A STUDY OF THE EFFECTS OF ORDERING, PARTITIONING AND FACTORIZATION ALGORITHMS ON DISTRIBUTED SPARSE CHOLESKY FACTORIZATION

MUKUND RAGHAVACHARI AND ANNE ROGERS
DEPARTMENT OF COMPUTER SCIENCE, PRINCETON UNIVERSITY
PRINCETON, NJ 08544
MRM@CS.PRINCETON.EDU AMR@CS.PRINCETON.EDU
PHONE: (609)-258-5388
FAX:(609)-258-1771
CS-TR-505-96

Abstract.

In this paper, we perform a comprehensive evaluation of ordering, partitioning, and factorization algorithms under a unified framework. Previous research in distributed, sparse Cholesky factorization has considered each of the stages in the factorization process — ordering, partitioning and numerical factorization — in isolation. However, due to the strong dependencies between the stages, it is difficult to derive conclusions from such an approach.

Specifically, in our experiments, we use input files representative of practical problems to study the benefits and shortcomings of ordering algorithms — multiple minimum degree and spectral nested dissection, column-based partitioning algorithms — wrapped mapping and proportional mapping, and block-based partitioning. We use, as the base for our experiments, three factorization algorithms: Fanin, Fanout and Block Fanout. In addition, we extend the research of Rothberg[19] by performing an in-depth analysis of the advantages of panel-based partitionings versus column-based partitionings on message-passing machines. All experiments were run on a Thinking Machines CM-5.

1. Introduction. The need to solve sparse linear systems of equations arises in several fields, such as, structural analysis, linear programming and circuit simulation. One popular method of solving such systems is sparse Cholesky factorization. On parallel machines, the factorization algorithm proceeds in several phases. First, the columns of the matrix are *ordered* to reduce the number of operations performed or increase the parallelism of the factorization phase. Next, the work associated with the factorization of the matrix is *partitioned* among the processors. Finally, the matrix is factored using a *factorization* algorithm. Researchers have expended considerable effort in developing algorithms for ordering [6, 12, 16], partitioning [9, 17], and numerical factorization [1, 7, 15, 19]. In addition, several theoretical bounds have been derived for aspects of the problem such as communication volume [9], fill [8], and scalability [21]. On current parallel machines, however, the results have been disappointing — it has been difficult to attain reasonable speedups for this problem.

Previous research in sparse Cholesky factorization has considered each of the stages in the factorization process – ordering, partitioning and numerical factorization – in isolation. The strong dependencies between the stages, however, make it difficult to derive conclusions from such an approach. For example, Ashcraft [2] compares three factorization algorithms by reporting the execution times of the algorithms on a single input file. Under this framework, it is impossible to separate the effects of the ordering and partitioning algorithms used from those of the factorization algorithms. Similarly, the performance of ordering algorithms depends on the factorization algorithm used, and therefore, they cannot be evaluated independently.

In this paper, we perform a comprehensive evaluation of ordering, partitioning, and factorization algorithms under an unified framework. Specifically, we use input files representative of practical problems to compare the benefits and short-

$$\begin{pmatrix}
a_{00} & & & & & & & & \\
a_{10} & a_{11} & & & & & & & \\
 & a_{21} & a_{22} & & & & & & \\
 & a_{31} & & a_{33} & & & & & \\
 & & & & a_{44} & & & & \\
 & & & & a_{54} & a_{55} & & & \\
 & & & & & & a_{66} & & \\
 & a_{71} & a_{72} & & a_{74} & & a_{76} & a_{77} & \\
a_{80} & & a_{82} & & & & & a_{87} & a_{88}
\end{pmatrix}$$

(a) Lower triangular part of sparse matrix A

$$\begin{pmatrix}
l_{00} & & & & & & & & \\
l_{10} & l_{11} & & & & & & & \\
 & l_{21} & l_{22} & & & & & & \\
 & l_{31} & l_{32} & l_{33} & & & & & \\
 & & & & l_{44} & & & & \\
 & & & & l_{54} & l_{55} & & & \\
 & & & & & & l_{66} & & \\
 & l_{71} & l_{72} & l_{73} & l_{74} & l_{75} & l_{76} & l_{77} & \\
l_{80} & l_{81} & l_{82} & l_{83} & & & & l_{87} & l_{88}
\end{pmatrix}$$

(b) Factor matrix L of A

FIG. 1. 9×9 *sparse, symmetric matrix A and its factor L.*

comings of ordering algorithms — multiple minimum degree [12] and spectral nested dissection[16], column-based partitioning algorithms — wrapped mapping and proportional mapping [17], and block-based partitioning. We use, as the base for our experiments, three factorization algorithms: Fanin [1], Fanout [7] and Block Fanout [19, 20]. In addition, we extend the research of Rothberg [19] by performing an in-depth analysis of the advantages of panel-based partitionings versus column-based partitionings on message-passing machines. The results of our experiments can be summarized as follows:

- The ordering algorithm that generates the least fill generally performs best. Orderings generated by spectral nested dissection have greater parallelism, but not enough to offset the cost of executing a greater number of floating point operations.
- The effectiveness of partitioning algorithms depends on the structure of the input file and the number of processors used. Block Fanout performs well when there are few small blocks, that is, when the matrices are large or the number of processors is small. On small files, Fanin coupled with proportional mapping seems to be the algorithm of choice.
- We found no strong correlation between factorization time and communication volume.
- One cannot use the work distribution per processor as a metric for load balance. This metric, though commonly used, does not have a strong correlation with factorization time.
- It is important to consider the effects of sequential efficiency on parallel machines — panel-based methods do considerably better than column-based methods despite having less concurrency.

The paper is organized as follows: §2 presents background on sparse Cholesky factorization and the ordering, partitioning, and factorization algorithms used in the paper. §3 describes the platforms used for our experiments. In §4, we present an evaluation of the two ordering algorithms. §5 discusses the advantages of panel-based partitioning and compares the effects of wrapped mapping, proportional mapping, and blocked mapping on factorization time. In §6, we summarize our results and identify avenues of further research.

