June 1990

Report No. STAN-CS-90-1318

Cross-Listed with the Computer Systems Lab

Techniques for Improving the Performance of Sparse Matrix Factorization on Multiprocessor Workstations

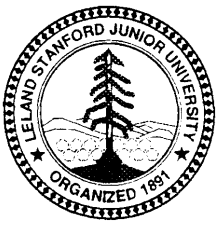
by

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

Techniques for Improving the Performance of Sparse Matrix Factorization on Multiprocessor Workstations

5. FUNDING NUMBERS

87-K-0828

6. AUTHOR(S)

Edward Rothberg and Anoop Gupta

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Computer Science Department
Stanford University
Stanford, CA 94305

8. PERFORMING ORGANIZATION REPORT NUMBER

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 look at the problem of factoring large sparse systems of equations on high-performance multiprocessor workstations. While these multiprocessor workstations are capable of very high peak floating point computation rates, most existing sparse factorization codes achieve only a small fraction of this potential. A major limiting factor is the cost of memory accesses performed during the factorization. In this paper, we describe a parallel factorization code which utilizes the supemodal structure of the matrix to reduce the number of memory references. We also propose enhancements that significantly reduce the overall cache miss rate. The result is greatly increased factorization performance. We present experimental results from executions of our codes on the Silicon Graphics 4D/380 multiprocessor. Using eight processors, we find that the supernodal parallel code achieves a computation rate of approximately 40 MFLOPS when factoring a range of benchmark matrices. This is more than twice as fast as the parallel nodal code developed at the Oak Ridge National Laboratory running on the SGI 4D/380.

14. SUBJECT TERMS			15. NUMBER OF PAGES 12
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRAC

Techniques for Improving the Performance of Sparse Matrix Factorization on Multiprocessor Workstations

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

June 22, 1990

Abstract

In this paper we look at the problem of factoring large sparse systems of equations on high-performance multiprocessor workstations. While these multiprocessor workstations are capable of very high peak floating point computation rates, most existing sparse factorization codes achieve only a small fraction of this potential. A major limiting factor is the cost of memory accesses performed during the factorization. In this paper, we describe a parallel factorization code which utilizes the supemodal structure of the matrix to reduce the number of memory references. We also propose enhancements that significantly reduce the overall cache miss rate. The result is greatly increased factorization performance. We present experimental results from executions of our codes on the Silicon Graphics 4D/380 multiprocessor. Using eight processors, we find that the supemodal parallel code achieves a computation rate of approximately 40 MFLOPS when factoring a range of benchmark matrices. This is more than twice as fast as the parallel nodal code developed at the Oak Ridge National Laboratory running on the SGI 4D/380.

1 Introduction

As microprocessors become more and more powerful, the set of domains in which they can be effectively used continues to grow. Inexpensive microprocessors now offer performance exceeding that of supercomputers for integer computations, and offer nearly equal performance on scalar floating point computations (e.g., Intel i860, IBM RS/6000). The one domain where vector supercomputers continue to prevail, however, is in vectorizable floating point computations. Inexpensive machines have been making inroads into this domain as well, as multiprocessor systems based on high-performance microprocessors are developed. These multiprocessors typically have very high theoretical floating point computation rates. In practice, however, these rates are difficult to realize. A number of factors, including high cache-miss rates, contention for the shared bus and main memory, and the difficulty of distributing work, usually limit the attained computation rates.

It is therefore important to consider how various numerical problems can be effectively solved on such machines. Just as vector supercomputers required the recasting of numerical algorithms to better fit the vector architecture of these machines, parallel machines also necessitate a restudying of these applications in order to deal with a new set of bottlenecks. In this paper, we study the commonly occurring and important problem of factoring a large sparse positive definite matrix. We look at how this problem can be attacked on a multiprocessor. In order to make our results concrete, we perform experiments on the Silicon Graphics 4D/380, a modestly parallel machine containing eight MIPS R3000/R3010 processors.

The main technique we use for improving sparse factorization performance is that of supernodal elimination [2]. By exploiting the fact that adjacent columns in the sparse factor frequently have identical non-zero structures, supernodal elimination allows one to perform operations with these columns more efficiently. A factorization code which employs supemodal elimination can perform the factorization with many fewer memory references and cache misses than a nodal code. Since the memory system is the main bottleneck in sparse factorization on a high-performance workstation, the result is substantially improved performance [12]. A sequential supernodal

code runs more than twice as fast as SPARSPAK [9] on a single processor of the SGI 4D/380.

When incorporating supernodal elimination into a parallel code, a substantial load balancing problem arises. The task grain size which results from a parallel supernodal scheme is often too large to allow effective distribution of work to the processors. We describe a solution to the problem which involves heuristically splitting the supemodes into smaller pieces in order to decrease the grain size. The parallel code that we develop performs sparse factorization at a rate of approximately 40 MFLOPS when using 8 processors of the SGI 4D/380. This is more than twice the performance obtained with the parallel factorization scheme developed at the Oak Ridge National Laboratory [8].

