LAPACK WORKING NOTE 62 (UT CS-93-201) DISTRIBUTED SOLUTION OF SPARSE LINEAR SYSTEMS *

MICHAEL T. HEATH † AND PADMA RAGHAVAN ‡

Abstract. We consider the solution of a linear system $Ax = b$ on a distributed memory machine when the matrix A is large, sparse and symmetric positive definite. In a previous paper we developed an algorithm to compute a fill-reducing nested dissection ordering of A on a distributed memory machine. We now develop algorithms for the remaining steps of the solution process. The large-grain task parallelism resulting from sparsity is identified by a tree of separators available from nested dissection. Our parallel algorithms use this separator tree to estimate the structure of the Cholesky factor L and to organize numeric computations as a sequence of dense matrix operations. We present results of an implementation on an Intel iPSC/860 parallel computer. An an alternative to estimating the structure of L using the separator tree, we develop an algorithm to compute the elimination tree on a distributed memory machine. Our algorithm uses the separator tree to achieve better time and space complexity than earlier work.

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

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

1. Introduction and Overview. Consider the solution of a system of linear equations $Ax = b$, where A is an $N \times N$, symmetric positive definite matrix. Direct solution requires a Cholesky decomposition $A = LL^T$, where L is lower triangular. This step is followed by solution of the triangular systems $Ly = b$ and $L^T x = y$. When the matrix A is sparse, the numeric steps are preceded by a symbolic phase in which a symmetric permutation is applied to the rows and columns of A. The purpose of reordering the system is to ensure that the factor L suffers low fill-in, i.e., only a small number of zero values in A become nonzero during factorization. Various serial algorithms such as the minimum degree heuristic and automatic nested dissection [5] produce appropriate orderings of the matrix A by manipulating its graph $G(A)$.

As a consequence of sparsity, during numeric factorization a specific column is updated only by columns in a *subset* of lower numbered columns instead of *all* lower numbered columns. Such column dependencies, implicit from the ordering, are represented using a tree structure known as an *elimination tree.* The elimination tree is computed from the structure of A and the ordering, and is used in turn to compute the exact structure of the factor L and to allocate storage to complete the symbolic phase. In the numeric phase, transformations are applied as dictated by column dependencies given by the elimination tree. The factor L and the vector y are computed bottom-up on the elimination tree, followed by triangular solution applied top-down to compute x.

Sparse linear systems occur at the core of a wide variety of large-scale scientific applications that require the computing speed provided by parallel architectures. Therefore, the parallel solution of such linear systems is a problem of considerable importance. In our earlier work [8], we developed and implemented a Cartesian nested dissection algorithm to compute a fill-reducing ordering in parallel. In this work, we

* This research was supported by the Defense Advanced Research Projects Agency through the Army Research Office under contract number DAAL03-91-C-0047.

† Department of Computer Science and National Center for Supercomputing Applications, University of Illinois, 405 N. Mathews Ave., Urbana, IL 61801.

‡ National Center for Supercomputing Applications, University of Illinois, 405 N. Mathews Ave., Urbana, IL 61801.

provide distributed parallel algorithms for the remaining symbolic and numeric computations to complete a suite of algorithms for the parallel solution of sparse linear systems.

We now briefly describe the set of steps used in the serial case. First, the matrix is reordered according to a fill-reducing permutation in an initial symbolic step. A second symbolic step then estimates the structure of L, i.e., the column and row subscripts that will contain nonzero values in L. These values are either a result of nonzero values in the same position of A or are fill-in values that occur in the course of factorization. The position of fill-in in a column j depends on the sequence of columns by which a column j is updated. Such column dependencies are represented by paths in the elimination tree, whose nodes represent columns. The elimination tree can be computed from the structure of the reordered matrix A. This tree can be used in turn to compute the number of nonzero values in each column of L, and the row subscripts of such values, in time proportional to L, which is significantly less than the cost of computing the factor L. Finally, the tree can be viewed as a task graph for the numeric phase; columns of L are computed in post order on the elimination tree. The interested reader is referred to the excellent survey article [11] on the role of elimination trees in sparse matrix computations.

In the parallel case, a coherent suite of algorithms has not previously been proposed for this problem. Most of the earlier work has focussed on the parallelization of the numeric phase. Work on parallel numeric computations is based on the following two assumptions: (1) an elimination tree is available, and (2) the structure of the factor L is available, or at least the number of nonzeros in each column of L is known. Based on these assumptions, several authors [9, 13, 15] have proposed that a structure called a clique tree be computed (sequentially) to organize parallel numeric computations. A clique tree is a tree of maximal cliques in the filled graph, i.e., the graph of L. Since cliques correspond to dense submatrices, the clique tree is suitable for structuring sparse numeric factorization as a sequence of dense matrix operations. Clique tree computations incur exactly the fill-in and arithmetic cost arising from the ordering.

Other authors [1] have proposed using the elimination tree to compute a supernode partition; each supernode is a collection of columns that are considered effectively dense. The supernode partition is a relaxation of a clique partition. Some extra fill-in and arithmetic work are allowed in order to structure the computation in terms of dense matrices of a size suitable for efficient data parallelism in each matrix operation. Once again, such a supernode partition is based on the elimination tree. One conclusion that can be drawn from this body of work is that researchers have found it suitable to organize numeric computations in terms of tree-structured dense matrix operations. Although there is no consensus on the exact tree structure to be used, their work draws upon the elimination tree (in which each node is a column) to compute another tree structure in which each node represents a subset of effectively dense columns, i.e., a dense submatrix. In this paper, we use the term *decomposition tree* to denote the tree structure representing the sequence of dense matrix operations.

It follows from the preceding paragraphs that parallel numeric factorization draws significantly upon the elimination tree and the structure of L. The serial and parallel complexity of computing the elimination tree and the structure of L are therefore of much import. Let N be the number of equations and M the number of nonzeros in A. Let $\eta(L)$ denote the number of nonzeros in the factor L. We also introduce $\alpha(i, j)$ to stand for a quantity related to the inverse of Ackerman's function; for all practical values of i and j, $\alpha(i,j)$ is bounded above by 5. We consider parallel MIMD machines with P processors and we use N_{max}, M_{max}, and $\eta(L_{max})$ to denote, respectively, the maximum number of columns of A, the maximum number of nonzeros of A, and the maximum number of nonzeros of L assigned to any one processor.

A sequential algorithm to compute the elimination tree requires space proportional to M and time proportional to $M\alpha(M,N)$. Computing the structure of L requires time and space proportional to $\eta(L)$. The best known algorithm for computing the elimination tree on MIMD machines [20] has time requirements proportional to $(M_{max}\,\alpha(M_{max},N) + N\,\log_2 P\,\alpha(N,N))$, and space requirements proportional to $\max(M_{max},N)$ on each processor. This space complexity makes the algorithm unsuitable for large systems regardless of the number of processors available, since N may be beyond the storage available on any one processor. Furthermore, the time required increases with the number of processors.

In developing distributed algorithms for estimating the structure of L and performing numeric computations, the first question that arises is: What is a good decomposition tree for a distributed memory machine? We attempt an answer in terms of what is required of the tree structure. A primary consideration is that it should be possible to compute this tree structure efficiently in parallel. Secondly, this tree should be suitable for structuring parallel numeric computations, i.e., the tree should be short and bushy as opposed to a long chain. Finally, computations on this tree structure must result in fill-in and operation counts comparable to that of a serial algorithm based on the elimination tree; in other words, any fill-in and additional arithmetic cost introduced to enhance parallelism must be quite small. Although these requirements certainly do not result in a precise definition of the appropriate tree structure, they can be used to provide alternative tree structures that can effectively capture as many as possible of the desired characteristics.

