMSSL implementations, and the preconditioner is about as efficient as a high-performance red-black preconditioner. The preconditioner accelerates the solver's convergence more than a red-black preconditioner, leading to speedups by a factor of 1.5 to 2.5 over the nonpreconditioned solver.

I believe that the form and the amount of parallelism in an algorithm must match the architecture of the machine which executes it. In our case, the parallelism in the rowwise decoupled ordering is easier to exploit on the CM-5 than the wavefront parallelism in the rowwise ordering, and the amount of parallelism is smaller than the parallelism in a red-black ordering. The smaller amount of parallelism is probably an advantage, since it seems to improve the convergence rate while still admitting an efficient parallel implementation on hundreds of processors.

It is important that programming languages, especially High Performance Fortran [59, 68], allow programmers to express algorithms in which serial execution on each node is used. This could be accomplished either via compiler optimizations of expressions involving arrays with serial axes, or by allowing the programmer to write nodal code, which is not currently possible in CM Fortran[4]. I prefer the second option since it is simpler for both the programmer and the compiler writer. Writing sequential nodal code does not necessitate the use of message passing libraries, since the communication can be done in data-parallel Fortran using ghost arrays.

[4] It is possible to write nodal subroutines in Fortran 77 on the CM-5, but they do not utilize its vector units.

Chapter 12

Conclusions

12.1 Performance Modeling and Prediction

The main contribution of the first part of the thesis is the idea that the performance of computer systems can be specified with models of entire runtime systems. Benchmaps, which are models of runtime systems, are an enabling technology for a wide range of applications.

The thesis demonstrates that accurate benchmapping of data-parallel runtime systems is possible. These benchmaps can be incorporated into software tools that predict the running time of programs automatically, accurately, and rapidly.

In some cases, the performance of compiler generated code must be modeled as well as the performance of the runtime system. PERFSIM demonstrates that predicting the performance of compiler generated object code is possible. Atapattu and Gannon use this technique too [5]. The data-parallel programming models helps here too, because the object code consists mostly of well structured loops.

Performance modeling must be approached as an experimental science. A chemist who knows all the basic rules of physics still must perform experiments to understand the behavior of complex physical systems. Even when we know exactly how a computer operates and how an algorithm is implemented, complex interactions in the system make it virtually difficult to predict a priori the behavior of the system.

An important goal of experimentation is to substantiate or refute assumptions. Most models and most modeling techniques depend on certain assumptions, such as the identity of performance determinants and the distribution of noise. The assumptions should be explicitly stated, and an effort to validate or refute them should be undertaken.

When asked to provide performance models of their libraries, writers of libraries and runtime systems often express two concerns. First, if they provide performance models, they have to support and maintain them. Second, users have the right to expect from performance models the same level of reliability the library provides. The models should either predict performance within some known margin of error, or fail and inform the user about its failure.

Computer vendors are concerned about fairness. If they are to provide performance models which are used to guide acquisition decisions, they want other vendors to provide models which are accurate. The folklore and secrecy surrounding performance models suggest that it may be hard to achieve such level of confidence among vendors. The difficulties associated with robust test suites only complicate the issue further.

These concerns lead me to believe that the most plausible avenue for introducing performance models is through a third parties, which will model libraries and runtime systems, and provide the models to users. This model is similar to the introduction of benchmarks, such as the LINPACK benchmark. If the models prove popular, we expect vendors to get involved in modeling their systems, in much the same way that vendors implement the programs in the NAS benchmark suite.

12.2 Locality in Numerical Algorithms

The main contribution of the second part of the thesis is a collection of out-of-core iterative numerical algorithms which are based on the idea of blockers. The algorithms we present are novel implementations of well-known iterative methods. No efficient out-of-core implementation of any of these methods has been previously published. My implementations, which use blockers, perform more arithmetic operations than naive implementations, but they perform much less I/O's, and are therefore faster.

A computer can be **unbalanced** with respect to an algorithm. The term unbalanced, coined by Kung [69], roughly means that when the algorithm executes, some resources are saturated and some are idle most of the time. For example, in many out-of-core algorithms, the I/O channel is saturated whereas the CPU is idle much of the time. The Jacobi and red-black relaxation algorithms provide another example, in which the CPU is saturated, but the algorithms are using fewer dependencies in loops than can be handled by the computer without putting additional load on the CPU (that is, some internal data paths are not fully utilized).