This paper is organized as follows. In section 2, we discuss preliminaries. We give a brief overview of two different approaches to sparse factorization, we describe the machine on which our study is performed, and we describe the benchmark matrices which we use to evaluate factorization performance. In section 3, we describe the parallel factorization scheme developed at the Oak Ridge National Laboratory, and present performance numbers. Then in section 4, we briefly discuss supemodal techniques, and we describe a sequential factorization implementation which uses these techniques. In section 5, we discuss a parallel implementation of the supemodal factorization code. We discuss several issues which arise when supemodes are used in a parallel code. We also present the performance of the supemodal parallel factorization code, and compare it with that of the nodal code. Finally, in section 6 we give our conclusions.

2 Background

2.1 Sparse Factorization

The problem which we wish to solve in this paper is the direct solution of a system of equations $Ax = b$, where A is a sparse, symmetric, positive definite matrix. This system is solved by performing a Cholesky factorization of the matrix A, yielding a factor matrix L, where $A = LL^T$, and then performing a pair of triangular system solves to arrive at the value of x. The solution of this system is typically done in four steps. The first step, *ordering*, heuristically reorders the rows and columns of A to decrease the amount of fill which will appear in the factor. The next step, *symbolic factorization*, determines the structure of the factor matrix L and allocates the appropriate storage for that structure. The third step, *numerical factorization*, determines the actual numerical values which reside in the structure determined in the previous step. The fourth step, *triangular solution*, does a pair of triangular system solves, using the computed factor L and the vector b, to arrive at the value of x. In this paper, we focus on the most time-consuming of the four steps in Cholesky factorization, the numerical factorization step (see [9] for more details).

In column-oriented Cholesky factorization, the primary computation consists of adding a multiple of one column of factor L into another, in order to zero a non-zero in the upper triangle of L. This computation is called a column cancellation. Column-oriented methods in general use one of two different approaches, the *left-looking* and *right-looking* approaches[1]. All factorization schemes which we discuss in this paper are based on one of these two schemes. As will be seen in later sections, each of these two approaches will have a context in which they result in a more efficient factorization code.

The left-looking approach performs column cancellations at the time the destination column is processed. The following pseudo-code gives the structure of a left-looking factorization computation:

```
1.  Set L = A
2.  for j = 1 to n do
3.      for each k s.t. l_jk ≠ 0 do
4.          l_*j ← l_*j − l_jk * l_*k
5.      l_jj ← √l_jj
6.      for each i s.t. l_ij ≠ 0 do
7.          l_ij ← l_ij / l_jj
```

In step 4 of this pseudo-code, column is canceled by column k. Note that this single step represents an operation on the entire column. In steps 5 through 7, column j is scaled by the square root of the diagonal

[1] In [9] these are called the *inner-product* and *outer-product* approaches, respectively.

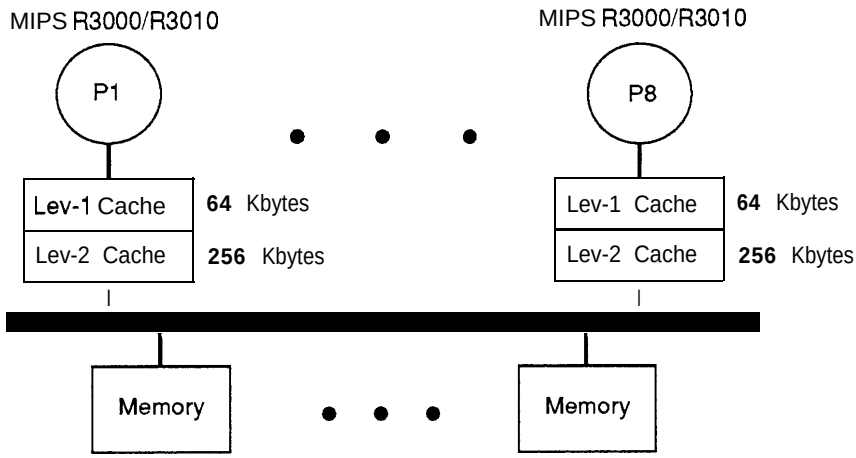


Figure 1: Overview of the SGI 4D/380 multiprocessor architecture.

element. The primary left-looking factorization method is the general sparse method, which is employed in SPARSPAK [9].

The right-looking approach performs column cancellations at the time that the source column is processed.

```
1.  Set L = A
2.  for k = 1 to n do
3.      l_kk ← √l_kk
4.      for each i s.t. l_ik ≠ 0 do
5.          l_ik ← l_ik / l_kk
6.      for each j s.t. l_jk ≠ 0 do
7.          l_*j ← l_*j − l_jk * l_*k
```

Here the cancellations are done in step 7, and the column scaling is done in steps 3 through 5. The primary right-looking method is the multifrontal method [6].

