November 1992

Report No. STAN-CS-92-1459

Also numbered as CSL-TR-92-555

Thesis

Exploiting the Memory Hierarchy in Sequential and Parallel Sparse Cholesky Factorization

by

Edward Rothberg

Department of Computer Science

Stanford University

Stanford, California 94305

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE**	**3. REPORT TYPE AND DATES COVERED**

4. TITLE AND SUBTITLE
Exploiting the Memory Hierarchy in Sequential and Parallel Sparse Cholesky Factorization

5. FUNDING NUMBERS
N00039-91-C-0138

6. AUTHOR(S)
Edward Rothberg

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Stanford University
Computer Science Dept. & Electrical Engineering
Stanford, CA 94305

8. PERFORMING ORGANIZATION REPORT NUMBER
CSL-TR-92-555

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DARPA
Arlington, VA

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Cholesky factorization of large sparse matrices is an extremely important computation, arising in a wide range of domains including linear programming, finite element analysis, and circuit simulation. This thesis investigates crucial issues for obtaining high performance for this computation on sequential and parallel machines with hierarchical memory systems. The thesis begins by providing the first thorough analysis of the interaction between sequential sparse Cholesky factorization methods and memory hierarchies. We look at popular existing methods and find that they produce relatively poor memory hierarchy performance. The methods are extended, using blocking techniques, to reuse data in the fast levels of the memory hierarchy. This increased reuse is shown to provide a three-fold speedup over popular existing approaches (e.g., SPARSPAK) on modern workstations.

The thesis then considers the use of blocking techniques in parallel sparse factorization. We first describe parallel methods we have developed that are natural extensions of the sequential approach described above. These methods distribute panels (sets of contiguous columns with nearly identical non-zero structures) among the processors. The thesis shows that for small parallel machines, the resulting methods again produce substantial performance improvements over existing methods. A framework is provided for understanding the performance of these methods, and also for understanding the limitations inherent in them. Using this framework, the thesis shows that panel methods are inappropriate for large-scale parallel machines because they do not expose enough concurrency. The thesis then considers rectangular block methods, where the sparse matrix is split both vertically and horizontally. These methods address the concurrency problems of panel methods, but they also introduce a number of complications. Primary among these are issues of choosing blocks that can be manipulated efficiently and structuring a parallel computation in terms of these blocks. The thesis describes solutions to these problems and presents performance results from an efficient block method implementation.

14. SUBJECT TERMS
Hierarchical-memory machines, sparse cholesky factorization, parallel processing

15. NUMBER OF PAGES
153

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
unclassified	unclassified	unclassified	

EXPLOITING THE MEMORY HIERARCHY IN SEQUENTIAL AND PARALLEL SPARSE CHOLESKY FACTORIZATION

Edward Rothberg

Technical Report No. CSL-TR-92-555

November 1992

This thesis has been supported by DARPA contract N00039-91-C-0138.
Author also acknowledges support from an Office of Naval Research graduate fellowship.

EXPLOITING THE MEMORY HIERARCHY IN SEQUENTIAL AND PARALLEL SPARSE CHOLESKY FACTORIZATION

Edward Rothberg

Technical Report: CSL-TR-92-555

November 1992

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

Cholesky factorization of large sparse positive definite matrices is an extremely important computation, arising in a wide range of domains including linear programming, finite element analysis, and circuit simulation. This thesis focuses on crucial issues for obtaining high performance for this computation on sequential and parallel machines with hierarchical memory systems. Hierarchical memory machines offer the potential to perform this computation both quickly and cost-effectively. By structuring memory in the form of a hierarchy, with a small, high-speed cache near the processor and larger but slower levels further away, these machines allow appropriately-structured computations to behave as if all their data were stored in very fast memory. The thesis investigates how well sequential and parallel Cholesky factorization algorithms can make use of a hierarchical memory organization.

The thesis begins by providing the first thorough analysis of the interaction between sequential sparse Cholesky factorization methods and memory hierarchies. We look at popular existing methods and find that they produce relatively poor memory hierarchy performance. The methods are extended, using blocking techniques, to reuse data in the fast levels of the memory hierarchy. This increased reuse is shown to provide roughly a factor of three increase in performance on modern workstation-class machines. The primary contribution of this work is its investigation and quantification of the specific factors that affect sparse Cholesky performance on hierarchical memory machines. This work also presents and compares a disparate set of factorization methods within a consistent framework, thus isolating and identifying the important similarities and differences between the methods and unifying a large body of previously uncomparable work.

The thesis then studies the use of blocking techniques for parallel sparse Cholesky factorization. The sequential methods are quite easily extended to small-scale multiprocessors (2-16 processors), producing parallel methods that make excellent use of memory hierarchies. Data reuse is achieved by working with sets of contiguous columns, or panels. However, important scalability questions arise concerning the use of panel-oriented methods on larger parallel machines. At issue is whether panels can be made large enough to provide significant data reuse while at the same time providing enough concurrency to allow a large number of processors to be used effectively. The thesis uses a parallel performance model to understand the performance of these methods and to show that such methods are in fact inappropriate for larger hierarchical memory multiprocessors.

The thesis then proposes an alternative parallel factorization approach that manipulates rectangular sub-blocks of the matrix. This block-oriented approach is found to overcome the scalability limitations of the panel-oriented methods. However, several issues complicate its implementation. Primary among these are issues of choosing blocks in a sparse matrix that can be manipulated efficiently and structuring a parallel computation in terms of these blocks. The thesis presents solutions to these problems and investigates the parallel performance of the resulting methods. The contributions of this work come both from its theoretical foundation for understanding the factors that limit the scalability of panel- and block-oriented methods on hierarchical memory multiprocessors, and from its investigation of practical issues related to the implementation of efficient parallel factorization methods.

Key Words and Phrases: Hierarchical-memory machines, sparse Cholesky factorization, parallel processing, sparse matrices.

This report is also referenced in Computer Science Department as STAN-CS-92-1459.

Acknowledgements

I would like to thank everyone who has helped me during my years in graduate school. In particular, I would like to thank my principal advisor, Anoop Gupta. His insightful comments and seemingly boundless enthusiasm for the field were an enormous help and were greatly appreciated. I would also like to thank John Hennessy and Gene Golub for serving on my reading committee. I would also like to thank the members of the DASH group, particularly Aaron Goldberg, JP Singh, and Michael Wolf, who have been a great pleasure to work with. Finally, I would like to thank my wife Jessica for her constant support.

I would also like to acknowledge support from an Office of Naval Research graduate fellowship and from DARPA grants N00014-87-K-0828 and N00039-91-C-0138.

Contents

Abstract i

Acknowledgements iii

1 Introduction 1

1.1 Trends in Computer Architecture 1

1.2 Algorithm Design for Hierarchical Memory Organizations 4

1.3 Organization of Thesis and Summary of Results 6

2 Sparse Cholesky Factorization 8

2.1 Matching Non-Zeroes 10

2.2 Elimination Tree 10

2.3 The Multifrontal Method 11

2.4 Supernodes 13

2.5 Generalized Factorization 14

3 Sequential Sparse Cholesky Factorization 16

3.1 Introduction 16

3.2 Experimental Environment 17

3.3 Sparse Cholesky Methods 20

3.3.1 Assorted Details 20

3.3.2 Column-column Methods 21

3.3.3 Supernode-column Methods 25

3.3.4 Column-supernode Methods 30

3.3.5 Supernode-pair Methods 31

3.3.6 Supernode-supernode Methods 35

3.3.7 Supernode-matrix Methods 41

3.3.8 Summary 45

3.4 Cache Parameters 45

3.4.1 Line Size . . . 45
3.4.2 Set-Associativity . . . 46
3.4.3 Cache Size . . . 49
3.5 Alternative Blocking Strategies . . . 51
3.5.1 The Benefits of Blocking . . . 51
3.5.2 The Impact of Cache Interference . . . 54
3.5.3 Advantages and Disadvantage of Square-Block Methods . . . 58
3.6 Discussion . . . 63
3.6.1 Square-Block Methods: Performance Improvement on Benchmark Machines . . . 63
3.6.2 Improving Multifrontal Performance . . . 64
3.6.3 Choice of Primitives . . . 64
3.6.4 Performance Robustness . . . 65
3.7 Related Work and Contributions . . . 66
3.8 Conclusions . . . 67

4 Evaluation Environment for Multiprocessors **68**
4.1 Introduction . . . 68
4.2 The Stanford DASH multiprocessor . . . 68
4.3 Performance Model . . . 70
4.3.1 Computation Costs . . . 71
4.3.2 Cache Miss Counts . . . 71
4.3.3 Communication Costs . . . 73
4.4 Benchmark Matrices . . . 75

5 Parallel Panel Methods **76**
5.1 The Panel Multifrontal Method . . . 77
5.1.1 General Structure . . . 77
5.1.2 Multifrontal Method . . . 79
5.1.3 Parallel Multifrontal Example . . . 80
5.1.4 Implementation Details . . . 82
5.1.5 Distributing the Matrix Among Processors . . . 82
5.1.6 Proportional Mapping . . . 83
5.1.7 Panels . . . 83
5.1.8 Supernode Amalgamation . . . 84
5.2 Parallel Performance . . . 85
5.3 Performance Bounds . . . 86
5.3.1 Maximum Load: Load Balance and Load Efficiency . . . 86
5.3.2 Critical Path . . . 87

5.3.3 Performance Compared to Bounds 87
5.4 Improving Performance 96
5.5 Panel Size Selection 98
5.6 DASH Performance 100
5.7 Contributions 101
5.8 Conclusions 102

6 **Dense Block-Oriented Factorization** **103**
6.1 Introduction 104
6.2 Block-Oriented Factorization 104
6.3 Parallel Factorization Algorithms 106
6.3.1 Block Mapping 106
6.3.2 Destination-computes method 106
6.3.3 Source-computes method 109
6.3.4 Summary 113
6.4 Predicting Performance 113
6.5 Model Verification 115
6.6 Conclusions 115

7 **Sparse Block-Oriented Factorization** **117**
7.1 Introduction 117
7.2 Block Formulation 118
7.2.1 Block Decomposition 118
7.2.2 Structure of the Block Factorization Computation 121
7.2.3 Performance of Block Factorization 123
7.2.4 Improving Performance 125
7.2.5 Block Decomposition Summary 128
7.3 Parallel Block Method 128
7.3.1 Parallel Factorization Organization 129
7.3.2 Block Mapping for Reduced Communication 131
7.3.3 Enhancement: Domains 131
7.4 Evaluation 133
7.4.1 Small Parallel Machines 133
7.4.2 Moderately Parallel Machines 134
7.4.3 Massively-Parallel Machines 141
7.4.4 Summary 144
7.5 Discussion 145
7.6 Future Work 146

7.7 Related Work and Contributions 146
7.8 Conclusions 147

8 Conclusions **148**

List of Tables

1 Benchmark matrices. 18

2 Benchmark matrix statistics. 18

3 Performance of column-column methods on DECstation 3100 and IBM RS/6000 Model 320. 22

4 References and cache misses for column-column methods, 64K cache with 4-byte cache lines. 23

5 Performance of supernode-column methods on DECstation 3100 and IBM RS/6000 Model 320. 27

6 Mean performance numbers on DECstation 3100 and IBM RS/6000 Model 320. 27

7 References and cache misses for supernode-column methods, 64K cache with 4-byte cache lines. 28

8 Mean memory references and cache misses per floating-point operation. References are to 4-byte words. Cache is 64 KBytes with 4-byte lines. 29

9 Performance of supernode-pair methods on DECstation 3100 and IBM RS/6000 Model 320. 33

10 References and cache misses for supernode-pair methods, 64K cache with 4-byte cache lines. 33

11 Mean performance numbers on DECstation 3100 and IBM RS/6000 Model 320. . . 34

12 Mean memory references and cache misses per floating-point operation. References are to 4-byte words. Cache is 64 KBytes with 4-byte lines. 35

13 Performance of supernode-supernode methods on DECstation 3100 and IBM RS/6000 Model 320. 39

14 References and cache misses for supernode-supernode methods, 64K cache with 4-byte cache lines. 40

15 Mean performance numbers on DECstation 3100 and IBM RS/6000 Model 320. . 40

16 Mean memory references and cache misses per floating-point operation. References are to 4-byte words. Cache is 64 KBytes with 4-byte lines. 41

17 Performance of supernode-matrix methods on DECstation 3100 and IBM RS/6000 Model 320. 42
18 References and cache misses for supernode-matrix methods. 64K cache with 4-byte cache lines. 43
19 Mean performance numbers on DECstation 3100 and IBM RS/6000 Model 320. 43
20 Mean memory references and cache misses per floating-point operation. References are to 4-byte words. Cache is 64 KBytes with 4-byte lines. 44
21 Effect of increasing cache line size from 4 bytes to 64 bytes, for 64 KByte cache. Memory system traffic is measured in 4-byte words. 48
22 Effect of increasing cache set-associativity from direct-mapped to 4-way set-associative. Cache is 64 KBytes and line size is 64 bytes. Traffic is measured in 4-byte words. 48
23 Performance of square-block uncopied methods on DECstation 3100 and IBM RS/6000 Model 320. 58
24 Performance of square-block copied methods on DECstation 3100 and IBM RS/6000 Model 320. 59
25 Percentage of panel-blocked performance achieved with square-blocked codes, on DECstation 3100 and IBM RS/6000 Model 320. 59
26 Increase in memory references due to data copying. 62
27 Benchmark matrices. 75
28 Benchmark matrix statistics. 75
29 Frequency of relative index computations and scatters for block method, compared with floating-point operations ($B = 16$). 124
30 Frequency of relative index computations and scatters for block method, compared with sequential multifrontal method ($B = 16$). 125
31 Supernode amalgamation results. 127

List of Figures

1 Modern sequential machine organization . . . 2

2 Modern parallel machine organization . . . 3

3 Non-zero structure of a matrix A and its factor L . . . 8

4 Elimination tree of A. . . . 11

5 Update matrix for column 2 . . . 12

6 Assembly of update matrix from column 2 into update matrix of column 4 . . . 12

7 The *CompleteSuper()* primitive. . . . 36

8 The *ModifySuperBySuper()* primitive. . . . 37

9 Increase in data traffic due to longer cache lines. Cache size is 64 KBytes in all cases . . . 47

10 Cache miss behavior for various methods, matrix BCSSTK15 . . . 50

11 Update creation . . . 51

12 Panel blocking for update creation. . . . 52

13 Matrix-matrix multiply as a series of matrix-vector multiplies . . . 53

14 Submatrix blocking for update creation. . . . 53

15 Cache miss behavior for multifrontal supernode-matrix method, using square blocks for matrix BCSSTK15. Cache sizes 2K, 8K, and 32K. . . . 56

16 Cache miss behavior for multifrontal supernode-matrix method, using square blocks and copying, for matrix BCSSTK15. . . . 57

17 Cache miss behavior for various methods, matrix BCSSTK15. S-S is supernode-supernode. S-M is supernode-matrix. . . . 61

18 A simple grid example. . . . 78

19 Grid example. . . . 81

20 Parallel speedups for two sparse matrices. . . . 85

21 Parallel speedups for two sparse matrices, versus performance upper bounds . . . 87

22 Maximum processor utilizations when considering load imbalance alone . . . 89

23 Fraction of all floating-point operations performed within owned domains . . . 90

24 Concurrency in sparse problems. . . . 93

25 Communication to computation ratios for sparse problems . . . 95

26 Performance for panel size of 8, relative to performance of best panel size 98
27 Performance relative to best case 100
28 Parallel speedups on DASH machine 101
29 Blocks used for update operations. 106
30 A 2-D round-robin distribution. 107
31 Performance results for destination-computes method, $n = 2048$, $P = 64$ 108
32 Performance results for source-computes method, $n = 2048$, $P = 64$ 110
33 A simple block example. 111
34 Performance results for prioritized source-computes method, $n = 2048$, $P = 64$ 112
35 Performance versus problem size for destination-computes method. 114
36 Simulated and actual speedups for destination-computes method, for 25 and 36 processors. Actual speedups are from the Stanford DASH machine 116
37 Example of irregular block interaction. Dotted lines indicate boundaries of affected areas. 119
38 Example of globally partitioned matrix. 120
39 Performance of a sequential block approach, relative to a sequential left-looking supernode-supernode approach, on a single processor of the Stanford DASH machine. 124
40 Average floating-point operations per block operation. 126
41 Average floating-point operations per block operation, before and after supernode amalgamation. 127
42 Performance of a sequential block approach, before and after supernode amalgamation, relative to a sequential left-looking supernode-supernode approach 128
43 Parallel block fan-out algorithm. 130
44 Parallel speedups for block fan-out method on SGI 4D-280, $B = 24$. 134
45 Simulated parallel efficiencies for block fan-out method, $B = 24$. 135
46 Simulated parallel performance, compared with load balance upper bound ($B = 24$) 135
47 Parallel utilization upper bounds due to load balance for BCSSTK15 and BCSSTK29, compared with load balance upper bounds for dense problems ($B = 24$). In both plots, sparse and dense problems perform the same number of floating-point operations. 136
48 Communication versus computation for the block fan-out method. 138
49 Parallel speedups for block approach for BCSSTK15 and BCSSTK29. 138
50 Parallel speedups for block approach on the Stanford DASH machine. 139
51 Communication volume of block approach, relative to a panel-oriented parallel multifrontal approach. 140
52 Performance of block approach relative to performance of panel approach. 141
53 Communication pattern for row/column multicast 144

Chapter 1

Introduction

Large sparse positive definite systems of linear equations arise in a wide variety of application domains, including linear programming, finite element analysis, and process simulation. The most widely used method for solving such systems is sparse Cholesky factorization. Given a system $Ax = b$, sparse Cholesky factorization decomposes A into the form $A = LL^T$, where L is lower triangular with positive diagonal elements. The system is then solved by solving $Ly = b$ and $L^T x = y$, both of which are easily done since L is lower triangular.

Sparse Cholesky factorization is unfortunately not without its limitations. Perhaps the most important is the computational demands it makes. It is the bottleneck in applications that can require days or even weeks of machine time to solve today's problems, and in many domains the only thing preventing people from solving larger problems is the enormous runtimes they would require. As a result, there is great interest in obtaining higher performance from the sparse Cholesky computation.

Our goal in this thesis is to understand how this higher performance may be obtained. Our primary emphasis will be on obtaining not only *higher* performance, but also *cost-effective* performance. That is, our focus will be on issues that are important for obtaining high performance from inexpensive machines.

1.1 Trends in Computer Architecture

Recent trends in computer architecture have made it clear that affordable high performance is indeed achievable. The trends we are referring to are the enormous increase in the speeds of inexpensive, commodity microprocessors and the emergence of parallel processing technology to interconnect large numbers of these processors together. Engineers will soon see affordable machines with close to 1 GFLOPS performance, and active research is being done on 1 TFLOPS machines.

While the details of high-performance machines naturally vary quite a bit across machines, at the

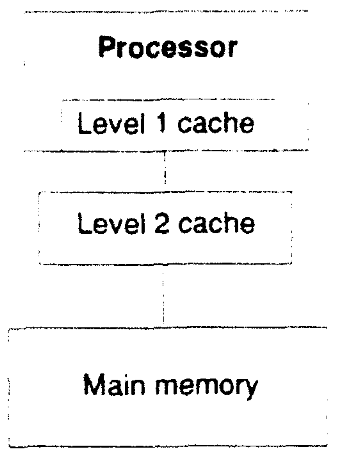


Figure 1: Modern sequential machine organization.

same time their most important aspects appear to have converged. An overview of the most common sequential machine organization is shown in Figure 1. Processors with clock speeds of 100 MHz are not uncommon in today's machines, with 200 MHz clock speeds on the horizon. An important aspect of these processors for our purposes is their potential for extremely high floating-point performance. Today's processors may perform as many as 2 floating-point operations per clock cycle (although some perform only one operation every 3 to 5 cycles).

Of course, floating-point computations can only be performed as fast as the relevant data can be fed into the floating-point units. Unfortunately, the speed of the memory from which this data is fetched has not kept up with the speed increases of processors. While it is *possible* to build a main memory that can provide data to the floating-point units as quickly as they can perform operations on this data, the cost of such a memory system would be enormous. The majority of the cost of a vector supercomputer, for example, goes to its high-bandwidth memory system.

In microprocessor-based machines, this memory bottleneck is alleviated through the use of hierarchical memory organizations, in which one or more levels of cache are interposed between the fast processor and the slow, inexpensive main memory. The caches are made up of small amounts of very high speed memory. When the processor references a memory location, a copy of that location is held in the cache so that a later reference to that location can be serviced quickly.

As for the hierarchy, the first level cache is frequently found on the actual processor chip. Fast processors can often not afford to go off-chip to fetch data. On-chip first level caches are typically quite small, since space is tight on the processor chip. Common on-chip caches today are between 8 KBytes and 32 KBytes, and they typically service processor memory requests in a single processor cycle.

While many machines are built with only a single level of cache, two-level caches are also quite

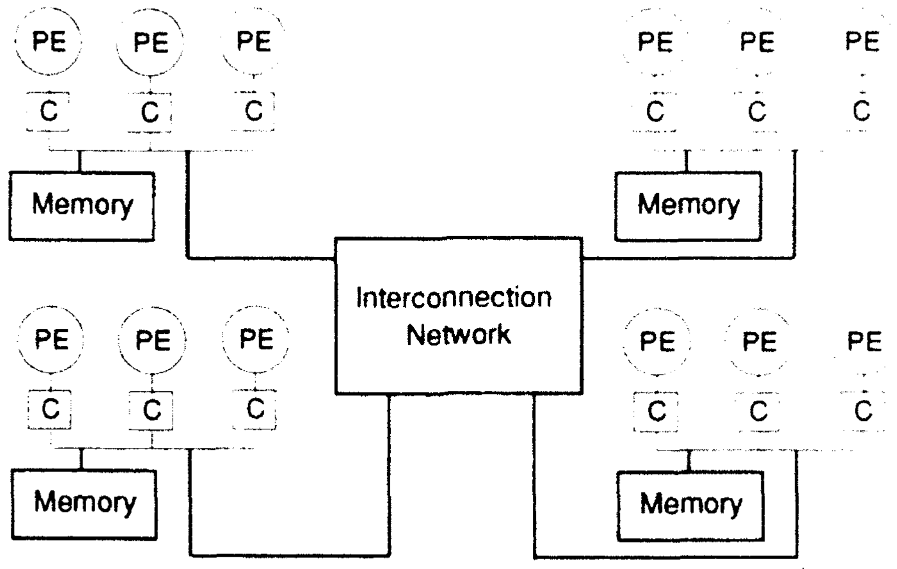


Figure 2: Modern parallel machine organization.

common, especially in machines with small on-chip first level caches. The second-level cache is generally significantly larger than the first level cache, typically containing between 64 KBytes and 1 MByte of relatively fast memory. Access times are larger than those of the first level cache, requiring anywhere from 5 to 20 cycles, but they are still significantly faster than main memory accesses, which may take anywhere from a few tens to a hundred or more cycles.

Parallel machine organizations appear to have converged as well, with virtually all modern parallel machines looking like the machine shown in Figure 2. The memory hierarchy in parallel machines is further extended due to the introduction of a distributed main memory. That is, main memory is distributed among the processors, with some portion of the global main memory being physically local to each processor. A processor/local memory combination is typically referred to as a *cluster*. A cluster often contains a single processor, although clusters with multiple processor are becoming more common. Examples include the Stanford DASH machine (4 processors per cluster) [27], the Intel Paragon MP node (4 processors per cluster), and the Thinking Machines CM5 (4 vector units per cluster).

In distributed memory parallel machines, access locality is even more important than it is in sequential machines. A memory access from a processor to a non-local portion of memory is many times more expensive than a reference to local memory (typically three or more times). Furthermore, the interconnect network generally provides relatively low aggregate interprocessor communication bandwidth. It could not possibly support the traffic that would be generated if processors were to access non-local memory locations frequently.

We should note that there are a variety programming models for distributed-memory machines

The two most common are the message-passing model, where a processor can access data in another processor's memory only by receiving a message from the other processor, and the shared-memory or uniform-address-space model, where a processor can access any location in the entire machine with ordinary memory references. This thesis will make few assumptions about which programming model a parallel machine provides.

The appeal of a hierarchical memory organization, whether for sequential or parallel machines, is clear. Machines with such an organization offer both cost-effective and scalable performance. They are cost-effective because the individual components that they are built out of, including high-speed microprocessors, slow main memories, and small amounts of high-speed cache memory, are all inexpensive. They are scalable because the machines themselves have no inherent performance limitations. For parallel programs that make good use of the memory hierarchy, so that processors service the vast majority of their memory accesses from their caches and their local memories, the performance of the program can be improved by adding more processors (with the corresponding caches and local memories).

1.2 Algorithm Design for Hierarchical Memory Organizations

The performance of a computation on a machine with a hierarchical memory organization will clearly depend on the extent to which that computation reuses data in the faster, closer levels of the hierarchy. Unfortunately, the majority of linear algebra computations, as they would most naturally be written, make very poor use of a memory hierarchy. In streaming through large matrices, these computations wind up displacing data items from the cache before they are reused, resulting in extremely high cache miss rates and low performance.

As an example, consider the matrix multiplication $Z = XY$, where all matrices are $N \times N$.

```
for $i = 1$ to $N$ do
    for $j = 1$ to $N$ do
        for $k = 1$ to $N$ do
            $Z[i,j] = Z[i,j] + X[i,k] * Y[k,j]$
```

The entries in X, Y, and Z are each reused N times throughout the course of the computation, thus providing significant opportunities to reuse data in a memory hierarchy. Unfortunately, it is extremely unlikely that the Y elements will be retained in a cache. Between one use of an element of Y and the next, the entire Y matrix is referenced. Unless the whole Y matrix fits in the cache (an unlikely prospect), each reference to Y will result in a cache miss.

Fortunately, many such computations can be reorganized through the use of *blocking* techniques to make good use of a memory hierarchy. A computation is said to be blocked when it is restructured so that a block of data that fits in the cache is intentionally reused after it has been loaded. In the matrix multiplication example above, the computation would be blocked as follows:

```
for $I = 1$ to $N/B$ do
    for $J = 1$ to $N/B$ do
        for $K = 1$ to $N/B$ do
            for $i = 1$ to $B$ do
                for $j = 1$ to $B$ do
                    for $k = 1$ to $B$ do
                        $Z[I * B + i, J * B + j] = Z[I * B + i, J * B + j] +$
                            $X[I * B + i, K * B + k] * Y[K * B + k, J * B + j]$
```

Given a particular I, J, and K iteration, the inner three loops in the above example access $B \times B$ submatrices of X, Y, and Z. The block size B can be chosen so that these submatrices are small enough to remain in the cache. As a result, the inner three loops cache miss on $3B^2$ data items, but they reference these items $3B^3$ times, thus reusing every data item B times.

A large variety of linear algebra computations can be blocked. The BLAS3 library [14], for example, provides a number of important dense matrix kernels in blocked forms, and the LAPACK linear algebra library [2] implements several important dense linear algebra computations, including dense linear system solvers and dense eigenvalue solvers, on top of these blocked BLAS3 kernels. Progress has also been made on compiler-automated blocking [12, 47].

Blocking techniques are even more relevant for parallel machines with hierarchical memory organizations, since these machine present several additional challenges for achieving high performance. The individual processors must still achieve significant data reuse to avoid the latencies associated with accessing main memory. Furthermore, in cases where several processors share a portion of main memory, data reuse is crucial for avoiding saturation of this memory. Processors must also minimize traffic on the interprocessor interconnect, both because such traffic will suffer from large latencies and also because the interconnect network may saturate. Progress has been made on performing dense matrix computations efficiently on parallel machines with memory hierarchies [3, 19, 44], again through the use of blocking techniques.

In contrast to most earlier work which has focused on blocking techniques for *dense* matrix computations, this thesis considers the use of blocking techniques for *sparse* Cholesky factorization on sequential and parallel machines with hierarchical memory organizations. Our goal is to evaluate the memory system behavior of existing approaches and to propose and evaluate new approaches that address the performance bottlenecks that are observed.

1.3 Organization of Thesis and Summary of Results

Chapter 2 begins by discussing sparse Cholesky factorization. The structure of the computation is described, and several important sparse factorization concepts are discussed.

Chapter 3 then considers specific methods for sequential sparse Cholesky factorization. It describes the data structures and computational kernels used for this computation. It also describe the three primary algorithmic approaches that are used to perform the factorization: the left-looking, right-looking, and multifrontal approaches.

Chapter 3 continues by exploring the performance of these sparse factorization approaches on hierarchical memory machines. Not surprisingly, we find that traditional approaches to the factorization, called nodal methods, achieve extremely low performance on such machines, due to their poor utilization of the memory hierarchy. We then consider methods that take advantage of the existence of supernodes (sets of columns with identical non-zero structure) to alleviate this bottleneck. We look at supernodal variants of the left-looking, right-looking, and multifrontal approaches and find that these methods achieve significantly higher performance than their nodal counterparts, due primarily to significantly better reuse of data in the memory hierarchy. Overall, we find that by restructuring the sequential sparse Cholesky computation to make better use of a cache, performance can be increased by a factor of roughly three over nodal methods on today's hierarchical memory machines. We also find that after restructuring the computation in this way, the performance differences between the left-looking, right-looking, and multifrontal variants effectively disappear.

Having established the importance of data reuse on a single processor, the thesis then turns to the issue of data reuse on a parallel machine. Before investigating specific parallel factorization methods, Chapter 4 first describes our parallel evaluation environment. We describe the Stanford DASH machine, a 64 processor machine that will provide some of our performance numbers. We also describe a parallel performance simulation model that we use to better understand the performance of parallel methods and to obtain further performance numbers.

Chapter 5 then proposes an algorithm that achieves significant data reuse for parallel sparse Cholesky factorization. The algorithm, a panel multifrontal method, is a natural extension of an existing column-oriented parallel version of the multifrontal method. The extension involves the use of contiguous sets of columns, called panels, to increase data reuse within the processors. The performance of this method is studied in detail. We find that this panel method improve performance by a factor of two to three over column methods. However, we also find that both have several serious limitations for large parallel machines. The most important is that these methods do not expose enough concurrency in the sparse problem to allow a large number of processors to be used effectively.

The limitations we run into in Chapter 5 for sparse matrices are identical to those that have been experienced by others [44] for dense matrices. These same problems have been overcome for dense matrices using a two-dimensional or block decomposition of the sparse matrix (as opposed to the one-dimensional decomposition used for the panel approach). We therefore turn our attention to the question of whether a block decomposition would provide significant benefits for sparse problems. Chapter 6 considers general issues related to the use a block decomposition for Cholesky factorization. We restrict our study in Chapter 6 to dense matrices in order to focus on the more general issues that are relevant for any block method.

Chapter 7 then considers the use of a block decomposition for sparse matrices. The main challenges for a block approach are in decomposing a sparse matrix into blocks that can be manipulated efficiently and structuring a parallel computation in terms of such blocks. Obvious approaches lead to high overheads and substantial complexity. We propose a block decomposition strategy that is both simple and efficient. Our approach is found to provide good performance on a wide range of parallel machine sizes. We compare this block method with the panel method of the previous chapter. The block method is shown to provide numerous advantages, including demonstratably higher performance on small parallel machines and asymptotically better performance on large machines.

Finally, Chapter 8 presents conclusions.

Chapter 2

Sparse Cholesky Factorization

This chapter gives a brief description of the sparse Cholesky factorization computation. The goal of sparse Cholesky factorization is to factor a sparse, symmetric, positive definite matrix A into the form $A = LL^T$, where L is lower triangular. The computation is typically performed as a series of three steps. The first step, *heuristic reordering*, reorders the rows and columns of A to reduce *fill* in the factor matrix L. A fill entry is one that is zero in the original sparse matrix but becomes non-zero during the factorization process. The second step, *symbolic factorization*, performs the factorization symbolically to determine the non-zero structure of L after the fill has occurred. Storage is allocated for L in this step. The third step is the *numerical factorization*, where the actual non-zero values in L are computed. This step is by far the most time-consuming, and it is the focus of this thesis. We refer the reader to [23] for more information on all of these steps.

To make our discussion in this chapter more concrete, we will use a simple example matrix. The example matrix and its factor are shown in Figure 3 (Dots represent non-zeroes; the diagonal elements are non-zero as well.)

$$A = \begin{pmatrix}
1 & \bullet & & \bullet & & \bullet & & & & \bullet & \\
\bullet & 2 & & & & \bullet & & & & & \\
 & & 3 & \bullet & & & & & & & \bullet \\
\bullet & & \bullet & 4 & \bullet & & & & & & \\
 & & & \bullet & 5 & \bullet & & & & & \\
\bullet & \bullet & & & \bullet & 6 & & & & & \\
 & & & & & & 7 & \bullet & & \bullet & \\
 & & & & & & \bullet & 8 & \bullet & & \\
 & & & & & & & \bullet & 9 & & \\
\bullet & & & & & & \bullet & & & 10 & \bullet \\
 & & \bullet & & & & & & & \bullet & 11
\end{pmatrix}
\qquad
L = \begin{pmatrix}
1 & & & & & & & & & & \\
\bullet & 2 & & & & & & & & & \\
 & & 3 & & & & & & & & \\
\bullet & \bullet & \bullet & 4 & & & & & & & \\
 & & & \bullet & 5 & & & & & & \\
\bullet & \bullet & & \bullet & \bullet & 6 & & & & & \\
 & & & & & & 7 & & & & \\
 & & & & & & \bullet & 8 & & & \\
 & & & & & & \bullet & \bullet & 9 & & \\
\bullet & \bullet & & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & 10 & \\
 & & \bullet & \bullet & \bullet & \bullet & & & & \bullet & 11
\end{pmatrix}$$

Figure 3: Non-zero structure of a matrix A and its factor L.

The following pseudo-code performs numerical factorization.

```
1.  for $k = 1$ to $n$ do
2.      for $i = k$ to $n$ do
3.          $L_{ik} \leftarrow L_{ik}/\sqrt{L_{kk}}$
4.      for $j = k + 1$ to $n$ do
5.          for $i = j$ to $n$ do
6.              $L_{ij} \leftarrow L_{ij} - L_{ik}L_{jk}$
```

The computation is typically expressed in terms of columns of the sparse matrix. Within a column-oriented framework, steps 2 and 3 are typically thought of as a single operation, called a column division or *cdiv*() operation. Similarly, steps 5 and 6 form a column modification, or *cmod(j, k)*, operation. The computation then looks like:

```
1.  for $k = 1$ to $n$ do
2.      cdiv(k)
3.      for $j = k + 1$ to $n$ do
4.          cmod(j, k)
```

Only the non-zero entries in the sparse matrix are stored. The standard storage scheme stores the matrix by columns, with each non-zero entry in a column having both a value and a row number associated with it. The factorization computation only performs operations on non-zeroes. This means that step 4 is only necessary when L_{jk} is non-zero. In our example matrix, column 1 would therefore modify columns 2, 4, 6, and 10. It also means that only a subset of the non-zeroes in the destination column j are affected by a *cmod(j, k)* operation.

The above formulation of the sparse Cholesky computation is typically referred to as a ***right-looking*** (or ***submatrix-Cholesky***) approach, since column k is used to modify several columns to its right in the matrix. A ***left-looking*** (or ***column-choleksy***) formulation is obtained by rearranging the loops above, giving:

```
1.  for $j = 1$ to $n$ do
2.      for $k = 1$ to $j - 1$ do
3.          cmod(j, k)
4.      cdiv(j)
```

In this case, column j is modified by several columns to its left. Note that the convention in both cases is that k iterates over source columns and j iterates over destination columns. Note also that the *cmod*() operation is performed several times per column while the *cdiv*() operation is performed only once. The *cmod*() operation therefore dominates the runtime.

In a *cmod*(j, k) operation on a sparse problem, the columns j and k generally have different non-zero structures; the structure of the destination j is a superset of the structure of source k. To add a multiple of column k into column j, the problem of matching up the appropriate entries in the columns must be solved. The left-looking and right-looking approaches to the factorization lead to three different approaches to the non-zero matching problem.

2.1 Matching Non-Zeroes

In the left-looking approach, the same destination column j is used for a number of consecutive *cmod*() operations. The non-zero matching problem is resolved by scattering the destination column into a full vector. In other words, a non-zero in row i of column j would be held in absolute position i in the full vector. Columns are added into the full destination vector using an indirection, where the destinations are determined by the non-zero structure of the source column. The full vector is gathered back into the sparse representation after all column modifications have been performed. This approach is used in the SPARSPAK sparse linear algebra package [25]. Further details will be provided in the next chapter.

A simple right-looking implementation solves the non-zero matching problem by searching through the destination to find the appropriate locations into which the source non-zeroes should be added. If the non-zeroes in a column are kept sorted by row number, which they typically are, then the search is not extremely expensive, although it is much more expensive than the simple indirection used in the left-looking approach. This approach was used in the fan-out parallel factorization code [22].

Another approach to right-looking factorization, called the *multifrontal method* [17], performs right-looking non-zero matching much more efficiently. The multifrontal method is more complicated than the methods that have been described so far, so we describe it using a simple example.

2.2 Elimination Tree

Before describing the multifrontal method, we first describe the *elimination tree* [42] of the sparse matrix, a structure that will be crucial for understanding the multifrontal method and several other methods considered in this thesis. In the elimination tree, each node represents a column of the matrix. The edges are defined as:

$$parent(j) = \min\{i \mid L_{ij} \neq 0, i > j\}.$$

In other words, the parent of column j is determined by the first sub-diagonal non-zero in column j. Equivalently, the parent of column j is the first column modified by column j. Figure 4 shows the

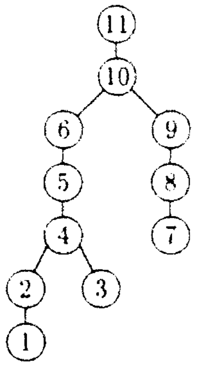


Figure 4: Elimination tree of A.

elimination tree of the example matrix from Figure 3. The elimination tree provides a great deal of information about the structure of the sparse Cholesky computation. For example, it can be shown that a column can only modify its ancestors in the elimination tree. Equivalently, a column can only be modified by its descendents.

2.3 The Multifrontal Method

Returning to our description of the multifrontal method, we note that the most important data item in this method is an update from an entire subtree in the elimination tree to some destination column. In a sequential multifrontal method, all updates from a single subtree to subsequent columns are kept together in a dense lower-triangular structure, called a *frontal update matrix*. As an example, the subtree rooted at column 2 in the matrix of Figure 3 would produce an update matrix that looks like the matrix in Figure 5. Note that the affected destination columns are a subset of the ancestor columns of that subtree. Note also that the columns of the update matrix have the same non-zero structure as the column at the root of the subtree that produces them. In the example, column 2 produces updates to rows 4, 6, and 10 in the destination columns.

To compute the frontal update matrix from a subtree rooted at some column k, the update matrices of the children of k in the elimination tree are recursively computed. These update matrices are then combined, in a step called *assembly*, into a new update matrix that has the same structure as column k. For example, the update matrix for column 4 is computed by assembling the update matrices from columns 2 and 3. The assembly of the update from column 2 into the update from column 4 is depicted in Figure 6. The update matrix from column 3 would be handled in a similar manner. The actual assembly operation typically makes use of *relative indices* [7, 42] for the child relative to the parent. These relative indices determine the locations where updates from the child update matrix are added in the destination. The relative indices in this example would be {1, 3, 4}

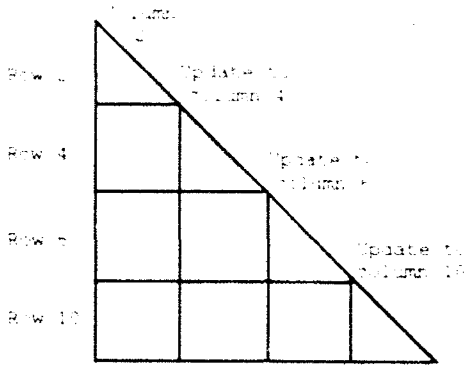


Figure 5: Update matrix for column 2

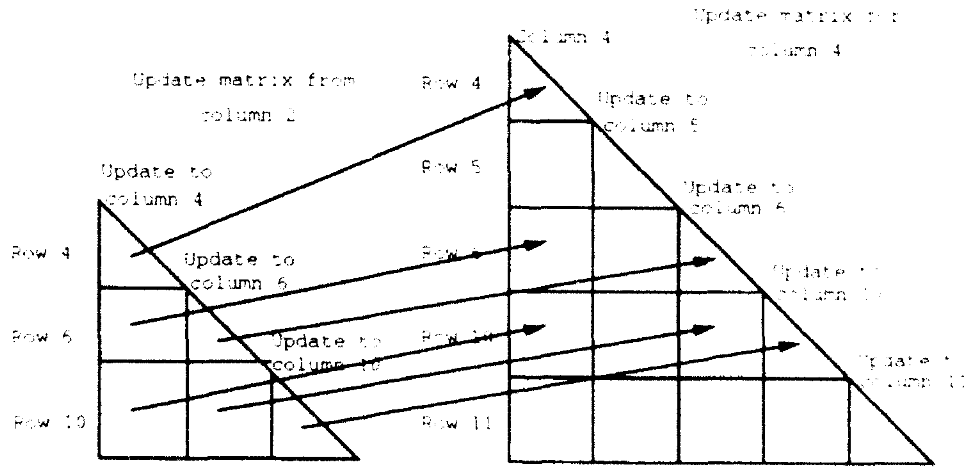


Figure 6: Assembly of update matrix from column 2 into update matrix of column 4

indicating that the first row in the child corresponds to the first row in the destination, the second row corresponds to the third, and the third row corresponds to the fourth. Note that the same correspondence holds between the columns. Once the relative indices have been computed, it is a simple matter to scatter the child update columns into the destination.

Once the child update matrices have been added into the current update matrix, the next step is to compute the final values for the entries of the current column. In the example, note that the updates from the children affect column 4 as well as columns updated by column 4. After the update matrix has been assembled, the original non-zeroes from column 4 are added into the update matrix. A *cdiv()* operation is then performed on column 4 to compute the final values in that column. The next step is to compute the updates produced directly from column 4 to the rest of the matrix. These updates are added into the update matrix. In the last step, the final values for column 4 are copied from the update matrix back into the storage for column 4.

An important issue in the multifrontal method is how the update matrices are stored. If the columns of the elimination tree are visited using a post-order traversal, then the update matrices can be kept on a stack, known as the *update matrix stack*. When a column is visited, the update matrices from its children are available at the top of the stack. They are removed from the stack, assembled, and a new update matrix is placed at the new top of the stack. The update matrix stack typically increases data storage requirements by a significant amount, ranging from 15% to 25% or more depending on the matrix. For more information on the multifrontal method, see [17].

2.4 Supernodes

An important concept in sparse Cholesky factorization is that of a *supernode*. A supernode is a set of contiguous columns in the factor whose non-zero structure consists of a dense triangular block on the diagonal, and an identical set of non-zeroes for each column below the diagonal. A supernode must also form a simple path in the elimination tree, meaning that each column in the supernode must have only one child in the elimination tree. As an example, consider the matrix of Figure 3. Columns 1 through 2 form a supernode in the factor, as do columns 4 through 6, columns 7 through 9, and columns 10 through 11. Supernodes arise in any sparse factor, and they are typically quite large.

Probably the most important property of a supernode is that each member column modifies the same set of destination columns. Thus, the Cholesky factorization computation can be expressed in terms of supernodes modifying columns, rather than columns modifying columns. A left-looking supernodal approach would look like:

```
1.  for $j = 1$ to $n$ do
2.      cdiv($j$)
3.      for each $s$ that modifies $j$ do
4.          smod($j$, $s$)
```

where *smod*(j, s) is the modification of a column j by supernode s. The modification of a column by a supernode can be thought of as a two-step process. In the first step, the modification, or update, from the supernode is computed. This update is the sum of multiples of each column in the supernode. Since all columns in the supernode have the same structure, this computation can be performed without regard for the actual non-zero structure of the supernode. The update can be computed by adding the multiples of the supernode columns together as dense vectors. In the second step, the update vector is added into the destination, taking the non-zero structure into account. Supernodes have been exploited in a variety of contexts [11, 17, 38].

The supernodal structure of the matrix is crucial to the multifrontal method, since it greatly reduces the number of assembly operations required. Since columns in a supernode share the same non-zero structure, they can share the same frontal update matrix. The update matrix therefore contains the updates from a supernode and its descendents in the elimination tree, rather than simply the updates from a single column and its descendents.

Supernodes will be exploited for a variety of purposes in this thesis.

2.5 Generalized Factorization

The three high-level approaches to sparse Cholesky factorization, the left-looking, right-looking, and multifrontal methods, have so far been expressed in terms of column-column or supernode-column modifications. This thesis will actually consider a wider range of primitives for expressing the computation. It is therefore useful to think of the factorization computation in more general terms. A generalized left-looking Cholesky factorization computation would look like:

```
1.  for $j = 1$ to $NS$ do
2.      for each $k$ that modifies $j$ do
3.          ComputeUpdateToJFromK($j$, $k$)
4.          PropagateUpdateToJFromK($j$, $k$)
5.      Complete($j$)
```

In the above pseudo-code, the *Complete*() primitive computes the final values of the elements within a structure (a column, for example), once all modifications from structures to its left have been performed. The *ComputeUpdate*() primitive computes the update from one structure to the other. The *PropagateUpdate*() primitive subsequently adds the computed update into the appropriate destination locations. In the case of the *cmod*() primitive, the computation and propagation of the update are performed as a single step. The NS term in the above pseudo-code represents the number of different destination structures in the matrix. An important thing to note is that j and k do not necessarily iterate over the same types of structures.

The right-looking and multifrontal methods generalize in a similar manner. A generalized right-looking approach would use the same primitives in a different order. A generalized multifrontal approach would be similar, but it would compute the update directly into the appropriate subtree update matrix and it would perform update propagation during the *assembly* step instead of in a *PropagateUpdate*() primitive.

