EFFICIENT PARALLEL SOLUTIONS OF LARGE SPARSE SPD SYSTEMS ON DISTRIBUTED-MEMORY MULTIPROCESSORS

CHUNGUANG SUN*

Abstract. We consider several issues involved in the solution of sparse symmetric positive definite systems by multifrontal method on distributed-memory multiprocessors. First, we present a new algorithm for computing the partial factorization of a frontal matrix on a subset of processors which significantly improves the performance of a distributed multifrontal algorithm previously designed. Second, new parallel algorithms for computing sparse forward elimination and sparse backward substitution are described. The new algorithms solve the sparse triangular systems in a multifrontal fashion. Numerical experiments run on an Intel iPSC/860 and an Intel iPSC/2 for a set of problems with regular and irregular sparsity structure are reported. More than 180 million flops per second during the numerical factorization are achieved for a three-dimensional grid problem on an iPSC/860 machine with 32 processors.

Key words. Cholesky factorization, clique tree, distributed-memory multiprocessors, multifrontal method, parallel computing, sparse matrix algorithms

AMS(MOS) subject classifications. 05C50, 15A23, 65F05, 65F50

1. Introduction.

The numerical solution of a large sparse symmetric positive definite system $Ax = b$ lies at the heart of many scientific and engineering computations. One way of solving such a system is first to compute the sparse Cholesky factorization $PAP^T = LL^T$, where L is a lower triangular matrix and P is a permutation matrix for reducing the number of nonzeros in L. The solution vector is computed by solving the two sparse triangular systems $Lu = Pb$ and $L^Tv = u$, and setting $x = P^Tv$. Typically, a direct solution process involves four steps: (i) computation of a fill-reducing ordering, (ii) symbolic factorization, (iii) numerical factorization, and (iv) triangular solution. George and Liu [13] provide a comprehensive discussion on these four steps and their implementations on serial machines. We focus our attention on the design of efficient parallel algorithms for the numerical factorization and the triangular solution on distributed-memory multiprocessors. We assume that the symbolic processing steps including ordering and symbolic factorization are done sequentially.

Significant effort has been invested on designing efficient parallel algorithms for solving sparse symmetric positive definite systems on distributed-memory multiprocessors. Heath, Ng and Peyton [17] have recently published a survey on parallel sparse matrix factorization algorithms. There are roughly three classes of algorithms for computing sparse Cholesky factorization on distributed-memory machines—fan-out algorithm [12], fan-in algorithm [2] and multifrontal algorithm [8]. Ashcraft, Eisenstat, Liu and Sherman [3] have reported that the multifrontal method is more efficient than both fan-out and fan-in algorithms on an iPSC/2 machine for the 63×63 nine-point regular grid.

Pothen and Sun [25] have designed and implemented a parallel multifrontal algorithm for computing the sparse Cholesky factorization on distributed-memory multiprocessors. The clique tree data structure is used to organize the numerical factorization. The overall

* Advanced Computing Research Institute, Cornell Theory Center, Cornell University, Ithaca, NY 14853-3801 (csun@cs.cornell.edu).

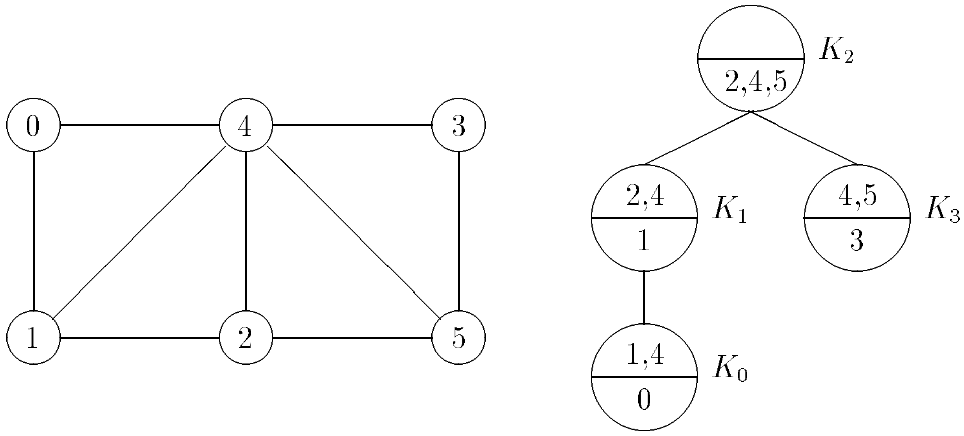

FIG. 1. *The adjacency graph of a Cholesky factor and a corresponding clique tree.*

computation is decomposed into a sequence of operations on a set of dense submatrices. A crucial issue is the partial factorization of a frontal matrix on a subset of processors.

