SIAM REVIEW
Vol. 33, No. 3, pp. 420–460, September 1991

004

PARALLEL ALGORITHMS FOR SPARSE LINEAR SYSTEMS*

MICHAEL T. HEATH†‡, ESMOND NG†, AND BARRY W. PEYTON†

Abstract. This paper surveys recent progress in the development of parallel algorithms for solving sparse linear systems on computer architectures having multiple processors. Attention is focused on direct methods for solving sparse symmetric positive definite systems, specifically by Cholesky factorization. Recent progress on parallel algorithms is surveyed for all phases of the solution process, including ordering, symbolic factorization, numeric factorization, and triangular solution.

Key words. parallel algorithms, sparse linear systems, Cholesky factorization

AMS(MOS) subject classifications. 65F, 65W

1. Introduction.

Dense matrix computations are of such central importance in scientific computing that they are usually among the first algorithms implemented in any new computing environment. The need for high performance on common operations such as matrix multiplication and solving systems of linear equations has had a strong influence on the design of many architectures, compilers, etc., and such computations have become standard benchmarks for evaluating the performance of new computer systems. A survey of parallel algorithms for dense matrix computations is given in [34]. Sparse matrix computations are equally as important and pervasive, but both their performance and their influence on computer system design have tended to lag behind those of their dense matrix counterparts. In a sense this relative lack of attention and success is not surprising: sparse matrix computations involve more complex algorithms, sophisticated data structures, and irregular memory reference patterns, making efficient implementations on novel architectures substantially more difficult to achieve than for dense matrix computations. It could plausibly be argued, however, that the greater complexity and irregularity of sparse matrix computations make them much more realistic representatives of typical scientific computations, and therefore even more useful as design targets and benchmark criteria than the dense matrix computations that have usually played this role.

Despite the difficulty and relative neglect of sparse matrix computations on advanced computer architectures, there have been some notable successes in attaining very high performance (e.g., [14]), and the needs of sparse matrix computations have had some effect on computer design (e.g., the inclusion of scatter/gather instructions on some vector supercomputers). Nevertheless, it is ironic that sparse matrix computations contain more inherent parallelism than the corresponding dense matrix computations (in a sense to be discussed below), yet typically show significantly lower efficiency on today's parallel architectures. In this paper we will examine the reasons for this state of affairs, reviewing the major issues and progress to date in sparse matrix computations on parallel computer architectures. In addition to surveying the literature in this area, we will try to sketch the conceptual framework in which this work has taken place. To keep the scope of the article within reasonable

* Received by the editors September 12, 1990; accepted for publication October 10, 1990.

† Mathematical Sciences Section, Oak Ridge National Laboratory, P.O. Box 2009, Oak Ridge, Tennessee 37831-8083. This research was supported by the Applied Mathematical Sciences Research Program, Office of Energy Research, U.S. Department of Energy under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems, Incorporated.

‡ Present address, Computer Science Department, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801.

bounds, we will focus our attention on the solution of sparse symmetric positive definite linear systems by Cholesky factorization. There has, of course, also been progress on parallel algorithms for other matrix problems (e.g., nonsymmetric linear systems, least squares, eigenvalues), other factorizations (e.g., LU and QR), and other basic approaches (e.g., iterative methods), but a comprehensive treatment of all of these topics would easily require an entire book. Our discussion of sparse Cholesky factorization illustrates some of the major issues that also arise in other parallel sparse matrix factorizations as well, but there are many additional issues associated with parallel iterative algorithms or parallel sparse eigenvalue algorithms that we do not specifically address.

An outline of the paper is as follows. First, we will sketch briefly some necessary background material on serial algorithms for solving sparse symmetric positive definite linear systems. For a much more complete treatment, the reader should consult [25] or [47]. We then survey the progress to date in developing parallel implementations for each of the major phases of the solution process. We will see that the same graph theoretic tools originally developed for analyzing sequential sparse matrix algorithms also play a critical role in understanding parallel algorithms as well. We conclude with some observations on future research directions.

2. Background. Consider a system of linear equations

$$Ax = b,$$

where A is an $n \times n$ symmetric positive definite matrix, b is a known vector, and x is the unknown solution vector to be computed. One way to solve the linear system is first to compute the Cholesky factorization

$$A = LL^T,$$

where the Cholesky factor L is a lower triangular matrix with positive diagonal elements. Then the solution vector x can be computed by successive forward and back substitutions to solve the triangular systems

$$Ly = b, \qquad L^T x = y.$$

If A is a sparse matrix, meaning that most of its entries are zero, then during the course of the factorization some entries that are initially zero in the lower triangle of A may become nonzero entries in L. These entries of L are known as *fill* or *fill-in*. Usually, however, many zero entries in the lower triangle of A remain zero in L. For efficient use of computer memory and processing time, it is desirable for the amount of fill to be small, and to store and operate on only the nonzero entries of A and L.

It is well known that row or column interchanges are not required to maintain numerical stability in the factorization process when A is positive definite. Furthermore, when roundoff errors are ignored, a given linear system yields the same solution regardless of the particular order in which the equations and unknowns are numbered. This freedom in choosing the ordering can be exploited to enhance the preservation of sparsity in the Cholesky factorization process. More precisely, let P be any permutation matrix. Since PAP^T is also a symmetric positive definite matrix, we can choose P based solely on sparsity considerations. That is, we can often choose P so that the Cholesky factor $\bar{L}$ of PAP^T has less fill than L. The permuted system is equally useful for solving the original linear system, with the triangular solution phase simply

becoming

$$\bar{L}y = Pb, \quad \bar{L}^T z = y, \quad x = P^T z.$$

Unfortunately, finding a permutation P that minimizes fill is a very difficult combinatorial problem (an NP-complete problem) [107]. Thus, a great deal of research effort has been devoted to developing good heuristics for limiting fill in sparse Cholesky factorization, including the nested dissection algorithm [39], [45] and the minimum degree algorithm [48], [70], [98]. Limiting fill is also the primary motivation for a number of methods based on reducing the bandwidth or profile of A. These band-oriented methods have been less successful, however, than the more general sparse ordering techniques, and as we shall see, they are at an even greater disadvantage in a parallel context.

Since pivoting is not required in the factorization process, once the ordering is known, the precise locations of all fill entries in L can be predicted in advance*, so that a data structure can be set up to accommodate L before any numeric computation begins. This data structure need not be modified during subsequent computations, which is a distinct advantage in terms of efficiency. The process by which the nonzero structure of L is determined in advance is called "symbolic factorization." Thus, the direct solution of $Ax = b$ consists of the following sequence of four distinct steps:

1. *Ordering.* Find a good ordering P for A; that is, determine a permutation matrix P so that the Cholesky factor L of PAP^T suffers little fill.
2. *Symbolic factorization.* Determine the structure of L and set up a data structure in which to store A and compute the nonzero entries of L.
3. *Numeric factorization.* Insert the nonzeros of A into the data structure and compute the Cholesky factor L of PAP^T.
4. *Triangular solution.* Solve $Ly = Pb$ and $L^T z = y$, and then set $x = P^T z$.

Note that the first two steps are entirely symbolic, involving no floating-point computation. Several software packages [17], [27], [29] for serial computers use this basic approach to solve sparse symmetric positive definite linear systems. We now briefly discuss algorithms and methods for performing each of these steps on sequential machines.

2.1. Ordering. As might be expected from the combinatorial nature of the ordering problem for sparse factorization, graph theory has proved to be an extremely helpful tool in modeling the symbolic or structural aspects of sparse elimination algorithms. The use of a graph theoretic model dates to the early work of Parter [91] and Rose [97], and has now come to permeate the subject. The graph of an $n \times n$ symmetric matrix A, denoted by $G(A)$, is a labeled undirected graph having n vertices (or nodes), with an edge between two vertices i and j if the corresponding entry a_{ij} is nonzero in the matrix. The structural effect of Gaussian elimination on the matrix is then easily described in terms of the corresponding graph. The fill introduced into the matrix as a result of eliminating a variable adds fill edges to the corresponding graph precisely so that the neighbors of the eliminated vertex become a clique. This fact suggests that fill can be limited, or at least postponed, by eliminating first those vertices having fewest neighbors (i.e., vertices of lowest degree). The elimination or factorization process can thus be modeled by a sequence of graphs, each having one

* We assume that exact cancellation never occurs, and thus *fill* refers to the *structural nonzeros* of L, i.e., every location of the factor that is occupied by a nonzero entry at some point in the factorization.

less vertex than the previous graph but possibly gaining edges, until only one vertex remains. We will also have occasion to refer to the *filled graph*, $F(A)$, which is the graph of A with all fill edges added (i.e., there is an edge between two vertices i and j of $F(A)$, with $i > j$, if $\ell_{ij} \neq 0$ in the Cholesky factor matrix L).

The foregoing discussion provides the basis for the minimum degree algorithm, which is the most successful and widely applicable heuristic developed to date for limiting fill in sparse Cholesky factorization. At each step of the elimination process, this simple heuristic selects as the next node to be eliminated a node of minimum degree in the current elimination graph. Despite its simplicity, the minimum degree algorithm produces reasonably good orderings over a remarkably broad range of problem classes. Another strength is its efficiency: as a result of a number of refinements over several years, current implementations are extremely efficient on most problems. George and Liu [48] review a series of enhancements to implementations of the minimum degree algorithm and demonstrate the consequent reductions in ordering time.

As might be expected from the "greedy" nature of the algorithm, however, several weaknesses of the minimum degree ordering heuristic are well documented in the literature. Experiments in both [24] and [48] illustrate the sensitivity of the quality of minimum degree orderings to the way ties are broken when there is more than one node of minimum degree from which to choose. Attempts to make the selection more intelligent or less myopic, however, have proven to be computationally expensive. No tie-breaking scheme proposed to date is both effective and efficient, though some interesting results using deficiency (the number of fill edges created by the elimination step) to break ties are reported in [15]. Berman and Schnitger [13] show that for a model problem there exists a minimum degree tie-breaking scheme for which the time and space complexity of the factorization is worse than that of known asymptotically optimal orderings. To summarize, minimum degree is, on balance, an effective and efficient ordering heuristic, but its success is not well understood, and no robust and efficient way is known for dealing with the wide variability in the quality of the orderings it produces.

Another effective algorithm for limiting fill in Cholesky factorization is nested dissection, which is based on a divide-and-conquer paradigm. Let S be a set of nodes (called a separator) whose removal, along with all edges incident on nodes in S, divides the graph into at least two remaining pieces. If the matrix is reordered so that the variables in each piece are numbered contiguously and the variables in the separator are numbered last, then the matrix will have a bordered block diagonal nonzero pattern. More importantly, elimination of a node within one of the pieces cannot introduce fill into any of the other pieces; fill is restricted to the diagonal blocks and the border [47], [99]. This idea can be applied recursively, breaking the pieces into smaller and smaller pieces with successive sets of separators, giving a nested sequence of dissections of the graph. The effectiveness of nested dissection in limiting fill is highly dependent on the size of the separators used to split the graph. For highly regular, planar problems (e.g., two-dimensional finite difference or finite element grids), suitably small separators can usually be found [68], [69]. For problems in dimensions higher than two, or for highly irregular problems with less localized connectivity, nested dissection is much less effective. Nevertheless, nested dissection is important not only for its practical usefulness on suitable problems, but also for its asymptotically optimal fill properties for certain model problems, which serves as a kind of theoretical benchmark for the quality of orderings [39], [60].

2.2. Symbolic factorization. A naive approach to symbolic factorization is simply to carry out Cholesky factorization on the structure of A symbolically. However, such an algorithm would then have the same time complexity as the numeric factorization itself (i.e., it would require the same number of symbolic operations as the number of floating-point operations required by the numeric factorization). With a little care, the complexity of symbolic factorization can be reduced to $O(\eta(L))$, where $\eta(L)$ denotes the number of nonzeros in L, as follows.

For a given sparse matrix M, define

$$\text{Struct}(M_{i*}) := \{k < i \mid m_{ik} \neq 0\},$$

$$\text{Struct}(M_{*j}) := \{k > j \mid m_{kj} \neq 0\}.$$

In other words, $\text{Struct}(M_{i*})$ is the sparsity structure of row i of the strict lower triangle of M, and $\text{Struct}(M_{*j})$ is the sparsity structure of column j of the strict lower triangle of M. For a given lower triangular Cholesky factor matrix L, define the function p as follows:

$$p(j) := \begin{cases} \min\{i \in \text{Struct}(L_{*j})\}, & \text{if } \text{Struct}(L_{*j}) \neq \emptyset, \\ j, & \text{otherwise.} \end{cases}$$

Thus, when there is at least one off-diagonal nonzero in column j of L, $p(j)$ is the row index of the first off-diagonal nonzero in that column. It is easy to show that

$$\text{Struct}(L_{*j}) \subseteq \text{Struct}(L_{*,p(j)}) \cup \{p(j)\}.$$

Moreover, it can be shown that the structure of column j of L can be characterized as follows [47]:

$$\text{Struct}(L_{*j}) := \text{Struct}(A_{*j}) \cup \left(\bigcup_{i<j} \{\text{Struct}(L_{*i}) \mid p(i) = j\} \right) - \{j\}.$$

That is, the structure of column j of L is given by the structure of the lower triangular portion of column j of A, together with the structure of each column of L whose first off-diagonal nonzero is in row j. This characterization leads directly to an algorithm for performing the symbolic factorization, shown in Fig. 1, in which the sets R_j are used to record the columns of L whose structures will affect that of column j of L.

This simple symbolic factorization algorithm is already very efficient, with time and space complexity $O(\eta(L))$, but it is subject to further refinement. For example, if R_j contains only one column, say i, and $\text{Struct}(A_{*j}) \subseteq \text{Struct}(L_{*i})$, then clearly $\text{Struct}(L_{*j}) = \text{Struct}(L_{*i}) - \{j\}$. This shortcut can be used to speed up the symbolic factorization algorithm and to reduce the storage requirements using a technique known as "subscript compression" [104]. In fact, these conditions are often satisfied when j is relatively large, as the columns tend to become more dense toward the end of the factorization. An efficient implementation of the symbolic factorization algorithm is presented in [47]. With its low complexity and an efficient implementation, the symbolic factorization step usually requires less computation than any of the other three steps in solving a symmetric positive definite system by Cholesky factorization.

Once the structure of L is known, a compact data structure is set up to accommodate all of its nonzero entries. Since only the nonzero entries of the matrix are

```
for j := 1 to n do
    R_j := ∅
for j := 1 to n do
    S := Struct(A_{*j})
    for i ∈ R_j do
        S := S ∪ Struct(L_{*i}) − {j}
    Struct(L_{*j}) := S
    if Struct(L_{*j}) ≠ ∅ then
        p(j) := min {i ∈ Struct(L_{*j})}
        R_{p(j)} := R_{p(j)} ∪ {j}
```

FIG. 1. *Symbolic factorization algorithm.*

stored, additional indexing information must be stored to indicate the locations of the nonzeros. Although this integer overhead potentially rivals the space requirements for the nonzeros themselves, in practice the subscript compression technique mentioned above greatly reduces this overhead storage [46].

2.3. Numeric factorization. In its simplest form, Gaussian elimination on a dense matrix A can be described as a triple nested loop around the single statement

$$a_{ij} = a_{ij} - (a_{ik}a_{kj})/a_{kk}.$$

The loop indices i, j, and k can be nested in any order, each with a different pattern of memory access. This freedom can be exploited to take better advantage of particular architectural features of a given machine (cache, virtual memory, vectorization, etc.) [21]. Specializing to Cholesky factorization, where symmetry is exploited so that only the lower triangle of the matrix is accessed, we see that there are three basic types of algorithms, depending on which of the three indices is placed in the outer loop:

1. *Row-Cholesky.* Taking i in the outer loop, successive rows of L are computed one by one, with the inner loops solving a triangular system for each new row in terms of the previously computed rows.
2. *Column-Cholesky.* Taking j in the outer loop, successive columns of L are computed one by one, with the inner loops computing a matrix-vector product that gives the effect of previously computed columns on the column currently being computed.
3. *Submatrix-Cholesky.* Taking k in the outer loop, successive columns of L are computed one by one, with the inner loops applying the current column as a rank-1 update to the remaining partially-reduced submatrix.