This thesis will consider a range of possible choices for the structures j and k. Clearly, to be interesting choices, the chosen structures must lead to efficiently implementable primitives. For sequential factorization, we limit ourselves to three choices: columns, supernodes, and entire matrices. For parallel factorization, we will also consider panels, which are subsets of supernodes. We will give more details about how the actual factorization computation is performed in terms of these structures in later chapters.

Chapter 3

Sequential Sparse Cholesky Factorization

3.1 Introduction

This chapter will consider sparse Cholesky factorization on sequential hierarchical-memory machines. We present a comprehensive analysis of the performance of a variety of factorization methods. Our goal is to understand the impact of several important implementation decisions on the performance. The first and probably most visible implementation decision is the structure of the overall computation. We consider three common approaches: left-looking, right-looking, and multifrontal. A second, independent implementation decision is the choice of primitives on which to base the computation. The most commonly used primitives are *column-column* primitives, where columns of the matrix are used to modify other columns. We demonstrate that these primitives yield low performance on hierarchical-memory machines, primarily because they exploit very little data reuse. With such primitives, data items are fetched mainly from the more expensive levels of the memory hierarchy.

We next consider factorization methods based on supernode-column primitives, where a column is modified by an entire supernode at once. An important property of the supernode-column modification operation is that it can be *unrolled* [15]. The unrolling allows data items from the destination to be kept in processor registers across multiple modifications, thus increasing data reuse. As a result, for moderately large sparse problems, memory references are reduced by more than 50% and performance is improved by between 50% and 100%. We also consider *column-supernode* primitives, where a single column is used to modify several columns in a supernode. While the reuse benefits of such primitives are qualitatively similar to those of supernode-column primitives, the achieved benefits are much smaller.

We then consider primitives that modify several destination columns by several source columns at once. We first look at a simple case, consisting of *supernode-pair* primitives, where pairs of columns are modified by supernodes. Such methods further increase the amount of exploitable reuse. Memory references are reduced by another 35% from the supernode-column methods, and performance is improved by between 30% and 45%. We then consider the use of *supernode-supernode* primitives, where supernodes modify entire supernodes. Such primitives allow the computation to be *blocked* to increase reuse in the processor cache. Factorization codes based on these primitives further improve performance; we observe a 10% to 30% improvement over supernode-pair codes.

Finally, we look at *supernode-matrix* primitives, where a supernode is used to modify the entire matrix. The multifrontal method is typically expressed in such terms. Supernode-matrix primitives make even more data reuse available. We find, however, that the impact of this increase is small; supernode-matrix methods yield roughly the same performance as supernode-supernode methods. The reason is simply that supernode-supernode methods exploit almost all of the available reuse.

This chapter then continues by considering issues that are important for realistic cache designs, including the effects of cache size, cache line size, cache set associativity, and cache interference.

This chapter makes the following contributions to the understanding of the sparse Cholesky computation. First, it compares a number of different methods using a consistent framework. For each method, we factor the same set of benchmark matrices on the same set of machines, thus allowing for a more detailed analysis of the performance differences between the methods. This chapter also provides a detailed study of the cache behavior of the different methods. We study the impact of a number of cache parameters on the miss rates of each of the factorization methods. Finally, this chapter analyzes supernode-supernode methods, a class of methods that have so far received little attention [11]. We believe that we are the first to publish detailed performance evaluations of practical implementations of supernode-supernode methods.

The chapter is organized as follows. Section 3.2 begins by describing our experimental environment. Section 3.3 then consider several implementations of the high-level approaches based on different primitives. We look at the memory system performance of each of these variations, as well as the achieved performance on two hierarchical-memory machines. In section 3.4, we consider the consequences of changing a number of cache parameters, including the size of the cache, the size of the cache line, and the degree of set-associativity of the cache. Section 3.5 then discusses different approaches to blocking the sparse factorization computation, and considers how each approach interacts with the memory hierarchy. Finally, we discuss the results in section 3.6 and present conclusions in section 3.8.

3.2 Experimental Environment

This chapter will provide performance figures for the factorization of a range of benchmark matrices on two hierarchical-memory machines. We now describe the benchmark matrices and the machines

Table 1: Benchmark matrices

	Name	Description	Equations	Non-zeroes
1.	LSHP3466	Finite element discretization of L-shaped region	3,466	20,430
2.	BCSSTK14	Roof of Omni Coliseum, Atlanta	1,806	61,648
3.	GRID100	5-point discretization of rectangular region	10,000	39,600
4.	DENSE750	Dense symmetric matrix	750	561,750
5.	BCSSTK23	Globally Triangular Building	3,134	42,044
6.	BCSSTK15	Module of an Offshore Platform	3,948	113,868
7.	BCSSTK18	Nuclear Power Station	11,948	137,142
8.	BCSSTK16	Corps of Engineers Dam	4,884	285,494

Table 2: Benchmark matrix statistics.

	Name	Floating-point operations	Non-zeroes in factor
1.	LSHP3466	4,029,836	83,116
2.	BCSSTK14	9,795,237	110,461
3.	GRID100	15,707,205	250,835
4.	DENSE750	140,906,375	280,875
5.	BCSSTK23	119,158,381	417,177
6.	BCSSTK15	165,039,042	647,274
7.	BCSSTK18	140,919,771	650,777
8.	BCSSTK16	149,105,832	736,294

To evaluate performance, we have chosen a set of eight sparse matrices as benchmarks. These matrices are described in Tables 1 and 2. With the exception of matrices DENSE750 and GRID100, all of these matrices come from the Harwell-Boeing Sparse Matrix Collection [16]. Most are medium-sized structural analysis matrices, generated by the GT-STRUDL structural engineering program. Note that these matrices represent a wide range of matrix sparsities, ranging from the highly sparse LSHP3466, all the way to the completely dense DENSE750. In order to reduce fill in the factor, the rows/columns in all benchmark matrices are reordered using the multiple-minimum-degree heuristic [30] before the factorization.

Performance results for the factorization of these matrices will be presented in two ways in this chapter. We will typically present numbers for each matrix and summary numbers. The summary numbers will take three forms. One will be mean performance (harmonic mean) over all the benchmark matrices. In order to give some idea of how the methods perform on small and large problems, we will also present means over subsets of the benchmark matrices. In particular, we call matrices LSHP3466, BCSSTK14, and GRID100 *small matrices*, and similarly we call BCSSTK15, BCSSTK16, and BCSSTK18 *large matrices*. We do not mean to imply that the latter three matrices are large in an absolute sense. In fact, they are of quite moderate size by current standards. We simply mean that they almost fill the main memories of the benchmark machines, and thus are the largest matrices in our benchmark set.

The two machines on which we perform the sparse factorization computations are the DECstation 3100 and the IBM RS/6000 Model 320. Both are RISC machines with memory hierarchies. The DECstation 3100 uses a MIPS R2000 processor and an R2010 floating-point coprocessor, each operating at 16MHz. It contains a 64-KByte data cache, a 64-KByte instruction cache, and 16 MBytes of main memory. The machine is nominally rated at 1.6 double-precision LINPACK MFLOPS. The IBM RS/6000 Model 320 uses the IBM RS/6000 processor, operating at 20 MHz. The Model 320 contains 32 KBytes of data cache, 32 KBytes of instruction cache, and 16 MBytes of main memory. The Model 320 is nominally rated at 7.4 double-precision LINPACK MFLOPS [1].

The data cache on the DECstation 3100 is direct-mapped, meaning that each location in memory maps to a specific line in the cache. A fetched location displaces the data that previously resided in the appropriate line. Two memory data items that map to the same line and frequently displace each other are said to *interfere* in the cache. The cache lines in the DECstation 3100 are 4 bytes long.

The data cache on the IBM RS/6000 Model 320 is 4-way set-associative, meaning that each location in memory maps to any of 4 different lines in the cache. Replacement in the cache is done on an LRU, or least-recently-used basis, meaning that a fetched location displaces the least recently used of the data items that reside in its four possible lines. Each cache line contains 64 bytes.

The relative costs of various operations on these machines are quite important for understanding their performance. On the DECstation 3100, a double-precision multiply requires 5 cycles, and a double-precision add requires 2 cycles. Adds and multiplies can be overlapped in a limited manner. A single add can be performed while a multiply is going on, but an add cannot be overlapped with another add, and similarly a multiply cannot be overlapped with another multiply. The peak floating-point performance of the machine is therefore one multiply-add combination every 5 cycles. A cache miss requires roughly 6 cycles to service. A double-precision number spans two cache lines, thus requiring double the cache miss time to fetch. On the IBM RS/6000 Model 320, adds and multiplies each require two cycles to complete. However, the floating-point unit is fully pipelined, meaning that adds and multiplies can be overlapped in any possible way. A floating-point instruction can be initiated every cycle. Furthermore, the machine contains a multiply-add instruction that performs both instructions simultaneously. The RS/6000 can issue up to four different instructions in a single cycle. The peak floating-point performance of the IBM RS/6000 is one multiply-add per cycle. A cache miss on the Model 320 requires roughly 15 cycles to service, bringing in a 64-byte cache line.

From these performance numbers, it is clear that memory system costs are an extremely important component of the runtime of a matrix computation. The cost of performing floating-point arithmetic is dwarfed by the cost of moving data between the various levels of the memory hierarchy.

[1] Since we performed this study, newer models of the above machines have been released (the DECstation 5000/240, with a 40 MHz R3000 processor and the IBM RS/6000 model 980, with a 50 MHz RS/6000 processor). We expect the results presented here to be similar for the newer machines.

As a simple example, the RS/6000 requires more instructions to load three operands from the cache to processor registers than it does to perform a double-precision multiply-add operation on them. The cost of loading them from main memory is much higher. For this reason, the performance of a linear algebra program in general depends more on the memory system demands of the program than on the number of floating-point operations performed. Our analysis of factorization performance will concentrate on the memory system behavior of the various approaches.

To provide concrete numbers for comparing the memory system behaviors of the various factorization methods, we will present counts of the number of memory references and the number of cache misses a method generates in factoring a matrix. These numbers are gathered using the Tango simulation environment [13]. Tango is used to instrument the factorization programs to produce a trace of all data references the programs generate. We count these references to produce memory reference counts and feed them into a cache simulator to produce cache miss counts.

Another factor that will be important in understanding the performance of the IBM RS/6000 is the amount of instruction parallelism available in the various factorization approaches. This machine has the ability to issue up to four instructions at once, but such a capability will naturally go unused if the program is unable to perform many useful instructions at the same time. Unfortunately, the impact of this factor on performance is difficult to quantify. We will give intuitive explanations for why one approach would be expected to allow more instruction parallelism than another.

3.3 Sparse Cholesky Methods

We now consider a number of different primitives for expressing the sparse Cholesky computation. For each set of primitives, we consider left-looking, right-looking, and multifrontal implementations. Our goals are to examine the benefits derived from moving from one set of primitives to another, to examine the differences between the three high-level approaches when implemented in terms of the same primitives, and to explore the different behaviors of the methods on the two different machines. Our goal is not to explain every performance number, but instead to discuss the general issues that are responsible for the observed differences. To keep the differences as small as possible, the three approaches use identical implementations of the primitives whenever possible.

3.3.1 Assorted Details

To make the performance numbers that will be presented in this section more easily interpretable, we now provide additional details of our specific implementations. In particular, we provide details on our multifrontal implementation.

The implementation of the multifrontal method has a number of possible variations. One variation involves the particular post-order traversal that is used to order the columns. We choose the traversal order that minimizes necessary update stack space, using the techniques of [32]. We do not include the time spent determining this order in the computation times presented in the chapter.

Another possible source of variation in the multifrontal method is in the approach used to handle the update matrix stack. We use an approach that differs slightly from the traditional one, in order to remove an obvious source of inefficiency for hierarchical-memory machines. In order to add a new update matrix to the top of the stack, the multifrontal method must first consume a number of update matrices already there. A traditional implementation would compute the new update matrix at one location, remove the consumed update matrices from the top of the stack, and then copy the completed update matrix to the new top of the stack. Such copying is very expensive on a hierarchical-memory machine, so we introduce a simple trick to remove it. Rather than keeping a single update matrix stack, we keep two stacks that grow towards each other. Update matrices are consumed from the top of one stack, and produced onto the top of the other stack. Another way of thinking about this trick is in terms of the depth of a supernode in the elimination tree. The update matrices from supernodes of odd depth are kept on one stack, with the update matrices from supernodes of even depth on the other. This trick eliminates the necessity of copying update matrices. This approach is not without costs, however. We observed a 20-50% increase in the amount of stack space required. This modification introduces a tradeoff between the performance of the computation and the amount of space required to perform it. We investigate the higher performance approach.

We note that another way to obtain the benefits of this trick would be to use heap-allocated dynamic memory. That is, a multifrontal method could obtain new update matrices using *malloc*() calls (using C syntax), and it could return them to heap when finished with them using *free*() calls. One potential problem with such an approach is fragmentation. The multifrontal method requires many small update matrices near the leafs of the elimination tree and thus near the beginning of the computation. To make efficient use of storage, the resulting small memory blocks would have to be combined into larger blocks for the later stages of the computatio when fewer, larger updates are required. While many heap memory managers perform this free block combining, many others do not. We therefore do not use dynamic memory allocation in our implementations.

3.3.2 Column-column Methods

We first consider factorization approaches based on *cdiv*() and *cmod*() primitives. Since these primitives work with individual columns, we refer to the corresponding methods as column-column methods. We begin by presenting performance numbers for left-looking, right-looking, and multifrontal column-column methods (Table 3). We use double-precision arithmetic in these and all other implementations in this chapter. Note that in this and all other multifrontal implementations, one frontal update matrix is computed per supernode.

One interesting fact to note from this table is that the three methods achieve quite similar performance on the DECstation 3100. The fastest of the three methods, the multifrontal method

Table 3: Performance of column-column methods on DECstation 3100 and IBM RS/6000 Model 320.

	Left-looking		Right-looking		Multifrontal	
	MFLOPS		MFLOPS		MFLOPS	
Problem	DEC	IBM	DEC	IBM	DEC	IBM
LSHP3466	1.31	4.29	1.34	2.61	1.47	6.06
BCSSTK14	1.29	5.19	1.26	2.78	1.53	7.03
GRID100	1.31	4.56	1.22	2.67	1.44	5.93
DENSE750	0.94	5.98	1.17	8.53	1.17	8.72
BCSSTK23	0.96	5.70	0.97	3.21	1.11	7.90
BCSSTK15	0.97	5.71	0.94	2.82	1.14	8.04
BCSSTK18	0.96	5.52	0.92	2.64	1.09	7.55
BCSSTK16	1.03	5.59	0.96	2.94	1.15	7.95
Means:						
Small	1.30	4.65	1.27	2.68	1.48	6.30
Large	0.99	5.61	0.94	2.79	1.13	7.84
Overall	1.07	5.25	1.08	3.05	1.24	7.27

is roughly 16% faster than the slowest. In contrast, the multifrontal method is two to three times as fast as the right-looking method on the RS/6000. We now investigate the reasons for the obtained performance.

As was discussed earlier, one important determinant of performance is memory system behavior. We therefore begin by presenting memory system data for the three factorization methods in Table 4. The data in this table assumes a memory system similar to that of the DECstation 3100, where the cache is 64 KBytes and each cache line is 4 bytes long. While the cache on the RS/6000 has a different design and would result in different cache miss numbers, the numbers in this table will still give information about the relative cache performance of the different factorization methods. This table presents two figures for each matrix, memory references per floating-point operation and cache misses per floating-point operation. The units on all of these numbers are 4-byte words. We now discuss the reasons for the observed memory system numbers.

The *refs-per-op* numbers for the three methods can easily be understood by considering their computational kernels. Recall that the dominant operation in each method is the *cmod*() operation, in which a multiple of one column is added into another, $\mathbf{y} \leftarrow a\mathbf{x} + \mathbf{y}$. In the left-looking method, this conceptual operation is accomplished by scattering a multiple of the vector $\mathbf{x}$ into a full destination vector, using the indices of entries of $\mathbf{x}$ to determine the appropriate locations in the full vector to add the entries. The inner loop therefore looks like:

```
1.  for $i = 1$ to $n$ do
2.      $y[index[i]] = y[index[i]] + a * x[i]$
```

Table 4: References and cache misses for column-column methods. 64K cache with 4-byte cache lines.

	Left-looking		Right-looking		Multifrontal	
Problem	Refs/op	Misses/op	Refs/op	Misses/op	Refs/op	Misses/op
LSHP3466	4.22	0.30	4.32	0.19	3.82	0.17
BCSSTK14	3.88	0.39	3.99	0.43	3.53	0.32
GRID100	4.05	0.37	4.23	0.38	3.81	0.24
DENSE750	3.57	1.00	3.04	1.04	3.03	1.05
BCSSTK23	3.62	0.95	3.85	1.07	3.23	1.07
BCSSTK15	3.63	1.05	4.03	1.05	3.20	1.03
BCSSTK18	3.65	1.06	4.15	1.05	3.33	1.06
BCSSTK16	3.66	0.82	4.00	1.00	3.22	1.00
Means:						
Small	4.04	[illegible]5	4.18	0.29	3.72	0.22
Large	3.65	0.96	4.06	1.03	3.25	1.03
Overall	3.77	0.58	3.91	0.53	3.37	0.44

This kernel will be referred to as the *scatter kernel*. We assume that a resides in a processor register and generates no memory traffic during the loop. Thus, for every multiply/add pair the kernel loads one element of **x**, one index element from **index**, and one element from **y**, and writes one element of **y**. Assuming that the values are two-word double-precision floating-point numbers and the indices are single-word integers, then this kernel loads 5 words and store 2 words for every multiply/add pair, performing 3.5 memory operations per floating-point operation. This figure agrees quite well with the numbers in Table 4. The numbers in the table are understandably higher because they count all memory references performed in the entire program whereas our estimate only counts those performed in the inner loop.

The inner loop for the right-looking method is significantly more complicated than that of the left-looking method. This method adds a multiple of a vector **x** with non-zero structure **xindex** into a destination vector **y** with non-zero structure **yindex**. A search must be done in **y** for the appropriate locations into which elements of **x** should be added. The kernel looks like:

```
1.  $yi = 1$
2.  for $xi = 1$ to $n$ do
3.      while ($yindex[yi] \neq xindex[xi]$) do
4.          $yi = yi + 1$
5.      $y[yi] = y[yi] + a * x[xi]$
```

This kernel will be referred to as the *search kernel*. To perform a multiply/add, the search kernel must load one element of **x**, one element of **y**, one element of **xindex**, and *at least* one element of **yindex**. It must also write one element of **y**. The kernel would therefore be expected to perform at least 8 memory references for every multiply/add, or 4 word references for every floating-point operation. The numbers in the table are often less than this figure because of a special case in the right-looking method. One can easily determine whether the source and destination vectors have the same length. Since the structure of the destination is a superset of the structure of the source, the two vectors necessarily have the same structure if they have the same length. The index vectors can then be ignored entirely and the vectors can be added together directly.

The multifrontal method has a much simpler kernel than either of the previous two methods. Recall that the multifrontal method adds a column of the matrix into an update column, and the update column has the same non-zero structure as the updating column. Thus the computational kernel is a simple DAXPY:

```
1.  for $i = 1$ to $n$ do
2.      $y[i] = y[i] + a * x[i]$
```

This kernel loads 4 words and writes 2 words for every iteration, for a ratio of 3 memory operations per floating point operation. The multifrontal method must also combine, or *assemble*, update matrices to form subsequent update matrices. The memory references performed during assembly are responsible for the fact that the numbers in the table are larger than would be predicted by the kernel.

The cache miss rates for the three methods can be understood by considering the following. In each method, some column is used repetitively. The left-looking method modifies the destination column by several columns to its left, while the right-looking and multifrontal methods use a source column to modify several columns to its right. Thus, in each of the three $\mathbf{y} \leftarrow \mathbf{ax} + \mathbf{y}$ kernels from above, one of the two vectors $\mathbf{x}$ or $\mathbf{y}$ does not change from one invocation to the next. With a reasonably large cache, this vector would be expected to remain in the cache, implying that one vector would miss in the cache per column modification. In other words, every multiply/add pair would be expected to cache miss on one double-precision vector element, yielding a miss rate of one word per floating-point operation.

The index vectors may appear to cause significant misses as well, but recall that adjacent columns frequently have the same non-zero structures. These columns share the same index vector in the sparse matrix representation. Thus, even when the miss rate on the non-zeroes is high, the miss rate on the index structures is typically quite low.

Looking at achieved performance in the context of this memory system data, we see that the performance of the three method on the DECstation 3100 can be easily understood in terms of this memory system data. A substantial portion (roughly 35% for the larger matrices) of the runtime goes to servicing cache misses. Since the three methods generate roughly the same number of cache misses, this cost is the same for all three. The performance differences between the methods are due primarily to the differences in the number of memory references.

Understanding the performance of these methods on the IBM RS/6000 is somewhat more complicated. Again the cache miss numbers are roughly the same, but cache miss costs play a less important role on this machine. We will see in later methods that cache miss costs can have a significant effect on performance on this machine, but they are not as important as they were on the DECstation 3100. More important for the column-column methods is the amount of instruction parallelism in the computational kernels, and the extent to which the compiler can exploit it. We have examined the generated code and noticed the following. Firstly, the DAXPY kernel of the multifrontal method yields extremely efficient machine code. This is not surprising, since this kernel appears in a wide range of scientific programs, and it is reasonable to expect machines and compilers to be built to handle it efficiently. The scatter kernel of the left-looking method yields quite efficient code as well. While this kernel is not as simple or efficient as the DAXPY kernel, it is still quite easily compiled into efficient code. The search kernel of the right-looking method is another matter entirely. The kernel is quite complex, containing a loop within what would ordinarily be considered the inner loop, greatly complicating the code. This kernel meshes poorly with the available instruction parallelism in the RS/6000, yielding a much less efficient kernel.

3.3.3 Supernode-column Methods

The previous section considered factorization approaches that made no use of the supernodal structure of the matrix. In this section, we consider the effect of incorporating supernodal modifications into the computational kernel, where the update from an entire supernode is formed using dense matrix operations, and then the aggregate update is added into its destination. Supernodal elimination can be easily integrated into each of the approaches of the previous section [11].

We now consider the implementation of supernode-column primitives. Recall that our generalized phrasing of the factorization computation identifies three primitives: *ComputeUpdate*(), *PropagateUpdate*(), and *Complete*(). For a particular set of primitives, the same *ComputeUpdate*() can be used for the left-looking, right-looking and multifrontal approaches. The *PropagateUpdate*() primitive will differ among the three.

We begin by briefly describing the implementation of the update propagation step. Recall that this step begins once the update from a supernode to a column has been computed. The update has the same structure as the source supernode. As a result, the propagation steps for the left-looking, right-looking, and multifrontal supernode-column approaches are quite similar to the corresponding column-column modification operations. Note that this does not imply that the overall performance of the supernode-column and column-column methods will be similar. The propagation primitives in the supernode-column methods occur much less frequently than the modification kernels in the column-column methods, so they have a much smaller impact on performance.

We now turn our attention to the *ComputeUpdate*() step, a step that is common among the three methods. In fact, to make the three methods more directly comparable, we use the identical code for each. Recall that the update from a supernode to a column is computed using a dense rank-k update, where the K vectors used in the update are the columns of the source supernode below the diagonal of the destination. The basic kernel appears as follows

```
1.  for $k = 1$ to $K$ do
2.      for $i = 1$ to $n$ do
3.          $y[i] = y[i] + a_k * x_k[i]$
```

Each column x_k is successively added into the destination **y**. This kernel would be expected to load 3 words for every floating-point operation, since the inner loop is a DAXPY, identical to the kernel of the column-column multifrontal method. However, the number of memory references can be significantly decreased by unrolling the loop [15] over the modifying columns, as follows:

```
1.  for $k = 1$ to $K$ by 2 do
2.      for $i = 1$ to $n$ do
3.          $y[i] = y[i] + a_k * x_k[i] + a_{k+1} * x_{k+1}[i]$
```

The above loop uses 2-way unrolling. Each iteration of the inner loop now loads two elements of **x**, one element of **y**, and stores one element of **y**. The code would also perform 4 floating-point operations on this data. This gives a ratio of 2 memory operations per floating-point operation. In general, a u-way unrolled loop would perform $u + 1$ double-word loads, 1 store, and $2u$ floating-point operations per iteration, for a ratio of $1 + 2/u$ memory references per operation. Of course there is a limit to the degree of unrolling that is possible or desirable. Since the values a_k must be stored in registers to avoid memory traffic in the inner loop, the degree of unrolling is limited by the number of registers available in the machine. Unrolling also expands the size of the code, possible causing extra misses in fetching instructions from the instruction cache. Furthermore, the benefits of unrolling decrease rapidly beyond a point. For example, sixteen-way unrolling generates only 10% fewer memory references than eight-way unrolling. We perform eight-way unrolling in our implementation. Ideally, a ratio of 1.25 references per operation would be obtained.

In Table 5 we present performance numbers for the three supernode-column methods. We also present summary information for these methods and the column-column methods of the previous section in Table 6. In comparing these performance numbers we see that the supernode-column methods are significantly faster than the column-column methods, ranging from 30% faster for the multifrontal method on the IBM RS/6000, to more than 3 times faster for the right-looking method on the IBM RS/6000. We also see that the performance of the three supernode-column methods is quite similar. In particular, the overall performance on the RS/6000 differs by less than 3% among the three methods. To better explain these performance numbers, we present memory reference and cache miss numbers for these three methods in Table 7. We also present memory reference summary

Table 5: Performance of supernode-column methods on DECstation 3100 and IBM RS/6000 Model 320

	Left-looking		Right-looking		Multifrontal	
	MFLOPS		MFLOPS		MFLOPS	
Problem	DEC	IBM	DEC	IBM	DEC	IBM
LSHP3466	1.88	6.72	2.38	7.00	1.84	6.56
BCSSTK14	2.07	8.90	2.91	9.13	2.34	9.75
GRID100	1.88	7.48	2.54	7.27	1.90	7.09
DENSE750	1.69	12.61	1.71	12.89	1.68	12.42
BCSSTK23	1.60	11.13	1.80	10.15	1.70	11.06
BCSSTK15	1.66	11.31	1.97	10.90	1.86	11.58
BCSSTK18	1.56	10.24	1.86	8.97	1.70	10.18
BCSSTK16	1.66	10.94	2.15	11.01	2.02	11.61
Means						
Small	1.94	7.60	2.59	7.69	2.00	7.57
Large	1.63	10.81	1.99	10.20	1.85	11.08
Overall	1.74	9.51	2.10	9.30	1.86	9.55

Table 6: Mean performance numbers on DECstation 3100 and IBM RS/6000 Model 320

	Left-looking		Right-looking		Multifrontal	
	MFLOPS		MFLOPS		MFLOPS	
Method	DEC	IBM	DEC	IBM	DEC	IBM
Small						
Column-column	1.30	4.55	1.27	2.68	1.48	6.30
Supernode-column	1.94	7.60	2.59	7.69	2.00	7.57
Large						
Column-column	0.99	5.61	0.94	2.79	1.13	7.84
Supernode-column	1.63	10.81	1.99	10.20	1.85	11.08
Overall						
Column-column	1.07	5.25	1.08	3.05	1.24	7.27
Supernode-column	1.74	9.51	2.10	9.30	1.86	9.55

Table 7: References and cache misses for supernode-column methods. 64K cache with 4-byte cache lines.

	Left-looking		Right-looking		Multifrontal	
Problem	Refs/op	Misses/op	Refs/op	Misses/op	Refs/op	Misses/op
LSHP3466	2.61	0.25	2.35	0.09	2.68	0.17
BCSSTK14	2.05	0.39	1.92	0.08	2.14	0.16
GRID100	2.32	0.37	2.17	0.09	2.60	0.18
DENSE750	1.30	1.05	1.27	1.04	1.30	1.05
BCSSTK23	1.57	0.98	1.57	0.81	1.63	0.86
BCSSTK15	1.53	0.94	1.51	0.71	1.57	0.75
BCSSTK18	1.70	0.96	1.72	0.69	1.78	0.77
BCSSTK16	1.67	0.87	1.63	0.53	1.66	0.59
Means:						
Small	2.30	0.32	2.13	0.09	2.45	0.17
Large	1.63	0.92	1.62	0.63	1.67	0.69
Overall	1.76	0.55	1.71	0.20	1.81	0.33

information in Table 8.

Before analyzing the behavior of these methods, we make a brief observation about the multifrontal and left-looking supernode-column methods. When these methods are compared, one of the most frequently stated performance advantages of the multifrontal method [17] is its reduction in indirect addressing, and one of the most frequently stated disadvantages is that it performs more floating-point operations. We note that these two points of comparison are actually describing the advantages and disadvantages of supernodal versus nodal elimination. Recall that in both the left-looking and multifrontal methods, an update is computed from an entire supernode to a column. The resulting update must then be added into some destination. In the multifrontal method, the update is scattered into the update matrix of the parent supernode in the assembly step. In the left-looking supernode-column method, the update is scattered into a full destination vector. In each method, these are the only indirect operations that are performed, and this is virtually the only source of extra floating-point operations. Thus, the two methods are almost entirely equivalent in terms of indirect operations and extra floating-point operations.

Returning to the memory reference numbers, an important thing to note is that the numbers for the supernode-column methods are significantly lower than those for the column-column methods (see Table 8). Depending on the problem and the method, the number of references has decreased to between 45% and 55% of their previous levels. This decrease is due to two factors. First, the supernode-column methods access index vectors much less frequently. Second, the supernodal methods achieve improved reuse of processor registers due to loop unrolling. For the left-looking method, we find that the reduced index vector accesses bring references down to roughly 90% of their previous levels. The loop unrolling accounts for the rest of the decrease.

Table 8: Mean memory references and cache misses per floating-point operation. References are to 4-byte words. Cache is 64 KBytes with 4-byte lines.

	Left-looking		Right-looking		Multifrontal	
Method	Refs/op	Misses/op	Refs/op	Misses/op	Refs/op	Misses/op
Small:						
Column-column	4.04	0.35	4.18	0.29	3.72	0.22
Supernode-column	2.30	0.32	2.13	0.09	2.45	0.17
Large:						
Column-column	3.65	0.96	4.06	1.03	3.25	1.03
Supernode-column	1.63	0.92	1.62	0.63	1.67	0.69
Overall:						
Column-column	3.77	0.58	3.91	0.53	3.37	0.44
Supernode-column	1.76	0.55	1.71	0.20	1.81	0.33

Something else to note is that the references per operation numbers are well above the 1.25 ideal number. The reason is simply that not all supernodes are large enough to take full advantage of the reuse benefits of supernodal elimination.

Regarding the cache performance of the three methods, we also notice an interesting change. The cache miss numbers for the left-looking method have remained virtually unchanged between the column-column and supernode-column variants. The numbers for the right-looking and multifrontal methods, on the other hand, have decreased significantly. This fact ca be understood by considering where reuse occurs in the cache. In the left-looking column-column method, the data that is reused is the destination column. In the supernode-column left-looking method, this reuse has not changed. The destination column is expected to remain in the cache, and the supernodes that update it are expected to miss in the cache, again resulting in a miss rate of approximately one word per floating-point operation.

In the right-looking and multifrontal methods, updates are now produced from a supernode to several destination columns. The item that is reused is a supernode. We see three possibilities for the behavior of the cache, depending on the size of the supernode. If the supernode contains a single column, then the supernode is expected to remain in the cache and the destination column is expected to cache miss, resulting in one miss per floating-point operation. If the supernode contains more than one column but is smaller than the cache, then the supernode is again expected to remain in the cache, and the destination is expected to miss. However, many more floating-point operations are now being performed on each entry in the destination. In particular, if c columns remain in the cache, then we perform c times as many operations per cache miss. The third possibility, where the supernode is much larger than the processor cache, would cause the destination column to remain in the processor cache while the supernode update is being computed, as would happen in the left-looking method. The result is one miss per floating-point operation. The cache miss numbers in Table 8 indicate that the case where a supernode fits in the cache occurs quite frequently, resulting in significantly fewer misses than one miss per floating-point operation overall.

Returning to the performance numbers (Table 5), we note that the right-looking method is now the fastest on the DECstation 3100, and the left-looking method is the slowest. The primary cause of the performance differences is the cache behavior of the various methods. The right-looking method generates the fewest misses, and is therefore the fastest. Similarly, the left-looking method generates the most and is the slowest. On the RS/6000, the left-looking and right-looking methods execute at roughly the same rate. While the right-looking method has the advantage of generating fewer cache misses, it has the disadvantage of the inefficient propagation primitive.

One thing to note regarding memory system behavior for the supernode-column methods and indeed for all the methods we consider is that the multifrontal approach has an important disadvantage in comparison to the other two approaches: it performs more data movement. This is due to two subtle differences between it and the other approaches. The first is in the approaches used to determine the final values of a column. The multifrontal method gathers all updates to a column into an update matrix, adds the original values of that column into the update matrix, computes the final values, and then copies these values back into the storage for the column. The left-looking and right-looking approaches add updates directly into the destination column, thus avoiding this data shuffling and reducing the amount of data movement. The other difference relates to the manner in which supernodes containing a single column are handled. The process of producing an update matrix for a single column and then propagating it is significantly less efficient than the process used in, for example, the column-column left-looking method, where the update is computed and propagated in the same step. In the left-looking and right-looking methods, we can fall back to the column-column kernels for supernodes containing only a single column. This option does not exist for the multifrontal method. Because of these two differences, the multifrontal method will produce more memory references and more cache misses than might otherwise be expected.

3.3.4 Column-supernode Methods

The supernode-column primitives of the previous section took advantage of the fact that a single destination is reused a number of times in a supernode-column update operation to increase reuse in the processor registers. They also took advantage of the fact that every column in the source supernode had the same non-zero structure to reduce the number of accesses to index vectors. A symmetric set of primitives, where a single column is used to modify an entire supernode, would appear to have similar advantages. We briefly show in this section that while the advantages are qualitatively similar, they are not of the same magnitude.

Consider the implementation of a column-supernode *ComputeUpdate*() primitive. A column would be used to modify a set of destinations, appearing something like:

```
1.  for $j = 1$ to $J$ do
2.      for $i = 1$ to $n$ do
```

```
3.          $y_j[i] = y_j[i] + a_j * x[i]$
```

Unrolling the j loop by a factor of two yields:

```
1.  for $j = 1$ to $J$ by 2 do
2.      for $i = 1$ to $n$ do
3.          $y_j[i] = y_j[i] + a_j * x[i]$
4.          $y_{j+1}[i] = y_{j+1}[i] + a_{j+1} * x[i]$
```

The inner loop loads two entries of **y**, one entry of **x**, and stores two entries of **y**, for a total of 5 double-word references to perform 4 floating-point operations. In general, a loop that is unrolled u ways loads u entries of **y**, one entry of **x**, and writes u entries of **y** to perform $2u$ floating-point operations, for a ratio of $2 + 1/u$ memory references per floating-point operation. This ratio is still more than two-thirds of the ratio obtained without unrolling, and double the ratio obtained by unrolling the supernode-column primitive (Recall that this ratio was $1 + 2/u$ references per floating-point operation.). Thus, while column-supernode primitives realize some advantages due to reuse of data, they are not nearly as effective as supernode-column primitives. We therefore do not further study such methods.

3.3.5 Supernode-pair Methods

In this section, we consider a simple modification of the three supernode-column factorization methods that further improves the efficiency of the computational kernels and also reduces the cache miss rates. These improvements will be accomplished through the use of *supernode-pair* primitives that modify two destination columns at a time.

Devising factorization methods that make use of supernode-pair primitives is quite straightforward. For all three approaches, the *ComputeUpdate*() primitive involves a pair of simultaneous rank-k updates, using the same vectors for each update. To handle update propagation in a left-looking method, we maintain two full vectors, one for each destination column, and use the supernode-column left-looking propagation primitive to update each. The bookkeeping necessary to determine which supernodes modify both current destinations, and which modify only one or the other is not difficult. The right-looking and multifrontal methods are also quite easily modified. In both, we simply generate the updates to two destination columns at once. In the right-looking method, the two updates are propagated individually using the supernode-column right-looking propagation primitive.

The *ComputeUpdate*() step in a supernode-pair method looks like the following:

```
1.  for $k = 1$ to $K$ do
```

```
2.      for $i = 1$ to $n$ do
3.          $y_1[i] = y_1[i] + a_{1k} * x_k[i]$
4.          $y_2[i] = y_2[i] + a_{2k} * x_k[i]$
```

A set of K source vectors x_k are used to modify a pair of destination vectors y_j. This kernel can be unrolled, producing:

```
1.  for $k = 1$ to $k$ by 2 do
2.      for $i = 1$ to $n$ do
3.          $y_1[i] = y_1[i] + a_{1k} * x_k[i] + a_{1k+1} * x_{k+1}[i]$
4.          $y_2[i] = y_2[i] + a_{2k} * x_k[i] + a_{2k+1} * x_{k+1}[i]$
```

Counting memory references, we find that 2 entries of x and one entry of each y are loaded and one entry of each y is stored during each iteration. Each iteration performs 8 floating-point operations. Thus, a ratio of 1.5 memory reference per operation is achieved. In general, by unrolling u ways we achieve a ratio of $1/2 + 2/u$ memory references per operation, which is half that of the supernode-column kernel. As it turns out, the ratios are not directly comparable. The degree of unrolling is limited by the number of available registers, and the supernode-pair kernel uses roughly twice as many registers as the supernode-column kernel for the same degree of unrolling. The net effect is that on a machine with 16 registers, like the DECstation 3100, we can perform 8-by-1 unrolling (8 source columns modify one destination column), for a memory reference to floating-point operation ratio of 1.25, or we can perform 4-by-2 unrolling, for a ratio of 1.0. On the IBM RS/6000 which has 34 double-precision registers, the difference in memory references is significantly larger. We can perform 16-by-1 unrolling, for a ratio of 1.125, or we can perform 8-by-2 unrolling, for a ratio of 0.75. Another important advantage of creating two updates at a time is that each iteration of the loop updates two independent quantities, $y_1[i]$ and $y_2[i]$, leading to fewer dependencies between operations and thus increasing the amount of instruction-level parallelism.

We now present performance figures for the three supernode-pair methods (Table 9), using the identical supernode-pair kernel for each. We also present memory system data for the three methods in Table 10. The memory reference numbers are for a machine with 32 double-precision floating-point registers. These numbers are estimates, obtained by compiling the code for a machine with 16 registers and then removing by hand any references that we believe would not be necessary if the machine had 32 registers. We believe these numbers are more informative than numbers for a machine with 16 registers would be, since reference numbers for the latter would be quite similar to those of the supernode-column methods. Also, the trend in microprocessor designs appears to be towards machines with more floating-point registers.

In Tables 11 and 12 we present summary information, comparing supernode-pair methods with the methods of previous sections. The performance data shows that supernode-pair methods give

Table 9: Performance of supernode-pair methods on DECstation 3100 and IBM RS/6000 Model 320.

	Left-looking		Right-looking		Multifrontal	
	MFLOPS		MFLOPS		MFLOPS	
Problem	DEC	IBM	DEC	IBM	DEC	IBM
LSHP3466	1.95	7.86	2.57	8.01	1.95	7.71
BCSSTK14	2.28	11.74	3.08	11.39	2.52	12.36
GRID100	2.05	8.90	2.72	8.29	2.02	8.37
DENSE750	2.46	20.13	2.54	20.85	2.47	19.65
BCSSTK23	2.21	16.20	2.48	14.17	2.26	15.85
BCSSTK15	2.29	16.77	2.68	15.57	2.47	16.96
BCSSTK18	2.08	14.30	2.46	11.82	2.16	14.22
BCSSTK16	2.22	15.92	2.82	15.59	2.57	16.67
Means:						
Small	2.08	9.24	2.77	9.00	2.14	9.09
Large	2.19	15.59	2.64	14.09	2.39	15.85
Overall	2.18	12.73	2.66	12.03	2.28	12.63

Table 10: References and cache misses for supernode-pair methods, 64K cache with 4-byte cache lines.

	Left-looking		Right-looking		Multifrontal	
Problem	Refs/op	Misses/op	Refs/op	Misses/op	Refs/op	Misses/op
LSHP3466	2.08	0.21	1.77	0.09	2.06	0.17
BCSSTK14	1.50	0.29	1.38	0.08	1.56	0.16
GRID100	1.85	0.28	1.64	0.09	2.02	0.17
DENSE750	0.80	0.55	0.77	0.54	0.80	0.56
BCSSTK23	1.06	0.56	1.05	0.44	1.09	0.51
BCSSTK15	1.03	0.54	1.00	0.39	1.04	0.44
BCSSTK18	1.21	0.57	1.19	0.40	1.24	0.48
BCSSTK16	1.13	0.53	1.08	0.30	1.10	0.36
Means:						
Small	1.78	0.26	1.58	0.08	1.85	0.17
Large	1.12	0.55	1.08	0.36	1.12	0.42
Overall	1.22	0.39	1.15	0.16	1.24	0.28

Table 11: Mean performance numbers on DECstation 3100 and IBM RS/6000 Model 320

	Left-looking		Right-looking		Multifrontal	
	MFLOPS		MFLOPS		MFLOPS	
Method	DEC	IBM	DEC	IBM	DEC	IBM
Small:						
Column-column	1.30	4.65	1.27	2.68	1.48	6.30
Supernode-column	1.94	7.60	2.59	7.69	2.00	7.57
Supernode-pair	2.08	9.24	2.77	9.00	2.14	9.09
Large:						
Column-column	0.99	5.61	0.94	2.79	1.13	7.84
Supernode-column	1.63	10.81	1.99	10.20	1.85	11.08
Supernode-pair	2.19	15.59	2.64	14.09	2.39	15.85
Overall:						
Column-column	1.07	5.25	1.08	3.05	1.24	7.27
Supernode-column	1.74	9.51	2.10	9.30	1.86	9.55
Supernode-pair	2.18	12.73	2.66	12.03	2.28	12.63

significantly higher performance than the supernode-column methods. The performance increase is between 20% and 30% over the entire set of benchmark matrices for both machines, with an increase of 30% to 45% for the larger matrices. The memory reference data of Table 12 indicate that the practice of modifying two columns at a time is quite effective at reducing memory references. For all three methods, the memory reference numbers are roughly 30% below the corresponding numbers for the supernode-column methods. The supernode-pair numbers are above the ideal of 0.75, but they are still quite low.

The cache miss numbers for the supernode-pair methods are substantially lower as well. For example, the numbers are 30% lower for the left-looking method. This difference can be understood as follows. In the left-looking supernode-pair method, a pair of columns is now reused between supernode updates. When a supernode is accessed, it will typically update both columns, thus performing twice as many floating-point operations as would be done in the supernode-column method. The cache miss numbers for the right-looking and multifrontal methods have improved by roughly 15%, not nearly as much as they did for the left-looking method. Recall that we described the cache behavior of these methods in terms of the sizes of the supernodes relative to the size of the cache. Of the three cases we outlined, only the case where the supernode is larger than the cache benefits from this modification. We note that the right-looking and multifrontal cache miss rates are still significantly lower than the left-looking numbers.

Thus, the performance for supernode-pair methods can be explained as follows. The performance gains from the supernode-pair method on the DECstation 3100 are due mainly to the reduction in cache miss rates. We note that the right-looking method has the lowest miss rate of the three methods, and achieves the highest performance as well. Recall that the decrease in memory references

Table 12: Mean memory references and cache misses per floating-point operation. References are to 4-byte words. Cache is 64 KBytes with 4-byte lines

	Left-looking		Right-looking		Multifrontal	
Method	Refs/op	Misses/op	Refs/op	Misses/op	Refs/op	Misses/op
Small:						
Column-column	4.04	0.35	4.18	0.29	3.72	0.22
Supernode-column	2.30	0.32	2.13	0.09	2.45	0.17
Supernode-pair	1.78	0.26	1.58	0.08	1.85	0.17
Large:						
Column-column	3.65	0.96	4.06	1.03	3.25	1.03
Supernode-column	1.63	0.92	1.62	0.63	1.67	0.69
Supernode-pair	1.12	0.55	1.08	0.36	1.12	0.42
Overall:						
Column-column	3.77	0.58	3.91	0.53	3.37	0.44
Supernode-column	1.76	0.55	1.71	0.20	1.81	0.33
Supernode-pair	1.22	0.39	1.15	0.16	1.24	0.28

is not as relevant for the DECstation 3100, since the numbers we give assume a machine with 32 registers. The 16 registers of the DECstation limit the memory reference benefits of updating a pair of columns at a time. The performance gains on the IBM RS/6000 are due to three factors. First, the number of memory references has been significantly reduced. Second, the supernode-pair kernel updates two destinations at once in the inner loop, allowing for a greater degree of instruction parallelism. Finally, the supernode-pair method decreases the number of cache misses. The overall result is a 40% increase in performance for the larger matrices. Unfortunately, we are unable to isolate the portions of the increase in performance that come from each of these three factors.

3.3.6 Supernode-supernode Methods

An obvious extension of the supernode-pair methods of the previous section would be to consider methods that update some fixed number (greater than 2) of columns at a time. Rather than further investigating such approaches, we instead consider primitives that modify an entire supernode by another supernode. Such primitives were originally proposed in [11]. By expressing the computation in terms of supernode-supernode operations, the *ComputeUpdate*() step becomes a matrix-matrix multiply. This kernel will allow us to not only reduce traffic between memory and the processor registers through unrolling, but it will also allow us to *block* the computation to reduce the traffic between memory and the cache. The use of supernode-supernode primitives to reduce memory system traffic in a left-looking method has also been independently proposed in [36]. We use a simple form of blocking in this section. We discuss alternative blocking strategies in a later section

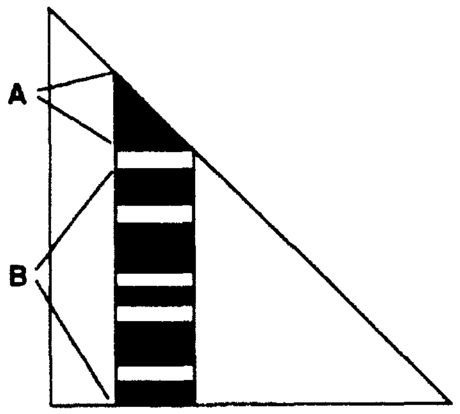


Figure 7: The *CompleteSuper*() primitive.

