August 1991

Report No. STAN-CS-91-1377

Also Numbered CSL-TR-91-487

An Evaluation of Left-Looking, Right-Looking and Multifrontal Approaches to Sparse Cholesky Factorization on Hierarchial-Memory Machines

by

Edward Rothberg and Anoop Gupta

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 An Evaluation of Left-Looking, Right-Looking and Multifrontal Approaches to Sparse Cholesky Factorization on Hierarchical-Memory Machines

5. FUNDING NUMBERS
87-K-0828

6. AUTHOR(S)
Edward Rothberg and Anoop Gupta

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Computer Science Dept.
Stanford University
Stanford, CA 94305-2140

8. PERFORMING ORGANIZATION REPORT NUMBER
STAN-CS-91-1377
CSL-TR-91-487

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)

Abstract

In this paper we present a comprehensive analysis of the performance of a variety of sparse Cholesky factorization methods on hierarchical-manor-y machines. We investigate methods that vary along two different axes. Along the first axis, we consider three different high-level approaches to sparse factorization: left-looking, right-looking, and multifrontal. Along the second axis, we consider the implementation of each of these high-level approaches using different sets of primitives. The primitives vary based on the structures they manipulate. One important structure in sparse Cholesky factorization is a single column of the matrix. We first consider primitives that manipulate single columns. These arc the most commonly used primitives for expressing the sparse Cholesky computation. Another important structure is the supemode, a set of columns with identical non-zero structures. We consider sets of primitives that exploit the supemodal structure of the matrix to varying degrees. We find that primitives that manipulate larger structures greatly increase the amount of exploitable data reuse, thus leading to dramatically higher performance on hierarchical-memory machines. We observe performance increases of two to three times when comparing methods based on primitives that make extensive use of the supemodal structure to methods based on primitives that manipulate columns. We also find that the overall approach (left-looking. right-looking, or multifrontal) is less important for performance than the particular set of primitives used to implement the approach.

14. SUBJECT TERMS
Hierarchical-memory machines, sparse Cholesky factorization

15. NUMBER OF PAGES
47

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	

An Evaluation of Left-Looking, Right-Looking, and Multifrontal Approaches to Sparse Cholesky Factorization on Hierarchical-Memory Machines

Edward Rothberg and Anoop Gupta
Department of Computer Science
Stanford University
Stanford, CA 94305

August 13, 1991

Abstract

In this paper we present a comprehensive analysis of the performance of a variety of sparse Cholesky factorization methods on hierarchical-memory machines. We investigate methods that vary along two different axes. Along the first axis, we consider three different high-level approaches to sparse factorization: left-looking, right-looking, and multifrontal. Along the second axis, we consider the implementation of each of these high-level approaches using different sets of primitives. The primitives vary based on the structures they manipulate. One important structure in sparse Cholesky factorization is a single column of the matrix. We first consider primitives that manipulate single columns. These are the most commonly used primitives for expressing the sparse Cholesky computation. Another important structure is the supemode, a set of columns with identical non-zero structures. We consider sets of primitives that exploit the supernodal structure of the matrix to varying degrees. We find that primitives that manipulate larger structures greatly increase the amount of exploitable data reuse, thus leading to dramatically higher performance on hierarchical-memory machines. We observe performance increases of two to three times when comparing methods based on primitives that make extensive use of the supernodal structure to methods based on primitives that manipulate columns. We also find that the overall approach (left-looking, right-looking, or multifrontal) is less important for performance than the particular set of primitives used to implement the approach.

1 Introduction

The Cholesky factorization of a large sparse positive definite matrix is an extremely important computation, arising in a wide variety of problem domains. This paper considers the performance of workstation-class machines on this computation, using several different factorization methods. The performance of such machines has been increasing rapidly, with workstation-class machines now performing dense matrix computations at significant fractions of the performance of vector supercomputers. Such machines have the potential to solve linear algebra problems extremely cost-effectively.

Much research has been done on performing sparse Cholesky factorization efficiently. The performance of the computation is well understood for machines with simple memory systems (see, for example, [12]) and for vector supercomputers (see [1, 3, 5]). This paper instead focuses on the performance of sparse Cholesky factorization on machines with hierarchical-memory systems. Memory hierarchies are crucial for building high-performance machines at low cost. They bridge the speed gap between extremely fast processors, whose speed is almost doubling every year, and the relatively slow main memories that must be used to keep costs down. If a computation can make good use of the hierarchy, then it can avoid the large latencies associated with fetching data from the slower levels of the hierarchy, yielding much higher performance. We look at several different sparse Cholesky factorization methods, and consider how well each one uses the memory hierarchy.

The implementor of a sparse Cholesky factorization code faces a number of implementation decisions. Our goal in this paper is to study the impact of these decisions on the performance of the overall code on hierarchical-memory machines. The first and probably most visible implementation decision we consider is the structure of the overall computation. We consider three approaches: left-looking, right-looking, and multifrontal. These approaches will be summarized in the next section.

Another important implementation decision that we consider is the choice of primitives on which the computation is based The most commonly used primitives *are column-column* primitives, where columns of the matrix are used to modify other columns. We demonstrate that column-column 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 supemode-column primitives. Such primitives take advantage of the existence of sets of columns with identical non-zero structures, called *supernodes,* to increase data reuse. These primitives modify columns of the matrix by all columns in an entire supemode at once. The increased reuse comes from the fact that the destination column is modified a number of times, allowing the supemode-column modification to *be unrolled* [7]. The unrolling allows data items from the destination to be kept in processor registers across multiple modifications. As a result, for moderately large sparse problems, memory reference are reduced by more than 50% and performance is improved by between 50% and 100%. We also consider *column-super-node* primitives, where a single column is used to modify a number of columns in a supemode. 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 a number of destination columns by a number of 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 supemode-column methods, and performance is improved by between 30% and 45%. We then consider the *use* of *supernode-supernode* primitives, where supemodes modify entire supemodes. 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 supemode-pair codes.

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

This paper 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 paper 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 paper analyzes supernode-supemode methods, a class of methods that have so far received little attention. The use of supemode-supemode primitives has been proposed before [5]. However, we believe that we are the first to publish detailed performance evaluations of practical implementations.

The paper is organized as follows. We discuss three different high-level approaches to the sparse factorization problem in section *2.* Section *3* then briefly discusses the primitives used to implement these three approaches. In section 4, we describe our experimental environment. We then consider a number of implementations of the high-level approaches based on different primitives in section 5. We look at the cache performance of each of these variations, as well as the achieved performance on two hierarchical-memory machines, In section 6, 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 7 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 8 and present conclusions and section 9.

2 Sparse Cholesky Factorization

2.1 Left-looking, Right-looking, and Multifrontal Approaches

This section describes three high-level approaches to sparse cholesky factorization: the left-looking, right-looking, and multifrontal approaches. The goal of any approach to the sparse cholesky computation is to factor a sparse positive definite matrix A into the form LL^T, where L is lower-triangular. The column-oriented computation is typically expressed in terms of two primitives:

- $cmod(j,k)$: add into column j a multiple of column k
- $cdiv(j)$: divide column j by the square root of its diagonal

Using these two primitives, the column-oriented sparse cholesky can be phrased in two different manners, known as *left-looking* and *right-looking* (or *column-cholesky* and *submatrix-cholesky*). The general structure of the left-looking approach is as follows, with the k term iterating over columns to the left of column j in the matrix.

```
for j= 1 to n do
    cdiv(j)
    for each $k$ that modifies j do
        cmod(j, $k$)
```

The general structure of the right-looking approach is as follows, with the j term iterating over column to *the* right of column k.

```
for $k = 1$ to $n$ do
    cdiv (k)
    for each $j$ modified by $k$ do
        cmod(j, $k$)
```

Note that in both cases, j iterates over destination columns and k iterates over source columns. Note also that the $cmod(\,)$ operation is performed a number of times per column while the $cdiv(\,)$ operation is performed only once. The $cmod(\,)$ operation therefore dominates the runtime.

In a sparse problem, the columns j and k have different non-zero structures; the structure of the destination j is a superset of the structure of source k. In order 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 problem.

In the left-looking approach, the same destination column 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. Columns are added into the full destination vector using an indirection, where the destinations are determined by the non-zero structure of the 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 [13]. Further details are provided in a later section.

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 [11].

Another approach to right-looking factorization, called the *multifrontal method* [9], 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.