Our major contributions in this work are described as follows. First, we seek to improve the performance of a distributed multifrontal method described in [25]. A new algorithm is designed for computing the partial factorization of a frontal matrix on a subset of processors. More than 180 million flops per second during the numerical factorization is achieved for a three-dimensional twenty-seven point grid problem on an iPSC/860 with 32 processors. Second, we present new distributed algorithms for solving the sparse triangular systems $Lu = Pb$ and $L^T v = u$. The clique or supernode structure of L that is crucial to the efficiency of the numerical factorization is fully exploited in the triangular solution. The solution of a sparse triangular system is decomposed into a sequence of solutions of dense triangular systems according to the clique tree structure of L. The sequence of dense triangular systems is solved in a multifrontal manner. Numerical experiments on a set of model and practical problems with number of equations ranging from 2,003 to 24,389 are described.

Our paper is organized as follows. A multifrontal Cholesky factorization is briefly reviewed in Section 2. In Section 3, a new parallel algorithm for computing the partial factorization of a frontal matrix is described, and its efficiency is demonstrated. In Section 4, new parallel algorithms for computing sparse forward elimination and sparse backward substitution and their experimental results are presented. Concluding remarks are contained in Section 5.

2. A multifrontal method. We assume that the reader is familiar with the graph-theoretic concept of sparse Cholesky factorization [13]. The maximal cliques of the adjacency graph of a Cholesky factor L can be organized into a tree structure called *clique tree*. A discussion of clique trees and their applications in sparse matrix computations may be found in [18, 22, 24]. A maximal clique is referred to as a clique for brevity. The adjacency graph of a 6×6 Cholesky factor and a corresponding clique tree are shown in Fig. 1, where each clique is shown as a circle, and the vertices of the clique are listed in the circle.

Each clique K is partitioned into two sets—K^f and K^u, where the former is shown below the line in the circle representing K and the latter above the line in the circle. The vertices in K^f are numbered lower than the vertices in K^u. Let the vertices of clique K be $i_0, \cdots, i_{n-1}$ in ascending order. Let K^f contain the first $m \leq n$ vertices of clique K and K^u contain the remaining $n-m$ vertices of clique K. Then clique K is written as

$$K = \{i_0, \cdots, i_{m-1}; i_m, \cdots, i_{n-1}\}.$$

Clique K corresponds to an $n \times n$ dense triangular matrix in L consisting of elements $l_{i,j}$ such that $j \leq i$ and $\{i, j\} \subset K$. This dense matrix is called the *frontal matrix* of clique K and is denoted by F_K. Note that F_K is the lower triangle of a dense symmetric positive definite matrix. The first m columns of F_K corresponding to vertices in K^f form a dense trapezoidal matrix called the *factor matrix* of clique K and denoted by T_K. The last $n-m$ columns of F_K corresponding to vertices in K^u form a dense $(n-m) \times (n-m)$ triangular matrix called the *update matrix* of clique K and denoted by U_K. Columns in T_K and columns in U_K are referred to as *factor columns* and *update columns* of clique K, respectively. The computation associated with clique K is a partial Cholesky factorization of F_K which eliminates its factor columns.

Now consider the example in Fig. 1. Let $K_0 = \{0; 1, 4\}$, $K_1 = \{1; 2, 4\}$, $K_2 = \{2, 4, 5\}$ and $K_3 = \{3; 4, 5\}$. Let F_i denote the frontal matrix and U_i the update matrix of clique K_i for $0 \leq i \leq 3$. The factorization of F_0 is illustrated as

$$\begin{pmatrix} a_{00} & & \\ a_{10} & 0.0 & \\ a_{40} & 0.0 & 0.0 \end{pmatrix} \Longrightarrow \begin{pmatrix} l_{00} & & \\ l_{10} & u_{11}^0 & \\ l_{40} & u_{41}^0 & u_{44}^0 \end{pmatrix}.$$

The first column of factored F_0 is the first column in L, and the last two columns of factored F_0 form the update matrix U_0 representing the updates on column 1 and column 4 by eliminating column 0. The update on matrix F_1 by U_0 is illustrated as

$$\begin{pmatrix} u_{11}^0 & \\ u_{41}^0 & u_{44}^0 \end{pmatrix} \bigoplus \begin{pmatrix} a_{11} & & \\ a_{21} & 0.0 & \\ a_{41} & 0.0 & 0.0 \end{pmatrix} \Longrightarrow \begin{pmatrix} a_{11} + u_{11}^0 & & \\ a_{21} & 0.0 & \\ a_{41} + u_{41}^0 & 0.0 & u_{44}^0 \end{pmatrix}.$$

The updated frontal matrix F_1 is partially factored to obtain

$$\begin{pmatrix} l_{11} & & \\ l_{21} & u_{22}^1 & \\ l_{41} & u_{42}^1 & u_{44}^1 \end{pmatrix}.$$

The first column $(l_{11}, l_{21}, l_{41})^T$ is the second column in L. The last two columns form U_1 representing the accumulated updates on column 2 and column 4 by eliminating column 0 and column 1. Similarly, the frontal matrix F_3 can be partially factored and the factorization is independent of the computation associated with either F_0 or F_1. Then the frontal matrix F_2 is updated by matrices U_1 and U_3 and factored to complete the overall factorization.