Implementation of Supernode-supernode Primitives

We begin our discussion of supernode-supernode methods by describing the implementation of the appropriate primitives, beginning with the *Complete*() primitive. Expressed in terms of the columns of the supernode, the *Complete*() primitive performs the following operations:

```
1.  for $j = 1$ to $n$ do
2.      for $k = 1$ to $j - 1$ do
3.          cmod($j$, $k$)
4.      cdiv($j$)
```

It will also be informative to consider the implementation of this and all other primitives in this section in terms of dense submatrices. An equivalent description of this computation, in terms of such submatrices, would be:

```
1.  $A \leftarrow Factor(A)$
2.  $B \leftarrow BA^{-1}$
```

where A is the dense diagonal block of the supernode, and B is the matrix formed by condensing the sub-diagonal non-zeroes of the supernode into a dense matrix (see Figure 7). We note that the primitives in this section will all be implemented in terms of columns of the matrix, but we will look at blocking approaches that are based on dense submatrix computations in a later section.

One thing to note about the above computation is that the inverse of A is not actually computed in step 2 above. Since A is triangular, the second step is instead accomplished by solving a sequence of triangular systems. This step can be done in-place. Another thing to note is that the entire

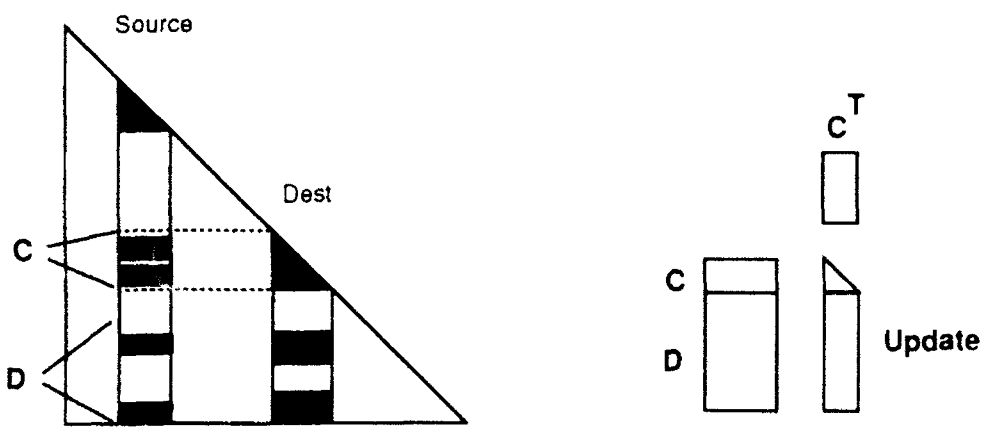


Figure 8: The *ModifySuperBySuper*() primitive.

operation can be performed without consulting the indices for the sparse columns that comprise the supernode. The whole computation can be done in terms of dense matrices.

The *ComputeUpdate*() and *PropagateUpdate*() primitives are significantly more complicated than the *Complete*() primitive. The *ComputeUpdate*() primitive produces a dense trapezoidal update matrix whose non-zero structure is a subset of the non-zero structure of the destination supernode. The *PropagateUpdate*() primitive must then add the update matrix into the destination supernode.

The *ComputeUpdate*() step involves the addition of a multiple of a portion of each column in the source supernode into the update matrix. The operation can be thought of in terms of dense submatrices as follows. Assume the destination supernode is comprised of columns d_f through d_l. The only non-zeroes in the source supernode that are involved in the computation are those at or below row d_f. These non-zeroes can be divided into two sets. The first is the matrix C of Figure 8, corresponding to the non-zeroes in the source supernode in rows d_f through d_l. The second is the matrix D, corresponding to the non-zeroes in the source supernode in rows below d_l. The upper portion of the update matrix is created by multiplying C by C^T. Since the result is symmetric, only the lower triangle is computed. The lower portion of the update matrix is created by multiplying D by C^T.

Both the *Complete*() and *ComputeUpdate*() primitives can easily be blocked to reduce the cache miss rate. We perform a simple form of blocking in this section. Each supernode of the matrix is partitioned into a set of *panels*, where a panel is a set of contiguous columns that fits wholly in the processor cache. When a *ComputeUpdate*() operation is performed, the cache is loaded with the first panel in the source supernode, and all of the contributions from this panel to the update are computed. The operation then computes contributions from the next panel.

and so on. The contributions from an individual panel are computed using the supernode-pair *ComputeUpdate*() primitive repetitively. In other words, an update is computed from the panel to each pair of destination columns. A similar scheme is employed for the *Complete*() primitive. This panel blocking is appealing because it is probably the simplest and most intuitive to implement. We will consider alternative blocking strategies in a subsequent section.

Once the update matrix is computed, the next step is propagation, where the entries of the update matrix are added into the appropriate locations in the destination supernode. In general, the update matrix contains updates to a subset of the columns in the destination supernode, and to a subset of the entries in these columns. The determination of which columns are modified is trivial. This information is available in the non-zero structure of the source supernode. The more difficult step involves the addition of a column update into its destination column. To perform this addition efficiently, we borrow the *relative index* technique [7, 42]. The basic idea is as follows. For each entry in the update column, we determine the entry in the destination column that is modified by it. This information is stored in relative indices. If $rindex[i] = j$, then the update in row i of the source should be added into row j in the destination. Since all of the columns in the update matrix have the same structure, and all of the destination columns in the destination supernode have the same structure, a single set of relative indices suffices to scatter the entire update matrix into the appropriate locations in the destination.

The only issue remaining is the question of how these relative indices are computed. The process of computing relative indices is quite similar to the process of performing a column-column modification. The main difference is that in the case of the modification, the entries are added into the appropriate locations, whereas in the case of computing indices, we simply record where those updates would be added. We therefore use quite similar methods. Note that once these indices have been computed, the left-looking and right-looking approaches can use the same code to actually perform the update propagation.

One important special case that is treated separately in both of these methods is the case of a supernode consisting of a single column. As we discussed earlier, the process of computing a large update matrix from a single column to some destination and then propagating it results in an increase in memory references and cache misses. A more efficient approach adds the updates directly into the destination supernode. It is relatively straightforward to implement such an approach, once the appropriate relative indices have been computed. The implementation involves a small modification to the supernode-supernode update propagation code, where instead of adding an entry from the update matrix into the destination, the appropriate update is computed on the spot and added into the destination. This special case code is again shared between the left-looking and right-looking methods.

We have implemented left-looking, right-looking, and multifrontal supernode-supernode methods, again using the identical *ComputeUpdate*() routine for each. Any performance differences

Table 13: Performance of supernode-supernode methods on DECstation 3100 and IBM RS/6000 Model 320.

	Left-looking		Right-looking		Multifrontal	
	MFLOPS		MFLOPS		MFLOPS	
Problem	DEC	IBM	DEC	IBM	DEC	IBM
LSHP3466	2.34	7.96	2.40	7.94	1.87	6.95
BCSSTK14	3.05	13.63	3.10	13.51	2.50	11.97
GRID100	2.49	8.86	2.56	8.51	1.87	7.02
DENSE750	3.75	22.77	3.83	22.81	3.68	21.38
BCSSTK23	3.34	19.20	3.34	18.50	2.97	17.34
BCSSTK15	3.52	20.50	3.57	20.01	3.17	18.55
BCSSTK18	3.07	15.98	3.02	15.22	2.61	14.54
BCSSTK16	3.41	19.36	3.47	19.14	3.12	18.21
Means:						
Small	2.59	9.62	2.66	9.45	2.04	8.11
Large	3.32	18.40	3.34	17.87	2.94	16.89
Overall	3.05	14.01	3.09	13.72	2.58	12.27

between the three approaches are due entirely to three differences between the methods. First, the relative indices are computed in different ways. Second, the multifrontal method performs more data movement. And finally, the methods execute the primitives in different orders, potentially leading to different cache behaviors.

Performance of Supernode-supernode Methods

We now present performance numbers for the supernode-supernode methods. Table 13 gives factorization rates on the two benchmark machines, and Table 14 gives memory system information. We present comparative information between these and previous methods in Tables 15 and 16. These tables show that the performance of the supernode-supernode methods is again higher than that of the previous methods, giving performance that is 10% to 40% higher than that of a supernode-pair method on the DECstation 3100 over the whole set of benchmark matrices, and 0% to 10% higher on the IBM RS/6000. For the larger matrices, supernode-supernode methods are 20% to 50% faster on the DECstation, and 5% to 20% faster on the IBM.

Moving to the cache miss information, we note that the miss rates for the three methods are similar, and in all cases they are substantially lower than those achieved by the supernode-pair methods. For the larger problems, miss rates have decrease by a factor of more than 2 for the right-looking and multifrontal methods, and by a factor of more than 3.5 for the left-looking method. The reason is the effectiveness of the blocking at reducing cache misses.

The observed performance can therefore be explained as follows. For the larger problems, the

Table 14: References and cache misses for supernode-supernode methods. 64K cache with 4-byte cache lines.

	Left-looking		Right-looking		Multifrontal	
Problem	Refs/op	Misses/op	Refs/op	Misses/op	Refs/op	Misses/op
LSHP3466	1.85	0.13	1.80	0.11	2.22	0.17
BCSSTK14	1.36	0.11	1.35	0.11	1.59	0.16
GRID100	1.74	0.13	1.68	0.11	2.22	0.18
DENSE750	0.81	0.16	0.81	0.16	0.84	0.17
BCSSTK23	1.03	0.17	1.03	0.17	1.12	0.22
BCSSTK15	0.99	0.14	0.99	0.14	1.07	0.18
BCSSTK18	1.16	0.19	1.16	0.19	1.29	0.25
BCSSTK16	1.06	0.13	1.06	0.14	1.12	0.18
Means:						
Small	1.62	0.12	1.58	0.11	1.96	0.17
Large	1.07	0.15	1.06	0.15	1.15	0.20
Overall	1.16	0.14	1.16	0.14	1.29	0.18

Table 15: Mean performance numbers on DECstation 3100 and IBM RS/6000 Model 320.

	Left-looking		Right-looking		Multifrontal	
	MFLOPS		MFLOPS		MFLOPS	
Method	DEC	IBM	DEC	IBM	DEC	IBM
Small:						
Column-column	1.30	4.65	1.27	2.68	1.48	6.30
Supernode-column	1.94	7.60	2.59	7.69	2.00	7.57
Supernode-pair	2.08	9.24	2.77	9.00	2.14	9.09
Supernode-supernode	2.59	9.86	2.66	9.45	2.04	8.11
Large:						
Column-column	0.99	5.61	0.94	2.79	1.13	7.84
Supernode-column	1.63	10.81	1.99	10.20	1.85	11.08
Supernode-pair	2.19	15.59	2.64	14.09	2.39	15.85
Supernode-supernode	3.32	18.40	3.34	17.87	2.94	16.89
Overall:						
Column-column	1.07	5.25	1.08	3.05	1.24	7.27
Supernode-column	1.74	9.51	2.10	9.30	1.86	9.55
Supernode-pair	2.18	12.73	2.66	12.03	2.28	12.63
Supernode-supernode	3.05	14.01	3.09	13.72	2.58	12.27

Table 16: Mean memory references and cache misses per floating-point operation. References are to 4-byte words. Cache is 64 KBytes with 4-byte lines.

	Left-looking		Right-looking		Multifrontal	
Method	Refs/op	Misses/op	Refs/op	Misses/op	Refs/op	Misses/op
Small:						
Column-column	4.04	0.35	4.18	0.29	3.72	0.22
Supernode-column	2.30	0.32	2.13	0.09	2.45	0.17
Supernode-pair	1.78	0.26	1.58	0.08	1.85	0.17
Supernode-supernode	1.62	0.12	1.58	0.11	1.96	0.17
Large:						
Column-column	3.65	0.96	4.06	1.03	3.25	1.03
Supernode-column	1.63	0.92	1.62	0.63	1.67	0.69
Supernode-pair	1.12	0.55	1.08	0.36	1.12	0.42
Supernode-supernode	1.07	0.15	1.06	0.15	1.15	0.20
Overall:						
Column-column	3.77	0.58	3.91	0.53	3.37	0.44
Supernode-column	1.76	0.55	1.71	0.20	1.81	0.33
Supernode-pair	1.22	0.39	1.15	0.16	1.24	0.28
Supernode-supernode	1.16	0.14	1.16	0.14	1.29	0.18

cache miss rates have decreased dramatically, leading to higher performance. For the smaller problems, performance in many cases has decreased, because the effort spent searching for opportunities to increase reuse is wasted. Overall, supernode-supernode methods significantly increase performance over supernode-pair methods.

3.3.7 Supernode-matrix Methods

We now consider methods based on primitives that produce updates from a single supernode to the entire remainder of the matrix. The multifrontal method is most frequently expressed in terms of such primitives (see, for example, [1]). One thing to note about supernode-matrix methods is that they are all right-looking. We therefore are restricted to two different approaches, right-looking and multifrontal.

Before discussing the implementation of supernode-matrix primitives, we note that the *Complete()* and *ComputeUpdate()* primitives are typically merged into a single operation. The final values of the supernode are determined at the same time that the updates from the supernode to the rest of the matrix are computed. Our implementations perform these as a single step as well, but our discussion is simplified if we consider them as separate steps.

We now briefly discuss the implementation of supernode-matrix primitives. The implementation of *ComputeUpdate()* is relatively straightforward. The trapezoidal update matrix from the supernode-supernode methods becomes a lower triangular matrix. This update matrix is computed

Table 17: Performance of supernode-matrix methods on DECstation 3100 and IBM RS/6000 Model 320.

	Right-looking		Multifrontal	
	MFLOPS		MFLOPS	
Problem	DEC	IBM	DEC	IBM
LSHP3466	2.43	8.33	1.96	7.69
BCSSTK14	3.08	13.56	2.54	12.58
GRID100	2.59	9.06	2.03	8.43
DENSE750	3.85	22.81	3.63	21.29
BCSSTK23	3.32	18.63	2.94	17.77
BCSSTK15	3.52	20.16	3.02	18.85
BCSSTK18	3.01	15.63	2.65	15.52
BCSSTK16	3.42	19.26	3.14	18.53
Means:				
Small	2.67	9.86	2.15	9.14
Large	3.30	18.13	2.92	17.50
Overall	3.09	14.10	2.63	13.27

by performing a symmetric matrix-matrix multiplication, $C = BB^T$, where B is the portion of the source supernode below the diagonal block. Since the result matrix C is symmetric, only the lower triangle is computed. We use the same panel-based blocking as we did for the supernode-supernode methods to reduce cache misses.

The propagation of the update matrix for the right-looking method is done using the propagation code from the right-looking supernode-supernode method. In fact, the supernode-supernode and supernode-matrix right-looking methods are nearly identical. The difference is in the order in which primitives are invoked. In the supernode-supernode method, the update to a single supernode is immediately added into the destination. In the right-looking supernode-matrix method, the updates from a single supernode to all destination supernodes are computed, and then these updates are propagated one at a time to the appropriate destination supernodes.

In Table 17 we present performance numbers for the supernode-matrix schemes, and in Table 18 we present memory system information. We present comparative information in Tables 19 and 20. Surprisingly, the performance of the supernode-matrix methods is quite similar to the performance of the corresponding supernode-supernode methods.

One would expect that in moving from an approach that produces updates from a supernode to a single destination supernode to an approach that produces updates from a supernode to the entire rest of the matrix, the amount of exploitable reuse would increase significantly. The memory reference figures in Table 18 indicate that this is not the case. A number of factors account for the lack of observed increase. The most important factor has to do with the relative sizes of supernode-matrix and supernode-supernode updates. Specifically, a single supernode-matrix update typically corresponds to a small number of supernode-supernode updates. Therefore, little reuse is lost in

Table 18: References and cache misses for supernode-matrix methods, 64K cache with 4-byte cache lines.

	Right-looking		Multifrontal	
Problem	Refs/op	Misses/op	Refs/op	Misses/op
LSHP3466	1.75	0.11	2.08	0.17
BCSSTK14	1.33	0.12	1.56	0.16
GRID100	1.64	0.12	2.03	0.17
DENSE750	0.81	0.16	0.84	0.17
BCSSTK23	1.02	0.18	1.11	0.21
BCSSTK15	0.98	0.15	1.06	0.17
BCSSTK18	1.14	0.19	1.26	0.24
BCSSTK16	1.06	0.14	1.11	0.17
Means:				
Small	1.55	0.12	1.86	0.17
Large	1.06	0.16	1.14	0.19
Overall	1.15	0.14	1.26	0.18

Table 19: Mean performance numbers on DECstation 3100 and IBM RS/6000 Model 320

	Left-looking		Right-looking		Multifrontal	
	MFLOPS		MFLOPS		MFLOPS	
Method	DEC	IBM	DEC	IBM	DEC	IBM
Small:						
Column-column	1.30	4.65	1.27	2.68	1.48	6.30
Supernode-column	1.94	7.60	2.59	7.69	2.00	7.57
Supernode-pair	2.08	9.24	2.77	9.00	2.14	9.09
Supernode-supernode	2.59	9.86	2.66	9.45	2.04	8.11
Supernode-matrix	-	-	2.67	9.86	2.15	9.14
Large:						
Column-column	0.99	5.61	0.94	2.79	1.13	7.84
Supernode-column	1.63	10.81	1.99	10.20	1.85	11.08
Supernode-pair	2.19	15.59	2.64	14.09	2.39	15.85
Supernode-supernode	3.32	18.40	3.34	17.87	2.94	16.89
Supernode-matrix	-	-	3.30	18.13	2.92	17.50
Overall:						
Column-column	1.07	5.25	1.08	3.05	1.24	7.27
Supernode-column	1.74	9.51	2.10	9.30	1.86	9.55
Supernode-pair	2.18	12.73	2.66	12.03	2.28	12.63
Supernode-supernode	3.05	14.01	3.09	13.72	2.58	12.27
Supernode-matrix	-	-	3.09	14.10	2.63	13.27

Table 20: Mean memory references and cache misses per floating-point operation. References are to 4-byte words. Cache is 64 KBytes with 4-byte lines.

	Left-looking		Right-looking		Multifrontal	
Method	Refs/op	Misses/op	Refs/op	Misses/op	Refs/op	Misses/op
Small:						
Column-column	4.04	0.35	4.18	0.29	3.72	0.22
Supernode-column	2.30	0.32	2.13	0.09	2.45	0.17
Supernode-pair	1.78	0.26	1.58	0.08	1.85	0.17
Supernode-supernode	1.62	0.12	1.58	0.11	1.96	0.17
Supernode-matrix	-	-	1.55	0.12	1.86	0.17
Large:						
Column-column	3.65	0.96	4.06	1.03	3.25	1.03
Supernode-column	1.63	0.92	1.62	0.63	1.67	0.69
Supernode-pair	1.12	0.55	1.08	0.36	1.12	0.42
Supernode-supernode	1.07	0.15	1.06	0.15	1.15	0.20
Supernode-matrix	-	-	1.06	0.16	1.14	0.19
Overall:						
Column-column	3.77	0.58	3.91	0.53	3.37	0.44
Supernode-column	1.76	0.55	1.71	0.20	1.81	0.33
Supernode-pair	1.22	0.39	1.15	0.16	1.24	0.28
Supernode-supernode	1.16	0.14	1.16	0.14	1.29	0.18
Supernode-matrix	-	-	1.15	0.14	1.26	0.18

splitting a supernode-matrix update into a set of supernode-supernode updates.

Another important reason for the lack of improvement is the existence of significant fractions of the computation that are not affected by the change from supernode-matrix to supernode-supernode updates. One example is the propagation of updates, a step that is performed by each of the methods. This computation achieves extremely poor data reuse, and generates a significant fraction of the total cache misses. For example, the assembly step in the multifrontal supernode-supernode method accounts for roughly 12% of the memory references, yet it generates roughly 30% of the total cache misses. The reuse in this step is not increased in going to supernode-matrix primitives.

The small performance differences between the supernode-supernode and supernode-matrix methods are easily understood. Cache miss numbers are nearly identical for the two, making a significant component of runtime identical. Memory reference figures are slightly lower for the supernode-matrix methods, especially for the the small problems. The main reason is simply that the increased task size of the supernode-matrix methods leads to slightly fewer conflicts between the numbers of columns in the task and the degree of unrolling. The supernode-matrix methods achieve slightly higher performance overall on both machines.

3.3.8 Summary

This section has studied the performance of a wide variety of methods for performing sparse Cholesky factorization. In doing so, we have identified the aspects of these methods that are important for achieving high performance. We showed that performance depends most heavily on the efficiencies of the primitives used to manipulate structures in the matrix. The simplest primitives, in which columns modified other columns, led to low performance. They also led to large differences in performance among the left-looking, right-looking, and multifrontal approaches, since each of these approaches used different primitive implementations. As the structures manipulated by the primitives increased in size, the efficiency of these primitives increased as well. The primary source of performance improvement was the increased amount of reuse that was exploited within the primitives. Another effect of using primitives that manipulated larger structures was that the differences between the left-looking, right-looking, and multifrontal approaches decreased. These primitives allowed more of the factorization work to be performed within code that could be shared among the three approaches. Thus, while the conventional wisdom had previously been that the high-level approach, whether left-looking, right-looking, or multifrontal, is an important determinant of performance, we have shown that it actually has a very small impact.

Our attention so far has been focused on the performance of sparse factorization methods on two specific benchmark machines. We now attempt to broaden the scope of our study by considering the effect of varying a number of machine parameters. In particular, we consider the impact of different cache designs.

3.4 Cache Parameters

This chapter has so far only considered a memory system similar to the one found on the DECstation 3100, a 64 KByte direct-mapped cache with one-word cache lines. We now consider a number of variations on cache design, including different cache sizes, different cache line sizes, and different set-associativities.

3.4.1 Line Size

A common technique for decreasing the aggregate amount of latency a processor incurs in waiting for cache misses is to increase the amount of data that is brought into the cache in response to a miss. In a standard implementation of this technique, the cache is divided into a number of *cache lines*. A miss on any location in a line brings the entire line into the cache. This practice is of course only beneficial if the extra data that is brought in is requested by the processor shortly after being loaded. Many programs possess this *spatial locality* property. We now evaluate the extent to which this property is present in sparse Cholesky factorization.

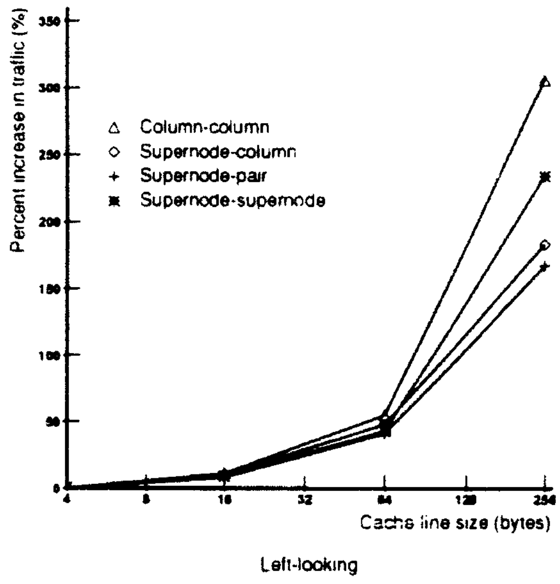
In Figure 9 we show the magnitude of the increase in total data traffic that results when the size of the cache line is increased. These figures show the percent increase in cache traffic, averaged over the entire benchmark matrix set, when using a particular factorization method on a cache with a larger line size, as compared with the traffic generated by the same method on a cache with a 4-byte line size. Note that we do not mean to imply by our data traffic measure that an increase in traffic is bad. An increase is almost unavoidable, since more data is fetched than is requested. The data traffic measure is simply a means of obtaining an absolute sense of the effect of an increased line size. The amount of traffic increase that constitutes effective use of a cache line is difficult to quantify, and in general depends on the relation between the fixed and the per-byte costs of servicing a miss. We note that in moving to a 16 byte line size, the increase in traffic is between 5% and 10%, thereby reducing the total number of misses by almost a factor of four. This represents an excellent use of longer cache lines. On the other hand, traffic is increased by between 100% and 350% when we move to a 256 byte line. While the result is a factor of between 14 and 32 decrease in cache misses, it is not clear that the cost of moving 2 to 4.5 times as much data between memory and the cache will be made up for by the decrease in the number of misses.

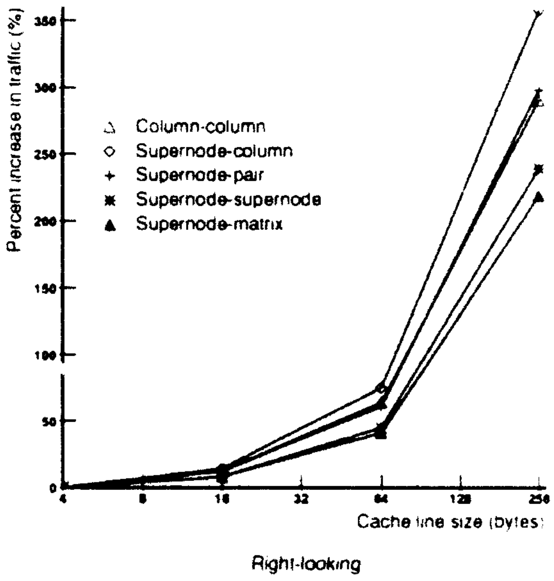
Of the three high-level approaches, the data shows that the multifrontal approach is best able to exploit long cache lines. This is to be expected, since this method performs its work within dense update matrices. The data brought into the cache on the same line as a fetched item almost certainly belongs to the update matrix that is currently active. The other two methods frequently work with disjoint sets of columns. Data fetched in the same cache line as a requested data item often belong to an adjacent column that is not relevant to the current context. Also, for reasons that will become clear in a later section, the fact that the update matrix occupies a contiguous area in memory means that the multifrontal method incurs less cache interference than the other methods. Cache interference has a larger impact on overall miss rates when cache lines are long. The extra data movement in the multifrontal method therefore has some benefit to offset its cost on machines with long cache lines.

We now focus on a subset of the above data. In order to better understand the performance of the IBM RS/6000 Model 320, we look at the cache miss numbers for a cache with 64 byte cache lines, which is the line size of this machine. Table 21 shows the increase in traffic for this line size as well as the absolute amount of cache traffic that results for each of the methods. These numbers are again averaged over the entire benchmark set. Note that while the increase in traffic is smallest for the multifrontal approach, the overall miss rates for the multifrontal approach are still higher than those of the other approaches.

3.4.2 Set-Associativity

Another technique to reduce the aggregate cost of cache misses is to increase the set-associativity of the cache. As we mentioned earlier, a direct-mapped cache maps each memory location to a specific





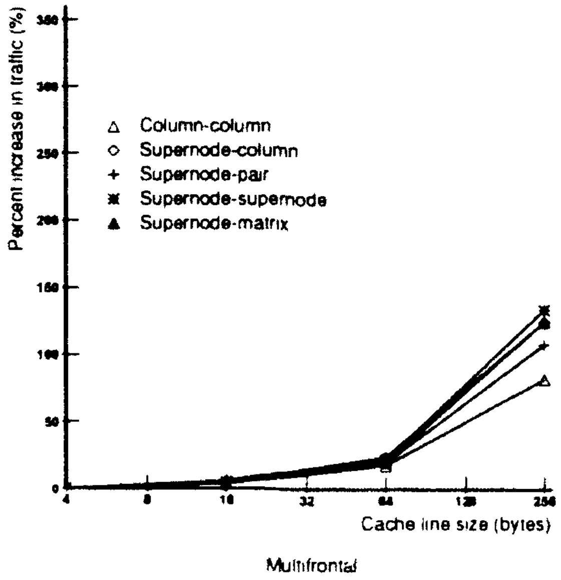


Figure 9: Increase in data traffic due to longer cache lines. Cache size is 64 KBytes in all cases.

Table 21: Effect of increasing cache line size from 4 bytes to 64 bytes, for 64 KByte cache. Memory system traffic is measured in 4-byte words.

	Left-looking		Right-looking		Multifrontal	
Problem	Traffic: Words/op	Incr. in traffic	Traffic: Words/op	Incr. in traffic	Traffic: Words/op	Incr. in traffic
Column-column	0.90	55%	0.88	64%	0.52	18%
Supernode-column	0.82	48%	0.34	75%	0.41	24%
Supernode-pair	0.54	41%	0.27	61%	0.33	20%
Supernode-supernode	0.20	43%	0.20	45%	0.22	22%
Supernode-matrix	-	-	0.20	41%	0.22	20%

Table 22: Effect of increasing cache set-associativity from direct-mapped to 4-way set-associative. Cache is 64 KBytes and line size is 64 bytes. Traffic is measured in 4-byte words.

	Left-looking		Right-looking		Multifrontal	
Problem	Traffic: Words/op	Decr. in traffic	Traffic: Words/op	Decr. in traffic	Traffic: Words/op	Decr. in traffic
Column-column	0.56	38%	0.46	48%	0.47	10%
Supernode-column	0.61	25%	0.19	45%	0.32	23%
Supernode-pair	0.43	20%	0.16	39%	0.26	21%
Supernode-supernode	0.15	26%	0.14	31%	0.18	19%
Supernode-matrix	-	-	0.14	29%	0.18	19%

location in the cache. When a data item is brought into the cache, the item that previously resided in the same cache location is displaced. A problem arises when two frequently accessed memory locations map to the same cache line. To reduce this problem, caches are often made with a small degree of set-associativity, where each memory location maps to some small set of cache locations. When a memory location is brought into the cache, it displaces the contents of one member of this set. With an LRU (least recently used) replacement policy, the displaced item is the one that was least recently accessed. While set-associative caches are slower and more complicated than direct-mapped caches, they often result in a substantial decrease in the number of cache misses incurred.

In Table 22 we show data traffic volumes (measured in 4-byte words per floating-point operation) for a 64 KByte, 4-way set-associative cache with 64-byte lines. Note that these parameters are quite similar to those of the RS/6000 cache, the only difference being in the cache size. The table also shows the percent decrease in traffic when 4-way set-associativity is added to a 64 KByte cache with 64-byte lines. These numbers are again averages over the entire benchmark set. We see from these numbers that set-associativity produces a significant miss rate reduction for all of the factorization methods.

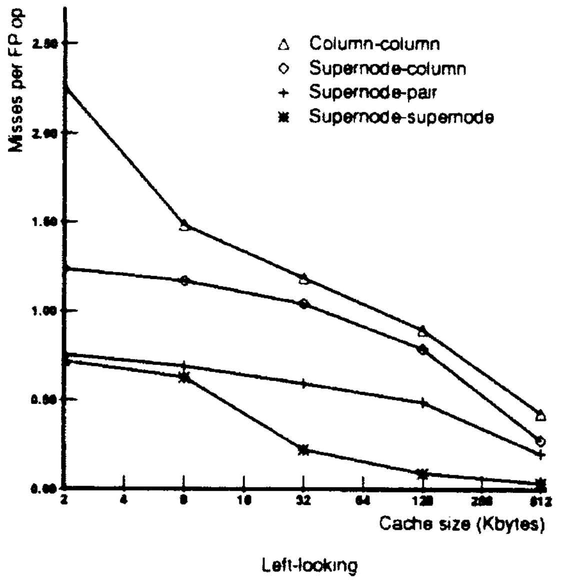
3.4.3 Cache Size

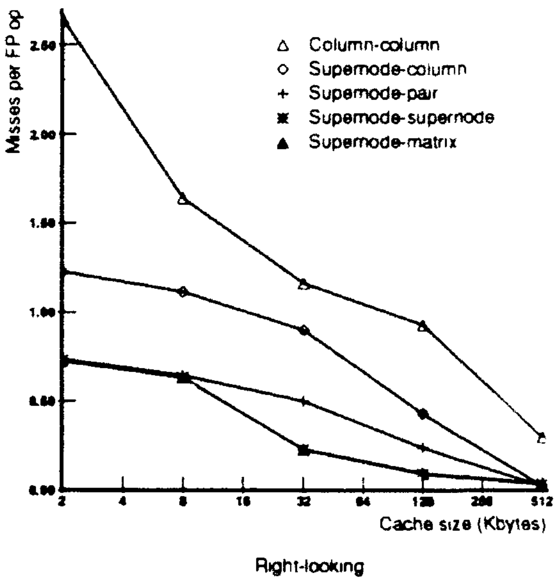
Up to this point in this chapter, we have only presented cache miss data for 64 KByte caches We now consider the effect of varying the size of the cache for the various methods that we have considered. The curves of Figures 10 show the miss rates for a range of cache sizes for matrix BCSSTK15. The three graphs depict the cache behavior for the three different high-level approaches (left-looking, right-looking, and multifrontal), and the individual curves within each chart show the cache behaviors for the different primitives. In the interest of saving space we show charts for a single matrix, BCSSTK15. We have looked at data for other matrices, and found their behavior to be quite similar. Similar charts for other matrices can be found in [38].

While exact explanations of the observed behavior would be impractical, we now provide brief, intuitive explanations. We begin by noting that a 2 KByte cache yields roughly 100% cache read miss rates for each of the methods, implying that the differences in behavior between the various approaches are determined by the number of read references that the approaches generate per floating-point operation. The multifrontal method generates the fewest references among the column-column methods, thus it generates the fewest misses. Similarly, the supernode-column methods generates fewer references than the column-column methods, explaining their lower cache miss numbers.

As the size of the cache increases, we observe two distinct types of behavior. The methods that do not attempt to reuse data (column-column, supernode-column, and supernode-pair methods) realize a gradual decrease in miss rate, as more of the matrix is accidentally reused in the cache. Note that the miss figures fall more quickly for the two right-looking methods, because of the different manner in which reuse is achieved. As an example, note that the left-looking and right-looking supernode-column methods achieve roughly equal miss rates with a 2 KByte cache. When the cache size is increased to 128 KBytes, the left-looking method incurs nearly twice as many misses as the right-looking method. From a previous discussion, we know that right-looking methods achieve enhanced reuse when supernodes fit within the cache. A larger cache makes it more likely that supernodes will fit. The left-looking methods do not share such benefits.

The methods that block the computation to reuse data (supernode-supernode, and supernode-matrix methods) show significantly different behavior. At a certain cache size, which happens to be roughly 8K for this matrix, the miss rates begin to fall off dramatically. This is because the blocking strategy relies on sets of columns fitting in the cache. When the cache is small, one or fewer columns fit, thus achieving no benefit from the blocking. Once the cache is large enough to hold a few columns, then the benefit of blocking the computation begins to grow. We observe that the miss rates fall off quickly for the blocked approaches.





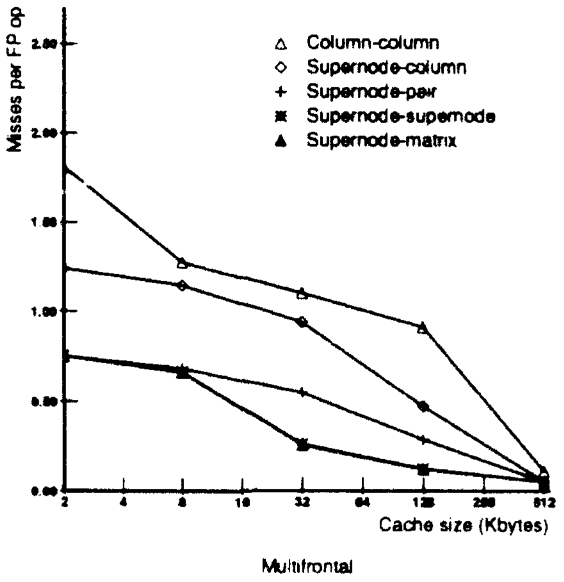


Figure 10: Cache miss behavior for various methods, matrix BCSSTK15.

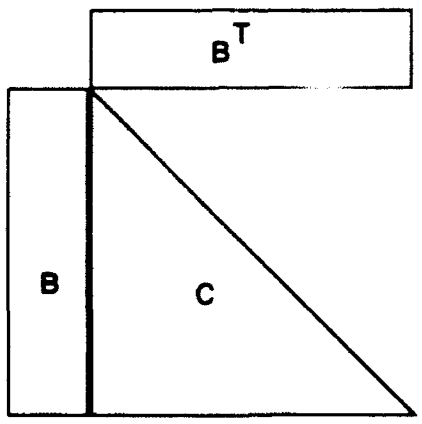


Figure 11: Update creation.

3.5 Alternative Blocking Strategies

It is clear from the figures of the previous section that the simple *panel-based* blocking strategy that has been employed so far is not very effective for small caches. The reason is clear: the amount of achieved reuse depends on the number of columns that fit in the cache. In a small cache, few columns fit so the reuse is minimal. This section considers the use of different blocking strategies. We consider the impact of a strategy where square blocks are used instead of long, narrow panels.

3.5.1 The Benefits of Blocking

We begin by describing an alternate strategy for blocking the sparse factorization computation, and describing the potential advantages of such an approach. This blocking strategy will be described in terms of the multifrontal supernode-matrix method, although the discussion applies to the other supernode-supernode and supernode-matrix methods as well. Recall that in the multifrontal method, a supernode is used to create an update matrix. Consider the matrices of Figure 11. The B matrix represents the non-zeroes within the supernode. The matrix B is multiplied by its transpose to produce the update matrix C. In the previous section, this computation was blocked by splitting B vertically, into a number of narrow panels. Figure 12 shows the case where the supernode is split into two panels. A panel is loaded into the cache and multiplied by a block-row of the transpose of B, which is actually the transpose of the panel itself. The result is added into C. We now briefly examine the advantages and disadvantages of such an approach.

To better understand the panel-oriented blocked matrix multiply, it is convenient to think of the matrix-matrix multiply as a series of matrix-vector multiplies. The matrix in one matrix-vector multiply is a portion of the panel that is reused in the cache; the vector is a single column from the

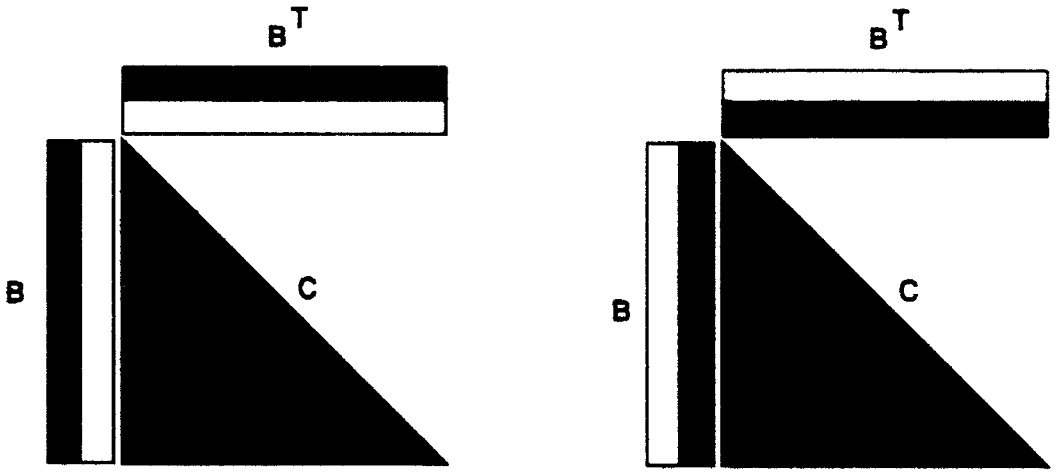


Figure 12: Panel blocking for update creation.

transpose of the panel; the destination vector is a column from the destination matrix (see Figure 13). Thus, to produce one column of the destination, the code touches the entries in the block, one column from the transpose, and of course the de ination column. In terms of cache behavior, we expect the block to remain in the cache across the entire matrix multiply. Consequently, once the block has been fetched into the cache, the fetching of both the block and the column from the transpose causes no cache misses. Only the destination column causes cache misses. If we assume that a panel is r rows long and c columns wide and that such a panel fits in the processor cache, then $2cr(r+1)/2$ operations are performed, and $rc + r(r+1)/2$ cache misses are generated in computing the entire update from a single panel. It is reasonable to assume that a panel is much longer than it is wide, so we can ignore the rc term in the cache miss number. By taking the ratio of the resulting quantities, we see that $2c$ floating-point operations are performed for every cache miss. The problem with such an approach is that the program has no control over the number of columns in the panel. The parameter c is determined by the size of the cache and the lengths of the columns in the source supernode.

As we saw in the previous section, with small caches and large matrices the panel dimension c may be too small to provide significant cache benefits. It is clearly desirable to allow a blocked program to control the dimensions of the block. The benefits of doing so have been discussed in a number of papers (see, for example, [19]). We now briefly explain these benefits.

In a sub-block approach, the matrix B is divided both vertically and horizontally. A single sub-block of B is loaded into the cache, and is multiplied by a block-row from B^T. The result is added into a block-row of C (see Figure 14). As can be seen in the figure, the contribution from a sub-block of B to C is computed by performing a matrix-matrix multiply of the block with the transpose of the blocks above it in the same block-column. The lower-triangular update that is added into the

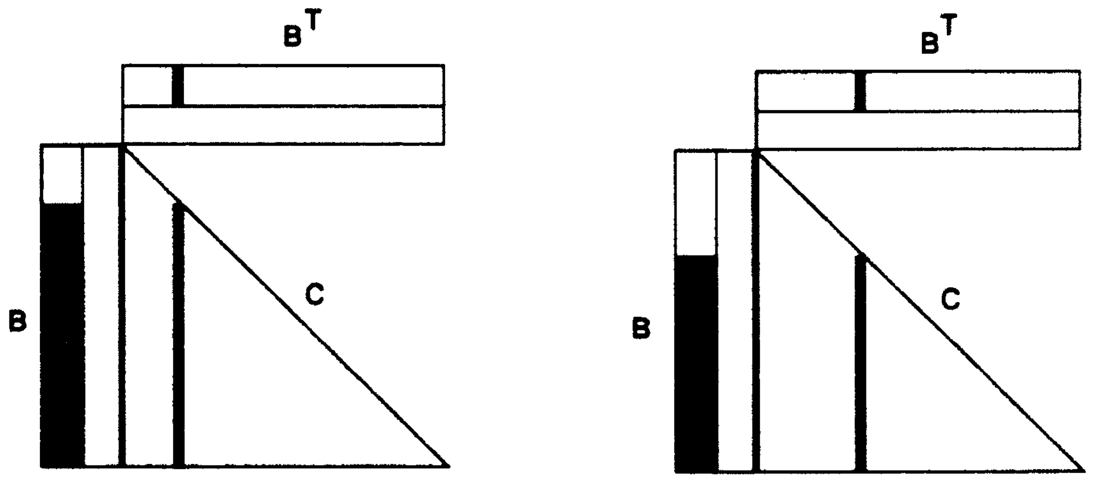


Figure 13: Matrix-matrix multiply as a series of matrix-vector multiplies.

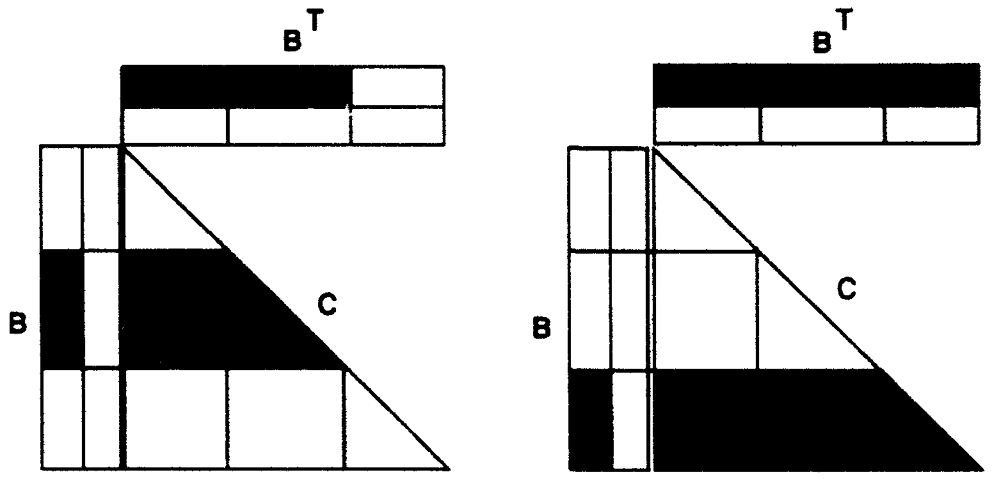


Figure 14: Submatrix blocking for update creation.

diagonal of C is computed by performing a matrix-matrix product of the block with its transpose.

To explain the cache behavior of such an approach, we again consider the block matrix-matrix multiply as a series of matrix-vector multiplies. In this case, we can choose the dimensions $r \times c$ of the block to be reused in the cache. For each matrix-vector multiply, the reused block remains in the cache, while a column of length c is read from the transpose and a column of length r is read from the destination. In the sub-block case, the column from the transpose does not come from the block that is reused in the cache (except in the infrequent case where the block on the diagonal is being computed). In terms of operations and cache misses, $2rc$ operations are performed during each matrix-vector multiply, and cache misses are generated for a column of length r and a column of length c. We again assume that the initial cost of loading the reused block into the cache can be ignored. To maximize performance, we wish to minimize the number of cache misses per floating-point operation, subject to the constraint that the $r \times c$ block must fit in the cache. In other words, we want to minimize $r + c$ subject to the constraint that $rc < C$, where C is the size of the cache. This minimum is achieved when $rc = C$ and $r = c = \sqrt{C}$. Thus, the maximum number of operations per cache miss is $2c^2/2c = c$, and that maximum is achieved using square blocks that fill the cache. This ratio may appear worse than the $2c$ ratio obtained with a panel-oriented approach, but recall that c will be much larger in general for square-block approaches.

