October 1989 Report No. STAN-CS-89-1286

Fast Sparse Matrix Factorization on Modern Workstations

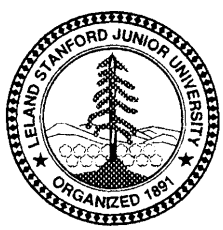
by

Edward Rothberg and Anoop Gupta

Department of Computer Science

Stanford University

Stanford, California 94305



SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form *Approved*
OMB No. 0704-0188

1 **a** REPORT SECURITY CLASSIFICATION

1 b RESTRICTIVE MARKINGS

2a SECURITY CLASSIFICATION AUTHORITY

3 DISTRIBUTION /AVAILABILITY OF REPORT

2b DECLASSIFICATION /DOWNGRADING SCHEDULE

4 PERFORMING ORGANIZATION REPORT NUMBER(S)

5 MONITORING ORGANIZATION REPORT NUMBER(S)

6a NAME OF PERFORMING ORGANIZATION
Computer Science Department
Stanford University

6b OFFICE SYMBOL
(If **applicable)**

7a. NAME OF MONITORING ORGANIZATION

6c. ADDRESS (City, **State,** *and ZIP Code)*
Stanford, CA 94303

7b ADDRESS (City, **State, and** *ZIP Code)*

8a NAME OF FUNDING I SPONSORING ORGANIZATION
DARPA

8b OFFICE SYMBOL
(If applicable)

9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
N00014-87-K-0828

8c. ADDRESS (City, **State, and** *ZIP* **Code)**
Arlington, VA 22161

10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO

1 1 TITLE **(Include** *Security* **Classification)**
Fast Sparse Matrix Factorization on Modern Workstations

12 PERSONAL AUTHOR(S)
Edward Rothberg and Anoop Gupta

13a TYPE OF REPORT	13b TIME COVERED FROM _____ TO _____	14 DATE OF REPORT *(Year, Month, Day)*	15 PAGE COUNT

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (*Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	

19 ABSTRACT *(Continue on reverse if necessary* and *identify by block number)*

The performance of workstation-class machines has experienced a dramatic increase in the recent past. Relatively inexpensive machines which offer 14 MIPS and 2 MFLOPS performance are now available, and machines with even higher performance are not far off. One important characteristic of these machines is that they rely on a small amount of high-speed cache memory for their high performance. In this paper, we consider the problem of Cholesky factorization of a large sparse positive definite system of equations on a high performance workstation. We find that the major factor limiting performance is the cost of moving data between memory and the processor. We use two techniques to address this limitation; we decrease the number of memory references and we improve cache behavior to decrease the cost of each reference. When run on benchmarks from the Harwell-Boeing Sparse Matrix Collection, the resulting factorization code is almost three times as fast as SPARSPAK on a DECStation 3100. We believe that the issues brought up in this paper will play an important role in the effective use of high performance workstations on large numerical problems.

20 DISTRIBUTION /AVAILABILITY OF ABSTRACT
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS

21 ABSTRACT SECURITY CLASSIFICATION

22a NAME OF RESPONSIBLE INDIVIDUAL

22b TELEPHONE *(Include Area Code)*

22c OFFICE SYMBOL

DD Form 1473, JUN 86 *Previous editions are obsolete*

SECURITY CLASSIFICATION OF THIS PAGE

S/N 0102-LF-014-6603

Fast Sparse Matrix Factorization on Modern Workstations

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

October 2, 1989

Abstract

The performance of workstation-class machines has experienced a dramatic increase in the recent past. Relatively inexpensive machines which offer 14 MIPS and 2 MFLOPS performance are now available, and machines with even higher performance are not far off. One important characteristic of these machines is that they rely on a small amount of high-speed cache memory for their high performance. In this paper, we consider the problem of Cholesky factorization of a large sparse positive definite system of equations on a high performance workstation. We find that the major factor limiting performance is the cost of moving data between memory and the processor. We use two techniques to address this limitation; we decrease the number of memory references and we improve cache behavior to decrease the cost of each reference. When run on benchmarks from the Harwell-Boeing Sparse Matrix Collection, the resulting factorization code is almost three times as fast as SPARSPAK on a DECStation 3100. We believe that the issues brought up in this paper will play an important role in the effective use of high performance workstations on large numerical problems.

1 Introduction

The solution of sparse positive definite systems of linear equations is a very common and important problem. It is the bottleneck in a wide range of engineering and scientific computations, from domains such as structural analysis, computational fluid dynamics, device and process simulation, and electric power network problems. We show that with effective use of the memory system hierarchy, relatively inexpensive modem workstations can achieve quite respectable performance when solving large sparse symmetric positive definite systems of linear equations.

Vector supercomputers offer very high floating point performance, and are suitable for use on a range of numerical problems, Unfortunately, these machines are extremely expensive, and consequently access to them is limited for the majority of people with large numeric problems to solve. These people must therefore content themselves with solving scaled-down versions of their problems. In comparison to vector supercomputers, engineering workstations have been increasing in performance at a very rapid rate in recent years, both in integer and floating point performance. Relatively inexpensive machines now offer performance nearly equal to that of a supercomputer on integer computations, and offer a non-trivial fraction of supercomputer floating point performance as well. In this paper we investigate the factors which limit the performance of workstations on the factorization of large sparse positive definite systems of equations, and the extent to which this performance can be improved by working around these limitations.

A number of sparse system solving **codes** have been written and described extensively in the literature [2, 4, 6]. These programs, in general, have the property that they do not execute any more floating point operations than are absolutely necessary in solving a given system of equations. While this fact would make it appear that only minimal performance increases over these codes can be achieved, this turns out not to be the case. The major bottleneck in sparse factorization on a high performance workstation is not the number of floating point operations, but rather the cost of fetching data from main memory. Consider, for example, the execution of SPARSPAK [6] on a DECStation 3100, a machine which uses the MIPS R2000 processor. Between 40% and 50% of all instructions executed and between 50% and 80% of all runtime spent in factoring

a matrix is incurred in moving data between main memory and the processor registers. We use two techniques to decrease the runtirne; we decrease the number of memory to register transfers executed and we improve the program's cache behavior to decrease the cost of each transfer.