2. Background. The Cholesky factorization of a matrix is an important step in the direct method of finding the solution to the equation $Ax = b$, where A is a symmetric, positive-definite matrix. The Cholesky factorization of A produces the

lower triangular matrix L, where $LL^T = A$. The triangular equations $Ly = b$ and $L^T x = y$ are then solved to arrive at the solution, x. In this paper, we use a_{uv} to refer to elements of a matrix A and l_{uv} to refer to elements of its factor matrix L, where u refers to the row and v to the column. The following pseudocode describes the general factorization algorithm:

```
1.   for i = 1 to n do
2.           l_ii = sqrt(a_ii)
3.           for j = i + 1 to n do
4.                   l_ji = a_ji / l_ii
5.           for j = i + 1 to n do
6.                   for k = j to n do
7.                           a_kj = a_kj - a_ji * a_ki
```

Lines 2–4 constitute the *elimination* step of the algorithm and lines 5–7, the *modification* step. When A is sparse, line 4 needs to be executed only when a_{ji} is non-zero and line 7 affects only those elements a_{kj} where a_{ji} and a_{ki} are non-zero. The modification step induces additional non-zeros in the structure of A when a_{ji} and a_{ki} are non-zero and a_{kj} is zero. As a result, the factor matrix L is usually less sparse than A. The number of non-zeros added during factorization is called the *fill* of the factor matrix. If column i modifies column j, then, in L, the non-zero structure of i is a subset of j's. Figure 1 is an example of a sparse matrix A and its factor matrix L. We only show the lower triangular portions of the matrices since they are symmetric.

Sparse Cholesky factorization algorithms for distributed memory machines typically have four steps. First, the rows and columns are *ordered* to reduce the fill in the factor matrix. Since the number of floating point operations performed in the factorization phase is directly proportional to the fill in the factor matrix, a good ordering can lead to substantial savings in running time. The second step, *symbolic factorization*, determines the non-zero structure of the factor matrix and allocates storage for it. The matrix is then *partitioned* among the processors, after which, the processors perform the *numerical factorization*. The symbolic factorization phase is generally performed sequentially and is not as important as the other phases in the factorization process [10]. We describe the algorithms used for ordering, partitioning, and numerical factorization in §2.2, §2.3, and §2.4 respectively. For more details on sparse Cholesky factorization, please refer to [8, 10].

2.1. Definitions. In this section, we provide definitions for terms that are used in the rest of paper to discuss sparse Cholesky factorization. If each column in a sparse matrix is represented by a node in a graph, the *elimination tree* of the matrix is defined by the vertex set V of columns and the edge set E given by the following equation:

$$E = \{(i,j) \mid i,j \in V \wedge j = min(k \mid k \in V \wedge l_{ik} \neq 0, i < k)\}$$

In other words, there exists an edge between two vertices i and j, $i < j$, if and only if l_{ij} is the first non-zero in column i. Columns modify only their ancestors in the elimination tree [14]. Hence, the elimination tree defines a partial order on the elimination of columns in the numerical factorization phase. A consequence of this partial order is that disjoint subtrees of the elimination tree may be processed in parallel. Figure 2a shows the elimination tree defined by the matrix L in Figure 1b.

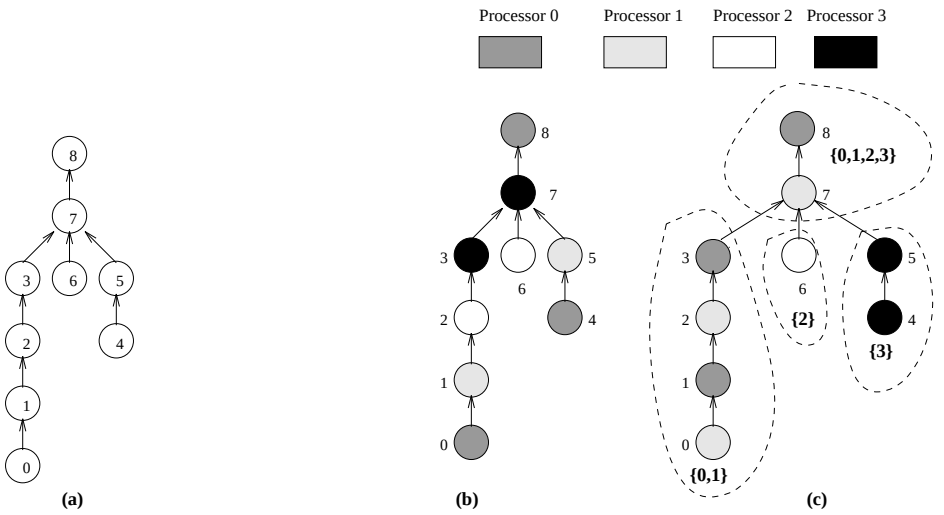

FIG. 2. *(a) Elimination tree for matrix in Figure 1 (b) Wrapped partitioning of columns (c) Proportional mapping of columns, dotted regions show active range of processors.*

An optimization that increases the performance of sparse Cholesky factorization on parallel machines is to use the elimination tree to partition the matrix into as many disjoint subtrees, called *domains*, as possible. Since domains are disjoint, they receive no updates from any other domain or column. If all columns in a domain are allocated to the same processor, domains can be factored independently without any communication. For example, in Figure 2a, columns 0-3, 6, and 4-5, form three domains.

There are usually several sets of columns in a factor matrix L where each column in a set shares the same non-zero structure. A set of adjacent columns S in L is called a *supernode* if its non-zero structure consists of a dense lower triangular part on the diagonal and an identical set of non-zeros in each column below the diagonal [3]. In an elimination tree, each supernode appears as a chain of nodes though not all chains are supernodes. The term *separator* is used interchangeably with supernode. For example, in Figure 1b, columns 1–3 form a supernode.