3.5.2 The Impact of Cache Interference

Since the use of a square-block approach has the potential to greatly increase reuse for large matrices and small caches, we now evaluate factorization methods based on such an approach. The implementation of a multifrontal square-block method is relatively straightforward. We have implemented such a method and simulated its cache behavior. The results were somewhat surprising. The miss rates for small caches were slightly lower than those obtained from panel-blocked approaches, but they were not nearly as low as would have been predicted by the previous discussion. The reason is that the analysis of the previous discussion assumed that some amount of data would remain in the cache across a number of uses. The problem is that even though the data that was assumed to remain in the cache was smaller than the cache size, much of it nonetheless did not remain in the cache between uses. We now consider the reasons for this cache behavior and consider methods for improving it.

To understand the cause of interference in the cache, it is first important to understand how a cache is built. Recall that the primary benefit of a cache is that the data contained in it can be accessed extremely quickly. The cache must consequently be able to quickly determine whether it contains a requested data item and if so where it is held. In order to keep caches fast, they must be kept extremely simple. One of the most common means of designing a simple, fast cache is to build it like a hash table, where a particular data location can only reside in one location in the cache. Such a design is called a direct-mapped cache. A slightly more complicated design, the set-associative cache, maps a data location to some small set of cache locations. In either case, the determination of whether a data location is held in the cache is as simple as determining which cache locations could contain that data location, and then determining whether the data item is indeed present in any of them. The hash function, called an address mapping function, is typically extremely simple, almost always using the address of the data item modulo some power of two to determine the cache location in which that data location would reside. Computationally, such a mapping function corresponds to the use of some number of low-order bits of the data address, yielding an extremely inexpensive function to compute.

One important consequence of such a cache design is that the amount of data that can be held in the cache at one time is determined not only by the size of the set of data, but also by whether each data item in the set maps to a different location in the cache. If any two items map to the same location (or any $a + 1$ items in a set-associative cache of degree a), then they displace each other. We now consider the relevance of this fact to the blocking approaches that have been discussed so far.

Both panel-blocking and square-blocking assume that some block of the matrix remains in the cache across multiple uses. In the case of panel-blocking, the block that is reused and is assumed to remain in the cache corresponds to the non-zeroes from a panel, a set of adjacent columns whose size is less than the size of the cache. One important property of a panel is that the non-zeroes of its member columns are stored contiguously in memory, and thus cannot possible interfere with each other in the cache. Contiguous data locations map to contiguous cache locations, making interference impossible.

If we consider the case of square-blocking, we note that this approach assumes that a square sub-block whose size is less than the cache size remains in the cache. However, in this case, the sub-block does not occupy a contiguous address range in memory. Whenever we advance from one column of the sub-block to the next, a jump in memory addresses occurs. Consequently, it is possible for one column to reside in the same cache locations as the previous column, or indeed any other column of the block (we termed the resulting cache interference *self-interference* in [26]). It is therefore extremely likely that a block whose size is roughly equal the size of the cache will experience self-interference. In fact, the impact of such self-interference is typically extremely large, requiring a significant fraction of the block expected to remain the cache to be reloaded for each use. This interference is responsible for the poor performance that we observed for the square-block approach.

The reused block is not the only data item that experiences interference in the cache. Another form of interference, which we term *cross-interference*, occurs when the two vectors fetched for a single matrix-vector multiply interfere with the block or with each other. Fortunately, the impact of such cross-interference is much less severe. Recall that during a single matrix-vector multiply, the data items that are touched are the entries from the block and the entries from a pair of vectors. In

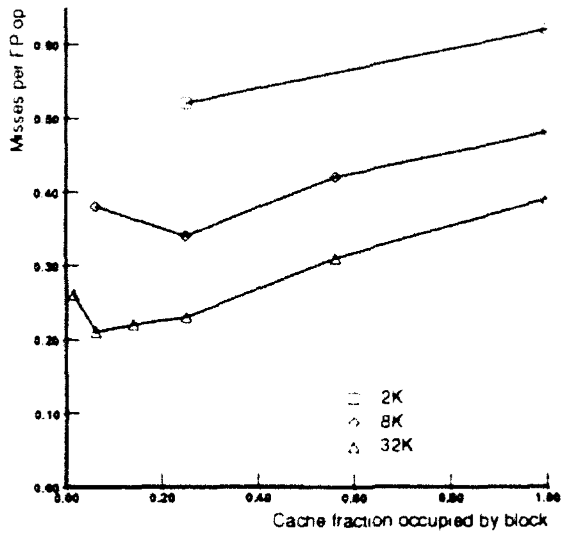


Figure 15: Cache miss behavior for multifrontal supernode-matrix method, using square blocks for matrix BCSSTK15. Cache sizes 2K, 8K, and 32K.

the case of square blocks, the block would contain C data items, where C is the size of the cache, while the vectors would each contain $\sqrt{C}$ data items. Recall that each matrix-vector multiply is assumed to cache miss on the two vectors, resulting in $2\sqrt{C}$ cache misses. If the block interferes with itself, then the matrix-vector multiply could generate C cache misses instead. On the other hand, the increase in cache misses due to cross-interference from the two vectors is limited by the size of the vectors themselves. Therefore, cross-interference increases cache misses by a small constant factor.

An obvious solution to the problem of a reused block interfering with itself in the cache is to choose a block size that is much smaller than the cache size, so that the cache mapping is not as crucial. To determine an appropriate choice for the block size, we have considered a range of different cache sizes, and a range of different block sizes for each cache size. The results for matrix BCSSTK15, using a direct-mapped cache, are shown in Figure 15. It is clear from the figure that the optimal choice of block size uses only a small fraction of the cache. Indeed the optimal choice with respect to cache misses is most likely suboptimal for overall performance. For the case of a 32 KByte data cache, the block size that minimizes cache misses is 16 by 16. In general, such a small block size would certainly lead to decreased spatial locality on a machine with long cache lines. It would also lead to small inner loops, potentially leading to increased time filling and draining pipelines (as short vectors would lead to decreased performance on vector machines).

Another possible solution to the problem of a reused block interfering with itself in the cache is to copy the block to a contiguous data area, where it is certain not to interfere with itself [19, 26]. In effect, the cache is treated as a fast local memory. The data to be reused are explicitly copied

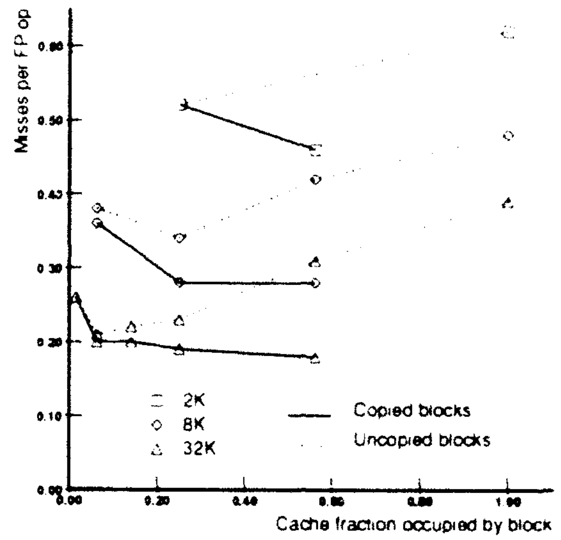


Figure 16: Cache miss behavior for multifrontal supernode-matrix method, using square blocks and copying, for matrix BCSSTK15.

into it before being used. The result of employing the copying optimization to the sparse problem (BCSSTK15) is shown in Figure 16. The solid curves in this figure show the cache miss rates for a copying code, and the dotted lines show the miss rates for the previous uncopied code. It is clear from this figure that copying leads to a significant decrease in cache misses and allows for larger block size choices. This data copying naturally has a cost, which we will investigate in the next subsection. While the cost is moderate, it is not completely negligible. We therefore briefly note that it may be advantageous to work with both copied and uncopied blocks in the same code, switching between them depending on whether or not a block would derive benefit from copying.

Before presenting performance results for square-block supernode-matrix approaches on our benchmark machines, we briefly consider the use of square blocks in supernode-supernode methods. We omit the implementation details, and simply mention that the identical considerations, including cache interference and block copying, apply. An important difference exists in the amount of copying that must be done, however. Recall that the main difference between supernode-matrix and supernode-supernode methods is that in the former, a single supernode is used once to modify the entire matrix. Since each supernode is used only once, a code that copies blocks will copy each non-zero in the matrix at most once. In supernode-supernode methods, on the other hand, a single supernode is used to modify several other supernodes. Consequently, if non-zeroes are copied, then the entries of a single supernode must be copied multiple times, once for each supernode-supernode operation in which they participate. At this point, we simply note that the cost of data copying is larger. The magnitude of this increase will be considered in the next subsection.

We now present performance numbers for square-block approaches to sparse factorization on our

Table 23: Performance of square-block uncopied methods on DECstation 3100 and IBM RS/6000 Model 320.

	Left-looking supernode-supernode		Right-looking supernode-matrix		Multifrontal supernode-matrix	
	MFLOPS		MFLOPS		MFLOPS	
Problem	DEC	IBM	DEC	IBM	DEC	IBM
LSHP3466	1.97	6.70	2.19	7.82	1.79	7.64
BCSSTK14	2.74	11.94	2.76	12.30	2.38	11.62
GRID100	2.12	7.44	2.35	8.34	1.84	8.04
DENSE750	4.03	23.70	4.04	23.66	3.94	22.15
BCSSTK23	3.24	17.26	3.21	17.05	2.97	16.43
BCSSTK15	3.43	18.46	3.38	18.33	3.17	17.34
BCSSTK18	2.86	14.20	2.90	14.41	2.54	14.39
BCSSTK16	3.21	17.15	3.16	17.23	3.00	16.68
Means:						
Small	2.23	8.17	2.41	9.12	1.97	8.79
Large	3.15	16.40	3.13	16.48	2.88	16.03
Overall	2.80	12.30	2.90	13.07	2.54	12.61

two benchmark machines. We give performance figures for a square uncopied approach in Table 23, and for a square copied approach in Table 24. These tables give performance numbers for the highest performance versions of each of the three factorization approaches. The block sizes on the DECstation 3100 are 24 by 24 for the uncopied code and 64 by 64 for the copied code. The block sizes on the IBM RS/6000 are 24 by 24 for the uncopied code and 48 by 48 for the copied code. These block sizes empirically give the fewest cache misses on the caches of these machines. We show a comparison of mean performances of square-block and panel-block schemes in Table 25. Surprisingly, both the copied and the uncopied square-block methods are slower than the panel-blocked methods on both machines. On the DECStation 3100, the square-block schemes are between 3% and 13% slower than the panel-blocked schemes. The left-looking supernode-supernode method with block copying yields the largest difference in performance. On the IBM RS/6000, the uncopied square-block code is between 5% and 12% percent slower than the panel-blocked code, and the copied code is between 8% and 21% slower. Again, the left-looking supernode-supernode code with block copying shows the largest difference in performance. We now study the performance of square-block methods in more detail in order to explain the performance on the two benchmark machines and also to predict their performance on other machines and matrices.

3.5.3 Advantages and Disadvantage of Square-Block Methods

It is clear from the results presented so far in this section that square-block methods have certain advantages and certain disadvantages relative to panel-blocked methods. On the two benchmark machines, the disadvantages outweigh the advantages. We now study where the performance differences

Table 24: Performance of square-block copied methods on DECstation 3100 and IBM RS/6000 Model 320.

	Left-looking supernode-supernode		Right-looking supernode-matrix		Multifrontal supernode-matrix	
	MFLOPS		MFLOPS		MFLOPS	
Problem	DEC	IBM	DEC	IBM	DEC	IBM
LSHP3466	1.74	5.69	2.01	6.94	1.61	6.98
BCSSTK14	2.46	10.74	2.59	11.71	2.23	11.18
GRID100	1.94	6.36	2.22	7.64	1.71	7.45
DENSE750	4.40	25.08	4.40	25.08	4.21	23.21
BCSSTK23	3.25	16.87	3.29	17.36	3.02	16.64
BCSSTK15	3.42	17.90	3.48	18.39	2.99	17.76
BCSSTK18	2.77	13.00	2.94	14.35	2.60	14.37
BCSSTK16	3.07	15.94	3.00	16.97	2.99	16.72
Means:						
Small	2.00	7.04	2.25	8.32	1.81	8.18
Large	3.06	15.34	3.12	16.39	2.85	16.15
Overall	2.66	11.10	2.83	12.48	2.44	12.20

Table 25: Percentage of panel-blocked performance achieved with square-blocked codes, on DECstation 3100 and IBM RS/6000 Model 320.

	Left-looking		Right-looking		Multifrontal	
	supernode-supernode		supernode-matrix		supernode-matrix	
Method	DEC	IBM	DEC	IBM	DEC	IBM
Small:						
Uncopied square blocks	86%	85%	90%	92%	92%	96%
Copied square blocks	77%	73%	84%	84%	84%	89%
Large:						
Uncopied square blocks	95%	89%	95%	91%	99%	92%
Copied square blocks	92%	83%	95%	90%	98%	92%
Overall:						
Uncopied square blocks	92%	88%	94%	93%	97%	95%
Copied square blocks	87%	79%	92%	89%	93%	92%

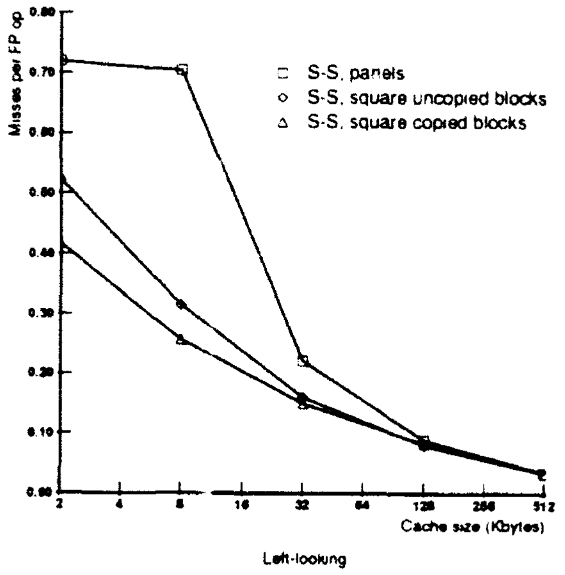
between the approaches lie, and we consider their relative importance.

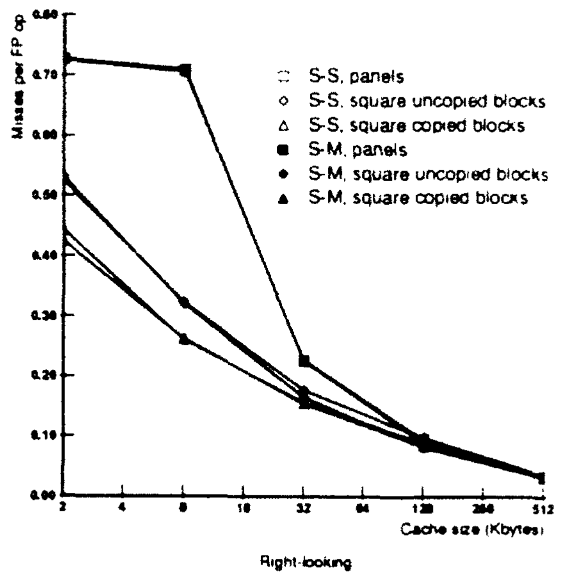
We begin by looking at the advantages of square-block approaches. Recall that the main motivation for considering square-block approaches was to improve the cache performance of sparse factorization on machines with small caches. We now show the effects of using a square-block approach for a variety of cache sizes. We present cache miss figures for supernode-supernode and supernode-matrix methods for matrix BCSSTK15 in Figure 17. The first curves in each of these graphs are the cache miss figures for the panel-oriented methods. These curves were presented in an earlier set of graphs. We have added cache miss results for square-block approaches, both with and without copying. The square-block approaches use block sizes near the empirical optimums for reducing cache misses. The curves clearly show the advantages of a square-block approach. Such an approach generates many fewer misses than a panel-blocked approach for small cache sizes.

Interestingly, the uncopied approach does not generate significantly more misses than the copied approach, even though the uncopied approach uses much smaller blocks. The small size of this difference is especially surprising because we have observed factors of two or more reductions in miss rate when using a copied approach to compute the update matrix from a single large supernode. The main reason for the small difference is that much of the sparse computation does not contain significant reuse. The advantages of the copied approach are diluted by the misses incurred in operations that do not derive an advantage from copying. Another reason is that supernodes that fit in the cache do not derive any cache benefit from copying. As the cache grows, the number of such supernodes increases.

To relate these numbers to the performance of our benchmark machines, we note that the reductions in miss rates for the square-block approaches on 32 KByte and 64 KByte caches are moderate. Furthermore, the miss rates for the panel-blocked approach at these cache sizes are extremely low, meaning that the costs of cache misses are already a small fraction of the overall cost of the factorization. Square-block methods therefore provide only a small performance advantage due to cache misses for our benchmark machines.

We now turn our attention to the disadvantages of square-blocked approaches. We first consider square copied block methods, where the most important disadvantage is the added cost of explicitly copying data to a separate data area. One observation to be made at this point is that this copying is similar to the extra data movement that is done in the multifrontal method, where supernode entries are added into the update matrix and their final values are later copied back. In fact, we found that the extra data movement in the multifrontal method resulted in lower performance when compared with methods that did not perform this extra data movement. The copying therefore has some non-trivial cost associated with it. In the case of supernode-supernode methods, more data copying is performed, and the related costs will be even larger. In order to obtain a rough idea of how large these costs are, we present in Table 26 the percent increase in memory references caused by the copying. These numbers show the increase in total references over the entire program in





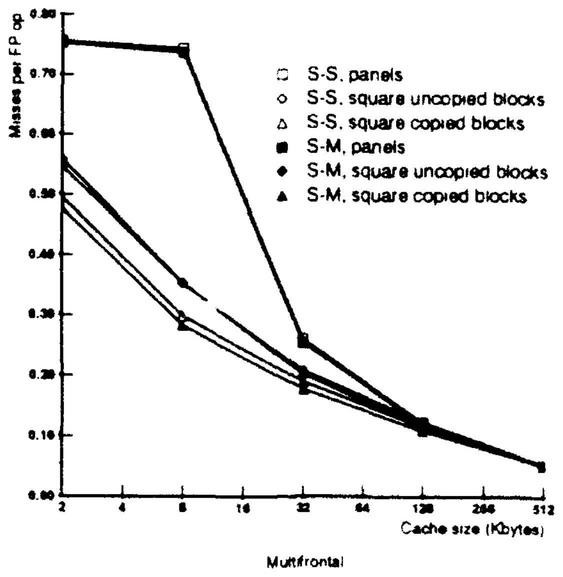


Figure 17: Cache miss behavior for various methods, matrix BCSSTK15. S-S is supernode-supernode. S-M is supernode-matrix.

Table 26: Increase in memory references due to data copying

Problem	Supernode-matrix increase	Supernode-supernode increase
LSHP3466	5.5%	9.7%
BCSSTK14	5.2%	9.8%
GRID100	4.6%	9.9%
DENSE750	2.0%	2.0%
BCSSTK23	2.3%	4.9%
BCSSTK15	2.8%	5.5%
BCSSTK18	2.6%	8.4%
BCSSTK16	3.5%	7.1%

moving from a multifrontal method that does not copy to one that does. These numbers assume that supernodes containing a single column are not copied, since any sub-blocks of such supernodes trivially can not interfere with themselves.

The numbers of Table 26 indicate that block copying incurs a moderate cost for the supernode-matrix methods, and a larger cost (two to three times higher) for the supernode-supernode methods. Regarding the trends in the cost of data copying, we have noticed that the relative costs of copying decrease as the size of the matrix increases.

Moving to the square uncopied approach, we note that the primary disadvantage of this approach comes from the smaller blocks that it must use. These smaller blocks increase the overheads associated with performing block operations and thus lead to less efficient kernels. They also result in higher miss rates than copied blocks.

The performance of square-block approaches on our benchmark machines can therefore be understood as follows. Square blocks decrease the cache miss rates slightly for our benchmark matrices and benchmark machines. However, in the case of the copied approach, the benefits of this reduction in misses are offset by the cost of the copying. The left-looking supernode-supernode method performs the most copying, and consequently it suffers the largest decrease in performance. In the case of the uncopied approach, the benefits of the reduction in misses are offset by the increase in overhead associated with working with small blocks.

We therefore find that the square-block approaches offer no advantage for the machines and matrices that we have considered. However, the reader should not conclude from this that such approaches have no advantages at all. Two important trends make it likely that square-block approaches will provide significantly higher performance than panel-block approaches in the near future. First, we expect that ever-increasing processor speeds, combined with ever-increasing memory densities, will allow workstation-class machines to solve much larger problems than are solved today. Second, we expect small on-chip caches to become more common as processors become faster and thus require faster access to cached data. These two important trends, larger problems and smaller on-chip caches, both contribute to a decrease in the number of columns that can be held in the cache, thus making panel-oriented approaches much less effective at reducing cache miss rates than square-block approaches.

3.6 Discussion

This section will briefly discuss a number of issues that have been brought up by this chapter

3.6.1 Square-Block Methods: Performance Improvement on Benchmark Machines

We begin by briefly reconsidering the performance of square-block approaches on our benchmark machines. While the previous section showed that these methods do not improve performance for the benchmark matrices that we have chosen, an unanswered question is whether they would improve performance for any matrices. As it turns out, the answer depends on the relationship between the size of the cache and the size of main memory. Cache reuse in a panel-blocked approach is limited by the length of the longest column in the matrix. Let us consider how long this column can be. Since we are interested in in-core factorization, we know that the matrix of interest must fit in main memory. We also know that if a column has length l, then the column produces an update matrix of size $l(l+1)/2$, and thus the matrix must be at least this large. Since this is a lower bound on the space required, and a dense matrix achieves this lower bound, then the sparse matrix with the longest columns that fits in a given amount of main memory is a dense matrix. If we consider our DECstation 3100, which contains 64 KBytes of cache and 16 MBytes of main memory, a simple calculation reveals that the largest dense matrix that fits in 16 MBytes is roughly 2000 by 2000. Since a 64 KByte cache fits 8192 entries, at least four columns from this dense matrix, and thus from any matrix that fits in main memory, will fit in cache. Thus, any problem that fits in main memory will achieve some degree of cache reuse on this machine.

If we consider the dense benchmark matrix (DENSE750), we note that at least 10 columns of this matrix fit in a 64K cache, and thus a panel-blocked method would use panels of that size. On the DECstation 3100, a panel-blocked method factored DENSE750 at a rate of 3.8 MFLOPS. The uncopied square-block method used a block size of 24, which would be expected to slightly increase reuse. Indeed, the uncopied method factored the matrix at roughly 4.0 MFLOPS. The copied square-block method, with a block of size 64, significantly increases the amount of reuse and factors the matrix at a rate of 4.4 MFLOPS, or 16% faster.

A dense matrix provides only a lower bound on the number of columns that fit in the cache. A truly sparse matrix would have much shorter columns, so we would expect more reuse. For example, the largest 5-point square grid problem that fits in 16 MBytes of memory is roughly 220 by 220. The longest column in this matrix contains 330 entries, meaning that at least 24 columns would fit in the DECstation 3100 cache. Thus, for a machine with 64 KBytes of cache and 16MBytes of memory, we would expect a sparse matrix that fits in main memory to achieve significant data reuse using a panel-blocked algorithm. Of course, a general-purpose factorization method should not make assumptions about the relative sizes of cache and main memory. We are simply explaining the reasons for the lack of observed improvement, and pointing out that we would need a much larger problem and much more memory to realize significant benefits from a square-block approach on a machine with such a large cache.

3.6.2 Improving Multifrontal Performance

Another question that we now consider is whether the performance disadvantage that the multifrontal method suffers relative to the other methods due to extra data movement can be overcome. Much of this extra data movement is caused by the absence of a special case for handling supernodes that contain a single column. The problem of dealing with small supernodes in the multifrontal method has been recognized in the context of vector supercomputers. One solution that has been investigated is *supernode amalgamation* [10, 17], where supernodes with similar structure are merged together to form larger supernodes. The cost of such merging is the introduction of extra non-zeroes and extra floating-point operations. The merging is done selectively, so that the costs associated with combining two supernodes are less than benefits derived from creating larger supernodes. Our observations about the lower performance of the multifrontal method on our benchmark machines indicate that amalgamation techniques have a role in sparse factorization on workstation-class machines as well. Note, however, that the potential benefits of amalgamation are not specific to the multifrontal method. The performance of the left-looking and right-looking approaches also suffers somewhat due to the existence of small supernodes. We will discuss amalgamation in more detail in later chapters. For now, we simply note that we have found that when amalgamation is performed on the matrix before the factorization, the performance gap between the multifrontal method and the other two methods is narrowed somewhat.

Another approach to improving the performance of the multifrontal method would be to use a hybrid method, as suggested in [31]. Normally, the multifrontal method traverses the elimination tree all the way down to the leaves. Briefly, a multifrontal hybrid uses a multifrontal approach above certain nodes in the elimination tree and an approach that is more efficient for small problems for the subtrees below those certain nodes. The selection of the nodes at which the hybrid method would switch approaches would depend on the relative strengths and weaknesses of the two blended approaches of the hybrid.

3.6.3 Choice of Primitives

Our study has considered a range of methods for factoring sparse matrices, including a number of methods that are obviously non-optimal. We have included such methods for a number of reasons

The first is to obtain a better understanding of the benefits obtained by moving from one approach to another. In general, the methods based on higher-level primitives are much more complicated to implement. In particular, unrolling and blocking are quite tedious and time-consuming. We wanted to understand how much benefit was derived by expending the effort required to implement them. Also, through a gradual transition from relatively inefficient methods to efficient methods, we were able to obtain an understanding of where the important sources performance improvement were.

Another reason for considering factorization primitives that are inefficient on hierarchical-memory machines is that parallel factorization methods are typically implemented in terms of them. One reason has simply been that few distributed-hierarchical-memory multiprocessors have been available in the past. Another reason is that primitives expressed in terms of larger structures in the sparse matrix limit the amount of available concurrency. We will return to this issue later in this thesis.

3.6.4 Performance Robustness

Our final point of discussion relates to the performance robustness of the methods that we have considered. While the relative performance levels of the fastest methods have been quite consistent across the different benchmark matrices on the machines that we have considered, it is quite possible that the methods have important weak points that were not brought out in the benchmark set.

The first thing to note when considering the robustness of factorization methods is that cache miss rates can play an important role in determining performance. We therefore conclude that panel-blocked approaches are not robust. They generate significantly higher miss rates than square-blocked methods for large problems or small caches, thus potentially resulting in significantly lower performance. A robust general-purpose code would employ square-blocking. A similar but less important consideration is whether blocks should be copied. A copied code gives lower miss rates for small caches, but it also gives lower performance for small problems due to the cost of performing the actual copying. As was mentioned in this chapter, a reasonable alternative is to use both approaches within the same code, switching between them on a per-supernode basis, depending on whether copying would provide significant cache miss benefits for the current supernode.

Given the above considerations, we now consider the robustness of each method. Beginning with the left-looking supernode-supernode method, we note that this method contains a certain degree of unpredictability. When supernodes are copied, this method must perform more copying than the supernode-matrix methods. While the increase for the benchmark matrices we considered was moderate, there is no guarantee that it will always be moderate. We can invent sparse matrices where supernode copying happens much more frequently.

The right-looking supernode-matrix method also contains some degree of uncertainty. This method must compute relative indices using an expensive search. While this search occurs extremely infrequently for the benchmark matrices that we have considered, again there is no guarantee that it will not occur much more frequently for other matrices. Also, the search code is likely to become even more expensive in the future, as the amount of instruction parallelism that processors can exploit increases.

The multifrontal method provides the most robust performance among the three approaches. The cost of copying data in a copied square-block approach is guaranteed to be moderate, since each matrix entry is copied at most once. The cost of computing relative indices is also moderate, since they are computed once per supernode. The performance of the multifrontal method on our benchmark machines was observed to be slightly lower than that of the other methods, but the difference was small. The multifrontal method contains some unpredictability, but it is not in the performance. Instead, the unpredictability is in the amount of space required to factor a matrix, since the size of the update matrix stack can vary widely.

3.7 Related Work and Contributions

An enormous amount of work has been done in the past on the problem of performing sparse Cholesky factorization efficiently. We now briefly comment on the contributions the work described in this chapter have made to the sparse Cholesky factorization literature.

The focus of our work in this chapter has of course been on maximizing the performance of sparse factorization on workstation-class, and on understanding the issues that are most important for obtaining high performance. Virtually all previous work that considered performance issues for sparse factorization either considered performance on vector supercomputers or it considered performance on workstation-class machines using column-oriented methods. Indeed, we believe it is fair to say that the common belief was that workstation-class machines were inherently a low-performance platform, and as such provided few opportunities for performance improvement. Indeed, column-oriented methods are still the most commonly used methods on workstation-class machines. Our work has made it clear that substantial performance improvements over these earlier methods are possible, that workstation-class machines are indeed a high-performance platform for sparse factorization, and it has described simple techniques that provide high performance on this class of machines.

We should note that some previous work on the multifrontal method [1] had considered performance on a machine with a memory hierarchy. By expressing frontal update matrix computation in terms of dense matrix operations, this work was able to block the computation for a memory hierarchy. Results were presented for an Alliant FX/8 vector mini-supercomputer with a one-level cache. However, this earlier work provided no context for interpreting the results. It was not clear whether the resulting performance was significantly higher than the performance that would have been obtained with a traditional column method. It was also not clear whether the use of a multifrontal framework was essential for obtaining good performance. In our study, by looking at several methods in a consistent framework we were able to isolate and quantify the magnitudes of the performance improvements that came from virtually all of the important design choices in a sparse factorization method.

Another important contribution of our work comes from our investigation of the effects of realistic caches. We showed that a panel-blocked approach, although conceptually appealing, has important limitations for large sparse problems. We showed that the alternative, a square-blocked approach, has limitations as well related to cache interference. We described the steps that are necessary to get around these limitations, namely block copying, and considered the complexity of these steps. Finally, we considered the effects of cache line size and cache set associativity on overall cache performance for the various methods.

3.8 Conclusions

An obvious end goal of a study like the one performed in this chapter is to arrive at a particular choice of method that yields consistently higher performance than the other choices. Unfortunately, no factorization method fit this description. Instead, a number of methods achieved roughly the same levels of performance, each with its own advantages and disadvantages. A less ambitious goal for a general-purpose factorization method is that it provide robust levels of performance, near the best performance of all other methods in almost all cases. From our discussion, it is clear that the multifrontal method best fits this description, although this method has the disadvantages that it performs more data movement than other methods, it is more complicated to implement, and its storage requirements are larger and less predictable.

The primary conclusions we draw from this chapter relate less to which method is best overall and more to what is required to achieve high performance with a sparse Cholesky factorization implementation. Our conclusions are: (1) primitives that manipulate large structures are important for achieving high performance on hierarchical-memory machines; (2) the choice of left-looking, right-looking, or multifrontal approaches is less important than the choice of the set of primitives on which to base them.

Chapter 4

Evaluation Environment for Multiprocessors

4.1 Introduction

Having established in the previous chapter that reusing data is crucial for achieving good performance from sequential machines with hierarchical memory organizations. we now turn our attention to parallel machines with similar memory system organizations. Before looking at specific algorithms. we first provide a detailed description of the machine environment in which our parallel method evaluations will be performed.

The parallel performance numbers we present will come from two parallel environments. The first is the Stanford DASH multiprocessor [27, 28], a single-address-space distributed-memory machine currently being built at Stanford. This chapter briefly describes the organization of this machine. including a discussion of the specific costs of the machine operations that will be important for parallel factorization. To obtain a better understanding of the performance of this and other real parallel machines, we also consider performance on a simulated parallel machine. This chapter will describe our simulation model, including a discussion of the factors we believe are most important in determining parallel machine performance and a discussion of the specific performance models we use to capture these factors.

4.2 The Stanford DASH multiprocessor

An important part of investigating the performance of a parallel algorithm is naturally to look at performance on a real parallel machine. This thesis presents performance numbers from the Stanford DASH multiprocessor, a 48 processor (currently) machine designed at Stanford. The most interesting aspect of this machine for our purposes is its memory system organization. Memory is physically distributed among the processors, with each processor containing some portion of the global main memory (see Figure 2 in Chapter 1). The machine provides a single-address-space (shared-memory) programming model, where any processor can access any memory location in the entire machine with an ordinary load instruction. The cost of such a load naturally depends on the physical location of the requested memory block.

The machine is organized as a set of clusters, where each cluster contains 4 processors and some portion of global main memory. Each processor is a high-performance MIPS R3000 integer unit and an R3010 floating-point co-processor. Each processor has a 64 KByte instruction cache, a 64 KByte first-level data cache, and a 256 KByte second-level data cache. All are direct-mapped. The caches can cache any memory location from the global shared memory. The processors execute at 33 MHz and are rated at 27 MIPS and 4.9 double-precision LINPACK MFLOPS.

Processors are connected to each other at two different levels. The four processors within a cluster are connected with a bus having a peak throughput of approximately 67 MBytes per second. The connection between clusters uses a two-dimensional mesh interconnection network, and the interconnect provides a peak point-to-point bandwidth of roughly 60 MBytes per second.

As mentioned earlier, the cost of a memory reference on the DASH machine depends on the physical position of the referenced memory location. If the location is available in the cache of the requesting processor, then the reference is serviced in a single cycle. If the location resides in the portion of memory that is local to the requesting processor, then the reference requires roughly 30 cycles and brings a 16-byte memory block into the cache. If the location resides in a non-local cluster, the request requires roughly 100 cycles and it again brings a 16-byte block into the cache. Note that in both cases the processors sits idle while the memory request is serviced. Other relevant costs for the DASH machine are the cost of a floating-point multiply (5 cycles) and a floating-point add (2 cycles). From these numbers, it is clear that memory reference costs play an extremely important role in determining the performance of a parallel program. To achieve maximum performance, programs should be written to reuse data in the processor cache as much as possible. When cache misses occur, those misses should ideally be serviced from the memory that is local to the processor that issued the request.

We should note that although the DASH machine provides a shared-memory programming model, our implementations will make little use of this feature. All important factorization data items (such as the columns of the matrix) will be explicitly placed in the local memories of their respective 'owner' processors. Furthermore, nearly all processor memory references that access non-local data will do so in order to copy blocks of data (messages) from the local memory of some sending processor into the local memory of the requesting processor. The only situations where we actually make use of the shared memory feature are for shared variables that are accessed sufficiently infrequently that their placement has no significant effect on performance (for example, the original A matrix).

4.3 Performance Model

While it is important to look at algorithm performance on real parallel machines, real machines also have a number of drawbacks. For one, they do not provide detailed performance information. They tell the user what level of performance is achieved, but they provide little information to allow the user to understand why this level is achieved. Real machines also do not allow machine parameters to be changed. A study of the effect of an increase in the cost of a cache miss, for example, would be difficult or impossible. Any real machine will also contain some number of performance quirks, coming either from unusual features of the machine or alternatively from design compromises. Without the ability to abstract these quirks away, the implementor may wind up designing parallel algorithms that specifically address these quirks instead of addressing more general and more fundamental bottlenecks in parallel machines.

An alternative to real machines that is often used in computer architecture studies is detailed multiprocessor simulation, where each instruction executed by each processor in the simulated machine is emulated, and the effect of that instruction on all other processors is computed. While such an approach addresses all the problems with real machines discussed above, it also possesses a crucial weakness. Such simulations typically require between one thousand and one hundred thousand times the runtime of the machine they are simulating to run the same program. Computer architecture studies that use such simulation understandably study small programs. Our interest here is on factoring relatively large matrices. The resulting simulation costs would simply be prohibitive.

The approach we take instead is to use coarse multiprocessor simulation, where the parallel factorization is expressed in terms of high-level operations (e.g., the modification of one supernode by another, or the transmission of a supernode between two processors). The simulator computes a runtime cost for these high-level operations based on a performance model that we will describe shortly. The parallel computation is simulated using a discrete-event simulation of these high-level tasks.

Our goal when modelling the parallel factorization is to capture only the most important factors that affect parallel performance. To this end, we only model two costs. The first is the cost of executing a task, which we model in terms of the number of floating-point operations performed by the task and the number of data items that the task fetches from memory. The second is the cost of moving data between processors, which we model in terms of the size of the chunk of data to be moved and the communication bandwidth available between the source and destination processors.

Note that our goal in this performance model is not to exactly model the Stanford DASH machine. We instead want to devise a general performance model that captures aspects that are expected to be common to a variety of parallel machines and critical to determining parallel performance. We use the Stanford DASH machine to estimate relevant parameters for the model, but we do so only so that these parameter choices represent realistic, achievable values.

4.3.1 Computation Costs

We model the cost of executing a factorization task in terms of two quantities: O, the cost of executing the machine instructions necessary to perform one floating-point operation, and M, the cost of loading one double-precision word of data from local memory. The cost of the whole task is simply the sum of the floating-point operation and data fetch costs. The O term is meant to capture not only the cost of the single instruction that performs the actual floating-point operation, but also the costs of any supporting integer operations, such as address calculations or loop instructions. Regarding the values we assign to O and M, we note that a floating-point multiply-add pair on a DASH processor requires roughly 7 machine instruction cycles. A cache miss on a 16-byte cache requires 30 cycles and fetches two double-precision items, giving a ratio of 4.3 between M and O. For the sake of normalization, we define the cost O to be equal one unit, and thus M is equal 4.3. We note that this ratio is quite close to the numbers we have calculated for a range of other current hierarchical-memory processors. For example, similar calculations for the Hewlett Packard HP/9000 Model 720, the IBM RS/6000 Model 320, and the Intel iPSC/860 machines give ratios of roughly 4.6, 2.4, and 2.6, respectively. We use the value $M = 4.3$ throughout this thesis.

Note that this computation cost model makes several simplifying assumptions. Probably the most obvious is the assumption that the processor blocks when a cache miss is serviced. While this is an accurate assumption for today's processors, future processors may be built with the ability to hide some of the latency of a cache miss by overlapping this latency with computation. It is our belief that such latency hiding will have only limited success. We expect that the ratio of memory reference latencies to floating-point operation costs will continue to increase in the future as it has in the past, so while it may become easier to hide latencies there will also be more latencies to hide. Thus we believe it is reasonable to assume that the processor will have to pay some cost for each cache miss.

We should also note that we are *not* claiming that it is impossible to build a memory system that can keep up with the memory fetch requests of a high-speed processor. Indeed, today's vector supercomputer memory systems could certainly fill the need. Our claim is simply that such memory systems will be much more expensive and consequently much less common than memory systems that rely on low cache miss rates to achieve good performance.

4.3.2 Cache Miss Counts

Our task cost model has so far been expressed as a simple function of the floating-point operations and cache misses required by the task. While it is trivial to predict the number of floating-point operations a task will require, predicting cache misses is significantly more difficult. This number depends on a variety of factors, including the size of the cache, the amount of data that is actually being reused, the cache line size and set associativity, and the way in which the reused data maps into the cache. Again, we attempt to keep our model as simple as possible by considering only the most important factors.

To illustrate our assumptions, let us consider the most important operation in sparse factorization, the modification of one supernode by another. This operation involves a source supernode S_s and a destination supernode S_d, with potentially different non-zero structures. Assume that S_s contains s columns, and assume that d columns in S_d are affected by the modification. Recall that the modification operation is performed in two steps. In the first, the update from S_s to S_d is computed using dense matrix operations. In the second, the resulting update is added into the appropriate locations in S_d using information about how the source and destination non-zero structures relate.

The computation of the update to S_d is performed as a dense matrix multiplication. The first matrix, call it X, contains the non-zeroes of S_s at or below the diagonal block of S_d. This is an $l \times s$ dense matrix, where l is the number of non-zeros in a column of S_s below the destination diagonal. The second matrix, Y, is the transpose of the block of non-zeroes in S_s that are adjacent to the diagonal block of S_d. This is an $s \times d$ dense matrix. The resulting Z matrix, an $l \times d$ dense matrix, is the update that is to be added into S_d.

We assume that the update computation interacts with a memory hierarchy in the following way. We assume that the Y matrix, of size $s \times d$, is loaded into the cache once. The X matrix is then read a row at a time, with that row being multiplied by Y to produce a row of Z. Thus, the X and Z matrices are each read from memory only once. As a result, the operation generates sd cache misses on Y, sl misses on X, and dl misses on Z. During this computation, $2sdl$ floating-point operations are performed. A common case that merits special attention is the case where both the source and destination contain the same number of columns ($s = d = B$). In this case, $2B^2l$ operations are performed and $2Bl + B^2 \approx 2Bl$ misses are generated, giving a ratio of 1 miss for every B operations. In order to be able to conveniently discuss the costs of such operations, we define a quantity $T_{op}(B)$ to be the average cost of a floating-point operation when performing an update operation for which the source and destination both contain B columns. Given this definition, we have

$$T_{op}(B) = O + M/B.$$

We should note that our description of the cache behavior of a supernode update computation is somewhat oversimplified. In particular, due to stride-of-access issues the update operation may not be coded in terms of rows of X producing rows of Z. We have also ignored issues of cache line size, set associativity, and cache interference. We believe, however, that careful coding of the matrix-multiply operation, including the use of data copying to avoid interference, would give results that are quite close to those predicted above.

From the cache miss rates above, it is clear that the benefits of large supernodes grow with the sizes of the supernodes. The benefits do not grow without bound, however. They are limited by the fact that the $s \times d$ matrix Y must remain in the cache across multiple uses. The maximum benefit is therefore determined by the size of the cache. To eliminate the need to include the machine cache size as a parameter in our model, we assume that some reasonably large block size is good enough to reduce cache miss costs to a negligible level while at the same time being small enough to fit in any reasonable processor cache. A block size of 32 fits these requirements quite well. A code that produces one cache miss for every 32 floating-point operations will generally be nearly as efficient as a code that produces no misses at all. The 32 by 32 block that must remain in the cache requires 8K of storage, which would fit in virtually any cache. Note that update operations that involve sets of more than 32 columns can be handled quite easily by breaking them up into 32 column chunks.

The other important part of the update computation is the addition of the update into the destination. We assume that for each entry in the update, the processor must fetch that entry plus one entry in the destination from memory. In other words, we assume that nothing is available from the cache. The reader may object that the update would be available in the cache since it was computed soon before, but our assumption is that the update is generally larger than the cache and thus does not remain there.

This thesis will use this performance model to simulate a variety of matrix operations, not all of which are directly related to the two steps of the supernode modification operation discussed above. Rather than explicitly enumerating the operations that we model and the costs we assign to them, we instead simply note that we use the same framework, where blocks are loaded into the cache and reused several times when possible, for all such operations. The majority of these operations are based on dense matrix-matrix multiplication, which lead to identical formulations. However, even the ones that are not still have very natural block interpretations.

To validate the use of this performance model, subsequent chapters will compare the predicted performance with actual performance on the DASH machine. Although the performance model is not meant to model the DASH machine exactly, we will see that the two actually produce quite similar results.

4.3.3 Communication Costs

The other important part of our performance model is interprocessor communication. Our model assumes that communication takes places in the form of 'messages'. We assume the time required for a message to move from its source to its destination is:

$$\alpha + \beta L$$

where α is the fixed cost of sending a message, β is the additional cost of each word of data in the message, and L is the length of the message. The quantity α is a measure of the overhead associated with a message, while β is a measure of the bandwidth of the interconnection network.

To obtain values for α and β, let us consider the Stanford DASH machine. This machine can move a 16-byte cache line of data (two words) between two different clusters in 100 cycles. It can perform a multiply-add in roughly 7 cycles, giving a β of $100/7 = 14$. To obtain an estimate for α, the fixed cost of initiating a message, we assume that the machine uses a software message scheme

To initiate a message transfer, the source processor would enqueue some indicator that a message is available in the input queue for the destination processor. The destination would then have to dequeue it before initiating the transfer. We assume that this exchange requires roughly 1000 cycles, giving an α of roughly 300.

Note that our estimate for β is somewhat pessimistic in the context of more general parallel machines, especially those with special-purpose hardware for handling messages (and even for the DASH architecture, as we will explain shortly). Our estimate above assumes that a block is moved one cache line at a time between processors; a more specialized message passing system would send the block in a more pipelined fashion.

Let us consider briefly where both β and α lie for parallel machines with more specialized message-passing support. One example, the Intel Touchstone DELTA multiprocessor, achieves around 40 double-precision MFLOPS per processor on programs that generate few cache misses. It provides around 25 MBytes per second of interprocessor communication bandwidth. Thus it performs a floating-point operation in roughly 25 nanoseconds and it can transfer an 8-byte word of data between processors in 320 nanoseconds, giving a β of 13. The fixed cost of sending a message is roughly 160 microseconds [46], giving an α of 6500. (Note that we are assuming contention-free messages to compute β here. We will discuss how we handle contention shortly.) The newer Intel Paragon machine provides roughly 50 MFLOPS per processor and 200 MBytes per second of bandwidth, giving a β of around 2. The fixed cost of a message is roughly 30 microseconds, giving an α of 1500. The Thinking Machines CM-5, on the other hand, provides 128 MFLOPS per processing node and 20 MBytes per second of communication bandwidth, giving a β of around 50. The fixed cost of sending a message is roughly 86 microseconds, giving an α of roughly 11000. In our performance model, we use aggressive values of $\alpha = 500$ and $\beta = 10$, since the trend in both of these is decidely downward.

Another important aspect of our communication model is that it assumes that the cost of a message is a latency cost only. Neither the source nor the destination processor sits idle when a message is being transferred. The source simply indicates that a message should be sent and continues on with other work. At some later time, determined by the latency model described above, the destination processor is notified that the message has arrived. This assumption is meant to account for the fact that most distributed-memory machines contain message coprocessors that handle the mechanics of moving the message. Note that the DASH machine, as described, does not fit this model. The destination processor must wait while 16-byte pieces of the message are moved. The DASH architecture does have the ability to hide the cost of sending a message from the involved processors through the use of a non-blocking prefetch instruction [35]. The actual DASH machine is unable to achieve the full benefits of this provision, however, due to limitations in the processors from which the DASH machine was built.