2.2 Factorization Benchmarking

In order to provide concrete performance comparisons for the various factorization schemes which we discuss in this paper, we will present performance numbers for the factorization of a number of benchmark matrices on a commercially-available multiprocessor. In this subsection, we describe both the multiprocessor on which we perform our experiments and the benchmark matrices which we factor on this machine.

The multiprocessor we use is the Silicon Graphics 4D/380 [3]. It supports 8 high-performance RISC processors, each processor consisting of a MIPS R3000 integer unit and an R3010 floating point coprocessor. The processors run at 33 MHz, and are rated at 29 MIPS and 4.9 double precision LINPACK MFLOPS. They are interconnected with a bus, having a peak throughput of approximately 67 Mbytes per second. The processor caches are kept coherent using a snoopy cache-coherence protocol [1]. The high-level organization of the machine is shown in Figure 1.

Each processor in the 4D/380 has a 64 Kbyte instruction cache, a 64 Kbyte first-level data cache, and a 256 Kbyte second-level data cache. References which hit in the first-level data cache are serviced in a single cycle. References which miss in the first-level cache but hit in the second-level cache require 8 cycles to service, and result in a single 16-byte cache line being loaded into the first-level cache. References which miss in both data caches require roughly 53 cycles to service. They result in four 16-byte lines being loaded into the second-level cache and one 16-byte line being loaded into the first. The machine presents a shared-memory parallel programming model.

The R3010 floating point coprocessor can perform a double precision add in 2 cycles and a double precision multiply in 5 cycles. These figures do not include the time necessary to fetch the operands. In general, fetching

Table 1: Benchmarks

	Name	Description	Equations	Non-zeroes
1.	DENSE750	Dense symmetric matrix	750	561,750
2.	LSHP3466	Finite element discretization of L-shaped region	3.466	20,430
3.	BCSSTK15	Module of an Offshore Platform	3,948	113,868
4.	BCSSTK16	Corps of Engineers Dam	4,884	285,494
5.	BCSSTK29	Boeing 767 Rear Bulkhead	13,992	3 16,740

Table 2: Factorization information and runtimes on one processor of an SGI 4D/380.

	Nonzeroes	Floating point	SPARSPAK		Right-looking	
Name	in L	ops (millions)	runtime (s)	MFLOPS	runtime (s)	MFLOPS
DENSE750	280,875	141.19	45.22	3.12	49.93	2.83
LSHP3466	83,116	4.14	1.24	3.34	1.51	2.74
BCSSTK15	647,274	165.72	52.73	3.14	74.72	2.22
BCSSTK16	736,294	149.89	46.18	3.25	61.70	2.43
BCSSTK29	1,680,804	394.87	128.54	3.07	181.34	2.18

the operands and storing the result may require more time than performing the floating point operation, due to the high cost of cache misses.

The matrices which we use to evaluate sparse factorization performance are drawn from the Boeing-Harwell sparse matrix test set [5], with the exception of matrix DENSE 750. In Table 1 we give brief descriptions of each of the benchmark matrices which we study. We have tried to choose a variety of sizes and sparsities. The matrix LSHP3466, for example, is small and very sparse. The matrix DENSE750, on the other hand, is completely dense. The others are of medium size and sparsity.

3 Performance of Nodal Factorization Codes

In this section we look at the performance of the parallel factorization method developed at the Oak Ridge National Laboratory [8]. This method is a parallel implementation of the right-looking scheme which was described in the previous section. While the ORNL scheme was originally designed for a message-passing parallel computer, it is appropriate for shared-memory parallel machines as well.

The ORNL parallel factorization scheme can briefly be described as follows. Each column of the factor matrix is owned by a particular processor, and all cancellations to a column are done by its owner processor. The distribution of columns to processors is done using the scheme developed by Geist and Ng [7]. When a column is complete, that is once the column has received all cancellations from other columns and has been scaled, then a message is sent from the owner processor to all processors which own columns canceled by that column. The message contains the values of the completed column. The processors receiving the messages perform the appropriate column cancellations. This process continues until all columns are complete. Some bookkeeping is necessary in order to keep track of when a column has received all cancellations which will be done to it.

We now look at the performance obtained with the ORNL parallel factorization method on the SGI 4D/380. The typical method for measuring the performance of a parallel code is to compute its parallel speedups relative to an efficient sequential code. We therefore first determine the performance of a sequential factorization code. While the parallel code is based on the right-looking approach, we consider sequential codes of both varieties, It is not immediately clear which approach leads to a more efficient sequential code.