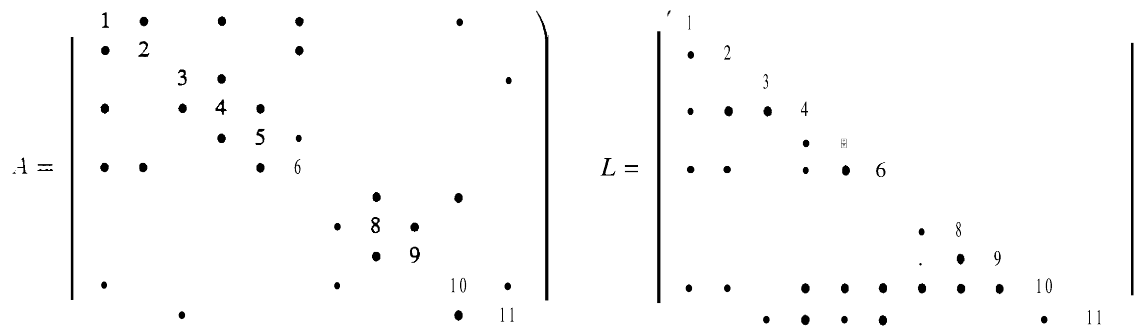

Figure 1: Non-zero structure of a matrix A and its factor L.

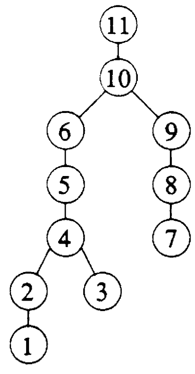

Figure 2: Elimination tree of A.

The multifrontal *method* is more easily explained *in* terms of *the elimination tree* [21] of the factor, a structure that we now describe. The elimination tree of a factor L is 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. As an example, in Figure 1 we have a matrix A, and its factor L. In Figure 2 we show the elimination tree of this matrix. The elimination tree provides a great deal of information about the structure of the sparse cholesky computation. One important piece of information that can be obtained from the elimination tree is the set of columns that can possibly be modified by a column. A column can only modify its ancestors in the elimination tree. Equivalently, a column can only be modified by its descendents.

Returning to our description of the multifrontal method, we note that the most important data structure in the multifrontal method is the set of updates from a subtree in the elimination tree to the rest of the matrix. *These* updates *are* kept *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 1 would produce an update matrix that looks like the matrix in Figure 3. Note 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. The updates from column 2 are 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 4. The update matrix from column 3 would be handled in a similar manner. The actual assembly operation typically makes use of *relative*

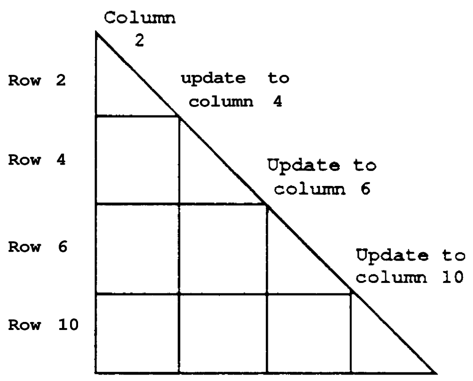

Figure 3: Update matrix for column 2.

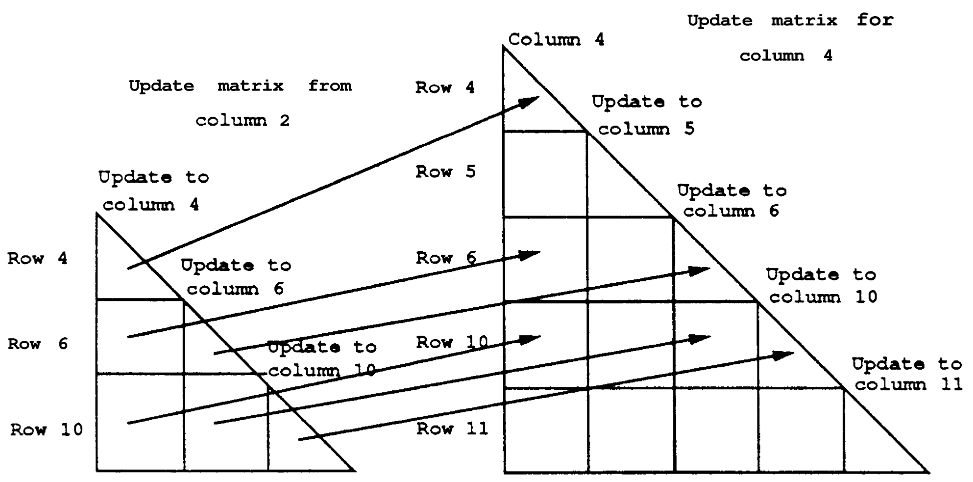

Figure 4: Assembly of update matrix from column 2 into update matrix of column 4.

indices [3, 21] 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**}, 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 use these indices 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 *cmod*( ) 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** stuck*. 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 [9].

2.2 Supernodes

An important concept in sparse cholesky factorization is that of a *supernude.* A supemode 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 supemode must also form a simple path in the elimination tree, meaning that each column in the supemode must have only one child in the elimination tree. As an example, consider the matrix of Figure 1. Columns 1 through 2 form a supemode in the factor, as do columns 4 through 6, columns 7 through 9, and columns 10 through 11. Supemodes arise in any sparse factor, and they are typically quite large.

Probably the most important property of a **supemode** is that each member column modifies the same set of destination columns. Thus, the Cholesky factorization computation can be expressed in terms of supemodes modifying columns, rather than columns modifying columns. For example, a left-looking **supemodal** 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 **supemode** s. The modification of a column by a supemode can be thought of as a two-step process. In the **first** step, the modification, or update, from the supemode is computed. This update is the sum of multiples of each column in the supemode. Since all columns in the supemode have the same structure, this computation can be performed without regard for the actual non-zero structure of the supemode. The update can be computed by adding the multiples of the supemode columns together as dense vectors. The result can be considered to have the same structure as each column of the supemode. In the second step, the update vector is added into the destination, taking the non-zero structure into account. Supemodes have been exploited in a variety of contexts [5, 9, 19].

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

Supemodes will be exploited for a variety of purposes in this paper.

2.3 Assorted Details

This paper considers the performance of a number of implementations of each of the three described high-level approaches. In order to make these performance numbers 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 [16]. We do not include the time spent determining this order in the computation times presented in the paper.

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 updates 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 supemode in the elimination tree. The update matrices from supemodes of odd depth are kept on one stack, with the update matrices from supemodes 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.

3 Sparse Cholesky Primitives

The three high-level approaches to sparse Cholesky factorization that have been described, the left-looking, right-looking, and multifrontal methods, have so far been expressed in terms of column-column or supemode-column modifications. In this paper, we consider a range of possible primitives for expressing the sparse Cholesky computation. In terms of more general primitives, a 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)
```

The $Complete()$ primitive computes the final values of the elements within a structure, once all modifications from other *structures* 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.

A generalized right-looking approach would look like:

```
1.  for $k = 1$ to $NS$ do
2.      Complete(k)
```

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

```
3.      for each j modified by $k$ do
4.          ComputeUpdateToJFromK(j, $k$)
5.          PropagateUpdateToJFromK(j, k)
```

A generalized multifrontal approach would be similar to the right-looking approach, except that the *PropagateUpdate()* step would be handled at a different tune. The update is computed directly into the update matrix, and propagation is performed during the *assembly* step.

This paper considers a range of possible choices for the structures j and k, and considers the efficiencies of the resulting computations. Clearly, to be interesting choices, the chosen structures must lead to efficiently implementable primitives. We limit ourselves to three structure choices: columns, supemodes, and entire matrices. The actual primitives will be described in more detail in later sections.

4 Experimental Environment

This paper will consider the relative performance of various approaches to sparse Cholesky factorization. To that end, we 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 on which we perform the factorizations.

We have chosen a set of eight sparse matrices as benchmarks for evaluating the performance of sparse factorization approaches (see Tables 1 and 2). With the exception of matrices DENSE750 and GRID100, all of these matrices come from the Hanvell-Boeing Sparse Matrix Collection [8]. 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 sparse LSHP3466, all the way to the completely dense DENSE750. In order to reduce fill in the factor, all benchmark matrices are ordered using the multiple minimum degree heuristic [15] before the factorization.

We will be presenting performance numbers for the factorization of these matrices throughout this paper. We will typically present numbers for each matrix, as well as summary numbers. The summary numbers will take three forms. One number will be the 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 axe the DECstation 3100 and the IBM RS/6000 Model 320. Both are high-performance 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.

The data cache on the DECstation 3100 is direct-mapped, meaning that each location in memory maps to a specific line in the cache. If a location is fetched into the cache, then the fetched location displaces the data that previously resided in that 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 3 100 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 in 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. In particular, a floating-point instruction can be initiated every cycle. Furthermore, the machine contains a multiply-add instruction that performs both instructions in the same time it would take to perform either individual operation. 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. 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 [6]. 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

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

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.

5 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.

5.1 Column-column Methods

We first consider factorization approaches based on *cdiv*( ) and *cmod*( ) primitives. Recall that these primitives work with individual columns of the matrix. We refer to these 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 paper. Note that in this and all other multifrontal implementations, one frontal update matrix is computed per supemode.

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, 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

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	0.35	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

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 the computational kernels of the three methods. Recall that the dominant operation in each of these methods is the *cmod()* operation, in which a multiple of one column is added into another, **y** — ax + **y**. In the left-looking method, this conceptual operation is accomplished by scattering a multiple of the vector **x** into a full destination vector, using the indices of entries of 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.      $y[index[i]] = y[index[i]] + a * x[i]$
```