We assume that the reader is familiar with the concepts of sparse Cholesky factorization, although we do give a brief overview in section 2. Section 3 describes the benchmark sparse matrices which are used in this study. Section 4 briefly describes the concept of a supemode and describes the supemodal sparse factorization algorithm. Then, in section 5, the characteristics of modem workstations which are relevant to our study are discussed. Section 6 discusses the performance of the supemodal sparse solving code. Then in section 7, the factorization code is tailored to the capabilities of the workstation, and the results of the modifications are presented. Future work is discussed in section 8, and conclusions are presented in section 9.

2 Sparse Cholesky Factorization of Positive Definite Systems

This section provides a brief description of the process of Cholesky factorization of a sparse linear system. The goal is to factor a matrix A into the form $L\,\mathrm{L}^{T}$. The equations which govern the factorization are:

$$l_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2}$$

$$l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk})/l_{jj}$$

Since the A matrix is sparse, many of the entries in L will be zero. Therefore, it is only necessary to sum over those k for which $l_{jk} \neq 0$. The number of non-zero entries in L is highly dependent on the ordering of the row and columns in A. The matrix A is therefore permuted before it is factored, using a fill-reducing heuristic such as nested dissection [5], quotient minimum degree [5], or multiple minimum degree [8].

The above equations lead to two primary approaches to factorization, the *general sparse method* and the *multifrontal method.* The general sparse method can be described by the following pseudo-code:

```
1.  for  j = 1 to n do
2.       for each k s.t. l_jk ≠ 0 do
3.            l_*j — l_*j − l_jk * 1*x-
4.       ljj — √l_jj
5.       for each i s.t. l_ij ≠ 0 do
6.            l_ij — l_ij / l_jj
```

In this method, a column j of L is computed by gathering all contributions to j from previously computed columns. Since step 3 of the above pseudo-code involves two columns, j and k, with potentially different non-zero structures, the problem of matching corresponding non-zeroes must be resolved. In the general sparse method, the non-zeroes are matched by scattering the contribution of each column k into a dense vector. Once all k's have been processed, the net contribution is gathered from this dense vector and added into column j. This is probably the most frequently used method for sparse Cholesky factorization; for example, it is employed in SPARSPAK [6].

The multifrontal method can be roughly described by the following pseudo-code:

```
1.  for k = 1 to n do
2.       l_kk — √l_kk
3.       for each i s.t. l_ik ≠ 0 do
4.            l_ik — l_ik / l_kk
5.       for each j s.t. l_jk ≠ 0 do
6.            1_*j — l_*j − l_jk * 1*x-
```

Table 1: Benchmarks

	Name	Description	Equations	Non-zeroes
1.	D750	Dense symmetric matrix	750	561,750
2.	BCSSTK14	Roof of Omni Coliseum, Atlanta	1,806	61,648
3.	BCSSTK23	Globally Triangular Building	3,134	42,044
4.	LSHP3466	Finite element discretization of L-shaped region	3,466	20,430
5.	BCSSTK15	Module of an Offshore Platform	3,948	113,868
6.	BCSSTK16	Corps of Engineers Dam	4,884	285,494
7.	BCSPWR10	Eastern US Power Network	5,300	16,542
8.	BCSSTK17	Elevated Pressure Vessel	10,974	417,676
9.	BCSSTK18	Nuclear Power Station	11,948	137,142

In the multifrontal method, once a column k is completed it immediately generates all contributions which it will make to subsequent columns. In order to solve the problem of matching non-zeroes from columns `j` and k in step 6, this set of contributions is collected into a dense lower triangular matrix, called the *frontal update matrix.* This matrix is then stored in a separate storage area, called the update matrix stack. When a later column k of L is to be computed, all update matrices which affect k are removed from the stack and combined, in a step called assembly. Column k is then completed, and its updates to subsequent columns are combined. with the as yet unapplied updates from the update matrices which modified k, and a new update matrix is placed on the stack. The columns are processed in an order such that the needed update matrices are always at the top of the update matrix stack. This method was originally developed [3] as a means of increasing the percentage of vectorizable work in sparse factorization. It has the disadvantage that it requires more storage than the general sparse method, since an update matrix stack must be maintained in addition to the storage for L. It also performs more floating point operations than the general sparse scheme.

An important concept in sparse factorization is that of the *elimination tree* of the factor L [7]. The elimination tree is defined by

$$parent(\texttt{j}) = \min\{i | l_{ij} \neq \texttt{0}.\ i > \texttt{j}\}$$

In other words, column `j` is a child of column i if and only if the first sub-diagonal non-zero of column j in L is in row i. The elimination tree provides a great deal of information concerning dependencies between columns. For example, it can be shown that in the factorization process a column will only modify its ancestors in the elimination tree, and equivalently that a column will only be modified by its descendents. Also, it can be shown that columns which are siblings in the elimination tree are not dependent on each other and thus can be computed in any order. The information contained in the elimination tree is used later in our paper to reorder the columns for better cache behavior.

3 Benchmarks

We have chosen a set of nine sparse matrices as benchmarks for evaluating the performance of sparse factorization codes. With the exception of matrix D750, all of these matrices come from the Harwell-Boeing Sparse Matrix Collection [2]. Most are medium-sized structural analysis matrices, generated by the GT-STRUDL structural engineering program. The problems are described in more detail in Table 1. Note that these matrices represent a wide range of matrix sparsities, ranging from the very sparse BCSPWR10, all the way to the completely dense D750. Table 2 presents the results of factoring each of the nine benchmark matrices with the SPARSPAK sparse linear equations package [6] on a DECStation 3100 workstation. All of the matrices are ordered using the multiple minimum degree ordering heuristic. These runtimes, as well as all others which will be presented in this paper, are for 64-bit, IEEE double precision arithmetic. Note that although the machine on which these factorizations were performed is a virtual memory machine, no paging occurred. The factorizations for each of the matrices were performed entirely within the physical memory of the machine. The runtimes obtained with SPARSPAK are used throughout this paper as a point of reference for evaluating alternative factorization methods.

Table 2: Factorization information and runtimes on DECStation 3100.