These three families of algorithms have markedly different memory reference patterns in terms of which parts of the matrix are accessed and modified at each stage of the factorization (see Fig. 2), and each has its advantages and disadvantages in a given context. For sparse Cholesky factorization, row-Cholesky is seldom used because of the difficulty in designing a compact row-oriented data structure for storing the nonzeros of L that can also be accessed efficiently in the numerical factorization phase [71]. Efficient implementation of sparse row-Cholesky is even more difficult on vector and parallel architectures since it is difficult to vectorize or parallelize sparse triangular solutions (see discussions in §§2.4 and 3.5). We will therefore concentrate our attention on the two column-oriented methods, column-Cholesky and submatrix-Cholesky.

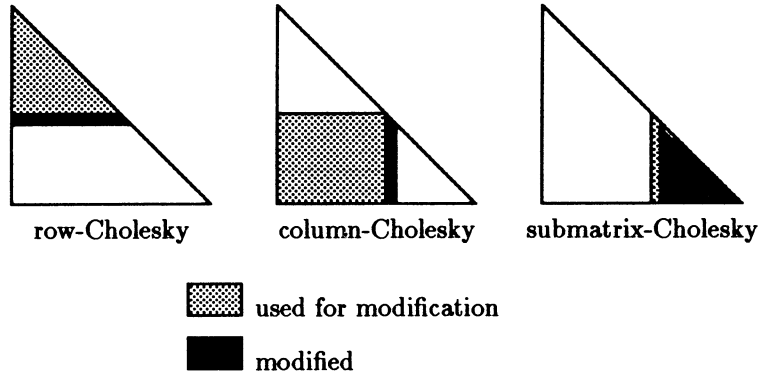

FIG. 2. *Three forms of Cholesky factorization.*

In column-oriented Cholesky factorization algorithms, there are two fundamental types of subtasks:

1. $\mathrm{cmod}(j,k)$: modification of column j by column k, $k < j$,
2. $\mathrm{cdiv}(j)$: division of column j by a scalar.

These sparse matrix operations correspond to `saxpy` and `sscal` in the terminology of the BLAS [64] for dense linear algebra, but we use different notation to emphasize that we are dealing with their sparse counterparts. In terms of these basic operations, high-level descriptions of the column-Cholesky and submatrix-Cholesky algorithms are given in Figs. 3 and 4.

for $j = 1$ **to** n **do**
  for $k \in \mathrm{Struct}(L_{j*})$ **do**
    $\mathrm{cmod}(j,k)$
  $\mathrm{cdiv}(j)$

FIG. 3. *Sparse column-Cholesky factorization algorithm.*

for $k = 1$ **to** n **do**
  $\mathrm{cdiv}(k)$
  for $j \in \mathrm{Struct}(L_{*k})$ **do**
    $\mathrm{cmod}(j,k)$

FIG. 4. *Sparse submatrix-Cholesky factorization algorithm.*

In column-Cholesky, column j of A remains unchanged until the index of the outer loop takes on that particular value. At that point the algorithm updates column j with a nonzero multiple of each column $k < j$ of L for which $\ell_{jk} \neq 0$. After all column modifications have been applied to column j, the diagonal entry ℓ_{jj} is computed and used to scale the completely updated column to obtain the remaining nonzero entries of L_{*j}. Column-Cholesky is sometimes said to be a "left-looking" algorithm, since at each stage it accesses needed columns to the left of the current column in the matrix. It can also be viewed as a "demand-driven" algorithm, since the inner

products that affect a given column are not accumulated until actually needed to modify and complete that column. It is also sometimes referred to as a "fan-in" algorithm, since the basic operation is to combine the effects of multiple previous columns on a single subsequent column. The column-Cholesky algorithm is the most commonly used method in commercially available sparse matrix packages [17], [27], [29].

In submatrix-Cholesky, as soon as column k is completed, its effects on all subsequent columns are computed immediately. Thus, submatrix-Cholesky is sometimes said to be a "right-looking" algorithm, since at each stage columns to the right of the current column are modified. It can also be viewed as a "data-driven" algorithm, since each new column is used as soon as it is completed to make all modifications to all the subsequent columns it affects. It is also sometimes referred to as a "fan-out" algorithm, since the basic operation is for a single column to affect multiple subsequent columns. We will see that these characterizations of the column-Cholesky and submatrix-Cholesky algorithms have important implications for parallel implementations.

Having stated the "pure" column- and submatrix-Cholesky algorithms, we note that many variations and hybrid implementations of these schemes are possible, which essentially amount to different ways of amalgamating partial results. For example, frontal methods [61], and their generalizations to multifrontal methods [28], are essentially variations on submatrix-Cholesky. But while the $\mathrm{cmod}(j,k)$ updating operations are computed in the order shown in Fig. 4, they are not applied directly to the column j being updated. Instead they are accumulated and passed on through a succession of update matrices until finally they are incorporated into the target column. The reason for this approach is that in the frontal method most of the matrix is kept out of core on auxiliary storage, with only a relatively small "frontal" matrix representing currently "active" columns kept in main memory. Similarly, the out-of-core version of the multifrontal method can be implemented so that only a few small "frontal" matrices are kept in main memory. To minimize I/O traffic, access to inactive portions of the matrix, both columns already completed and columns yet unreduced, must be kept to a minimum. For further details on multifrontal methods, see [28] or [78].

One of the main motivations for frontal and multifrontal methods is that the frontal matrices can be treated as dense, and therefore we can take advantage of vectorization more readily on hardware that supports it [3], [5], [11], [19]. Moreover, the localization of memory references in these methods is advantageous in exploiting cache [100] or on machines with virtual memory and paging [76].

Before leaving the general topic of sparse factorization, we introduce two additional concepts that are useful in analyzing and efficiently implementing sparse factorization algorithms. A *supernode* is a set of contiguous columns in the Cholesky factor L that share essentially the same sparsity structure. More specifically, the set of contiguous columns $j, j+1, \cdots, j+t$ constitutes a supernode if $\mathrm{Struct}(L_{*,k}) = \mathrm{Struct}(L_{*,k+1}) \cup \{k+1\}$ for $j \le k \le j+t-1$. A set of supernodes for an example matrix is shown in Fig. 5. Columns in the same supernode can be treated as a unit for both computation and storage. Supernodes have long played an important role in enhancing the efficiency of both the minimum degree ordering [50] and the symbolic factorization [104]. More recently, supernodes have been used to organize sparse factorization algorithms around matrix-vector or matrix-matrix operations that reduce memory traffic by making more efficient use of vector registers [5], [11] or cache [3], [100]. The cited reports document the substantial gains in performance obtained by

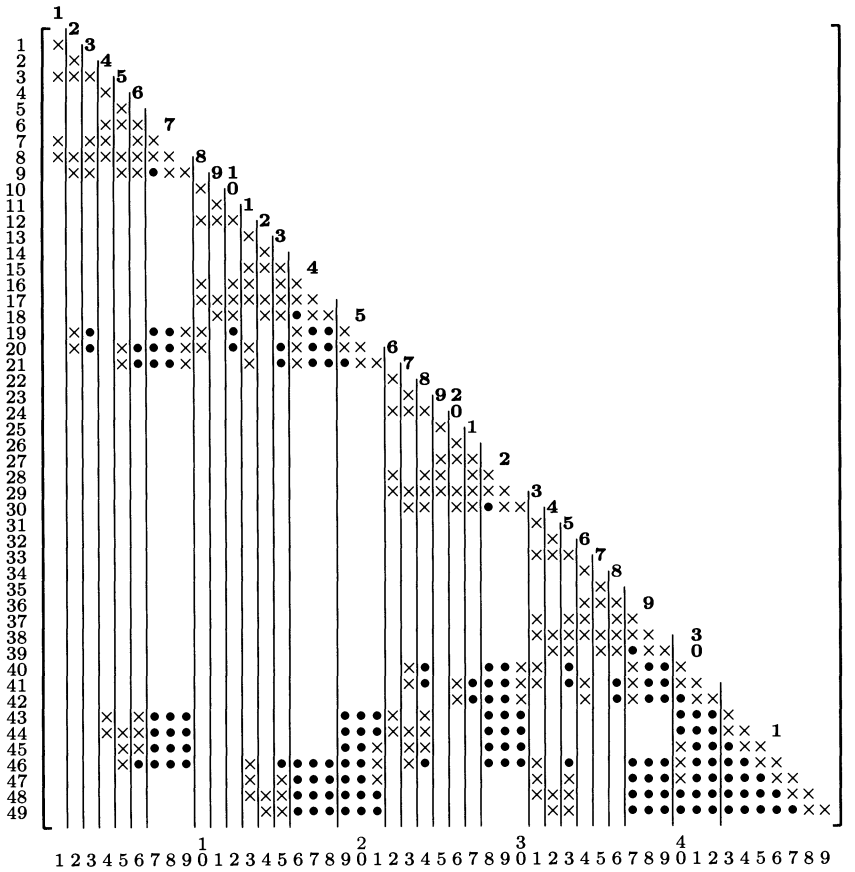

FIG. 5. *Supernodes for* 7×7 *nine-point grid problem ordered by nested dissection.* ($\times$ *and* $\bullet$ *refer to nonzeros in* A *and fill in* L*, respectively. Numbers over diagonal entries label supernodes.*)

using these techniques.

The *elimination tree* $T(A)$ [71], [103] associated with the Cholesky factor L of a given matrix A has $\{1, 2, \cdots, n\}$ as its node set, and has an edge between two vertices i and j, with $i > j$, if $i = p(j)$, where p is the function defined in § 2.2. In this case, node i is said to be the *parent* of node j, and node j is a *child* of node i. Liu [79] discusses the many uses of elimination trees in sparse matrix computations. Among these is their use in managing the frontal and update matrices in the multifrontal method. Another key role is in the analysis of data dependencies that must be observed when factoring the matrix, which has obvious implications for implementing the factorization in parallel. Figure 6 shows the elimination tree for the matrix shown in Fig. 5.

Let $T[j]$ denote the subtree of $T(A)$ rooted at node j. It is shown in [71] and [103] that the set of columns/nodes that modify column/node j (namely, the set $\text{Struct}(L_{j*})$) is a subset of $T[j]$ denoted by $T_r[j]$. Moreover, $T_r[j]$ is also a subtree of $T(A)$ rooted at node j. For this reason, $T_r[j]$ is called the *row subtree of* j. It follows that column j can be completed only after every column in $T_r[j]$ has been computed. It also follows that the columns that receive updates from column j are ancestors of j in $T(A)$. In other words, the node set $\text{Struct}(L_{*j})$ is a subset of the ancestors of j

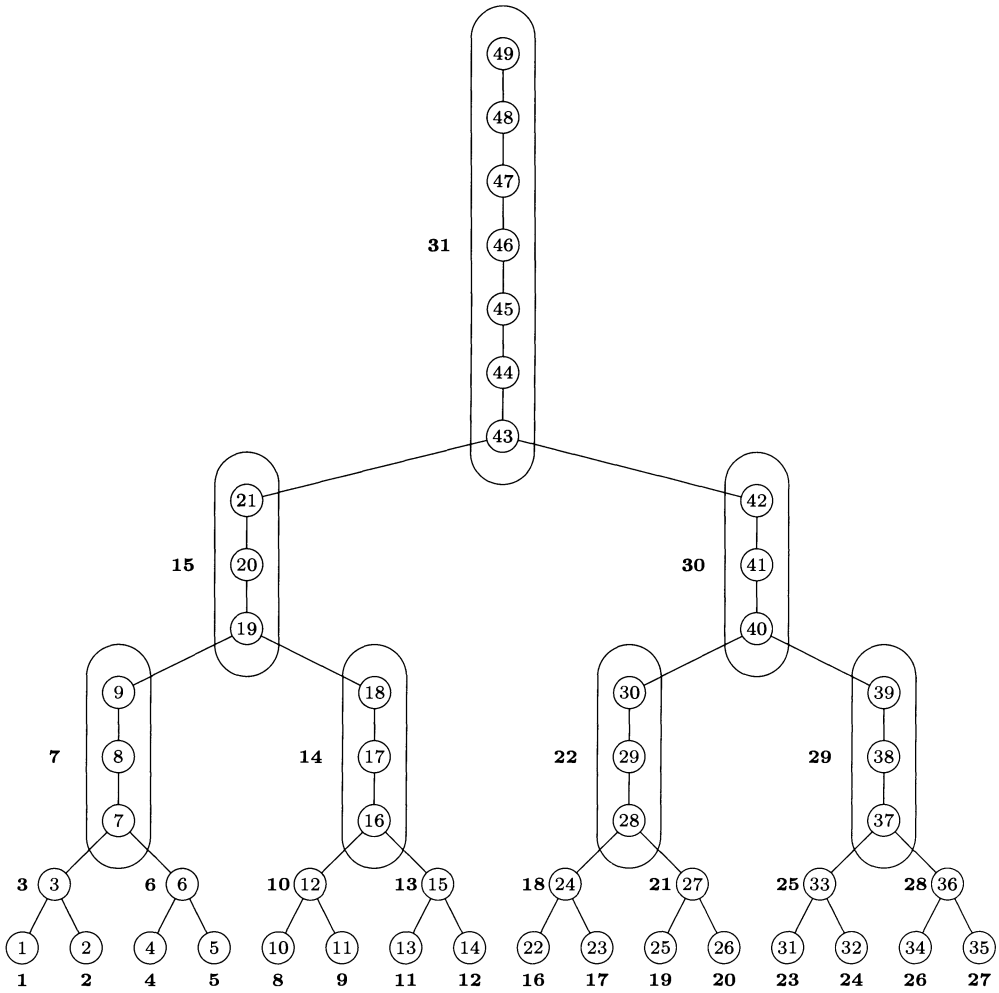

FIG. 6. *Elimination tree for the matrix shown in Fig.* 5. *Ovals enclose supernodes that contain more than one node. Nodes not enclosed by an oval are singleton supernodes. Boldface numbers label supernodes.*

in the tree.

2.4. Triangular solution. There is relatively little to be said about the triangular solution step. The structure of the forward and back substitution algorithms is more or less dictated by the sparse data structure used to store the triangular Cholesky factor L and by the structure of the elimination tree $T(A)$. Because triangular solution requires many fewer floating-point operations than the factorization step that precedes it, the triangular solution step usually requires only a small fraction of the total time to solve a sparse linear system on conventional sequential computers. These proportions can change, however, with more advanced computer architectures, since it is often more difficult to take full advantage of vector or parallel processors in performing triangular solutions. We will discuss these issues in greater detail in §3.5.

3. Parallel algorithms. In this section we summarize the progress to date in adapting direct methods for the solution of sparse symmetric positive definite lin-

ear systems to perform well on the various parallel architectures that have become available in recent years. The most widely available and commercially successful parallel architectures thus far fall into three rough categories: shared-memory MIMD (multiple-instruction, multiple-data stream) architectures typically having 30 or fewer processors, distributed-memory MIMD architectures typically having on the order of 32 to 1024 processors, and SIMD (single-instruction, multiple-data stream) architectures typically having tens of thousands of processors. Some machines have an additional level of parallelism in the form of vector units within each individual processor. Parallel architectures display an enormous variation in the number and power of processors, organization of memory, control mechanisms, and synchronization and communication overhead, so it is not surprising that they demand a comparable range of algorithmic techniques to achieve good efficiency in the various settings. Nevertheless, we will try to concentrate on general principles that are widely applicable, while focusing occasionally on implementation issues that may arise in a more specific context.

In exploiting parallelism to solve any problem, the computational work must be broken into a number of subtasks that can be assigned to separate processors. The most appropriate number and size of these tasks (e.g., a small number of large tasks or a large number of small tasks) depend on the target parallel architecture and the levels at which parallelism naturally occurs in the problem. The term often used to denote the size of computational tasks in a parallel implementation is *granularity*. In sparse factorization, as in most problems, a number of levels of computational granularity can potentially be exploited. Liu [72] uses the elimination tree to analyze the following levels of parallelism in Cholesky factorization:

1. *fine-grain parallelism*, in which each task is a single floating-point operation or *flop*, i.e., multiply-add pair,
2. *medium-grain parallelism*, in which each task is a single cmod or cdiv column operation,
3. *large-grain parallelism*, in which each task is the completion of all columns in a subtree of the elimination tree.

Here, large-grain parallelism refers to the independent work done in computing columns in disjoint subtrees. Consider two disjoint subtrees $T[j]$ and $T[i]$, where neither root node is a descendent of the other. All work required to compute the columns of $T[j]$ is completely independent of all work required to compute the columns of $T[i]$. For example, in Fig. 6 the columns of $T[9]$ (columns 1–9) are completely independent of the columns of $T[18]$ (columns 10–18). This type of parallelism is available only in sparse factorization; it is not available in the dense case. But of course we are not limited to exploiting only parallelism of this nature. There is much more parallelism to be found at the medium-grain level of the individual cmod operations. Let j_1 and j_2 be two column indices whose subtrees $T[j_1]$ and $T[j_2]$ are *not* disjoint. Suppose that k_1 and k_2 are indices of columns that must be used to modify columns j_1 and j_2, respectively. Clearly, the updates $\text{cmod}(j_1, k_1)$ and $\text{cmod}(j_2, k_2)$ can be performed in parallel. This is the primary source of parallelism in the dense case, and it is an extremely important source of parallelism in the sparse case as well.