Our work draws upon the partition tree associated with the divide-and-conquer strategy used for computing a fill-reducing ordering. A single step in nested dissection divides the matrix into two submatrices by computing a vertex separator in the graph of the matrix. This process is applied recursively and leads to a tree of separators. If nested dissection is performed efficiently in parallel [8], then the separator tree is readily available on the set of processors. We may use this separator tree in various ways to achieve good performance. For example, we can use the separator tree directly as the decomposition tree, which leads to the following sequence of steps:

1. Use the separator tree to compute an estimate of the structure of L in parallel.
2. Use the separator tree to perform numeric computations in parallel.

Alternatively, we can use a decomposition tree based on the elimination tree, but use the separator tree to implement the various steps efficiently in parallel. This approach leads to the following sequence of steps:

1. Use the separator tree to compute the elimination tree in parallel.
2. Use the elimination tree to compute the structure of L in parallel.
3. Use the elimination tree and the structure of L to compute a suitable decomposition tree (e.g., supernode or clique tree) for distributed numeric computations.
4. Use this decomposition tree to perform numeric computations in parallel.

Both of these approaches lead to a natural formulation of distributed algorithms in which each processor initially performs computations independently and in parallel in a local subtree. Processors then cooperate to perform computations associated with nodes of the decomposition tree along paths to the root. Direct use of the separator tree as the decomposition tree is attractive because it is readily available at no extra cost after parallel nested dissection. Significant additional computation would be required to determine a decomposition tree based on the elimination tree, but may lead to somewhat less work later due to the exact computation of the structure of L. In any case, the latter approach is now more viable on distributed memory machines as a consequence of our algorithms.

Our work relies significantly on earlier algorithms to compute an elimination tree [10, 19, 20], compute the structure of L [11], perform numeric factorization using the multifrontal method [3, 12], and to perform dense matrix operations on distributed machines [4, 6]. Thus, throughout this paper we include brief reviews of relevant background material interspersed with descriptions of our algorithms. Section 2 contains parallel algorithms for symbolic computations, while § 3 contains algorithms for parallel numeric factorization and triangular solution. These algorithms are suitable for any decomposition tree. Empirical results and conclusions are provided in § 4 and § 5. We conclude this section with a note on notation.

We consider an irreducible matrix A. We use the terms *vertex* and *column* interchangeably with respect to A, $G(A)$, and L. We assume that the matrix has been reordered by a fill-reducing permutation β. A symbol of the form x_j denotes a vertex in $G(A)$, or the corresponding column in A, whose β number is j. The symbol $Adj_{G(X)}(Y)$ represents the set of vertices adjacent to the set Y in the graph of the matrix X. We use $a_{k,j}$ ($l_{k,j}$) to denote the value in row k and column j of A (L). We use the phrase "structure of column j," and the notation $struc(x_j)$, to refer to the set of row subscripts of nonzero values in column x_j of L. More generally, we will use the notation $struc(X)$ to denote the structures of a set X of columns, or of an entire matrix X, as will be clear from context. We use $\mathcal{T}$ to denote either the separator tree or the decomposition tree and $\mathcal{T}_e$ to denote the elimination tree. Further subscripts are added to each of these symbols to denote suitable subtrees. The symbol $\mathcal{S}$, with or without a suitable subscript, is used to denote a collection of vertices that form a separator in $G(A)$, or a node (which might be a supernode or a clique) in a decomposition tree. Further notation is introduced as needed. For additional relevant background information on sparse linear systems, we refer the reader to [5] for serial algorithms and [7] for parallel algorithms.

2. Symbolic Algorithms. In this section we define a separator tree and use it to develop parallel symbolic algorithms. We present an algorithm to compute an (over)estimate of the structure of L in parallel using the separator tree. We then relate the separator tree to the elimination tree and provide an algorithm to compute the latter in parallel. This is followed by a brief sketch of a parallel algorithm to compute a supernode partition.

We begin the presentation of our symbolic algorithms with a more precise specification of the separator tree. A separator tree has as many nodes as the total number of separators for $G(A)$. We use the term *representative vertex* to denote a node in this tree since it represents a set of vertices of $G(A)$. Consider a separator $\mathcal{S}$ produced at some stage during nested dissection of $G(A)$. Any two vertices in $\mathcal{S}$ are related by an ancestor-descendant relationship in the elimination tree, and a similar relationship applies to any pair of separators in the separator tree. Let $\mathcal{S} = \{x_{p_1}, \ldots, x_{p_k}\}$ be a separator. Vertices within $\mathcal{S}$ are related based on the value of their β numbers; if $p_1 < p_2 < \cdots < p_k$ denotes the ordering of their β numbers, then x_{p_i} is an ancestor of x_{p_j} whenever $p_j < p_i$ for $1 \leq i, j \leq k$. In other words, the set of vertices in $\mathcal{S}$ form a chain, which we denote by the representative vertex, $rep(\mathcal{S}) = x_{p_1}$; $rep(\mathcal{S})$ is the vertex with the smallest β value in $\mathcal{S}$. Let $\mathcal{S}$ dissect G into components $G_1, \ldots, G_k$, and let $\mathcal{S}_i$ denote a separator of the subgraph $G_i, 1 \leq i \leq k$. Now $rep(\mathcal{S}_i)$ is a child of $rep(\mathcal{S})$ in the separator tree. In this work we consider a separator tree in which each representative vertex has at most two children corresponding to the two sets (not necessarily connected components) produced by the separator.

In our explanation of distributed algorithms, the separator tree plays a central role in partitioning the computation among processors. We describe a simple partition that allows easy exposition of the algorithms in later sections. For simplicity, we assume that the total number of processors, denoted by P, is a power of two. We denote the processor numbers by $\pi_0, \ldots, \pi_{P-1}$. Let p processors be assigned to subtree rooted at $rep(\mathcal{S}_i)$. Then if $p > 1$, each subtree rooted at a child of $rep(\mathcal{S}_i)$ is assigned $p/2$ processors. This partition is applied recursively, starting at the root. At level $l = \log_2 P$ from the root, each subtree is assigned a single processor. For a processor π_i, we use $L(\pi_i)$ to denote the columns in this subtree. We use the symbol $\pi_{i \pm d}$ to denote the neighbor of π_i whose processor number differs from that of π_i in the d-th bit, and $\mathcal{P}(\pi_i, d)$ is used to denote the processor subset of size 2^d to which π_i belongs. Furthermore, we use $\mathcal{S}_1(\pi_i)$ to denote the first separator such that $T_{\mathcal{S}_1(\pi_i)}$ is mapped to processors $\mathcal{P}(\pi_i, 1)$. Finally, $\mathcal{S}_i(\pi_i), \ldots, \mathcal{S}_j(\pi_i)$ denotes a sequence of separators on the path from $\mathcal{S}_1(\pi_i)$ to the root $\mathcal{S}_l(\pi_i)$.

Our aim is to use the separator tree to estimate the structure of the factor L. The following result, which characterizes nonzero positions in L, has been proved in [18].

LEMMA 2.1. *Let $j < i$. Then $l_{i,j} \neq 0$ if and only if there exists a path*

$$x_i, x_{p_1}, \ldots, x_{p_k}, x_j$$

in $G(A)$ such that all subscripts in $\{p_1, \ldots, p_k\}$ are numbered less than j.