Name	Nonzeroes in L	Floating point ops (millions)	SPARSPAK runtime (s)	MFLOPS
D750	280,875	141.19	157.25	0.90
BCSSTK14	110.461	9.92	8.10	1.22
BCSSTK23	417,177	119.60	126.97	0.94
LSHP3466	83,116	4.14	3.45	1.20
BCSSTK15	647,274	165.72	172.38	0.96
BCSSTK16	736,294	149.89	150.80	0.99
BCSPWR10	22.764	0.33	0.40	0.82
BCSSTK17	994.885	145.37	136.67	1.06
BCSSTK18	650,777	141.68	148.10	0.96

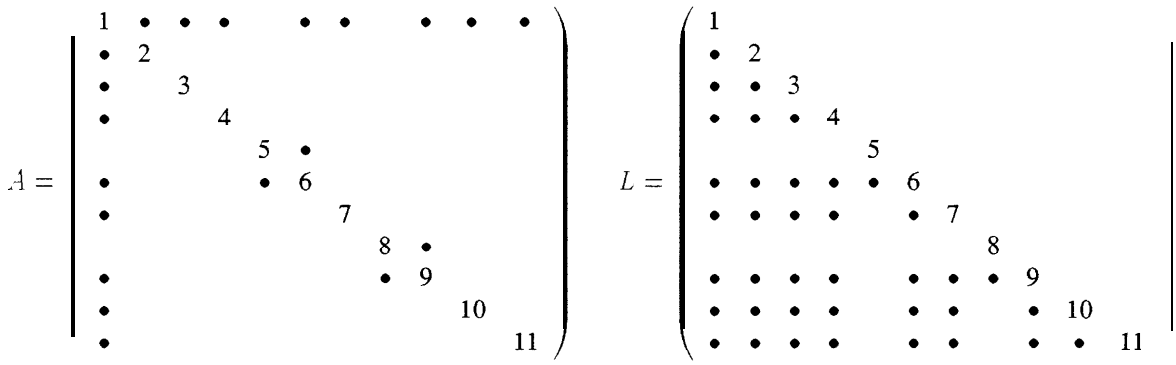


Figure 1: Non-zero structure of a matrix *A* and its factor *L*.

4 Supernodal Sparse Factorization: Reducing the Number of Memory References

The problem of Cholesky factorization of large sparse matrices on vector supercomputers has recently received a great deal of attention. The concept of supernodal elimination, proposed by Eisenstat and successfully exploited in [1, 10], has allowed factorization codes to be written which achieve near-full utilization of vector supercomputers. Supernodal elimination is important in this paper not because it allows high vector utilization, but because it decreases the number of memory references made during sparse factorization. Consequently, it alleviates one of the major bottlenecks in sparse factorization on high performance workstations. This section describes the concept of a supernode, and describes supernodal sparse factorization. It also discusses the reasons for the decrease in the number of memory references when supemodal elimination is employed.

In the process of sparse Cholesky factorization when column j of L modifies column k, the non-zeroes of column j form non-zeroes in corresponding positions of column k. As the factorization proceeds, this unioning of sparsity structures tends to create sets of columns with the same structures. These sets of columns with identical structures are referred to as a *supernudes.* For example, in Figure 1 the set $\{1,2,3,4\}$ of columns forms a supemode. Supemodal elimination is a technique whereby the structure of a matrix's supemodes is exploited in order to replace sparse vector operations by dense vector operations. When a column from a supemode is to update another column, then every column in that supemode will also update that column, since they all have the same structure. In the example matrix, the four columns in supemode $\{1,2,3,4\}$ all update columns 6 through 7 and 9 through 11.

The general sparse super-nodal method of factorization exploits supemodes in the following way. Instead of scattering the contribution of each column of the supemode into the dense vector, as would ordinarily be done in general sparse factorization, the contribution of all columns in the supemode are first combined into a

single dense vector, and that vector is then scattered. Since the storage of the non-zeroes of a single column is contiguous and the columns all have the same structure, this combination can be done as a series of dense vector operations.

The multifrontal factorization method can also be modified to take advantage of supemodal elimination. Where previously each column generated a frontal update matrix, in the *multifrontal supernodal method* each supemode generates one. Consider, for example, the matrix of Figure 1. In the vanilla multifrontal method, columns 1, 2, 3, and 4 would each generate separate update matrices. When it comes time to assemble the contributions of previous columns to column 6, these four update matrices would have to be combined. In contrast, in the multifrontal supemodal method, the contributions of supemode { 1.2.3.4) are combined into a single update matrix. This modification substantially reduces the number of assembly steps necessary. Since the assembly steps are the source of all sparse vector operations in the multifrontal factorization scheme, the net result is a reduction in the number of sparse vector operations done. Note that by reducing the number of assembly steps, supemodal elimination also decreases the number of extra floating point operations performed by the multifrontal scheme.

As stated earlier, a major advantage of supemodal techniques is that they substantially decrease the number of memory references when performing Cholesky factorization on a scalar machine. The reduction in memory to register traffic is due to two factors. First, the supernodal technique replaces a sequence of indirect vector operations with a sequence of direct vector operations followed by a single indirect operation. Each indirect operation requires the loading into processor registers of both the index vector and the values vector, while the direct operation loads only the values vector. Second, the supemodal technique allows a substantial degree of loop unrolling. Loop unrolling allows one to perform many operations on an item once it has been loaded into a register, as opposed to non-unrolled loops, where only a single operation is performed on a loaded item. As will be seen later, the supemodal scheme generates as few as one half as many memory reads and one fourth as many memory writes as the general sparse scheme.

5 Characteristics of Modern Workstations

The machine on which our study is performed is the DECStation 3100 workstation. This machine uses a MIPS R2000 processor and a R2010 floating point coprocessor, both running at 16.7 MHz. It contains a 64K high-speed data cache, a 64K instruction cache, and 16 Megabytes of main memory. The cache is direct-mapped, with 4 bytes per cache line. The machine is nominally rated at 1.6 double precision LINPACK MFLOPS.

The most interesting aspect of DECStation 3100 performance relevant to this study is the ratio of floating point operation latency to memory latency. The MIPS R2010 coprocessor is capable of performing a double precision add in two cycles, and a double precision multiply in five cycles. In contrast, the memory system requires approximately 6 cycles to service a cache miss. Consider, for example, the cost of multiplying two floating point quantities stored in memory. The two numbers must be fetched from memory, which in the worst case would produce four cache misses, requiring approximately 24 cycles of cache miss service time. Once fetched, the multiply can be performed at a cost of five cycles. It is easy to see how the cost of fetching data from memory can come to dominate the runtime of floating point computations. Note that this is an oversimplification, since these operations are not strictly serialized. On the R2000, the floating point unit continues execution on a cache miss, and floating point adds and multiplies can be overlapped.

We believe that the DECStation 3100 is quite representative of the general class of modem high performance workstations. Although our implementation is designed for this specific platform, we believe that the techniques which we discuss would improve performance on most workstation-class machines.