While we will have a great deal to say about algorithms that employ the first two sources of parallelism, we will have little to say about finer grain parallelism. Fine-grain parallelism can be exploited in two distinctly different ways:

1. vectorization of the column operations cmod and cdiv on vector supercomputers,

2. parallelizing the rank-1 update that constitutes a major step of submatrix-Cholesky on an SIMD machine.

Exploiting vectorization requires some changes and refinement of the basic sequential algorithms [3], [5], [11], [19], but it does not require changes as extensive and basic as those required to exploit higher levels of parallelism. Developing parallel sparse submatrix-Cholesky algorithms for SIMD machines presents a more difficult challenge, and research on this topic is still in its infancy [54].

To date, implementation on parallel architectures has caused no fundamental change in the overall high-level approach to solving sparse symmetric positive definite linear systems. On parallel machines the same sequence of four distinct steps is performed: ordering, symbolic factorization, numeric factorization, and triangular solution. However, both shared-memory and distributed-memory MIMD machines require an additional step to be performed: the tasks into which the problem is decomposed must be mapped onto the processors. Obviously, one of the goals in mapping the problem onto the processors is to ensure that the work load is balanced across all processors. Moreover, it is desirable to schedule the problem so that the amount of synchronization and/or communication is low. On shared-memory machines the scheduling problem is relatively easy to deal with: a shared queue of tasks can be used to achieve dynamic load balancing. Dynamic load balancing tends to be inefficient on current distributed-memory machines, however, so a static assignment of tasks to processors must be determined in advance.

We now proceed to discuss the progress made in developing parallel algorithms for each of these five steps.

3.1. Ordering. There are two distinct issues associated with the ordering problem in a parallel environment:

1. Determining an ordering appropriate for performing the subsequent factorization efficiently on the parallel architecture in question.
2. Computing the ordering itself *in parallel*.

3.1.1. Orderings for parallel factorization. On sequential or vector machines, while there are sometimes other secondary considerations, the primary goal of reordering the matrix is simply to lower the work and space required for the factorization step. Experience and intuition suggest that the two almost inevitably rise and fall together, so that the goal can be further simplified to lowering fill only. Simply lowering fill, however, may not provide an ordering appropriate for parallel factorization.

Orderings for a tridiagonal system serve to illustrate the point. Let us call the ordering that preserves the tridiagonal structure the *natural ordering*. Under the natural ordering, the matrix incurs no fill during factorization. In fact, both the fill and work are minimized. Nevertheless, the natural ordering is the poorest possible ordering for parallel factorization. First, note that the natural ordering results in an elimination tree that is a *chain* (see Fig. 7). Indeed, there is no large-grain (subtree-level parallelism) to exploit. Moreover, each column j, $2 \leq j \leq n$, requires a single column modification $\mathrm{cmod}(j, j-1)$ before it can be completed with the $\mathrm{cdiv}(j)$ operation, then and only then becoming available for the subsequent column modification $\mathrm{cmod}(j+1, j)$. Thus, there is no medium-grain (column-modification level) parallelism to exploit. There is also no fine-grain parallelism to exploit. Thus, there is no parallelism at all to exploit in the floating-point computation; the floating-point work is strictly sequential. But it is well known that even-odd reduction schemes for these systems, though they introduce more work, also greatly increase the parallelism.

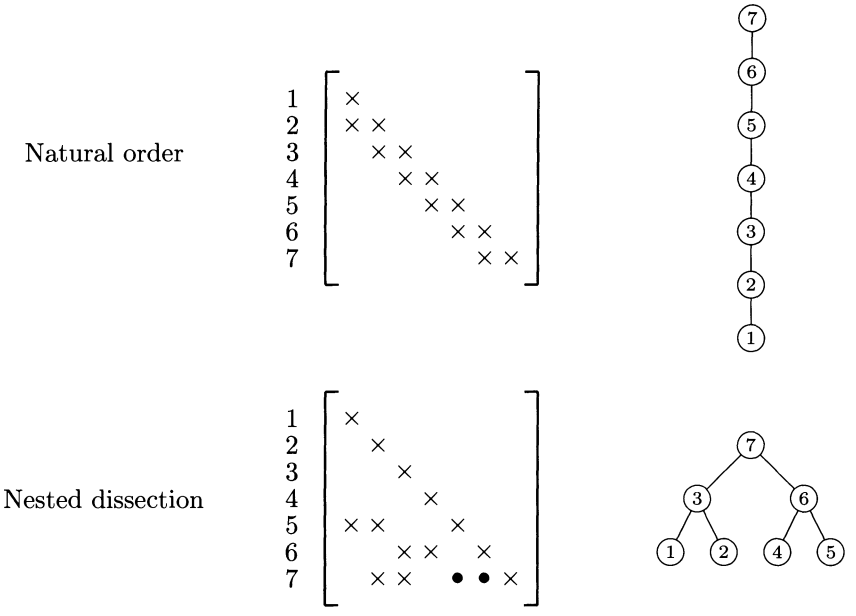

FIG. 7. *Factor matrices and corresponding elimination trees for tridiagonal matrix using natural ordering and nested dissection reordering (even-odd reduction).* × *and* • *refer to original nonzeros and fill nonzeros, respectively.*

These solution schemes are equivalent to reordering the system with a nested dissection ordering (again, see Fig. 7). Using the nested dissection ordering, the height of the elimination tree is approximately $\log_2 n$, which is much shorter than the height $n - 1$ obtained using the natural ordering. While the total floating-point work (ignoring square roots) increases by a factor between two and three, parallel completion time using the nested dissection ordering is ideally $O(\log n)$ compared with $O(n)$ using the standard ordering.

This example is an extreme illustration of how inappropriate the goal of fill-reduction can be in the parallel setting. However, there have been no systematic attempts to develop metrics for measuring the quality of parallel orderings. Thus far, most work on the parallel ordering problem has used elimination tree height as the criterion for comparing orderings, with short trees assumed to be superior to taller trees [62], [65], [77], [80], but with little more than intuition as a basis for this choice. For massively parallel SIMD machines, it has been suggested that small elimination tree height may indeed be a suitable goal [54], [65]. This contention is based on the assumption of a submatrix-Cholesky parallel factorization algorithm that requires roughly uniform time for the elimination of each column. It remains to be shown that this assumption is in fact realized for sparse problems on available SIMD machines. The assumption is more doubtful on other parallel architectures. Moreover, it is worth noting that the problem of ordering a matrix to minimize its elimination tree height, like the problem of minimizing fill, is a very difficult combinatorial problem [93]. In [77], Liu suggests some more realistic measures of *parallel completion time*, but there is not yet an agreed upon objective function for the parallel ordering problem.

3.1.2. Computing the ordering in parallel. A separate problem is the need to compute the ordering in parallel on the same machine on which the other steps of the solution process are to be performed. The highly sophisticated ordering algorithms discussed earlier, namely, minimum degree and nested dissection, are extremely efficient and normally constitute only a small fraction of the total execution time in solving a sparse system on sequential computers. Despite the limited potential for any gain in execution time, however, there is still motivation for adapting these algorithms, or developing new ones, to run on parallel architectures, especially in the case of distributed-memory machines. In particular, a distributed implementation of the ordering step is necessary to take advantage of the large amount of local memory available on such machines in solving very large problems. Otherwise, the ordering step will remain a bottleneck limiting the size of problems that can be solved on distributed-memory parallel architectures. We now consider some of the difficulties in performing the ordering step efficiently in parallel.

The basic minimum degree algorithm has an inherently sequential outer loop, with a single node eliminated at each stage. Multiple elimination of independent nodes of minimum or near-minimum degree [70], [92] could potentially be exploited to permit parallel execution. Moreover, the search for nodes of minimum degree and the necessary graph transformations and degree updates could conceivably be spread across multiple processors. However, there are several problems with this approach. First, it is not clear that minimum degree orderings would be particularly appropriate for parallel factorization. For example, applying the basic minimum degree algorithm to the tridiagonal system discussed above produces an elimination tree that is a chain, and thus the resulting elimination tree height would be at least $\lfloor n/2 \rfloor$. Duff et al. [26] contains several suggestions for dealing with this problem, the most promising of which increases the size of the independent sets by allowing all nodes whose degrees are within a constant factor α of the current minimum degree, where $1.1 \leq \alpha \leq 1.5$, to be candidates for inclusion in the next independent set. A different approach for computing independent sets for parallel elimination is described in [66]. Second, the highly successful enhancements incorporated into current implementations of the method [48] have resulted in an intricate and extremely efficient algorithm: there is very little work to be partitioned among the processors, and that work is of a highly irregular and somewhat sequential nature. Nevertheless, an algorithm based on this approach has been developed for use on a massively parallel SIMD machine [54]. It is possible that such an approach could also be reasonably effective on some shared-memory MIMD machines, but we know of no such implementations. It is doubtful, however, that this approach would have acceptable efficiency on distributed-memory MIMD machines, and we are not aware of any attempt to produce such an implementation. It is ironic that much of the research on parallel algorithms for sparse factorization has been performed on the latter class of machines, yet it is on this class of machines that the ordering problem seems most difficult to address.

The standard nested dissection ordering heuristic [45] would appear to offer much greater opportunity for an effective parallel implementation. The divide-and-conquer paradigm introduces a natural source of parallelism, both in computing the ordering and in subsequently using it for the factorization step, due to the independence of the successive pieces into which the graph is split. Unfortunately, there are also difficulties with this approach. First, the nested dissection heuristic (based on the generation of level structures) is effective in reducing fill for a much more restricted class of problems than minimum degree. Second, the divide-and-conquer approach provides

only a logarithmic potential speedup, with relatively little parallelism in the first few levels of the dissection. Third, for a distributed-memory implementation there is something of a bootstrapping problem: in order to utilize all of the local memory and simultaneously minimize interprocessor communication costs, the original graph should be distributed across the processors in some intelligent way *before* the dissection process is begun. Finally, nested dissection is similar to minimum degree in that it enjoys a very efficient sequential implementation, and its primary subtask (generating a level structure via breadth-first search) is inherently serial.

To summarize this discussion, it is evident that the problem of computing effective parallel orderings *in parallel* is very difficult and remains largely untouched by research efforts to date. We focus our attention in the remainder of this section on the effectiveness of various ordering strategies in facilitating the subsequent parallel factorization, with little regard for whether the ordering can itself be computed effectively in parallel.

3.1.3. Parallel ordering algorithms. We now turn our attention to the problem of ordering for parallel factorization and/or executing the ordering algorithms on the target parallel machine. As noted above, these problems are very difficult to deal with, and much work remains to be done before mature, reliable algorithms and software become available.

Tree restructuring for parallel elimination. One approach to generating low-fill orderings that are suitable for parallel sparse factorization is to decouple the reduction of fill and enhancement of parallelism into separate phases. First a standard ordering technique, such as minimum degree, is applied to produce a low-fill ordering for the matrix, then based on this initial ordering an *equivalent* reordering is produced that is more suitable for parallel factorization. By "equivalent" we mean an ordering that generates the same fill edges but may substantially restructure the elimination tree. Thus, an equivalent ordering is simply a different perfect elimination ordering for the filled graph $F(A)$ that models the sparsity structure of L determined by the initial fill-reducing ordering. The effectiveness of this approach depends in part on whether there is in fact a good parallel ordering within the class of orderings equivalent to the initial low-fill ordering. The tridiagonal example cited earlier demonstrates that there may be no such ordering. On the other hand, since some of the parallelism in sparse factorization is due specifically to sparsity, low-fill would seem to enhance potential parallelism rather than suppress it. Very little is known, however, about the conditions under which good equivalent parallel orderings might exist for realistic classes of problems.

Implementation of the equivalent ordering approach requires an initial fill-reducing ordering, a mechanism for restructuring the elimination tree, and a computable criterion for determining when a given reordering will in fact reduce the subsequent parallel factorization time. In [77], Liu uses tree rotations [73] to find equivalent orderings that reduce elimination tree height, where the initial ordering used is a minimum degree ordering. He reports substantial reductions for a number of test problems. In the same report, Liu proves that the Jess and Kees algorithm [62] produces an equivalent ordering whose associated elimination tree height is *minimum* among all equivalent orderings. In [80] Liu and Mirzaian present a practical $O(\eta(L))$ implementation of the Jess and Kees algorithm. Tests comparing Liu's tree rotations heuristic with their implementation of the Jess and Kees algorithm showed that the heuristic almost always produces a minimum-height tree. This interesting phenomenon is not fully understood. Their timings showed the tree rotations heuristic to be far more

efficient than their implementation of the Jess and Kees algorithm. In [67] a more efficient implementation of the Jess and Kees algorithm is presented. Roughly speaking, the latter implementation is linear in the number of compressed subscripts used to represent the structure of L. Tests of this implementation indicate that a Jess and Kees ordering can usually be obtained in roughly the same amount of time as an ordering using the tree rotations heuristic.

Of course, the height of the elimination tree may not be a very accurate indicator of the actual parallel factorization time. Moreover, elimination trees produced by minimum degree orderings typically have height already close to the minimum, so that the potential gain from restructuring may be quite small. Perhaps the primary problem with this approach is that it fails to get at the heart of the problem. Our intuition based on limited experience is that equivalent orderings have the capacity to modify only relatively minor features of the parallelism possessed by the initial fill-reducing ordering. Thus, this approach may be able to fine-tune an ordering for use in parallel factorization, but the key question of how much parallelism might be available in the original underlying problem goes unanswered.

Nested dissection and graph partitioning heuristics. Given the natural divide-and-conquer parallelism exhibited by nested dissection, several researchers have explored various implementations of nested dissection in an effort to generate good orderings for parallel factorization. The effectiveness of nested dissection in reducing fill and enhancing parallelism depends on graph partitioning heuristics to find small node separators for the graph. Some of the graph partitioning heuristics employed in fact produce edge separators, which then must be converted into node separators.

The basic scheme in nested dissection is as follows:

1. Use a graph partitioning heuristic to obtain a small edge separator of the graph, or more specifically, a small set of edges whose removal from the graph separates the graph into two vertex sets of roughly equal size.
2. Transform the small edge separator into a small node separator, or more specifically, a small set of nodes whose removal separates the graph into two portions of roughly equal size.
3. Number the nodes of the separator last in the ordering, and recursively apply steps 1 and 2 to the two subgraphs produced in step 2.

We now review some specific implementations of this approach.

Level structures. In [44] the adaptation of an automatic nested dissection algorithm [45] for execution on distributed-memory MIMD machines is discussed. The algorithm first generates a level structure by means of a breadth-first search. The choice of starting node in the search can be crucial; see [45] for details. Then one of the middle levels is chosen as a node separator, subdividing the problem into two or more independent subgraphs, to which the process is applied recursively. This method generates a node separator directly, and therefore omits step 1 from the general scheme given above. An advantage of this method is that it is simple and generally inexpensive to compute. But the automatic nested dissection heuristic is generally not as effective at reducing fill as the minimum degree heuristic, and thus the quality of the ordering is poorer on many, but not all, problems. As with most nested dissection algorithms, the algorithm for finding a separator appears to be inherently sequential. Thus, there is little parallelism to exploit until the ordering algorithm is several levels down into the recursion, where there are adequately many independent subproblems to work on.

Kernighan–Lin. Gilbert and Zmijewski [55] use the Kernighan–Lin heuristic [63]

to generate a small edge separator. Associated with an edge separator are *wide* and *narrow* node separators, defined as follows. Let P_1 and P_2 be the two sets of nodes into which the edge separator partitions the graph. Let V_1 contain the nodes in P_1 incident on at least one edge in the separator set, and define $V_2 \subset P_2$ in the same way. The set $V = V_1 \cup V_2$ is the associated wide separator and both V_1 and V_2 are the associated narrow separators. Gilbert and Zmijewski ran tests using both kinds of separators and report ordering times and factorization times on an Intel iPSC/1 hypercube.