Since columns have different non-zero structures, modifying a column j by column i usually requires two steps: calculating updates from i, and *scattering* the modifications into j. Since every column in a supernode shares the same structure, it is often more efficient to perform a *supernode-column* update, that is, aggregate the modifications from a supernode and scatter the aggregation into the destination column. It is sometimes useful to break up supernodes into smaller disjoint subsets of adjacent columns, called *panels*, to achieve better cache locality or to increase concurrency [19]. For example, in Figure 1b, we can break up the supernode consisting of columns 1–3 into two panels, one containing columns 1 and 2, the other containing column 3.

2.2. Ordering. The goal of the ordering phase is to find a permutation of the rows and columns that minimizes the fill generated (and thus the floating point operations performed), while increasing the parallelism of the factorization phase. While the problem of finding a permutation of the rows and columns of a matrix that minimizes fill is NP-complete [25], there are several heuristics for limiting fill in sparse Cholesky factorization. Most of these heuristics operate on the graph $G(A)$ associated

FIG. 3. *(a) Original graph for matrix in Figure 1 (b) Graph after removing column 1.*

with a sparse, symmetric $n \times n$ matrix A, where $G(A)$ is a n-vertex undirected graph with an edge between two vertices i and j if the entry a_{ij} is non-zero in A. The effect of eliminating a column i in A can be described in $G(A)$ by removing vertex i from $G(A)$ and adding fill edges so that the neighbors of the eliminated vertex become a clique. This process is repeated until all vertices are eliminated. The total number of fill edges added is the fill of the matrix. Figure 3 gives the graph for the example matrix in Figure 1 and shows the result of eliminating column 1. In our experiments, we use two popular heuristics: the multiple minimum degree algorithm (MMD) [12] and the spectral nested dissection algorithm (SND) [16]. We present a brief overview of these algorithms:

1. **MMD:** At each step in the multiple minimum degree algorithm, the heuristic chooses to limit fill by eliminating the node with the least degree. This greedy algorithm is very efficient and produces good orderings for most input matrices.[1]
2. **SND:** This heuristic is based on the divide-and-conquer paradigm. At each stage, a set of vertices S are chosen so that their removal, along with the edges incident on them, divides the graph into two or more disjoint pieces. Each of the pieces is ordered contiguously and the vertices in S are numbered last. Notice that the elimination of a vertex in one piece cannot induce fill in other pieces. This process is applied recursively to each of the pieces. The vertex set S, which is a separator, is chosen by using components of an eigenvector of the Laplacian matrix associated with the graph. The quality of the ordering produced by the algorithm depends on finding sufficiently small separators at each stage.

2.3. Partitioning. The goal of the partitioning phase is to divide the matrix among the processors so that the amount of work performed by each processor is roughly the same, and the time spent waiting for dependencies to complete is minimized. The elimination tree encapsulates information about the dependencies between tasks, and therefore, it is used by many partitioning algorithms as a guide for mapping work to processors. An important parameter in any partitioning algorithm is the granularity at which the matrix is divided. Methods can be distinguished by whether

[1] We developed a heuristic that, at each step, eliminated the node that induced the least number of fill edges to be added amongst its neighbors. Though this heuristic usually produced orderings that generate less fill than MMD, the time taken to calculate the orderings was prohibitively large.

they treat blocks, columns, or panels as atomic units. Wrapped mapping and proportional mapping can be applied at both column (panel size = 1) and panel-based granularities. We briefly describe the column-based versions of these two algorithms. The algorithm used for block-based mapping is described in §2.4. In this paper, we shall use WM to refer to wrapped mapping, PM to refer to proportional mapping, and BM to refer to block mapping.

1. **Wrapped mapping (WM):** As its name suggests, this algorithm allocates tasks to processors in a round-robin fashion – on a p processor machine, the i^{th} column is allocated to processor $i \ mod \ p$. This method generally provides good load and communication balance among the processors.
2. **Proportional mapping (PM):** Proportional mapping is a generalization of the subtree-subcube method [9]. Given the elimination tree of a matrix, the algorithm assigns the columns in the root separator to the available processors in a wrapped manner. The processors are divided among the separator's children based on the relative amount of work in each child subtree. The algorithm is then applied recursively to each subtree. Since the proportion of work in a child subtree is not necessarily integral, heuristics are used to decide the exact number of processors to allocate to a subtree [17].

Figures 2b-c show the results of applying the two algorithms to the elimination tree in Figure 2a.

2.4. Factorization. In sparse Cholesky factorization, the numerical factorization phase is typically the most expensive step. In our experiments, we use three factorization algorithms: *Fanout* [7], *Fanin* [1], and *Block Fanout* [19]. The Fanout and Fanin algorithms can be used at either a column-based or panel-based granularity. In this section, we describe the column-based versions of these algorithms (the panel-based versions are conceptually similar).

1. **Fanout:** Each processor is responsible for a set of columns and every processor keeps track of the number of modifications that need to be performed on each column that it owns. Once all modifications have been performed on a column j, it is eliminated. The eliminated column is then sent to all processors that own columns that j affects. On receiving a column j, a processor modifies all of its columns that are affected by j. This process continues until all columns are eliminated.
2. **Fanin:** As in the Fanout algorithm, each processor is responsible for eliminating the columns that it owns. The difference between the two algorithms lies in how modifications are performed. In the Fanin algorithm, each processor collaborates on processing every column of the matrix. The processors proceed through the columns of the matrix in ascending order. For each column j, every processor that owns columns that modify j aggregates the updates from its columns to j into a buffer. The processors send these aggregated updates to the processor that owns j, where the modifications are applied to j. Once all aggregated updates have been processed, j is eliminated.
3. **Blocked Fanout:** This algorithm, introduced by Rothberg [19], operates at the granularity of blocks to increase the concurrency in the factorization phase. In the first step of the algorithm, the matrix is partitioned into domains. In the second step, the supernodal structure of the matrix is used to partition the remainder of the matrix into blocks that are then assigned to the