We state part of this path condition in terms of the separator tree to enable us to use the latter to estimate $struc(L)$.

LEMMA 2.2. *For any vertex x_j in $G(A)$, let $\mathcal{S}_j$ denote the separator to which x_j belongs. Let $\mathcal{T}_{\mathcal{S}_j}$ denote the separator subtree rooted at $rep(\mathcal{S}_j)$. If $l_{i,j} \neq 0$, where $j < i$, then there exists a path*

$$x_i, x_{p_1}, \ldots, x_{p_k}, x_j$$

in $G(A)$ such that $\{x_{p_1}, \ldots, x_{p_k}\} \subseteq \mathcal{T}_{\mathcal{S}_j}$. Furthermore, either $rep(\mathcal{S}_i)$ is an ancestor of $rep(\mathcal{S}_j)$ or $rep(\mathcal{S}_i) = rep(\mathcal{S}_j)$.

Proof: By Lemma 2.1, each p_l, $1 \leq l \leq k$, is numbered less than j. Let $C = \{x_{p_1}, \ldots, x_{p_k}, x_j, x_i\}$. At some stage in nested dissection, vertices in C belong to the same subgraph. Since x_i is the highest numbered column, at least x_i is contained in the first separator that separates the component C. Since $i > j$, and x_i and x_j are connected, there are two possibilities. In one case, they could both belong to the same separator $\mathcal{S}_i$. In this case, since $i > j$, x_i is an ancestor of x_j and $rep(\mathcal{S}_i) = rep(\mathcal{S}_j)$. Alternatively, $C \backslash \{x_i\}$ could be in exactly one of the resulting components. In this case x_j will belong to a later separator. Since all separators of $C \setminus \{x_i\}$ are descendants of $\mathcal{S}_i$, it follows that $rep(\mathcal{S}_i)$ is an ancestor of $rep(\mathcal{S}_j)$. Furthermore, x_j is in $\mathcal{T}_{\mathcal{S}_j}$. The same argument can be applied to the two highest numbered vertices in $C \setminus \{x_i\}$, namely, x_j and x_{p_k} to show that x_{p_k} is a descendant of x_j and belongs to $\mathcal{T}_{\mathcal{S}_j}$. The proof follows from repeated application of the above argument.

2.1. Symbolic Factorization Using a Separator Tree. We now show that an estimate of the structure of L can be determined using a separator tree by a process similar to symbolic factorization using the elimination tree. For the separator tree, however, the structure determined can be an overestimate. Let the separator tree be as previously defined and let $\mathcal{S}_j = \{x_j, \ldots, x_k\}$. Let $child(\mathcal{S}_j)$ be the set of separators that are immediate descendants of x_j. Consider

$$\begin{aligned} lset(\mathcal{S}_j) = & \quad \bigcup_{\mathcal{S}_k \in child(\mathcal{S}_j)} struc(\mathcal{S}_k) \\ struc(\mathcal{S}_j) = & \quad \{lset(\mathcal{S}_j) \cup Adj_{G(A)}(\mathcal{S}_j)\} \setminus \{x_k : k < j\}. \end{aligned}$$

The set $struc(\mathcal{S}_j)$ provides the structure of columns corresponding to vertices in $\mathcal{S}_j$. The column corresponding to a vertex $x_k \in \mathcal{S}_j$ has a nonzero value in row i, where $i \in struc(\mathcal{S}_j)$ and $i > k$. According to this scheme, the columns of L in $\mathcal{S}_j$ are effectively dense in the subscript set given by $struc(\mathcal{S}_j)$. We next show that the above construction accounts for any nonzero value in L.

LEMMA 2.3. *Let $s_{i,j}$ denote a position in the structure computed by symbolic factorization on a separator tree. If $l_{i,j} \neq 0$ then $s_{i,j} \neq 0$, i.e., the structure computed contains space for all nonzero values of L.*

Proof: Follows from Lemma 2.2 and symbolic factorization on the separator tree.

2.2. Symbolic Factorization in Parallel. Using a separator tree, the symbolic factorization can be computed effectively in parallel as follows. Assume that processors have been mapped to subtrees of the separator tree as described earlier. Each processor initially computes the structure of the columns in its local tree independently and in parallel without any communication. Each processor π_i also computes the partial structure of columns in $\mathcal{S}_1(\pi_i), \ldots, \mathcal{S}_l(\pi_i)$ based on the data it contains. Denote this partial structure by $struc(\mathcal{S}_1, \ldots, \mathcal{S}_l)(\pi_i)$. Now each processor π_i needs the complete structure of each separator $\mathcal{S}_k(\pi_i)$ for $1 \leq k \leq l$, which we denote by $struc(\mathcal{S}_1, \ldots, \mathcal{S}_l)$. We show that the latter can be computed by pairwise merging using $\log_2 P$ steps. Let $\pi_{i\pm d}$ denote the d-th neighbor of π_i. In a first step, π_i sends $struc(\mathcal{S}_1, \ldots, \mathcal{S}_l)(\pi_i)$ to $\pi_{i\pm 1}$ and receives in exchange $struc(\mathcal{S}_1, \ldots, \mathcal{S}_l)(\pi_{i\pm 1})$. Now both processors can merge structure information to form $struc(\mathcal{S}_1)$ and $struc(\mathcal{S}_2, \ldots, \mathcal{S}_l)(\pi_i, \pi_{i\pm 1})$. In a second step π_i and $\pi_{i\pm 2}$ do the same with respect to structures of columns in $\mathcal{S}_2(\pi_i), \ldots, \mathcal{S}_l(\pi_i)$ to finish computing $struc(\mathcal{S}_2)$ and to accumulate the structure corresponding to the remaining portion over a subset of processors $\mathcal{P}(\pi_i, 2)$ of size 2^2. At the end of $\log_2 P$ such steps, each processor π_i would have the complete structure of columns in $\mathcal{S}_1(\pi_i), \ldots, \mathcal{S}_l(\pi_i)$.

We now compute the complexity of distributed symbolic factorization using the separator tree. Let $\eta(L_{\mathcal{D}})$ denote the size of the largest set of subscripts over all $struc(\mathcal{S}_1, \ldots, \mathcal{S}_l)$, and let $\eta(L_{\mathcal{L}})$ denote the maximum number of subscripts over all separators in a local phase computation. Then the communication for each processor is $O(\log_2 P \ \eta(L_{\mathcal{D}}))$ and involves $\log_2 P$ messages. The computation at each processor is $O(\eta(L_{\mathcal{L}}) + \log_2 P \, \eta(L_{\mathcal{D}}))$.

2.3. Computing the Elimination Tree in Parallel. Elimination trees have played a central role in numerous aspects of sparse matrix computations [11]. The elimination tree has N nodes and each node corresponds to a column of the matrix. An edge (x_j, x_i) links column x_j of A to the smallest column x_i, $i > j$, such that

$l_{i,j} \neq 0$, with x_i being the parent of x_j in the elimination tree. The elimination tree is a spanning tree of the filled graph $G(L)$. The elimination tree is represented by a vector *parent* of size N; $parent[j] = i$ if $i = \min\{k : l_{k,j} \neq 0, k > j\}$. Computing the *parent* vector is equivalent to determining the elimination tree.

We begin with a review of the sequential elimination tree algorithm [11]. The parent vector is first initialized to contain *nil* values. Nonzeros in the lower triangle of A are scanned in increasing order of rows. For any $a_{k,j} \neq 0$ with $k < j$, the root of the subtree containing x_k is searched for and is made the child of x_j. The running time of the algorithm is determined by the efficiency of computing the root of a subtree containing a column x_k. This can be done efficiently by using disjoint set union operations on a short and squat version of the tree. The details are not presented since they are not significant for our purposes. It suffices to state that the algorithm runs in time proportional to $M\alpha(M, N)$.