Fiduccia–Mattheyses. Leiserson and Lewis [65] use a variant of the Kernighan–Lin heuristic due to Fiduccia and Mattheyses [31] to generate edge separators. They use a greedy heuristic to generate node separators from edge separators. Their heuristic is guaranteed to find a *minimal* node separator among the nodes belonging to $V = V_1 \cup V_2$. In their tests they use elimination tree height to compare the quality of their orderings with those obtained by using tree rotations to reduce the elimination tree height of minimum degree orderings. They report fairly substantial and consistent reductions in tree height for their test problems. However, they did not implement their algorithm on a parallel machine; all their tests were run on an unspecified sequential machine and no timings results were reported.

Spectral separators. Pothen, Simon, and Liou [95] study the use of *spectral partitions* [32], [33] in the framework described above. To generate an edge separator, they first compute the eigenvector y associated with the smallest positive eigenvalue of the Laplacian matrix associated with the $G(A)$. They use an implementation of the Lanczos algorithm to compute the required eigenvector for general sparse graphs. Then the median entry y_m of y is found, and the vertices in P_1 are taken to be those corresponding to entries y_i of y for which $y_i < y_m$, while the vertices in P_2 are those corresponding to entries y_i of y for which $y_i \geq y_m$. The authors use matching theory for bipartite graphs, in particular the Dulmage–Mendelsohn decomposition, to generate from the edge separator a minimum-cardinality node separator [94]. Thus, their bipartite-matching method for transforming an edge separator into a node separator is optimal in the sense that it minimizes the size of the node separator over all possible node separators that can be obtained from the given edge separator (i.e., over all separators contained in the set of nodes incident on the separator edges). The report cited here does not include statistics for complete nested dissection orderings based on this technique; it includes statistics for the top-level separator only. Since most of the time is spent performing Lanczos iterations, which can be parallelized in a fairly straightforward manner, their method should run efficiently in parallel even in the top few levels of the nested dissection recursion.

A hybrid approach. In [74] and [75] Liu presents a hybrid approach that combines elements of both the minimum degree and nested dissection algorithms. The primary emphasis of the two papers is simply to produce improved fill-reducing orderings, but the application of the method to parallel factorization is noted in both papers. The method proceeds as follows. After a standard minimum degree ordering algorithm is initially applied to the problem, a "middle" separator determined by the minimum degree ordering is chosen. A technique based on matching theory for bipartite graphs is then used to improve (i.e., shrink) this separator. The nodes of the new separator are numbered last in the ordering, and then the process is applied recursively to the subproblems remaining to be ordered.

This method generates a nested dissection ordering (a top-down ordering), but uses a minimum degree ordering (a bottom-up ordering), along with some matching

theory, to obtain the separators. Thus, it is a hybrid of two very different ordering techniques. Again, computing the ordering in parallel with this approach appears to be very difficult. However, the timings and ordering statistics reported indicate that it obtains good orderings in a reasonably efficient manner on a sequential machine.

3.2. Task partitioning and scheduling.

3.2.1. Shared-memory MIMD machines. In implementing sparse column-Cholesky on a shared-memory MIMD machine, the problem of partitioning the factorization into tasks for concurrent execution on multiple processors is fairly simple. Each column j corresponds to a task $T\mathrm{col}(j)$ defined by

$$T\mathrm{col}(j) := \{\mathrm{cmod}(j,k) \mid k \in \mathrm{Struct}(L_{j*})\} \cup \{\mathrm{cdiv}(j)\}.$$

That is, $T\mathrm{col}(j)$ consists of all column modifications, as well as the final scaling operation, to be applied to column j. The tasks $T\mathrm{col}(j)$ are maintained in a queue and doled out to processors as they complete previous tasks. Since all necessary data are globally accessible by all processors, there need be no concern over which specific processor picks up a given task. This approach achieves good load balancing dynamically, an ideal arrangement for the highly irregular task profile usually generated by sparse problems. In short, uniform access to main memory permits the use of dynamic load balancing and a fairly simple restructuring of a sequential sparse Cholesky algorithm to obtain a good parallel algorithm. See §3.4.1 and [41], [88] for parallel implementations of sparse Cholesky based on these ideas.

Efficient scheduling of the tasks $T\mathrm{col}(j)$ on shared-memory MIMD machines is also easily accomplished. An ordering of the elimination tree is a *topological* ordering if each node is numbered higher than all of its descendants. Performance usually is not very sensitive to which topological ordering is used to schedule the column tasks, and it is often adequate to use the fill-reducing ordering to schedule the tasks. In this case, the task queue Q is given by

$$Q := \{T\mathrm{col}(1), T\mathrm{col}(2), \cdots, T\mathrm{col}(n)\}.$$

However, scheduling columns by their height in the elimination tree usually improves performance by reducing synchronization delays, as shown in [88]. The ordering of the elimination tree shown in Fig. 8 is particularly appropriate. Scheduling the column tasks in this manner is especially worthwhile, since the overhead required to do so is trivial—a single n-vector computed in $O(n)$ time. A more dynamic queue management strategy is to initialize the queue to contain only the tasks corresponding to the leaf nodes, with additional column tasks appended to the queue after their descendants have been completed.

3.2.2. Distributed-memory MIMD machines. The situation is much more difficult on distributed-memory MIMD machines, the target architecture for much of the algorithm development for parallel sparse factorization reported in the literature. On these machines, the lack of globally accessible memory means that issues concerned with data locality are dominant considerations. Currently, there is no efficient means of implementing dynamic load balancing on these machines for problems of this type. Thus, a static assignment of tasks to processors is normally employed in this setting, and such a mapping must be determined in advance of the factorization, based on the tradeoffs between load balancing and the cost of interprocessor communication.

Elimination trees. As we have seen, the elimination tree contains information on data dependencies among tasks and the corresponding communication requirements.

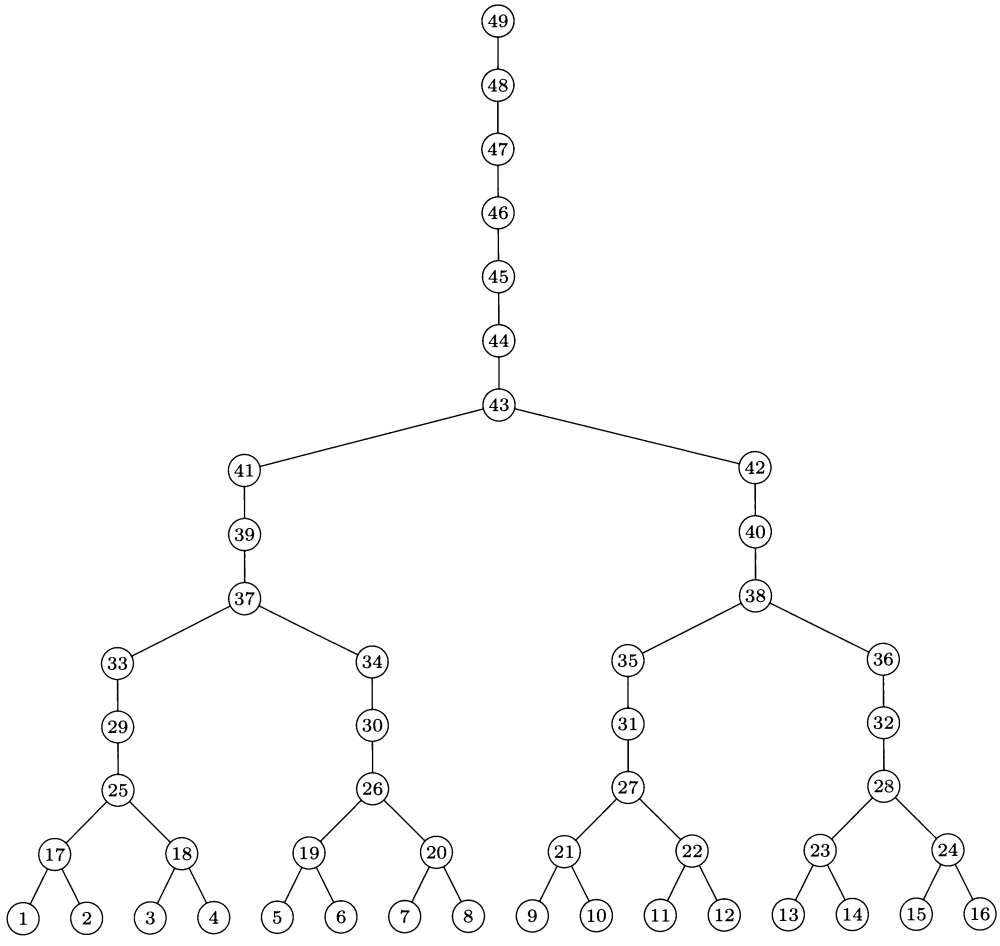

FIG. 8. *A good ordering of column tasks in task queue used by parallel column-Cholesky algorithm for shared-memory* MIMD *machines.*

Thus, the elimination tree is an extremely helpful guide in determining an effective assignment of columns (and corresponding tasks) to processors in the distributed-memory case. In attempting to compute the elimination tree, however, we appear to be confronted by a bootstrapping problem: prior to symbolic factorization, we do not yet know the structure of L on which the definition of $T(A)$ is based. Fortunately, $T(A)$ can be generated directly from the structure of A by an extremely efficient algorithm [79]. It is desirable to compute the elimination tree in parallel, but again we face the recurring problem of having very little work to distribute over the processors. For large problems, if a single processor cannot store the adjacency structure of A, then the structure of A must be distributed among the processors, which also requires distributed computation of the elimination tree. In [110], Zmijewski and Gilbert present an algorithm for computing the elimination tree in parallel on a distributed-memory multiprocessor. Roughly speaking, their algorithm proceeds as follows. Each

processor uses its portion of the adjacency structure of A to compute a "local" version of the elimination tree. In essence, this "local" tree contains in a compressed form the contribution of each processor's local adjacency list to the final elimination tree. The final phase of the algorithm combines these "local" trees to obtain the final elimination tree. All communication associated with the algorithm is restricted to this final "combining" operation. In the experiments reported in [110], the parallel algorithm takes considerably more time than the sequential algorithm, though the differences are not unreasonable.

Mapping the problem onto the processors. After the elimination tree has been generated, the next step is to use it in mapping the columns onto the processors. The primary goals of the mapping are good load balance and low interprocessor communication. These goals can be in conflict, however, especially for highly irregular problems.

In the early work on this problem, successive levels in the elimination tree were wrap-mapped to the processors, as shown in Fig. 9. This results in good load balancing for the model problem, but it also often results in unnecessarily high message volume. The "subtree-to-subcube" mapping, introduced in [49], does an excellent job of reducing communication while maintaining good load balance for model grid problems and other problems with similar regularity in their structure. Although the use of subcubes is specific to hypercube architectures, a similar processor clustering concept is applicable to most distributed-memory architectures.

The basic idea is quite simple. If P is the number of processors, select an appropriate set of P subtrees of the elimination tree, say $T_0, T_1, \cdots, T_{P-1}$, and then assign the columns corresponding to T_i to processor i $(0 \leq i \leq P-1)$. Where two subtrees merge together into a single subtree, their processor sets are merged together and wrap-mapped onto the nodes/columns of the separator that begins at that point. The root separator is wrap-mapped onto the set of all available processors. Figure 10 shows this mapping for our model problem. George, Liu, and Ng [49] show that for the fan-out distributed factorization algorithm (see §3.4.2) applied to model problems defined on $k \times k$ grids, communication volume can be limited to $O(Pk^2)$, which is asymptotically optimal. Gao and Parlett [35] prove the slightly stronger result that the communication volume for each processor is $O(k^2)$, which indicates that the overhead associated with communication is, in some sense, balanced among the processors. Closely related results can be found in two papers by Naik and Patrick [86], [87].

It is quite easy and natural to obtain a good "subtree-to-subcube" mapping for elimination trees obtained by applying standard nested dissection orderings to model problems. It is difficult, however, to generalize the subtree-to-subcube mapping to more irregular problems. Progress in that direction is reported in [38] and [101]. However, an adequate understanding of the tradeoffs between communication and load balance for more realistic problems will require further study.

3.3. Symbolic factorization. On a distributed-memory MIMD multiprocessor, it is necessary to compute $\text{Struct}(L_{*j})$ for every column j of L and to store $\text{Struct}(L_{*j})$ on the processor responsible for computing that column. Thus, a distributed algorithm for computing the symbolic factorization is required. The sequential algorithm for this step is remarkably efficient, and so once again we find ourselves with little work to distribute among the processors, so that good efficiency is difficult to achieve in a parallel implementation.

As we have seen, $\text{Struct}(L_{*j})$ depends on $\text{Struct}(A_{*j})$ and on $\text{Struct}(L_{*k})$ for every k such that $p(k) = j$ (i.e., for every child k of j in the elimination tree).

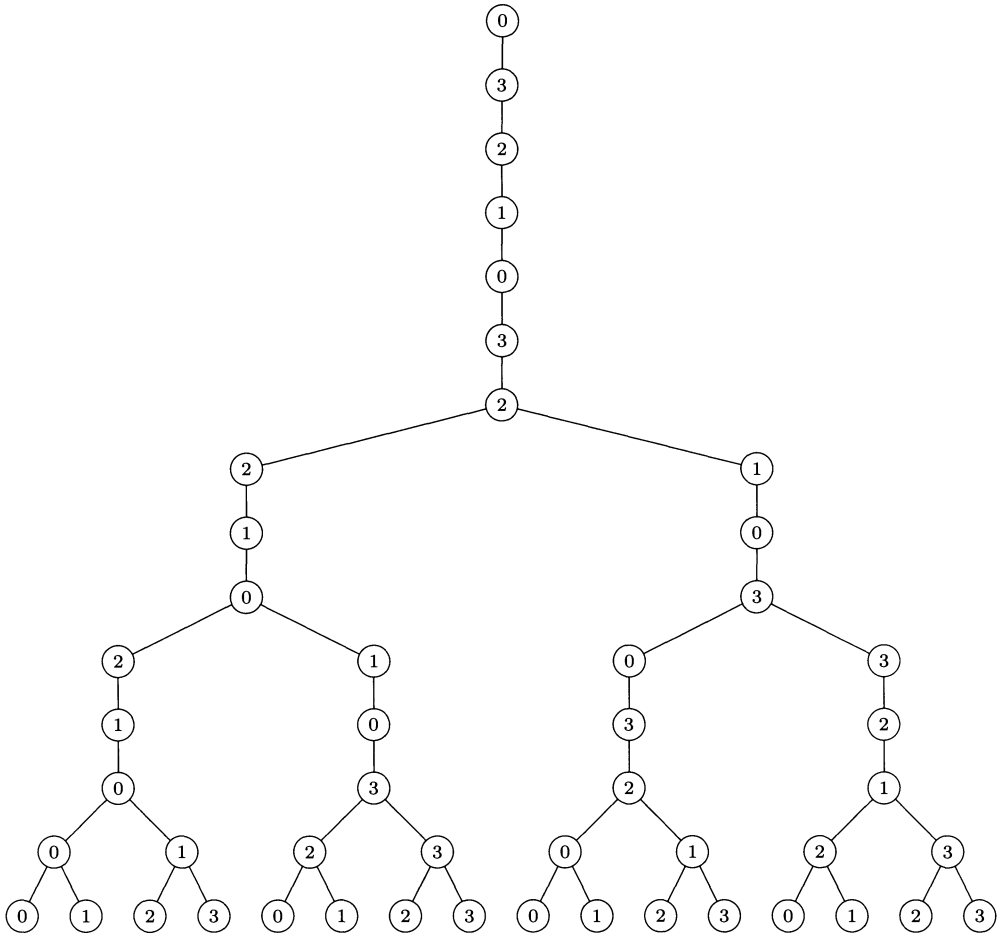

FIG. 9. *A wrap-mapping of the factor columns onto four processors numbered* 0, 1, 2, *and* 3. *Nodes belonging to the same separator in the elimination tree are assigned to the processors in wrap fashion.*

In [42] a column-oriented parallel symbolic factorization algorithm is presented. At any point during the execution of this algorithm, the number of tasks available for parallel execution is limited to the number of leaves in the subtree of the elimination tree induced by nodes whose structures are not yet complete. Limited parallelism, small task sizes, and communication overhead make it difficult to attain good speed-ups. Moreover, the subscript compression technique so critical to the space and time efficiency of the sequential symbolic factorization algorithm can be only partially realized on these machines. For example, let columns j and $j+1$ of L be two columns belonging to the same supernode but assigned to two distinct processors, say, p_0 and p_1, respectively. The sequential algorithm exploits the fact that $\text{Struct}(L_{*,j+1}) = \text{Struct}(L_{*j}) - \{j+1\}$ to save both time and storage, as discussed earlier in §2.2. The parallel algorithm, however, must store $\text{Struct}(L_{*j})$ on processor p_0 and $\text{Struct}(L_{*,j+1})$