A final aspect of our communication model addresses the issue of contention in the interconnection

Table 27: Benchmark matrices

	Name	Description	Equations	Non-zeroes
1.	GRID100	5-point discretization of rectangular region	10,000	39 600
2.	GRID200	5-point discretization of rectangular region	40,000	159,200
3.	BCSSTK15	Module of an Offshore Platform	3,948	113,868
4.	BCSSTK16	Corps of Engineers Dam	4,884	285,494
5.	BCSSTK17	Corps of Engineers Dam	10,974	417,676
6.	BCSSTK18	Nuclear Power Station	11,948	137 142
7.	BCSSTK29	Nuclear Power Station	13,992	605,496

Table 28: Benchmark matrix statistics

		Floating-point	Non-zeroes
	Name	operations	in factor
1.	GRID100	15,707,205	250,835
2.	GRID200	137,480,183	1,280,743
6.	BCSSTK15	165,039,042	647,274
8.	BCSSTK16	149,105,832	736,294
8.	BCSSTK17	144,280,005	994,885
7.	BCSSTK18	140,919,771	650,777
8.	BCSSTK29	393,059,150	1,680,804

network. We have so far assumed that the message transfer time depends only on communication bandwidth. Realistically, however, this time will also depend on the amount of traffic that is globally placed on the interconnect by all processors. A saturated interconnect will clearly delay messages. Rather than attempting to incorporate some notion of interconnect contention into our latency model, we will instead look at bandwidth demands independently. The runtime of a computation will be computed using strictly contention-less latency estimates, and the bandwidth demands of that computation will then be examined to determine whether the assumption that communication was effectively contention-free was realistic.

4.4 Benchmark Matrices

We will use a somewhat different set of benchmarks to evaluate parallel factorization methods. The matrices in this new set are described in Table 27 and Table 28. The primary difference between this set and the previous one is that we have removed some of the smaller matrices and added a few larger ones. Our assumption is that parallel machines will generally be used to solve larger matrices.

Chapter 5

Parallel Panel Methods

This chapter will consider efficient sparse Cholesky factorization on parallel machines with hierarchical memory systems. We have two primary objectives in this chapter The first is to describe a parallel method that makes good use of a memory hierarchy, and the second is to investigate and understand the performance of this parallel method.

Several methods have been proposed for performing sparse factorization on parallel distributed memory machines, including the fan-out method [22], the fan-in method [8], and the multifrontal method [9, 34]. These approaches all distribute the computation by assigning columns of the matrix to processors. Unfortunately, the computational primitive for a column-wise distribution is a column-column modification, an operation that gives poor data reuse and low performance on machines with hierarchical memory systems. The parallel factorization approach we consider in this chapter is a straightforward extension of a column-oriented approach. Rather than distributing columns among processors, our approach instead distributes sets of adjacent columns from within the same supernode, which we call ***panels***. By working with sets of columns, a panel method can obtain many of the same benefits that were obtained earlier through the use of supernodes in sequential factorization. While panels can be integrated into fan-out, fan-in, and multifrontal methods, we will only study a multifrontal panel method. The multifrontal method is generally considered to give higher performance than the other two parallel methods [9]. As confirmation, we note that our panel multifrontal implementation provides higher performance than our panel fan-out and fan-in implementations.

To clear up a potential source of confusion, we note that the panels we use in this chapter differ somewhat from those of Chapter 3. Previously, the term panel referred to a contiguous set of columns that fit in the processor cache. Here, a panel is any set contiguous set of columns. It can be larger than the cache.

We begin this chapter with a brief description of the process of parallel sparse factorization including a description of the parallel multifrontal method and a description of our panel extension to this method. We then present performance results from our parallel panel multifrontal implementation. The results indicate that panels are a valuable addition to a parallel method, giving two- to three-fold performance improvements over column approaches. However, the results are also somewhat disappointing in that parallel speedups are well below linear over a supernodal sequential code.

We then turn our attention to obtaining an understanding of the achieved performance. Several factors that influence performance are described, and the impact of each of these factors on overall performance is investigated. We find that the main factor that limits parallel performance is the dearth of concurrency available in the sparse problems we consider, and indeed in virtually all sparse problems. This lack of concurrency is found to be quite constraining for moderately parallel machines. Furthermore, we find that concurrency grows extremely slowly with the problem size. As a result, large parallel machines would require enormous problems to achieve reasonable performance. We also find that while concurrency is the most severe limiting factor for performance, it is not the only factor. Communication costs, load imbalance, and task scheduling are also found to play important roles. We briefly consider ways to alleviate some of these factors, but conclude that in the absence of significantly higher concurrency any such approaches would lead to only minor performance improvements.

Finally, we use this understanding of achieved parallel performance to investigate the important question of how to choose panel sizes so as to maximize performance. The most important factors that must be traded off are the data reuse benefits that result from larger panels and the loss of concurrency that comes with these larger panels. We describe a simple method for determining a reasonable choice that balances these two considerations. We then discuss related work, and finally we present conclusions.

5.1 The Panel Multifrontal Method

Let us begin by describing the parallel panel multifrontal method. We start by describing the column-oriented multifrontal method and later discuss the modifications necessary to perform the computation using panels.

5.1.1 General Structure

The parallel multifrontal method is perhaps best explained using the domain/separator model of sparse factorization [9]. We use a simple 2-D grid problem (Figure 18) as our example. The elimination tree play an important role in parallel factorization, so we include the elimination tree of the example matrix in the figure. This subsection will present a very high-level description of the parallel factorization process, with more specific details to follow in the next subsection.

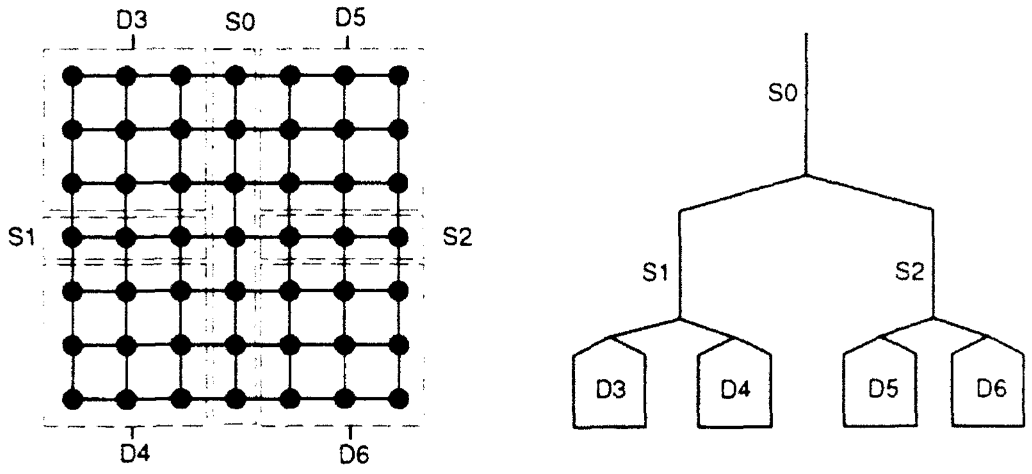


Figure 18: A simple grid example.

The process of factoring a sparse matrix on a parallel machine can be thought of as a divide-and-conquer process. The division is accomplished by finding *separators* in the graph representation of the sparse matrix. A separator is a set of nodes that breaks the graph into some number of disconnected pieces when removed. In the example, separator S0 divides the graph into two pieces, one to the left and one to the right. These pieces are referred to as *domains*. In the elimination tree, the separator nodes form a straight-line path; domains form distinct subtrees of the separator nodes. One important property of these domains is that they are computationally independent: nodes in one domain are not affected by nodes in other domains. Another important thing to note is that the separator nodes form a supernode in the sparse matrix. Thus, the terms separator and supernode are synonymous.

On a parallel machine, a separator clearly forms the basis for a division of work. Some subset of the available processors can be used to factor each domain that results from a separator. Once these domains have been factored, then the available processors can cooperate to factor the separator that formed them. Naturally, the factorization of a domain can be further sub-divided by finding subsequent separators. In the example, separators S1 and S2 divide the two domains formed by separator S0, resulting in four domains, labeled D3 through D6.

Consider the factorization of the example matrix on a four processor machine. Two processors would be assigned to each of the domains created by separator S0. Similarly, a single processor would be assigned to each of the four domains created by separators S1 and S2 (assume that processor P_i handles domain D_{i+3}). We refer to a domain with a single processor assigned to it as an *owned domain*. The computation would begin with each processor factoring their respective owned domains. Processors would then cooperate to factor the separators that formed these domains.

Thus, P_0 and P_1 would factor separator $S1$, and similarly P_2 and P_3 would factor $S2$. Finally, all processors would cooperate to factor separator $S0$.

Note that the notion of domains being divided into smaller domains by separators does not stop at the point where a domain is wholly owned by a processor. In general, any domain in the sparse matrix can be subdivided into smaller sub-domains.

Before proceeding, we should first point out two important relationships between domains, separators, and columns. First, there is a one-to-one mapping between domains and separators. A domain can be uniquely identified by the separator at its root and vice-versa. And second, a node in the graph (and thus a column in the matrix) belongs to a hierarchy of domains, but it belongs to one and only one separator.

5.1.2 Multifrontal Method

The description so far has been intentionally vague about the details of how processors actually cooperate to factor a portion of the matrix, primarily because there are several ways to perform this cooperation. We now briefly describe one way, the parallel multifrontal method [9, 34]. The multifrontal method is a column-oriented method, meaning that the computation is performed in terms of columns of the matrix, and columns are distributed among processors. The actual mechanics of the parallel multifrontal method are intimately related to the elimination tree of the sparse matrix. As was the case in the sequential multifrontal method, the most important data item in the parallel multifrontal method is an update from an entire subtree of the elimination tree to an ancestor column. The sequential and parallel methods work with these updates in very different ways, however. Recall that the sequential multifrontal method kept all updates from a subtree together in a single lower-triangular frontal update matrix. In contrast, the parallel method works with individual column updates, never actually collecting them together. In fact, two updates from the same subtree will often reside on entirely different processors.

To be more precise, consider a subtree D of the elimination tree. This subtree contains some set of columns, and these columns produce updates to ancestor columns. Recall that the update to some ancestor column j is computed by taking advantage of the elimination tree structure. Assume the domain D is divided into subsequent domains D_i by separator S. Then the update from D to j is equal the sum of the updates from D_i to j plus the updates from the columns in separator S to j.

$$update(j, D) = \sum update(j, D_i) + \sum_{k \in S} update(j, k).$$

The $update(j, D_i)$ are computed recursively in child domains. The $update(j, k)$ are computed by adding a multiples of columns k into the aggregate update. In the parallel method, responsibility for computing the update $update(j, D)$ is assigned to a particular processor. Thus, in order to compute an $update(j, D)$, the responsible processor must receive all $update(j, D_i)$ as well as all columns k in separator S.

The parallel multifrontal method performs the actual factorization in terms of domains and separators as follows. Each processor is assigned some set of matrix columns. The non-zeroes of these columns are stored in the local memory of that processor. Each processor is also assigned responsibility for computing a set of domain-column updates. For each update *update*(j, D), a count is kept of the number of *update*(j, k) and *update*(j, D_i) that must eventually be added into it, so that it will be possible to determine when a domain-column update is complete.

In performing the factorization, processors exchange two types of messages: completed column messages and completed update messages. Processors act on received messages as follows. When a processor receives a completed update *update*(j, D_i) from another processor, the receiving processor adds the update into *update*(j, D), where D is the parent domain of D_i. When a processor receives a completed column k, where k belongs to the separator at the root of some domain D, the processor locates the set of updates *update*(j, D) that it is responsible for computing. The processor computes the new contributions to these updates that are generated from the received column, and adds these contributions into the appropriate domain-column update.

One very important class of domain-column updates *update*(j, D) is those for which $j \in S$ and S is at the root of D. Clearly, once such a domain update is complete, it contains all updates that will ever be added into the destination column j. Thus, once the update is subtracted from the destination column, a *cdiv*() can be performed on that column, and the column can be broadcast to all processors that produce updates that involve it. In other words, if the column is a member of the separator at the root of domain D, then the column is broadcast to all P that are responsible for some *update*(j, D).

5.1.3 Parallel Multifrontal Example

To make the above description more concrete, let us consider the actual mechanics of a simple example. A reader who is familiar with the parallel multifrontal method may wish to skip to the next subsection.

In Figure 19 we have isolated the portions of the earlier grid example that are most relevant when the example is factored on four processors. The parallel computation would begin with each processor factoring the nodes that are internal to the owned domain assigned to it. The processors would then compute the updates from these domains to ancestor columns. Assume processor P_i owns domain D_{i+3}. Processor P_0 would compute all *update*$(j, D3)$ for $j \in \{37, 38, 39, 43, 44, 45\}$. Similarly, processor P_1 would compute all updates *update*$(j, D4)$ for $j \in \{37, 38, 39, 47, 48, 49\}$. Note that no interprocessor communication is necessary up to this point.

Now consider domain D1. This domain produces updates to nodes 37 through 39, and 43 through 49. Processors P_0 and P_1 cooperate to produce these updates. Assume that P_0 is responsible for all *update*$(j, D1)$ for even j, and P_1 is responsible for the updates with odd j. The nodes in separator $S1$ are also divided among P_0 and P_1. Assume that P_0 owns node 38, and P_1 owns 37 and 39.

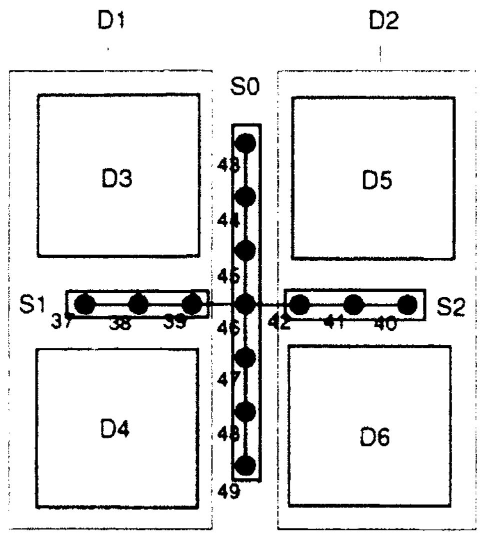


Figure 19: Grid example.

The next step in the computation involves sending the updates from domains $D3$ and $D4$ to the processors that are responsible for the corresponding updates from the parent domain $D1$. Thus, for example, P_0 would send *update*$(37, D3)$ to P_1, where it would be added into *update*$(37, D1)$. Similarly, P_1 would send *update*$(37, D4)$ to P_1 (itself), to be added into *update*$(37, D1)$. The same would occur for all other updates from $D3$ and $D4$.

Once P_1 has received the two child updates to column 37, then *update*$(37, D1)$ is complete. Since node 37 is a member of the separator at the root of $D1$, it has now received all updates that will be done to it. The update is added into column 37, a *cdiv*(37) is performed, and then column 37 is sent to all processors that produce updates from domain $D1$. This is just P_0 and P_1. When processors P_0 and P_1 receive column 37, they use it to modify all *update*$(j, D1)$ with even and odd j, respectively. One such update on P_0 is *update*$(38, D1)$, which is now complete. After adding the update into column 38 and performing a *cdiv*(38), processor P_0 then sends column 38 to P_0 and P_1. The result will another set of updates and the completion of column 39. Column 39 is then sent to P_0 and P_1, producing updates. At this point, all updates from $D1$ are complete. Of course, while P_0 and P_1 were computing updates from $D1$, P_2 and P_3 were simultaneously computing updates from $D2$.

The final step in the computation is the computation of $S0$. We can assume these nodes and the corresponding updates are distributed evenly among all four processors. The computation of $S0$ would begin when *update*$(43, D1)$ and *update*$(43, D2)$ arrive at the owner of *update*$(43, D0)$, at which point column 43 would be complete and it would be sent to all four processors. The mechanics of the remainder of the computation are hopefully now clear to the reader.

5.1.4 Implementation Details

Several implementation details are important for the parallel multifrontal method, most having to do with the non-zero structures of the columns and the domain-column updates. These details are important when considering the means by which domain-column updates are actually computed.

A domain-column update *update*(j, D) in the multifrontal method keeps the same non-zero structure as the columns in the separator at the root of D. Recall that the separator columns form a supernode and thus all share the same non-zero structure. The benefit of such an update storage scheme is that the update from a column $k \in S$ to some ancestor column j can be added into the appropriate *update*(j, D) without considering sparsity structures. Since both k and *update*(j, D) have identical structures, the update can be added directly into *update*(j, D) using a dense DAXPY operation. Since most of the work in the computation involves adding column updates into aggregate domain updates, the vast majority of the parallel computation is performed as dense DAXPY operations.

The point when non-zero structures become important is when an update *update*(j, D_i) is added into its parent update, *update*(j, D). The two updates generally have different sparsity structures (with the parent structure being a superset of the child structure), so some non-zero matching must be performed. This matching can be done by computing a set of ***relative indices*** [7, 42], indicating where in the destination a particular source non-zero can be found. Given such indices, it is a simple matter to scatter the update into its parent. An interesting thing to note about these relative indices is that they depend only on the sparsity structures of D_i and D, and are independent of the destination column j. Thus, a single set of relative indices suffices to add all updates from domain D_i into corresponding updates from domain D.

5.1.5 Distributing the Matrix Among Processors

One issue that has not yet been discussed is that of mapping columns and domain-column updates to processors. The mapping naturally has important implications, affecting both the quality of the computational load balance and the volume of interprocessor communication. This mapping actually has two components. The first relates to how processors are divided among domains. We use the ***proportional mapping*** scheme of Pothen and Sun [37] for this task. This strategy is discussed in more detail in the next subsection. For now, we simply assume that each domain D has some set of processors $P(D)$ assigned to it. The second question, which we now briefly discuss, is how individual columns and domain-column updates within a domain are assigned to processors, given an assignment of processors to domains.

Since all processors assigned to a domain D cooperate to factor the separator S at the root of D, it is natural to distribute the columns in S in a round-robin manner among the processors in $P(D)$. More precisely, we map columns to processors by working up from all leafs of the elimination tree simultaneously, assigning column k in separator S at the root of domain D to the processor in $P(D)$

that has least recently received a column. This strategy achieves a round-robin distribution within a single separator S, and also avoids processor mapping glitches when moving from a separator to its parent separator. The updates $update(j, D)$ from D to ancestor columns are distributed in a round-robin manner among the processors in $P(D)$ as well.

An obvious question at this point is why this complicated mapping strategy is preferable to one that, for example, allows any column to be mapped to any processor. The main benefit of this stricter mapping approach comes in the form of reduced communication volume [9]. Recall that a column k in the root separator of domain D must be broadcast to all processors that produce updates from D. If these updates are computed by a subset $P(D)$ of the whole processor set, then the broadcast can be limited to the set $P(D)$ as well.

5.1.6 Proportional Mapping

When a separator divides a domain D into a number of child domains D_i, the processor set assigned to D must be split among the child domains D_i. An obvious goal is to have the child domains complete at roughly the same time. Thus, the processors should be divided among the domains in proportion to the relative amounts of work required by the domains. In some cases, the decision is simple. If a separator produces two child domains, each of which requires an equal amount of work, and if the number of processors assigned to the original domain is divisible by 2, then clearly half of the processors should be assigned to each child domain. However, with arbitrary sparse matrices and unrestricted numbers of processors, the division task becomes much more difficult. We use a mapping scheme called proportional mapping developed by Pothen and Sun [37] to handle this problem.

The basis of the proportional mapping scheme is quite simple. If one child domain requires $x\%$ of the total amount of work of all child domains, then $x\%$ of the processors should be assigned to work on that domain. Of course, this may not correspond to an integral number of processors. Pothen and Sun give simple heuristics for making reasonable division choices. The details are not relevant to our presentation, so we refer the reader to [37] for a precise description.

5.1.7 Panels

Having described the parallel column multifrontal method, we now discuss modifications to this method that improve data reuse. Recall that the vast majority of the actual computation performed by the multifrontal method involves DAXPY operations. As we have mentioned earlier, such operations are inefficient on machines with memory hierarchies because they provide little opportunity to reuse data.

It is clear that data reuse can be increased in sparse factorization by manipulating sets of contiguous columns with identical non-zero structures as a group. Our approach divides each separator that is not in an owned domain into a set of *panels*, where a panel is a set of contiguous columns

We use a single target panel size throughout the entire matrix. That is, separators are split into panels that are as close as possible to the global target panel size.

This change in data distribution requires surprisingly few changes to the column parallel code. Naturally, the computation is now expressed in terms of panel updates, with processors exchanging panels among each other in order to compute domain-panel updates. The most important operation in the computation becomes the computation of a panel-panel update, which we refer to as a *pmod()* operation. This operation can still be performed without regard for sparsity structures, in this case as a dense matrix-matrix multiplication. The mechanics of computing the update are identical to those of a sequential supernode-supernode update operation. When all updates have been received by a panel, then a *pdiv()* operation is performed. This operation is identical to the *Complete()* operation in a sequential supernode-supernode method. The other important operation is the addition of a domain-panel update *update(J, D)* into a parent domain-panel. The primary difference in the case of a panel approach is that the update can be sparse along two dimensions. It can affect a subset of the rows in the parent update, and it can affect a subset of the columns. Both types of sparsity are easily accommodated.

Recall that the amount of data reuse that can be obtained in a supernode-supernode update operation, or equivalently in a *pmod()*, is determined by the width of the involved supernodes (or panels, in this case). In particular, recall that a panel size of B results in roughly one cache miss for every B floating-point operations, according to our performance model. It is therefore desirable to make the panels as large as possible.

5.1.8 Supernode Amalgamation

A panel method clearly relies heavily on the presence of large supernodes in the sparse matrix to group substantial numbers of columns into panels. While most supernodes will typically be quite large, any sparse matrix will also contain small supernodes. To increase opportunities for our method to collect columns together into panels, we perform ***supernode amalgamation*** [10, 17] as a first step in the factorization. Amalgamation merges supernodes with similar non-zero structures to produce larger supernodes. The merging is accomplished by relaxing the restriction that a sparse matrix only contain non-zeroes. Zeroes are introduced as non-zeroes in order to give two supernodes identical structures.

Two important issues for supernode amalgamation are the selection of amalgamation candidates and the criteria for deciding whether a pair of candidates should actually be merged. Regarding the selection of amalgamation candidates, the most logical choice is to consider merging a supernode into its parent in the elimination tree. We consider all such pairings. To evaluate a particular amalgamation candidate, we use our machine performance model. We compare the modelled cost of performing all updates from the two individual supernodes with the cost of performing the updates from the larger supernode that would result after amalgamation. Amalgamation is performed if the

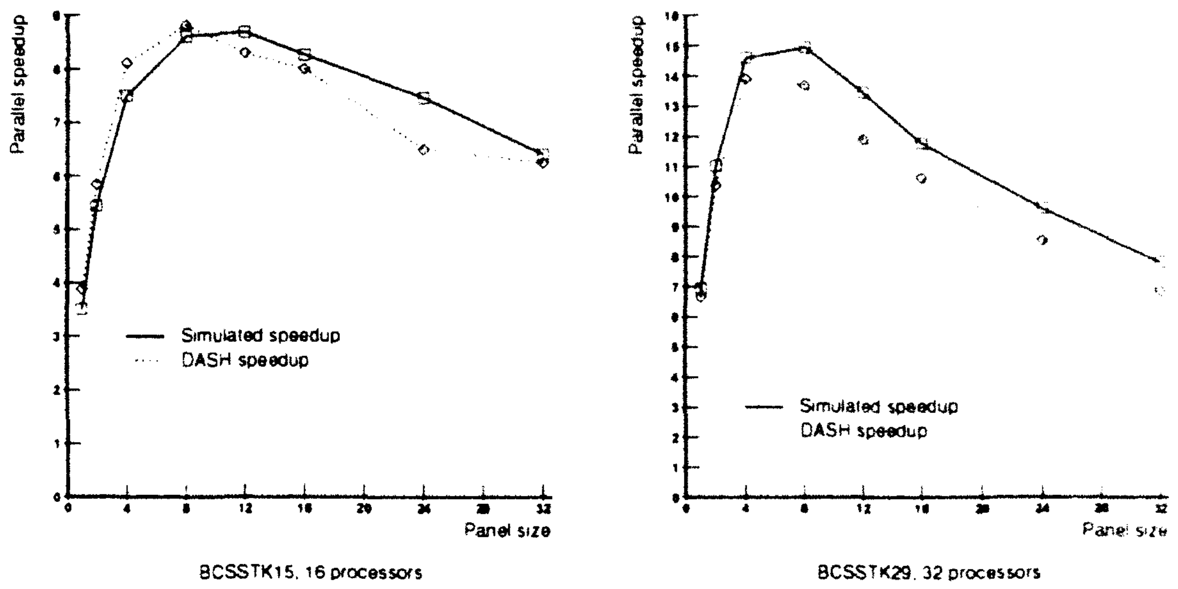


Figure 20: Parallel speedups for two sparse matrices.

latter cost is smaller. The empirical results presented in this chapter will all assume that supernode amalgamation has been performed prior to the factorization. The cost of this amalgamation is actually quite modest. We have found that it increases the runtime of the symbolic factorization phase by less than 10%.

5.2 Parallel Performance

Having described the parallel panel multifrontal method, we now look at the performance of our implementation. Figure 20 shows speedups for the parallel method, compared with a left-looking supernode-supernode sequential code. The curves on the left show speedups for matrix BCSSTK15 as a function of panel size, when factored on 16 processors. The curves on the right show similar data for matrix BCSSTK29 when factored on 32 processors. These two performance curves are representative of the behavior we have observed for a wide range of matrices and machine sizes. We will present performance results for other matrices later in this chapter. Note that these curves show both simulated speedups and speedups from the Stanford DASH machine. For reference, the DASH machine achieves roughly 8 MFLOPS for a sequential code.

An important thing to note from this figure is that our parallel performance model gives relatively accurate performance predictions, even though the model attempts to capture only the most basic aspects of parallel performance. Modelled performance differs somewhat from achieved performance, especially for BCSSTK29 on 32 processors, but even then the curves have the same general shape. We believe the observed differences are primarily due to the assumption in the performance model that interprocessor communication costs can be hidden from the processors. Since the DASH machine does not hide these costs, and since communication volume is higher for the larger machine BCSSTK29 on 32 processors exhibits the larger difference between modelled and actual performance. Overall, however, we believe it is reasonable to expect conclusions about modelled performance to apply to real performance as well.

Another important thing to note from this figure is that the panel size has a significant performance impact. For example, a panel size of 8 gives nearly 3 times the performance of panel size of 1 (a column method) for BCSSTK15. Similarly, a panel size of 4 roughly doubles performance for BCSSTK29. Another interesting thing to note is that the best panel size is not constant. Performance increases significantly when the panel size is increased from 4 to 8 for BCSSTK15. For BCSSTK29, however, performance decreases somewhat. From this and the previous observation, it is safe to say that the choice of panel size is an important issue for maximizing performance in a panel method. We will return to this issue later in this chapter.

5.3 Performance Bounds

At this point, it is natural to further investigate the performance of the panel multifrontal method. The issues we wish to understand are (1) what factors determine panel method performance and (2) to what extent can this performance be improved. It is our belief that the best way to understand parallel performance is to compare achieved performance with simple performance upper bounds. Within such a context, performance can be interpreted in two parts. First, one can look at how close achieved performance is to the upper bounds and consider the reasons for any observed differences. Second, one can look at the upper bounds themselves and consider the reasons why they are less than perfect. This is the general approach that is taken in this section. We begin by describing two simple but important upper bounds on parallel performance, the maximum load and critical path bounds.

5.3.1 Maximum Load: Load Balance and Load Efficiency

One obvious factor that bounds the performance of a parallel computation is the computational load assigned to each processor. Specifically, the computation cannot complete in less time than the maximum of the times taken by the individual processors to perform the tasks assigned to them, ignoring all dependencies between tasks. This bound is typically thought of as a *load balance* bound, with the underlying assumption that performance is less than ideal because some processors receive a larger fraction of the global work pool than others. In the case of a panel method, however less-than-ideal performance is also caused by changes in the size of the global pool. While one potential measure of the size of this pool, the number of floating-point operations in a panel-oriented factorization clearly remains fixed, the true measure, the number of operations multiplied by the cost of each operation, actually changes with the panel size. The maximum load assigned to a

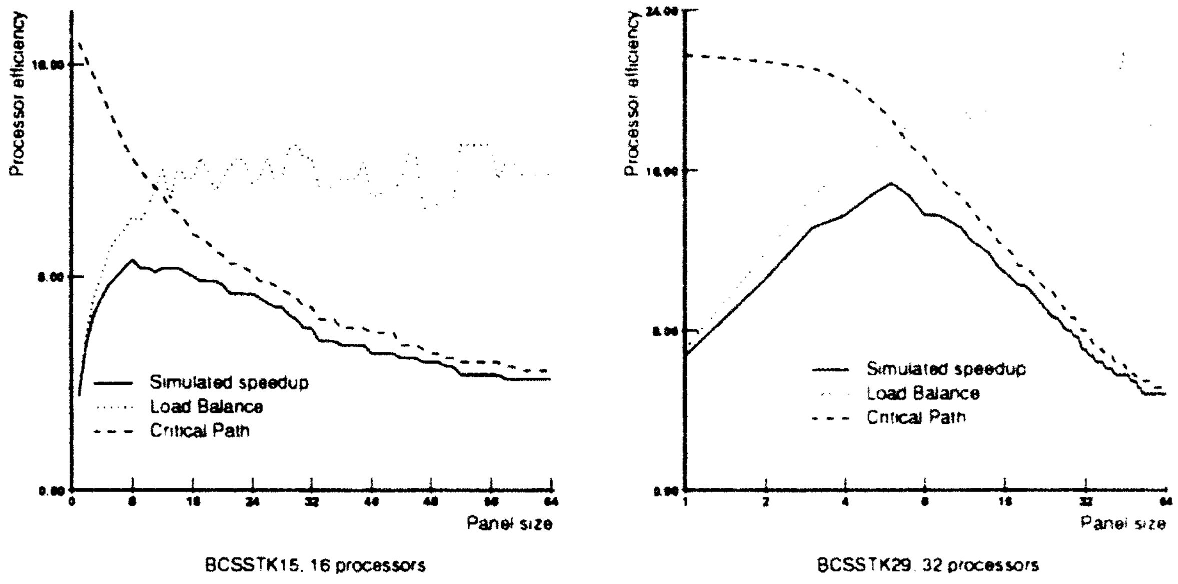


Figure 21: Parallel speedups for two sparse matrices, versus performance upper bounds

processor thus also depends on the *load efficiency*, or the time required to perform each operation. The maximum load performance upper bound is easily determined by computing the modelled costs of all tasks assigned to each processor.

5.3.2 Critical Path

Another important performance bound in a parallel computation comes from the critical path. This bound simply states that the computation cannot complete in less time than the time required for the longest chain of dependent tasks in the computation. In the case of sparse factorization, every path from a leaf in the elimination tree to the root of the tree forms a dependent chain, with each step on the chain involving a *pmod*() of a parent panel by its child, the communication of the resulting update from the child processor to the parent processor, the addition of that update into the parent, a *pdiv*() operation on the parent, a *pmod*() from the parent to its parent, and so on. Again, this bound is easily computed using our performance model. We will discuss both of these bounds in more detail shortly.

5.3.3 Performance Compared to Bounds

Figure 21 compares simulated parallel speedups with the two upper bounds from above. As can be seen from the figure, parallel performance is well predicted by these bounds. For small panels, performance is nearly equal the maximum load bound. The main limiting factor is the limited reuse of data and thus poor load efficiency. As the panel size increases, opportunities for data reuse increase and performance improves. However, at some point the critical path becomes constraining

Processors are forced to sit idle because there is a limited amount of work that can be performed in parallel. Performance flattens and eventually begins to decrease. We note that while this figure presents results for only two matrices, a number of other matrices and machine sizes that we examined yielded similar results.

While we have touched on some of the more general issues that determine parallel performance, it is important to consider these factors in more detail and also to consider the broader spectrum of issues that limit parallel performance. This subsection will now undertake a more systematic investigation of the factors that determine parallel performance. We perform detailed investigations of the maximum load upper bound, the critical path upper bound, task scheduling issues, and interprocessor communication volumes. We do this to determine the individual importance of the various limiting factors for the two example matrices and also to consider how these factors affect performance for other matrices and other parallel machine sizes.

Maximum Load

As mentioned earlier, one important factor in determining parallel performance is the maximum load assigned to any processor. Let us now look at this bound in terms of its individual components.

One factor that plays a role in the maximum load is the quality of the load balance. The load balance is primarily determined by the mapping strategy. Recall that we use a proportional mapping strategy, which performs a recursive assignment of subtrees of the elimination tree to processor subsets. This strategy clearly leads to some amount of imbalance, since an ideal division will not necessarily correspond to an integral number of processors being assigned to a subtree.

The load imbalance that results from this mapping strategy can be better quantified by comparing the maximum amount of work assigned to a processor by this strategy with the amount of work that would ideally be assigned to a processor (which is simply total work divided by P). We compute the former by assuming that for each domain D in the sparse matrix, the work required to compute updates from panels in the separator S at the root of D is distributed perfectly among the processors $P(D)$ assigned to that domain. For the two example factorizations, BCSSTK15 on 16 processors and BCSSTK29 on 32 processors, the processor that receives the maximum amount of work gets 27% and 49% more work than the ideal, respectively. Best-case processor utilizations for these two problems are thus 79% (1/1.27) and 67% (1/1.49), respectively.

To provide a broader picture of the effect of load balance, Figure 22 shows maximum processor utilization numbers due to load imbalances for a variety of matrices and machine sizes. While the data points are sufficiently chaotic in this figure that it would be difficult to give a precise description of the effect of load imbalance, it appears safe to say that load imbalances limit processor utilization levels to between 70% and 85% of what they might be with perfect load distribution.

We should note that even these estimates are optimistic, since they assume that the work from a particular separator is distributed perfectly evenly among the processors assigned to that separator

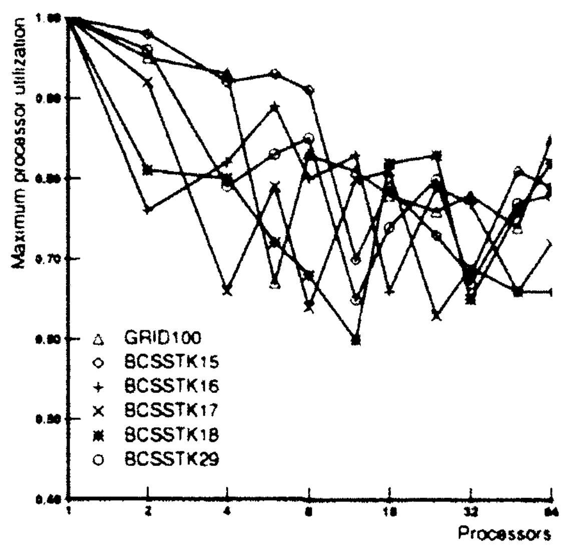


Figure 22: Maximum processor utilizations when considering load imbalance alone

In reality, a separator may produce some small number of updates (especially with large panels) that must be distributed among some large number of processors. An obvious question here is how the actual work distributions that result from real assignments of panels and panel updates to processors compare with the optimistic estimates above. We have found that the standard round-robin approach, where the panels and panel updates at the root of a domain D are assigned round-robin to the processors in $P(D)$, yields a distribution that is almost identical to the optimistic distribution over a range of panel sizes. With both BCSSTK15 on 16 processors and BCSSTK29 on 32 processors, for example, any panel size choice between 1 and 32 gives a work distribution that is within 10% of the simple prediction. Panels larger than 32 naturally result in worse work distributions, but more important constraints than work distribution would come into play in such cases.

When considering how load imbalance would affect performance over a wider range of matrices and machine sizes, an intuitive analysis of the proportional mapping approach indicates that imbalance will represent a constant-factor drag on performance in both the best and the worst case.

The other important component of the maximum load bound is load efficiency. This efficiency is determined primarily by the panel size. With a panel size of B, most of the computation involves the modification of one panel of width B by another. A larger panel size B thus leads to more data reuse and a more efficient computation.

Of course, some portions of the computation are unaffected by the panel size choice since only supernodes that are not in owned domains are split into panels. As it turns out, supernodes within owned domains play a relatively small role in the factorization. Figure 23 shows the fraction of all

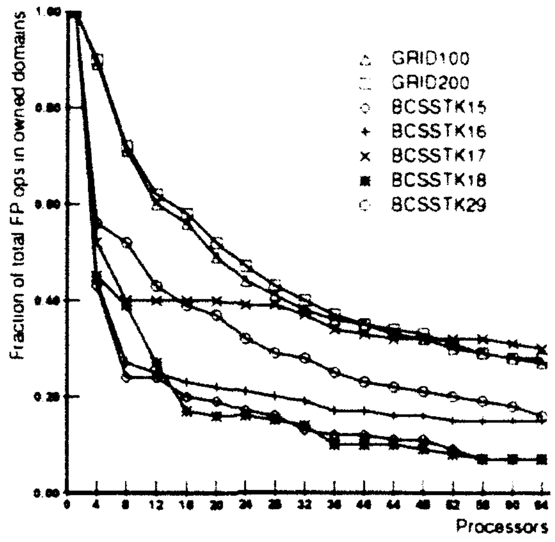


Figure 23: Fraction of all floating-point operations performed within owned domains.

floating-point operations performed within owned domains for several matrices. For any substantial number of processors the majority of the computation occurs outside owned domains. The load efficiency is thus heavily dependent on the panel size.

A simple combination of these two individual effects, load balance and load efficiency, gives the overall maximum load upper bound. For example, if load efficiency limits parallel performance to 75% of ideal, and the load balance is such that parallel speedups are at most 80% of ideal, then the overall maximum load bound would constrain performance to 60% of ideal.

Critical Path

The other important upper bound on performance is the critical path. Clearly this path is affected by the panel size as well. In this case, however, the relationship is not immediately obvious. We now consider this relationship in more detail.

To help simplify our discussion of the critical path upper bound, we make use of *critical sub-paths*. The length of a critical sub-path from a panel J is the shortest amount of time that must elapse between the time panel J has received all panel modifications and the time the entire factorization can be completed. The length of a CSP can be computed quite easily using the following recursive expression:

$$CSP(J) = T(pdiv(J)) + T(pmod(J, parent(J))) + T(sendupdate(J, parent(J))) + CSP(parent(J))$$

In other words, if panel J has not yet been completed, then the whole computation cannot complete until a $pdiv(J)$ is performed, an update is computed from J to its parent, and the update is sent to the parent. At this point, the best-case completion time is determined by the critical sub-path of the parent panel.

The critical sub-path $CSP(J)$ can be computed for all J by a simple top-down traversal of the elimination tree. The maximum value overall determines the critical path for the whole computation. Since $CSP(J)$ is always larger than $CSP(parent(J))$, the critical path always begins with a leaf in the elimination tree.

When owned domains are introduced, the notion of a task changes somewhat. An owned domain encapsulates several panel tasks into a single larger domain task. We assume that a processor handles all owned domains assigned to it, both factoring the columns within the domain and computing domain-panel updates from the domain to all affected panels, before moving on to separator tasks. Under this assumption, it is quite straightforward to assign a completion time estimate $CT(D_i)$ to each owned domain. Each domain would then impose the following constraint on parallel runtime

$$\text{runtime} >= CT(D_i) + T(sendupdate(D_i, parent(D_i))) + CSP(parent(D_i)).$$

Since domains occupy the leaf positions in the elimination tree, the true critical path for the whole computation begins with a domain (if a separator has no domains as children, then assume the separator is the parent of an empty domain). Thus, the critical path is the maximum over all domains of the above domain runtime bound.

While the above expression allows us to determine the length of the critical path given a panel size B, it unfortunately says nothing about how the path length changes with B. To understand the effect of the panel size, let us define $CSP(J, B)$ to be the length of the critical sub-path from J with a panel size of B. From before we have:

$$\begin{aligned} CSP(J, B) &= T(pdiv(J)) + T(pmod(J, parent(J))) + T(sendupdate(J, parent(J))) + \\ &\quad CSP(parent(J), B). \end{aligned}$$

Simple computations reveal that the runtime costs of these operations are:

$$CSP(J, B) = B^2 L_J T_{op}(B) + 2B^2 L_J T_o p(B) + BL_J\beta + CSP(J', B),$$

where L_J is the length of the first column in panel J, and J' is the parent panel of J in the elimination tree. Recall that $T_{op}(B)$ and β were defined in Chapter 4 to be the average cost of a floating-point operation and the time to communicate one word of data, respectively. Note that several lower-order terms have been dropped.

Now let us compare the length of this path to the length of the path from the first column in J to the first column in J' when using a panel size of 1.

$$CSP(J, 1) = BL_J T_{op}(1) + 2BL_J T_o p(1) + BL_J\beta + CSP(J', 1).$$

These two sub-path expressions bear a simple relationship to each other. If the critical sub-path expression is broken into a computation term and a communication term, we find that

$$CSP(J, B) = B\frac{T_{op}(B)}{T_{op}(1)}CSP_{comp}(J, 1) + CSP_{comm}(J, 1)$$

Thus, the path length for a panel size of B can easily be estimated from the computation and communication components of the critical path for a panel size of 1. The amount of computation on the critical path increases roughly linearly in the panel size, while the amount of communication remains constant. The owned domain at the bottom of the critical path is unaffected by a change in panel size, which mitigates the effects of an increased panel size somewhat, but we note that the work within a domain will typically be a small part of the path.

We observed in the earlier examples that the critical path limited parallel performance. The path was too long to allow for a large panel size, thus forcing a tradeoff between the efficiency of the individual processors and the number of processors that could effectively cooperate. To obtain a broader feel for the importance of this critical path bound, let us consider the length of the critical path for a range of matrices. It is actually somewhat easier to think about the critical path as it affects concurrency, the maximum parallel speedup that can be obtained for a problem. As we mentioned earlier, concurrency is computed by dividing the sequential runtime of the computation by the length of the critical path.

To simplify the analysis somewhat, let us first consider dense factorization. A simple computation reveals that concurrency for a dense $n \times n$ matrix, using a panel size of B, is:

$$\frac{n}{\frac{9}{2}BT_{op}(B) + \frac{3}{2}B}$$

In other words, the maximum speedup and thus the maximum number of processors that can be used for an $n \times n$ problem is proportional to n/B. This is not at all surprising, since columns are being distributed among processors and there are only n columns in the matrix. Recall that the amount of work performed in dense factorization is $n^3/3$. Thus the amount of work in the problem grows much more quickly than the number of processors that can be used to perform that work. A factor of two increase in concurrency requires a factor of eight increase in work. From this disparity in growth rates, one can conclude that large parallel machines will require enormous problems. Equivalently, one can conclude that concurrency will be quite limited for reasonable problem sizes.

Of course our interest in this chapter is not on studying dense problems, but rather sparse problems. We find that sparse matrices suffer from the identical scalability problems. Specifically, when normalized to do the same number of floating-point operations, 2-D sparse grid problems only expose roughly 3 times more concurrency than dense problems; 3-D grid problems expose less than two times more. To give some idea of how much concurrency is available in less regular sparse problems, Figure 24 plots available concurrency against floating-point operations for a variety of matrices from the Boeing/Harwell sparse matrix test set. This plot shows maximum possible

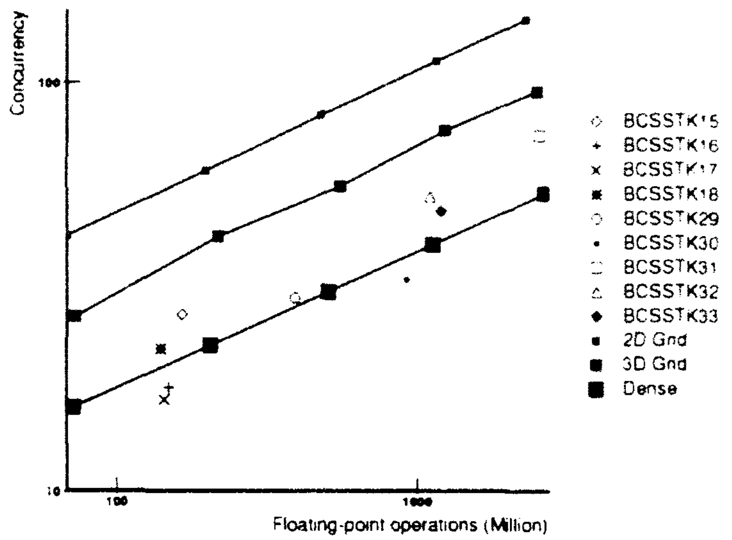


Figure 24: Concurrency in sparse problems.

speedups for these matrices, due to their critical paths, under our parallel performance model[1]. The less regular sparse problems can be seen to contain comparable amounts of concurrency to the dense, 2-D grid, and 3-D grid problems. Indeed, the growth rates appear quite similar. We therefore expect to see the same sorts of concurrency problems for sparse problems that we described for dense problems.

To put these growth rates in better perspective, let us consider a single example. Matrix BCSSTK33 requires roughly 1.2 billion floating-point operations to factor, and it allows a maximum parallel speedup of roughly 50. This matrix is much larger than those typically considered in parallel sparse factorization studies, yet it can only make good use of relatively few processors. Keep in mind that this 50-fold speedup bound is an optimistic upper bound. Now consider the case where we want a problem with a 100-fold speedup bound instead. That problem would require roughly 8 times as many floating-point operations, or roughly 10 billion. It is clear that the problem sizes quickly overwhelm the resources that can be brought to bear on them.

When considering the panel sizes that would be appropriate for the parallel factorization of sparse problems, one thing that is clear is that a large panel size would cause a significant reduction in concurrency, a reduction that most problems simply could not afford on all but the smallest of parallel machines.