TABLE 1
Matrices used in the experiments

Name	*Description*	*Cols*	*Non-zeros*
BCSSTK29	Buckling Model of 767 Rear Bulkhead	13,992	316,740
TK17	Elevated Pressure Vessel	10,974	219,812
BCSSTK15	Module of an Offshore Platform	3,948	60,882
GRID100	Finite Difference Discretization	10,000	39,601
BCSPWR10	Representation of U.S. Power Network	5,300	13,571

processors in a wrapped manner. Block-based methods have been shown to have better sequential kernels and concurrency and less communication than column-based methods, but at the cost of larger memory requirements and data structure overheads [19]. The algorithm factors the domains first and applies their updates to the blocks, after which, it factors the blocks. Since the domains can be factored independently without any communication between processors, the algorithm performs well when the domains are large. It is more efficient when there are not many small blocks.

The size of the blocks generated by Block Fanout depends on the size of the supernodes in the matrix. To create large supernodes, Block Fanout adds non-zeros to columns that have similar (but not the same) structure. This heuristic creates larger blocks during the partitioning phase, but results in a higher number of floating point operations in the factorization phase. This heuristic could be applied to panel-based Fanin and Fanout as well. In order to have a common base for our experiments, however, we disabled this optimization.

3. Experimental Environment. The test matrices used in our experiments are listed in Table 1. BCSSTK29, BCSSTK15, and BCSPWR10 are from the Harwell-Boeing Suite [5], while TK17 is from the SPLASH-2 benchmark suite [24]. GRID100 is the 9-point finite difference discretization of a 100x100 grid.

The ordering algorithms were compiled with `f77 -O2` and the ordering experiments were run on a MIPS R3000, with 120MB of memory, a 64K instruction cache, and a 64K data cache. We measured the runtimes using the `ftime()` routine from the standard C library, and in each case, we report the average of ten runs.

The factorization experiments were run on the Thinking Machines CM-5 at the NCSA Supercomputing Center at the University of Illinois. Each processing node of the CM-5 is equipped with a 32 Mhz SPARC processor rated at a peak performance of 32 Mips or 5 MFlops [4].[2] The code was compiled with `gcc -O2` and uses the CM-5 Active Messages layer [23] to handle communication. The times reported measure the time spent in the numerical factorization phase and were obtained using the CM-5 CMMD timers. We report communication volume as a per processor averages, that is, in MBytes/processor. Both the Fanout and Fanin codes are based on the Cholesky factorization code in SPLASH [22]. The Blocked Fanout code is based on the shared-memory version in SPLASH-2 [24].

[2] We did not use the vector units on the CM-5.

TABLE 2
MMD vs. SND: Ordering Time (in seconds)

Name	*SND*	*MMD*
BCSSTK29	265.36	5.61
TK17	200.39	2.71
BCSSTK15	20.42	1.99
GRID100	59.69	0.65
BCSPWR10	32.57	0.41

TABLE 3
MMD vs SND: Fill, MFLOP, and Tree height

Name	*Fill*		*MFLOP*		*Tree Height*	
	SND	*MMD*	*SND*	*MMD*	*SND*	*MMD*
BCSSTK29	1,801,779	1,378,056	609	393	1040	1757
TK17	1,194,256	830,056	340	165	1053	1841
BCSSTK15	500,161	590,340	122	165	586	965
GRID100	256,199	250,102	20	19	334	633
BCSPWR10	33,349	13,417	1	3	129	145

4. Ordering. We use five criteria to compare the effectiveness of the ordering schemes on the input matrices: ordering time, fill induced, number of floating point operations performed in the factorization phase, height of the elimination tree, and effects on execution time during the factorization phase. The results of our experiments are summarized in Tables 2–4.

Ordering Time: It is clear from Table 2 that SND is between one and two orders of magnitude slower than MMD. In many cases, the cost of ordering is higher than the cost of factorization. The SND ordering algorithm is more easily parallelizable than the MMD algorithm [16] and may benefit from a parallel implementation. Even assuming perfect speedup for SND on 32 processors, though, for all files except BCSSTK15, SND would still take more time than MMD. Note that the ordering time in MMD is dependent on the number of non-zeros in the matrix, and the ordering time in SND is dependent on the number of columns.

Fill and FLOP count: In our experiments, MMD performs better on 4 of the 5 input files (see Table 3). It must be noted that since both MMD and SND are heuristics, the amount of fill generated by the algorithms is highly dependent on the input file. The relationship between fill and the number of floating point operations is nonlinear. In the case of BCSSTK29, SND induces 28% more fill than MMD, but during the numerical factorization phase, the ordering generated by SND performs 54% more floating operations than the ordering generated by MMD.

Elimination Tree height: MMD usually produces tall, highly irregular trees, while SND produces more balanced trees. Figure 4 shows the elimination trees produced by the two ordering schemes for BCSSTK15. Recall that disjoint subtrees of an elimination tree may be processed in parallel; hence, well-balanced, "bushy" trees are considered more conducive to parallel factorization than long, narrow, trees. For all our input files, the trees produced by SND were shorter than those generated by