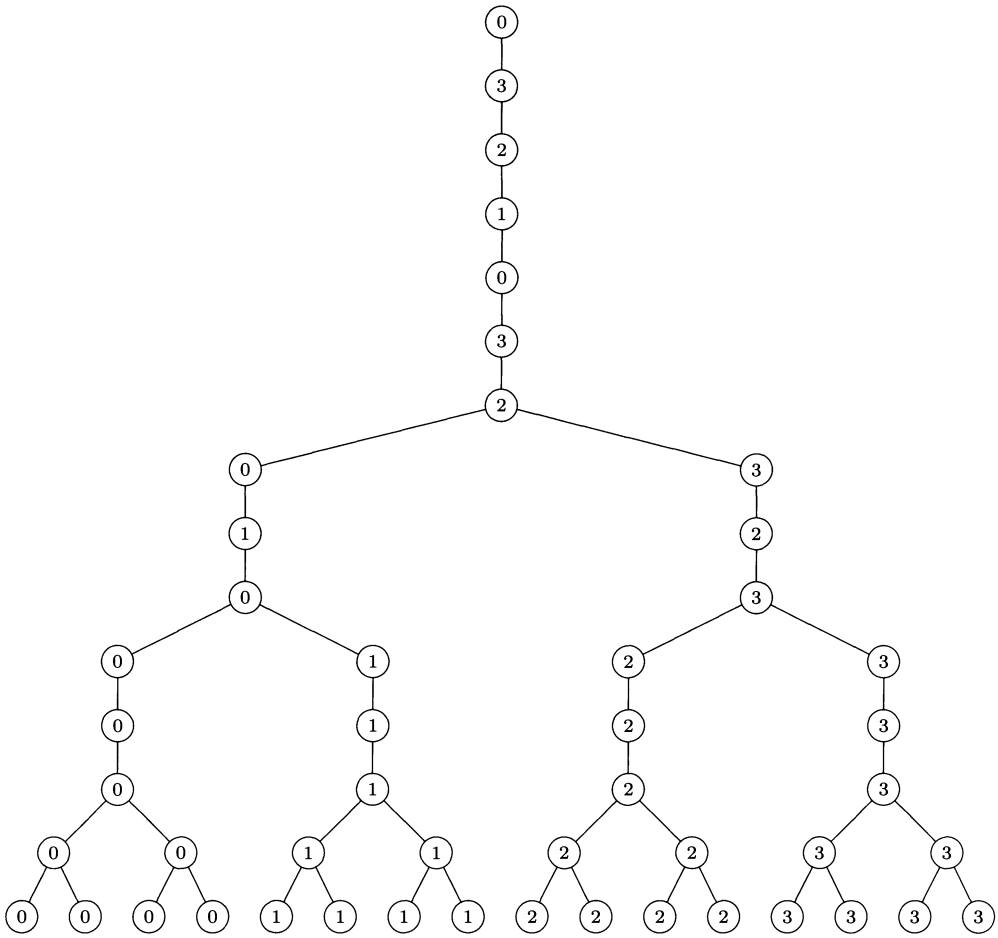

FIG. 10. *Subtree-to-subcube mapping of the columns of the matrix to four processors numbered* 0, 1, 2, *and* 3.

on processor p_1. Good mappings typically wrap-map columns belonging to the same supernode. Thus the situation in our illustration is typical—even pervasive; hence parallel symbolic factorization necessarily requires more total work and storage on distributed-memory MIMD multiprocessors, although the *parallel completion* time will usually still be less. The test results reported in [42] confirm that currently only modest speed-ups are attainable.

It is possible to improve parallel symbolic factorization on distributed-memory MIMD multiprocessors if the supernodal structure is known in advance [81]. The key observation is that it is necessary to compute only the structure of the first column of each supernode. Processors holding other columns in that supernode do not have to compute the structures of these columns; all they need to do is to retrieve the structure from the processor that is responsible for computing the structure of the first column.

In [110] Zmijewski and Gilbert present a row-oriented parallel symbolic factorization algorithm that has more potential parallelism, but is more complicated and requires rearrangement of the output into a column-oriented format. Timing results for this algorithm are not presented, but the authors indicate that its cost is high. However, the problems they experimented with were quite small, so it remains unclear how competitive the algorithm might be on larger problems. In a study [53] that may be applicable on massively parallel machines, Gilbert and Hafsteinsson show that using a shared-memory CRCW (concurrent-read, concurrent-write) PRAM (parallel random access machine) model of computation, there is a parallel algorithm for symbolic factorization that requires $O(\log^2 n)$ time using $\eta(L)$ processors.

3.4. Numeric factorization. On sequential machines, numeric factorization is typically much more expensive than the other steps in the solution process. As a result, parallel numeric factorization has received considerably more attention than the other steps in the parallel solution process. It is also more amenable to parallelization than the other solution steps, though it is still much more difficult to deal with than dense factorization. Development of reasonably good parallel sparse Cholesky algorithms has taken longer than development of their dense counterparts. The bookkeeping and irregular structure dealt with in the sparse algorithms present a greater challenge to the algorithm developer; consequently, many more issues and difficulties remain to be addressed in future work.

Most of the work has been directed towards the development of parallel algorithms that exploit medium- and large-grain parallelism on shared-memory or distributed-memory MIMD machines. Some exceptions are work on vectorizing sparse Cholesky factorization on powerful vector supercomputers [3], [5], [11], [19], work on fine-grained algorithms for massively parallel SIMD machines [54], and work on systolic-like algorithms for multiprocessor grids [18], [105]. We will restrict our discussion to algorithms designed for MIMD machines.

3.4.1. Parallel column-Cholesky for shared-memory machines. Of the three formulations of sparse Cholesky, column-Cholesky is in many ways the simplest to implement. As noted earlier, it has been more commonly used in sparse matrix software packages [17], [27], [29] than other methods, such as the multifrontal method. It is probably better known to a broader audience than the other methods. George et al. [41] show that the algorithm can be adapted in a straightforward manner to run efficiently in parallel on shared-memory MIMD machines. For all these reasons this algorithm is an ideal place to begin our discussion of parallel sparse Cholesky algorithms.

A parallel algorithm. To facilitate our discussion, we introduce a more detailed version of the column-Cholesky algorithm shown earlier in Fig. 3. In particular, we need to indicate how the row structure sets $\text{Struct}(L_{j*})$ are generated by the algorithm. The more detailed version of the algorithm shown in Fig. 11 requires the following new notation. Let $\text{next}(j,k)$, $k \leq j$, be the lowest numbered column greater than j that requires updating by column k. That is, $\text{next}(j,k)$ is the row index of the first nonzero in column k after row j. (Note that $\text{next}(j,j)$ is merely the parent of j in the elimination tree.) The column index sets S_i $(1 \leq i \leq n)$ are initially empty, but when column j is processed, $S_j = \text{Struct}(L_{j*})$, as required. For simplicity and brevity, the algorithm in Fig. 11 does not detail how to handle the case when there is no "next" column to be updated. The use of the index sets S_i and other implementation details of the serial algorithm are discussed in [47]. However, we note one particular detail in the implementation. Since each completed column k appears

in no more than one set S_i at any time during the algorithm's execution, a single n-vector *link* suffices to maintain each set S_i $(1 \leq i \leq n)$ as a singly-linked list [47].

```
for j = 1 to n do
    S_j := ∅
for j = 1 to n do
    for k ∈ S_j do
        cmod(j, k)
        i := next(j, k)
        S_i := S_i ∪ {k}
    cdiv(j)
    i := next(j, j)
    S_i := S_i ∪ {j}
```

FIG. 11. *Sparse column-Cholesky factorization algorithm, showing the computation of row structure sets* $\text{Struct}(L_{i*})$ *in the sets* S_i, $1 \leq i \leq n$.

This algorithm can be implemented in parallel on a shared-memory MIMD machine in a fairly straightforward manner [40]. Each column j corresponds to a task

$$T\text{col}(j) := \{\text{cmod}(j,k) \mid k \in \text{Struct}(L_{j*})\} \cup \{\text{cdiv}(j)\},$$

as discussed in §3.2. Initially, the task queue, denoted by Q, contains all column tasks $T\text{col}(j)$ ordered by some topological ordering of the elimination tree. For ease of notation, we assume that the elimination ordering and the schedule-prescribed ordering are the same, so we have

$$Q := \{T\text{col}(1), T\text{col}(2), \cdots, T\text{col}(n)\}.$$

As the computation proceeds, a processor obtains (and removes) the column task currently at the front of the queue and proceeds to compute that task. After completing the task, the processor obtains from Q another column task to compute, and it continues in this manner, as do all the other processors, until the factorization is complete. This simple "pool of tasks" approach does an excellent job of dynamically balancing the load, even though the column task profile for typical sparse problems is quite irregular. Obviously, access to this queue must be synchronized to ensure that each column task $T\text{col}(j)$ is executed by one and only one processor. The parallel algorithm also must synchronize access to the n-vector *link* in which the sets S_i $(1 \leq i \leq n)$ are maintained. Only one processor at a time can modify this array, and thus the two sequences of instructions that manipulate *link* must be critical sections in the algorithm. A high-level description of the parallel algorithm is given in Fig. 12.

Recent improvements. The algorithm in Fig. 12 has two significant drawbacks. First, the number of synchronization operations (obtaining and relinquishing a lock) is $O(n + \eta(L))$, which is quite high. Second, since the algorithm does not exploit supernodes, it will not vectorize well on vector supercomputers with multiple processors, natural target machines for the algorithm. The introduction of supernodes into the algorithm deals quite effectively with both problems [88].

The use of supernodes to improve computational rates on vector supercomputers is well documented [3], [5], [11], [19]. The duplicate sparsity structure found in columns within the same supernode enables us to organize the computation around level-2 or level-3 BLAS-like computational kernels. Such block operations reduce memory traffic by retaining and reusing data in cache, vector registers, or whatever

```
Q := {Tcol(1), Tcol(2), ..., Tcol(n)}
for j = 1 to n do
    S_j := ∅
while Q ≠ ∅ do
    pop Tcol(j) from Q
    while column j requires further cmod's do
        if S_j = ∅ do
            wait until S_j ≠ ∅
        obtain k from S_j
        i := next(j, k)
        lock
        S_i := S_i ∪ {k}
        unlock
        cmod(j, k)
    cdiv(j)
    i := next(j, j)
    lock
    S_i := S_i ∪ {j}
    unlock
```

FIG. 12. *Parallel sparse column-Cholesky factorization algorithm for shared-memory* MIMD *machines.*

limited rapid-access memory resource is provided on the particular machine in question.

In the following discussion, we will let boldface integers **1**, **2**, $\cdots$, **N** stand for the supernodes. Thus, $\mathbf{N} \leq n$ is the number of supernodes. We will also use boldface capital letters such as **J** and **K** to denote each supernode by its index, and use lowercase letters such as i, j, and k to denote each individual column by its number.

Let **K** be a supernode comprising the set of contiguous columns $\{k, k+1, k+2, \cdots, k+t\}$. Because of the sparsity structure shared by each column of **K**, every column of **K** modifies column j, $j > k+t$, if and only if at least one column of **K** modifies column j. For example, column 40 in supernode **30** in Fig. 5 is modified by each column 37, 38, and 39 in the previous supernode, but it is modified by none of the columns 19, 20, and 21 that compose supernode **15**. The block operation used to improve the algorithms in Figs. 3 and 12 is a level-2 BLAS-like kernel, $\mathrm{cmod}(j, \mathbf{K})$, which modifies column j with a multiple of the appropriate entries of each column $k \in \mathbf{K}$. In particular, the modifications from the columns in **K** can be accumulated as dense `saxpy` operations and no indirect addressing is required until the result is applied to column j. For a column $k+i \in \mathbf{K}$, we let $\mathrm{cmod}(k+i, \mathbf{K})$ denote the operation of updating column $k+i$ with every column of **K** numbered earlier than $k+i$. That is, $\mathrm{cmod}(k+i, \mathbf{K})$ is given by

$$\mathrm{cmod}(k+i, \mathbf{K}) := \{\mathrm{cmod}(k+i, k), \mathrm{cmod}(k+i, k+1), \cdots, \mathrm{cmod}(k+i, k+i-1)\}.$$

For the matrix in Fig. 5, $\mathrm{cmod}(30, \mathbf{22})$ is given by

$$\mathrm{cmod}(30, \mathbf{22}) := \{\mathrm{cmod}(30, 28), \mathrm{cmod}(30, 29)\}.$$

Since columns k, $k+1$, $\cdots$, $k+i-1$ in supernode **K** have the same structure below row $k+i-1$, the modifications to column $k+i$ can again be performed by dense `saxpy` operations, with no indirect addressing required. The next column to be updated by supernode **K** after it has updated column j is denoted by $\mathrm{next}(j, \mathbf{K})$, and similarly

the first column *outside* supernode $\mathbf{K}$ requiring modification by the columns of $\mathbf{K}$ is denoted by $\text{next}(\mathbf{K}, \mathbf{K})$. Using this notation, Figs. 13 and 14 display supernodal versions of the sequential and parallel column-Cholesky algorithm shown in Figs. 11 and 12, respectively.

```
for j = 1 to n do
    S_j := ∅
for J = 1 to N do
    for j ∈ J do
        for K ∈ S_j do
            cmod(j, K)
            i := next(j, K)
            S_i := S_i ∪ {K}
        cmod(j, J)
        cdiv(j)
    i := next(J, J)
    S_i := S_i ∪ {J}
```

FIG. 13. *Sequential sparse supernodal column-Cholesky factorization algorithm.*

```
Q := {Tcol(1), Tcol(2), ···, Tcol(n)}
for j = 1 to n do
    S_j := ∅
while Q ≠ ∅ do
    pop Tcol(j) from Q
    let J be the supernode containing column j
    while column j requires further cmod's do
        if S_j = ∅ do
            wait until S_j ≠ ∅
        obtain K from S_j
        i := next(j, K)
        lock
        S_i := S_i ∪ {K}
        unlock
        cmod(j, K)
    cmod(j, J)
    cdiv(j)
    if j is the last column of supernode J do
        i := next(J, J)
        lock
        S_i := S_i ∪ {J}
        unlock
```

FIG. 14. *Parallel sparse supernodal column-Cholesky factorization algorithm for shared-memory* MIMD *machines.*

Let $\sigma(L)$ denote the number of subscripts in the supernodal representation of the sparsity structure of L. The use of supernodes reduces the number of synchronization operations to a number proportional to $\sigma(L)$, which is often much less than $\eta(L)$, sometimes by as much as an order of magnitude [46].

3.4.2. Distributed fan-out algorithm. The algorithm introduced in [43], now known as the fan-out algorithm, was the first sparse Cholesky factorization algorithm

developed for distributed-memory machines. It is a parallel version of the submatrix-Cholesky factorization algorithm shown in Fig. 4. We will denote the kth task performed by the outer loop of the algorithm by $T\text{sub}(k)$, which is defined by

$$T\text{sub}(k) := \{\text{cdiv}(k)\} \cup \{\text{cmod}(j,k) \mid j \in \text{Struct}(L_{*k})\}.$$

That is, $T\text{sub}(k)$ first obtains L_{*k} by performing the $\text{cdiv}(k)$ operation, and then performs all column modifications that use the new column.

Algorithms for distributed-memory machines are usually structured around some prior distribution of the data to the processors. In order to keep the cost of interprocessor communication at acceptable levels, it is essential for the algorithm to make *local* use of *local* data as much as possible. The distributed fan-out, fan-in, and multifrontal algorithms are typical examples of this type of distributed algorithm (the fan-in and multifrontal algorithms will be discussed in the following subsections). These three distributed algorithms are all designed within the following framework.

- All three require assignment of the matrix columns to the processors.
- All three use the column assignment to distribute among the processors the tasks found in the outer loop of one of the serial implementations of sparse Cholesky factorization.

The differences among these algorithms stem from the various formulations of *serial* sparse Cholesky upon which they are based. The fan-in algorithm is based on column-Cholesky; it partitions each task $T\text{col}(j)$ among the processors. The distributed multifrontal algorithm partitions among the processors the tasks upon which the sequential multifrontal method is based: partial dense submatrix-Cholesky factorization and the assembly operations, both of which are introduced later in the subsection dealing with this algorithm. The fan-out algorithm is based on submatrix-Cholesky; it partitions each task $T\text{sub}(k)$ among the processors.