[1]Note that these concurrency figures are heavily dependent on $T_{op}(1)$ and β. Lower values of these machine parameters would produce higher concurrency numbers.

Task Scheduling

Another important issue when considering the performance of the parallel computation is the fact that achieved performance is below both the maximum load and critical path upper bounds at points where the two are nearly equal (see Figure 21), a disparity that we loosely attribute to task scheduling issues. What we mean by task scheduling is simply that some processors sit idle during the course of the computation not because there are no tasks to be performed, but instead because tasks are not available when those processors are free to perform them.

Note that achieving performance equal to the upper bounds at all times would require an extremely good schedule. Consider, for example, the point where the maximum load and critical path upper bounds are equal. To achieve performance equal to the maximum load upper bound at this point, the processor with the most work assigned to it would have to be executing tasks continuously. To achieve performance equal the critical path upper bound as well, that processor would also have to execute tasks on the critical path as soon as they are ready, an unlikely prospect if the processor is always executing some task. It is thus understandable that performance is below the upper bounds.

Overall, we have found that at the panel size where the upper bounds are least constraining, achieved parallel performance is 15% to 35% below the bounds. In other words, scheduling issues play an important role in limiting performance. Note that scheduling issues would be much less important if there were an abundance of available concurrency. With more concurrency, processors would be much less likely to be without a task to execute. Unfortunately, as we discussed earlier, concurrency will generally be in extremely short supply. We have generally found that any excess concurrency is better spent on larger panels rather than on 'slack' to improve the task schedule.

Communication Volume

Our assumption so far has been that the time required for an interprocessor message depends only on the size of the message and the communication bandwidth available between the source and destination processors. Clearly this assumption is only valid if the message does not experience significant contention on the interconnect. Let us briefly consider the volume of communication placed on the interconnect by this computation to obtain some feel for the amount of contention that might arise.

Interprocessor communication for a panel method can perhaps best be understood by looking at how communication volume and computation volume grow with larger problems and larger parallel machines. Communication volume for a panel multifrontal method can be shown to grow as $O(n^2P)$ for an $n \times n$ dense problem, and as $O(k^2P)$ for a $k \times k$ 2D grid problem [24]. The computation required for these problems grows as $O(n^3)$ and $O(k^3)$, respectively. Thus, in both cases a panel method would require $O(P/n)$ words of communication for every floating-point operation, where n is a measure of the problem size. Note that communication volume is independent of the panel size.

Now consider a factorization problem that produces a manageable amount of communication

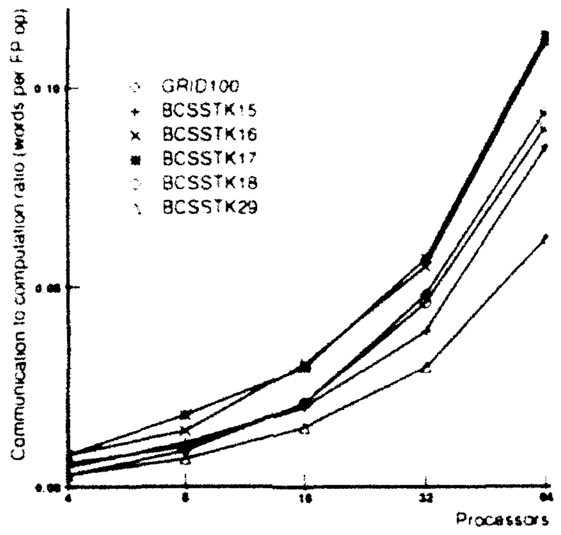


Figure 25: Communication to computation ratios for sparse problems

when using P processors. In order to increase the number of processors without overwhelming the processor interconnect, the amount of communication per floating-point operation should remain constant. The communication to computation ratio is $O(P/n)$, so consequently n must grow linearly with P. Recall that this growth rate is identical to the rate required to expose sufficient concurrency for P processors, and that this rate was considered excessive. Thus, communication volume is also a crucial limiting factor.

We should note that keeping the communication to computation ratio constant may not be sufficient to keep the processor interconnect from saturating. The problem here is that each word of interprocessor communication may become more and more expensive as the machine size grows, since it may have to traverse more and more links on the interconnection network. In fact, it can be shown that a constant ratio is inadequate when using a panel method that sends point-to-point messages on a machine with a mesh interconnect. This problem can be overcome in the multifrontal method since it relies on multicast messages, where the identical message is sent to several processors. Such multicasts can be implemented to make more efficient use of the processor interconnect.

While growth rates are interesting for understanding asymptotic behavior, it is also important to consider the 'constants' for realistic problems. Figure 25 shows communication volume figures for several sparse problems across a range of machine sizes. The figure plots the ratio of total words of communication (8-byte words) to total floating-point operations. Note that communication volume growth rates are somewhat faster than linear in the number of processors for small machines. They level off to roughly linear for the larger machines. Regarding the question of what communication ratios are sustainable on real machines, this will of course depend on the specific parameters of the machine. On today's machines, a ratio of 0.025 (40 FP ops per word of communication) would most likely be sustainable. When comparing these numbers to machine communication/computation ratios, keep in mind that the computation is only achieving roughly 50% processor utilization. Ratios of 0.05 (20 FP ops per word) or more would be difficult to sustain.

Summary

In summary, a panel method faces a number of rather formidable performance obstacles. In order to achieve high processor utilization levels, the method would require

- an extremely effective subtree-to-processor-subset mapping to keep load imbalances low;
- a large panel size to keep load efficiencies high;
- an abundance of concurrency, so that the panel size can be made large to increase load efficiency and also so that task scheduling issues would be unimportant; and
- sufficiently high interconnect bandwidth that the interconnect does not saturate.

For the examples we have considered here, with matrices that require between 100 million and 1 billion floating-point operations to factor and machines with 16 to 64 processors, each of these factors reduces achieved performance somewhat. The mapping strategy was seen to reduce performance by 15% to 30%. Concurrency limitations led to panel size choices that reduced performance by another 25%. We believe that imperfect task scheduling further reduced performance by another roughly 15%. As a result, maximum processor utilization levels were roughly 50%.

For larger parallel machines, two of these factors stand out as being particularly constraining: critical path length and interprocessor communication volume. Both require that a factor of two increase in the number of processors be accompanied by a factor of eight increase in the number of floating-point operations in the problem in order to afford any hope of achieving similar processor utilization levels. In general, we would expect these problem size growth rates to be unsustainable. As a result, more realistic problem sizes would achieve extremely low utilization levels for larger machines.

5.4 Improving Performance

Having investigated several factors that limit parallel performance, we now briefly consider the extent to which these factors can be improved. Recall that one source of inefficiency is load imbalance due to the panel mapping. The main source of this imbalance is the need to round to an integral number of processors when assigning subtrees to processor sets in the proportional mapping. One obvious means of alleviating this problem is to remove the requirement that the processor sets be disjoint, essentially allowing fractions of processors to be assigned to subtrees. We have performed some experiments using such a division scheme. While this scheme improves load balance significantly it also dramatically increases the difficulty of mapping individual panels to processors. The mapping strategy must somehow share a single processor among several distinct subtrees. Using the same mapping strategy that we used for the unmodified method, overall performance was not significantly improved. The advantages of the improved load balance were almost entirely offset by the reduced quality of the mapping.

An alternative approach to improving the load balance might use a more dynamic approach to task distribution. For example, a processor might have a 'preferred' set of tasks, corresponding to those tasks that it would perform in a statically scheduled computation. If a processor finds that it has no preferred tasks available, then it would steal a preferred task of another processor. One cost of such stealing would be increased interprocessor communication, since stolen tasks would presumably access data that is not local to the stealing processor. Initial experiments have indicated that the communication costs of this task stealing outweigh the load balance benefits on the DASH machine.

Another important limitation in a panel method is the length of the critical path, which plays a role in determining panel sizes and ultimately limits the number of processors that can be effectively used to solve a sparse problem. As far as the possibility of reducing the length of the critical path, we note that this problem has received some attention (see [29] and [33], for example). Note that the multiple-minimum-degree ordering heuristic we used to preorder the sparse matrices is known to produce 'tall' elimination trees and long critical paths, and thus would appear amenable to parallelism-increasing techniques. However, we believe any improvements will be small constant factors. The 2D grid problems, for example, are in many ways ideal for parallel machines, but they still suffer from critical path constraints.

On the question of whether the task schedule could be improved, we note that there appear to be significant opportunities for improvement. While finding an optimal schedule is clearly impractical, the schedule we have been using, which is implicit in the round-robin mapping of panels to processors, may be far enough off from optimal that it can be improved upon substantially. We have experimented with a more sophisticated simulation-based mapping strategy, where the panel-to-processor mapping is done using a rough simulation of the parallel computation. When a panel task is mapped to a processor, the simulated time of that processor is advanced to reflect the time at which the panel task was made available and the time required to perform that task. Each new panel task is assigned to the first available processor that is eligible to perform that task. This more sophisticated mapping strategy has shown initial promise, improving performance over a simple round-robin strategy by between 5% and 20%. However, this moderate overall performance improvement would most likely not warrant the increased complexity of this mapping approach.

In either case, the improvements discussed in this section would at best lead to small constant factor increases in performance. The most important factors limiting the performance, the available concurrency and the interprocessor communication volume, remain as imposing obstacles.

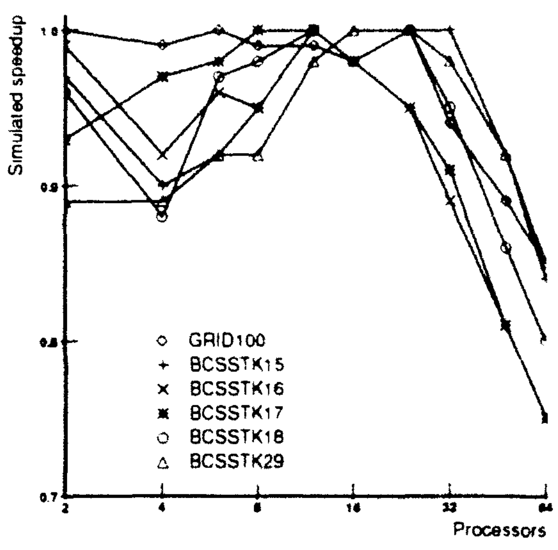


Figure 26: Performance for panel size of 8, relative to performance of best panel size.

5.5 Panel Size Selection

While the previous sections have made it clear that panel methods have some important limitations, at the same time they are still quite useful methods, particularly for moderately parallel machines. Indeed, they provide much higher performance than popular column methods. This section looks at an important issue for panel methods that has so far not been considered, the issue of choosing a panel size. Results from the previous section showed that overall performance varies quite dramatically with panel size. What it did not show was how to choose an effective size for a given sparse matrix and a given machine size.

Given that the marginal benefits of a larger panel size B fall off quickly as B increases, a reasonable strategy would be to always choose some fixed, relatively small panel size. The ideal size would be large enough so that it provides most of the benefit of large panels, while at the same time not being so large that it swallows enormous amounts of concurrency. For our performance model, a panel size of 8 strikes quite a reasonable balance. In cases where a larger panel size could be used, a choice of 8 still yields per-processor performance that is at least 75% of peak. In cases where a smaller panel size would have been more appropriate, a panel size of 8 still provides more than half of maximum concurrency.

To evaluate the effectiveness of this approach, Figure 26 shows relative efficiency numbers, comparing performance using a panel size of 8 with the best achieved performance over all panel size choices (under our performance model). As expected, performance for this strategy is quite reasonable, although it is less than ideal in two ranges. For small numbers of processors, a panel size of 8 represents lost opportunity, since a larger panel would be quite appropriate. For large numbers of processors, a panel size of 8 is too large, forcing processors to sit idle for significant portions of the computation.

A potentially better way to choose the panel size is to specifically tailor it to the matrix and the machine size. Performance results from the previous section indicated that the point at which performance is maximized is heavily dependent on the maximum load and critical path upper bounds. We now consider a panel size selection strategy based on these bounds.

Our approach to choosing a panel size considers panel sizes as falling into three different ranges. Consider the speedup graphs in Figure 21. In the first range, very small panels, the critical path bound is much higher than the maximum load bound, and performance is nearly equal the maximum load bound. With plenty of 'slack' in the computation, scheduling issues are less crucial and thus processors rarely sit idle. The second range includes panel sizes for which the two upper bounds are comparable. At such points, parallel performance is below both bounds (and performance is generally highest in this range). The third range includes large panel sizes, where the critical path is much more constraining than the maximum load and performance is nearly equal the former bound. The split points for these ranges naturally depend on the particular matrix and the particular machine size.

Based on these simple observations about performance in the various ranges, we use the following panel size selection strategy. The optimal panel size is clearly not within the first range. Overall performance can be increased here by increasing the panel size. Based on empirical observation, we consider any panel size for which the critical path bound is more than twice the maximum load bound to be in this first range. The optimal panel is also clearly not in the third range. Smaller panels would reduce processor idle time without significantly decrease per-processor performance. We consider this third range to include panel sizes where the maximum load upper bound is more than twice the critical path upper bound. The best panel size choice therefore sits somewhere within the second range.

To find a reasonable choice within this range, we make the following assumptions. First, we assume that when the panel size is increased, per-processor performance increases in proportion to the increase in the maximum load upper bound. And second, we assume that the number of processors that are active at a time decreases in proportion to the critical path upper bound. Maximizing performance is then a matter of finding the panel size where increasing the panel size leads to a marginal decrease in the critical path upper bound that is larger than the marginal increase in the maximum load upper bound. Note that this point can be computed quite inexpensively. We discussed simple and inexpensive ways to estimate the maximum load and critical path upper bounds given an arbitrary panel size earlier.

Applying this strategy to a range of sparse matrices from the Boeing/Harwell test set gives results shown in Figure 27. This figure again compares performance using our panel selection strategy

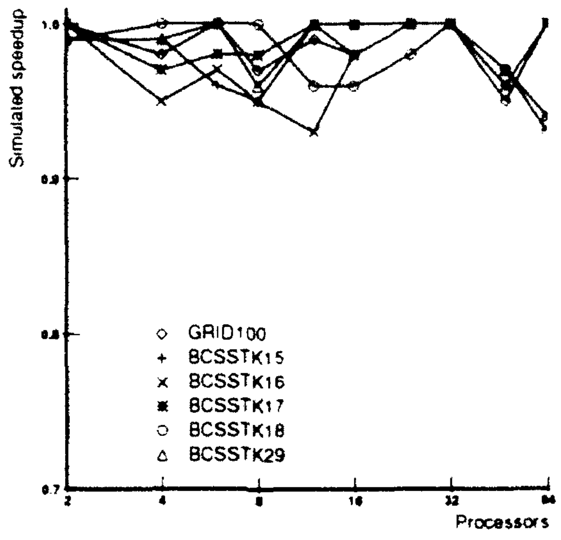


Figure 27: Performance relative to best case.

against the best performance with any panel size choice. Our strategy is clearly quite effective, choosing panel sizes that give 95% or more of peak performance in all cases. We should note that this strategy has been observed to be quite robust over a variety of machine model assumptions as well. We looked at machines with a range of different interprocessor communication bandwidths and cache miss costs, and in all cases this strategy chose panel sizes that gave near-optimal performance.

In summary, the choice of panel size plays an important role in determining overall parallel performance. The simple strategy of choosing a fixed panel size is reasonably effective. However, higher and more robust performance can be obtained by making use of information about performance bounds.

5.6 DASH Performance

To give a more global perspective on the results of this chapter, we now present performance results for the Stanford DASH machine across a range of problems and machine sizes. Figure 28 shows speedups over an efficient sequential code (left-looking supernode-supernode) for between 4 and 40 processors of the DASH machine when the best panel size is chosen. For 16 or fewer processors, this best panel size is usually 8. For more than 16 processors, the best panel size is usually 4. For reference, we note that the sequential code used to compute speedups performs at roughly 8 MFLOPS.

The reader should draw two conclusions from these performance results. First, parallel speedups are relatively low for this method, for reasons that have been discussed earlier in this chapter. The

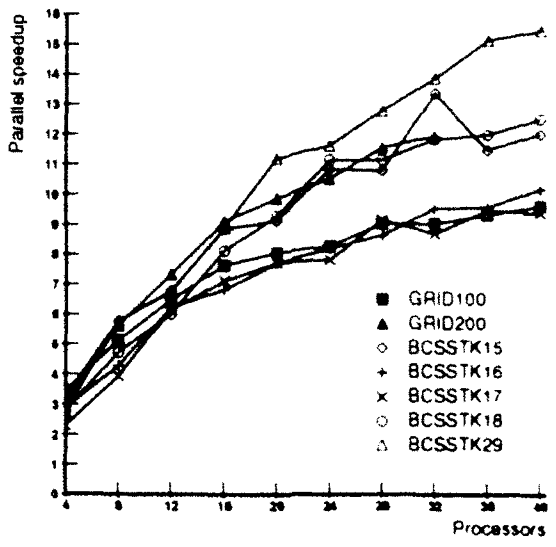


Figure 28: Parallel speedups on DASH machine.

individual processors in the parallel machine are not being very well utilized. At the same time the reader can also conclude that parallel distributed-memory machines can indeed provide high performance for sparse Cholesky factorization. In factoring a range of sparse matrices, the DASH machine consistently provides in excess of 75 MFLOPS, and it provides well over 100 MFLOPS for the larger matrices in the set. Thus, even with the relatively low speedups, a parallel machine still represents a cost-effective means of obtaining high performance for sparse Cholesky factorization.

5.7 Contributions

The first contribution of the work described in this chapter is our proposal to structure parallel factorization methods in terms of panels. While numerous methods for performing sparse Cholesky factorization on distributed-memory machines have been proposed (a few examples are [8, 9, 22, 34]), we are the first to have considered the use of a panel distribution to improve the use of a memory hierarchy. This chapter has investigated a multifrontal panel method, but we note that panels can be integrated into almost any column method, and in all cases they produce significantly higher performance.

Another contribution of this work is that it provides the first results for a high-performance factorization implementation on a relatively powerful distributed-memory parallel machine. Previous work has only considered very low performance machines (usually the iPSC/2). Parallel sparse factorization will only attract widespread interest once parallel machines provide performance that is substantially higher than that available on sequential machines. By investigating a high-performance parallel implementation in this chapter, we have demonstrated that good performance is indeed possible for this computation.

Another contribution of this work comes from its emphasis on understanding parallel performance and characterizing the factors that limit this performance. In particular, our performance modelling provides a strong foundation for understanding precisely why the parallel method obtains the performance that it does. It also allows us to understand the impact of changes in the panel width on overall performance, and thus to choose a good panel width. It also allows us to demonstrate the limitations that are inherent in any method that works with columns (or sets of columns) in the matrix. Little work had previously been done on modelling and understanding performance

5.8 Conclusions

This chapter has proposed and investigated a panel multifrontal approach to parallel sparse Cholesky factorization. We have found that panels are quite effective at increasing data reuse and thus improving performance over a more traditional column approach on parallel machines with caches. We observed factors of two to three improvement. However, we also found that panel methods, and indeed any methods that distributes columns of the matrix among processors, suffer from two severe limitations. They do not expose enough concurrency in the problem and they generate too much interprocessor communication traffic. Parallel speedups over efficient sequential methods were observed to be low for moderately parallel machines, and we would expect only moderate performance improvements from larger parallel machines.

Chapter 6

Dense Block-Oriented Factorization

The previous chapter showed that a panel decomposition has some severe limitations for Cholesky factorization on large parallel machines, both because it exposes limited amounts of concurrency and because it generates enormous amounts of interprocessor communication traffic. An obvious alternative to a panel decomposition is a 2-D decomposition, where the matrix is divided into a checkerboard of rectangular blocks. Such an approach has been used successfully for dense factorization on large parallel machines [44], and it has been proposed for sparse problems as well [4, 43, 45]. This chapter will investigate several important issues for methods that use a 2-D decomposition strategy.

While our ultimate aim in this thesis is to perform *sparse* factorization efficiently, this chapter will actually be devoted to a study of parallel *dense* factorization methods. Our intent is to thoroughly study several important questions that are relevant to all block methods, whether dense or sparse. Primary among these are questions of how the overall computation should be structured and what factors limit its performance. We will consider questions that relate specifically to sparse methods in the next chapter.

We should note that in many ways, efficient parallel dense Cholesky factorization is a well-understood problem. Indeed, an existing method has been shown to provide excellent performance on a wide range of parallel machine sizes [44]. There are, however, other possible approaches to this computation that have understandably received less attention. It may be the case that a block-oriented *sparse* method could obtain higher performance using one of these other approaches. This chapter investigates the performance of a range of dense factorization approaches to determine which would provide viable frameworks for building a sparse method.

6.1 Introduction

This chapter begins by considering the ways in which a parallel block-oriented dense Cholesky factorization can be structured. Just as panel factorization could be performed using several alternative formulations (fan-out, fan-in, multifrontal), a block decomposition leads to several different approaches. The primary difference among these approaches is in where updates to non-zeroes in the matrix are performed. We consider the two obvious choices. The first is a *destination-computes* (DC) approach, where all updates to the non-zeroes in a block are computed on the processor that owns the destination block. This is the approach used in [44]. The second is a source-computes (SC) approach, where updates are computed by the processor that owns one of the source blocks. We will describe simple parallel programs that implement both of these approaches.

The chapter continues by looking at the simulated performance of these two methods. As in the previous chapter, simulated speedups are compared against simple upper bounds, a maximum load bound and a critical path upper bound. The DC approach is found to provide performance that is nearly equal the upper bounds. The SC approach, on the other hand, gives performance that is well below the bounds and quite erratic. Since a SC approach could potentially be interesting in a sparse matrix context, we decide to further investigate its performance. We discuss the reasons for its erratic behavior and describe modifications that improve this behavior.

6.2 Block-Oriented Factorization

A 2D decomposition divides a dense matrix into a number of square blocks. A sequential factorization computation, expressed in terms of these blocks, would look like:

```
1.  for $K = 0$ to $N-1$ do
2.      $L_{KK}$ = Factor($L_{KK}$)
3.      for $I = K+1$ to $N-1$ do
4.          $L_{IK} = L_{IK} L_{KK}^{-1}$
5.      for $J = K+1$ to $N-1$ do
7.          for $I = J$ to $N-1$ do
8.              $L_{IJ} = L_{IJ} - L_{IK} L_{JK}^T$
```

Consider the set of operations that involve a particular off-diagonal block $L_{I'J'}$. The block receives a number of block updates (Step 8), where each update involves a pair of blocks from a previous block-column. Once all such updates have been performed, the block is multiplied by the inverse of the diagonal block $L_{J'J'}$ (Step 4). The block then acts a source block for updates in Step 8, updating subsequent blocks. Note that a block can appear as either the first or second source block in Step 8. In the first position ($I' = I$ and $J' = K$), block $L_{I'J'}$ updates blocks in block-*row* I'. In the second position ($I' = J$ and $J' = K$). $L_{I'J'}$ updates blocks in block-*column* I' This pattern will be important later in this chapter.

A diagonal block L_{KK} participates in a similar set of operations. It receives updates from all previous block-columns in Step 8. Once all updates have been performed, Cholesky factorization is performed on that block (Step 2). The block (its inverse, actually) is then used to solve blocks below it (Step 4).

Throughout this chapter, we will concentrate on the implementation of Step 8, the block update operation. This is by far the most important step in the computation.

Turning to a parallel implementation of this block-oriented computation. we note that each block will naturally be mapped to some processor, $map[L_{IJ}]$. That processor will hold the non-zeroes of the block in its local memory. Given a block mapping, a crucial question is where the block updates in Step 8 will be computed. Since this step involves three different blocks, there are three obvious processor candidates. One is to compute the update on the processor $map[L_{IJ}]$. Such a strategy is typically referred to as a ***destination-computes*** approach for obvious reasons. The second is to compute the update at $map[L_{IK}]$, a ***source-computes*** approach. The third candidate, computing the update on $map[L_{JK}]$, is also a source-computes approach.

We should note that other alternatives for distributing the computation exist. For example, Ashcraft has described a family of ***fan-both*** methods [6] that are hybrids of the SC and DC approaches. In these methods, multiple processors compute updates *from* a given block, and multiple processors compute updates *to* a given block. The main advantage of this class of methods is that they reduce interprocessor communication volumes. However, they also significantly increase storage requirements in an already memory-intensive computation. We therefore do not further consider these methods.

Another alternative for distributing the computation is to use a dynamic mapping strategy, where processors grab blocks when they are ready to perform computations with them. We will comment on this alternative later in this chapter.

Returning to the destination-computes and the two source-computes approaches to the computation, let us consider how these approaches affect the structure of a parallel method. When a block is mapped to a processor, that processor is then committed to performing the corresponding set of update operations. Figure 29 shows the updates that must be performed for the three task assignment strategies. If $map[L_{IK}]$ computes all updates, then in the course of the computation. $map[L_{IK}]$ will need to receive all blocks L_{JK}, $J < K$ (all blocks above it in the same block-column), and it will produce updates to blocks L_{IJ} to the right of L_{IK}. Similarly, if $map[L_{JK}]$ computes the updates, then it will need to receive all blocks L_{IK}, $K > J$ and will produce updates to blocks in a later block-column. If $map[L_{IJ}]$ computes updates, it will need to receive pairs of blocks, L_{IK} and LJK, from all earlier block-columns K.

We now consider the performance levels these approaches attain, the amounts of storage they

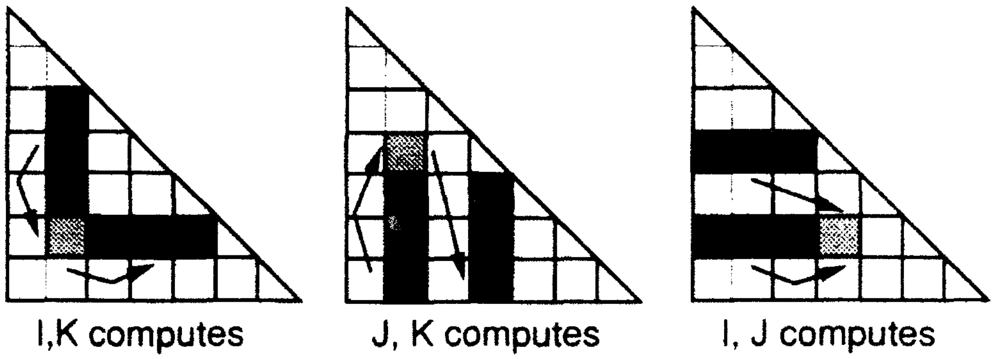


Figure 29: Blocks used for update operations.

require, and the communication volumes they generate. We actually restrict our attention to two of the three approaches, the L_{IJ} (*destination-computes*) approach and the L_{IK} (*source-computes*) approach. The other approach is sufficiently similar to the L_{IK} approach that conclusions about the latter should hold for the former as well.

6.3 Parallel Factorization Algorithms

6.3.1 Block Mapping

Before describing specific algorithms, we first describe the strategy we use for mapping blocks to processors. The same mapping will be used for both the SC and DC methods. Our mapping is done using a simple 2-D round-robin distribution. This commonly used approach looks at the set of processors P as a 2-D $\sqrt{P}$ by $\sqrt{P}$ grid, where each processor has some label $P_{i,j}$. This grid is then used in a "cookie-cutter" fashion to map sections of blocks to processors. That is, a block L_{IJ} is assigned to processor $P_{I \bmod \sqrt{P}, J \bmod \sqrt{P}}$. A four-processor example is shown in Figure 30. Besides doing a reasonable job of distributing the factorization work among the processors, this mapping strategy also possesses two properties that will be important for a parallel method. First, blocks that are neighbors in the matrix are mapped to processors that are neighbors in the processor grid. And second, a row of blocks is mapped to a row of processors, and similarly a column of blocks is mapped to a column of processors. We will discuss the benefits that these properties bestow shortly.

6.3.2 Destination-computes method

Structure of computation

Recall that the destination-computes approach performs block updates to a block by pairing blocks from earlier block-columns. The parallel computation is structured so that once a block is *completed*,

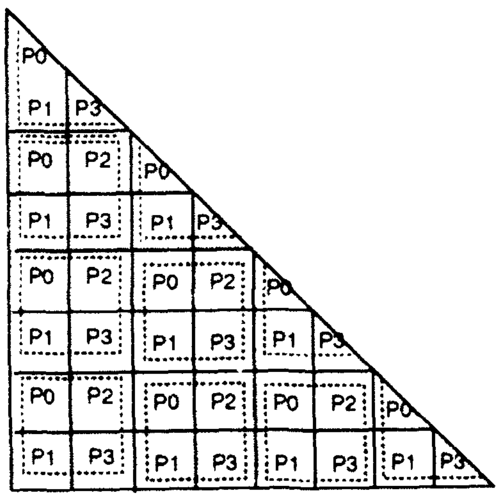


Figure 30: A 2-D round-robin distribution.

meaning that it has received all updates from previous blocks and has been multiplied by the inverse of the diagonal block, then it is sent to all processors that own blocks affected by it. When a processor p receives an off-diagonal block L_{IK} from another processor, it determines whether it has already received any blocks L_{JK} such that $map[L_{IJ}] = p$. The set of blocks that fit this condition is easily determined from the block mapping function. For each appropriate block L_{JK}, the corresponding update to L_{IJ} is performed. When a diagonal block L_{KK} arrives at a processor, all blocks owned by that processor in block-column K are checked to determine whether they have received all updates and are ready to by multiplied by the inverse of the diagonal. If not, the diagonal block is queued.

Recall that a block L_{IJ} receives one update from each block-column to its left in the matrix. To determine when a block has received all such updates, a count is kept of the number of updates performed so far. When the count reaches $J - 1$, then the block is multiplied by the inverse of the diagonal block L_{JJ} (this is done immediately if the diagonal is available, or when the diagonal block arrives otherwise). The block is then sent to all processors that own blocks modified by L_{IJ}. If the block is a diagonal block, then it is factored and sent to all processors that own blocks below it.

An important question here is how to determine the set of processors that own blocks affected by a particular block. Recall from an earlier discussion that a block L_{IK} only affects blocks in row I or column I. Recall also that the 2D round-robin mapping strategy maps a row/column of blocks to a row/column of processors. Thus, the block can simply be multicast to row/column $I \bmod \sqrt{P}$ of the processor grid. This technique for limiting communication was originally proposed in [18] and has been exploited in parallel implementations of several linear algebra computations.

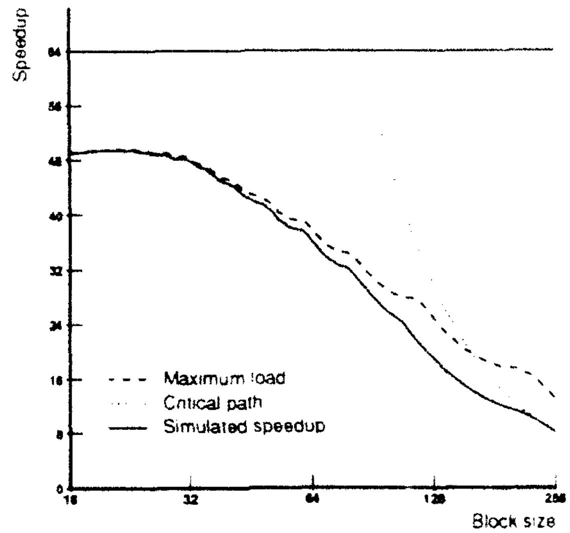


Figure 31: Performance results for destination-computes method. $n = 2048$, $P = 64$.

Simulated performance

Figure 31 shows simulated performance for the method described above, using 64 processors to factor a 2048 x 2048 dense matrix. The figure compares simulated speedups with critical path and maximum load upper bounds across a range of block sizes, as we have done in previous chapters. The results show that this destination-computes method is quite effective, yielding performance that is nearly equal the maximum load upper bound for all but the largest block sizes. Furthermore, this method can make excellent use of a memory hierarchy. The 2-D decomposition exposes sufficient concurrency in the problem to allow a relatively large block size to be used. The block size of 32 which is used in the figure, for example, achieves excellent use of a processor cache without exhausting available concurrency.

Comparing this approach to a panel method, we find the maximum parallel speedup achieved with a panel method is roughly 31. This is significantly below the roughly 48-fold speedup for the block approach. The differences between the two approaches are expected to be even more pronounced with more processors.

Communication volume

Another important quantity in a parallel method is the volume of interprocessor communication. Total communication volume for a destination-computes method can be computed as follows. All blocks, with the exception of the diagonal blocks, are sent to a row and a column of processors, or $2\sqrt{P}$ total processors, in the course of the computation. Since every non-zero in the matrix belongs to some block and there are $n^2/2$ non-zeroes, total communication volume is the product of $n^2/2$ and $2\sqrt{P}$, or $n^2\sqrt{P}$ words.

To compare this communication volume with that generated by a panel or 1-D decomposition, note that a dense panel approach would broadcast every panel to every processor, giving $n^2P/2$ total communication volume. The block distribution thus substantially reduces communication volume.

Another important thing to note about this destination-computes block-oriented approach is that its communication volume is independent of the block size. The block size can therefore be chosen with other issues in mind. The block size we use is 32 by 32. Such blocks are sufficiently large that they fully exploit the processor cache, as per our performance model, and thus give optimal per-processor performance. While the load balance would be somewhat better with a finer division, any improvement would come at the cost of a reduction in per-processor performance.

6.3.3 Source-computes method

Structure of computation

The other block factorization approach we consider is the L_{IK}-computes, or source-computes approach. The structure of this parallel computation is quite straightforward. When a block L_{JK} is completed, it is multicast to all processors $map[L_{IK}]$, $I > J$ (i.e., all processors that own blocks below L_{JK} in column K). When a processor receives a complete off-diagonal block L_{JK} destined for a block L_{IK} that it owns, it checks whether L_{IK} is complete as well. If so, the processor computes an update to block L_{IJ} and sends it to $map[L_{IJ}]$. If not, the received block L_{JK} is queued with L_{IK}. When a processor receives a complete diagonal block destined for some L_{IK}, it checks to see if L_{IK} has received all updates, and performs the inverse multiplication if it has. If it has not, then the diagonal block is queued with L_{IK}.

To determine when a block is ready, an update count is again kept with each block. When the count for L_{IK} reaches $K - 1$, the block is multiplied by the inverse of the corresponding diagonal block (if the diagonal block is available). Once completed, the block can be used to compute updates corresponding to queued blocks.

One small modification to the above approach allows it to interact more naturally with a grid of processors. Rather than sending the update directly from $map[L_{IK}]$ to $map[L_{IJ}]$, which may lead to messages between physically distant processors in the parallel machine, the update can instead be sent from $map[L_{I,K}]$ to $map[L_{I,K+1}]$, an immediate neighbor. The update can then be combined with the update from $L_{I,K+1}$ to L_{IJ}, with the combined update being sent off to $L_{I,K+2}$, and so on.

Simulated performance

Figure 32 shows simulated performance for this source-computes implementation, again using 64 processors for a 2048 by 2048 dense factorization. Note that performance is quite erratic and is

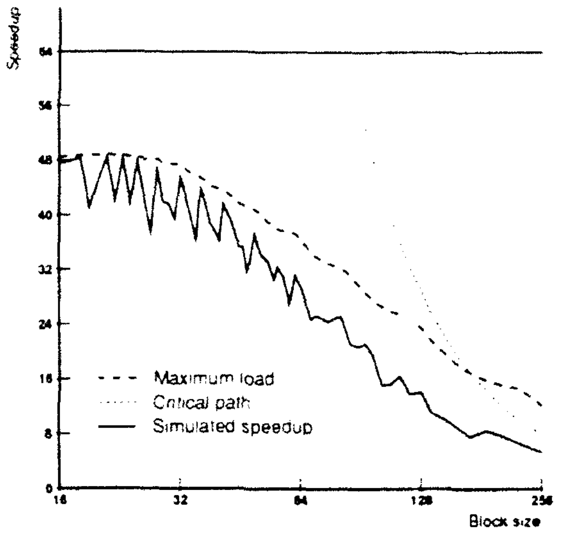


Figure 32: Performance results for source-computes method. $n = 2048$, $P = 64$.

generally well below the maximum load upper bound. Let us briefly look at the reasons for this behavior.

Consider the simple example in Figure 33. Assume that each block is assigned to a different processor. The parallel computation begins with the factorization of $L_{0,0}$. This block is then multicast to all blocks in column 0, and the corresponding owner processors perform inverse multiplications. Several off-diagonal blocks in column 0 complete roughly simultaneously, and they are then multicast to all blocks below them. Messages corresponding to each of these blocks will arrive at block $L_{4,0}$, and these messages will cause the corresponding block update operations to be performed. Since the blocks above $L_{4,0}$ complete at roughly the same time, and assuming there is some small random component to their completion times, it is reasonable to assume that the blocks will arrive in a random order, and thus the updates from $L_{4,0}$ will be computed in a random order.

Now consider block $L_{4,1}$. Processor $map[L_{4,1}]$ cannot begin computing updates until $L_{4,1}$ receives an update from $L_{4,0}$. Ideally, this would be the first update computed by $L_{4,0}$. However, due to the random arrival order this is quite unlikely. If updates are computed in a first-come, first-served order, then the update to $L_{4,1}$ would typically happen after several other updates. Note that these other updates are much less important than the update to $L_{4,1}$. An update to $L_{4,2}$, for example, does not enable $map[L_{4,2}]$ to begin computing updates because $L_{4,2}$ must also receive an update from $L_{4,1}$. Note that while we have only looked at column 0, similar delays will occur in subsequent columns as well.

The observed performance is therefore easily understood. It is below the upper bounds because processors spend significant amounts of time sitting idle, waiting for important updates that happen

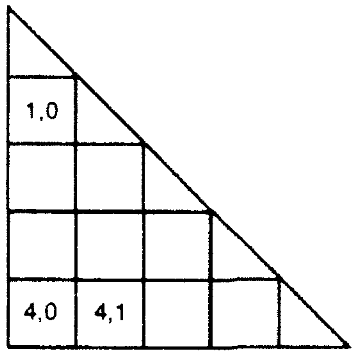


Figure 33: A simple block example.

after less important updates. Performance is erratic because the amount of time a processor must wait depends on the order in which blocks arrive at a processor, which is random and thus can change from run to run.

Note that the performance numbers shown here are by no means worst-case results for a source-computes approach. Subtle differences in implementation can lead to huge swings in performance. For example, our initial source-computes implementation handled blocks that arrived at a pair block before the pair was complete somewhat differently. Instead of holding them in a queue of waiting blocks, we instead held them in a stack, which is somewhat easier to implement. The fact that the matrix is triangular actually leads to a somewhat reasonable arrival order for later columns in the matrix, but this reasonable order was reversed by the stack implementation. Performance was often a factor of two or more lower than performance for the queue-based approach.

A poor task execution order is not the only problem with this source-computes approach to the computation. Another problem is its per-processor storage requirements. A processor that owns a block towards the bottom of a column would receive all blocks in that column nearly simultaneously. Unfortunately, as we will show shortly, this approach requires large blocks to keep communication volumes low. The column of large blocks that arrive at a processor would generally require more storage than the blocks actually assigned to that processor, thus severely limiting the size of problem that could be solved.

Prioritized method

The obvious solution to the problem of updates not being computed in the right order is to prioritize the computation of these updates. We use the following simple scheme. Each processor chooses as its *working block* the leftmost block it owns that has not yet been used to produce updates. The leftmost block is chosen because it will generally be the one that is first ready to produce updates

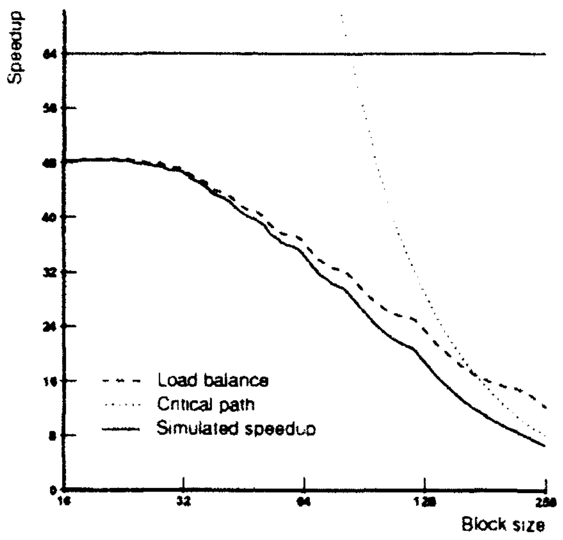


Figure 34: Performance results for prioritized source-computes method. $n = 2048$, $P = 64$.

The processor then produces all updates from this working block in order of increasing destination column number. This order more closely matches the true urgencies of the updates. Only once all updates from a block have been computed does the processor move on to another working block. In order to reduce storage requirements, a processor explicitly requests blocks from other processors when it is ready to use them.

As a simple example, consider block $L_{4,0}$ from the earlier example. In a prioritized scheme, processor $map[L_{4,0}]$ would begin the computation by requesting that $map[L_{0,0}]$ send the diagonal block. After modifying $L_{4,0}$ by this block, the processor would request $L_{1,0}$ from its owner processor. Once the corresponding update is computed, the processor would continue by requesting $L_{2,0}$ and so on. Software pipelining can be used to avoid having the processor sit idle while a block request is serviced. That is, a processor can request block $L_{J+1,K}$ while computing the update that results from block $L_{J,K}$.

Figure 34 shows the simulated performance for this simple prioritization scheme (64 processors, 2048 by 2048 matrix). The prioritization removes the erratic behavior of the first-come, first-served approach, and it also yields performance that is nearly equal the load balance upper bound. Indeed, predicted performance for the prioritized source-computes method is roughly equal that of the destination-computes approach for equal block sizes.

Communication volume

Communication volume for the source-computes approach is easily computed as follows. For every block update from some block L_{IK}, one block L_{JK} is communicated from above and one update to L_{IJ} is sent to the right, giving a total of $2B^2$ communication, where B is the block size. This update operation performs $2B^3$ floating-point operations. Thus, B floating-point operations are performed for every word of interprocessor communication. Since the entire computation performs $n^3/3$ floating-point operations, the parallel computation therefore communicates $n^3/3B$ words between processors.

Comparing this n^3/B communication rate to the $n^2\sqrt{P}$ rate for the destination-computes approach, we find that in order for the two to produce the same volume of communication, the block size in the source-computes method must grow with the problem size. In particular, the two approaches produce identical communication volumes when $B = n/3\sqrt{P}$. In general, the corresponding block size will be much larger than the block size that can be used with a destination-computes approach, yielding significantly worse load balance. In the earlier example, where 64 processors were used to factor a 2048 by 2048 matrix, the block size that yields equal communication is 85. The simulated parallel speedup for the destination-computes method with a block size of 32 is roughly 49, while the speedup for the source-computes approach with a block size of 85 is roughly 28.

As a brief aside, we note that the source-computes approach may have advantages over a destination-computes approach in environments where the number of processors is either not known a-priori or is subject to change during the computation (i.e., in a multiprogrammed environment). If the source-computes computation were structured using an entirely dynamic schedule, where at run-time processors grabbed the first available L_{IK} block and produced the corresponding set of updates, the resulting computation would generate a comparable volume of communication as the statically scheduled version. The above communication results for the source-computes approach assume little about block placement; the results are little changed when blocks are scattered randomly about the machine. The destination-computes approach, on the other hand, makes several assumptions about block placement and thus would not be nearly as amenable to dynamic scheduling.

6.3.4 Summary

Based on the results of this section, we conclude that both the destination-computes and source-computes approaches to dense Cholesky factorization are viable approaches, although the source-computes approach requires more attention to the details of scheduling and storage. Of the two, the destination-computes approach is preferable because of its communication behavior. The remainder of this chapter will concentrate on the destination-computes approach.

6.4 Predicting Performance

So far, we have only presented performance results for a single problem size and a single machine size. To expand our results, Figure 35 shows simulated parallel processor utilization numbers across a wider range of problem/machine sizes. This figure shows performance for a destination-computes

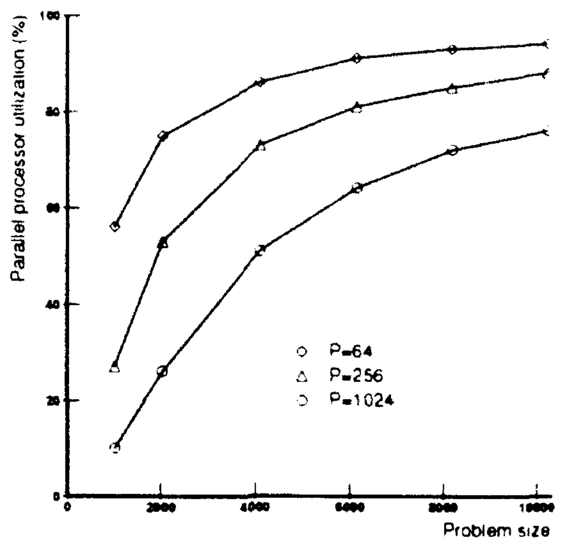


Figure 35: Performance versus problem size for destination-computes method

method using a block size of 32. As was the case with the examples shown earlier, performance has been observed to be nearly equal the maximum load upper bound at all points.

In order to obtain a better feel for the way in which achieved performance relates to parameters such as the block size, the machine size, and the problem size, we now derive an analytical expression for maximum processor utilization. Since performance is constrained by load balance, the bound is based on a calculation of maximum load assigned to any processor. We derive this expression for a destination-computes strategy, although the identical bound holds for the source-computes method.

The balance of computational load will naturally be determined by the set of blocks assigned to a processor, and the amount of work required for each block. Recall that block receives one update for each block-column to its left. Since each individual block update operations performs the same amount of work, the work associated with a block L_{IJ} is therefore proportional to J.

The processor that receives the most work in a 2-D round-robin mapping is easily determined. Think of the dense matrix as an S by S matrix of super-blocks, where each super-block is one cookie-cutter worth of blocks in the round-robin mapping. In the example of Figure 30, S is 4. The processor in the lower-right corner of the cookie-cutter (processor P3 in the example) always receives the block within a super-block that requires the most work, and thus it receives the most work overall.