6 Supernodal Sparse Factorization Performance

We now study the performance of the general sparse, the general sparse supemodal, and the multifrontal supemodal methods of factorization on the DECStation 3100 workstation, Our multifrontal implementation differs slightly from the standard scheme, in that no update stack is kept. Rather, when an update is generated, it is added directly into the destination column. Since the update will only affect a subset of the non-zeroes

Table 3: Runtimes on DECStation 3100.

	SPARSPAK			General sparse supemodal			Multifrontal supemodal		
Problem	Time (s)	FP ops (M)	MFLOPS	Time (s)	FP ops (M)	MFLOPS	Time (s)	FP ops (M)	MFLOPS
D750	157.25	141.19	0.90	86.62	141.47	1.63	84.54	140.91	1.67
BCSSTK14	8.10	9.92	1.22	5.06	10.22	1.96	3.73	10.03	2.66
BCSSTK23	126.97	119.60	0.94	75.67	121.25	1.58	68.20	120.48	1.75
LSHP3466	3.45	4.14	1.20	2.26	4.33	1.83	1.90	4.19	2.18
BCSSTK15	172.38	165.72	0.96	101.40	167.88	1.63	87.84	166.67	1.89
BCSSTK16	150.80	149.89	0.99	92.10	152.88	1.63	71.74	151.51	2.09
BCSPWR10	0.40	0.33	0.82	0.46	0.34	0.71	0.39	0.32	0.84
BCSSTK17	136.67	145.37	1.06	85.66	148.76	1.70	62.35	146.94	2.33
BCSSTK18	148.10	141.68	0.96	89.70	143.78	1.58	77.46	142.62	1.83
Avg. (of 7)			1.05			1.70			2.10

in the destination column, a search is now required in order to locate the appropriate locations into which the update values must be added. The multifrontal method is modified in this way because this paper studies in-core factorization techniques, and the extra storage costs incurred in keeping an update stack would severely limit the size of problems which could be solved on our machine. This modification trades increased work due to index searching for less storage space and fewer floating point operations. It is not clear whether this scheme would be faster or slower than a true multifrontal scheme.

Table 3 gives the runtimes of the two supemodal schemes compared with those of the general sparse scheme, as employed in SPARSPAK, for the nine benchmark matrices. Note that in order to avoid skewing the average MFLOPS figures, the averages reported in the table do not include either the number for the most dense matrix, D750, or for the least dense, BCSPWRlO. Also note that in this and subsequent tables, the MFLOPS rate of a factorization method on a particular matrix is computed by dividing the number of floating point operations executed by SPARSPAK when factoring the matrix by the runtime for the method. Thus, the MFLOPS numbers take into account any floating point overhead introduced by the different factorization schemes. As can be seen from the table, however, these overheads are relatively small, and for some matrices the multifrontal supernodal scheme actually executes fewer floating point operations than SPARSPAK.

Since the standard general sparse code of SPARSPAK executes substantially more memory operations than the general sparse supemodal code, the latter is understandably much faster. However, since the multifrontal supemodal scheme executes essentially the same number as the general sparse supernodal scheme, it is somewhat surprising that their runtimes differ by such a large amount. In order to better explain this difference, Table 4 presents the total number of cache misses which must be serviced, as well as the number of memory references which are generated, when solving each of the above problems using a direct-mapped 64 Kbyte cache with 4 byte lines, as found on the DECStation 3100. Note that the memory reference numbers count one word, or 32-bit, references. A fetch of a double precision number, which is a 64-bit entity, is counted as two references. It is clear from this table that the general sparse supemodal scheme generates substantially more cache misses than the multifrontal supemodal scheme in factoring a given matrix, and thus spends much more time servicing cache misses.

We now consider how the above factorization schemes interact with the processor cache, in order to better understand the cache miss numbers in the table and also to give a better understanding of how the schemes can be modified to make better use of the cache. All of the factorization methods execute essentially the same number of floating point operations, and almost all of these occur in either a DAXPY or a DAXPYI loops. A DAXPY loop is a loop of the form $y \leftarrow a * x + y$, where x and y are double precision vectors and a is a constant. A DAXPYI, or indirect DAXPY, loop is of the form $y \leftarrow a * x(index) + y$, where x and y are again double precision vectors, a is a constant, and $index$ is a vector of indices into x. Since almost all cache misses come from fetching the x and y vectors in loops of this sort, we base our intuitive analysis of caching behavior on how many of these two operands we expect will miss in the cache.

We first consider the general sparse method. Recall that in this method, updates from all source columns to

Table 4: Cache behavior (numbers are in millions).

	SPARSPAK		General sparse supemodal		Multifrontal supernodal	
Problem	Refs	Misses	Refs	Misses	Refs	Misses
D750	356.07	141.67	161.77	148.44	159.55	146.80
BCSSTK14	26.00	4.29	14.31	4.33	14.28	1.36
BCSSTK23	303.57	112.11	150.40	117.58	154.22	98.60
LSHP3466	11.24	1.22	6.75	1.17	6.65	0.64
BCSSTK15	421.39	151.48	206.47	156.32	208.62	120.47
BCSSTK16	382.77	123.98	194.42	131.70	194.19	82.47
BCSPWR10	1.02	0.21	0.90	0.21	0.94	0.15
BCSSTK17	374.28	106.09	194.07	105.40	193.17	53.85
BCSSTK18	361.25	127.96	181.22	133.90	187.19	102.74

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

Figure 2:

a given destination column are first scattered into a single dense vector, and then the net update is gathered into the destination column. SPARSPAK maintains with each column j, a list of columns which have a non-zero entry in row j. A column is a member of this list if the next column which it is to update is column j. Thus, a column can only be a member of one such list at a time. After a column i is used to update column j, it is placed at the head of the list of columns which will update column k, where k is the location of the next non-zero entry in column i. The list of columns which will update some column j is therefore sorted by the order in which the update columns were last used. The general sparse method therefore performs greedy caching, in the sense that when a column is updated, the set of columns which will update it are applied in the order in which they were last used, and thus were most recently in the cache.

The main limitation of this caching scheme is apparent if we consider two adjacent columns which have similar sets of columns which update them. For example, consider columns 6 and 7 of Figure 2. Once column 6 has been completed, the cache will be loaded with some set of entries from columns 1 through 5, the columns which modified 6. If the size of the set of columns which modified column 6 is larger than the cache, then the set of entries remaining in the cache will be a subset of the entries which modified 6, and thus only a portion of the set of columns needed to update column 7 will be in the cache. Therefore, sets of columns which have large, similar sets of update columns get little benefit from the processor cache. In order to get an intuitive feel for how many misses will be incurred, we note that the vast majority of the factoring time is spent in scattering the various source columns to the dense update column. One would expect the dense column to be present in the cache, and that for large problems most of the update columns would not be present. Thus, one would expect that for most DAXPY loops, one of the operands would not be in cache.