We now detail how the fan-out algorithm partitions the task $T\text{sub}(k)$ among the processors. Each column L_{*k} is stored on one and only one of P available processors. An n-vector map is required to record the distribution of columns to processors: if column k is stored on processor p, then $\text{map}[k] := p$. We let $\text{mycols}(p)$ denote the set of columns owned by processor p. The fan-out algorithm is a data-driven algorithm, where the data sent from one processor to another are the completed factor columns. The outer loop of the fan-out algorithm constantly checks the message queue for incoming columns. When it receives a column L_{*k}, it uses it to modify every column $j \in \text{mycols}(p)$ for which $\text{cmod}(j,k)$ is required. In other words, it performs the following set of cmods:

$$\{\text{cmod}(j,k) \mid j \in \text{Struct}(L_{*k}) \cap \text{mycols}(p)\}.$$

Indeed, each task $T\text{sub}(k)$ is partitioned among the processors by the partition defined by the column mapping. More precisely, the column partition

$$\{\text{mycols}(1), \text{mycols}(2), \cdots, \text{mycols}(P)\}$$

induces the partition of $T\text{sub}(k)$ into subtasks of the form

$$\{T\text{sub}(k,1), T\text{sub}(k,2), \cdots, T\text{sub}(k,P)\}$$

where

$$T\text{sub}(k,p) := \{\text{cmod}(j,k) \mid j \in \text{Struct}(L_{*k}) \cap \text{mycols}(p)\},$$

with each nonempty task $T\text{sub}(k, p)$ assigned to processor p, the owner of the columns updated by the task.

Of course, many of these tasks will be empty. Only the processors in the set

$$\text{procs}(L_{*k}) := \{\text{map}[j] \mid j \in \text{Struct}(L_{*k})\}$$

require column L_{*k}. When processor $p = \text{map}[j]$ has completed all column modifications required by column j, it then performs $\text{cdiv}(j)$ and sends it to every processor in $\text{procs}(L_{*j})$, where it eventually is used to modify later columns in the matrix. The algorithm is shown in Fig. 15.

```
for j ∈ mycols(p) do
    if j is a leaf node in T(A) do
        cdiv(j)
        send L_{*j} to the processors in procs(L_{*j})
        mycols(p) := mycols(p) − {j}
while mycols(p) ≠ ∅ do
    receive any column of L, say L_{*k}
    for j ∈ Struct(L_{*k}) ∩ mycols(p) do
        cmod(j, k)
        if column j required no more cmod's do
            cdiv(j)
            send L_{*j} to the processors in procs(L_{*j})
            mycols(p) := mycols(p) − {j}
```

FIG. 15. *Fan-out Cholesky factorization algorithm for processor p of a distributed-memory* MIMD *machine.*

Historically, the fan-out algorithm was first to be implemented on a distributed-memory machine, but due to several weaknesses it has since been superseded by fan-in algorithms and distributed multifrontal algorithms. The distributed fan-out algorithm incurs greater interprocessor communication costs than the other two methods, both in terms of total number of messages and total message volume. It simply does not exploit a good communication-reducing column mapping, such as the subtree-to-subcube mapping, as effectively as the other methods do. Ashcraft et al. [9] and Zmijewski [109] have independently improved the algorithm by having it send aggregated update columns rather than individual factor columns for columns belonging to a subtree that has been mapped to a single processor. Though the resulting improvement in performance is substantial, it still is insufficient to make the method competitive.

Another problem with the method is the expense of mapping the entries of the updating column k to the corresponding entries of the updated column j when performing $\text{cmod}(j, k)$. The set $\text{Struct}(L_{*k})$ must accompany the factor column L_{*k} when it is sent to other processors to enable these processors to complete column modifications of the form $\text{cmod}(j, k)$. This roughly doubles the communication volume and creates a more complicated message that must be packed by the sending processor and unpacked by the receiving processor. Moreover, each $\text{cmod}(j, k)$ requires that both index sets $\text{Struct}(L_{*j})$ and $\text{Struct}(L_{*k})$ be searched in order to match indices. This results in poor *serial* efficiency. These weaknesses have provoked efforts to develop better distributed factorization algorithms.

3.4.3. Distributed fan-in algorithm. One of the improved distributed factorization algorithms is the fan-in algorithm, introduced by Ashcraft, Eisenstat, and Liu

in [10]. Based on the sparse column-Cholesky algorithm, it distributes each column task $T\text{col}(j)$ among the processors in a manner similar to the distribution of tasks $T\text{sub}(k)$ in the fan-out algorithm. Viewed in a more general way, the fan-in method is analogous to the standard parallel algorithm for a dot product, in which each processor first *locally* reduces the data *assigned to it* down to a single number, and then participates in a *global* phase during which the processors cooperate in reducing down to a single number the P local reductions generated during the preceding "perfectly parallel" phase. Indeed, the name "fan-in" is taken from the fan-in distributed algorithm for dense triangular solution [58], which computes a series of inner product calculations in precisely this manner. Note that throughout this subsection we freely use the notation introduced in the previous subsection.

As with the fan-out algorithm, each processor p is responsible for computing $\text{cdiv}(j)$ for every column $j \in \text{mycols}(p)$. Of course, $\text{cdiv}(j)$ cannot be computed until all modifications $\text{cmod}(j,k)$, $k \in \text{Struct}(L_{j*})$, have been performed. The fan-in algorithm is a demand-driven algorithm, where the data required are aggregated update columns computed by the sending processor *using columns it owns*, and needed by the receiving processor to update a target column. Let $u(j,k)$ denote the scaled column accumulated into the factor column by the $\text{cmod}(j,k)$ operation. The outer loop of the algorithm processes every column j of the matrix in ascending order by column number. When processor p processes column j, it aggregates into a single update vector u every update vector $u(j,k)$ for which $k \in \text{mycols}(p) \cap \text{Struct}(L_{j*})$. Indeed, each task $T\text{col}(j)$ is partitioned among the processors by the partition of the columns induced by the column mapping. More precisely, the column partition

$$\{\text{mycols}(1), \text{mycols}(2), \cdots, \text{mycols}(P)\}$$

induces the partition of $T\text{col}(j)$ into subtasks of the form

$$\{T\text{col}(j,1), T\text{col}(j,2), \cdots, T\text{col}(j,P)\}$$

where $T\text{col}(j,p)$ aggregates into a single update vector every update vector $u(j,k)$ for which $k \in \text{Struct}(L_{j*}) \cap \text{mycols}(p)$, with each *nonnull* task $T\text{col}(j,p)$ assigned to processor p, the owner of the updating columns used by the task.

After performing $T\text{col}(j,p)$, if processor p does *not* own column j, then it sends the resulting aggregated update column to processor $q = \text{map}[j]$, which will eventually incorporate it into column j. If, on the other hand, processor p does own column j, it must receive and process any aggregated update columns required by column j from other processors before it can complete the $\text{cdiv}(j)$ operation. The fan-in algorithm is given in Fig. 16.

It is interesting to note that any column $j \in \text{mycols}(p)$ will receive an aggregated update column from every processor in the set

$$\text{procs}(L_{j*}) := \{\text{map}[k] \mid k \in \text{Struct}(L_{j*})\}.$$

In contrast, the fan-out algorithm sent the factor column L_{*j} to every processor in the processor set $\text{procs}(L_{*j})$. Consider the communication costs incurred by the two algorithms during the computation of columns that constitute a subtree of the elimination tree that has been mapped to a single processor by a subcube-to-subtree mapping. For the fan-in algorithm there will be *no communication* during this portion of the computation, because for every column j in the subtree, $\text{Struct}(L_{j*})$ also belongs to the subtree. On the other hand, the fan-out algorithm must send L_{*j} to another

```
for j := 1 to n do
    if j ∈ mycols(p) or Struct(L_{j*}) ∩ mycols(p) ≠ ∅ do
        u := 0
        for k ∈ Struct(L_{j*}) ∩ mycols(p) do
            u := u + u(j, k)
        if map[j] ≠ p do
            send u to processor q = map[j]
        else
            incorporate u into the factor column j
            while any aggregated update column for column j remains unreceived do
                receive in u another aggregated update column for column j
                incorporate u into the factor column j
            cdiv(j)
```

FIG. 16. *Fan-in sparse Cholesky factorization algorithm for processor p of a distributed-memory* MIMD *machine.*

processor if there is a column index $k \in \text{Struct}(L_{*j})$ for some column j in the subtree, such that $\text{map}[k] \neq \text{map}[j]$. This observation is an informal indication of why the fan-in algorithm is better than the fan-out algorithm at exploiting a good mapping to reduce interprocessor communication.

A more visual comparison of the communication patterns of the fan-out and fan-in algorithms is given in Figs. 17 and 18. These figures illustrate snapshots of the execution of the two algorithms on an Intel iPSC/2 hypercube, with time on the horizontal axis. Processor activity is shown by horizontal lines and interprocessor communication by slanted lines. The horizontal line corresponding to each processor is either solid or blank, depending on whether the processor is busy or idle, respectively. Each message sent between processors is shown by a line drawn from the sending processor at the time of transmission to the receiving processor at the time of reception of the message. The problem being solved is the factorization of a matrix of order 225 derived from a model finite element problem on a 15×15 grid, using a nested dissection ordering and subtree-to-subcube mapping on eight processors. The divide-and-conquer nature of the nested dissection ordering is clearly visible in Fig. 18, which also illustrates the ability of the fan-in algorithm, given an appropriate mapping, to exploit this structure to reduce communication. By contrast, the fan-out algorithm shown in Fig. 17 exhibits much greater communication traffic as well as a less regular communication pattern, even under the ideal conditions represented here. These diagrams were produced using a package developed at Oak Ridge National Laboratory for visualizing the behavior of parallel algorithms [57].

Compute-ahead fan-in algorithm. In Fig. 16, observe that processor p will fall idle if, while receiving aggregated update columns destined for a column $j \in \text{mycols}(p)$, it has no such updates in its message queue. One straightforward enhancement to the method is to probe the queue for such messages, and when there are none, proceed with useful work on later factor columns. When unable to complete the current column j, the algorithm toggles between performing so-called *compute-ahead* tasks on columns $i > j$, and detecting and processing incoming aggregated updates for the current column j.

There are two types of compute-ahead tasks to be performed on later columns of the factor:

1. For some column $i > j$, aggregate into a work vector the update vector $u(i, k)$ for each completed column $k \in \text{Struct}(L_{i*}) \cap \text{mycols}(p)$.

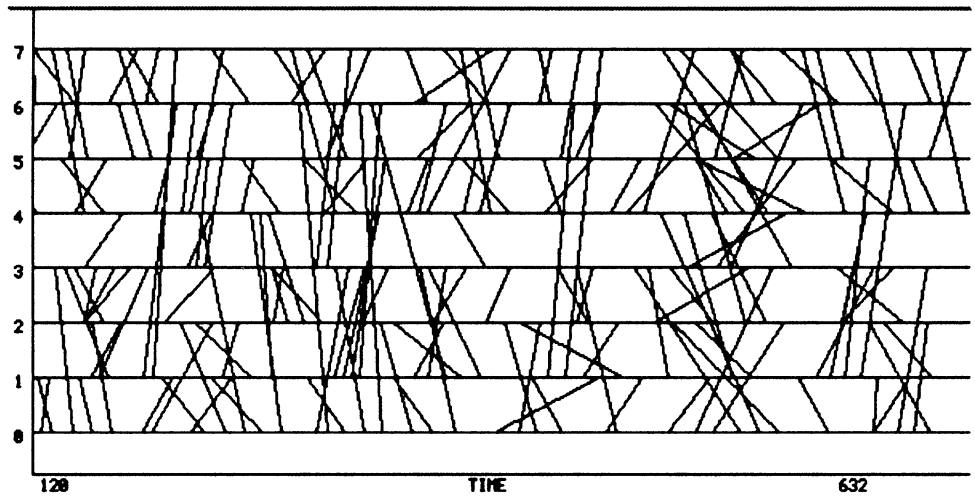

FIG. 17. *Communication pattern of fan-out algorithm for a model problem.*

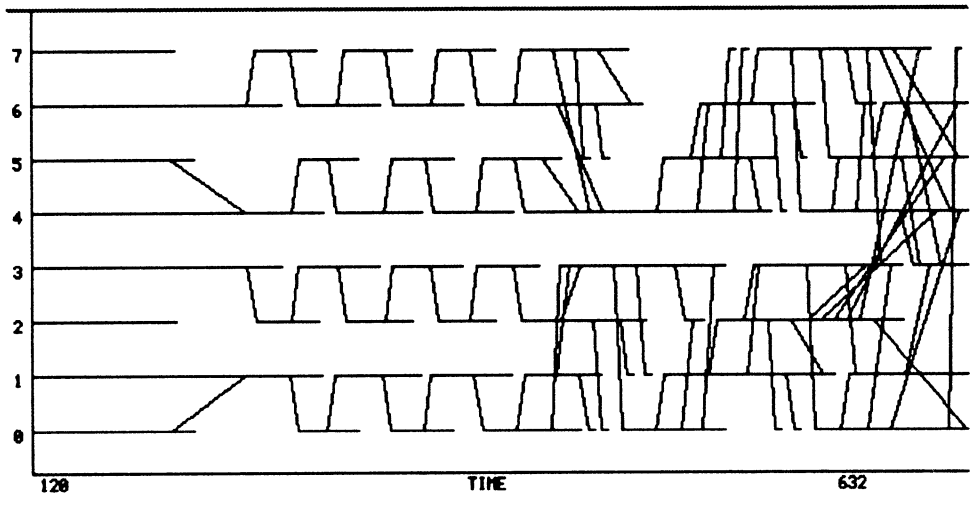

FIG. 18. *Communication pattern of fan-in algorithm for a model problem.*

2. Receive an aggregated update column for some column $i > j$, and incorporate it into the factor column.

Compute-ahead tasks of the first type have priority over compute-ahead tasks of the second type; that is, compute-ahead tasks of the second type are performed only when the algorithm has exhausted its supply of tasks of the first type.

Compute-ahead aggregating of update columns is limited to target columns $i > j$ that belong to the same supernode as the current column j. This is due primarily to the ease and "naturalness" with which successive aggregate update columns sharing the same sparsity pattern can be computed. Since the aggregated update columns are managed so that they share the same sparsity structure as the target column, no indirect indexing is required to incorporate them into the factor column. Thus, compute-ahead tasks of the second type require merely a receive, followed by a `saxpy`.

For details concerning these and other implementation issues, consult [8].

Though supernodes play an important role in organizing the compute-ahead fan-in algorithm, current implementations of both the basic and compute-ahead fan-in algorithms do not exploit supernodes to reduce memory traffic in the inner loops of the computation—one of their key roles in the parallel shared-memory column-Cholesky algorithm. There is no reason why supernodes cannot serve in this role in the fan-in algorithm also. However, it is interesting to note that the potential exploitation of supernodes in distributed-memory algorithms is somewhat limited because good mappings typically distribute the columns of a supernode among several processors.

3.4.4. Parallel multifrontal algorithms. As noted earlier, multifrontal methods are generalizations of single-front methods. The original motivation for developing frontal methods was for more effective use of auxiliary storage in the out-of-core solution of sparse systems, and more efficient inner-loop computations by avoiding the indirect addressing that is characteristic of general sparse data structures. The fundamental idea in frontal methods is to keep only a relatively small portion of the matrix in main memory at any given time, and to use a full matrix representation for this "active" portion of the matrix, so that computations involving it are more efficient on scalar machines and more readily vectorized on vector machines. Although the data structure for the active matrix is very simple, the overall data management required in frontal methods is quite complicated, involving the assembly of matrix elements, their insertion into the proper location in the full active matrix data structure, and the writing of completed portions of the factor to disk, all of which must account for the fact that the active matrix constitutes a moving "window" through the problem.

The success of frontal methods is dependent on keeping the size of the active matrix small, which in turn depends on the structure of the problem and the ordering used in solving it. In structural analysis, for example, a long thin truss is ideal for a frontal solution technique in that, with an appropriate ordering, a single narrow "front" passes along the length of the truss. If a single front should become unacceptably large, however, then multiple fronts could be employed, leading to multifrontal methods. Of course, the various fronts must eventually merge before the problem can be completed, but the hope is that with an appropriate ordering such mergers can be postponed as late as possible in the computation. The use of multiple fronts seems to suggest an obvious parallel implementation: simply assign a separate front to each processor. As we shall see, however, the situation is not quite so straightforward.