My main high-level conclusion from the research reported in this thesis is that when a computer is unbalanced and cannot be balanced, the algorithm can almost always be changed to take advantage of underutilized resources. Our out-of-core algorithms for example, put additional load on an underutilized CPU, in exchange for reducing the load on the saturated I/O channel. The net effect achieved is a reduction in the running time of the algorithm. The reduction is not as large as one could get from increasing the bandwidth of the I/O channel to the point where it does not limit performance, but the reduction is significant nonetheless. I believe that in most iterative numerical algorithms idle resources can be put to use to improve performance.

Bibliography

[1] A. Aggarwal and J. S. Vitter. The input/ouput complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] 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. Also available online from `http://www.netlib.org`.

[3] Mordechai Ben Ari. On transposing large $2^n \times 2^n$ matrices. *IEEE Transactions on Computers*, C-28(1):72–75, 1979. Due to a typo, the pages in that issue are marked as Vol. C-27.

[4] C. Cleveland Ashcraft and Roger G. Grimes. On vectorizing incomplete factorization and SSOR preconditioners. *SIAM Journal on Scientific and Statistical Computing*, 9(1):122–151, 1988.

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

[6] B. Awerbuch and D. Peleg. Sparse partitions. In *Proceedings of the 31st Symposium on Foundations of Computer Science*, pages 503–513, 1990.

[7] David H. Bailey. FFTs in external or hierarchical memory. *The Journal of Supercomputing*, 4(1):23–35, 1990.

[8] David H. Bailey, Eric Barszcz, Leonardo Dagum, and Horst D. Simon. NAS parallel benchmarks results 10-94. Technical Report NAS-94-001, NASA Ames Research Center, October 1994. The latest benchmark results are available online from `http://www.nas.nasa.gov/RNR/Parallel/NPB/NPBindex.html`.

[9] 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.

[10] D. Barkai, K. J. M. Mortiarty, and C. Rebbi. A modified conjugate gradient solver for very large systems. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 284–290, August 1985.

[11] R. Barret, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadeplphia, PA, 1993.

[12] D. W. Barron and H. P. F. Swinnerton-Dyerm. Solution of simultaneous linear equations using a magnetic tape store. *Computer Journal*, 3:28–33, 1960.

[13] James V. Beck and Kenneth J. Arnold. *Parameter Estimation in Engineering and Science*. John Wiley and Sons, 1977.

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

[15] Petter E. Bjørstad. A large scale, sparse, secondary storage, direct linear equation solver for structural analysis and its implementation on vector and parallel architectures. *Parallel Computing*, 5(1):3–12, 1987.

[16] Guy E. Blelloch. *Scan primitives and parallel vector models*. PhD thesis, Massachusetts Institute of Technology, 1989. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-463.

[17] 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.

[18] 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.

[19] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–111, San Diego, CA, May 1993. Also available as Carnegie Mellon University Technical Report CMU-CS-93-112.

[20] Achi Brandt. Multilevel computations: review and recent development. In S. F. McCormick, editor, *Multigrid Methods*, pages 35–62. Marcel Dekker, 1988.

[21] N. M. Brenner. Fast Fourier transform of externally stored data. *IEEE Transactions on Audio and Electroacoustics*, AU-17:128–132, 1969.

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

[23] 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*, 1995. To appear.

[24] W. L. Briggs. *A Multigrid Tutorial*. SIAM, Philadeplphia, PA, 1987.

[25] Gilles Cantin. An equation solver of very large capacity. *International Journal for Numerical Methods in Engineering*, 3:379–388, 1971.

[26] David L. Chaiken. *Mechanisms and Interfaces for Software-Extended Coherent Shared Memory*. PhD thesis, Massachusetts Institute of Technology, 1994.

[27] S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. J. Sheffler. Modeling data-parallel programs with the alignment distribution graph. *Journal of Programming Languages*, 2(3):227–258, 1994.

[28] Peter M. Chen. *Input-Output Performance Evaluation: Self-Scaling Benchmarks, Predicted Performance*. PhD thesis, University of California at Berkeley, November 1992. Also available as Computer Science Departement Technical Report 92/714.

[29] Peter M. Chen and David A. Patterson. A new approach to I/O perfomance evaluation—self-scaling I/O benchmarks, predicted I/O performance. *Performance Evaluation Review*, 21(1):1–12, 1993. Proceedings of the *1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Santa Clara, CA.

[30] Tzu-Yi Chen. The effect of caches on the performance analysis of data parallel cm-fortran programs. In *Proceedings of the 1994 MIT Student Workshop on Scalable Computing*, pages 18–1–18–2, July 1994. Available as Technical Report MIT/LCS/TR-622.

[31] A. T. Chronopoulos and C. W. Gear. s-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25:153–168, 1989.

[32] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill, Cambridge, MA, and New York, NY, 1990.