The general sparse approach with supemodes is quite similar to the vanilla general sparse approach in terms of cache behavior. The behavior differs slightly, due to two factors. First, since the supemodal approach makes half as many memory accesses, one would expect it to perform better than the general sparse approach when the cache is relatively small and subject to frequent interference. On the other hand, the greedy caching strategy is not as effective in the supemodal case. If the greedy scheme indicates that a supemode was most recently in the cache, and that supemode is much larger than the cache, then only the tail end of the supemode will actually be present in the cache. However, the supemode will subsequently be processed starting from the head, and thus will get no caching benefit. In this respect, the general sparse supemodal approach exhibits worse caching behavior than the general sparse approach.

The multifrontal supemodal approach generates modifications to subsequent columns at the time the modifier supernode is completed. Consider supemode $\{1,2,3,4\}$ of Figure 2, and assume that all of its non-zeroes fit in the cache. The updates to subsequent columns are generated by adding each of the four columns in the supemode into a single update vector. If we assume that the update vector is not present in the cache, then we find that of the five operands in the four DAXPY loops (the four columns in the supemode and the update vector), only one operand will miss in the cache. In the multifrontal case, the cache behavior deteriorates when the supemode does not fit in the cache, eventually degenerating to one operand missing for each DAXPY loop, just as in the general sparse supemodal case. The reason for the multifrontal supemodal method's superior cache performance is simply that it is more likely for the supemode in the multifrontal method to fit in the cache than for a set of update columns in the general sparse method to fit.

Thus in summary, although the three schemes for sparse Cholesky factorization which have been discussed perform essentially the same set of floating point operations, they exhibit substantially different cache behaviors. The general sparse and general sparse supemodal schemes both use the cache in a greedy way, always attempting to access columns in the order in which they were most recently present in the cache. The multifrontal supemodal scheme, on the other hand, loads the cache with the non-zeroes of an entire supemode and reuses these values in computing updates to subsequent columns. Although the multifrontal supemodal scheme makes better use of the cache than the other two schemes, all three schemes rapidly degenerate as the size of the problem grows relative to the size of the cache. They all eventually reach a point where the cache provides little benefit.

7 Modification of Cholesky Factorization for Hierarchical Memory Systems

From the previous section's discussion, it is clear that there are two main sources of cache misses in Cholesky factorization: (i) adjacent columns which make use of dissimilar sets of columns, and (ii) supemodes which do not fit in the processor cache. If a number of consecutive columns make use of dissimilar sets of columns, it is clearly unlikely that the cache will be well utilized. The cache will be loaded with the set of columns used for one column, and subsequently loaded with an entirely different set of columns used for the next column. Similarly, supemodes which do not fit in the cache present difficulties for effective use of the cache. The non-zeroes in the supemode are used a number of times in succession. If the supemode doesn't fit in the cache, then each time the supemode is used it must be reloaded into the cache. This section presents modifications which are meant to deal with these problems.

In order to increase the locality of reference in processing a sequence of columns, the structure of the elimination tree is examined. As was discussed in section 2, a column will only modify its ancestors in the elimination tree. In order to increase locality, therefore, it is desirable to process columns with common ancestors in close succession. In this way, the ancestor columns are loaded into the cache and hopefully reused between one column and the next. One way to group columns with common ancestors together is to process columns from the same subtree of the elimination tree together. This order can be achieved by processing the columns in the order in which they would appear in a post-order traversal of the elimination tree. As was discussed before, such a reordering of the computation does not alter the computation since it only changes the order in which independent siblings of the elimination tree are processed.

Such a reordering was employed in [9] in order to decrease the amount of paging done when performing Cholesky factorization on a virtual memory system. The use of this reordering to reduce cache misses is clearly the same idea applied to a different level of the memory hierarchy. That is, in [9] anything which is not present in main memory must be fetched from the slow disk drive. In our case, anything which is not present in the cache must be fetched from the slow main memory. The improvement gained from performing the multifrontal supemodal factorization with a reordering of the computation based on a post-order traversal of the elimination tree turns out to be quite modest. The runtimes for the smaller test matrices are decreased by at most ten percent, and the runtimes for the large matrices are reduced by at most a few percent. The runtime differences aren't substantial enough to merit a new table. The reduction in cache misses due to this modification, which we call the *reordered multifrontal supernodal method,* will be presented in a future table.

Since the remaining source of poor cache behavior in the multifrontal supemodal scheme is the presence of supemodes which do not fit in the cache, the criteria for adding a column to a supemode is now modified. Initially, a column was a member of the current supemode if it had the same structure as the previous column. In order to improve the caching behavior of the multifrontal supernodal approach, we now require that the column must also satisfy the condition that, if it were added to the current supemode, the non-zeroes of the resulting supemode would fit in the cache. We call this the *multifrontal bounded-supernodal method.* Note that while this modification will improve cache performance, it will also increase the number of memory references, since it increases the number of supemodes.

As can be seen from Table 5 and Table 6, our expectations were correct. Cache misses are greatly reduced, while references are slightly increased, with an overall result of substantially shorter run times. The next modification attempts to exploit both the reduced number of memory references resulting from large supemodes and the improved cache behavior resulting from limiting the size of supemodes. In the supemodal multifrontal approach, the innermost routine is the creation of the frontal update matrix. This lower triangular dense matrix contains the updates of a supemode to all columns which depend on it. In the bounded-supemode approach, we split any supemode whose non-zeroes do not fit in the cache into a number of smaller supemodes. In our new approach, any such supemode is not split into multiple supemodes. It is still considered to be a single supemode, but is partitioned into a number of chunks corresponding to the smaller subset supemodes. The update matrix is now generated by computing the contribution of each chunk, one at a time, and then adding the contributions together into an update matrix for the entire supemode. The resulting update matrix is then distributed to those columns which are affected. In this way we maintain the caching behavior of the bounded-supemode approach, since each chunk fits in the cache, yet we maintain the full supemodal structure of the multifrontal supemodal approach. We call this the *multifrontal partitioned-supernodal method.* The only disadvantage of this approach

Table 5: Runtirnes on DECStation 3100.