A self-contained presentation of parallel multifrontal algorithms would occupy more space than we can afford in an article of this scope. The difficulties in producing a brief but clear description stem primarily from the complexity of the method: a *sequential* multifrontal code is considerably more complicated than a *sequential* sparse column-Cholesky code. As might be expected, modifying the method to run on MIMD machines is also more difficult and complicated, though it is by no means unmanageable; there have been implementations on both shared-memory [12], [22], [23], [106] and distributed-memory [9], [36], [82] machines. This section is limited to a brief overview of the literature on the subject and a short discussion of some of the problems that arise in parallel implementations. The reader should consult [28] or [78] for background material on multifrontal methods.

We should also point out that some of the codes and algorithms cited in this section are designed for nonsymmetric linear systems, and at least one includes pivoting for stability. For instance, the work in [22] and [23] is based on the Harwell MA37 code, which solves nonsymmetric systems and pivots for stability. Nevertheless, such

codes can be discussed within the framework of this article because they perform a symbolic factorization of the structurally symmetric matrix $A + A^T$, and compute a structurally symmetric numerical factorization of A within the resulting data structure. Therefore, much of the material in [22] and [23] is directly applicable to sparse multifrontal Cholesky factorization.

Background. As noted in §2, the multifrontal method is a sophisticated variant of the sparse submatrix-Cholesky factorization algorithm (Fig. 4) for which the cmod(j, k) operations are not applied directly to column j of the factor matrix. Instead, each is accumulated and passed on through a succession of update matrices until it is finally incorporated into the target column. The outer loop of the serial multifrontal algorithm processes the supernodes $\mathbf{1}, \mathbf{2}, \cdots, \mathbf{N}$ in order, completing the columns of each supernode when the supernode is processed. The order in which the supernodes are processed is critical. For reasons discussed below, they are processed in the order in which they are visited by a postorder traversal of a *supernodal elimination tree*. A supernodal elimination tree with 31 supernodes is displayed in Fig. 6.

Every supernode $\mathbf{K}$ has associated with it a frontal matrix in which the factor and update columns associated with the supernode are computed. The factor and update columns computed within this matrix are stored in a dense matrix format, essentially minimizing the use of indirect addressing—one of the major strengths of the method. The algorithm performs two tasks within this frontal matrix:

1. The *assembly* step inserts the required data into the frontal matrix.
2. After the assembly step, *dense partial submatrix-Cholesky factorization* within the frontal matrix generates the factor and update columns.

We discuss first the partial submatrix-Cholesky factorization step and then the assembly step in more detail.

Suppose that $\mathbf{K}$ contains r columns of the matrix, and assume that the assembly step for supernode $\mathbf{K}$'s frontal matrix has been completed. The algorithm then computes r major steps of *dense submatrix-Cholesky* factorization within the frontal matrix, after which the first r columns of the frontal matrix contain the r factor columns of $\mathbf{K}$, and the trailing columns in the frontal matrix contain aggregated update columns for later columns of the matrix. These trailing columns constitute the *update* matrix generated by this block elimination step. Henceforth, we will denote this task by Tsub($\mathbf{K}$). The update matrix is stored and assembled later into the frontal matrix of its "parent supernode" in the elimination tree.

The *assembly* step consists of the following three steps:

1. Zero out the frontal matrix.
2. Insert the required entries of A into the appropriate locations of the matrix.
3. For each "child supernode," obtain its associated update matrix and add each entry to its corresponding entry in the frontal matrix.

Because the supernodes are ordered by a postorder traversal of the elimination tree, the update matrices can be stored efficiently on a stack, limiting both the storage and time required to store them. New update matrices are pushed onto the stack as soon as they are generated, while update matrices for child supernodes are popped off the stack as needed during each assembly step.

Shared-memory MIMD *machines.* One key problem associated with parallel multifrontal algorithms for shared-memory MIMD multiprocessors is the management of auxiliary storage for the update matrices. The postordering of supernodes used in the sequential algorithm severely limits the parallelism available; in particular, it

limits exploitation of the parallelism that exists among the many disjoint subtrees of the elimination tree available in most realistic problems. To create more independent processes, algorithm developers have abandoned the postordering and the stack of update matrices. Instead, they process the supernodes in some order that allows greater exploitation of the large-grained (subtree-level) parallelism, but which complicates management of the working storage for update matrices, increasing both the storage and time required by this part of the algorithm [23], [83], [106].

In [22] and [23], Duff considered several strategies for dealing with the resulting fragmentation of working storage. Garbage collection to reclaim unused storage requires a critical section that seriously inhibits parallelism. Subdividing the working storage in an effort to localize the garbage collection operations and reduce their negative effect on parallelism proved to be too complicated and ineffective [22]. Breaking up individual update matrices to make better use of free storage was not considered because it would destroy the data locality vital for efficient use of cache—one of the important strengths of the multifrontal method and a key consideration on the Alliant FX/8 [23]. In [23], Duff used the *buddy system* to manage the storage for update matrices. For any given update matrix, the buddy system obtains a free block of storage of length 2^k, where k is the smallest power of two that provides enough contiguous storage locations to hold the matrix. The scheme is guaranteed to waste no more than half the working storage.

We are aware of two other parallel multifrontal codes designed to run on parallel shared-memory MIMD machines. A parallel multifrontal code developed by Lucas [83] for the CRAY 2 allocates subtrees to individual processors and has each processor manage a local stack for its assigned subtree. During the course of the computation, there are eventually more processors than independent subtrees. At that point, the code abandons the use of subtree-level parallelism. Instead, it successively processes the remaining tasks $T\text{sub}(\mathbf{K})$, using CRAY autotasking to partition each task $T\text{sub}(\mathbf{K})$ among *all* the processors. A parallel multifrontal code developed by Vu [106] for the CRAY Y-MP uses a similar strategy.

A second issue discussed in [23] is partitioning the work among the processors for execution in parallel. Here, we restrict our attention to issues associated with distributing the tasks $T\text{sub}(\mathbf{1})$, $T\text{sub}(\mathbf{2})$, $\cdots$, $T\text{sub}(\mathbf{N})$ among the processors. The situation is not as simple as it is for parallel column-Cholesky, where simply dealing out the column tasks $T\text{col}(j)$, with some care in the scheduling, is very effective in exploiting both subtree- and column-level parallelism (see §3.4.1). If the multifrontal method distributes indivisible tasks $T\text{sub}(\mathbf{K})$ among the processors in a similar fashion, then, as noted in [22] and [26], parallelism decreases as the computation proceeds toward the root supernode and disappears altogether when the root supernode is reached. Typically, most of the work is performed in the larger frontal matrices associated with supernodes near the root, and thus smaller granularity is required for acceptable performance. That is, the tasks $T\text{sub}(\mathbf{K})$ for supernodes $\mathbf{K}$ near the root supernode must be partitioned into smaller tasks and distributed among the processors. In [23], Duff parameterizes his code so that it can spawn tasks of any granularity between two extremes, the largest being the tasks $T\text{sub}(\mathbf{K})$, and the smallest being individual cmods and cdivs. His results indicate that working with small blocks of columns is most effective. Near the root of the supernodal elimination tree, the algorithms of Lucas [83] and Vu [106] use the autotasking capabilities of their target machines, the CRAY 2 and CRAY Y-MP, to partition the tasks $T\text{sub}(\mathbf{K})$ among the processors.

Distributed-memory MIMD *machines.* Lucas, with Blank and Tieman, [82], [84] developed the first implementation of the multifrontal method for distributed-memory MIMD machines. Since then, Ashcraft [9] has also developed parallel multifrontal codes for such machines. Lucas's code and the first code developed by Ashcraft implement essentially the same distributed multifrontal algorithm [6], [83]. This section contains a brief discussion of a few features of this algorithm. Further enhancements to the algorithm, and a systematic comparison of all the distributed-memory factorization algorithms will appear in [7].

As with other distributed factorization algorithms, each column k of the matrix is assigned to and stored on one processor, map[k]. Consider a supernode $\mathbf{K}$ and let map($\mathbf{K}$) denote the set of processors that own at least one column of $\mathbf{K}$ or a descendant of a column $\mathbf{K}$ in the elimination tree. The key feature of the algorithm is the distribution of *all* the columns of $\mathbf{K}$'s frontal matrix among the processors in map($\mathbf{K}$); that is, both the factor columns and the aggregated update columns generated by the task Tsub($\mathbf{K}$) are distributed among the processors in map($\mathbf{K}$).

The processors in map($\mathbf{K}$) work together to perform the task Tsub($\mathbf{K}$), i.e., dense submatrix-Cholesky factorization on the first $|\mathbf{K}|$ columns of the distributed frontal matrix. The algorithm used to perform this task can be viewed as a straightforward adaptation of the parallel dense submatrix-Cholesky algorithm presented in [37]. When processor $p = \text{map}[k] \in \text{map}(\mathbf{K})$ completes factor column $k \in \mathbf{K}$, it broadcasts L_{*k} to the other processors in map($\mathbf{K}$). The other processors in map($\mathbf{K}$) at some point receive L_{*k} and use it to modify every column of the frontal matrix that they own. Thus, this phase of the algorithm is very similar to the fan-out algorithm shown in Fig. 15.

Before the task Tsub($\mathbf{K}$) can be performed, supernode $\mathbf{K}$'s distributed frontal matrix must be assembled. Contributions from distributed update matrices for any children of $\mathbf{K}$ must be sent to the appropriate processors and scatter-added into the appropriate frontal matrix locations. More precisely, if an update column from a "child" update matrix is located on a different processor than the *corresponding* column of its "parent" frontal matrix, then the aggregated update column must be sent to its "new owner," where it is incorporated into the appropriate column of the frontal matrix.

Both phases of the factorization require interprocessor communication. The factorization phase performs a restricted broadcast of completed factor columns, while the assembly phase moves aggregated update columns from one processor to another. The two forms of communication result in somewhat higher communication cost for the multifrontal algorithm than that incurred by the fan-in algorithm. However, its extra communication overhead is far smaller than that incurred by the pure fan-out algorithm, and preliminary results indicate similar performance for the fan-in and distributed multifrontal algorithms [9].

3.5. Triangular solution. Unfortunately, there is relatively little to say about parallel algorithms for forward and backward triangular solutions. Data dependencies and a paucity of work to distribute among the processors make it very difficult to achieve high computational rates, even for dense problems. Heath and Romine [58] and Eisenstat et al. [30] have shown that intricate pipelining techniques are required to achieve computational rates as high as 50% efficiency for *large dense problems* on distributed-memory hypercube multiprocessors. Two factors make the situation even more difficult in the sparse case. First, due to preservation of sparsity in the factor matrix, there is usually far less work to distribute among the processors—

approximately $\eta(L)$ flops rather than the $n(n-1)/2$ flops available in the dense case. Second, the successful pipelining techniques used in [30], [58] appear to require the extremely regular structure of a dense matrix. Loss of this regularity in sparse Cholesky factors increases the difficulty of using these complicated techniques to speed up sparse triangular solution. Generalizing these techniques so that they can be incorporated into a parallel sparse triangular solution algorithm is a possible avenue for future improvement. A step in this direction has been made by Zmijewski [108], who considered the use of cyclic algorithms for solving sparse triangular systems.

These difficulties are mitigated somewhat by the subtree-level parallelism that is available only in the sparse case. Though the parallel algorithms for sparse forward and back triangular solutions contained in [44] exploit this parallelism, they nonetheless performed rather poorly. Other work on parallel sparse triangular solution algorithms [4], [56], [85], [102] has been directed primarily toward use in the preconditioned conjugate gradient algorithm. Some of the work in [4], however, is applicable to complete, as well as incomplete, Cholesky factorizations.

4. Concluding remarks. In this paper, we have provided a summary of parallel algorithms currently available for the four phases in the solution of sparse symmetric positive definite systems. It is clear from the relative length of the discussions that much of this research has been focused on the design and implementation of parallel numerical factorization algorithms. Some of these algorithms have exhibited reasonable speed-up ratios, particularly on shared-memory MIMD multiprocessors. Although there have been attempts to develop parallel algorithms for the other phases, namely, ordering, symbolic factorization, and triangular solutions, these algorithms have generally been less successful and lacking in efficiency. Much research is needed in those areas. The ordering problem seems particularly problematic in a distributed-memory environment because of the difficulty of partitioning the graph of the matrix among the processors in an intelligent way *before* the ordering is determined.

It may be argued that current sequential algorithms for symbolic factorization and triangular solution are so efficient that perhaps they can be used on one processor in a multiprocessor environment instead of developing parallel versions. This may be true for MIMD multiprocessors with globally shared memory. On MIMD multiprocessors with local memory, there are at least two reasons why parallel algorithms are needed for symbolic factorization and triangular solution, even if these algorithms may be less efficient than their sequential counterparts. First, although symbolic factorization and triangular solution are often the least expensive phases in the solution process on serial machines, they may become the dominant phases as more efficient parallel numerical factorization algorithms are developed. Thus, research on the design of efficient parallel algorithms for symbolic factorization and triangular solution will be necessary eventually. Second, even if they are somewhat inefficient, parallel algorithms are still needed to make use of the large (collectively) local memory available on distributed-memory parallel machines for solving large problems; there may not be enough memory on a single processor to carry out symbolic factorization and/or triangular solution serially. Third, many algorithms require multiple triangular solutions.

Our emphasis in this paper has been on parallel direct methods for solving sparse symmetric positive definite systems. Work has also been done on parallel algorithms for other matrix computations. In the case of direct methods for solving sparse nonsymmetric linear systems, much of the research has been carried out on shared-memory MIMD multiprocessors. Some recent examples can be found in [1]–[3], [20],

[22], [23], [51], [52]. Parallel algorithms for sparse least squares problems are discussed in [16], [59], [96]. There has been a great deal of research on parallel iterative methods for solving large sparse linear systems as well. For a summary of such work and references to this extensive literature, see the book by Ortega [89]. Many additional references on all aspects of parallel matrix computations can be found in [90].

Acknowledgments. The authors thank Eduardo D'Azevedo, Alan George, and Joseph Liu for their suggestions and comments, which have improved the presentation of the material.

REFERENCES

[1] G. ALAGHBAND, *Parallel pivoting combined with parallel reduction and fill-in control*, Parallel Comput., 11 (1989), pp. 201–221.

[2] G. ALAGHBAND AND H. JORDAN, *Multiprocessor sparse L/U decomposition with controlled fill-in*, Tech. Report 85-48, ICASE, NASA Langley Research Center, Hampton, VA, 1985.

[3] P. AMESTOY AND I. DUFF, *Vectorization of a multiprocessor multifrontal code*, Internat. J. Supercomp. Appl., 3 (1989), pp. 41–59.

[4] E. ANDERSON AND Y. SAAD, *Solving sparse triangular linear systems on parallel computers*, Internat. J. High Speed Comput., 1 (1989), pp. 73–95.

[5] C. ASHCRAFT, *A vector implementation of the multifrontal method for large sparse, symmetric positive definite linear systems*, Tech. Report ETA-TR-51, Engineering Technology Applications Division, Boeing Computer Services, Seattle, WA, 1987.

[6] ———, Personal communication, 1990.

[7] ———, *The aggregate model for the factorization of symmetric positive definite matrices*, Ph.D. thesis, Dept. of Computer Science, Yale University, New Haven, CT, 1990.

[8] C. ASHCRAFT, S. EISENSTAT, J. LIU, B. PEYTON, AND A. SHERMAN, *A compute-ahead implementation of the fan-in sparse distributed factorization scheme*, Tech. Report ORNL/TM-11496, Oak Ridge National Laboratory, Oak Ridge, TN, 1990.

[9] C. ASHCRAFT, S. EISENSTAT, J. LIU, AND A. SHERMAN, *A comparison of three column-based distributed sparse factorization schemes*, Tech. Report YALEU/DCS/RR-810, Dept. of Computer Science,Yale University, New Haven, CT, 1990.

[10] C. ASHCRAFT, S. EISENSTAT, AND J. W.-H. LIU, *A fan-in algorithm for distributed sparse numerical factorization*, SIAM J. Sci. Statist. Comput., 11 (1990), pp. 593–599.

[11] C. ASHCRAFT, R. GRIMES, J. LEWIS, B. PEYTON, AND H. SIMON, *Progress in sparse matrix methods for large linear systems on vector supercomputers*, Internat. J. Supercomp. Appl., 1 (1987), pp. 10–30.

[12] R. BENNER, G. MONTRY, AND G. WEIGAND, *Concurrent multifrontal methods: shared memory, cache, and frontwidth issues*, Internat. J. Supercomp. Appl., 1 (1987), pp. 26–44.

[13] P. BERMAN AND G. SCHNITGER, *On the performance of the minimum degree ordering for Gaussian elimination*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 83–88.