We refer to *this* kernel *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.** We assume that the values are 2-word double-precision floating-point numbers and the indices are single-word integers. We therefore load 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. In the right-looking method, we have two vectors, x and **y,** each with different non-zero structures, **xindex** and **yindex.** We must search in **y** for the appropriate locations into which elements of **x** should be added. The loop looks like the following:

```
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] = $y[yi] + a * x[xi]$
```

We refer to this *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.** We therefore expect 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. In the left-looking method, the destination column is modified by a number of columns to its left, while in the right-looking and multifrontal methods, the source column modifies a number of columns to its right. Thus, in each of the three **y** — ax + **y** kernels from above, one of the two vectors **x** or **y** does not change from one invocation to the next. With a reasonably large cache, we would expect this vector to remain in the cache, thus we would only expect one vector to miss per column modification. In other words, for every multiply/add, we would expect 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.

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.

5.2 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 supemodal modifications into the computational kernel, where the update from an entire supemode is formed using dense matrix operations, and then the aggregate update is added into its destination. Supemodal elimination can be easily integrated into each of the approaches of the previous section [5].

We now consider the implementation of supemode-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 supemode to a column has been computed. The update has the same structure as the source supemode. In the supemode-column left-looking method, the update is scattered into the full destination vector, using the structure of the source supernode to determine the appropriate positions. Notice that this operation is nearly identical to the scatter kernel of the column-column method. In the right-looking supemode-column method, the update is added into the appropriate positions in the destination column using a search. Again, this operations is nearly identical to the corresponding column-column kernel. In the multifrontal supemode-column method., the update is computed directly into the frontal update matrix, so no propagation is immediately necessary. Although the supemode-column propagation primitives are quite similar to the corresponding modification kernels of the column-column methods, this do not imply that the overall performance of the two approaches will be similar. The propagation primitives in the supemode-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 supemode 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 supemode, 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.            $y[i] = y[i] + a_k * x_k[i]$
```

where each column x_k is successively added into the destination **y.** If we consider the memory operations necessary to implement this kernel, we expect 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, since a number of columns are added into the same destination at once, we can take advantage of this reuse of data to reduce memory operations. The loop can be unrolled [7] over the modifying columns, as follows:

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

In the above, we have performed 2-way unrolling. We would expect each iteration of the inner loop to load two elements of x, load one element of **y,** and store one element of **y.** The code would also perform 4 floating-point operations on this data We therefore perform 8 memory operations to do 4 floating-point operations, for a ratio of 2 memory operations per floating-point operation. In general, with u-way unrolling, we 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 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, we would obtain a ratio of 1.25 references per operation.

In Table 5 we present performance numbers for the three supemode-column methods. We also present

Table 5: Performance of supemode-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	4.65	1.27	2.68	1.48	6.30
Supemode-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
Supemode-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

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 supemode-column methods are significantly faster than the column-column methods, ranging from 30% faster for the multifrontal method on the DECstation 3100, 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 supemode-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 information in Table 8.

Before we analyze the behavior of these methods, we make a brief observation about the multifrontal and left-looking supemode-column methods. When the multifrontal method is compared with the left-looking method, one of the most frequently stated performance advantages [9] of the multifrontal method 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 supemodal versus nodal elimination, rather than multifrontal versus left-looking methods. Recall that in both the left-looking and multifrontal methods, an update is computed from an entire supemode 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 supemode in the assembly step. In the left-looking supemode-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

Table 7: References and cache misses for supemode-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

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
Supemodecolumn	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
Supemode-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
Supemode-column	1.76	0.55	1.71	0.20	1.81	0.33

are almost entirely equivalent in terms of indirect operations and extra floating-point operations.

Returning to the memory reference numbers, it is somewhat surprising to note that even though the multi-frontal and left-looking methods exhibit roughly equivalent behavior, the two methods do not produce the same number of memory references. The differences are much smaller than they were in the column-column methods, because the methods share the same kernel, but differences still exist. The difference is caused by additional data movement in the multifrontal method, due to two subtle differences between the methods. The first difference is related to the destinations into which updates are added. In the left-looking method (and the right-looking method as well), updates are always added directly into the destination column. In the m&frontal method, updates are added into update matrices. When all updates to a column have been performed the left-looking and right-looking methods complete that column in-place. The multifrontal method, on the other hand, adds the original column entries into the update matrix, computes the final values of that column, and then copies the final values back to the column storage.

The other difference relates to the manner in which supemodes 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 supemodes containing only a single column. This option does not exist for the multifrontal method. We find through simulation that the lack of a single-column special case accounts for slightly more than half of the increase in memory references, with the movement of completed columns accounting for the rest. We also find that the relative cost of this additional data movement is larger for smaller problems. This is to be expected, for two reasons. First, the number of operations done in a sparse factorization grows much more quickly than the number of non-zeroes in the factor. Thus, the relative cost of moving each factor non-zero to and from an update matrix decreases as the problem size increases. The other reason is that single-column supemodes generally account for a much smaller portion of the total operations in larger problems. In summary, the multifrontal method has a performance disadvantage when compared to the left-looking and right-looking methods because it performs additional data movement.

Returning to the memory reference numbers, another thing to note is that the numbers for the supemode-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 supemode-column methods access index vectors much less frequently. Second, the supemodal 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.

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 supemodes are large enough to take full advantage of the reuse benefits of supemodal 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 supemode-column variants. The numbers for the right-looking and multifrontal methods, on the other hand, have decreased significantly. This fact can 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 supemode-column left-looking method, this reuse has not changed. We expect that the destination column to remain in the cache, and the supemodes that update it 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 supemode to a number of columns. Thus, the item that is reused is a supemode. We see three possibilities for the behavior of the cache, depending on the size of the supemode. If the supemode contains a single column, then we would expect the supemode to remain in the cache, and the destination columns to cache miss, resulting in one miss per floating-point operation. If the supemode contains more than one column but is smaller than the cache, then we would again expect the supemode to remain in the cache, and the destination to miss. However, we are now performing many more floating-point operations 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. If the supemode is much larger than the processor cache, then we expect the destination to remain in the processor cache while the supemode update is being computed, as would happen in the left-looking method, resulting in one miss per floating-point operation. The cache miss numbers in Table 7 indicate that the case where a supemode fits in the cache occurs quite frequently, resulting in significantly fewer misses than one miss per floating-point operation overall.

Another interesting item to note in the cache miss numbers is the difference between the miss rates of the multifrontal and right-looking methods. It would appear that since both methods take a right-looking approach to the factorization, they should have identical cache miss behaviors. The reason that they do not is that the multifrontal method performs more data movement. We discussed earlier the reasons why the multifrontal method performed more data movement than the left-looking method. The same reasons hold true when the multifrontal method is compared to the right-looking method. Simulation has shown that slightly more than half of the extra memory references are due to the methods used to handle single-column supemodes. In the case of cache misses, simulation has shown that most of the extra misses are due to the copying of supemode data to and from update matrices.

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.

5.3 Column-supernode Methods

The supemode-column primitives of the previous section took advantage of the fact that a single destination is reused a number of times in a supemode-column update operation to increase reuse in the processor registers. They also took advantage of the fact that every column in the source supemode 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 supemode, 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-supemode *ComputeUpdate( )* primitive. A column would be used to modify a set of destinations, appearing something like:

```
for j= 1 to $J$ do
    for i= 1 to $n$ do
        $y_j[i] = y_j[i] + a_j * x[i]$
```

Unrolling the j loop by a factor of two yields:

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

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, if we unroll u ways we load u entries of **y**, one entry of x, and write u entries of **y** to perform 2 u 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 supemode-column primitive. Thus, while column-supemode primitives realize some advantages due to reuse of data, they are not nearly as effective as supemode-column primitives. We therefore do not further study such methods.

5.4 Supernode-pair Methods

In this section, we consider a simple modification of the three supemode-column factorization methods that further improves the efficiency of the computational kernels and also reduces the cache miss rates. The modification involves a change in the number of destination columns for the supemode modification primitives. Rather than modifying one column by a supemode, we now modify two. We call the resulting methods ***supernode-pair methods***. We will study a more general form of this modification, where a supemode modifies an entire supemode, in the next subsection.

Devising factorization methods that make use of supemode-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 supemode-column left-looking propagation primitive to update each. The bookkeeping necessary to determine which supemodes 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 the 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 supemode-column right-looking propagation primitive.

The *ComputeUpdate* ( ) step in a supemode-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 Ii 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.            $y_2[i] = y_2[i] + a_{2k} * x_k[i] + a_{2k+1} * x_{k+1}[i]$
```

If we count memory references, we find 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 supemode-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 supemode-pair kernel uses roughly twice as many registers as the supemode-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 .O. 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 increasing the amount of instruction parallelism.