Zmijewski and Gilbert [20] have developed a parallel algorithm for MIMD machines with P processors, where each processor π_i holds M_i of a total of M nonzeros of A. In their algorithm, each processor computes a partial elimination tree based on its M_i nonzeros. These partial elimination trees are then merged to compute the actual elimination tree. The algorithm has a time complexity of $(M_{max}\ \alpha(M_{max}, N) + N\ \alpha(N, N)\ \log_2 P)$ and requires space proportional to $\max(M_{max}, N)$ on each processor. For sparse matrices arising from finite element applications, the value of M is essentially a constant times N. As a consequence, a parallel algorithm in which a processor requires storage proportional to N is not suitable since the solution of a large problem is limited by the memory available on a single processor, even though the total memory over the ensemble of processors may be more than sufficient. An added drawback is the fact that the time for computing the elimination tree increases as the logarithm of the number of processors. We propose an algorithm that overcomes these drawbacks by utilizing the separator tree to compute the elimination tree in parallel. In our algorithm, each processor computes a partial elimination tree using its own nonzeros. However, the merging of these partial elimination trees is done using the separator tree to exploit locality and thereby reduce the time and space requirements.

We first review some details of the algorithm of Zmijewski and Gilbert. Each processor π_i computes a partial elimination tree based on its nonzeros of A using the sequential algorithm. Let $\mathcal{T}_e(\pi_i)$ denote this partial elimination tree. These partial elimination trees are merged pairwise to obtain the elimination tree $\mathcal{T}_e$. Any two partial elimination trees are merged by observing that if x_k is a child of x_i in one elimination tree and x_k is a child of x_j in another, then $l_{k,j} \neq 0$ and $l_{k,i} \neq 0$. Hence, the elimination tree algorithm must process these nonzeros. If $j < i$, this would result in i being made the ancestor of j in the resulting elimination tree. This process can be formalized by scanning each $\mathcal{T}_e(\pi_i)$ and $\mathcal{T}_e(\pi_j)$ and constructing a list *nzrow* to contain the union of nonzero positions. The sequential elimination tree algorithm is used to process nonzeros represented in this list. In [20], the nonzeros of A are arbitrarily placed on processors and an $O(N)$ space and time algorithm is needed to compute the list *nzrow*. Furthermore, running the serial elimination tree algorithm on this list takes time proportional to $N\alpha(N, N)$. The elimination tree is available at a processor after $\log_2 P$ such merges.

Our algorithm performs merges more selectively since we assume that the nonzeros of A are partitioned among processors as explained earlier. Let $L(\pi_i)$ denote the columns assigned to π_i for local phase computations. The nonzeros of all columns in $L(\pi_i)$ are contained entirely on processor π_i. Assume the processor computes an elimination tree $\mathcal{T}_e(\pi_i)$ using its nonzeros and the sequential algorithm. Recall that π_i also has nonzeros corresponding to columns on the path from its local phase subtree to the root. Let this path consist of separators $\mathcal{S}_1(\pi_i), \ldots, \mathcal{S}_l(\pi_i)$, where $l \approx \log_2 P$. We show that $\mathcal{T}_e(\pi_i) = \mathcal{T}_e(L(\pi_i)) + \mathcal{T}_e(\mathcal{S}_1, \ldots, \mathcal{S}_l)(\pi_i)$. Now $\mathcal{T}_e(L(\pi_i))$ cannot be affected by nonzeros in any other processor. Only $\mathcal{T}_e(\mathcal{S}_1, \ldots, S_l)(\pi_i)$ need be merged with corresponding portions from other processors. We state this more formally in Lemma 2.4, where we show that after π_i and $\pi_{i\pm1}$ merge their elimination trees, a complete elimination tree is formed for all columns in the set $L(\pi_i)$, $L(\pi_{i\pm1})$, and $\mathcal{S}_1(\pi_i)$. This result can be applied repeatedly to obtain the entire elimination tree.

This merging of partial elimination trees could be done pairwise. In the first step, processor subsets of size 2 can exchange their lists and merge. At the end of that step, columns in the first separator S_1 would be completed for each pair of processors. In the next step processors π_i and $\pi_{i\pm2}$ communicate and merge lists pertaining to the remaining separators. At the end of $\log_2 P$ such steps the portion of the elimination tree required by each processor would be available at that processor. Let $N_{\mathcal{D}}$ denote the maximum number of columns over all separators in the distributed phase on a path to the root. Then the work at a processor for the pairwise merging would be proportional to $(N_{\mathcal{D}}\, \alpha(N_{\mathcal{D}}, N_{\mathcal{D}}) \log_2 P)$ and the total cost would be proportional to $(M_{max}\, \alpha(M_{max}, N) + N_{\mathcal{D}}\, \alpha(N_{\mathcal{D}}, N_{\mathcal{D}}) \log_2 P)$. The communication volume would be proportional to $(N_D \log_2 P)$ and would involve $\log_2 P$ messages.

If message latency is small, then more messages could be exchanged to cut down the amount of work at each processor. Assume $P_{\mathcal{S}_i}$ processors are involved for a separator $\mathcal{S}_i$. At the first step, $P_{\mathcal{S}_i} = 2$ and processors could communicate the portions corresponding to $\mathcal{S}_1$ and merge them. At the next step $P_{\mathcal{S}_2} = 4$, and over $\log_2(\mathcal{P}_{\mathcal{S}_2})$ steps lists corresponding to $\mathcal{S}_2$ could be merged, and so on. As a result, the communication at a processor would be bounded by $S_{max}(\log_2 P)^2$ over $(\log_2 P)^2$ messages, where S_{max} is the size of the largest separator. The cost of merging would then be bounded by $S_{max}(\log_2 P)^2\, \alpha(S_{max}, S_{max})$, leading to an $O(M_{max}\alpha(M_{max}, N_{max}) + S_{max}\, (\log_2 P)^2\, \alpha(S_{max}, S_{max}))$ algorithm.

We now show that the elimination tree computed by merging based on the separator tree is indeed correct. In the proof we use $\mathcal{T}_e$ to stand for the parent vector corresponding to nodes $x_1, \ldots, x_N$. The symbol $\mathcal{T}_e(\pi_i, \ldots, \pi_j)$ denotes the parent vector computed using information over processors $\pi_i, \ldots, \pi_j$, and $\mathcal{T}_e(S)(\pi_i, \ldots, \pi_j)$ denotes the portion of the parent vector corresponding to columns in S over all processors $\pi_i, \ldots, \pi_j$.

LEMMA 2.4. *The tree computed by a processor π_i based on its portion of A is $\mathcal{T}_e(\pi_i) = \mathcal{T}_e(L(\pi_i)) + \mathcal{T}_e(\mathcal{S}_1, \ldots, \mathcal{S}_l)(\pi_i)$. Furthermore, upon merging $\mathcal{T}_e(\mathcal{S}_1, \ldots, \mathcal{S}_l)(\pi_i)$ and $\mathcal{T}_e(\mathcal{S}_1, \ldots, \mathcal{S}_l)(\pi_{i\pm1})$, the resulting tree is $\mathcal{T}_e(\mathcal{S}_1) + \mathcal{T}_e(\mathcal{S}_2, \ldots, \mathcal{S}_l)(\mathcal{P}(\pi_i, 1))$.*

Proof: A processor π_i is assigned all nonzeros corresponding to columns in $L(\pi_i)$. Now $\mathcal{S}_1$ is a separator disconnecting the graph induced by the structure of $L(\pi_i)$ from the rest of $G(A)$. As a result, all paths of the form $x_i, x_{p_1}, \ldots, x_{p_k}, x_j$ with $j < i$ and $p_1, \ldots, p_k$ each less than i, must be contained in the subgraph induced by $L(\pi_i)$. It follows from Lemma 2.1 that $\mathcal{T}_e(\pi_i)$ contains $\mathcal{T}_e(L(\pi_i))$.