[33] J. M. Crotty. A block equation solver for large unsymmetric matrices arising in the boundary integral equation method. *International Journal for Numerical Methods in Engineering*, 18:997–1017, 1982.

[34] 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.

[35] Elizabeth Cuthill. Digital computers in nuclear reactor design. In Franz L. Alt and Morris Rubinoff, editors, *Advances in Computers*, volume 5, pages 289–348. Academic Press, 1964.

[36] George B. Dantzig. Linear programming. In J. K. Lenstra, A. H. G Rinnoy Kan, and A. Schrijver, editors, *History of Mathematical Programming*, pages 19–31. CWI North-Holland, 1991.

[37] James Demmel. *Numerical Linear Algebra*. Berkeley Mathematics Lecture Notes. University of California, Berkeley, 1993.

[38] Peter James Denning. Queueing models for file memory operation. Master's thesis, Massachusetts Institute of Technology, 1964. A revised version was published as Project MAC Technical Report number 21.

[39] J. J. Du Cruz, S. M. Nugent, J. K. Reid, and D. B. Taylor. Solving large full sets of linear equations in a paged virtual store. *ACM Transactions on Mathematical Software*, 7(4):527–536, 1981.

[40] Ian S. Duff and Gérard Meurant. The effect of ordering on preconditioned conjugate gradient. *BIT*, 29(4):635–657, 1989.

[41] S. C. Eisenstat, M. H. Schultz, and A. H. Sherman. Software for sparse Gaussian elimination with limited core memory. In Ian. S. Duff and C. W. Stewart, editors, *Sparse Matrix Proceedings*. SIAM, Philadelphia, 1978.

[42] J. O. Eklundh. A fast computer method for matrix transposing. *IEEE Transactions on Computers*, C-21(7):801–803, 1972.

[43] 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.

[44] Bernd Fischer and Roland W. Freund. An inner product-free conjugate gradient-like algorithm for hermitian positive definite systems. In *Proceedings of the Cornelius Lanczos 1993 International Centenary Conference*, pages 288–290. SIAM, December 1993.

[45] Bernd Fischer and Roland W. Freund. On adaptive weighted polynomial preconditioning for hermitian positive definite matrices. *SIAM Journal on Scientific Computing*, 15(2):408–426, 1994.

[46] Patrick C.. Fischer and Robert L. Probert. A not one matrix multiplication in a paging environment. In *ACM '76: Proceedings of the Annual Conference*, pages 17–21, 1976.

[47] Arno Formella, Silvia M. Müller, Wolfganf J. Paul, and Anke Bingert. Isolating the reasons for the performance of parallel machines on numerical programs. In Christoph W. Kessler, editor, *Automatic Parallelization*, pages 45–77. Vieweg, 1994.

[48] Roland W. Freund, Gene H. Golub, and Noël M. Nachtigal. Itertive solution of linear systems. *Acta Numerica*, pages 1–44, 1992. An annual published by Cambridge University Press.

[49] Nikolaus Geers and Roland Klee. Out-of-core solver for large dense nonsymmetric linear systems. *Manuscripta Geodetica*, 18(6):331–342, 1993.

[50] W. M. Gentleman and G. Sande. Fast Fourier transforms for fun and profit. In *Proceedings of the AFIPS*, volume 29, pages 563–578, 1966.

[51] Alan George and Hamza Rashwan. Auxiliary storage methods for solving finite element systems. *SIAM Journal on Scientific and Statistical Computing*, 6(4):882–910, 1985.

[52] J. A. George. Nested dissection of regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10:345–363, 1973.

[53] J. A. George, M. T. Heath, and R. J. Plemmons. Solution of large-scale sparse least squares problems using auxiliary storage. *SIAM Journal on Scientific and Statistical Computing*, 2(4):416–429, 1981.

[54] Roger G. Grimes. Solving systems of large dense linear equations. *The Journal of Supercomputing*, 1(3):291–299, 1988.

[55] Roger G. Grimes and Horst D. Simon. Solution of large, dense symmetric generalized eigenvalue problems using secondary storage. *ACM Transactions on Mathematical Software*, 14(3):241–256, 1988.

[56] Ivar Gustafsson. A class of first order factorization methods. *BIT*, 18:142–156, 1978.

[57] John L. Henessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, 1990.

[58] M. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *National Bureau of Standards Jounal of Research*, 49:409–436, 1952.

[59] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1–2):1–170, 1994. Available online from `http://www.erc.msstate.edu/hpff/home.html`.