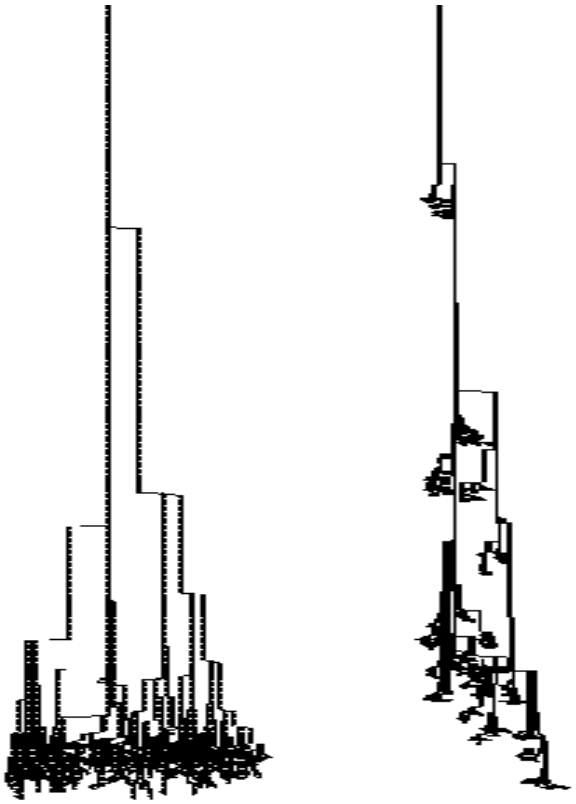

FIG. 4. *Elimination trees for BCSSTK15 – SND (left) and MMD(right).*

MMD (see Table 3). Therefore, on parallel machines, orderings generated by SND are *expected* to provide better performance [10, 16]. It must be noted that given an elimination tree, it is possible to postprocess the tree to find an equivalent reordering (in terms of fills and operations) that attempts to minimize tree height[11, 13]. For the purpose of our experiments, however, we consider only the base orderings generated by the algorithms.

Effects of Ordering on Factorization: The ordering of columns in a sparse matrix determines the number of operations executed and the concurrency available in the factorization phase. The two ordering algorithms studied have distinct goals; while MMD's objective is to reduce the number of floating point operations performed, SND optimizes elimination tree balance. To understand the effects of ordering on factorization time, we compare the best-case times (using the partitioning and factoring algorithm with the best performance on 32 processors) of MMD and SND for each of our input files. Table 4 summarizes the results of our experiments.

While comparing the two ordering algorithms, it is important to keep in mind that the cost of executing SND is an order of magnitude greater than that of MMD. Given the orderings generated by the algorithms, for every input file except GRID100, the best factorization time results from using the ordering with less fill. By ordering columns so that the elimination trees are short and wide, SND generates more concurrency than MMD. Notice that for all input files, the speedups with respect to performance on one processor is higher for SND, thus indicating greater parallelism.

TABLE 4

Best-case times (in seconds). Speedup is the ratio of execution time on 1 processor to the execution time on 32 processors.

Input	*Ordering*	*Best-case*	1	2	4	8	16	32	*Speedup*
BCSSTK29	SND	Fanout-PM	157.76	82.01	46.19	27.23	15.86	11.64	13.55
	MMD	Fanout-BM	100.93[a]	45.48	26.26	15.16	10.48	8.28	12.19[a]
TK17	SND	Fanout-BM	79.183	39.88	21.24	12.00	8.63	5.88	13.47
	MMD	Fanout-BM	32.53	18.29	10.43	6.57	5.23	4.06	8.01
BCSSTK15	SND	Fanin-PM	37.75	20.47	12.40	7.24	4.05	2.35	16.06
	MMD	Fanout-BM	32.20	16.80	10.39	6.79	4.82	3.72	8.66
GRID100	SND	Fanin-PM	7.20	3.49	2.35	1.45	0.76	0.54	13.33
	MMD	Fanin-PM	6.43	4.69	2.24	1.57	1.04	0.76	8.46
BCSPWR10	SND	Fanin-PM	0.66	0.38	0.22	0.13	0.10	0.09	7.36
	MMD	Fanout-PM	0.32	0.28	0.18	0.11	0.09	0.06	5.33

[a] Due to the large memory requirements of Blocked Cholesky, we were unable to run it on BCSSTK29 on 1 processor. The time reported is the time taken by the panel-based Fanout code on one processor.

However, except for GRID100, the increase in concurrency is not enough to offset the cost of executing a much higher number of floating point operations. In order to understand whether the increased parallelism in SND has a greater effect on a larger number of processors, we ran experiments on the larger files, TK17 and BCSSTK29, on 64 processors. We found that the orderings generated by MMD continue to have better factorization times indicating that fill is still the determining factor (TK17: MMD – 3.24s SND – 4.06s, BCSSTK29: MMD – 5.23s SND – 6.37s).

Notice that for BCSSTK15, for up to eight processors, Block Fanout-MMD performs better than Fanin-PM-SND, even though the ordering generated by SND has less fill. The Block Fanout algorithm performs well when there are few small blocks, and hence, on a small number of processors both Block Fanout-MMD and Block Fanout-SND outperform Fanin-PM-SND (with Block Fanout-SND being the best). As the number of processors gets larger, partitioning in Block Fanout results in a higher number of small blocks. The overhead induced by the small blocks causes the efficiency of Block Fanout-MMD and Block Fanout-SND to drop significantly. Consequently, Fanin-PM-SND has the best performance on 32 processors on BCSSTK15.

5. Partitioning. The concurrency and communication in the factorization phase depends strongly on the mapping of the matrix to the processors. In the following sections, we examine the role of granularity in partitioning, and evaluate the effects of the three partitioning schemes on factorization.

5.1. Panel vs Column-based partitioning. Unlike column-based methods, panel-based mappings can take advantage of the similar structures of columns in a panel to achieve better cache utilization, to use more efficient kernels and to send fewer messages. This increased efficiency comes at the cost of less concurrency. Rothberg [19] has shown that panel-based methods have good performance for distributed shared-memory machines. In this section, we analyze the benefits of panel-based mappings on message-passing systems.

As can be seen from Table 5 (For each file, we use the ordering, partitioning

and factorization algorithm that has the best result on 32 processors), panel-based methods offer substantial performance gains over column-based methods. The panel sizes in these experiments were chosen using the algorithm in the SPLASH Cholesky code [22], which uses the number of processors and size of the cache to determine the appropriate panel size. To quantify the relative benefits of improved sequential efficiency and fewer messages, and the cost of decreased concurrency, we analyze one input, BCSSTK29, in detail. We carefully modified the panel-based code in several stages to isolate the effects of each parameter (see Table 6).