Assume the merge is performed as described. We show that the resulting tree contains the final parent values for columns in $\mathcal{S}_1$. We do this by proving that if $x_i, x_{p_1}, \ldots, x_{p_l}, x_j$ is a path such that $i < j$ and x_i is in $\mathcal{S}_1$, then x_i is a descendant of x_j in the merged tree. The proof is by induction on the length of the path. If $l = 0$, then $a_{j,i} \neq 0$. It must be assigned to one of the processors, and hence x_i must be a descendant of x_j in either of $\mathcal{T}_e(\pi_i)$ or $\mathcal{T}_e(\pi_{i+1})$. By construction, x_i is a descendant of x_j in the merged tree.

Assume the statement holds for all paths of length l. Consider the path of length l given by $x_i, x_{p_1}, \ldots, x_{p_l}$. Now x_{p_l} is a descendant of x_i. By a similar argument we can show that x_{p_l} is a descendant of x_j. By the merging construction, *nzrow* has nonzeros corresponding to (x_i, x_{p_l}) and (x_j, x_{p_l}). The elimination tree algorithm makes x_i a descendant of x_j. Since $\mathcal{S}_1$ is separated from the rest of the graph by $\mathcal{S}_2$, this merge completes the subtree corresponding to columns in $\mathcal{S}_1$. For all columns in $\mathcal{S}_2, \ldots, \mathcal{S}_l$, the merging results in the parent vector based on contributions from $\mathcal{P}(\pi_i, 1) = \{\pi_i, \pi_{i\pm 1}\}$. Hence the proof.

2.4. Computing a Decomposition Tree. Once an elimination tree has been computed, processors could cooperate to determine an initial supernode partition. A simple initial supernode tree would result if chains (a sequence of nodes in the elimination tree in which all but the first and last have no siblings) are identified. This can be accomplished by having processors compute the number of children for each node and merging this information in pairwise fashion over $\log_2 P$ steps. The exact structure of the factor L can be computed by performing symbolic factorization on the initial supernode tree using an algorithm similar to the one in Section 2.1. This would result in sufficient information to compute a suitable decomposition tree [1, 9, 15]. We do not provide an explicit algorithm here since there are several possible choices for a clique or supernode partition.

3. Numeric Factorization and Triangular Solution. This section presents distributed algorithms for the numeric phase of the computation. The problem of partitioning computations among processors is discussed, followed by details of organizing numeric computations in terms of distributed dense kernels for factorization and triangular solution.

In order to solve the equation $Ax = b$, we need to compute the numerical entries of the Cholesky factor L so that $A = LL^T$. We then compute the vector y satisfying $Ly = b$, and finally the solution vector x satisfying $L^T x = y$. Numeric computations are based on a decomposition tree, in which each representative vertex is associated with processing a chain of effectively dense columns. Such a set of computations is essentially the multifrontal method [3]. We now provide a brief overview of the multifrontal method; the reader is referred to [3, 12] for a comprehensive treatment.

Consider a representative vertex $rep(\mathcal{S}_i)$, with $\mathcal{S}_i = \{x_i, \ldots, x_k\}$. Let $struc(\mathcal{S}_i)$ denote the structure of columns in $\mathcal{S}_i$ and let $anc(\mathcal{S}_i) = struc(\mathcal{S}_i) \setminus \mathcal{S}_i$ denote the set of vertices that are ancestors of vertices in $\mathcal{S}_i$. Consider the case when $rep(\mathcal{S}_i)$ is a leaf node in the decomposition tree. Then columns in $\mathcal{S}_i$ can be factored to result in corresponding columns of L. However, these factored columns affect columns in $anc(\mathcal{S}_i)$, and this information must be accumulated and propagated up the tree as needed. The process of factoring columns in $\mathcal{S}_i$ and accumulating updates to columns in $anc(\mathcal{S}_i)$ is simply a partial Cholesky factorization of a dense submatrix D_i, composed of columns in $struc(\mathcal{S}_i)$. Let the dense matrix be

$$D_i = \begin{pmatrix} F_i & 0 \\ \tilde{F}_i & H_i \end{pmatrix}.$$

The submatrices F_i and H_i correspond to columns in $\mathcal{S}_i$ and $anc(\mathcal{S}_i)$. In the case $rep(\mathcal{S}_i)$ is not a leaf, let its children vertices be given by the set $child(\mathcal{S}_i)$. For each child $rep(\mathcal{S}_j) \in child(\mathcal{S}_i)$, values in the submatrix associated with columns in $anc(\mathcal{S}_j)$ must be assembled into D_i. After this assembly process, a partial factorization of the columns of $struc(\mathcal{S}_i)$ in $\mathcal{S}_i$ can be computed to complete the factorization step at $rep(\mathcal{S}_i)$, resulting in

$$\begin{pmatrix} L_i & 0 \\ \tilde{L}_i & U_i \end{pmatrix}.$$

The submatrices L_i and $\tilde{L}_i$ correspond to completed columns of the factor L; the submatrix U_i must be assembled into the matrix for the parent of $\mathcal{S}_i$.

The forward solution step can be applied immediately after the partial factorization step for $rep(\mathcal{S}_i)$. Assume values of b and y corresponding to columns in $\mathcal{S}_i$ are gathered into b_i and y_i. Furthermore, let $\tilde{b}_i$ and $\tilde{y}_i$ correspond to columns in $anc(\mathcal{S}_i)$. Then y_i can be computed by partial triangular solution of the system

$$\begin{pmatrix} L_i & 0 \\ \tilde{L}_i & U_i \end{pmatrix} \begin{pmatrix} y_i \\ \tilde{y}_i \end{pmatrix} = \begin{pmatrix} b_i \\ \tilde{b}_i \end{pmatrix}.$$

Once again, the effect of the triangular solution for components in b corresponding to $anc(\mathcal{S}_i)$ must be propagated to the parent by using $\tilde{b}_i$. Likewise, if $\mathcal{S}_i$ is not a leaf, then effects from children vertices are assembled into b_i and $\tilde{b}_i$ prior to computing a partial triangular solution.

After partial factorization and triangular solution are applied at all representative vertices in a bottom-up traversal of the decomposition tree, a sequence of triangular solutions must then be applied top-down to compute the final solution x. If $rep(\mathcal{S}_i)$ is the root of the decomposition tree, then the associated matrix is L_i, and the system $L_i^T x_i = y_i$ is solved to compute x_i. Let $rep(\mathcal{S}_j)$ be a child of $rep(\mathcal{S}_i)$. The matrix associated with $rep(\mathcal{S}_j)$ is

$$\begin{pmatrix} L_j & 0 \\ \tilde{L}_j & U_j \end{pmatrix}.$$

Values of x corresponding to columns in $anc(\mathcal{S}_j)$ are available from the solution at $rep(\mathcal{S}_i)$. Let these values be denoted by $\tilde{x}_j$. Then the solution x_j is computed using

$$L_j^T x_j = y_j - \tilde{L}_j^T \tilde{x}_j.$$

3.1. Distributed Multifrontal Computation. We now consider the implementation of multifrontal numeric computations on a distributed-memory parallel architecture. The major issues to be addressed are the assignment of subsets of processors to decomposition subtrees, and the effective use of dense matrix kernels given that the effects of partial dense matrix operations must be propagated to ancestors or descendants with low overhead in storage, computation, and interprocessor communication. We discuss these issues in this section.