[14] J. BROWNE, J. DONGARRA, A. KARP, K. KENNEDY, AND D. KUCK, 1988 *Gordon Bell prize*, IEEE Software, 6 (May 1989), pp. 78–85.

[15] I. CAVERS, *Tiebreaking the minimum degree algorithm for ordering sparse symmetric positive definite matrices*, Master's thesis, Dept. of Computer Science, University of British Columbia, Vancouver, B.C., 1987.

[16] E. CHU AND A. GEORGE, *Sparse orthogonal decomposition on a hypercube multiprocessor*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 453–465.

[17] E. CHU, A. GEORGE, J. W.-H. LIU, AND E. G.-Y. NG, *User's guide for* SPARSPAK-A*: Waterloo sparse linear equations package*, Tech. Report CS-84-36, University of Waterloo, Waterloo, Ontario, 1984.

[18] J. CONROY, *Parallel direct solution of sparse linear systems of equations*, Tech. Report TR 1714, Dept. of Computer Science, University of Maryland, College Park, MD, 1986.

[19] A. DAVE AND I. DUFF, *Sparse matrix calculations on the Cray-2*, Parallel Comput., 5 (1987), pp. 55–64.

[20] T. DAVIS AND P. YEW, *A nondeterministic parallel algorithm for general unsymmetric sparse LU factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 383–402.

[21] J. DONGARRA, F. GUSTAVSON, AND A. KARP, *Implementing linear algebra algorithms for dense matrices on a vector pipeline machine*, SIAM Rev., 26 (1984), pp. 91–112.

[22] I. DUFF, *Parallel implementation of multifrontal schemes*, Parallel Comput., 3 (1986), pp. 193–204.
[23] ———, *Multiprocessing a sparse matrix code on the Alliant FX/8*, J. Comput. Appl. Math., 27 (1989), pp. 229–239.
[24] I. DUFF, A. ERISMAN, AND J. REID, *On George's nested dissection method*, SIAM J. Numer. Anal., 13 (1976), pp. 686–695.
[25] I. DUFF, A. ERISMAN, AND J. K. REID, *Direct Methods for Sparse Matrices*, Oxford University Press, Oxford, U.K., 1987.
[26] I. DUFF, N. GOULD, M. LESCRENIER, AND J. K. REID, *The multifrontal method in a parallel environment*, in Advances in Numerical Computation, M. Cox and S. Hammarling, eds., Oxford University Press, Oxford, U.K., 1990.
[27] I. DUFF AND J. REID, MA27 - *a set of Fortran subroutines for solving sparse symmetric sets of linear equations*, Tech. Report AERE R 10533, Harwell, 1982.
[28] ———, *The multifrontal solution of indefinite sparse symmetric linear equations*, ACM Trans. Math. Software, 9 (1983), pp. 302–325.
[29] S. EISENSTAT, M. GURSKY, M. SCHULTZ, AND A. H. SHERMAN, *The Yale sparse matrix package I. the symmetric codes*, Internat. J. Numer. Methods Engrg., 18 (1982), pp. 1145–1151.
[30] S. EISENSTAT, M. HEATH, C. HENKEL, AND C. ROMINE, *Modified cyclic algorithms for solving triangular systems on distributed-memory multiprocessors*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 589–600.
[31] C. FIDUCCIA AND R. MATTHEYSES, *A linear-time heuristic for improving network partitions*, in Proceedings of the 19th Design Automation Conference, 1982, pp. 175–181.
[32] M. FIEDLER, *Algebraic connectivity of graphs*, Czech. Math. J., 23 (1973), pp. 298–305.
[33] ———, *A property of eigenvectors of non-negative symmetric matrices and its application to graph theory*, Czech. Math. J., 25 (1975), pp. 619–633.
[34] K. GALLIVAN, R. PLEMMONS, AND A. SAMEH, *Parallel algorithms for dense linear algebra computations*, SIAM Rev., 32 (1990), pp. 54–135.
[35] F. GAO AND B. PARLETT, *Communication cost of sparse Cholesky factorization on a hypercube*, Tech. Report PAM-436, Center for Pure and Applied Mathematics, University of California, Berkeley, CA, 1988.
[36] G. GEIST, *Solving finite element problems with parallel multifrontal schemes*, in Hypercube Multiprocessors 1987, M. T. Heath, ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1987, pp. 656–661.
[37] G. GEIST AND M. HEATH, *Parallel Cholesky factorization on a hypercube multiprocessor*, Tech. Report ORNL-6211, Oak Ridge National Laboratory, Oak Ridge, TN, 1985.
[38] G. GEIST AND E. G.-Y. NG, *Task scheduling for parallel sparse Cholesky factorization*, Internat. J. Parallel Programming, 18 (1989), pp. 291–314.
[39] A. GEORGE, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363.
[40] A. GEORGE, M. HEATH, AND J. W.-H. LIU, *Parallel Cholesky factorization on a shared-memory multiprocessor*, Linear Algebra Appl., 77 (1986), pp. 165–187.
[41] A. GEORGE, M. HEATH, J. W.-H. LIU, AND E. G.-Y. NG, *Solution of sparse positive definite systems on a shared memory multiprocessor*, Internat. J. Parallel Programming, 15 (1986), pp. 309–325.
[42] ———, *Symbolic Cholesky factorization on a local-memory multiprocessor*, Parallel Comput., 5 (1987), pp. 85–95.
[43] ———, *Sparse Cholesky factorization on a local-memory multiprocessor*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 327–340.
[44] ———, *Solution of sparse positive definite systems on a hypercube*, J. Comput. Appl. Math., 27 (1989), pp. 129–156.
[45] A. GEORGE AND J. W.-H. LIU, *An automatic nested dissection algorithm for irregular finite element problems*, SIAM J. Numer. Anal., 15 (1978), pp. 1053–1069.
[46] ———, *An optimal algorithm for symbolic factorization of symmetric matrices*, SIAM J. Comput., 9 (1980), pp. 583–593.
[47] ———, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice–Hall, Englewood Cliffs, NJ, 1981.
[48] ———, *The evolution of the minimum degree ordering algorithm*, SIAM Rev., 31 (1989), pp. 1–19.
[49] A. GEORGE, J. W.-H. LIU, AND E. G.-Y. NG, *Communication results for parallel sparse Cholesky factorization on a hypercube*, Parallel Comput., 10 (1989), pp. 287–298.
[50] A. GEORGE AND D. MCINTYRE, *On the application of the minimum degree algorithm to finite element systems*, SIAM J. Numer. Anal., 15 (1978), pp. 90–111.

[51] A. GEORGE AND E. G.-Y. NG, *Parallel sparse Gaussian elimination with partial pivoting*, Ann. Oper. Res., 22 (1990), pp. 219–240.

[52] J. GILBERT, *An efficient parallel sparse partial pivoting algorithm*, Tech. Report CMI No. 88/45052-1, Centre for Computer Science, Dept. of Science and Technology, Chr. Michelsen Institute, Bergen, Norway, 1988.

[53] J. GILBERT AND H. HAFSTEINSSON, *Parallel symbolic factorization of sparse linear systems*, Parallel Comput., 14 (1990), pp. 151–162.

[54] J. GILBERT AND R. SCHREIBER, *Highly parallel sparse Cholesky factorization*, Tech. Report CSL-90-7, Xerox Palo Alto Research Center, 1990; SIAM J. Sci. Statist. Comput., submitted.

[55] J. GILBERT AND E. ZMIJEWSKI, *A parallel graph partitioning algorithm for a message-passing multiprocessor*, Internat. J. Parallel Programming, 16 (1987), pp. 427–449.

[56] A. GREENBAUM, *Solving sparse triangular linear systems using fortran with extensions on the NYU Ultracomputer prototype*, Tech. Report 99, NYU Ultracomputer Note, New York University, New York, April 1986.

[57] M. HEATH, *Visual animation of parallel algorithms for matrix computations*, in Proc. Fifth Distributed Memory Computing Conf., Charleston, SC, 1990.

[58] M. HEATH AND C. ROMINE, *Parallel solution of triangular systems on distributed-memory multiprocessors*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 558–588.

[59] M. HEATH AND D. SORENSEN, *A pipelined Givens method for computing the QR factorization of a sparse matrix*, Linear Algebra Appl., 77 (1986), pp. 189–203.

[60] A. HOFFMAN, M. MARTIN, AND D. ROSE, *Complexity bounds for regular finite difference and finite element grids*, SIAM J. Numer. Anal., 10 (1973), pp. 364–369.

[61] B. IRONS, *A frontal solution program for finite element analysis*, Internat. J. Numer. Methods Engrg., 2 (1970), pp. 5–32.

[62] J. JESS AND H. KEES, *A data structure for parallel L/U decomposition*, IEEE Trans. Comput., C-31 (1982), pp. 231–239.

[63] B. KERNIGHAN AND S. LIN, *An efficient heuristic procedure for partitioning graphs*, Bell Systems Tech. J., 49 (1970), pp. 291–307.

[64] C. LAWSON, R. HANSON, D. KINCAID, AND F. KROGH, *Basic linear algebra subprograms for Fortran usage*, ACM Trans. Math. Software, 5 (1979), pp. 308–371.

[65] C. LEISERSON AND J. LEWIS, *Orderings for parallel sparse symmetric factorization*, in Parallel Processing for Scientific Computing, G. Rodrigue, ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1989, pp. 27–32.

[66] M. LEUZE, *Independent set orderings for parallel matrix factorization by Gaussian elimination*, Parallel Comput., 10 (1989), pp. 177–191.

[67] J. LEWIS, B. PEYTON, AND A. POTHEN, *A fast algorithm for reordering sparse matrices for parallel factorization*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 1156–1173.

[68] R. LIPTON, D. ROSE, AND R. TARJAN, *Generalized nested dissection*, SIAM J. Numer. Anal., 16 (1979), pp. 346–358.

[69] R. LIPTON AND R. TARJAN, *A separator theorem for planar graphs*, SIAM J. Appl. Math., 36 (1979), pp. 177–199.

[70] J. W.-H. LIU, *Modification of the minimum degree algorithm by multiple elimination*, ACM Trans. Math. Software, 11 (1985), pp. 141–153.

[71] ———, *A compact row storage scheme for Cholesky factors using elimination trees*, ACM Trans. Math. Software, 12 (1986), pp. 127–148.

[72] ———, *Computational models and task scheduling for parallel sparse Cholesky factorization*, Parallel Comput., 3 (1986), pp. 327–342.

[73] ———, *Equivalent sparse matrix reordering by elimination tree rotations*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 424–444.

[74] ———, *A graph partitioning algorithm by node separators*, ACM Trans. Math. Software, 15 (1989), pp. 198–219.

[75] ———, *The minimum degree ordering with constraints*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 1136–1145.

[76] ———, *The multifrontal method and paging in sparse Cholesky factorization*, ACM Trans. Math. Software, 15 (1989), pp. 310–325.

[77] ———, *Reordering sparse matrices for parallel elimination*, Parallel Comput., 11 (1989), pp. 73–91.

[78] ———, *The multifrontal method for sparse matrix solution: theory and practice*, Tech. Report CS-90-04, Dept. of Computer Science, York University, North York, Ontario, 1990.

[79] ———, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 134–172.

[80] J. W.-H. LIU AND A. MIRZAIAN, *A linear reordering algorithm for parallel pivoting of chordal graphs*, SIAM J. Discrete Math., 2 (1989), pp. 100–107.

[81] J. W.-H. LIU AND E. G.-Y. NG, *A supernodal symbolic Cholesky factorization on a local-memory multiprocessor*, 1990, in preparation.

[82] R. LUCAS, *Solving planar systems of equations on distributed-memory multiprocessors*, Ph.D. thesis, Dept. of Electrical Engineering, Stanford University, Stanford, CA, 1987.

[83] ———, Personal communication, 1990.

[84] R. LUCAS, W. BLANK, AND J. TIEMAN, *A parallel solution method for large sparse systems of equations*, IEEE Trans. Computer Aided Design, CAD-6 (1987), pp. 981–991.

[85] R. MELHEM, *Parallel solution of linear systems with striped sparse matrices*, Parallel Comput., 6 (1988), pp. 165–184.

[86] V. NAIK AND M. PATRICK, *Data traffic reduction schemes for sparse Cholesky factorization*, Tech. Report ICASE Report no. 88-14, ICASE, NASA Langley Research Center, Hampton, VA, 1988.

[87] ———, *Data traffic reduction schemes for Cholesky factorization on asynchronous multiprocessor systems*, Tech. Report ICASE Report no. 89-40, ICASE, NASA Langley Research Center, Hampton, VA, 1989.

[88] E. NG AND B. PEYTON, *A supernodal Cholesky factorization algorithm for shared-memory multiprocessors*, 1990, in preparation.

[89] J. ORTEGA, *Introduction to parallel and vector solution of linear systems*, Plenum Press, New York, 1988.

[90] J. ORTEGA, R. VOIGT, AND C. ROMINE, *A bibliography on parallel and vector numerical algorithms*, Tech. Report ORNL/TM-10998, Oak Ridge National Laboratory, Oak Ridge, TN, 1989.

[91] S. PARTER, *The use of linear graphs in Gaussian elimination*, SIAM Rev., 3 (1961), pp. 364–369.

[92] F. PETERS, *Parallel pivoting algorithms for sparse symmetric matrices*, Parallel Comput., 1 (1984), pp. 99–110.

[93] A. POTHEN, *The complexity of optimal elimination trees*, Tech. Report CS-88-16, Dept. of Computer Science, The Pennsylvania State University, University Park, PA, 1988.

[94] A. POTHEN AND C.-J. FAN, *Computing the block triangular form of a sparse matrix*, Tech. Report CS-88-51, Dept. of Computer Science, The Pennsylvania State University, University Park, PA, 1988; ACM Trans. Math. Software, to appear.

[95] A. POTHEN, H. SIMON, AND K. LIOU, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 430–452.

[96] P. RAGHAVAN AND A. POTHEN, *Parallel orthogonal factorization*, SIAM Symposium on Sparse Matrices, Gleneden Beach, OR, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1989.

[97] D. ROSE, *Triangulated graphs and the elimination process*, J. Math. Anal. Appl., 32 (1970), pp. 597–609.

[98] ———, *A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations*, in Graph Theory and Computing, R. C. Read, ed., Academic Press, New York, 1972, pp. 183–217.

[99] D. ROSE, R. TARJAN, AND G. LUEKER, *Algorithmic aspects of vertex elimination on graphs*, SIAM J. Comput., 5 (1976), pp. 266–283.

[100] E. ROTHBERG AND A. GUPTA, *Fast sparse matrix factorization on modern workstations*, Tech. Report STAN-CS-89-1286, Stanford University, Stanford, CA, 1989.

[101] P. SADAYAPPAN AND V. VISVANATHAN, *Distributed sparse factorization of circuit matrices via recursive E-tree partitioning*, SIAM Symposium on Sparse Matrices, Gleneden Beach, OR, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1989.

[102] J. SALTZ, *Aggregation methods for solving sparse triangular systems on multiprocessors*, SIAM J. Sci. Statist. Comput., 11 (1990), pp. 123–144.

[103] R. SCHREIBER, *A new implementation of sparse Gaussian elimination*, ACM Trans. Math. Software, 8 (1982), pp. 256–276.

[104] A. SHERMAN, *On the efficient solution of sparse systems of linear and nonlinear equations*, Ph.D. thesis, Yale University, New Haven, CT, 1975.

[105] P. WORLEY AND R. SCHREIBER, *Nested dissection on a mesh-connected processor array*, in New Computing Environments: Parallel, Vector, and Systolic, A. Wouk, ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1986, pp. 8–38.

[106] C. YANG AND P. VU, *A vector/parallel implementation of the multifrontal method for sparse symmetric definite linear systems on the Cray Y-MP*, Tech. Report, CRAY Research, 1990.

[107] M. YANNAKAKIS, *Computing the minimum fill-in is NP-complete*, SIAM J. Algebraic Discrete Methods, 2 (1981), pp. 77–79.

[108] E. ZMIJEWSKI, *Sparse Cholesky Factorization on a Multiprocessor*, Ph.D. thesis, Dept. of Computer Science, Cornell University, Ithaca, NY, August 1987.

[109] ———, *Limiting communication in parallel sparse Cholesky factorization*, Tech. Report TRCS89-18, Dept. of Computer Science, University of California, Santa Barbara, CA, 1989.

[110] E. ZMIJEWSKI AND J. GILBERT, *A parallel algorithm for sparse symbolic Cholesky factorization on a multiprocessor*, Parallel Comput., 7 (1988), pp. 199–210.