The left-looking code which we consider is a translation of the numerical factorization routines of SPARSPAK into the C language. The right-looking code is simply a sequential version of the ORNL parallel scheme. The performance for these two sequential factorization schemes is shown in Table 2. For these and all other performance numbers in this paper, all floating point arithmetic is performed in double-precision, and all matrices

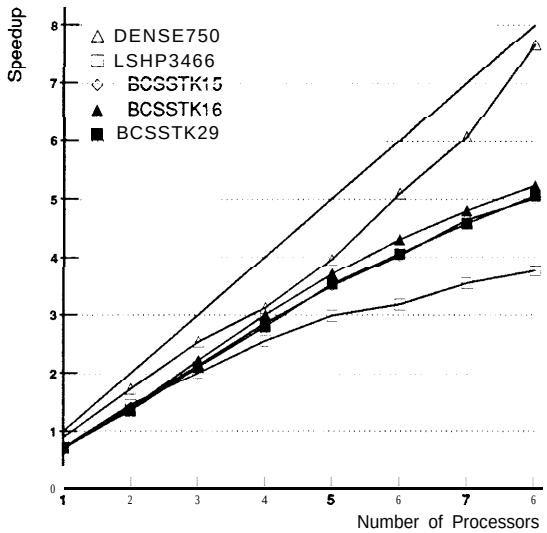


Figure 2: Speedups for ORNL parallel scheme, relative to SPARSPAK.

are ordered with the minimum degree till-reducing heuristic [9] before being factored.

It is clear from the performance results that the right-looking scheme is slightly less efficient than SPARSPAK. This difference is due mainly to the time spent matching non-zeroes during column cancellation. The two columns involved in the cancellation will in general have different non-zero structures, and the corresponding entries from the two columns must be matched. SPARSPAK does this matching by scattering the non-zeroes of the canceling columns into a dense contribution vector, and then gathering them into the destination column once all contributions have been scattered. The right-looking scheme, on the other hand, does a search through the destination column for each non-zero in the canceling column to find the matching entry. The SPARSPAK scheme is more efficient.

In Figure 2 we present the speedups obtained with the ORNL parallel factorization scheme, relative to the sequential SPARSPAK code. This figure shows that the ORNL scheme achieves substantial speedups for all but the smallest of the benchmark matrices. The speedups are not perfect primarily because the right-looking sequential code on which the ORNL parallel code is based is less efficient than SPARSPAK on a single processor.

At this point, it may appear that little gain is possible in parallel factorization performance. The ORNL parallel code is more than 5 times as fast as SPARSPAK when run on 8 processors. Thus, if we assume that SPARSPAK is an efficient sequential code, then the ORNL parallel code appears to provide excellent parallel performance. However, as we show in the next section the efficiency of the sequential code can be improved by making use of the supemodal structure of the matrix. Although the new algorithm complicates the parallelization of the factorization code, we show in section 5 that the added difficulty of parallelization can be overcome.

4 Supernodal Factorization with Cache-Miss Reducing Techniques

A -faster sequential factorization code can be created by taking advantage of the supemodal structure of the matrix [2]. Consider the matrix A of Figure 3, and its factor L. Although the columns of A appear quite dissimilar in structure, many of the columns of the factor L appear nearly identical. This coalescing of column structures occurs when factoring any matrix. Significant benefits can be obtained by taking advantage of this coalescing of structures and treating sets of columns with nearly identical structures as groups, or *supernodes*. A supemode is defined as a set of contiguous columns whose structure in the factor consists of a dense triangular block on the diagonal and an identical set of non-zeroes for each column below this diagonal block. In the

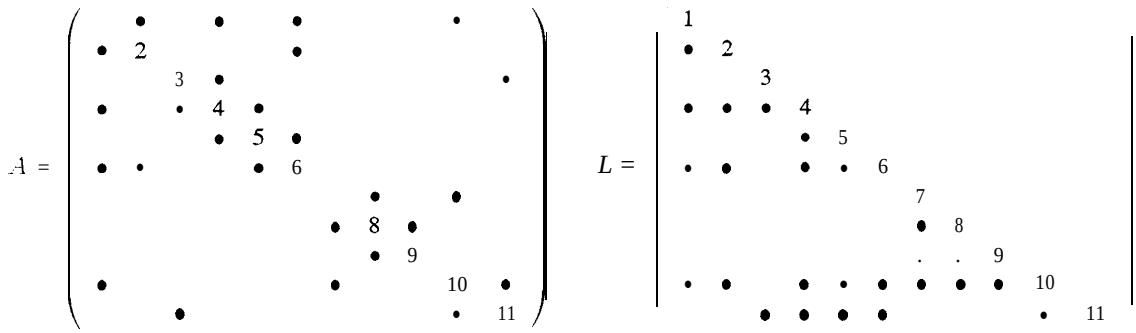


Figure 3: Non-zero structure of a matrix A and its factor L.

example matrix, the supernodes are $\{1, 2\}$, $\{3\}$, (4, 5, 6}, (7, 8, 9}, and $\{10, 11)$.