Primary considerations in assigning processor subsets to subtrees in the decomposition tree are to exploit both task parallelism and data parallelism while minimizing communication overhead and load imbalance. Since computations in disjoint subtrees are independent, a natural approach would be to select P disjoint subtrees and assign one to each processor as a local phase subtree. Computations within this subtree can be processed locally and in parallel on each processor without any communication. For any of the remaining representative vertices, say $rep(\mathcal{S}_i)$, let $\mathcal{P}_{\mathcal{S}_i}$ denote the set of processors whose local subtrees are contained in $\mathcal{T}_{\mathcal{S}_i}$. The data required for computations at $rep(\mathcal{S}_i)$ are distributed among processors in $\mathcal{P}_{\mathcal{S}_i}$ and it is natural to distribute the computations at $rep(\mathcal{S}_i)$ among processors in this set. Observe that in such an assignment, the size of disjoint processor subsets increases towards the root. This approach is well suited to tree-structured sparse computations in which the number of independent tasks decreases toward the root while the task size increases. Thus, as the amount of task parallelism decreases toward the root, the amount of data parallelism increases due to the increasing size of the dense matrices involved.

The remaining question is how to select the subtrees of the decomposition tree for the local phase. Most assignment schemes can be expressed recursively as follows. Consider a representative vertex $rep(\mathcal{S}_i)$ assigned a subset of processors $P_{\mathcal{S}_i}$ of size $p_i > 1$. Suppose that $rep(\mathcal{S}_i)$ has n children vertices labeled $rep(\mathcal{S}_j)$, $1 \leq j \leq n$. Partition the set of processors $P_{\mathcal{S}_i}$, into n disjoint subsets of size $c_j\, p_i$ for $1 \leq j \leq n$. Assign the processor subset of size $c_j\, p_i$ to $rep(\mathcal{S}_j)$ for $1 \leq j \leq n$ (i.e., c_j determines the proportion of processors assigned to child $rep(\mathcal{S}_j)$). Apply the above assignment starting with the root and P processors. The processor assignment could be changed for each major step of the computation, that is, the assignment for symbolic factorization could be different from that for numeric factorization. Each such change of assignment would require data redistribution, however, which is an expensive operation on current MIMD machines.

The effectiveness of a given assignment scheme is ultimately determined by the load balance and communication requirements it leads to, but the cost of computing the assignment must also be taken into consideration. In the case of using the separator tree as the decomposition tree, the balance criterion used in Cartesian nested dissection is meant to provide some control over the balance of the resulting separator tree, but of course the actual amount of work in each subtree is not completely determined simply by the number of vertices in the respective subgraphs. Thus, the simple assignment scheme described earlier, in which each subtree at a given level is given equal weight, may or may not provide an adequate load balance in subsequent computations. This simple assignment scheme does have the virtue, however, of resulting in exactly $\log_2 P$ distributed steps and very straightforward code for pairwise merging. It also requires no additional computation to determine the assignment, which can be accomplished directly after parallel nested dissection.

More sophisticated assignment schemes include making the processor subsets of size proportional to the number of nonzeros or columns in each subtree [14], or the total arithmetic work in each subtree [16]. For most practical problems, these would result in at most $c \log_2 P$ distributed steps at a processor, where c is a small constant. Using the actual arithmetic work could in principle provide the best possible load balance, but unfortunately this would require in an additional redistribution step, since the arithmetic work cannot be estimated until after the structure of L has been computed. Thus, there is a tradeoff here between load balance and communication overhead. Our initial implementation uses the separator tree as the decomposition tree and the simple assignment scheme.

The issue of propagating effects of partial factorization and triangular solution efficiently in the distributed phase is closely tied to that of the choice and effective use of distributed dense kernels. Distributed dense kernels are required for partial Cholesky factorization and triangular solution. A natural choice would a column-oriented Cholesky factorization algorithm such as fan-out or fan-in [4]. For relatively small dense matrices arising from sparse factorization, a wrap mapping of columns to processors is suitable for providing load balance for a given dense matrix operation. Such a wrap mapping is not readily available in sparse factorization, however, since a dense submatrix is obtained from the assembly of update matrices from descendent representative vertices, so that, for example, the columns of update matrices would be wrap mapped among processors in each of two disjoint processor subsets.

A redistribution step to allow wrap mapping prior to each distributed dense factorization would be prohibitively expensive on current architectures. We propose instead a strategy that weaves redistribution into the communication required for factorization. Consider a representative vertex $rep(\mathcal{S}_i)$ with children vertices $rep(\mathcal{S}_j)$ and $rep(\mathcal{S}_k)$. The matrix to be partially factored is D_i, which consists of original nonzeros of A corresponding to columns in $\mathcal{S}_i$ and update submatrices U_j and U_k. Each processor in the set $\mathcal{P}_{\mathcal{S}_i}$ allocates enough storage to contain its share of a wrap map of columns of D_i, while retaining its share of either of U_j or U_k. Consider dense factorization with $n = |struc(\mathcal{S}_i)|$ and columns in D_i numbered $1, \ldots, n$. We use the fan-in algorithm, in which, to compute a given column l, all processors in the set $\mathcal{P}_{\mathcal{S}_i}$ compute the update to column l from their columns numbered less than l and send it to the processor assigned column l. In the context of sparse factorization, a processor also adds the appropriate column of either of U_j or U_k. This approach provides the benefits of wrap-mapping, the systematic assembly of update matrices, and the use of a well established dense factorization kernel, without requiring an extra communication phase for data redistribution.

We now examine the computational work required. Let $t = |\mathcal{S}_i|$, and let $p = |\mathcal{P}_{\mathcal{S}_i}|$. The arithmetic work is that of factoring t columns of a dense matrix of size n and modifying the $n - t$ later columns by the first t columns. The higher order serial arithmetic cost is $A_1 = (1/2)(n^2t - nt^2 + t^3/3)$. The arithmetic cost for each processor is $A_p = A_1/p + n^2/2 + np/2$. However, the communication cost for each processor is approximately $n^2/2$. Given the large communication to computation ratio for current MIMD machines, this can seriously degrade performance when t is small. A way to reduce this cost would be to postpone the assembly and updating of the later $n - t$ columns. If the assembly and update were postponed until each column becomes a factor column, we would then have the sparse fan-in factorization algorithm [2], which requires sparse storage schemes and more overhead.

An approach similar to the numeric factorization is used for performing a partial forward triangular solution. The fact that D_i is wrap mapped as a result of factorization serves well for this step. The components of y_i and $\tilde{y}_i$ are wrap-mapped to processors, and a fan-in triangular solver [6] is used. Again, the assembly of contributions to the l-th component in b_i is woven into the communication required to compute the l-th component of y. The arithmetic work for each processor is $nt - t^2/2$, and the communication cost is n.

In performing a triangular backsolve, the problem is one of distributing the effects for use at a descendant vertex after computation. This is unlike forward solution (and factorization), in which effects from descendants were assembled or accumulated prior to computation. In this setting, a fan-out triangular solver [6] algorithm is most suitable and is the one we use. The complexity of this algorithm is the same as that of the fan-in algorithm used for forward solution.