1. **Panel-based:** Original panel-based version.
2. **Loop Unrolling:** Unrolling the loops that perform panel-column modifications results in a more efficient sequential kernel. Disabling this optimization causes performance to drop by 24%.
3. **Caching and Scattering:** Panel-based methods perform fewer scatters by coalescing updates from a panel into a buffer and scattering the aggregated update into the destination column. Panel-column updates also have better cache utilization. For example, in panel-column updates, the destination column can be reused in all updates from a panel. In column-column updates, the destination column has to be loaded into the cache for each update. In order to quantify the gains from cache locality and fewer scattering operations, we modified the code to perform only column-column updates. This results in an additional performance loss of 40%.
   Due to reasons of code-structure, our code still performs panel-column updates within a panel. The benefits of fewer scatters and cache locality are still present in intra-panel updates. Therefore, the performance loss number underestimates the actual loss.
4. **Message Vectorization:** While retaining a panel-based mapping, we modified the code so that instead of sending a panel at a time to processors, each column in a panel is processed and sent separately. Disabling the vectorization of messages results in a further performance loss of 45%.
5. **Column-based:** Removing all vestiges of panel-based code by partitioning the matrix on the basis of columns achieves a performance gain of 27% over Stage 4.

Note the difference in the factorization times of Stages 4 and 5. Though the column-based method has a higher communication volume and sends more messages, it does substantially better than the panel-based method in Stage 4. The panel-based method allocates tasks at a coarser granularity, and as a result, suffers from a lack of concurrency. Consider the elimination of a panel consisting of three columns a, b, c. In a panel-based method, a, b and c would be allocated to the same processor. One must first eliminate a, modify b and c and only then can we eliminate b. In a column-based approach, a, b and c would be allocated to different processors. After eliminating a, at the cost of more messages, b and c could be modified concurrently. We believe that the lack of concurrency in Stage 4 accounts for the difference in execution times between Stage 4 and Stage 5.

A more general conclusion evident from these experiments is that it is important to consider sequential efficiency in algorithm design for parallel machines. Despite having less concurrency, the panel-based method outperforms the column-based method due to its more efficient sequential kernel. Abstract models of parallel computation should take this factor into consideration in order to derive algorithms that perform well on

TABLE 5
Panel-based and Column-based execution times (in seconds)

Name	*Best-case*	*Panel*	*Column*
BCSSTK29	MMD-WM-Fanout	10.62	19.82
TK17	MMD-WM-Fanout	5.75	10.45
BCSSTK15	SND-PM-Fanin	2.35	3.74
GRID100	SND-PM-Fanin	0.54	0.87
BCSPWR10	MMD-PM-Fanout	0.06	0.09

practical machines.

TABLE 6
BCSSTK29, MMD, WM, Fanout on 32 processors

Stage	*Description*	*Time(s)*	*Msgs*	*Comm. Vol(MB)*
1	Panel-based	10.6	48,704	15.94
2	No Loop Unrolling	13.3	48,704	15.94
3	No scattering benefits	18.6	48,704	15.94
4	No message vectorization	27.0	307,584	15.94
5	Column-based	19.8	712,064	25.48

5.2. Effects of Partitioning on Factorization. We use two criteria for comparing the three mapping algorithms: the load balance of the generated mappings, and the effects on execution time of the numerical factorization phase.

Load Balance: We use the *coefficient of variation* (CV) to evaluate the load balance of the mappings generated by each partitioning algorithm. The coefficient of variation is calculated by dividing the standard deviation of the work per processor, in terms of floating point operations, by the average amount of work per processor. The result is an unitless statistic that quantifies the variation in the work allocated to processors. A higher CV represents a higher variation in processor workloads. We compare panel-based versions of WM and PM with BM.

From Table 7, it can be seen that WM generally has lower load imbalance than the other two schemes. WM achieves good load and communication balance among the processors, but the communication volume is usually higher than in PM[10] and BM. In PM, at each step, the proportion of the processor set needed by a subtree must be approximated by an integer. This rounding error, compounded over several stages, results in the higher load imbalance in PM. PM, however, usually has more parallelism since the algorithm attempts to allocate disjoint subtrees to different processors. BM's mapping consists of two stages. The domains are divided among processors using a scheme similar to PM. The blocks, however, are distributed among processors in a 2-D wrapped fashion. The load balance of the mappings generated by BM depends on the relative sizes of blocks and domains. On small files, the larger granularity of partitioning in BM causes it to perform worse than the other two partitionings.

Effects of Partitioning on Factorization: In this discussion, we use the panel-based versions of Wrapped Mapping and Proportional Mapping since they outperform the column-based versions. Tables 8–11 summarize the results of our experiments.

TABLE 7
Coefficients of variation of processor workloads (32 processors)

	SND			MMD		
Name	*PM*	*WM*	*BM*	*PM*	*WM*	*BM*
BCSSTK29	0.210	0.268	0.154	0.336	0.137	0.264
TK17	0.374	0.207	0.190	0.243	0.116	0.191
BCSSTK15	0.195	0.245	0.434	0.311	0.143	0.320
GRID100	0.229	0.213	0.394	0.266	0.114	0.389
BCSPWR10	0.388	0.239	1.134	0.319	0.205	0.664

1. **Fanin:** In the Fanin algorithm, all descendents of a column collaborate on calculating updates to the column. By assigning disjoint subtrees to disjoint sets of processors, PM has better concurrency and lower communication volume than WM. With WM, in order to eliminate a column, updates potentially have to be gathered from each processor, while in PM, updates arrive only from the set of processors allocated to the subtree rooted at that column. The increased amount of work per processor in calculating updates to a column in PM is more than offset by the savings in communication time. Table 8 reports the factorization times and communication volume for both mappings on 32 processors. Notice that PM applied to the more balanced trees produced by SND results in the least amount of communication.