By making use of the properties of supernodes, a number of performance enhancements are possible. The first has to do with reducing the number of references made to index vectors. Since the columns of a supemode all have the same structure, they all cancel the same set of destination columns. In supemodal factorization, a destination column is canceled by all the columns in a supemode as a single step; the step is called supemodal elimination. This mass cancellation can be done by first adding together the appropriate multiples of each of the columns in the supemode, and then adding the result into the destination column. If we assume that the non-zeroes of a column are stored contiguously in increasing order by row, then the addition of columns within the supemode can be done without regard for the rows in which each element resides. If we account for the differing lengths of the diagonal block, then the remaining non-zeroes can simply be added together as dense vectors. Only after all the columns of the supemode have been added together are the index vectors accessed to determine where the entries of the net update are to be placed in the destination column.

The second source of performance improvement comes from the loop unrolling [4] which is possible when adding the columns of a supemode together. Consider the memory references which occur when one column is added into another. An element from each column is first loaded, then the arithmetic takes place, and then the result is stored We therefore perform two loads and one store for each element. If, on the other hand, we add four columns into another column, then we would load one element from each of the four columns and one from the destination, perform a number of arithmetic operations, and store one result. In this case we have performed five loads and one store to do four column additions. This is substantially fewer references per column addition than are done when doing a single addition. The result is again substantially fewer memory operations.

The third source of higher performance is a modification which allows the factorization to make better use of the processor cache. In a nodal right-looking code, all of the cancellations done by a column k are performed before those done by column $k + 1$. In the course of performing these cancellations, each of the destination columns is read into the cache. This set of columns will typically require more space than is available in the cache. Thus, by reading in a column, a previously read column will necessarily be displaced. Once the cancellations by column k are complete, those of column $k + 1$ must be done. Column $k + 1$ will typically cancel virtually the same set of columns as column k. Unfortunately, most of these will have been displaced from the cache. The result is a very high cache miss rate. A similar effect occurs with SPARSPAK.

If we perform supemodal cancellation, and require that the supemode fits in the cache, then we can decrease this miss rate. Assume that a supemode of 4 columns fits in the cache. We combine the cancellations of the 4 columns to another column into a single contribution vector, as was discussed before. While it is still the case that the destination will most likely not be present in the cache, we will be able to perform four cancellations to this column once it is fetched from main memory. If more columns of the supemode fit in the cache, then more cancellations could be done per destination column fetch. Without this modification, we would expect to do a single column cancellation per column fetch. This partitioned-supemodal approach results in a substantial decrease in the number of cache misses.

A more detailed discussion of the advantages of supemodal factorization on a high-performance workstation can be found in [12]. We simply present the results of the supemodal modifications here. In Table 3, we

Table 3: Runtimes on one processor of an SGI 4D/380.

Problem	SPARSPAK Time (s)	SPARSPAK MFLOPS	Right-looking partitioned-supemodal Time (s)	Right-looking partitioned-supemodal MFLOPS	Runtime ratio
DENSE750	45.22	3.12	18.13	7.79	2.5
LSHP3466	1.24	3.34	0.89	4.66	1.4
BCSSTK15	52.73	3.14	23.88	6.94	2.2
BCSSTK16	46.18	3.25	22.36	6.70	2.1
BCSSTK29	128.54	3.07	54.92	7.19	2.3

give performance figures for the *right-looking partitioned-super-nodal* method of factorization, a method which incorporates all of the techniques which have been discussed in this section. These runtimes are compared with those of SPARSPAK; the *runtime ratio* column gives the ratio of the runtime of SPARSPAK to that of the right-looking partitioned-supemodal scheme. As can be seen from this table, the supemodal code performs sparse factorization at more than twice the speed of SPARSPAIL This supemodal sequential code forms the basis for our parallel factorization code in the next section.

5 Parallel Supernodal Factorization

Our parallel supemodal code is best explained in terms of the ORNL parallel code. Both codes are based on the right-looking approach to factorization, with the major difference being in the grain size of the parallel tasks. In the ORNL code, columns are owned by specific processors. In the supemodal scheme, *supernodes are* owned by processors. In the ORNL scheme, when a message is sent from one processor to another, it indicates that a column is complete and the cancellations by that column to columns owned by the receiving processor should be performed. In the supemodal scheme, a message indicates that an entire supemode is complete and the corresponding updates should be performed.

One important consequence of the supemodal modification is that it substantially increases the task grain size compared to the ORNL scheme. A single message now requires cancellations by an entire supernode. Similarly, more work is required in order to complete a supemode which has received all cancellations from other supernodes. In the ORNL scheme, a column which had received all cancellations simply has to be scaled by the square root of the diagonal In the supernodal scheme, cancellations must be done between columns within the supernode.