Now consider the exact amount of work this processor receives. For the super-block in position I, J, the lower-right processor owns a block that receives $J\sqrt{P}$ block updates, each of which requires $2B^3$ floating-point operations. Summing over the whole matrix, the number of floating-point operations assigned to the lower-right processor is

$$\sum_{I=1}^{S}\sum_{J=1}^{I} J\sqrt{P}2B^3 = \frac{S(S+1)(S+2)}{3}\sqrt{P}B^3$$

The total amount of work in the entire computation is $n^3/3$, and since $n = S\sqrt{P}B$, ideally work per processor would be:

$$\frac{S^3P^{3/2}B^3}{3P}$$

Dividing the maximum load on any processor by the ideal load per processor gives the following upper bound on processor utilization:

$$\frac{S^2}{(S+1)(S+2)}$$

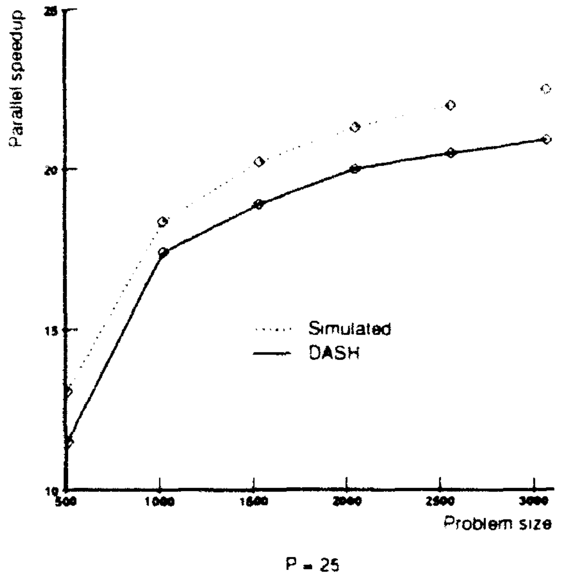
We have empirically found this simple function to be a very accurate predictor of parallel performance. It also tells us a great deal about the general behavior of the parallel method. For example, processor utilizations levels of 50% are reached quite quickly ($S = 4$). However, higher levels require much larger S values. A level of 75% requires $S = 10$ and a level of 90% requires $S = 28$. To put these numbers in better perspective, note that a 1024 processor machine using a block size of 32 would require an $n = 10,000$ problem to achieve 75% utilization, and an $n = 28,000$ problem to achieve 90% utilization. In other words, a cookie-cutter block distribution is quite effective at providing 'reasonable' processor utilization levels, but it requires quite large problems before utilizations are pushed into the 80-100% range.

6.5 Model Verification

So far in this chapter we have looked only at simulated performance. We now look at the accuracy of our model in comparison to real machine performance. Figure 36 compares actual parallel speedups on the Stanford DASH machine (25 and 36 processors) with simulated speedups for the same problems. The DASH speedups are somewhat below the predicted speedups, but they are quite close.

6.6 Conclusions

This chapter has considered parallel dense Cholesky factorization using a 2-D, or block-oriented matrix decomposition. An important objective in looking at dense factorization has been to understand more general issues of how a block-oriented Cholesky factorization should be structured. We conclude that of the two reasonable choices, destination-computes or source-computes, both are viable options but a destination-computes strategy is preferable. It is simpler to implement, provides



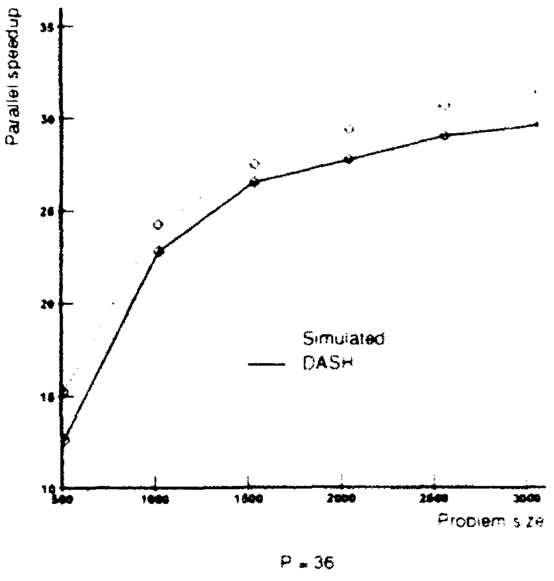


Figure 36: Simulated and actual speedups for destination-computes method, for 25 and 36 processors. Actual speedups are from the Stanford DASH machine.

more flexibility in the choice of the block size, and has fewer implementation pitfalls. Our goal in the next chapter will therefore be to devise a sparse block method that uses a destination-computes framework.

Chapter 7

Sparse Block-Oriented Factorization

7.1 Introduction

Having investigated general issues for dense block-oriented Cholesky factorization, we now turn specifically to *sparse* block methods. This chapter focuses on two practical and important questions related to sparse block-oriented factorization. First, we consider the complexity of a parallel sparse factorization program that manipulates sub-blocks. We show that a block approach need not be much more complicated than a column approach. We describe a simple strategy for performing a block decomposition and a simple parallel algorithm for performing the sparse Cholesky computation in terms of these blocks. The approach retains the theoretical scalability advantages of block methods. We term this block algorithm the ***block fan-out method***, since it bears a great deal of similarity to the parallel column fan-out method [21].

Another important issue in a block approach is the issue of efficiency. While parallel scalability arguments can be used to show that a block approach would give better performance than a column approach for extremely large parallel machines, these arguments have little to say about how well a block approach performs on smaller machines. Our goal is to develop a method that is efficient across a wide range of machine sizes. We explore the efficiency of the block approach in two parts. We first consider a sequential block factorization code and compare its performance to that of a true sequential program to determine how much efficiency is lost in moving to a block representation. The losses turn out be quite minor. We then consider parallel block factorization, looking at the issues that potentially limit its performance. The parallel block method is found to give extremely high performance even on small parallel machines. For larger machines, performance is good but not excellent, primarily due to load imbalances. We quantify these load imbalances and investigate the causes.

This chapter is organized as follows. Section 7.2 describes our strategy for decomposing a sparse matrix into rectangular blocks. Section 7.3 describes a parallel method that performs the factorization in terms of these blocks. Section 7.4 then evaluates the parallel method, both in terms of communication volume and achieved parallel performance. Section 7.5 gives a brief discussion of our results. Section 7.6 discusses future work. Section 7.7 discusses related work, and finally conclusions are presented in Section 7.8.

7.2 Block Formulation

Perhaps the most important question in a block-oriented sparse factorization is how to structure the sparse Cholesky computation in terms of blocks. Consequently, our first step in describing a block-oriented parallel method is to propose a strategy for decomposing the sparse matrix into blocks. Our goal in this decomposition is to retain as much of the efficiency of a sequential factorization computation as possible. Thus, we will keep a careful eye on the amount of computational overhead that is introduced.

7.2.1 Block Decomposition

We begin our discussion by considering some of the general issues that are important for a block approach. We also discuss how our approach addresses these issues. We believe the main issues that must be addressed are the following. First, blocks should be relatively dense. Since the blocks will be distributed among several processors, there will certainly be some overheads associated with manipulating and storing them. These overheads should be amortized over as many non-zeroes as possible. The block decomposition must therefore be tailored to match the non-zero structure of the sparse matrix. Another important issue is the way in which blocks in the matrix interact with each other. If the interactions are complex, then the parallel computation can easily spend more time figuring out how blocks interact than it would spend actually performing the block operations. Finally, the individual block operations should be efficient.

The primary motivation behind our decomposition approach is to keep the block computation as simple and regular as possible. Our hope is that a regular computation will be an efficient computation. We keep the computation simple by avoiding two distinct types of irregularity: (1) irregular interactions between blocks; and (2) irregular structure within blocks.

Irregular Interactions

Since a sparse matrix in general contains non-zeroes interspersed with zeroes throughout the matr
it would appear desirable for a block decomposition to possess a large amount of flexibility in choosing blocks. This flexibility could be used to locally tailor the block structure to match the actual

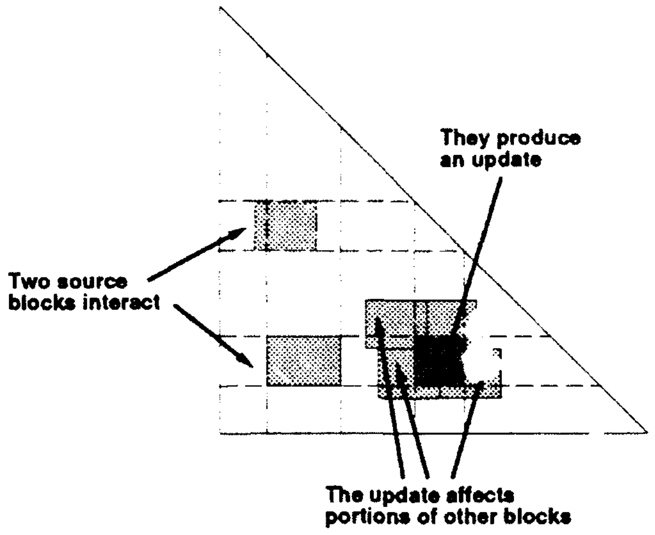

They produce an update
Two source blocks interact
The update affects portions of other blocks

Figure 37: Example of irregular block interaction. Dotted lines indicate boundaries of affected areas.

structure of the sparse matrix. One seemingly reasonable decomposition approach, for example, would locate clumps of contiguous non-zeroes in the matrix and group these clumps together into blocks. This approach has serious problems, however, and we now discuss the advantage of giving up some flexibility and instead imposing a significant amount of rigidity on the decomposition.

The primary problem with a flexible approach to block decomposition concerns the way in which the resulting blocks would interact with each other. Recall that in sparse Cholesky factorization a single non-zero L_{ik} is multiplied with non-zeroes above it in the same column L_{jk} to produce updates to non-zeroes L_{ij} in row i and column j. When the matrix is divided into a set of rectangular blocks, the blocks interact in a similar manner. Consider the simple example in Figure 37. This figure shows a small set of dense blocks from a potentially much larger matrix. During the factorization, the block in the lower left will interact with a portion of the block above it to produce the indicated update, which must be subtracted from portions of the blocks to its right. Keep in mind that each of these blocks is potentially assigned to a different processor. Thus, for each update operation the processor performing that update must keep track of the set of blocks that are involved, the portions of these blocks that are affected, the processors on which these blocks can be found, and it must dole out the computed update to the relevant processors. Keeping track of all such block interactions would be enormously complicated and expensive. With a large number of blocks scattered throughout the matrix, the cost of this irregularity would quickly become prohibitive.

In order to remove this irregularity and greatly simplify the structure of the computation, we decompose the matrix into blocks using global partitions of the rows and columns. In other

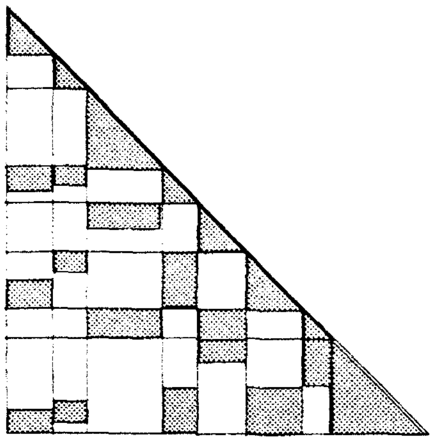


Figure 38: Example of globally partitioned matrix.

words, the columns of the matrix $(1 \dots n)$ are divided into contiguous sets $(\{1 \dots p_2 - 1\}, \{p_2 \dots p_3 - 1\}, \dots, \{p_N \dots n\})$, where N is the number of partitions and p_i *is the first column in partition* i. An identical partitioning is performed on the rows. A simple example is shown in Figure 38. A block L_{IJ} (we refer to partitions using capital letters) is then the set of non-zeroes that fall simultaneously in rows $\{p_I \dots p_{I+1} - 1\}$ and columns $\{p_J \dots p_{J+1} - 1\}$. The main advantage of this rigid distribution comes from the fact that blocks share common boundaries. A block L_{IK} now interacts with block L_{JK} in the same block column partition to produce an update to block L_{IJ}.

One possible weakness of a global partitioning strategy is that its global nature may not allow for locally good blocks. We will soon show that this is only a minor problem.

Irregular Block Structure

Another issue that can have a significant impact on the efficiency of the overall computation is the internal non-zero structure of a block. Just as we restricted the choice of block boundaries earlier to increase regularity across block operations, we now consider restrictions on the internal structures of blocks to increase regularity within a block operation.

Note first that allowing arbitrary partitionings of the rows and columns of the matrix would lead to blocks with arbitrary internal non-zero structures. Recall that a block update operation is performed by multiplying a block by the transpose of a block above it (as a matrix-matrix multiplication). With arbitrary non-zero structure within the blocks, the corresponding computation would be a sparse matrix multiplication, which is an inefficient operation in general.

In order to simplify the internal structure of the blocks and keep the computation as efficient as possible, we take advantage of the supernodal structure of the sparse matrix. Specifically, we choose partitions so that all member columns belong to the same supernode. Since the columns in a supernode all have the same non-zero structures, all resulting blocks will share this property. Thus, a block L_{IJ} will consist of some set of dense rows. A block may not be completely dense, since not all rows are necessarily present. A single structure vector keeps track of the set of rows present in a block. This sparsity within a block has little effect on the efficiency of the computation, as we shall soon show.

Before proceeding, we note that Ashcraft [4] proposed a similar decomposition strategy independently.

7.2.2 Structure of the Block Factorization Computation

Our goal in placing the above restrictions on blocks in the sparse matrix is to retain as much efficiency as possible in the block factorization computation. We now describe a sequential algorithm for performing the factorization in terms of these blocks and evaluate that algorithm's efficiency. The parallelization of the sequential approach that we derive here will be described later.

At one level, the factorization algorithm expressed in terms of blocks is quite obvious. The following pseudo-code, a simple analogue of dense block Cholesky factorization, performs the factorization. Note that I, J, and K iterate over the partitions in the sparse matrix.

```
1. for $K = 0$ to $N-1$ do
2.     $L_{KK} \leftarrow$ Factor($L_{KK}$)
3.     for $I = K+1$ to $N-1$ with $L_{IK} \neq 0$ do
4.         $L_{IK} \leftarrow L_{IK} L_{KK}^{-1}$
5.     for $J = K+1$ to $N-1$ with $L_{JK} \neq 0$ do
6.         for $I = J$ to $N-1$ with $L_{IK} \neq 0$ do
7.             $L_{IJ} \leftarrow L_{IJ} - L_{IK} L_{JK}^T$
```

The first thing to note about the above pseudo-code is that it works with a column of blocks at a time. Steps 2 through 4 divide block column K by the Cholesky factor of the diagonal block. Steps 5 through 7 compute block updates from all pairs of blocks in column K. We therefore store the blocks so that all blocks in a column can be easily located. This is accomplished by storing one column of blocks after another, just as sparse column representations would store one column of non-zeroes after another. One potential problem here is that step 7 updates some destination block L_{IJ} whose location cannot easily be determined from the locations of the source blocks. To make this step efficient, a hash table of all blocks is kept.

Now consider the implementation of the individual operations in the pseudo-code. The block factorization in step 2 is quite straightforward to implement. Diagonal blocks are guaranteed to be dense, so this step is simply a dense Cholesky factorization. The multiplication by the inverse of the diagonal block in step 4 is also quite straightforward. This step does not actually compute the inverse of L_{KK}. Instead, it solves a series of triangular systems. While the block L_{IK} is not necessarily dense, the computation can be performed without consulting the non-zero structure of the block.

The remaining step in the above pseudo-code, step 7, is both the most important and the most difficult to implement. It is the most important because it sits within a doubly-nested loop and thus performs the vast majority of the actual computation. It is the most difficult because it works with blocks with potentially different non-zero structures and must somehow reconcile these structures. More precisely, recall that a single block in L consists of some set of dense rows from among the rows that the block spans (see the example in Figure 38). When an update is performed in step 7 above, the structure of L_{IK} determines the set of rows in L_{IJ} that are affected. Similarly, the structure of L_{JK} determines the set of *columns* in L_{IJ} that are affected.

The block update computation is most conveniently viewed as a two-stage process. A set of updates is computed in the first stage, and these updates are subtracted from the appropriate entries in the destination block in the second, or scatter stage. The first stage, the computation of the update, can be performed as a dense matrix-matrix multiplication. The non-zero structures of the source blocks L_{IK} and L_{JK} are ignored temporarily; the two blocks are simply multiplied to produce an update.

During the second stage, the resulting update must be subtracted from the destination. The most simple case occurs when the update has the same non-zero structure as the destination block. We have coded our dense matrix-matrix multiplication routine as a multiply-subtract (i.e., $C = C - AB^T$), rather than a multiply-add, so the destination block can be used as the destination directly, without the need for a second scatter stage.

Consider the more difficult case where the non-zero structures differ. The first step in this case is to compute a set of ***relative indices*** [42]. These indices indicate the corresponding position in the destination for each row in the source. Two sets of relatives indices are necessary in order to scatter a single block update; rel_i, the affected set of rows and rel_j, the affected set of columns.

The computation of relative indices is quite expensive in general, since it requires a search through the destination to find the row corresponding to a given source row. Fortunately, such a search is only rarely necessary due to an important special case. When the destination block has dense structure, the relative indices bear a trivial relationship to the source indices. Note that the rel_j indices always fall into this category, since the destination block always has dense column structure. We will be more precise about exactly how often relative index computations are necessary shortly.

Once relative indices have been computed, the actual scatter is performed as follows:

```
1.  for $i = 0$ to $length_{IK} - 1$ do
2.      for $j = 0$ to $length_{JK} - 1$ do
```

3. $$L_{IJ}[rel_i[i]][rel_j[j]] \leftarrow L_{IJ}[rel_i[i]][rel_j[j]] - update[i][j]$$

Scattering is also somewhat expensive, and it is much more prevalent than relative index computation. The frequency with which relative index computations and scatters must be performed will be considered shortly.

In summary, the efficiency of a block update operation depends heavily on the non-zero structures of the involved blocks.

- The best case occurs when the update has the same structure as the destination. In this case the $C = C - AB^T$ operation can use the destination block as its destination.
- The next best case occurs when the destination block is dense. The update must be scattered but the relative indices can be computed inexpensively.
- The worst case occurs when the update has different structure from the destination and the destination block is sparse. The update must be scattered, and relative indices are relatively expensive to compute.

7.2.3 Performance of Block Factorization

We now look at the performance obtained with a sequential program that uses a block decomposition and block implementation. Since our end goal is to create an efficient *parallel* approach, performance is studied for the case where the matrix is divided into relatively small blocks. The blocks should not be too small, however, because of the overheads that will be associated with block operations. We consider 16 by 16, 24 by 24, and 32 by 32 block sizes. To produce blocks of the desired size B, we form partitions that contain as close to B rows/columns as possible. Since partitions are subsets of supernodes, some partitions will naturally be smaller than B.

The performance obtained with the sequential block approach on a single processor of the Stanford DASH machine is shown Figure 39. This performance is expressed as a fraction of the performance obtained with an efficient sequential code (a supernode-supernode left-looking method). From the figure, it is clear that the block approach is relatively efficient. Efficiencies for four of the seven matrices are roughly 65% for a block size of 16 and roughly 75% for a block size of 32. We will discuss the reasons why the other three matrices, GRID100, GRID200, and BCSSTK18, achieve significantly lower performance shortly.

Our earlier discussion indicated that the performance of the block method might suffer because of the need for relative index calculations and update scattering. In order to gauge the effect of these two issues on overall performance, Table 29 relates the amounts of scattering and relative index computation (for $B = 16$) to the number of floating-point operations performed in the factorization. The numbers are quite similar for the other block size choices. The first column compares the

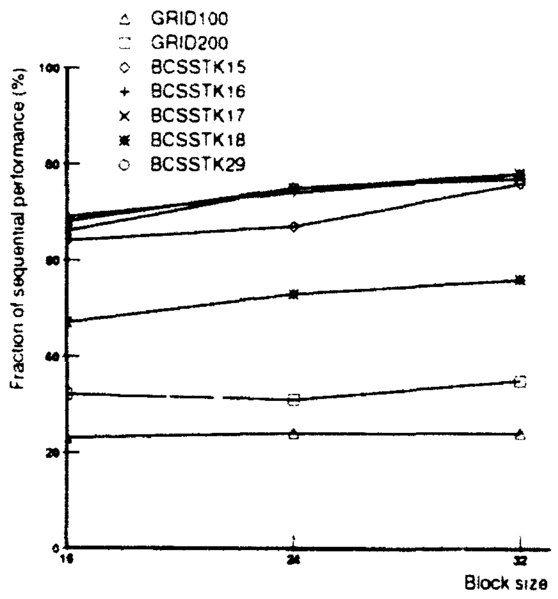


Figure 39: Performance of a sequential block approach, relative to a sequential left-looking supernode-supernode approach, on a single processor of the Stanford DASH machine.

Table 29: Frequency of relative index computations and scatters for block method, compared with floating-point operations ($B = 16$).

Problem	Relative indices (relative to FP ops)	Scatters (relative to FP ops)
GRID100	0.37%	4.0%
GRID200	0.18%	2.4%
BCSSTK15	0.04%	1.6%
BCSSTK16	0.02%	1.4%
BCSSTK17	0.04%	1.8%
BCSSTK18	0.11%	2.6%
BCSSTK29	0.01%	1.0%

Table 30: Frequency of relative index computations and scatters for block method, compared with sequential multifrontal method ($B = 16$).

Problem	Relative indices (relative to seq MF)	Scatters (relative to seq MF)
GRID100	78%	72%
GRID200	80%	69%
BCSSTK15	109%	105%
BCSSTK16	50%	88%
BCSSTK17	61%	90%
BCSSTK18	163%	91%
BCSSTK29	32%	40%

number of distinct relative indices computed against the number of floating-point operations. The second column compares distinct element scatters against floating-point operations. The table shows that even if relative index computations and scatters are much more expensive than floating-point operations, the related costs will be small. Clearly, the vast majority of block update operations produce an update with the same structure as the destination block.

It is also interesting to compare relative indices and scatters to those performed by a true sequential method. Table 30 gives the relevant numbers. In this case, the comparison is with a sequential multifrontal method, where notions of relative indices and scatters are easily quantified. The comparison is relevant for the left-looking supernode-supernode as well, since the two methods perform similar computations. Note that the block method performs a comparable number of relative index computations and scatters.

Ashcraft [4] has described methods for improving block structure and thus decreasing the need for scattering. It is our belief that a very simple block decomposition is more than adequate for keeping such costs in check.

7.2.4 Improving Performance

It is clear from the previous section that the block method is generally quite efficient. Recall, however, that the method was much less efficient than a true sequential method for several problems. Data on relative index and scatter frequency showed that these were not the source of the losses. The losses are actually due to overheads in the block operations.

Consider a single block update operation. It must find the appropriate destination block through a hash table, determine whether the source and destination blocks have the same structure, and then pay the loop startup costs for the dense matrix multiplication to compute the update. While these costs are trivial when all involved matrices are 32 by 32, in fact many blocks in the sparse matrix are quite small. In the case of matrix GRID100, for example, the average block operation performs only 96 floating-point operations when $B = 32$, as compared to the 65536 operations that would be

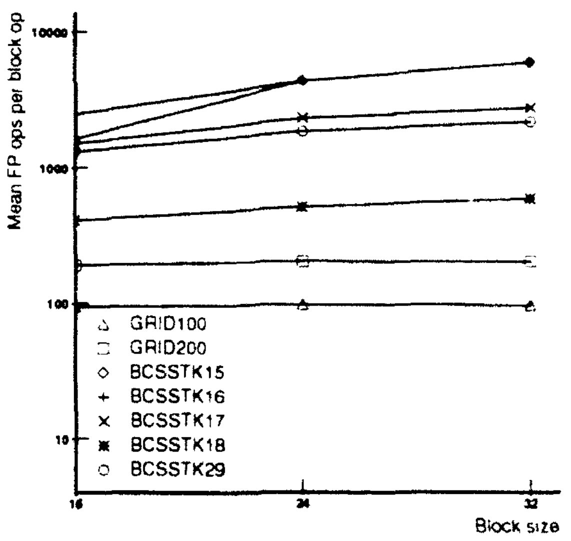


Figure 40: Average floating-point operations per block operation.

performed if all blocks were 32 by 32 full blocks. The average number of floating-point operations per block operation across the whole benchmark set is shown in Figure 40. Note that this figure quite accurately predicts the performance numbers seen in the previous figure.

The primary cause of small blocks in the block decomposition is the presence of small supernodes, and thus small partitions. To increase the size of these partitions, we now consider the use of *supernode amalgamation* [10, 17] techniques. Recall that the basic goal of supernode amalgamation is to find pairs of supernodes that are nearly identical in non-zero structure. By relaxing the restriction that the sparse matrix only store non-zeroes, some zeroes can be introduced into the sparse matrix in order to make the sparsity structures of two supernodes the same. These supernodes can then be merged into one larger supernode. We use the same amalgamation approach for the block approach as we did for the panel approach in a previous chapter.

In Figure 41 we show the average block operation sizes both before and after amalgamation. It is clear that amalgamation significantly increases the block operation grain size.

Before presenting performance comparisons, we first note that amalgamation does have a cost. By introducing zeroes into the sparse matrix, the amount of floating-point work is increased. To be fair, the performance of the block computation after amalgamation should therefore be compared with the performance of the sequential computation before this extra work is introduced. However, amalgamation also provides some benefit for sequential factorization, primarily related to improved use of the processor cache. We found that the benefit in fact outweighed the cost for the amalgamation strategy we employed on all benchmark matrices, with performance improvements ranging from 1% to 14% (see Table 31) for the true sequential method. Block method performance is therefore

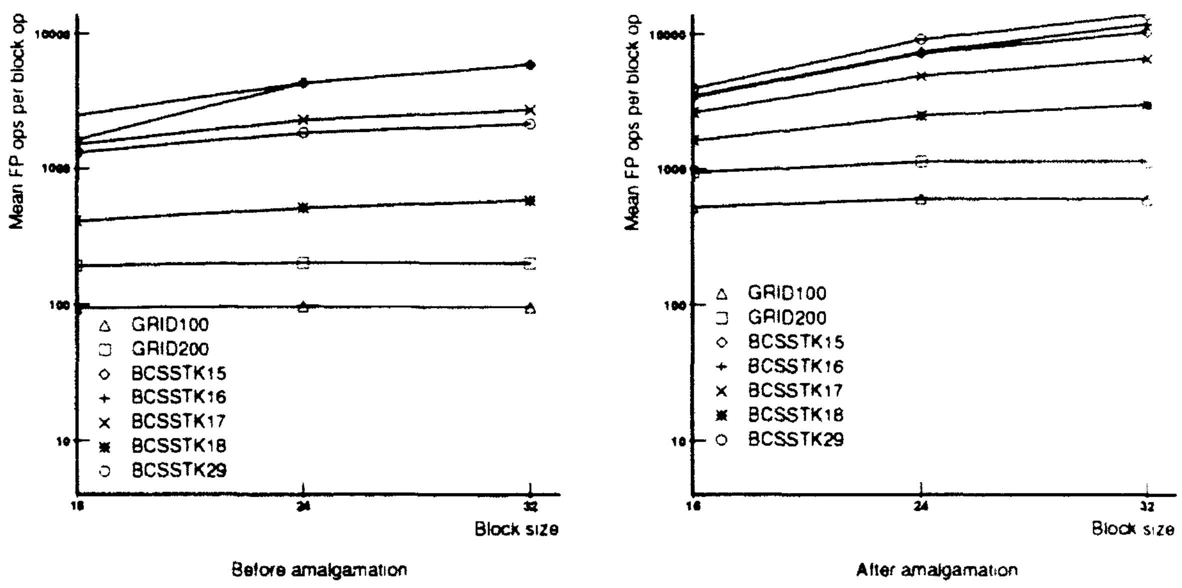


Figure 41: Average floating-point operations per block operation, before and after supernode amalgamation.

compared to the performance of the true sequential method *after* amalgamation.

Figure 42 shows relative performance levels after amalgamation. The results indicate that amalgamation is quite effective at reducing overheads. Performance roughly doubles for GRID100, where the average task grain size increases for $B = 32$ increases from 96 floating-point operations to 597. Performance increases for the other matrices as well. With only two exceptions, block method performance is roughly 80% of that of a true sequential method for $B = 32$. Performance falls off somewhat when $B = 24$, and it decreases further when $B = 16$, but the resulting efficiencies are still more than 70%.

Note that our chosen range of blocks sizes, 16 to 32, is meant to span the range of reasonable

Table 31: Supernode amalgamation results.

	Name	Supernodes		Performance improvement
		before amalgamation	after amalgamation	for true seq. method
1.	GRID100	6,672	2,786	5%
2.	GRID200	26,669	11,243	6%
3.	BCSSTK15	1,295	525	1%
4.	BCSSTK16	691	434	3%
5.	BCSSTK17	2,595	1,622	2%
6.	BCSSTK18	7,438	3,727	7%
7.	BCSSTK29	3,231	1,193	14%

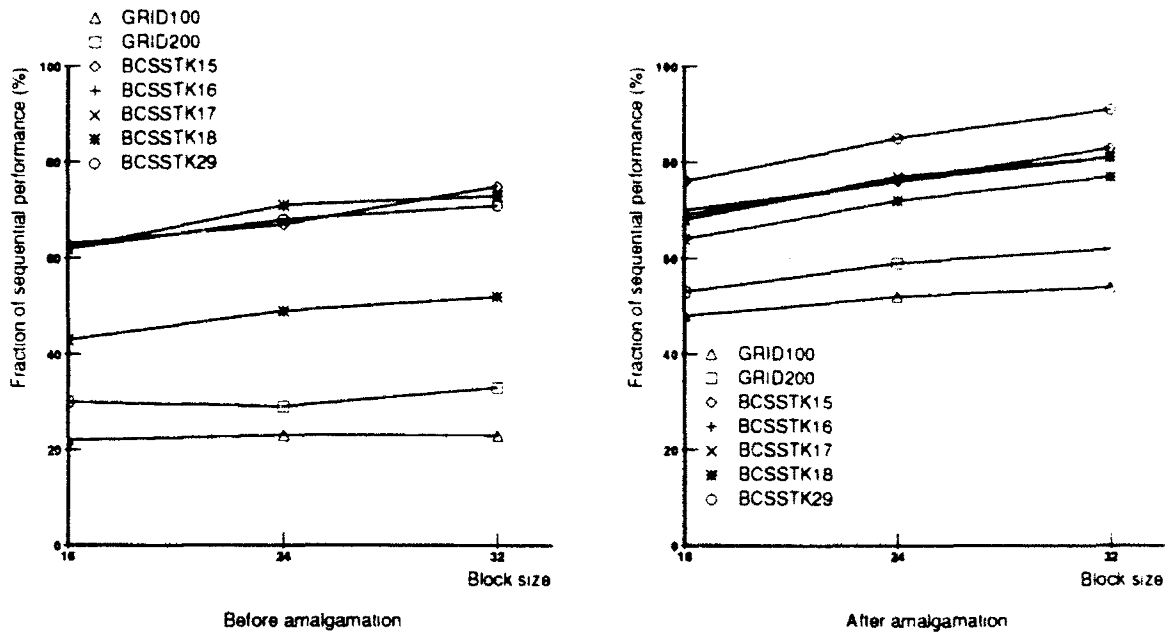


Figure 42: Performance of a sequential block approach, before and after supernode amalgamation, relative to a sequential left-looking supernode-supernode approach.

choices. Blocks that are smaller than 16 by 16 would be expected to lead to large overheads. Indeed, performance was observed to fall off quite quickly for block sizes of less than 16. At the other end of the spectrum, the marginal benefit of increasing the block size beyond 32 by 32 would be expected to be small. This expectation was also confirmed by empirical results.

7.2.5 Block Decomposition Summary

This section has described a simple means of decomposing a sparse matrix into a set of rectangular blocks. The performance of a method based on such blocks on a sequential machine is nearly equal to that of a true sequential method. Of course, our goal here is not an efficient sequential method, but instead an efficient parallel method. The next section will consider several issues related to the parallelization of the above approach.

7.3 Parallel Block Method

The question of how to parallelize the sequential block approach described so far can be divided into two different questions. First, how will processors cooperate to perform the work assigned to them? And second, what method will be used to assign this work to processors? This section will address these two questions in turn.

7.3.1 Parallel Factorization Organization

We begin our description of the parallel computation by assuming that each block will have some specific owner processor. In our approach, the owner of a block L_{IK} performs all block update operations with L_{IK} as their destination. That is, we use a destination-computes approach, an approach that was shown to have significant advantages over a source-computes approach in the previous chapter. With this choice in mind, we present the parallel block fan-out algorithm in Figure 43. The rest of this discussion will be devoted to an explanation of the algorithm.

The most important notion for the block fan-out method is that once a block L_{IK} is *complete*, meaning that it has received all block updates and has been multiplied by the inverse of the diagonal block, then L_{IK} is sent to all processors that could own blocks updated by it. Blocks that could be updated by L_{IK} fall in block-row I or block-column I of L. When a block L_{IK} is received by a processor p (step 2 in Figure 43), processor p performs all related updates to blocks it owns. The block L_{IK} only produces blocks updates when it is paired with blocks in the same column K. Thus, processor p considers all pairings of the received block L_{IK} with completed blocks it has already received in column K (these blocks are held in set $Rec_{K,p}$) to determine whether the corresponding destination block is owned by p (steps 10 and 11). If the destination L_{IJ} is owned by p ($map[L_{IJ}] = p$), then the corresponding update operation is performed (steps 12 and 13). Each processor maintains a hash table of all blocks assigned to it, and the destination block is located through this hash table.

A count is kept with each block ($nmod[L_{IK}]$), indicating the number of block updates that still must be done to that block. When the count reaches zero, then block L_{IK} is ready to be multiplied by the inverse of L_{KK} (step 20 if L_{KK} has already arrived at p; step 6 otherwise). A diagonal block L_{KK} is kept in $Diag_{K,p}$, and any blocks waiting to be modified by the diagonal block are kept in $Wait_{K,p}$. The sets $Diag$, $Wait$, and Rec can be kept as simple linked lists of blocks.

One issue that is not addressed in the above pseudo-code is that of block disposal. As described above, the parallel algorithm would retain a received block for the duration of the factorization. To determine when a block can be thrown out, we keep a count $ToRec_{K,p}$ of the number of blocks in a column K that will be received be a processor p. Once $|Rec_{K,p}| = ToRec_{K,p}$, then all blocks in column K are discarded.

We note that a small simplification has been made in steps 11 through 14 above. For all blocks L_{IJ}, I must be greater than J, a condition that is not necessarily true in the pseudo-code. The reader should assume that I is actually the larger of I and J, and similarly that J is the smaller of the two.

```
1.  while some $L_{IJ}$ with $map[L_{IJ}] = MyID$ is not complete do
2.      receive some $L_{IK}$
3.      if $I = K$ /* diagonal block */
4.          $Diag_{K,MyID} \leftarrow L_{KK}$
5.          foreach $L_{JK} \in Wait_{K,MyID}$ do
6.              $L_{JK} \leftarrow L_{JK} L_{KK}^{-1}$
7.              send $L_{JK}$ to all P that could own blocks in
                        row $J$ or column $J$
8.      else
9.          $Rec_{K,MyID} \leftarrow Rec_{K,MyID} \cup \{L_{IK}\}$
10.         foreach $L_{JK} \in Rec_{K,MyID}$ do
11.             if $map[L_{IJ}] = MyID$ then
12.                 Find $L_{IJ}$
13.                 $L_{IJ} \leftarrow L_{IJ} - L_{IK} L_{JK}^T$
14.                 $nmod[L_{IJ}] \leftarrow nmod[L_{IJ}] - 1$
15.                 if $(nmod[L_{IJ}] = 0)$ then
16.                     if $I = J$ then /* diagonal block */
17.                         $L_{JJ} \leftarrow$ Factor($L_{JJ}$)
18.                         send $L_{JJ}$ to all P that could own blocks in
                                    column $J$
19.                     else if $(Diag_{J,MyID} \neq \emptyset)$ then
20.                         $L_{IJ} \leftarrow L_{IJ} L_{JJ}^{-1}$
21.                         send $L_{IJ}$ to all P that could own blocks in
                                    row $I$ or column $I$
22.                     else
23.                         $Wait_{J,MyID} \leftarrow Wait_{J,MyID} \cup \{L_{IJ}\}$
```

Figure 43: Parallel block fan-out algorithm.

7.3.2 Block Mapping for Reduced Communication

We now consider the issue of mapping blocks to processors. Our general approach is identical to the approach we used for dense matrices. We assume that the processors are arranged in a $p \times p$ 2-D grid configuration, with the bottom left processor labeled $P_{0,0}$, and the upper right processor labeled $P_{p-1,p-1}$. To limit communication, a row of blocks is mapped to a row of processors. Similarly, a column of blocks is mapped to a column of processors. We choose round-robin distributions for both the rows and columns, where

$$map[L_{IJ}] = P_{I \bmod p, J \bmod p}.$$

Other distributions could be used. By performing the block mapping in this way, a block L_{IK} in the sparse factorization need only be sent to the row of processors that could own blocks in row I and the column of processors that could own blocks in column I. Every block in the matrix would thus be sent to a total of $2p = 2\sqrt{P}$ processors. Note that communication volume is independent of the block size with this mapping; every block in the matrix is simply sent to $2\sqrt{P}$ processors.

Recall from the previous chapter that this block mapping strategy is appealing not only because it reduces communication volume, but also because it produces an extremely simple and regular communication pattern. All communication is done through multicasts along rows and columns of processors. This pattern is simple enough that one might reasonably expect parallel machines with 2-D grid interconnection networks to provide hardware multicast support for it eventually. In the absence of hardware support, an efficient software multicast scheme can be used. We will return to this issue later in this chapter.

7.3.3 Enhancement: Domains

Before presenting performance results for the block fan-out approach, we first note that the method as described above produces more interprocessor communication than competing panel-based approaches for small parallel machines. This is despite the fact that the block approach has much better asymptotic communication behavior. To understand the reason, consider a simple 2-D $k \times k$ grid problem. The corresponding factor matrix contains $O(k^2 \log k)$ non-zeroes, and the parallel factorization of this matrix using a panel approach can be shown to generate $O(k^2 P)$ communication volume [24]. In the block approach, every non-zero in the matrix is sent $O(\sqrt{P})$ processors, so the total communication volume grows as $O((k^2 \log k)\sqrt{P})$. While the communication in the block approach grows less quickly in P, for any given 'k' it also has a larger 'constant' in front.

Recall that an important technique for reducing communication in panel methods was the use of *owned domains* [4, 9]. Domains are large sets of columns in the sparse matrix (corresponding to subtrees of the elimination tree of L) that are assigned en masse to a single processor. By assigning the columns of an entire subtree to a single processor, these columns can be factored without any interprocessor communication, and the updates from all columns in a domain to subsequent columns can also be computed without communication. Ashcraft suggested [4] that domains can be incorporated into a block approach as well. The basic approach is as follows. The non-zeroes within a domain are stored as they would be in a column-oriented method. The domain factorization is then performed using a column method. The aggregate domain updates to ancestor columns are computed column-wise as well. We use an efficient left-looking supernode-supernode method for both. Once the aggregate updates have been computed, they are sent out in a block-wise fashion to the appropriate destination blocks.

Note that one benefit of these domains is that they reduce the number of small blocks in the matrix, and thus they reduce related overheads. Recall that small supernodes are the main source of small blocks. In a sparse problem, most small supernodes lie towards the leafs of the elimination tree, where they are likely to be contained within domains.

One problem with the above approach to owned domains is that it introduces a 'seam' in the block-oriented computation. The matrix is stored as columns within domains and as blocks outside the domains. This seam can be avoided if the domain non-zeroes are still kept as blocks. Aggregation of updates to ancestor blocks can be accomplished by creating 'shadow blocks' for all affected ancestor blocks. The shadow blocks would have the same non-zero structures as the blocks they represent, but they would be initialized to have all zero entries. The domain factorization would then be handled in a block-oriented manner. Once a domain is complete, a shadow block would contain the aggregate update from the domain to the corresponding destination block. The shadow blocks could then be sent to the processors that own the corresponding real blocks, to be added as updates. This approach produces a much cleaner although slightly less efficient factorization code. We will prefer efficiency to elegance in this chapter, however. Performance results will come from a code that stores owned domains as columns.

Of course, the owned domains must be carefully assigned to processors to avoid having some processors sit idle, waiting for other processors to complete local domain computations. Geist and Ng [20] described an algorithm for assigning a small set of domains to each processor so that the amount of domain work assigned to each processor is evenly balanced. They considered domains in the context of column-oriented parallel methods, but their approach also applies for a block-oriented approach. All results from this point on use the algorithm of Geist and Ng to produce domains.

With the introduction of domains, the parallel computation thus becomes a three phase process. In the first phase, the processors factor their owned domains and compute the updates from these domains to blocks outside the domains. In the second phase, the updates are sent to the processors that own the corresponding destination blocks and are added into their destinations. Finally, the third phase performs the block factorization, where blocks are exchanged between processors. Note that these are only logical phases; no global synchronizations is necessary between the phases.

Consider the effect of domains on communication volume in a block method for a 2-D grid problem. We first note that the number of non-zeroes not belonging to domains in the sparse matrix can be shown to grow as $O(k^2 \log P)$, versus $O(k^2 \log k)$ without domains. Total communication volume for these non-zeroes using a block approach is thus $O(k^2 \sqrt{P} \log P)$. The other component of communication volume when using domains is the cost of sending domain updates to their destinations. The total size of all such updates can be shown to be $O(k^2)$, independent of P, so domain update communication represents a lower-order term. Total communication for a 2-D grid problem is thus $O(k^2 \sqrt{P} \log P)$.

We should note that this communication figure is not optimal for block-oriented factorization. In fact, communication volumes can be reduced to $O(k^2 \sqrt[3]{P})$ through the use of a fan-both approach [6]. However, it is not at all clear that these improved communication figures can be obtained in a simple, practical method.

7.4 Evaluation

This section evaluates the parallel block fan-out approach proposed in the previous section. The approach is evaluated in three different contexts. First, we look at performance on a small-scale multiprocessor. Then, we consider performance on moderately-parallel machines (up to 64 processors), using our multiprocessor simulation model and the Stanford DASH machine. Finally, we consider issues for more massively parallel machines.

7.4.1 Small Parallel Machines

The first performance numbers we present come from the Silicon Graphics SGI 4D/380 multiprocessor. Parallel speedups are shown in Figure 44 for 1 through 8 processors. All speedups are computed relative to a left-looking supernode-supernode sequential code, the sequential code that gave the best overall performance. The figure shows that the block fan-out method is indeed quite efficient for small machines. In fact, we have found that performance is higher than that of a panel method on this machine, due to better load balance. Recall that the static task mapping scheme that is used in a panel method causes some load imbalances. The block method assigns sufficiently many blocks to each processor so that imbalances are small. We also note that the performance of the block method is comparable to that of a highly efficient shared-memory panel code [40, 41] that dynamically doles out tasks to processors and thus does not have load imbalance problems.

Speedups for the block method on 8 processors are roughly 5.5-fold, corresponding to absolute performance levels of between 45 and 50 double-precision MFLOPS. Speedups are less than linear in the number of processors for two simple reasons. First, the block method is slightly less efficient than a column method. We believe this accounts for a roughly 20% performance reduction. Second, the block method still produces some load imbalance. Program instrumentation reveals that processors spend roughly 15% of the computation on average sitting idle. These two factors combine to give a relatively accurate performance prediction.

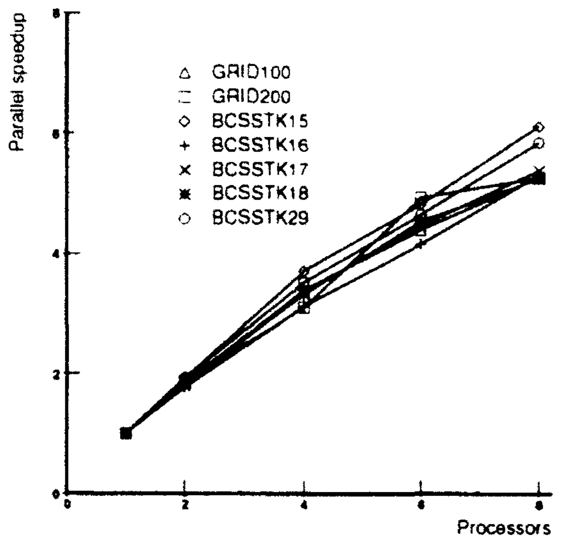


Figure 44: Parallel speedups for block fan-out method on SGI 4D-280, $B = 24$.

7.4.2 Moderately Parallel Machines

We now evaluate the parallel performance of the block fan-out approach on machines with up to 64 processors, using both the multiprocessor simulation model described earlier and also the Stanford DASH machine. We also discuss issues of communication volume.

Simulated Performance

To get a feel for how a block approach performs on larger parallel machines, Figure 45 shows simulated processor utilization levels for between 4 and 64 simulated processors, using a block size of 24. It is clear from the figure that the block approach exhibits less than ideal behavior as the machine size is increased. On 64 processors, for example, utilization levels drop to roughly 40%. Further investigation reveals that the primary cause of the drop in performance is a progressive decline in the quality of the load balance. Figure 46 compares simulated performance for matrices BCSSTK15 and BCSSTK29 with the best performance that could be obtained with the same block distribution. The load balance performance bound is identical to the maximum load bound that we used for panel methods; it is obtained by computing the runtime that would be required if there were no dependencies between blocks and if interprocessor communication were free. The difference here is that the other component of the maximum load bound, load efficiency, is unimportant since the vast majority of the computation makes good use of the cache and thus obtains near-perfect efficiency.