2. **Fanout:** PM does not offer any special benefits to Fanout as it does for Fanin. With both mapping schemes, in the worst case, each panel would have to be sent to every processor. However, since in practice, many of the dependencies between columns are local, restricting the processors that are allocated to subtrees of the elimination tree (as is done by PM) results in lower communication volume than WM (see Table 9). With respect to factorization times for the two mapping schemes (Table 9), one sees no clear advantage to either method. The quality of mappings generated by both methods is highly dependent on the structure of the input files and the number of processors used.

3. **Block Fanout:** Table 11 summarizes the results of our experiments with Block Fanout. Our experiments show that while the block-based partitioning performs better than panel-based partitionings on the larger files in our input set, BCSSTK29 and TK17, the reverse is true for the smaller files (see Table 10). When the blocks are small, that is when the matrix is small or the number of processors is large, the overhead of managing small blocks is greater than any gain achieved from additional concurrency.

From the data in Tables 8–11, it can be seen that communication volume does not correlate strongly with execution time. For example, for BCSSTK29, Fanout-MMD-WM takes 10.62 seconds to factorize with a total communication volume of 15.94 Mbytes/processor, while Fanin-MMD-WM with a communication volume of 13.68 Mbytes/processor takes 15.35 seconds. For the same file, Fanout-MMD-PM executes in 12.90 seconds with a communication volume of 11.92 Mbytes/processor.

Communication affects the execution time of sparse Cholesky factorization by

TABLE 8
Factorization time and Communication volume for Fanin on 32 processors

Input	Execution Time (sec)				Comm. Volume(MBytes)			
	SND		MMD		SND		MMD	
	PM	*WM*	*PM*	*WM*	*PM*	*WM*	*PM*	*WM*
BCSSTK29	12.27	23.06	11.96	15.35	3.59	18.87	4.63	13.67
TK17	8.15	14.48	5.79	7.46	2.87	10.58	2.84	7.08
BCSSTK15	2.35	5.31	3.91	5.97	1.65	5.19	2.64	6.52
GRID100	0.54	1.87	0.76	1.68	0.39	1.87	0.46	1.64
BCSPWR10	0.09	0.39	0.06	0.25	0.04	0.22	0.01	0.09

TABLE 9
Factorization time and Communication volume for Fanout on 32 processors

Input	Execution Time (sec)				Comm. Volume(MBytes)			
	SND		MMD		SND		MMD	
	PM	*WM*	*PM*	*WM*	*PM*	*WM*	*PM*	*WM*
BCSSTK29	11.64	14.56	12.90	10.62	15.39	23.92	11.92	15.94
TK17	7.90	7.56	6.00	5.75	6.28	8.46	6.10	7.49
BCSSTK15	2.73	3.33	4.47	4.66	5.64	6.70	6.22	7.43
GRID100	0.82	1.01	0.84	1.03	1.62	2.11	1.23	1.58
BCSPWR10	0.10	0.17	0.06	0.10	0.11	0.20	0.03	0.65

TABLE 10
Block Fanout vs Panel-based algorithms(best-case times) on 32 processors

Input	Blocked	Panel
BCSSTK29	8.28	10.62
TK17	4.06	5.75
BCSSTK15	2.78	2.35
GRID100	0.74	0.54
BCSPWR10	0.12	0.06

TABLE 11
Factorization time and Communication volume for Blocked Fanout on 32 processors

	Time (sec)		Volume(MBytes)	
Input	SND	MMD	SND	MMD
BCSSTK29	12.60	8.28	17.11	11.58
TK17	5.88	4.06	9.67	7.60
BCSSTK15	2.78	3.72	4.67	5.62
GRID100	0.76	0.74	1.15	1.11
BCSPWR10	0.25	0.12	0.20	0.07

magnifying the effects of lack of concurrency. Experiments reveal that processors spend over 40% of the time idle waiting for work. When there is little concurrency, communication lies in the critical path of execution, and hence, increases the factorization time. For partitionings and orderings with more parallelism, communication volume does not have as great an effect since more of the communication between processors is overlapped with communication between other processors. A simple example should illustrate this point. Consider a machine with p processors where the cost of sending and receiving a message is c. Sending a message from each processor i, $0 \leq i < \frac{p}{2}$, to processor $\frac{p}{2} + i$ concurrently would take c steps. Sending the messages sequentially — send a message from processor 0 to processor $\frac{p}{2}$, wait for it to be received, then send a message from processor 1 to $\frac{p}{2} + 1$ and so on — results in an execution time of $\frac{p}{2}c$ steps.

Our experiments reveal the danger in relying on static metrics such as communication volume and workload distribution. For example, for GRID100, MMD-WM has the best load balance on 32 processors (Table 7), but has considerably worse factorization times than the PM mappings. Rather than derive conclusions about execution time based on these static metrics, one must measure the effects that these issues have on concurrency. An appropriate metric for quantifying the concurrency of sparse Cholesky factorization is, however, yet to be discovered.

6. Summary. Though our experiments support the hypotheses made by other researchers[10, 16] that balanced elimination trees generated by SND have more parallelism, for most of our input files, the ordering algorithm with less fill performed better. We were not able attain good speedups for the factorization process – the best-case efficiency on 32 processors is 50%. Therefore, we believe that on large-scale parallel machines, it would be more efficient to pipeline the different stages of the linear system solution process than to allocate all processors to the factorization of the matrix.

The choice of partitioning granularity is an important factor in the efficiency of factorization algorithms. Panel-based methods perform substantially better than column-based methods on the input files leading us to conclude that on moderately-sized parallel machines, sacrificing concurrency for sequential efficiency can result in better factorization times. The effectiveness of the partitioning algorithms is strongly dependent on the structure of the input file and the number of processors used. The factorization times are not correlated to the load balance or communication volume of the partitionings, but depend on the concurrency of the mapping.