An increase in task grain size is naturally accompanied by a decrease in the total number of tasks. This decrease is not necessarily a bad thing, provided that there are still sufficiently many tasks to keep the available processors busy. In the case of parallel supemodal factorization, however, the increase in grain size causes severe load balancing problems. For example, in the extreme case of matrix DENSE750, the factor L forms a single supernode. It therefore presents no possibility for concurrent work if all of the work associated with a supemode is assigned to a single processor. While a more sparse matrix would clearly contain more supemodes, the reduction of concurrency due to large supernodes is a problem for the other benchmark matrices as well. As a result, it is necessary to decrease the task grain size. We choose to decrease the grain size by splitting supernodes into a smaller pieces.

Now that we have decided to split supernodes, a question which immediately arises is how much splitting should be done, That is, into how many pieces should a particular supemode be split? One major consideration is that we would like to keep supemodes as large as possible. The sequential supemodal factorization code was more efficient than the nodal code due entirely to the efficiencies which supemodal elimination make possible. By splitting supernodes, we are decreasing the benefits of supernodal elimination, and thus making the supemodal code more like the nodal code.

On the other hand, we wish to perform sufficient supemode splitting so that all available processors are kept busy. For example, if three processors will sit idle while another performs the work of a single supernode, then

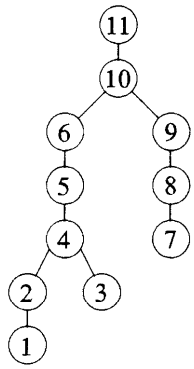


Figure 4: Elimination tree of A.

we would be better off splitting that supemode into sufficiently many pieces so that all four processors could work on it. We therefore wish to perform sufficient splitting to keep the available processors busy, while not performing splittings which do not contribute to higher processor utilizations.

A third consideration has to do with the scheduling constraints imposed by the dependencies between the factorization tasks. The degree to which a particular supemode should be split depends on the number of tasks available at the time the supemode is processed. This information is determined by the dependencies between the tasks, and by the schedule chosen for processing the tasks once they become available. The dependency constraints, combined with the size constraints mentioned in previous paragraphs, make an optimal solution to the problem infeasible.

We therefore resort to a heuristic solution to determine the supemode splitting. Our heuristic is based on the observation that the amount of concurrency available in a right-looking parallel factorization decreases as the computation proceeds. This decrease can be better understood by considering the *elimination tree* of the matrix [10]. The elimination tree captures the dependencies between the columns of the matrix, expressing these dependencies in the form of a tree structure. A column is dependent only on those columns in the subtree below it in the elimination tree, and equivalently a column only modifies the columns which are in the path from itself to the root of the tree. Also, columns which do not have an ancestor/descendent relationship are not dependent on each other. The elimination tree of the matrix in Figure 3 is shown in Figure 4.

The decrease in available concurrency is due to two factors. First, the elimination tree, which is processed bottom-up, clearly becomes narrower as we move up the tree. A narrower tree corresponds to a smaller set of independent tasks which may be processed in parallel. The second factor has to do with the overlapping of the cancellation work of a column. Consider a column which is closer to the leaves of the elimination tree. Once it is completed, it can potentially cancel all of the columns in the path from itself to the root of the elimination tree. A column which is closer to the root of the elimination tree, on the other hand, cancels a smaller set of columns. Since these cancellations can be done in parallel, the cancellations resulting from the completion of a column close to the root provide less opportunity for parallelism than those of a column closer to the leaves. Thus, the available concurrency decreases as we move up the elimination tree, due to (i) a decrease in the number of independent columns which can be processed in parallel, and (ii) a decrease in the number of cancellations done per column.

The collection of a number of columns into an indivisible set, a supernode, naturally has the effect of decreasing the amount of available concurrency. If more than enough concurrency is available (e.g. close to the leaves of the elimination tree), then we can afford to reduce the available concurrency by grouping columns into supemodes. However, as the amount of overall concurrency decreases, the size of the supernodes should decrease as well, so that enough concurrency is available to keep all of the available processors busy. We tried a number of heuristics which took these considerations into account. The most effective of these splits supernodes based solely on their height in the elimination tree. This heuristic begins at the supemode at the root of the elimination tree, and traverses down. Starting from the top, it breaks this supemode into $3 * P$ pieces of 4 columns each, where P is the number of processors. The multiplier 3 was determined empirically. It continues the splitting with $3 * P$ pieces of 8 columns each, followed by $3 * P$ pieces of 12 columns each, and so on.