One thing to note about the left-looking supemode-pair method is that there is no guarantee that adjacent columns will be modified by the same supernodes. In fact, it is possible to order the columns so that an equivalent computation is performed, but adjacent columns are rarely modified by the same supemode. Depending on the fill-reducing heuristic used, it may be necessary to heuristically reorder the matrix in order to achieve significant benefits from a supemode-pair left-looking method. For more information on this topic, we refer the reader to [2]. We note here that the SPARSPAK implementation of the multiple minimum degree ordering heuristic is

Table 9: Performance of supemode-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 supemode-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

quite effective at producing orderings that place columns that are frequently modified by the same supemodes adjacently.

We now present performance figures for the three supemode-pair methods (Table 9), using the identical supemode-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. For a machine with 16 registers, the memory reference numbers would be quite similar to those of the supemode-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 supemode-pair methods with the methods of previous sections. The performance data shows that supemode-pair methods give significantly higher performance than the supemode-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 supemode-column methods. The supemode-pair numbers are

Table 11: Mean performance numbers on DECstation 3 100 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

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
Supemode-column	2.30	0.32	2.13	0.09	2.45	0.17
Supemode-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
Supemode-column	1.63	0.92	1.62	0.63	1.67	0.69
Supemode-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
Supemode-column	1.76	0.55	1.71	0.20	1.81	0.33
Supemode-pair	1.22	0.39	1.15	0.16	1.24	0.28

above the ideal of 0.75, but they are still quite low.

The cache miss numbers for the supemode-pair methods are substantially lower as well. For example, the cache miss numbers are 30% lower for the left-looking method. This difference can be understood as follows. In the left-looking supemode-pair method, a pair of columns is now reused between supemode updates. When a supemode is accessed, it frequently updates both columns, thus performing twice as many floating-point operations as would be done in the supemode-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 supemodes relative to the size of the cache. Of the three cases we outlined, only the case where the supemode 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.

The performance gains from the supemode-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 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, we have a significant decrease in the number of memory references. The second factor has to do with the fact that the supemode-pair kernel updates two destinations at once in the inner loop, allowing for a greater degree of instruction parallelism. The third factor has to do with the decrease in the number of cache misses. The 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.

5.5 Supernode-supernode Methods

An obvious extension of the supemode-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 supemode by another supemode. Such primitives were originally proposed in [5]. By expressing the computation in terms of supemode-supemode 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 supemode-supemode primitives to reduce memory system traffic in a left-looking method has been proposed independently in [18]. We use a simple form of blocking in this section. We discuss alternative blocking strategies in a later section.

5.5.1 Implementation of Supemode-supemode Primitives

We begin our discussion of supernode-supemode methods by describing the implementation of the appropriate primitives. To better motivate the blocking that will be done in a subsequent section, we describe the implementation of supernode-supemode primitives in terms of dense matrix computations. We note that these primitives will all be implemented in terms of columns of the matrix in this section.

We begin with the *Complete*( ) primitive. Expressed in terms of the columns of the supemode, the *Complete*( ) 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($j$)
```

An equivalent description of this computation, in terms of dense matrices, would be:

```
1.  $A \leftarrow Factor(A)$
```

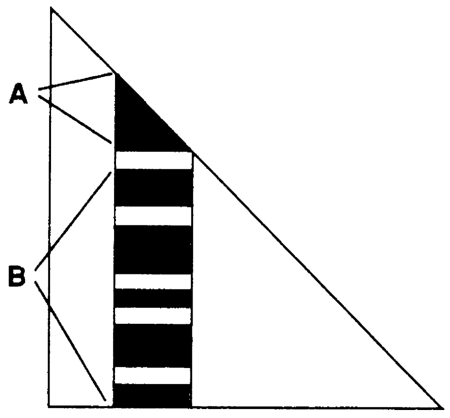

Figure 5: The *CompleteSuper*( ) primitive.

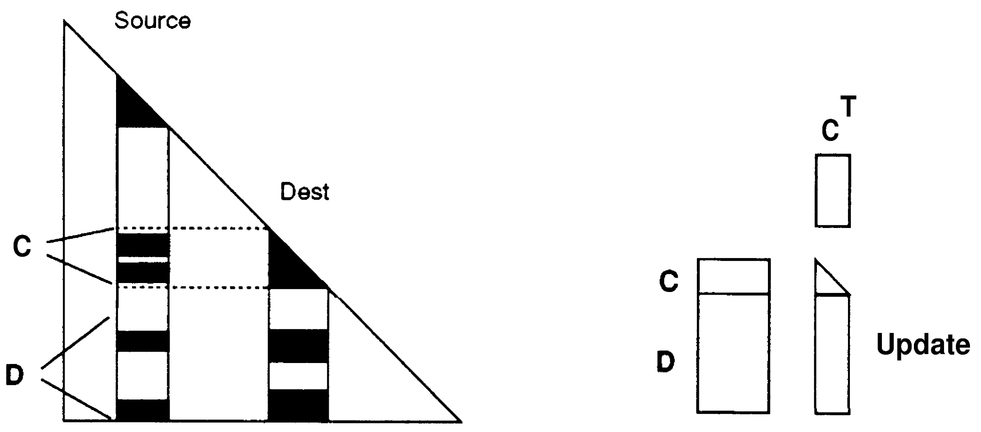

Figure 6: The *ModifySuperBySuper*( ) **primitive.**

2. $B \leftarrow BA^{-1}$

where A is the dense diagonal block of the supemode, and B is the matrix formed by condensing the sub-diagonal non-zeroes of the supemode into a dense matrix (see Figure 5). 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 large set of triangular systems. This step can be done in-place. One thing to note about the *Complete*( ) primitive is that the entire operation can be performed without consulting the indices for the sparse columns that comprise the supemode. 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 supemode. The *PropagateUpdate*( ) primitive must then add the update matrix into the destination supemode.

The *ComputeUpdate*( ) step involves the addition of a multiple of a portion of each column in the source supemode into the update matrix. The operation can be thought of in terms of dense matrices as follows. Assume the destination supemode is comprised of columns d_f through d_l. The only non-zeroes in the source supemode that are involved in the computation are those at or below row d_f. These non-zeroes can be divided into two sets. We have a matrix C, corresponding to the non-zeroes in the source supemode in rows d_f through d_l (see Figure 6). We also have a matrix D, corresponding to the non-zeroes in the source supemode 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. We partition each supernode of the matrix 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. We then proceed to 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 $ComputeUpdate(\,)$ primitive. We will consider different blocking strategies in a subsequent section.

Once the update matrix is computed, the next step is propagation, in which 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 indexing technique [3, 21]. 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 allows us 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, In the case of the left-looking method, the destination supernode is modified by a number of other supernodes. We scatter the structure of the destination into a full structure once. In other words, if a non-zero in row i appears at position j in the destination, then we store j in entry i of the full structure. Each supernode-supernode modification to this destination can use the full structure information to compute its own relative indices. For the right-looking method, relative indices are computed using a search, as is done in the column-column search kernel. Once the relative indices have been computed, it is a simple matter to add the entries of the update matrix into the appropriate locations in the destination supernode. Note that both 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 without storing them in an intermediate update matrix. 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 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.

Table 13: Performance of supemode-supemode 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

Table 14: References and cache misses for supemode-supemode 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

5.5.2 Performance of Supemode-supernode Methods

We now present performance numbers for the supemode-supemode 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 supemode-supemode methods is again higher than that of the previous methods. At one extreme, the left-looking supemode-supemode method is more than 40% faster that the left-looking supemode-pair method on the DECstation 3 100. At the other extreme, the multifrontal supemode-supemode method achieves roughly the same performance as the multifrontal supemode-pair method on the RS/6000. In this case, the performance on the smaller matrices has decreased, while the performance on the larger matrices has increased. Overall, the supemode-supemode methods are 10% to 40% faster on the DECstation 3100 over the whole set of benchmark matrices, and 0% to 10% faster on the IBM RS/6000. For the larger matrices, these methods are 20% to 50% faster on the DECstation, and 5% to 20% faster on the IBM.

Moving to the memory reference information, we note that the number of references has decreased by a few percent for the left-looking method. This is primarily because the increased flexibility in performing column modifications allows the use of supemode-pair updates more often. In the left-looking supemode-pair method, a pair-update was possible only if a supemode modified both of the current destination columns. If the supemode modified only one, then the less efficient supemode-column *ComputeUpdate( )* primitive was used. In the

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

supemode-supemode method, pair-updates are used whenever a supemode modifies two or more columns in the destination supemode. Memory references have increased slightly for the right-looking and multifrontal methods. The increased efficiency of the *PropagateUpdate( )* primitive in the right-looking method is balanced by small inefficiencies introduced by the blocking. By splitting the supemodes into a number of smaller panels, we frequently reduce the effectiveness of the loop unrolling. We also frequently waste effort, particularly in the smaller problems, looking for increased reuse that is not present. The effect can be seen more clearly in the memory reference numbers for the multifrontal method, which does not gain any offsetting memory reference advantage in moving to supemode-supemode primitives.

We note that the cache miss rates for the three methods are similar, and in all cases they are substantially lower than those achieved by the supemode-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 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-supemode methods significantly increase performance over supemode-pair methods.

5.6 Supernode-matrix Methods

We now consider methods based on primitives that produce updates from a single supemode to the entire rest of the matrix. The multifrontal method is most frequently expressed in terms of such primitives (see, for example, [1]). One thing to note about supemode-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 supemode-matrix primitives, we note that the *Complete( )* and *ComputeUpdate( )* primitives are typically merged into a single operation. The final values of the supemode are determined at the same time that the updates from the supemode 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 supemode-matrix primitives. The implementation of *ComputeUpdate(* is relatively straightforward. The trapezoidal update matrix from the supemode-supemode methods becomes a lower triangular matrix in supemode-matrix methods. This update matrix is computed by performing a symmetric matrix-matrix multiplication, $C = BB^T$, where B is the portion of the source supemode 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 supemode-supemode 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-supemode method. In fact, the supemode-supemode and supemode-matrix right-looking methods are nearly identical. The difference is in the order in which primitives are invoked. In the supemode-supemode method, the update to a single supemode is immediately added into the destination. In the right-looking supernode-matrix method, the updates from a single supemode to all destination supemodes are computed, and then these updates are propagated one at a time to the appropriate destination supemodes.