In our experiments, no factorization algorithm emerges as the algorithm of choice. For the larger files, Block Fanout coupled with an ordering algorithm that optimizes fill such as MMD performs well. Since Block Fanout suffers from high overhead when blocks are small, it does not do well on smaller files or when the number of processors is large relative to file size. On these files, we believe that the Fanin algorithm would take good advantage of the concurrency in the mappings generated by PM to be the most efficient. It must be noted that the current implementation of Block Fanout requires large amounts of memory. In addition, the initialization step is more expensive than in the panel-based algorithms. On some machines, these factors may preclude the use of Block Fanout.

Given the lack of parallelism in sparse Cholesky factorization, we plan to study the pipelining of factorization and solution algorithms to achieve higher efficiency.

Several emerging architectures, such as Typhoon [18], allow the use of both shared memory and message passing in an application and provide greater opportunities for overlapping communication with computation. We intend to study how factorization algorithms can take advantage of these features to achieve high performance.

Acknowledgments. We wish to thank Dr. Joseph Liu and Dr. Alex Pothen for providing us with the source code for MMD and SND respectively. We are grateful to Prof. Douglas Clark and Dr. Liu for their comments and suggestions. We would also like to thank Prof. M. Raghavachari for his help with statistical concepts. Mukund Raghavachari was supported, in part, by ARPA grant N00014-91-J-4039 and NSF grant MIP-9420653.

REFERENCES

[1] C. Ashcraft, S. C.isenstat, and J. Liu, *A fan-in algorithm for distributed sparse numerical factorization*, SIAM Journal on Scientific and Statistical Computing, 11 (1990), pp. 593–599.

[2] C. Ashcraft, S. Eisenstat, J. Liu, and A. Sherman, *A comparison of three column-based distributed sparse factorization schemes*, Tech. Rep. YALEU/DCS/RR-810, Dept of Computer Science, Yale University, New Haven, CT, 1990.

[3] C. Ashcraft, R. Grimes, J. Lewis, B. Peyton, and H. D. Simon, *Progress in sparse matrix methods for large linear systems on vector supercomputers*, International Journal of Supercomputer Applications, 1 (1987), pp. 10–30.

[4] C. Leiserson et al, *The network architecture of the connection machine CM-5*, in Symposium on Parallel and Distributed Algorithms, June 1992, pp. 272–285.

[5] I. Duff, R. G. Grimes, and J. Lewis, *Users' guide for the Harwell-Boeing sparse matrix collection(release 1)*, Tech. Rep. TR/PA/92/86, Boeing Computer Services, Seattle, Wa, 1992.

[6] A. George, *Nested dissection of a regular finite element mesh*, SIAM Journal on Numerical Analysis, 10 (1973), pp. 345–363.

[7] A. George, M. Heath, J. Liu, and E. Ng, *Sparse cholesky factorization on a local-memory multiprocessor*, SIAM Journal on Scientific and Statistical Computing, 9 (1988), pp. 327–340.

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

[9] A. George, J. Liu, and E. Ng, *Communication results for parallel sparse Cholesky factorization on a hypercube*, Parallel Computing, 10 (1989), pp. 287–298.

[10] M. Heath, E. Ng, and B. Peyton, *Parallel algorithms for sparse linear systems*, SIAM Review, 33 (1991), pp. 420–464.

[11] Lewis, Peyton, and Pothen, *A fast algorithm for reordering sparse matrices for parallel factorizaton*, SIAM Journal on Scientific and Statistical Computing, 10 (1989), pp. 1146–1172.

[12] J. Liu, *Modification of the multiple minimum degree algorithm by multiple elimination*, ACM Transactions on Mathematical Software, 11 (1985), pp. 141–153.

[13] ———, *Equivalent sparse matrix reordering by elimination tree rotations*, SIAM Journal on Scientific and Statistical Computing, 9 (1988), pp. 424–444.

[14] ———, *The role of elimination trees in sparse factorization*, SIAM Journal on Matrix Analysis and Applications, 11 (1990), pp. 134–172.

[15] ———, *The multifrontal method for sparse matrix solution: Theory and practice*, Siam Review, 34 (1992), pp. 82–109.

[16] A. Pothen, H. Simon, L. Wang, and S. Barnard, *Towards a fast implementation of spectral nested dissection*, in Supercomputing '92, 1992, pp. 42–51.

[17] A. Pothen and C. Sun, *A distributed multifrontal algorithm using clique trees*, Tech. Rep. CTC91TR72, Cornell University Theory Center, Cornell University, Ithaca, NY, August 1991.

[18] S. Reinhardt, J. Larus, and D. Wood, *Tempest and Typhoon: user-level shared memory*, in Proceedings of the 21st Annual International Symposium on Computer Architecture, Apr. 1994.

[19] E. Rothberg, *Exploiting the memory hierarchy in sequential and sparse Cholesky factorization*, PhD thesis, Stanford University Department of Computer Science, Jan. 1993.

[20] E. Rothberg and A. Gupta, *An efficient block-oriented approach to parallel sparse cholesky factorization*, SIAM Journal on Scientific Computing, 15 (1994), pp. 1413–1439.

[21] R. Schreiber, *Scalability of sparse direct solvers*, Tech. Rep. RIACS TR 92.13, NASA Ames Research Center, Moffet Field, CA, May 1992.

[22] J. Singh, W.-D.Weber, and A. Gupta, *SPLASH: Stanford parallel applications for shared memory*, Computer Architecture News, 20 (1992), pp. 5–44.

[23] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser, *Active messages: a mechanism for integrated commmunication and computing*, in Proceedings of the International Symposium on Computer Architecture, May 1992, pp. 256–266.

[24] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, *Methodological considerations and characterization of the SPLASH2 parallel applications*, in Proceedings of the International Symposium on Computer Architecture, June 1995.

[25] M. Yannakakis, *Computing the minimum fill is NP-complete*, SIAM Journal on Algebraic Discrete Methods, 2 (1981), pp. 77–79.