[60] Wilbur H. Highleyman. *Performance Analysis of Transaction Processing Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[61] Alan J. Hoffman. Linear programming at the National Bureau of Standards. In J. K. Lenstra, A. H. G Rinnoy Kan, and A. Schrijver, editors, *History of Mathematical Programming*, pages 62–64. CWI North-Holland, 1991.

[62] J.-W. Hong and H. T. Kung. I/O complexity: the red-blue pebble game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 326–333, 1981.

[63] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley and Sons, 1991.

[64] C. Kaklamanis, D. Krizanc, and S. Rao. New graph decompositions and fast emulations in hypercubes and butterflies. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 325–334, June 1993.

[65] James E. Kelly, Jr. An application of linear programming to curve fitting. *Journal of the Society of Industrial and Applied Mathematics*, 6(1), 1958.

[66] Tracy Kidder. *The Soul of a New Machine*. Little and Brown, Boston, 1981.

[67] K. Knobe, J. D. Lucas, and G. L. Steele Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, 1990.

[68] 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.

[69] H. T. Kung. Memory requirements for balanced computer architectures. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 49–54, June 1986.

[70] C.-C. Jay Kuo and Tony F. Chan. Two-color Fourier analysis of iterative algorithms for elliptic problems with red/black ordering. *SIAM Journal on Scientific and Statistical Computing*, 11(4):767–793, 1990.

[71] D. W. Ladd and J. W. Sheldon. The numerical solution of a partial differential equation on the IBM type 701 electronic data processing machines. In *Proceedings of the Association for Computing Machinery, ACM National Meeting*, Toronto, September 1952. Reprinted in *Annals of the History of Computing*, 5(2):142–145, 1983.

[72] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance*. Prentice-Hall, Englewood Cliffs, NJ, 1984.

[73] Charles 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.

[74] Charles E. Leiserson, Satish Rao, and Sivan Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers. In *Proceedings of the 34rd Annual Symposium on Foundations of Computer Science*, pages 704–713, November 1993. To appear in the *Journal of Computer and System Sciences*.

[75] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13(2):213–221, 1991.

[76] Joseph W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software*, 12(3):249–264, 1986.

[77] Joseph W. H. Liu. The multifrontal method and paging in sparse Cholesky factorization. *ACM Transactions on Mathematical Software*, 15(4):310–325, 1989.

[78] Joseph W. H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, 34(1):82–109, 1992.

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

[80] Donald MacKenzie. The influence of the Los Alamos and Livermore National Laboratories on the development of supercomputing. *Annals of the History of Computing*, 13(2):179–201, 1991.

[81] Allen Davis Malony. *Performance Observability*. PhD thesis, University of Illinois at Urbana-Champaign, 1990. Also available as Center for Supercomputing Research and Development Technical Report CSRD-1034.

[82] Jan Mandel. An iterative solver for p-version finite elements in three dimensions. *Computer Methods in Applied Mechanics and Engineering*, 116:175–183, 1994.

[83] A. C. McKeller and E. G. Coffman, Jr. Organizing matrices and matrix operations for paged memory systems. *Communications of the ACM*, 12(3):153–165, 1969.

[84] J. A. Meijerink and H. A. van der Vorst. An iterative method for linear systems of which the coefficient matrix is a symmetric m-matrix. *Mathematics of Computation*, 31:148–162, 1977.

[85] G. Meurant. The block preconditioned conjugate gradient method on vector computers. *BIT*, 24(4):623–633, 1984.

[86] G. Miller and W. Thurston. Separators in two and three dimensions. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 300–309, May 1990.

[87] G.L. Miller, S.-H. Teng, and S.A. Vavasis. A unified geometric approach to graph separators. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 538–547, 1991.

[88] Cleve B. Moler. Matrix computations with Fortran and paging. *Communications of the ACM*, 15(4):268–270, 1972.

[89] Digambar P. Mondkar and Graham H. Powell. Large capacity equation solver for structural analysis. *Computers and Structures*, 4:699–728, 1974.

[90] Katta G. Murty. *Linear Complementarity, Linear and Nonlinear Programming*. Heldermann Verlag, Berlin, 1988.

[91] J. Noye. Finite difference techniques for partial differential equations. In J. Noye, editor, *Computational Techniques for Differential Equations*, pages 95–354. Elsevier, 1984.

[92] William Orchard-Hays. *Advanced Linear Programming Computing Techniques*. McGraw Hill, 1968.

[93] Sharon E. Perl and William E. Weihl. Performance assertion checking. *Operating Systems Review*, 27(5):134–145, 1993. Proceedings of the *14th ACM Symposium on Operating Systems Principles*, Ashville, NC.

[94] Sharon Esther Perl. *Performance Assertion Checking*. PhD thesis, Massachusetts Institute of Technology, 1992. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-551.