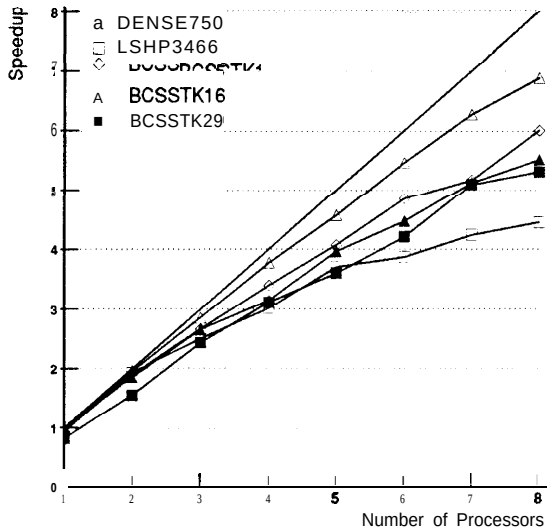


Figure 5: Speedups (relative to a sequential supernodal code).

When a branch in the elimination tree is reached, then the splitting continues with each of the supernodes below the branch as if the branch had not occurred Thus, for example, if the root supemode is exhausted after the third piece of size 8, then the children of this supemode will each begin with the fourth piece of size 8. Note that such a strategy gradually increases the size of the supernodes as the depth in the elimination tree increases.

This heuristic is by no means the only one which we tried. One of our other heuristics split all supernodes into tied size pieces. The piece size was chosen to be large enough so that most of the benefits from supemodal elimination were retained, but at the same time the pieces were small enough so that they did not create substantial load balancing problems. None of the piece sizes tried resulted in parallel runtimes faster than those obtained with the heuristic described in the previous paragraph. Another heuristic that we tried split supernodes based on the amount of work contained in the subtree of the elimination tree below that supernode. The intuition behind this approach was that if a subtree contained 40% of the factorization work, then 40% of the processors would be working on that subtree. The supemode at the root should therefore contain enough concurrency to keep these processors busy. Again, this heuristic did not result in improved performance. More details concerning variations on the splitting strategy can be found in [11].

In Figure 5 we present the speedups obtained with our parallel supemodal code, using the above splitting strategy. These speedups are all relative to the sequential right-looking partitioned-supemodal code described in the previous section. Note that the speedups are quite similar to those obtained by the ORNL parallel code. Since the sequential supernodal code is roughly twice as fast as SPARSPAK, the supernodal parallel code is therefore roughly twice as fast as the ORNL parallel code.

In order to better compare the relative performance of the two parallel codes, we show in Figure 6 the ratio of the runtime of the ORNL parallel code to that of the supernodal parallel code for the same matrix. Note that the ratio is between 2 and 3 for each choice of number of processors. In terms of absolute performance, the supernodal parallel code performs the factorization at a rate of approximately 40 MFLOPS for the majority of the benchmark matrices when using 8 processors.

It is interesting to consider how these parallel factorization schemes would compare on a machine with more processor;. In [11], we looked at the performance of our supernodal scheme, as compared with that of the ORNL scheme, on a simulated multiprocessor with up to 32 processors. We found that the performance advantage enjoyed by the supemodal code decreased noticeably as the number of processors was increased. When using more processors, the supernodes had to be split into smaller pieces to keep the processors busy, thus decreasing their benefit. This decrease is much less noticeable on the SGI 4D/380. We believe that this is

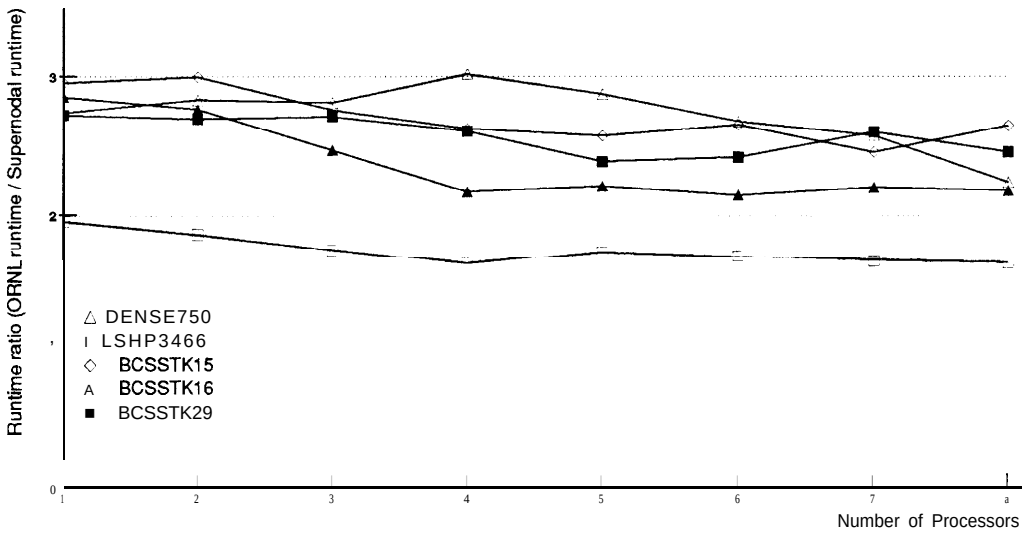


Figure 6: Relative runtimes (ORNL runtime divided by supernodal runtime).

due to the fact that the nodal code exhibits a much higher cache miss rate than the supemodal code. This higher miss rate translates into more traffic on the shared bus, and thus more contention between the processors. Since our simulation in [11] did not model contention for shared memory, its effect was not visible in that study. On a multiprocessor with a larger number of processors than the 4D/380 and a similar memory system, it is not clear which of these two effects, the decrease in relative performance due to supemode splitting or the increase due to less demand for the shared memory, would prevail.