4. Computational Results. The overall performance of the algorithms we have described above depends on the total amount of work done, how that work is distributed across multiple processors, the execution rate of the individual processors, and the cost of communication among the processors. The total amount of work for a given problem depends mainly on the effectiveness of the ordering in limiting fill. We demonstrated in [8] that Cartesian nested dissection is competitive with other standard methods in this regard. We present computational results in this section pertaining to the other performance issues just mentioned.

Our sparse test problems are described in Table 1. The first three problems are $k \times k$ regular square grids with $k = 400$, 500, and 600, respectively. The remaining problems were produced by the commercial finite element package PATRAN. These problems are patterned roughly after a series of test problems found in [5], but are highly irregular and graded, with elements varying widely in size and density, in order to present a challenge to our algorithms. In the table, the column headed "$\frac{1}{2}|A|$" indicates the number of nonzeros in the lower triangle of the symmetric matrix A.

TABLE 1
Description of test problems (numbers in thousands).

Label	N	$\frac{1}{2}\|A\|$	Comments
G400	160	319	square grid
G500	250	499	square grid
G600	360	718	square grid
GHS1	39	113	hollow square
GHS2	74	216	hollow square
GPH1	35	104	pinched hole
GPH2	62	185	pinched hole
G6H1	30	89	6 holes
G6H2	61	181	6 holes
GL1	34	101	L shape
GL2	69	208	L shape

Our numerical experiments were all done on an Intel iPSC/860 hypercube at Oak Ridge National Laboratory. This machine has 128 Intel i860 processors. Our programs were written in C, but we used assembler-coded BLAS to enhance performance of the dense computational kernels in our algorithm. To give some idea of the performance level of this machine for our dense kernels, we provide in Table 2 the execution rates of Cholesky factorization and triangular solution for a dense matrix of order 600 using various numbers of processors. The choice of order 600 for illustration is based on the fact that this is the largest dense submatrix that occurs for any of our sparse test problems in our multifrontal implementation of sparse numeric factorization and triangular solution. The single-processor speeds in Table 2 indicate the maximum performance per processor that we can expect during the local phase of our algorithms, and the multi-processor speeds give some idea of the incremental effect of additional processors during the global phase of our algorithms. As expected for a problem of fixed size, performance flattens out as the number of processors increases. In particular, there is essentially no incremental performance gain beyond 32 processors for a dense problem of this size.

TABLE 2
Execution rate in Mflops for dense matrix of order 600

P	1	2	4	8	16	32	64
factor	14.3	25.5	41.7	58.2	72.6	78.9	79.1
solve	1.8	2.2	2.9	3.5	3.7	3.8	3.8

The sparse matrices in Table 1 were ordered by our Cartesian nested dissection algorithm, and the resulting separator tree was used for organizing the computations and assigning work to processors, as described above. The balance criterion used in Cartesian nested dissection was chosen to maintain, at each level, an approximately equal split between the resulting subgraphs. This choice was based on our experience that good load balance is usually more important in a parallel implementation than the greater reduction in fill that might result from a smaller separator. Moreover, choosing the smallest possible separator at a given level of dissection may actually produce a worse choice of separator at some subsequent level, and so does not necessarily produce consistently better results over the whole computation.

We do not give results for the symbolic factorization step because the execution time for this step is negligible compared to the other steps and is difficult to measure consistently. Unlike conventional symbolic factorization algorithms, there is essentially no computation in our approach other than overhead, such as storage allocation, whose execution time is inherently somewhat erratic (due to fractionation of memory, garbage collection, etc.). Since our approach to the symbolic part of the computation is inherently scalable, we expect its execution time to remain negligible (relative to the other steps in the computation) for larger problems using larger numbers of processors.

We first examine the load balance across processors that results from the Cartesian nested dissection ordering and corresponding task assignment for multifrontal numeric factorization. We are particularly interested in the behavior of the load balance as the number of processors increases, and hence more levels of dissection are required. (Recall that our simple assignment scheme does not take the actual arithmetic work in subtrees directly into account, but instead relies on the natural balance that tends to be produced by nested dissection.) In Table 3 we show the minimum and maximum amount of work on any processor for each test problem and for various numbers of processors. In this and subsequent tables, blank entries indicate that the corresponding problem could not be solved using the amount of memory available on the given number of processors. We first note that for a given number of processors, the maximum difference in processor workloads is almost always within a factor of four, and usually within a factor of two. Thus, the overall load balance seems to be within reason, especially given the vagaries of the irregular problems We note further that the maximum load is generally cut approximately in half by each additional level of nested dissection, showing that the problem is indeed being distributed among the processors as intended. Thus, our algorithm seems to be doing a good job of splitting the problems into as many pieces as necessary for reasonably well balanced parallel execution. Similar comments also apply to the load balance for the triangular solution, shown in Table 4, but of course the total amount of work is much smaller.

We next examine the execution rate for distributed multifrontal numeric factorization. In Table 5 we show the overall execution rate using various numbers of processors. We do not give "speedup" figures (i.e., performance relative to a single processor) because none of the problems could be solved in memory on a single processor. In Table 5 we see a number of expected effects: larger problems yield greater performance, but for any fixed problem, relative performance flattens out as more processors are employed. We see a maximum execution rate of 400 Mflops, despite the fact that the largest dense subproblem runs at only 79 Mflops according to Table 2. The difference, of course, is due to the local phase of the distributed multifrontal algorithm, during which there is no communication, so that we can achieve a considerably higher execution rate for the overall computation than the largest dense

Table 3
Numeric factorization load balance. Minimum and maximum number of operations (in millions) on any processor

P	1	8		16		32		64		128	
		min	max	min	max	min	max	min	max	min	max
G400	1225			57	91	26	46	13	24	6	12
G500	2470					52	91	35	46	12	24
G600	4250					90	157	44	80	21	41
GHS1	82	8	11	4	6	1.7	2.8	0.9	1.5	0.6	0.8
GHS2	191	21	24	11	15	4.2	6.9	2.2	3.7	1.4	1.7
GPH1	179	13	30	5	19	2.3	11.3	1.2	5.5	0.6	2.8
GPH2	451	34	84	13	52	6	33	3	17	1.6	9.3
G6H1	75	8	11	4	6	2.0	3.2	1.0	1.8	0.5	0.9
G6H2	213	23	34	11	18	5.7	9.5	2.4	4.8	1.5	2.7
GL1	580	63	89	31	79	17	24	7.4	8.9	4.1	4.9
GL2	2492			96	299	55	151	29	77	15	40

Table 4
Triangluar solution load balance. Minimum and maximum number of operations (in millions) on any processor

P	1	8		16		32		64		128	
		min	max	min	max	min	max	min	max	min	max
G400	26			1.5	1.7	.72	.88	.34	.44	.16	.22
G500	46					1.2	1.5	.56	.74	.28	.38
G600	64					1.8	2.2	.86	1.1	.40	.56
GHS1	4.2	.48	.55	.22	.28	.10	.15	.05	.08	.02	.04
GHS2	8.4	4.0	2.0	.44	.60	.20	.32	.10	.16	.06	.08
GPH1	6.4	.56	1.0	.24	.62	.10	.36	.05	.18	.02	.10
GPH2	12.6	1.1	2.2	.48	1.3	.20	.82	.10	.44	.04	.22
G6H1	4.0	.46	.54	.22	.28	.10	.14	.04	.08	.02	.02
G6H2	9.0	1.0	1.2	.50	.64	.24	.32	.10	.16	.06	.08
GL1	8.0	.86	1.0	.40	.60	.20	.30	.09	.11	.04	.06
GL2	19			.96	1.9	.48	1.0	.24	.51	.12	.24

subproblem (which actually occurs during the later global phase of the algorithm) would seem to suggest. Thus, our algorithm is benefiting from the additional task parallelism present in sparse systems that is not available in the dense case.