In Table 17 we present performance numbers for the supemode-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 supemode-matrix methods is quite similar to the performance of the corresponding supemode-supemode methods.

One would expect that in moving from an approach that produces updates from a supernode to a single destination supemode to an approach that produces updates from a supemode 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 supemode-supemode updates. Specifically, a single supemode-matrix update typically corresponds to a small number of supemode-supemode updates. Therefore, little reuse is lost in splitting a supemode-matrix update into a set of supemode-supemode updates.

Another important reason for the lack of improvement is the existence of significant fractions of the compu-

Table 17: Performance of supemode-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

Table 18: References and cache misses for supemode-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

tation that are not affected by the change from supemode-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 supemode-supemode 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 supemode-matrix primitives. Another example of an operation that is little affected by the change is the computation of updates from a supemode containing few columns.

The supemode-supemode methods also have a small offsetting advantage over the supemode-matrix methods concerning the propagation of updates once they have been computed. If a large update is generated, as is the case in the supemode-matrix methods, then it is likely that the update will not remain in the cache between the time when it is computed and the time when it is propagated. If a small update is generated and then immediately propagated, it is more likely to be available in cache for the propagation operation.

The small performance differences between the supemode-supemode and supemode-matrix methods are therefore 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 supemode-matrix methods, especially for the the small problems. The main reason for this is simply that the increased task size of the supemode-matrix methods leads to slightly fewer conflicts between the numbers of columns in the task and the degree of unrolling. The supemode-matrix methods achieve slightly higher performance overall on both machines.

5.7 Summary

This section has studied the performance of a wide variety of methods for performing sparse Cholesky factorization. We found that the performance of the various methods depended 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.

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.

6 Cache Parameters

This paper 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.

6.1 Line Size

A common technique for decreasing the aggregate amount of latency a processor incurs in waiting for cache misses to be serviced 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. The practice of loading multiple words of data into the cache in response to a miss on one of them is beneficial only if the extra data that is brought in is requested by the processor shortly after being loaded. This property, called *spatial locality,* is present in many programs. We now evaluate to what extent it is present in sparse Cholesky factorization.

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	Increase in traffic	Traffic: Words/op	Increase in traffic	Traffic: Words/op	Increase 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%

In Figure 7 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 this results in a factor of between 14 and 32 decrease in cache misses, it is not so 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 memoy 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. In Table 21, we show 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 for this line size.

6.2 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 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. In a set-associative cache, 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

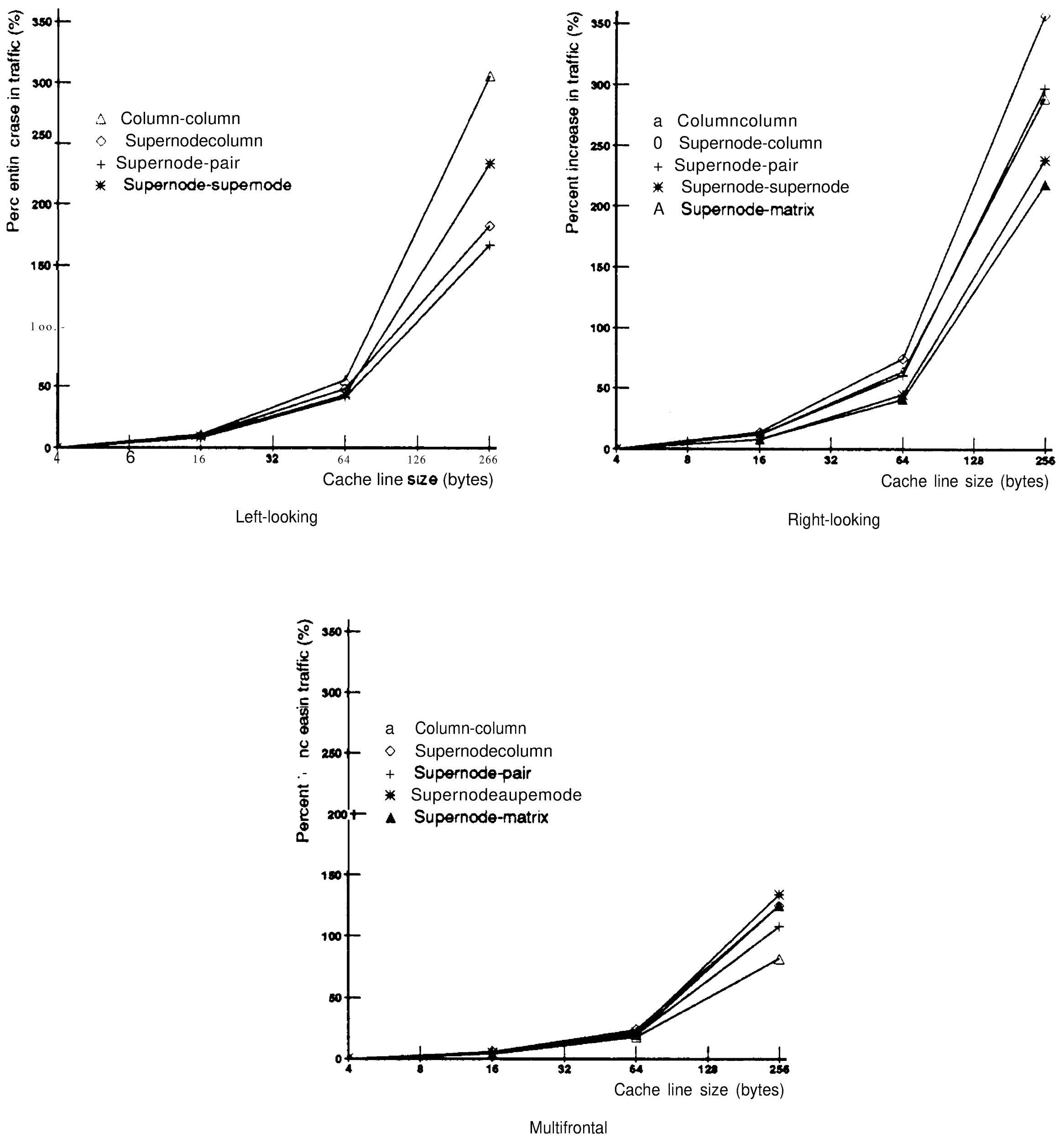

Figure 7: Increase in data traffic due to longer cache lines. Cache size is 64 KBytes in all cases.

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	Decrease in traffic	Traffic: Words/op	Decrease in traffic	Traffic: Words/op	Decrease 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%
Supemode-supemode	0.15	26%	0.14	31%	0.18	19%
Supemode-matrix	·	-	0.14	29%	0.18	19%

and more complicated to build than direct-mapped caches, they often result in a substantial decrease in the number of cache misses incurred.

In Table 22 we present the 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 each of the factorization methods.

6.3 Cache Size

Up to this point in this paper, 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 8 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 [19]

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 supemode-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 block the computation 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 supemode-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 neariy twice as many misses as the right-looking method. From a previous discussion, we know that right-looking methods achieve enhanced reuse when supemodes fit within the cache. A larger cache makes it more likely that supemodes will fit. The left-looking methods do not share such benefits.

The methods that block the computation 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 amount of benefit available from blocking the computation begins to grow. We observe that the miss rates fail off quickly for the blocked approaches.

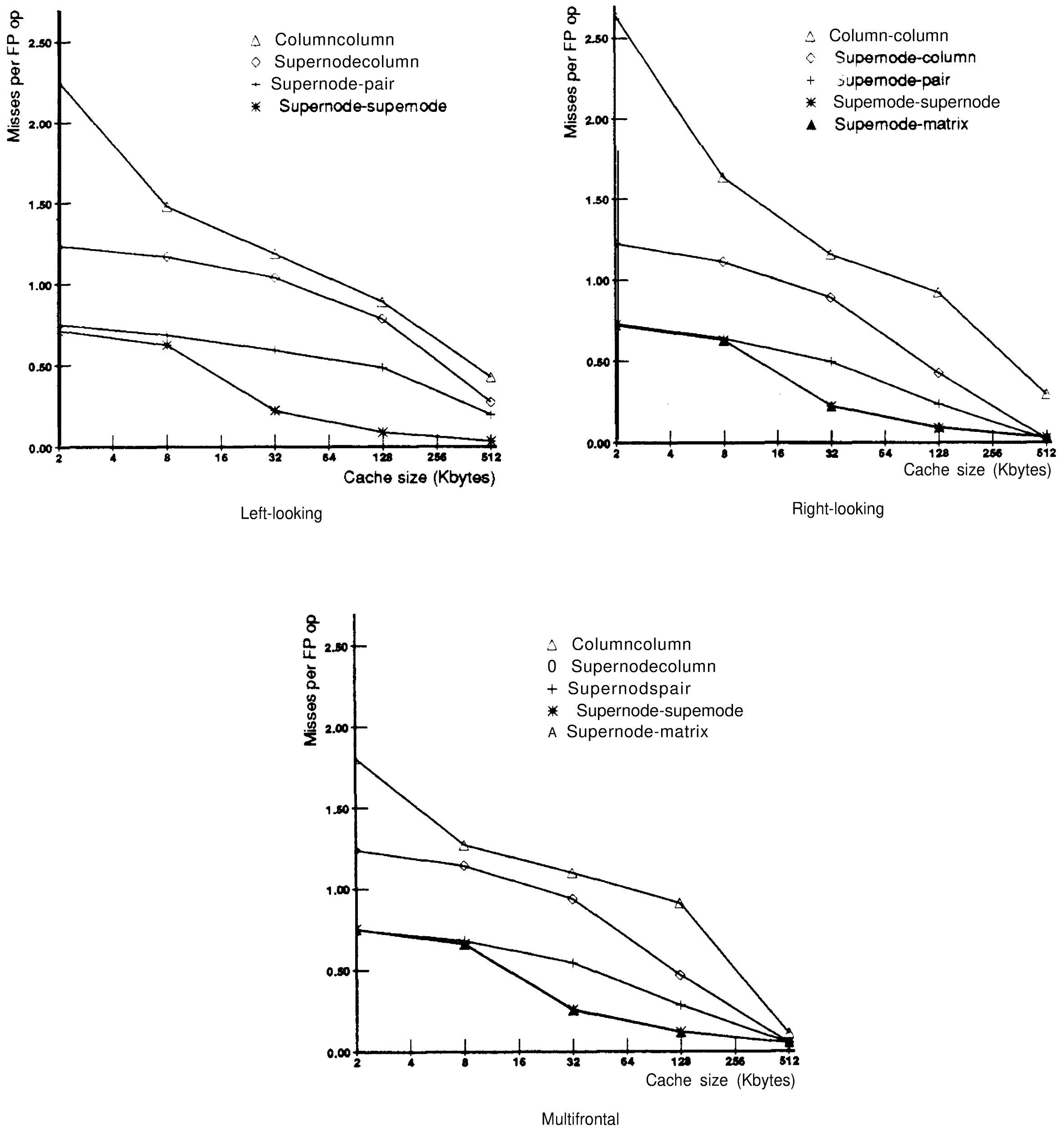

Figure 8: Cache miss behavior for various methods, matrix BCSSTK15.

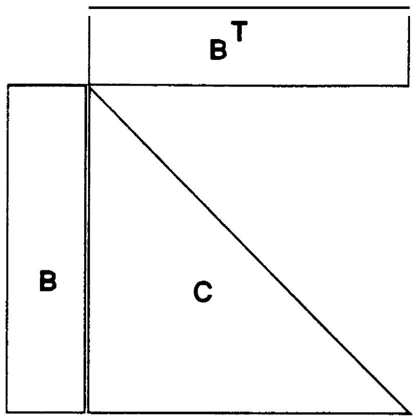

Figure 9: Update creation.

7 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.

7.1 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. We will describe this blocking strategy in terms of the multifrontal supemode-matrix method, although the discussion applies to the other supemode-supemode and supemode-matrix methods as well. Recall that in the **multifrontal** method, a supemode is used to create an update matrix. Consider the matrices of Figure 9. The B matrix represents the non-zeroes within the supemode. 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 10 shows the case where the supemode 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 transpose of the panel; the destination vector is a column **from** the destination matrix (see Figure 11). Thus, to produce one column of the destination, we touch the entries in the block, one column from the transpose, and of course the destination 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. Recall that this important parameter c represents the number of supemode columns that fit in the cache, which is determined by the size of the cache and the length of these columns.