	SPARSPAK		General sparse supemodal		Multifrontal supernodal		Multifrontal bounded-supemodal	
Problem	Time (s)	MFLOPS	Time (s)	MFLOPS	Time (s)	MFLOPS	Time (s)	MFLOPS
D750	157.25	0.90	86.62	1.63	84.54	1.67	58.22	2.43
BCSSTK14	8.10	1.22	5.06	1.96	3.73	2.66	3.62	2.74
BCSSTK23	126.97	0.94	75.67	1.58	68.20	1.75	45.13	2.65
LSHP3466	3.45	1.20	2.26	1.83	1.90	2.18	1.79	2.32
BCSSTK15	172.38	0.96	101.40	**1.63**	87.84	1.89	58.73	2.82
BCSSTK16	150.80	0.99	92.10	1.63	71.74	2.09	51.29	2.92
BCSPWR10	0.40	0.82	0.46	0.71	0.39	0.84	0.37	0.89
BCSSTK17	136.67	1.06	85.66	1.70	62.35	2.33	48.85	2.98
BCSSTK18	148.10	0.96	89.70	1.58	77.46	1.83	53.06	2.67
Avg. (of 7)		1.05		1.70		2.10		2.73

Table 6: Cache behavior (numbers are in millions).

	SPARSPAK		General sparse supemodal		Multifrontal supernodal		Multifrontal bounded-supemodal	
Problem	Refs	Misses	Refs	Misses	Refs	Misses	Refs	Misses
D750	356.07	141.67	161.77	148.44	159.55	146.80	200.50	35.57
BCSSTK14	26.00	4.29	14.3 1	4.33	14.28	1.36	14.3 1	0.80
BCSSTK23	303.57	112.11	150.40	117.58	154.22	98.60	170.19	22.83
LSHP3466	11.24	1.22	6.75	1.17	6.65	0.64	6.65	0.36
BCSSTK15	421.39	15 1.48	206.47	156.32	208.62	120.47	227.92	26.89
BCSSTK16	382.77	123.98	194.42	131.70	194.19	82.47	205.05	19.71
BCSPWRlO	1.02	0.21	0.90	0.21	0.94	0.15	0.92	0.09
BCSSTK17	374.28	106.09	194.07	105.40	193.17	53.85	199.43	15.42
BCSSTK18	361.25	127.96	181.22	133.90	187.19	102.74	203.57	24.32

Table 7: Runtimes on DECStation 3100.

	SPARSPAK		General sparse supemodal		Multifrontal supemodal		Multifrontal partitioned-supemodal	
Problem	Time (s)	MFLOPS	Time (s)	MFLOPS	Time (s)	MFLOPS	Time (s)	MFLOPS
D750	157.25	0.90	86.62	1.63	84.54	1.67	40.51	3.49
BCSSTK14	8.10	1.22	5.06	1.96	3.73	2.66	3.57	2.78
BCSSTK23	126.97	0.94	75.67	1.58	68.20	1.75	38.63	3.10
LSHP3466	3.45	1.20	2.26	1.83	1.90	2.18	1.79	2.32
BCSSTK15	172.38	0.96	101.40	1.63	87.84	1.89	50.83	3.26
BCSSTK16	150.80	0.99	92.10	1.63	71.74	2.09	47.44	3.16
BCSPWR10	0.40	0.82	0.46	0.71	0.39	0.84	0.35	0.94
BCSSTK17	136.67	1.06	85.66	1.70	62.35	2.33	46.80	3.11
BCSSTK18	148.10	0.96	89.70	1.58	77.46	1.83	46.36	3.06
Avg. (of 7)		1.05		1.70		2.10		2.97

is that the entire frontal update matrix must now be stored, as opposed to the previous approaches where only the updates to a single column had to be stored.

Table 7 presents the results of this modification, As can be seen from the table, the multifrontal partitioned-supernodal approach achieves very high performance, typically doing the factorization at almost three times the speed of SPARSPAK. The combination of the decreased memory references from supemodal elimination and the improved cache hit rate of supemode splitting yields an extremely efficient factorization code which performs sparse factorization of large systems at more than three double precision MFLOPS on the DECStation 3100.

In order to provide a more detailed picture of how the various factorization schemes which have been described interact with the processor cache, Figure 3 presents graphs of the number of cache misses incurred in factoring four of the test matrices as a function of cache size using each of the schemes. Interestingly, even though these four matrices have substantially different sparsities (see Table 1), the graphs appear quite similar. The cache for these graphs is again direct-mapped, with 4 byte lines. Note that the range of cache sizes depicted in these graphs falls within the limits of caches which one might reasonably expect to encounter. A one kilobyte cache might be found in an on-chip cache, where space is extremely tight. A one megabyte cache, on the other hand, would not be unreasonable in a very large machine.

In examining Figure 3, the question arises of what factors determine the cache behavior when working with a particular matrix. While the exact behavior is extremely difficult to predict, the general shape of the curve for the partitioned-supernodal scheme can be justified by an intuitive explanation. The partitioned-supemodal scheme depends on the ability to break supemodes into smaller chunks in order to improve cache behavior. The caching benefit of breaking them up comes from the ability to read a subsequent column into the cache and apply many column updates to it once it has been fetched. Clearly, if the largest piece of a supemode that will fit in the cache at one time is a single column, then no caching benefit is realized. Applying a single column update per fetch is exactly what would be done without the supemode splitting modification. The initial sections of the curves in Figure 3, where the cache miss reducing schemes perform no better than the other schemes, correspond to cache sizes in which no more than one matrix column will fit. In the graph for problem BCSSTK15, for example, no benefit is achieved for cache sizes of less than 4 kilobytes.