TABLE 5
Numeric factorization execution rate in Mflops

P	8	16	32	64	128
G400		144	187	227	247
G500			224	289	324
G600			253	340	400
GHS1	34	53	72	85	89
GHS2	31	63	94	110	115
GPH1	67	92	114	130	138
GPH2	86	120	158	187	206
G6H1	43	68	86	98	107
G6H2	57	95	126	152	167
GL1	63	79	100	108	127
GL2		94	131	179	207

In Table 6 we show the corresponding execution rates for the triangular solution phase. Here there is substantially less computation over which to amortize the communication cost, so that the flattening out of performance occurs more quickly than it does for factorization. Indeed, performance for the triangular solution is more or less flat as the number of processors varies, actually declining for the largest numbers of processors. Still, given the nature of triangular solution and our previous experience with parallel algorithms for it, just maintaining flat performance over a fairly wide range of processors suggests that our algorithms are controlling communication costs and spreading the small amount of computational work reasonably well. Moreover, this is all that is really required, given that the total computation time is still dominated by the numeric factorization phase.

TABLE 6
Triangular solution execution rate in Mflops

P	8	16	32	64	128
G400		24	26	23	18
G500			31	29	25
G600			36	33	29
GHS1	12	15	13	11	8
GHS2	14	17	18	13	11
GPH1	15	19	19	15	12
GPH2	16	21	23	22	16
G6H1	13	17	15	11	10
G6H2	15	22	23	20	15
GL1	10	10	9	8	7
GL2		10	9	8	8

5. Conclusions. In this paper we have described a series of symbolic and numeric factorization algorithms for solving sparse linear systems by Cholesky factorization. This work completes a suite of algorithms that began with the Cartesian nested dissection algorithm presented in [8]. This suite of algorithms is the first we know of that solves the entire problem, from initial ordering through triangular solution, in parallel on a distributed memory architecture.

We have used the concept of a decomposition tree to organize the computations in a way that balances the workload across the processors, limits communication, and permits the use of efficient dense kernels. The particular decomposition tree that we use is derived naturally from the sequence of separators computed by the Cartesian nested dissection algorithm. We showed how this approach could also produce the exact elimination tree, if desired, efficiently in parallel. This distributed multifrontal implementation of Cholesky factorization enjoys a local phase requiring no communication, followed by a global phase in which information is systematically merged across processors. Thus, the method is scalable in the sense that for any given number of processors, the local phase will be dominant for large enough problems.

We illustrated the effectiveness of the algorithms by computational experiments with a sequence of sparse test problems. The algorithm was seen to produce a good load balance across processors. The overall performance of the numeric factorization was quite satisfactory, given that our test problems did not saturate the full memory capacity of the machine, but of course there is an inevitable flattening out of performance for any fixed problem as the number of processors grows. Performance of the triangular solution was much lower, due to the much smaller amount of computation, and was rather flat as the number of processors varied. This behavior is actually an improvement over some previous efforts, and ultimately the triangular solution would also be dominated by its local phase for sufficiently large problems.

Our original goal in this work was to produce a complete suite of prototype scalable algorithms and software for all phases of solving sparse linear systems. We feel that we have largely succeeded in accomplishing this goal, but obviously much remains to be done. Among the potential algorithmic variations we have identified, our initial experimental implementation has, for the most part, taken the simplest option in each case. A number of the alternatives should be explored as well, however, such as different choices for the decomposition tree and various strategies for assigning work to processors. As another example, our one-dimensional, column-oriented partitioning of the matrix holds up well for problems of the sizes we have encountered thus far, but for ultimate scalability to extremely large problems and numbers of processors, a two-dimensional partitioning of the matrix may become necessary. In future work we expect to continue to improve the performance of these algorithms, to integrate them better, and to test them on much larger and more diverse problems as this becomes more logistically feasible. In particular, Cartesian nested dissection has recently been generalized to three-dimensional problems [17]. We also expect to generalize this work to include LU and QR factorizations for nonsymmetric and nonsquare problems.

6. Acknowledgement. We wish to thank Oak Ridge National Laboratory for providing access to the Intel iPSC/860 computer used in our computational experiments.

REFERENCES

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

[2] C. Ashcraft, S. Eisenstat, and J. W.-H. Liu, *A fan-in algorithm for distributed sparse numerical factorization*, SIAM J. Sci. Stat. Comput., 11 (1990), pp. 593–599.

[3] I. Duff and J. Reid, *The multifrontal solution of indefinite sparse symmetric linear equations*, ACM Trans. Math. Software, 9 (1983), pp. 302–325.

[4] G. Geist and M. Heath, *Matrix factorization on a hypercube multiprocessor*, in Hypercube Multiprocessors, M. T. Heath, ed., Philadephia, PA, 1986, SIAM Publications.

[5] J. A. George and J. W.-H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1981.

[6] M. Heath and C. Romine, *Parallel solution of triangular systems on distributed-memory multiprocessors*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 558–588.

[7] M. T. Heath, E. Ng, and B. W. Peyton, *Parallel algorithms for sparse linear systems*, SIAM Review, 33 (1991), pp. 420–460.

[8] M. T. Heath and P. Raghavan, *A Cartesian nested dissection algorithm*, Tech. Rep. UIUCDCS-R-92-1772, Department of Computer Science, University of Illinois, Urbana, IL 61801, October 1992.

[9] J. Lewis, B. Peyton, and A. Pothen, *A fast algorithm for reordering sparse matrices for parallel factorization*, SIAM J. Sci. Stat. Comput., 10 (1989), pp. 1156–1173.

[10] J. W.-H. Liu, *A compact row storage scheme for Cholesky factors using elimination trees*, ACM Trans. Math. Software, 12 (1986), pp. 127–148.

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

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

[13] B. Peyton, *Some applications of clique trees to the solution of sparse linear systems*, PhD thesis, Department of Mathematical Sciences, Clemson University, Clemson, SC, 1986.

[14] P. Plassmann, *The parallel solution of nonlinear least squares problems*, PhD thesis, Center for Applied Mathematics, Cornell University, Ithaca, NY, 1990.

[15] A. Pothen and C. Sun, *A distributed multifrontal algorithm using clique trees*, Tech. Rep. CS-91-24, Dept. of Computer Science, Pennsylvania State University, University Park, PA 16802, 1991.

[16] P. Raghavan, *Distributed sparse matrix factorization: QR and Cholesky decompositions*, PhD thesis, Department of Computer Science, Pennsylvania State University, University Park, PA, 1991.

[17] ———, *Line and plane separators*, Tech. Rep. UIUCDCS-R-93-1794, Department of Computer Science, University of Illinois, Urbana, IL 61801, February 1993.

[18] D. Rose, R. Tarjan, and G. Lueker, *Algorithmic aspects of vertex elimination on graphs*, SIAM J. Comput., 5 (1976), pp. 266–283.

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

[20] E. Zmijewski and J. Gilbert, *A parallel algorithm for sparse symbolic Cholesky factorization on a multiprocessor*, Parallel Computing, 7 (1988), pp. 199–210.