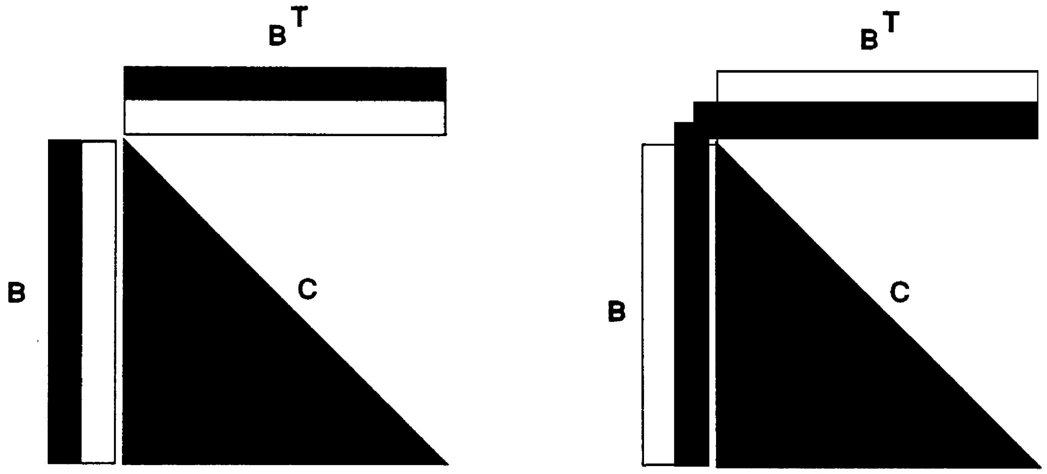

Figure 10: Panel blocking for update creation.

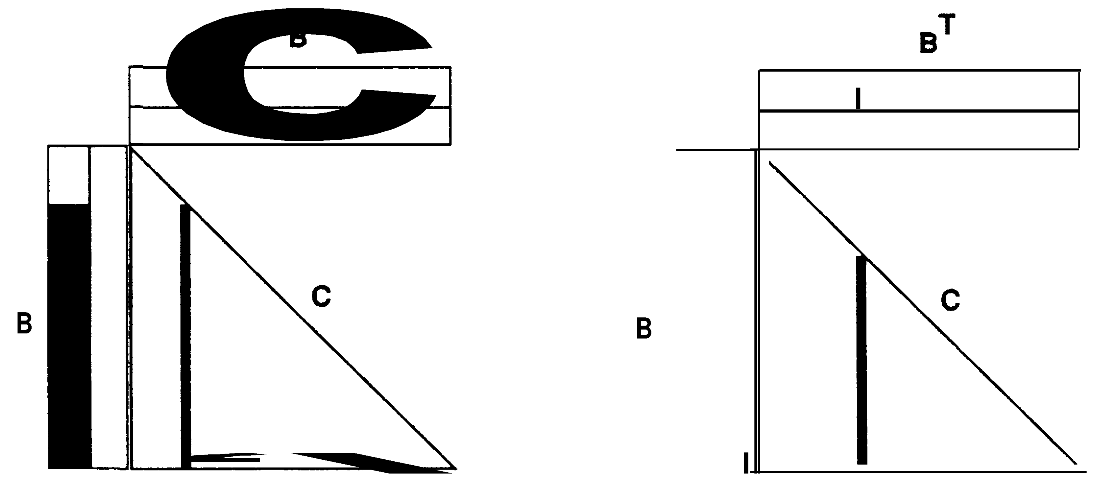

Figure 11: Matrix-matrix multiply as a series of matrix-vector multiplies.

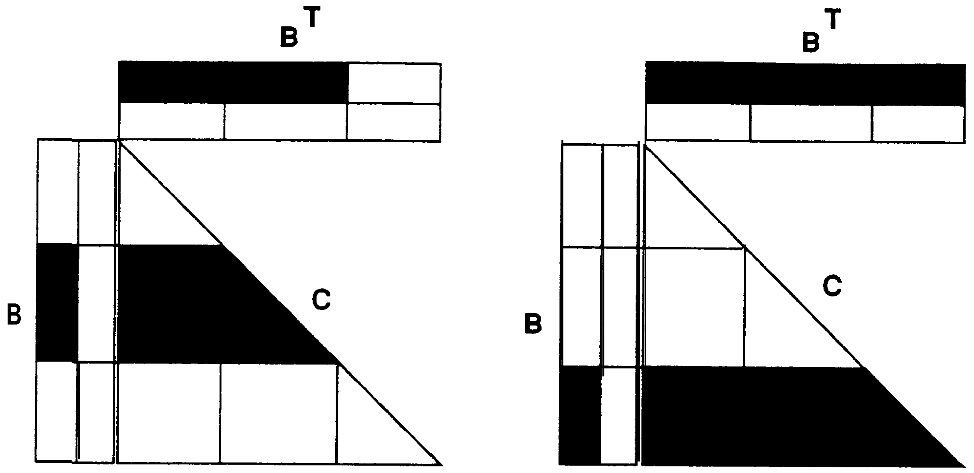

Figure 12: Submatrix blocking for update creation.

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, [lo]). 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 12). 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 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 x 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.

7.2 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.

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. If we consider how these non-zeroes would map into a direct-mapped cache, we find that they cannot possibly interfere with each other. 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 [14]). 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 reused block 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 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 13. 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

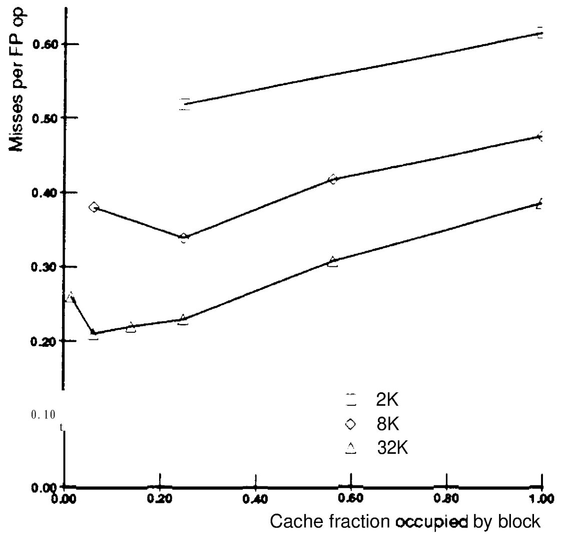

Figure 13: Cache miss behavior for multifrontal supemode-matrix method, using square blocks, for matrix BCSSTK15.

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 in the cache [10, 14]. In effect, the cache is treated as a fast local memory, and the data to be reused am explicitly copied into this local memory before being used The result of employing the copying optimization to the sparse problem (BCSSTK15) is shown in Figure 14. 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 the block 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 supemode-matrix approaches on our benchmark machines, we briefly consider the use of square blocks in supemode-supemode 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 supemode-matrix and supemode-supemode methods is that in the former, a single supemode is used once to modify the entire matrix. Since each supemode is used only once, a code that copies blocks will copy each non-zero in the matrix at most once. In supemode-supemode methods, on the other hand, a single supemode is used to modify a number of other supemodes, and each of these operations is performed as a separate step. Consequently, if non-zeroes are copied, then the entries of a single supemode must be copied multiple times, once for each supemode-supemode 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 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

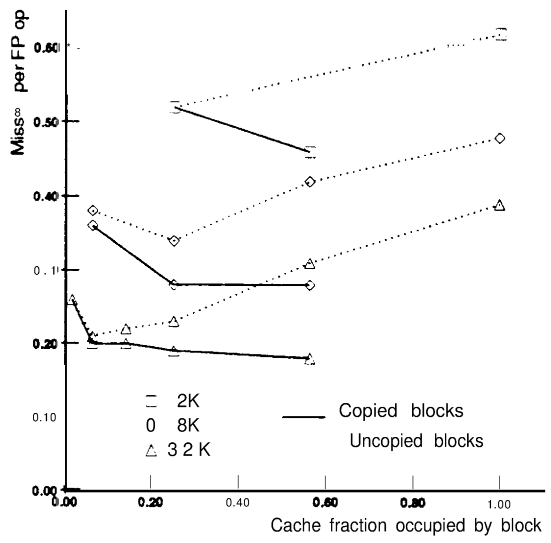

Figure 14: Cache miss behavior for multifrontal supemode-matrix method, using square blocks and copying, for matrix BCSSTK15.

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

	Left-looking supemode-supemode		Right-looking supemode-matrix		Multifrontal supemode-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

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 3 100 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%

the panel-blocked methods on both machines. On the DECStation 3 100, the square-block schemes are between 3% and 13% slower than the panel-blocked schemes. The left-looking supemode-supemode 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 supemode-supemode 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.

7.3 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 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 matrix BCSSTK15 in Figure 15. 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 supemode. The main reason for the small difference between the overall miss rates of the approaches is that much of the sparse computation does not contain much 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 supemodes that fit in the cache do not derive any cache benefit from copying. As the cache grows, the number of such supemodes 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 of data in the supemode-matrix methods is similar to the extra data movement that is done in the multifrontal method, where supemode entries are added into the update matrix and their final values are later copied back. In fact, we showed 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 moving from a multifrontal method that does not copy to one that does. These numbers assume that supemodes containing a single column are not copied, since any sub-blocks of such supemodes trivially can not interfere with themselves,

The numbers of Table 26 indicate that block copying incurs a moderate cost for the supemode-matrix methods. We also see that data copying incurs a much larger cost for the supemode-supemode methods. The cost is two to three times higher than that of the supemode-matrix method. Note that in both cases the relative

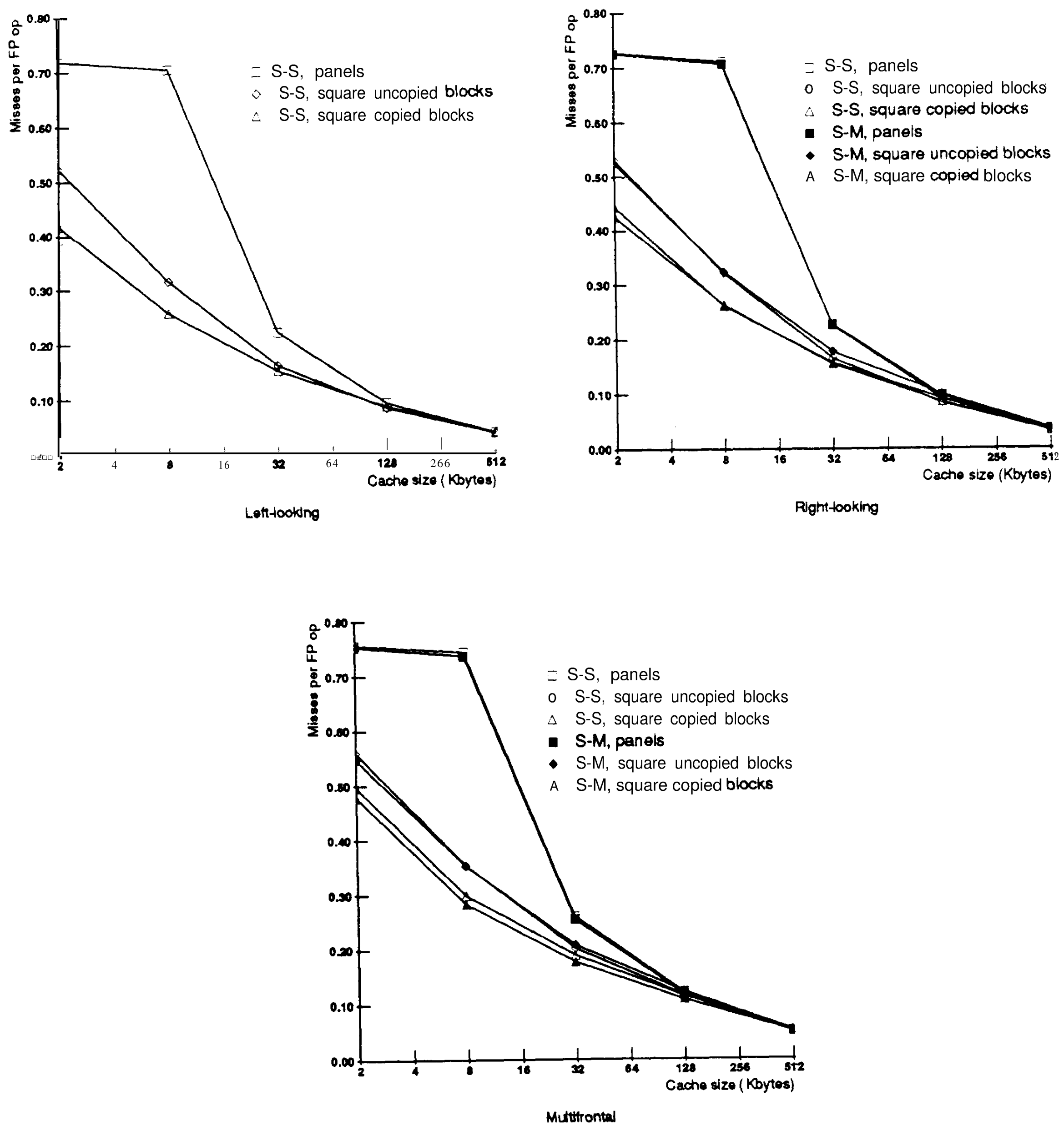

Figure 15: Cache miss behavior for various methods, matrix BCSSTK15. S-S is supemode-supemode, and S-M is supemode-matrix.

Table 26: Increase in memory references due to data copying.

Problem	Supemode-matrix increase	Supemode-supemode 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%

costs of copying would be somewhat larger for right-looking and left-looking approaches, since these methods generate fewer memory references than the multifrontal method. Regarding the trends in the cost of data copying, we have noticed that the relative cost of copying decreases 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 must be used. 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 supemode-supemode 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. We expect that square-block approaches will provide significantly higher performance in the near future. We expect processor speeds to continue to increase, making it possible to solve larger problems in the same amounts of time, Memory densities will certainly continue to increase as well, thus allowing a workstation-class machine to provide enough memory to hold these larger problems. On the other hand, these faster processors will naturally require faster access to cached data items. In order to provide data to these processors extremely quickly, the trend has been towards on-chip caches. Unfortunately, space is very tight on the processor chip, thus forcing these on-chip caches to be relatively small. These two important trends, larger problems and smaller 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.

8 Discussion

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

8.1 Square-Block 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. We now determine 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 8 192 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.