As the size of the cache grows, more columns of the supemodes fit in the cache. For the multifrontal supernodal schemes which do not break up supemodes, the number of misses declines gradually as more and more supemodes fit wholly in the cache. For the schemes which do break up supemodes, however, the number of misses declines much more quickly due to the reuse of fetched data. As was discussed earlier in this section, if only one column fits in the cache, then one operand is expected to cache miss for each DAXPY loop. If two columns fit, however, we would expect one operand to miss for every two DAXPY loops. This effect can be seen in the graphs of Figure 3; once the cache is large enough to contain more than one column of the supemodes, a doubling in the size of the cache results in a near halving in the number of cache misses. Note that this is somewhat of an oversimplification for a number of reasons. One reason is that doubling the size of the cache will have little effect on supemodes which already fit entirely in the cache. Another is that the

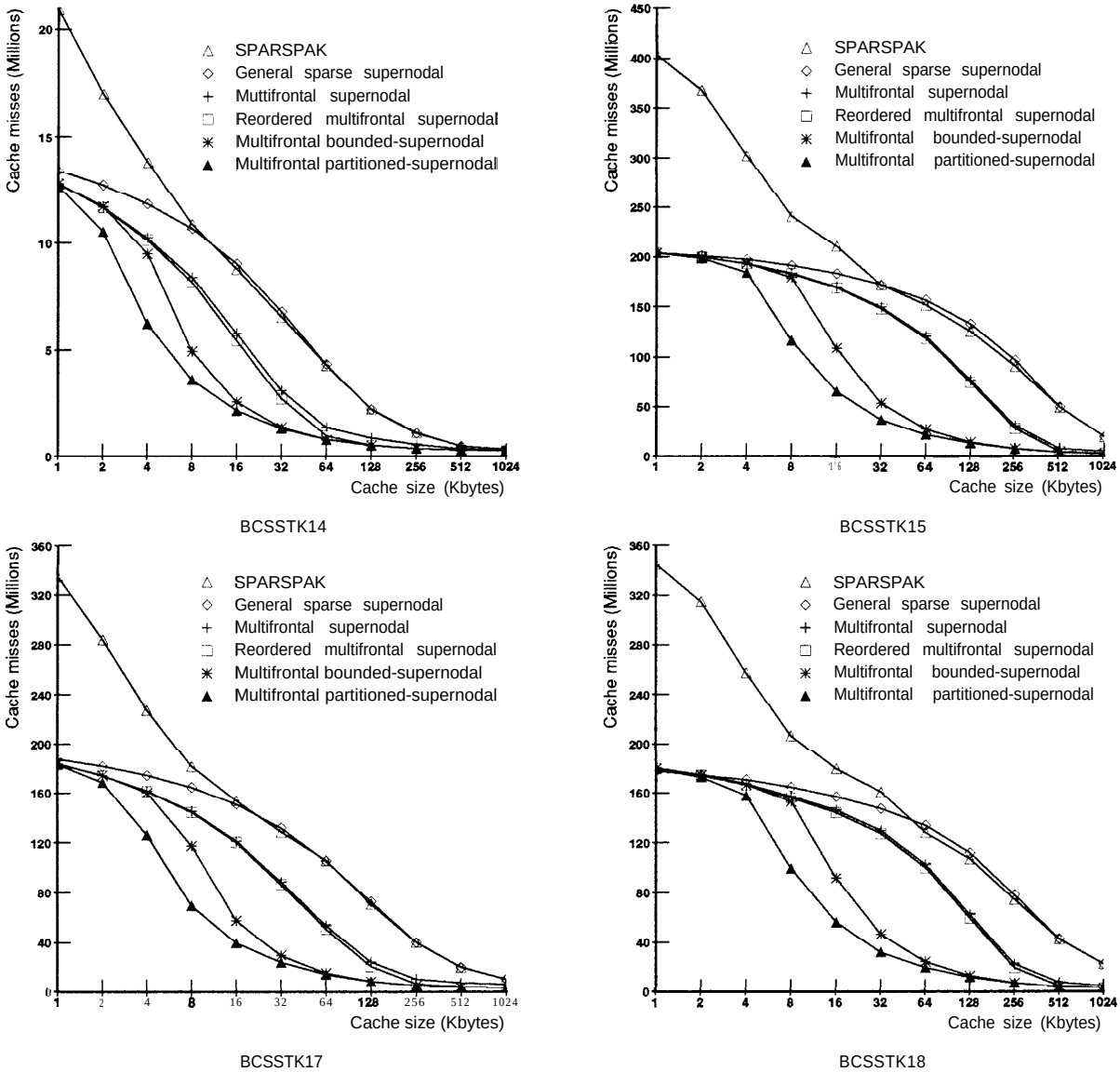


Figure 3: Cache misses (in millions).

number of non-zeroes per columns is not the same across supemodes, so that the point at which more than one column fits varies for different supemodes. All of the schemes eventually reach a point at which the cache is large enough so that the only cache misses incurred are due to data items which interfere with each other in the cache or due to data items which have never before been accessed.

8 Discussion

The performance gap between vector supercomputers and low cost workstations is definitely narrowing. In Table 8 we compare the performance obtained in our study of sparse factorization on a workstation with the performance obtained on a single processor of the CRAY Y-MP, as reported in [10]. As can be seen from this table, using the techniques described in this paper, an inexpensive workstation based on the MIPS R2000 and R2010 running at 16.7 MHz, such as the DECStation 3 100, can perform sparse factorization at approximately one-sixtieth of the speed of the CRAY Y-MP. Of this factor of sixty, a factor of approximately ten is due to the CRAY's faster clock speed, and a factor of approximately six is due to the CRAY's vector architecture and multiple functional units. It is likely that both of these factors will decrease when we compare future

Table 8: Comparison of DECStation 3 100 and CRAY Y-MP.

	DECStation 3 100		CRAY Y-MP[1]		
Name	Time (s)	MFLOPS	Time (s)	MFLOPS	Ratio
BCSSTK23	38.63	3.10	0.62	191.57	61.8
BCSSTK15	50.83	3.26	0.84	197.74	60.7
BCSSTK16	47.44	3.16	0.79	190.78	60.4

vector supercomputers to future workstations. Consider the factor of ten due to the CRAY's faster clock speed. Next generation microprocessors will be significantly faster than the 60 nanosecond cycle time of the R2000 used in the DECStation 3100. The Intel i860 microprocessor, for example, is currently available with a 25 nanosecond cycle time, and a version with a 20 nanosecond cycle time is not far off. Furthermore, prototype ECL microprocessors with 10 nanosecond cycle times have been developed, and such microprocessors will probably be generally available in a few years. Next generation vector supercomputers, on the other hand, will most likely experience a much less dramatic decrease in cycle time. The remaining factor of six due to the CRAY's machine architecture is expected to decrease as well, as future microprocessors (e.g., i860, iWarp) move to superscalar architectures with multiple functional units and floating point pipelines that produce a result per clock cycle.

An important item to note about performing sparse factorization on a vector supercomputer is that the matrix reordering step, which is necessary in order to reduce fill in the factor, is done entirely in scalar mode. It therefore achieves poor utilization of the vector hardware and typically consumes as much time as the numerical factorization. In cases where a number of matrices with identical structures are to be factored, the reordering time can be amortized over a number of subsequent factorizations. However, in some cases the linear system need only be solved once. In such cases, a CRAY Y-MP would only be approximately thirty times as fast as an R2000-based workstation, since the R2000-based machine can perform the ordering almost as quickly as the CRAY.