6 Conclusions

In this paper, we have considered the problem of performing sparse Cholesky factorization on a high-performance multiprocessor workstation. We began by looking at the parallel factorization scheme developed at the Oak Ridge National Laboratory. The performance of the ORNL parallel code was compared with that of SPARSPAK and that of a sequential implementation of the ORNL scheme. We found that the ORNL parallel code gave good parallel speedups, relative to the sequential codes. However, these sequential codes were later shown to be less efficient by a factor of two than another sequential code. Thus, good speedups relative to these sequential codes proved not to imply good performance overall.

We then discussed supemodal factorization, and described a parallel supemodal factorization code. A heuristic was described for dealing with the load balancing problems which supemodal elimination created. The parallel supemodal code with heuristic load balancing was between two and three times as fast as the ORNL code, when using up to eight processors. This increase in performance was mainly due to a reduction in memory traffic requirements.

We also found that both the nodal and supernodal codes achieved substantial speedups using as many as 8 processors attached to a single bus. One might expect eight high-performance processors to saturate a moderate bandwidth shared bus when performing computations on large matrix problems. This was not the case, due to the fact that sparse Cholesky factorization algorithms make effective use of a cache, thus keeping traffic off of the bus.

The parallel supernodal code performed the sparse factorization at a rate of roughly 40 MFLOPS on eight processors. In order to put this figure in perspective, we compare the computation rates obtained on our parallel scalar machine with those that can be obtained on a vector supercomputer. In Table 4 we compare the

Table 4: Comparison of SGI 4D/380 and CRAY Y-MP.

	SGI 4D/380, 8 processors		CRAY Y-MP[2]		
Name	Time (s)	MFLOPS	Time (s)	MFLOPS	Ratio
BCSSTK23	3.32	35.9	0.62	191.6	5.4
BCSSTK15	3.98	41.5	0.84	197.7	4.7
BCSSTK16	4.06	36.8	0.79	190.8	5.1
BCSSTK29	10.34	38.0	2.16	182.3	4.8

performance of an 8 processor SGI 4D/380 with that of a single processor of the CRAY Y-MP (as given in [13]). The CRAY numbers are from a hand-coded, CRAY assembly language implementation of supemodal sparse factorization. As can be seen from this table, the multiprocessor achieves approximately one fifth of the floating point performance of the CRAY on this problem. Considering the relative costs of these two machines, multiprocessor workstations appear to be extremely cost-effective machines for factoring large, sparse, positive definite matrices.

Acknowledgments

We would like to thank Jeff Doughty, James Winget, and Forest Baskett at Silicon Graphics for helping us to get access to an SGI 4D/380. We would also like to thank John Gilbert and Horst Simon for their helpful comments on sparse factorization terminology. This research is supported by DARPA contract N00014-87-0828. Edward Rothberg is also supported by a grant from Digital Equipment Corporation.

References

[1] Archibald, J., and Baer, J.-L., "An economical solution to the cache coherence problem", *Proc. of the 15th Annual Int. Sym. on Computer Architecture, 355-362, 1985.*

[2] 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.

[3] Baskett, F., Jemoluk, T., and Solomon, D., "The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 100,000 lighted polygons per second", *COMPCON 88,* 468-471, 1988.

[4] Dongarra, J.J., and Eisenstat, SC., "Squeezing the most out of an algorithm in CRAY FORTRAN", *ACM Transactions on Mathematical Software,* 10:219-230, 1984.

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

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

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

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

[9] George, A., and Liu, J., *Computer Solution of Large Sparse Positive Definite Systems,* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.

The Y-MP times reported in [13] are from a machine with a slower clock speed than that of the production Y-MP. We have adjusted the times to account for the fact that a production machine would be slightly faster.

[10] Liu, J., "The role of elimination trees in sparse factorization", *SIAM Journal on Matrix Analysis and Applications*, 11(1): 134-172, 1990.

[11] Rothberg, E., and Gupta, A., *A comparative evaluation of nodal and supernodal parallel sparse matrix factorization: Detailed simulation results*, Technical Report STAN-CS-90-1305, Stanford University, 1990.

[12] Rothberg, E., and Gupta, A., *Fast sparse matrix factorization on modern workstations* Technical Report STAN-CS-89-1286, Stanford University, 1989.

[13] Simon, H.D., Vu, P., and *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.