The quality of the load distribution clearly depends on the method used to map blocks to

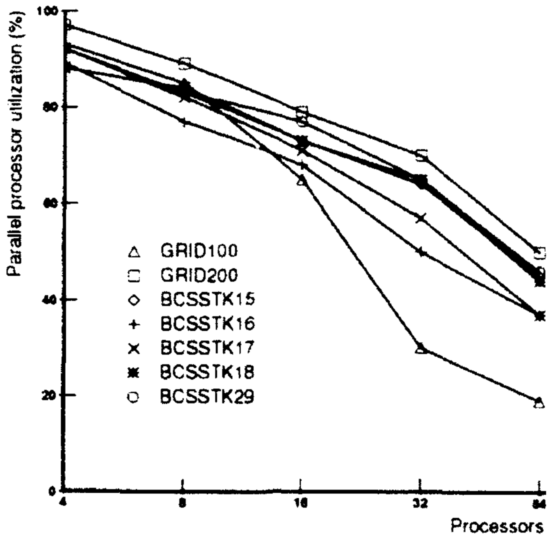
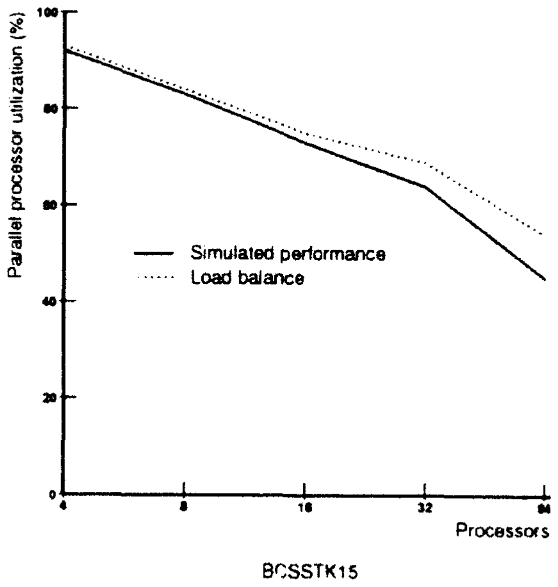


Figure 45: Simulated parallel efficiencies for block fan-out method, $B = 24$.



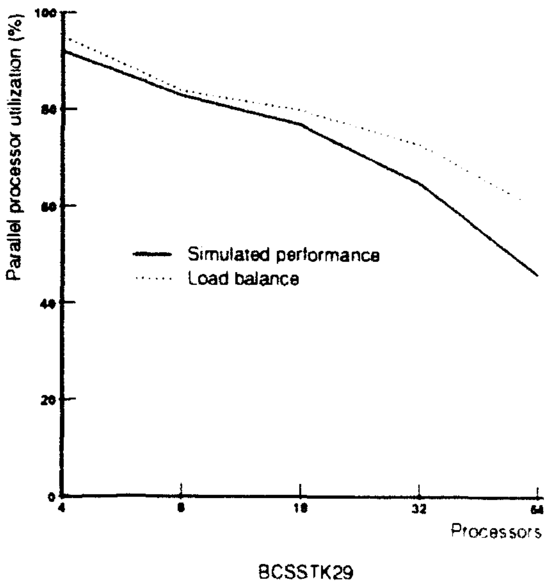
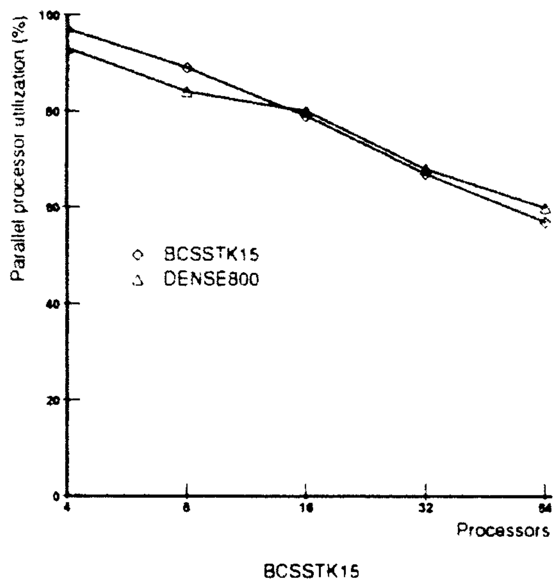


Figure 46: Simulated parallel performance, compared with load balance upper bound ($B = 24$).



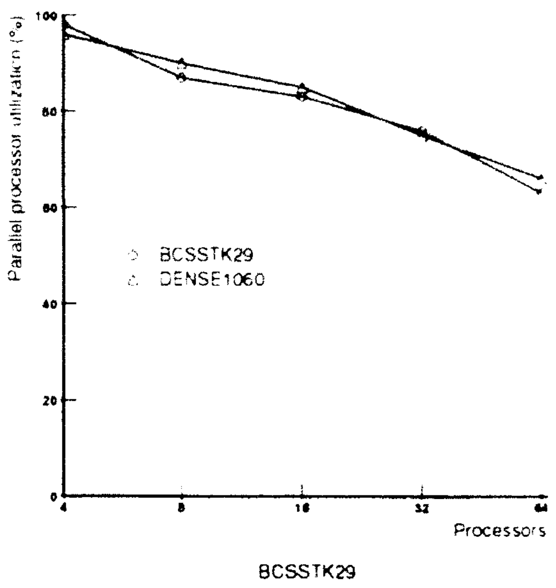


Figure 47: Parallel utilization upper bounds due to load balance for BCSSTK15 and BCSSTK29, compared with load balance upper bounds for dense problems ($B = 24$). In both plots, sparse and dense problems perform the same number of floating-point operations.

processors. Recall that we use a very rigid mapping strategy, where block L_{IJ} is assigned to processor $P_{I \bmod p, J \bmod p}$. One possible explanation for the poor behavior of this strategy is that it does not adapt to the structure of the sparse matrix; it tries to impose a very regular structure on a matrix that is potentially comprised of a very irregular arrangement of non-zero blocks.

While the mismatch between the regular mapping and the irregular matrix structure certainly contributes to the poor load balance, it is our belief that a more important factor is the wide variability in task sizes. In particular, since a block is modified by some set of blocks to its left, blocks to the far right in the matrix generally require much more work than blocks to the left (more accurately, blocks near the top of the elimination tree require more work than blocks near the leafs). Furthermore, since the matrix is lower-triangular, the number of blocks in a column decreases towards the right. The result is a small number of very important blocks in the bottom-right corner of the matrix.

To support our contention that the sparse structure of the matrix is less important than the more general task distribution problem, Figure 47 compares the quality of the load balance obtained for two sparse matrices, BCSSTK15 and BCSSTK29, to the load balance obtained using the same mapping strategy for a dense matrix. The curves show the maximum obtainable processor utilization levels given the block mapping. The dense problems are chosen so as to perform roughly the same number of floating-point operations as the two sparse problems.

Note that the load balance can be improved by moving to a smaller block size, thus creating more distributable blocks and making the block distribution problem easier. However, as discussed earlier, smaller blocks also increase block overheads. For the larger benchmark sparse matrices, decreasing the block size from $B = 24$ to $B = 16$ increases simulated parallel efficiencies by roughly 15% for $P = 64$. A block size of less than 16 further improves the load balance but achieves lower performance due to overhead issues.

The general conclusion to be drawn from these simulation results is simply that it is difficult to achieve high processor utilization levels on large machines using relatively small problems. Possible avenues to explore in order to improve performance include the use of a more dynamic task assignment strategy or a more general function for mapping blocks to processors. This matter will require further investigation.

Communication Volume

So far, our analysis has assumed that parallel performance is governed by two costs: the costs of executing block operations on individual processors and the latencies of communicating blocks between processors. Another important, although less easily modelled component of parallel performance is the total interprocessor communication volume. Communication volume will determine the amount of contention that is seen on the interconnection network. Such contention can have severe performance consequences, and can in many cases govern the performance of the entire computation (see [43], for example).

Rather than try to integrate these costs into our simple performance model, we instead look at interprocessor communication in a more qualitative way. To obtain a general idea of how much communication is performed, Figure 48 compares total interprocessor communication volume with total floating-point operation counts for a variety of sparse matrices and machine sizes. This figure shows the average number of floating-point values sent by a processor divided by the number of floating-point operations performed by that processor. Sustainable values will of course depend on the relative computation and communication bandwidths of the processor and the processor interconnect in the parallel machine. Current machines would most likely not have trouble supporting the 0.025 ratio (40 FP ops per word of communication) seen for 16 processors on these matrices. The 0.05 ratio (20 FP ops per work of communication) on 64 processors would be more difficult to support.

Real Machine Performance

Let use now consider how these simulation numbers translate into achieved performance on the Stanford DASH machine. We first compare predicted speedups with achieved speedups for matrices BCSSTK15 and BCSSTK29 in Figure 49. The block size for both is 24. The figure shows that DASH performance is significantly below simulated performance. The main reason is that communication costs are assumed to be hidden from the processors in the simulation, while they are not hidden

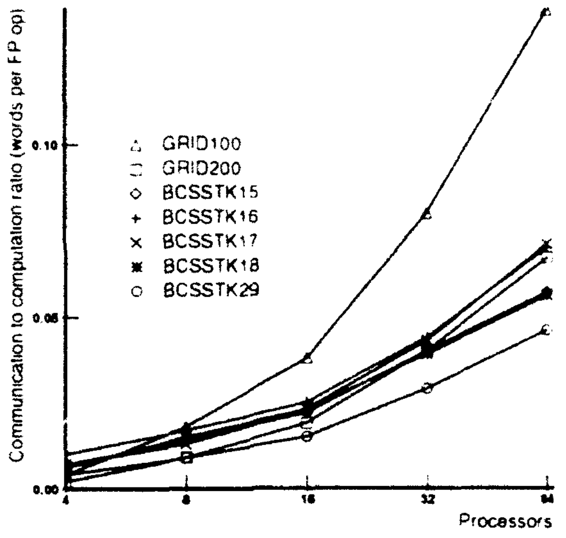


Figure 48: Communication versus computation for the block fan-out method

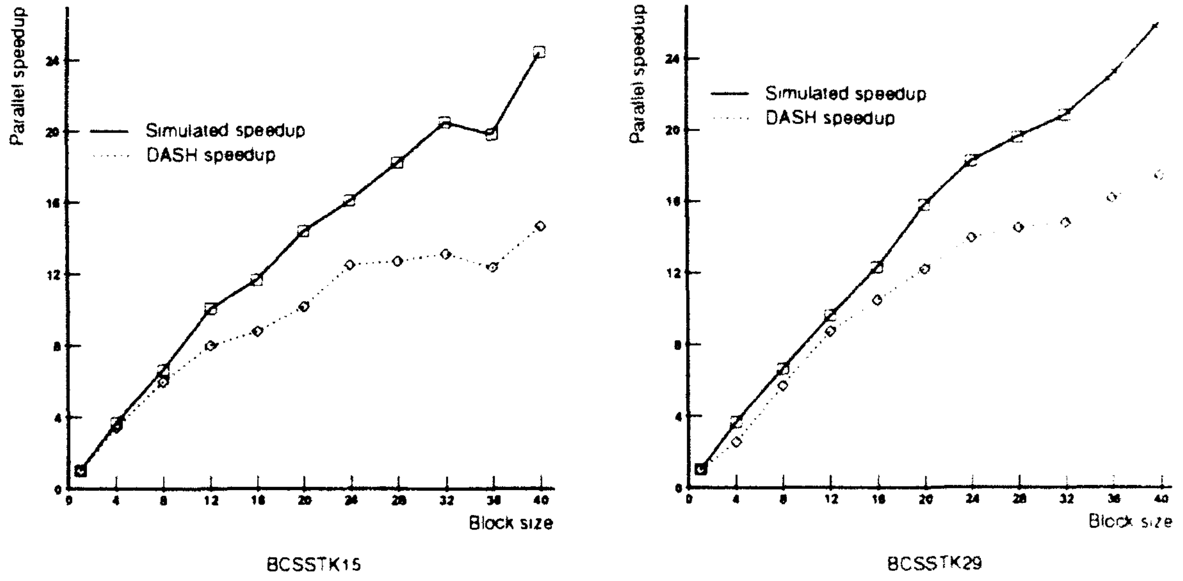


Figure 49: Parallel speedups for block approach for BCSSTK15 and BCSSTK29

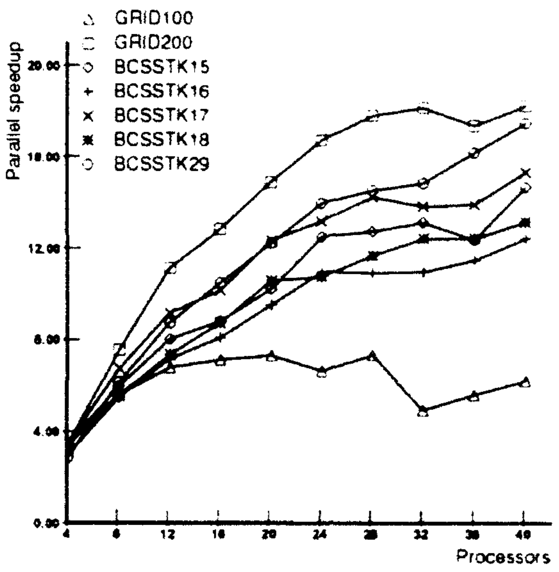


Figure 50: Parallel speedups for block approach on the Stanford DASH machine

in the DASH machine. The cost of this lack of latency hiding is substantial. On 40 processors, for example, the processors perform roughly one word of communication for every 20 floating-point operations for both matrices. This represents a substantial cost to the processors, since a word of communication costs roughly 50 cycles while a floating-point operation costs less than 4 cycles. These communication costs do not account for the entire difference between simulated and achieved performance. They do account for the majority of it, though.

Looking at parallel speedups across a wider range of sparse problems gives the results in Figure 50. For each data point, we report maximum speedups when using a block size of either 24 or 32. A choice of 32 typically gave better results for fewer than 32 processors, while a block size of 24 was better for 32 or more. In either case, the performance differences between the two choices were generally less than 10%.

For reasons discussed earlier in this section, the obtained parallel performance is relatively low, with speedups on 40 processors ranging from 12 to 18.

Comparison with Panel Method

To put the results for the block-oriented method into better perspective, we now compare them to the corresponding results for a panel method. Figure 51 shows relative communication volume. Interestingly, the block approach provides few communication-volume benefits on 64 processors. While the growth rates, $O(P)$ for panels and $O(\sqrt{P}\log P)$ for blocks, favor the block approach, constants make these rates less relevant for small P.

An interesting thing to note here is that relative communication is quite a bit higher for the two

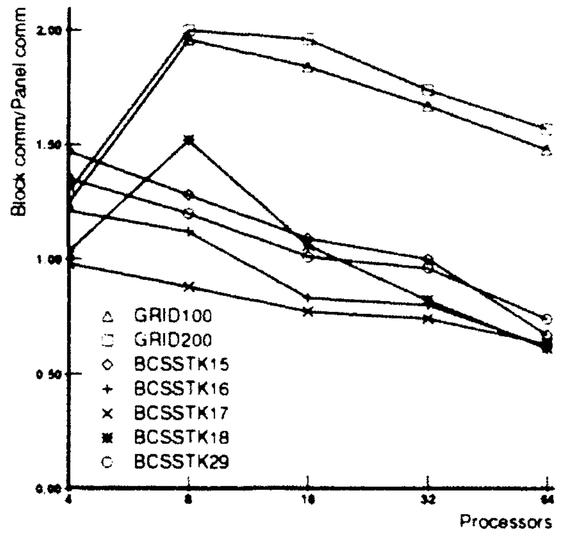


Figure 51: Communication volume of block approach, relative to a panel-oriented parallel multifrontal approach.

grid problems than for the other matrices. The reason is that the column multifrontal approach does very well communication-wise for sparse matrices whose elimination trees have few nodes towards the root and instead quickly branch out into several independent subtrees. The two grid problems have this property. The block approach derives no special benefit from this property.

Figure 52 compares block performance to panel performance, using both the DASH machine and multiprocessor simulation. The figure shows that the block approach does indeed provide higher performance on moderately parallel machines than the panel approach. The simulation predicts performance improvements of roughly 50% on 40 processors, while improvements of between 10% and 40% percent are observed on the DASH machine. We believe that the reason performance differences are larger in the simulation is again because communication costs are not being hidden on the DASH machine. We previously indicated that these costs are substantial for large numbers of processors, and we also showed that the costs were comparable for the panel and block methods. Since this large communication cost is shared between the two methods, the performance differences between the two are decreased.

Summary

To summarize this subsection, we note that our block fan-out approach provides good performance for moderately-parallel machines, although parallel speedups are well below linear in the number of processors for the matrices we have considered. An important limiting factor is the load balance that results from our quite rigid cookie-cutter block distribution scheme. We also find that the block

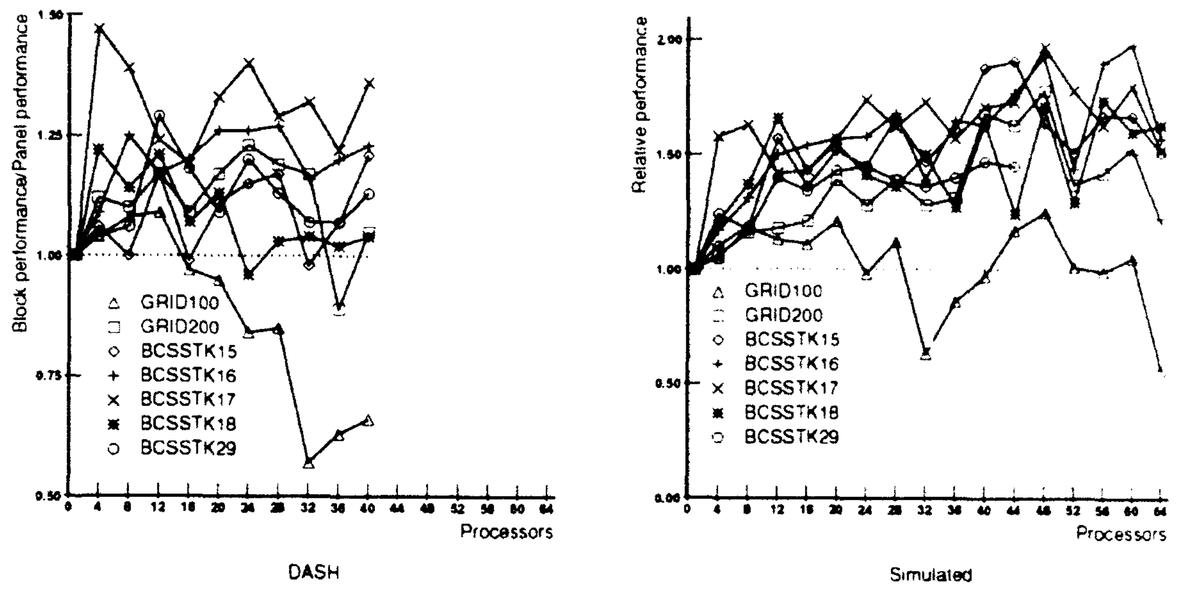


Figure 52: Performance of block approach relative to performance of panel approach.

approach produces comparable amounts of interprocessor communication traffic to a panel approach on 64 or fewer processors. Comparing the overall performance of a block approach to that of a panel approach, we find that the block approach has a small performance advantage. The load balance problems with the block approach are more than made up for by its better data reuse; the block approach performs virtually all computation in the form of B by B dense matrix multiplications, whereas the panel approach is forced to use narrow panels. The performance advantage of the block approach over the panel approach would be expected to be somewhat larger on a machine that hides interprocessor communication latencies from the processors. On the DASH machine, since communication volumes are comparable for both methods and the costs of such communication are considerable, the differences between the methods due to other sources are diluted.

7.4.3 Massively-Parallel Machines

Having concentrated on issues of efficiency on smaller machines in the first part of this section, we now turn our attention to three issues that will be important for very large parallel machines. First, we look at available concurrency in the problem. In other words, we look at how many processors can be productively used for a particular problem. Next we turn to the issue of per-processor storage requirements, and we consider how they grow as the number of processors and the problem size is increased. A common assumption for large parallel machines is that each processor will contain some constant amount of memory. Thus, it would be desirable for the amount of storage required per processor to remain constant. Finally, we consider interprocessor communication issues. Our discussions will use 2-D grid problems as examples.

Before further discussing these issues, we should first explain our goals. The primary advantage of a block approach over a panel approach for a massively parallel machine is that it exposes more concurrency and thus allows more processors to cooperate for the same sparse problem. For a $k \times k$ 2-D grid problem, for example, the column approach can be shown to allow $O(k)$ processors to participate. By some measures, a block approach can use $O(k^2)$. Our goal is to determine whether the use of $O(k^2)$ processors is a realistic goal, and to understand the difficulties that might be encountered in trying to reach this goal.

Concurrency

One important bound on the parallel performance of a computation is the length of the critical path. Determining the critical path in a computation requires an analysis of the dependencies between the various tasks in that computation. Such an analysis for block-oriented sparse Cholesky factorization reveals that the length of the critical path is proportional to the height of the elimination tree, assuming some constant block size. For a 2-D grid problem, the elimination tree can be shown to have height $3k$. Thus, in the best case the $O(k^3)$ work of the entire factorization can be performed in $O(k)$ time. Consequently, at most $O(k^2)$ processors can be productively applied to this problem. This figure is consistent with our goals for the block approach.

Storage

We now look at the issue of how per-processor storage requirements grow as the size of the machine and the size of the problem is increased. We first note the obvious fact that the processor must store the portion of the matrix assigned to it. If the factorization is performed on P processors and the problem being factored is a $k \times k$ grid problem, then each processor must store $O(\frac{k^2 \log k}{P})$ nonzeroes. Keeping per-processor storage requirements constant would thus require that the number of processors grow slightly faster than k^2. Since the critical path analysis showed that only $O(k^2)$ processors can be used productively for this problem, we must resign ourselves to a slow growth rate in per-processor storage.

Now consider the storage requirements of the auxiliary data structures that a processor must maintain. One important set of auxiliary data is the per-block information. An example is the count of how many times a block is modified. Another is the particular row and column of processors to which a particular block is sent when complete. This data adds a small constant to the size of each block, and consequently it represents a small constant factor increase in overall storage.

Another important set of auxiliary data is the column-wise data. One example is the arrival count information, which keeps track of how many blocks in a particular block-column a processor will receive. Since the number of block-columns in the matrix is k^2, this data structure would occupy $O(k^2)$ space per processor if every entry were kept. Fortunately, only $O(k^3/P)$ of these entries must be stored. The reason is as follows. If the factorization work is distributed evenly among the processors, then the work performed per processor is $O(k^3/P)$. Since a received block is only retained in a processor if it participates in some useful work, clearly the number of such retained blocks and thus the number of arrival counts that must be stored is also $O(k^3/P)$. We can keep a hash table, indexed by column number, of all non-zero arrival counts. When a block arrives, the corresponding arrival count is located and decremented. Note that not all blocks that arrive at a processor participate in an update on that processor. If no arrival count is found for the block column of an arriving block, then the block is immediately discarded. Similar hash structures can be used for the other column-wise data structures.

Regarding per-processor storage growth rates, note that if P grows as k^2, then the per-processor matrix storage costs grow as $O(\log k)$ while the arrival count storage costs grows as $O(k^3/P) = O(k)$. Fortunately, the $O(k)$ term has a very small constant in front of it, so this term will not be particularly constraining for practical P. However, asymptotic per-processor storage requirements will grow with P.

Communication

A crucial determinant of performance on massively parallel machines is the bandwidth of the processor interconnection network. In order to obtain a rough feel for whether the bandwidth demands of the block fan-out method are sustainable as the machine size increases, we look at these demands in relation to two common upper bounds on available communication bandwidth, in a manner similar to that used by Schreiber in [43]. The two upper bounds are based on bisection bandwidth and total available point-to-point bandwidth in the multiprocessor. We consider a 2-D mesh machine organization, which is in some sense a worst case since it offers lower connectivity than most alternative organizations.

A bisection bandwidth bound is obtained by breaking some set of point-to-point interconnection links in the parallel machine to divide it into two halves. Clearly, all communication between processors in different halves must be travel on one of the links that is split. The bisection bandwidth bound simply states that the parallel runtime is at least as large as the time that would be required for these bisection links to transmit all messages that cross the bisector.

In the case of the block fan-out method applied to a 2-D grid problem, recall that $O(k^2 \log P)$ messages are sent, and each is multicast to $O(\sqrt{P})$ processors (a row and column of processors). Figure 53 shows an example mesh of processors, an example bisector, and the communication pattern that can be used to multicast a message. For any simple bisector, a multicast to a row and column of processors crosses that bisector once or twice. Thus, total traffic across the bisector is $O(k^2 \log P)$. This traffic must travel on one of $O(\sqrt{P})$ communication links in the bisector, and this communication occurs in the $O(k^3/P)$ time required for the factorization. If we assume that communication is evenly distributed among the bisector links, then communication per bisector link per unit time is $O(\frac{k^2 \log P}{\sqrt{P}(k^3/P)}) = O(\frac{\sqrt{P} \log P}{k})$. If P grows as k^2, communication per link per unit time

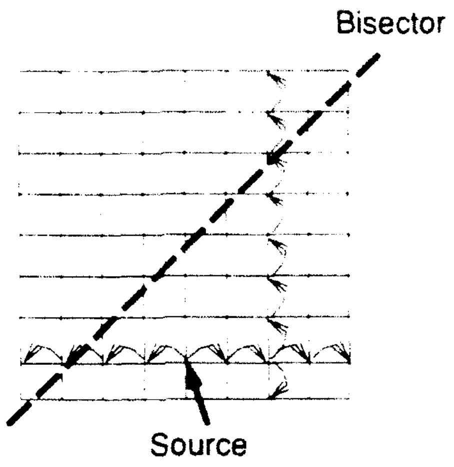


Figure 53: Communication pattern for row/column multicast.

is thus $O(\log P)$. Since the amount of data that can travel on a single link per unit time is constant, this growth rate represents a small problem. The number of processors P must grow slightly slower than k^2 in order to keep message volume per link constant.

Another common communication-based bound on parallel performance is the total amount of traffic that appears on any link in the machine, expressed as a fraction of the total number of links in the machine. For our example, there are $O(k^2 \log P)$ multicasts, each of which traverses $O(\sqrt{P})$ links. The number of links in the machine is $O(P)$, and again this communication occurs in $O(k^3/P)$ time. Thus, global traffic per link per time unit is $O(\frac{k^2\sqrt{P}\log P}{(k^3/P)(P)})$, or $O(\frac{\sqrt{P}\log P}{k})$. If P is $O(k^2)$, we obtain $O(\log P)$ traffic per link, which is identical to the bisector traffic.

We should note that the preceding arguments have said nothing about *achieved* performance. Demonstrating that certain performance levels can actually be achieved would require a detailed analysis of the structure of the sparse matrix, the way in which the factorization tasks are mapped to processors, and the order in which these tasks are handled by their owners. This would certainly be a daunting task. This discussion has simply shown that the approach is not constrained away from achieving high performance by any of the most common performance bounds.

7.4.4 Summary

To summarize our evaluation, we have found that the block fan-out method is quite appealing across a range of parallel machines. Overheads are low enough that the method is quite effective for small parallel machines. It is also effective for moderately parallel machines, although performance is somewhat limited by the quality of the computational load balance. For massively parallel machines, we found that the approach is not perfect. Per-processor storage requirements grow with the number of processors. Bisection bandwidth considerations also limit the number of processors to below ideal. However, these constraints are mild enough that the block fan-out approach appears to be quite practical even for very large P.

7.5 Discussion

At this point in this chapter, it would be desirable to choose a particular parallel method as being preferable to the other. The previous section provided some comparative information, but it did not address several more general and more practical considerations. Let us now consider some of these issues.

The first thing to note is that the block approach has huge asymptotic advantages over a panel approach for larger numbers of processors. The concurrency and communication growth rates so greatly favor the block approach that there is no question that it will eventually provide much higher performance. We therefore concentrate on issues that will be important for moderately parallel machines.

One important advantage of a block fan-out approach is its very regular communication pattern. Blocks are multicast to a row and column of processors. In contrast, the multifrontal panel approach multicasts a panel to an arbitrary subset of the processors. The block communication pattern is certainly easier to perform efficiently.

Another advantage of the block approach is its extremely simple and efficient computational kernel. High performance for this method simply requires an efficient dense matrix-matrix multiplication kernel.

One disadvantage of the block approach is the difficulty of balancing the computational load. While the panel approach did have some load balance problems, they were not nearly as severe.

Another potential disadvantage of a block approach is the less natural data representation it uses. Sparse matrices are decidely much easier to represent in terms of columns (or rows) of non-zeroes. Our hope is that the data representation in the Cholesky factorization routines can be hidden from the application by encapsulating the parallel factorization as a library routine that is accessed through high-level data manipulation routines. Since sparse Cholesky factorization is typically used to solve sparse linear systems, the output of the factorization would be a vector x such that $Ax = b$. The application would hopefully never have to access the factor matrix.

7.6 Future Work

While this chapter has explored several practical issues related to parallel block-oriented factorization, it also has brought up a number of questions that will require further investigation. Foremost among these is the question of whether the load balance could be significantly improved. We are currently investigating more flexible block mapping strategies.

Another interesting question concerns the choice of partitions for the 2-D decomposition. Recall that our partitions are chosen to contain sets of contiguous columns from within the same supernode. Ashcraft has shown [4] that by choosing columns that are not necessarily contiguous, it is often possible to divide the sparse matrix into fewer, denser blocks. While our results indicate that the simpler approach is quite adequate, we are currently looking into the question of how large the benefit of a more sophisticated approach may be.

We also hope to compare the block fan-out approach we have proposed here with the block multifrontal approach proposed by Ashcraft [4]. One thing we are certain of is that the block fan-out method is much less complex. So far, we have not discovered any significant advantages to a multifrontal approach, but the issue requires further study. We also hope to investigate a block analogue of the fan-in method.

Once a matrix A has been factored into the form $A = LL^T$, the next step is typically the solution of one or more triangular systems $Ly = b$, where b is given. An issue that we have left unaddressed in this chapter is the efficiency of this backsolve computation when L is represented as a set of blocks. Our belief is that this backsolve will be more efficient than the backsolve for a column representation, but further investigation will be required to fully answer this question.

Finally, we note that sparse Cholesky factorization requires several pre-processing steps. A block-oriented representation would require new implementations of many of these steps (particularly the symbolic factorization). It will be interesting to see whether it will be possible to perform these steps as efficiently on a parallel machine when using a block framework as opposed to a column framework.

7.7 Related Work and Contributions

Let us now briefly consider how our work in this chapter relates to existing work. One obvious set of related work discusses the use of block-oriented methods for dense matrix computations. The block fan-out method we describe is in many ways a sparse matrix analogue of the parallel dense destination-computes Cholesky factorization method described in [3]. However, it should be clear to the reader that our method represents a non-trivial extension of this previous work. Sparse matrices introduce a variety of complications, including issues of how to decompose the matrix into reasonable blocks, how to determine what blocks are affected by a block, and how to determine when a processor can discard a received block, that are not present in dense methods.

We should also note that we are not the first to suggest the use of a block-oriented formulation for parallel sparse Cholesky factorization. Other formulations have been suggested in [4] and [45]. One crucial difference between our work and this other work is that we have described the details of an extremely practical approach. This other work has described methods that we consider to be too complicated to ever be practical. The other important difference between our work and previous work is that we have produced the first high-performance implementation, and we have done the first detailed performance evaluation. We have demonstrated that a block representation does not severely limit performance, that an efficient parallel method is not all that complicated to implement, and we have provided communication volume and achieved performance comparisons against alternative parallel methods (panel methods). Previously, the only implementation was that of [4], which provided results from a relatively slow parallel machine (an iPSC/2) and provided little comparative information.

One final contribution of our work comes from our extensive use of performance modelling to understand the important factors affecting parallel performance. In doing so, we were able to identify load balance and interprocessor communication costs as important limiting factors, and we were able to quantify the effects of each of these factors.

7.8 Conclusions

The results of previous chapters have shown that panel methods are inappropriate for sparse Cholesky factorization on large parallel machines. This chapter has considered the natural alternative, a 2-D or block matrix decomposition. Our focus has been on answering the question of whether such a decomposition is truly practical. We described a parallel block algorithm that is both practical and appealing. The primary virtues of our approach are: (1) it uses an extremely simple decomposition strategy, in which the matrix is divided using global horizontal and vertical partitions; (2) it is straightforward to implement; (3) it is extremely efficient, performing the vast majority of its work within dense matrix-matrix multiplication operations; (4) it is efficient across a wide range of machine sizes, providing comparable performance to that of efficient panel methods on small parallel machines and better performance on larger machines.

Chapter 8

Conclusions

Machines organizations are continually evolving. Algorithms must evolve as well to make good use of these machines. This thesis has looked at sequential and parallel sparse Cholesky factorization on machines with hierarchical memory organizations, a machine organization that is becoming more and more important.

On sequential machines with a hierarchy of caches and main memory, it is important to reuse data in the faster levels of this hierarchy to avoid the long latencies of cache misses. This thesis performed a careful examination of the performance of three important approaches to the computation, left-looking, right-looking, and multifrontal. Our work is the first to evaluate all these methods in a consistent framework. We showed that each could achieve significant data reuse by exploiting the supernodal structure of the sparse factor. The performance of the methods benefited greatly from such reuse. Roughly three-fold performance improvements were observed for two modern sequential machines on which the evaluations were performed. We also found that when these methods were expressed in terms of supernodes, the performance differences between them effectively disappeared. Conventional wisdom had previously been that the methods were quite dissimilar.

On parallel machines, data reuse is even more crucial. Processors must reuse data not only to avoid the latencies of cache misses, but also to avoid saturation of shared resources, including shared memory modules and the processor interconnection network. We proposed a panel multifrontal method that achieves data reuse by distributing sets of adjacent columns (panels) among processors. We showed that this approach provides two to three times the performance of the existing column multifrontal method, thus demonstrating that our method is quite valuable for improving parallel performance. However, we also found that the method provides relatively low parallel speedups on larger machines. Using performance modelling and parallel machine simulation, we performed the first detailed investigation of the reasons for achieved performance. We demonstrated that this low performance was quite easily understood in terms of simple upper bounds on realizable performance. These methods do not expose sufficient concurrency in sparse factorization problems and they produce too much interprocessor communication volume.

T overcome these problems, we proposed a sparse factorization method that distributes rectangular sub-blocks of the sparse matrix among processors. While such an approach has the potential to be much more complicated than a column approach, the specific method we propose is actually quite simple. It uses an extremely straightforward matrix decomposition, it performs very regular interprocessor communication, and processors maintain simple data structures to determine how to act on received blocks. We demonstrated that this block method has large asymptotic advantages over panel methods for large parallel machines, both in problem concurrency and in interprocessor communication volume. By presenting performance results from the first high-performance block implementation, we also showed that this approach provides higher performance than panel methods even on moderately parallel machines, thus demonstrating that the block method can be implemented efficiently.

Obtaining high performance for the sparse Cholesky factorization computation has historically proven to be an extremely difficult problem, with the only real successes coming from expensive vector supercomputers. This thesis has demonstrated that much less expensive machines, sequential and parallel machines with hierarchical memory organizations, can and do provide high performance if the computation is blocked for the memory hierarchy. This thesis has provided a detailed investigation of the crucial issues for performing this blocking.

Regarding future work, one obvious area for future exploration would be the creation of a scalable library for parallel sparse Cholesky factorization. We believe our work on block-oriented factorization could form the foundation for a 'black box' method that provides good performance on a wide range of parallel machines. One potentially challenging issue for such a library would be the design of the interface between the application program and the sparse system solver. The interface would have to be general enough so that it could be used by a wide range of application programs. It would also have to be high-level enough so that the application program would not have to be intimately familiar with the data representation and data placement done inside the library. At the same time, it would also have to be efficient enough so that passing the matrix between the application and the parallel library would not become the bottleneck in the factorization.

Another potentially interesting area for future work would be an investigation of the use of similar blocking techniques for other sparse matrix methods, such as sparse QR and sparse LU factorization. While the techniques developed in this thesis would not be directly applicable to other sparse problems, it may be possible to apply them with minor modifications to obtain similar benefits.

Another interesting topic is the question of how our work on sparse Cholesky factorization would apply to preconditioned iterative methods that rely on some form of Cholesky factorization for their preconditioning. Important examples include block diagonal factorization and incomplete Cholesky factorization in the conjugate gradient method. It would be interesting to consider the tradeoff between the work required to perform the partial Cholesky factorization, the efficiency of the resulting computation, and the number of iterations the iterative method requires to converge.

Finally, it would be interesting to consider in more detail the implications of our results for an extremely important application, sparse Cholesky factorization, on the design of computer architectures, particularly parallel machine architectures. Machines should certainly be built with an eye toward providing high performance on real programs. Important questions to be answered are: How much memory should each processor have? How many processors can share a single memory? How much interprocessor communication bandwidth should be provided? How large a disparity between processor and memory speeds can be tolerated? Our work on sparse Cholesky can provide important insights for designing the next generation of parallel machines.

Bibliography

[1] Amestoy, P.R., and Duff, I.S., "Vectorization of a multiprocessor multifrontal code", *International Journal of Supercomputer Applications*, 3:41-59, 1989.

[2] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D., "LAPACK: A portable linear algebra library for high-performance computers", *Proceedings of Supercomputing '90*, November, 1990.

[3] Anderson, E., Benzoni, J., Dongarra, J., Moulton, S., Ostrouchov, S., Tourancheau, B., and van de Geijn, R., "LAPACK for distributed memory architectures: progress report", *Parallel Processing for Scientific Computing, Fifth SIAM Conference*, 1991.

[4] Ashcraft, C.C., *The domain/segment partition for the factorization of sparse symmetric positive definite matrices*, Boeing Computer Services Technical Report ECA-TR-148, November, 1990.

[5] Ashcraft, C.C., *The fan-both family of column-based distributed Cholesky factorization algorithms*, in Workshop on Sparse Matrix Computations: Graph Theory Issues and Algorithms, 1992.

[6] Ashcraft, C.C., *A taxonomy of distributed dense LU factorization methods*, Boeing Computer Services Technical Report ECA-TR-161, March, 1991.

[7] Ashcraft, C.C., *A vector implementation of the multifrontal method for large sparse symmetric positive definite linear systems*, Boeing Computer Services Technical Report ETA-TR-51, May, 1987.

[8] Ashcraft, C.C., Eisenstat, S.C., and Liu, J., "A fan-in algorithm for distributed sparse numerical factorization", *SIAM Journal on Scientific and Statistical Computing*, 11(3):593-599, 1990.

[9] Ashcraft, C.C., Eisenstat, S.C., Liu, J.L., and Sherman, A.H., *A comparison of three column-based distributed sparse factorization schemes*, Research Report YALEU/DCS/RR-810, Computer Science Department, Yale University, 1990.

[10] Ashcraft, C.C., and Grimes, R.G., "The influence of relaxed supernode partitions on the multifrontal method", *ACM Transactions on Mathematical Software*, 15:291-309, 1989.

[11] Ashcraft, C.C., Grimes, R.G., Lewis, J.G., Peyton, B.W., and Simon, H.D., "Recent progress in sparse matrix methods for large linear systems", *International Journal of Supercomputer Applications*, 1(4): 10-30, 1987.

[12] Carr, S., and Kennedy, K., "Compiler blockability of numerical algorithms", *Proceedings of Supercomputing '92*, November, 1992.

[13] Davis, H., Goldschmidt, S., and Hennessy, J., "Multiprocessing simulation and tracing using Tango", *Proceedings of the 1991 Internctional Conference on Parallel Processing*, August, 1991.

[14] Dongarra, J., Du Croz, J., Hammarling, S., and Duff, I., "A set of level 3 basic linear algebra subprograms", *ACM Transactions on Mathematical Software*, 16(1): 1-17, 1990.

[15] Dongarra, J.J., and Eisenstat, S.C., "Squeezing the most out of an algorithm in CRAY FORTRAN", *ACM Transactions on Mathematical Software*, 10(3): 219-230, 1984.

[16] Duff, I.S., Grimes, R.G., and Lewis, J.G., "Sparse Matrix Test Problems", *ACM Transactions on Mathematical Software*, 15(1): 1-14, 1989.

[17] Duff, I.S., Reid, J.K., "The multifrontal solution of indefinite sparse symmetric linear equations", *ACM Transactions on Mathematical Software*, 9(3): 302-325, 1983.

[18] Fox, G., et al, *Solving Problems on Concurrent Processors: Volume 1 - General Techniques and Regular Problems*, Prentice Hall, 1988.

[19] Gallivan, K., Jalby, W., Meier, U., and Sameh, A., "Impact of hierarchical memory systems on linear algebra algorithm design", *International Journal of Supercomputer Applications*, 2:12-48, 1988.

[20] Geist, G.A., and Ng, E., *A partitioning strategy for parallel sparse Cholesky factorization*, Technical Report TM-10937, Oak Ridge National Laboratory, 1988.

[21] George, A., Heath, M., Liu, J., and Ng, E., *Solution of sparse positive definite systems on a hypercube*, Technical Report TM-10865, Oak Ridge National Laboratory, 1988.

[22] George, A., Heath, M., Liu, J. and Ng, E., "Sparse Cholesky factorization on a local-memory multiprocessor", *SIAM Journal on Scientific and Statistical Computing*, 9:327-340, 1988.

[23] George, A., and Liu, J., *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.

[24] George, A., Liu, J. and Ng, E., "Communication results for parallel sparse Cholesky factorization on a hypercube", *Parallel Computing*, 10: 287-298, 1989.

[25] George, A., Liu, J., and Ng, E., *User's guide for SPARSPAK: Waterloo sparse linear equations package*, Research Report CS-78-30.

[26] Lam, M., Rothberg, E., and Wolf, M., "The Cache Performance and Optimizations of Blocked Algorithms", *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April, 1991.

[27] Lenoski, D., Laudon, J., Gharachorloo, K., Weber, Wolf-Dietrich, Gupta, A., Hennessy, J Horowitz, M., and Lam, M., "The Stanford DASH multiprocessor", *IEEE Computer*, 23(3):63-79, March, 1992.

[28] Lenoski, D., Laudon, J., Joe, T., Nakahira, D., Stevens, L., Gupta, A., and Hennessy, J., "The DASH prototype: logic overhead and performance", *to appear in IEEE Transactions on Parallel and Distributed Systems*, 1992.

[29] Lewis, J., Peyton, B., and Pothen, A., "A fast algorithm for re-ordering sparse matrices for parallel factorization", *SIAM Journal on Scientific and Statistical Computing*, 10: 1146-1173, 1989.

[30] Liu, J., "Modification of the minimum degree algorithm by multiple elimination", *ACM Transactions on Mathematical Software*, 12(2): 127-148, 1986.

[31] Liu, J., "The multifrontal method and paging in sparse Cholesky factorization", *ACM Transactions on Mathematical Software*, 15(4): 310-325, 1989.

[32] Liu, J., "On the storage requirement of the out-of-core multifrontal method for sparse factorization", *ACM Transactions on Mathematical Software*, 12(4), 1987.

[33] Liu, J., "Reordering sparse matrices for parallel elimination", *Parallel Computing*, 11(1): 73-91, 1989.

[34] Lucas, R. *Solving planar systems of equations of distributed-memory multiprocessors*, PhD thesis, Stanford University, 1988.

[35] Mowry, T., and Gupta, A., "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors", *Journal of Parallel and Distributed Computing*, June, 1991

[36] Ng, E.G., and Peyton, B.W., *A supernodal Cholesky factorization algorithm for shared-memory multiprocessors*, Technical Report ORNL/TM-11814, Oak Ridge National Laboratory, April, 1991.

[37] Pothen, A., and Sun, C., *A distributed multifrontal algorithm using clique trees*. Cornell Theory Center Report CTC91TR72, Cornell University, August, 1991.

[38] Rothberg, E., and Gupta, A., "Efficient Sparse Matrix Factorization on Hierarchical Memory Workstations — Exploiting the Memory Hierarchy", *ACM Transactions on Mathematical Software*, 17(2):313-334, 1991.

[39] Rothberg, E., and Gupta, A., *An evaluation of left-looking, right-looking, and multifrontal approaches to sparse Cholesky factorization on hierarchical-memory machines*. Technical Report STAN-CS-91-1377, Stanford University, 1991. Accepted for publication in the *Internation Journal of High-Speed Computing*.

[40] Rothberg, E., and Gupta, A., "Techniques for improving the performance of sparse matrix factorization on multiprocessor workstations", *Proceedings of Supercomputing '90*, p. 232-243 November, 1990.

[41] Rothberg, E., Gupta, A., Ng, E., and Peyton, B., "Parallel sparse Cholesky factorization algorithms for shared-memory multiprocessor systems", *Proceedings of the Seventh IMACS International Conference on Computer Methods for Partial Differential Equations*, 1992.

[42] Schreiber, R., "A new implementation of sparse Gaussian elimination", *ACM Transactions on Mathematical Software*, 8:256-276, 1982.

[43] Schreiber, R., "Are sparse matrices poisonous to highly parallel machines?", in Workshop on Sparse Matrix Computations: Graph Theory Issues and Algorithms, 1992.

[44] van de Geijn, R., *Massively parallel LINPACK benchmark on the Intel Touchstone Delta and iPSC/860 systems*, Technical Report CS-91-28, University of Texas at Austin, August, 1991

[45] Venugopal, S., and Naik, V.K., "Effects of partitioning and scheduling sparse matrix factorization on communication and load balance", *Proceedings of Supercomputing '91*, November, 1991.

[46] von Eicken, T., Culler, D., Goldstein, S., and Schauser, K., "Active messages: a mechanism for integrated communication and computation", *Proceedings of the 19th International Symposium on Computer Architecture*, May, 1992.

[47] Wolf, M., and Lam, M., "A data locality optimizing algorithm", *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation*, June, 1991.