9 Future Work

One issue which requires further study is that of improving cache behavior when the cache is too small for the techniques discussed here to have any benefit. As can be seen in Figure 3, these techniques only produce an improvement in cache behavior when the cache is larger than a certain size, the size depending on the matrix being factored. By splitting columns of the matrix into sub-columns, and performing similar techniques, it may be possible to substantially reduce the number of cache misses incurred for much smaller caches, at a cost of increased computational overhead and more memory references. This issue was not investigated here because the machine on which the study was performed had a sufficiently large cache that such a modification was not necessary for the matrices which were used.

Another issue which merits further investigation is the effect of varying the characteristics of the processor cache on the overall cache behavior. This paper has studied the behavior of a direct-mapped cache with a 4 byte line size. It would be interesting to observe the effect of varying the set-associativity or line size of the cache, for both the factorization codes which attempt to reduce cache misses and for those that do not, in order to discover to what extent the differences observed here would carry over to different types of caches.

Another interesting related issue is that of factorization on a machine with a multi-level cache. Numerous current machines have multiple levels of cache. For example, a machine might have a small, on-chip cache, and a larger, slower second level cache. Further investigation is necessary in order to determine how the results which have been presented in this paper would apply to such a machine. While it is clear that one could choose a particular level of the cache hierarchy at which to decrease cache misses and ignore the other levels, it is not clear which level should be chosen or whether it might be possible to achieve higher performance by taking more than one level of the cache into account.

[1]The early CRAY Y-MP which was used in [10] had a 6.49 nanosecond cycle time. More recent Y-MP's have a 6 nanosecond cycle time. In order to estimate the computation rate of the current CRAY Y-MP, we have adjusted the MFLOPS and runtime numbers reported in [10] to take the faster clock rate into account.

In this paper, only in-core factorization techniques have been studied. Thus, the size of matrix which could be studied was limited by the amount of physical memory which the machine contained. A number of out-of-core techniques have been described in the literature. They all, however, introduce a substantial amount of added complexity to the factorization program, since the programmer must deal with explicitly loading needed sections of the matrix from disk, and off-loading unneeded sections. We hope to study the effectiveness of a virtual memory system, guided by hints from the program, in dealing with this problem. The main constraint in using the paging facility of a virtual memory system is the fact that the program blocks when a location which must be fetched from disk is accessed With the ability to pm-fetch pages from disk, it may be possible to avoid the blocking associated with a virtual memory system. It may also be more efficient to deal with data in memory-page size chunks, which the virtual memory system is optimized to handle, rather than explicitly manipulating rows and columns of the matrix. A relatively simple modification to the factorization code could potentially allow full utilization of the processor on extremely large matrices.

We also hope to study the impact of the reduced memory traffic achieved in this paper on parallel sparse factorization on a shared-memory multiprocessor. The traditional bottleneck in a bus-based shared-memory machine is the bandwidth of the shared bus. By using cache-miss reduction techniques to reduce the bandwidth requirements of each processor, it should be possible to effectively use more processors on the same bus. Similarly, in a network-based shared-memory machine such as the Stanford DASH multiprocessor, a reduction in the cache miss rate of each of the cooperating processors should reduce the load on the interconnect network.

10 Conclusions

In this paper, we have demonstrated that the bottleneck in executing existing sparse Cholesky factorization codes on modem workstations is the time spent in fetching data from main memory. The floating point hardware in these machines is sufficiently fast that the time spent in performing the floating point calculations is a small fraction of the total runtime. We have proposed a number of new techniques for factoring these large sparse symmetric positive definite matrices. The intent of these techniques has been to improve performance by reducing the number of memory fetches executed and by improving cache behavior in order to reduce the cost of each fetch.

The techniques which we used in order to improve sparse factorization performance were based on the concept of supemodal elimination, a concept originally utilized to improve the performance of vector supercomputers on sparse factorization problems. Supernodal elimination allowed us to decrease the number of memory references executed, and also led to a method which reduced the number of cache misses incurred in the factorization. The result is an extremely efficient sparse factorization code; on a DECStation 3 100 workstation we achieve more than three double precision MFLOPS in factoring a wide range of large sparse systems. This is almost three times the performance of the popular SPARSPAK sparse linear equations package. In achieving this performance, we have shown that a very simple memory system can be exploited extremely effectively when performing sparse Cholesky factorization. At this level of performance, we believe that performance is limited by the processor, not by the memory system. Considering the high cost of main memory accesses on this machine, this is not what we would have expected. We have also shown that it is extremely important to exploit the characteristics of the memory system in order to achieve high performance. Modem workstations rely heavily on high-speed cache memory for their high performance, and programs which are modified to make better use of this hierarchical memory design will achieve substantially higher performance.

Acknowledgements

We would like to thank Roger Grimes at Boeing for sending us a copy of the Harwell-Boeing Sparse Matrix Collection, and we would like to thank all of the contributors to the collection for making these matrices available. We would also like to thank Esmond Ng for providing us with a copy of the SPARSPAK sparse linear equations package. This research is supported by DARPA contract N00014-87-K-0828. Edward Rothberg is also supported by an Office of Naval Research graduate fellowship. Anoop Gupta is also supported by a faculty award from Digital Equipment Corporation.

References

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

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

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

[4] Eisenstat, S., Schultz, M., and Sherman, A., "Algorithms and data structures for sparse symmetric Gaussian elimination", *SIAM Journal on Scientific and Statistical Computing, 2: 225-237,* 1981.

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

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

[7] Liu, J., "A compact row storage scheme for Cholesky factors using elimination trees", *ACM Transactions on Mathematical Software, 12: 127-148, 1986.*

[8] Liu, J., "Modification of the minimum degree algorithm by multiple elimination", *ACM Transactions on Mathematical Software,* 11: *141-153, 1985.*

[9] Liu, J., "A note on sparse factorization in a paging environment", *SIAM Journal on Scientific and Statistical Computing, 8:* 1085-1088, 1987.

[10] Simon, H., Vu, P., Yang, C., *Performance of a supernodal general sparse solver on the CRAY Y-MP: 1.68 GFLOPS with autotasking,* Technical Report SCA-TR-117, Boeing Computer Services, 1989.