[95] C. J. Pfeifer. Data flow and storage allocation for the PDQ-5 program on the Philco-2000. *Communications of the ACM*, 6(7):365–366, 1963.

[96] S. Plotkin, S. Rao, and W. Smith. Shallow excluded minors and improved graph decomposition. In *Proceedings of the 5th annual ACM-SIAM Symposium on Discrete Algorithms*, pages 462–470, 1994.

[97] Hans Riesel. A note on large linear systems. *Mathematical Tables and other Aids to Computation*, 10:226–227, 1956.

[98] V. Rohklin. Rapid solution fo intergal equation of classical potential theory. *Journal of Computational Physics*, 60:187–207, 1985.

[99] Saul Rosen. Electronic computers: a historical survey. *Computing Reviews*, 1(1):7–36, 1969.

[100] Y. Saad. Practical use of polynomial preconditioning for the conjugate gradient method. *SIAM Journal on Scientific and Statistical Computing*, 6(4):865–881, 1985.

[101] Allan Lee Scherr. *An Analysis of Time-Shared Computer Systems*. PhD thesis, Massachusetts Institute of Technology, 1962. Also published as Project MAC Technical Report number 18.

[102] Ulrich Schumann. Comments on "A fast computer method for matrix transposing" and application to the solution of Poisson's equation. *IEEE Transactions on Computers*, C-22(5):542–544, 1973.

[103] R. C. Singleton. A method for computing the fast Fourier transform with auxiliary memory and limited high-speed storage. *IEEE Transactions on Audio and Electroacoustics*, AU-15:91–98, 1967.

[104] G.D. Smith. *Numerical Solutions of Patial Differential Equations*. Oxford University Press, 2nd edition, 1978.

[105] R. D. Smith, J. K. Dukowicz, and R. C. Malone. Parallel ocean general circulation modeling. *Physica D*, 60(1-4):38–61, 1992.

[106] M. M. Stabrowski. A block equation solver for large unsymmetric linear equation systems with dense coefficient matrices. *International Journal for Numerical Methods in Engineering*, 24:289–300, 1982.

[107] William R. Swanson. *Connection Machine Run-Time System Architectural Specification, Version 7.2*. Thinking Machines Corporation, Cambridge, MA, March 1993.

[108] The MathWorks, Inc, Natick, MA. MATLAB *Reference Guide*, August 1992.

[109] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, December 1992.

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

[111] Thinking Machines Corporation, Cambridge, MA. *Prism Reference Manual,* December 1992.

[112] Thinking Machines Corporation, Cambridge, MA. *CMSSL for CM-Fortran*, June 1993.

[113] Thinking Machines Corporation, Cambridge, MA. *DPEAC Reference Manual*, March 1993.

[114] Thinking Machines Corporation, Cambridge, MA. *VU Programmer's Handbook*, August 1993.

[115] L. H. Thomas. Elliptic problems in linear difference equations over a network. Technical report, Watson Scientific Computing Laboratory, Columbia University, 1949.

[116] Eric Tomacruz, Jagesh V. Sanghavi, and Alberto Sangiovanni-Vincentelli. Algorithms for drift-diffution device simulation using massively parallel processors. *IEICE Transactions on Electronics*, E77-C(2):248–254, February 1994. Special issue on the *1993 VLSI Process and Device Modeling Workshop*.

[117] R. E. Twogood and M. P. Ekstrom. An extension of Eklundh's matrix transposition algorithm and its application in digital image processing. *IEEE Transactions on Computers*, C-25(9):950–952, 1976.

[118] H. van der Vorst. (M)ICCG for 2D problems on vector computers. In A. Linchnewsky and C. Saguez, editors, *Supercomputing*. North Holland, 1988.

[119] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM, Philadelphia, 1992.

[120] R. S. Varga. *Matrix Iterative Analysis*. Prentice Hall, Englewood Cliffs, 1962.

[121] Robert J. Venderbei. *LOQO User's Manual*. Program in Statistics & Operations Research, Princeton University, 1992.

[122] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

[123] Reinhold P. Weicker. A detailed look at some popular benchmarks. *Parallel Computing*, 17(10&11):1153–1172, 1991.

[124] Edward L. Wilson, Klaus-Jürgen Bathe, and William P. Doherty. Direct solution of large systems of linear equations. *Computers and Structures*, 4:363–372, 1974.

[125] Francis C. Wimberly. *Memory management for solution of linear systems arising in finite element substructuring*. PhD thesis, University of Pittsburgh, 1978. Also published as Institute for Computational Mathematics and Applications Technical Report ICMA-78-03.

Index