8.2 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 supemodes that contain a single column. The problem of dealing with small supemodes in the multifrontal method has been recognized in the context of vector supercomputers. One solution that has been investigated is *supernode amalgamation* [4, 93, where supemodes with similar structure are merged together to form larger supemodes. 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 supemodes 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 supemodes, so we expect that amalgamation may benefit their performance as well.

Another possible means of improving the performance of the multifrontal method is through the introduction of a special case for single column supemodes. Briefly, one possible approach would be to ignore singleton supemodes when they are visited in the tree traversal. When a non-singleton is visited, it would assemble all update matrices from the subtree rooted at itself in the elimination tree. If singletons were not ignored, then the subtree update matrices would simply be the children's update matrices. The values of the singleton supemodes could be computed during the assembly process, and similarly the updates from these singletons could be added directly into the new update matrix. A special case like this is not nearly as clean as the special cases for the left-looking and right-looking methods, however, and we have not further investigated such an approach.

Yet another approach to improving the performance of the multifrontal method would be to use a hybrid method, as suggested in [17]. 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 wealmesses of the two blended approaches of the hybrid.

8.3 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. The reason has simply been that few distributed-hierarchical-memory multiprocessors have been available in the past. Since column-column primitives have no major disadvantages on machines without memory hierarchies, there was no reason not to base parallel methods on them. This situation has been changing recently, however. As an example, the new Intel iPSC/860 Touchstone multiprocessor is made up of a network of hierarchical-memory processors. Furthermore, we expect that most future high-performance parallel machines will be made up of collections of hierarchical-memory processors. By understanding the performance of column-column primitives on hierarchical-memory machines, we can obtain a better understanding of how much the performance of these parallel approaches will suffer due to the primitives they use.

Based on the results of this paper, it is clear that parallel methods implemented in terms of column-column primitives will achieve quite low performance on hierarchical-memory multiprocessors. However, we believe that the performance of these methods can be greatly improved. Specifically, we believe that these methods can achieve significantly higher performance by distributing sets of columns, or panels, among the processors rather than individual columns, as is typically done. A panel-oriented approach would allow higher-level primitives to be used, yielding higher performance on each processor. We explored such an approach on a moderately parallel multiprocessor (an SGI 4D/380, 8 processor machine) in [20], with quite favorable results. However, a number of difficult questions remain to be answered. While higher-level primitives increase the efficiency of the computations performed on each processor, they also dramatically decrease the amount of concurrency in the problem, and thus decrease the number of processors that can effectively work on the problem at once. This decrease in concurrency was not a problem for the 8 processor machine with the matrices we considered, but it could conceivably become a problem on machines with larger numbers of processors. We are currently investigating this issue.

8.4 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 the paper, a reasonable alternative is to use both approaches within the same code, switching between them on a per-supemode basis, depending on whether copying would provide significant cache miss benefits for the current supemode.

Given the above considerations, we now consider the robustness of each method. Beginning with the left-looking supemode-supemode method, we note that this method contains a certain degree of unpredictability. When supemodes are copied, this method must perform more copying than the supemode-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 supemode copying happens much more frequently.

The right-looking supemode-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 supemode. 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.

9 Conclusions

An obvious end goal of a study like the one performed in this paper 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 study 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.

Acknowledgements

This research is supported by DARPA contract N00014-87-0828. Anoop Gupta is also supported by a Presidential Young Investigator award.

References

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

[2] 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.

[3] Ashcraft, *C.C., A vector implementation of the multi'ontal method for large sparse symmetric positive definite linear systems,* Boeing Computer Services Technical Report ETA-TR-5 1, May, 1987.

[4] Ashcraft, C.C., and Grimes, R.G., 'The influence of relaxed supemode partitions on the multifrontal method", *ACM Transactions on Mathematical Software,* 15(4): 291-309, 1989.

[5] 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.

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

[7] 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.

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

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

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

[11] 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.*

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

[13] George, A., Liu, J., and *Ng,* E., *User's guide for SPARSPAK: Waterloo sparse linear equations package,* Research Report CS-78-30, Department of Computer Science, University of Waterloo, 1980.

[14] 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, 199 1.

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

[16] Liu, J., "On the storage requirement in the out-of-core multifrontal method for sparse factorization", *ACM Transactions on Mathematical Software,* 12(3): *249-264, 1987.*

[17] Liu, J., "The multifrontal method and paging in sparse Cholesky factorization", *ACM Transactions on Mathematical Software,* 15(4): 3 10-325, 1989.

[18] Ng, E.G., and Peyton, B.W., *A supernodal Cholesky factorization algorithm for shared-memory multiprocessors, Oak* Ridge National Laboratory Technical Report ORNL/TM- 118 14, April, 1991.

[19] Rothberg, E., and Gupta, A., "Efficient Sparse Matrix Factorization on Hierarchical Memory Workstations — Exploiting *the* Memory Hierarchy", to *appear in ACM Transactions on Mathematical Software.*

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

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