The process of updating a matrix F by another matrix U is also referred to as assembly of the matrix U into the matrix F. In the example above, the factorization of F_1 can not be started until the factorization of F_0 is finished and U_0 is assembled into F_1. The factorization of F_3 is, however, independent of the factorization of F_0 or F_1. The clique tree structure clearly shows data dependency and parallelism in the numerical factorization.

3. Distributed numerical factorization.

3.1. The original algorithm. We assume that the processors in a distributed-memory multiprocessor system are numbered as $0, 1, \cdots, \rho - 1$. The distributed multifrontal Cholesky factorization method described in [25] is briefly reviewed here. The cliques on a clique tree are mapped by a proportional mapping scheme [25] which is shown to be more efficient than the bin-pack mapping scheme [10] or the subtree-to-subcube mapping scheme [15]. A subtree T with root clique K is allocated a set S of consecutively numbered processors. The clique K is partitioned among the processors in S—i.e., the frontal matrix F_K of clique K is wrapped around the processors in S. Let the columns of F_K be numbered as $0, \cdots, n-1$ and let $S = \{l, l+1, \cdots, l+q-1\}$. Column j of F_K is mapped to processor $l + j \bmod q$. The subtree rooted at a child clique of K is allocated a set of consecutively numbered processors which is a subset of S, and the number of processors allocated to the subtree is proportional to the arithmetic work associated with the subtree. This process is recursively applied to each subtree rooted at a child of K. If $|S| = 1$, then the subtree with root K is entirely mapped to the only processor in S—i.e., all cliques in the subtree with root K are mapped to the single processor in S.

The numerical values of L are distributed among processors according to the mapping of the cliques. The factor matrices of all cliques are initialized with the numerical values of matrix A. If a clique K is entirely mapped to a processor p, the factor matrix of clique K is allocated to p. If a clique K is partitioned among a set of processors, the factor columns of clique K mapped to a processor p are allocated to p. During numerical factorization, a processor p processes cliques on the clique tree in postorder ordering. If a clique K is entirely mapped to the processor p, p assembles the update matrices of the children of K into F_K and then computes the partial factorization of F_K. If a clique K is partitioned among a set of processors, the assembly of F_K and the partial factorization of F_K are performed in parallel. The original algorithm in [25] for computing the partial factorization of F_K on a processor p is shown in Fig. 2. Details on the overall distributed multifrontal algorithm can be found in [25].

---

for each column j in K^f in increasing order **do**
    if column j is mapped to p **then**
        compute column j;
        send column j to other processors in S;
    else
        receive column j;
    end if
    update unfinished columns of K mapped to p by column j;
end for

FIG. 2. *The original algorithm for distributed partial factorization of a dense submatrix.*

3.2. A new distributed algorithm for factoring a frontal matrix. We propose a new algorithm for computing the partial Cholesky factorization of a dense matrix on a subset of processors in a distributed-memory multiprocessor system. This algorithm is applicable to any multifrontal method in which the clique or supernode structure of L is exploited. Let F be a frontal matrix with n columns. The columns of matrix F is numbered as $0, 1, ..., n-1$. Matrix F is wrapped around a set of processors $S = \{l, l+1, .., l+q-1\}$ by columns. Column j of matrix F is mapped to processor $l + j \bmod q$. The partial Cholesky factorization of matrix F is to compute the first m columns of the Cholesky factor of F, and accumulate the effects on the remaining $n-m$ columns. A processor requires a column j if it owns columns numbered higher than j and not yet updated by column j. The new algorithm sends a completed column to other processors which require the column as early as possible. The details of this algorithm for processor p are described in Fig. 3. The factor columns mapped to a processor p are stored contiguously in a one-dimensional array which holds the portion of L mapped to p, and the update columns mapped to p are stored in temporary storage. The column updates at lines 6, 12, 16, 20 and 24 in Fig. 3 are implemented as `daxpy` operations. The scaling operations at lines 4 and 13 are implemented as `dscal` operations.

3.3. Experimental results. The implementation of a multifrontal method described in [25] is referred to as the original implementation. The implementation obtained from the original implementation by replacing the original algorithm for distributed factorization of a partitioned frontal matrix shown in Fig. 2 with the new algorithm in Fig. 3 is referred to as the new implementation. We experiment with a set of problems with both regular and irregular sparsity structure. The problems with regular sparsity structure include two-dimensional(2D) $k \times k$ nine-point grid and three-dimensional(3D) $k \times k \times k$ twenty-seven point grid. The 2D grids have often been used in the literature to demonstrate the performance of sparse Cholesky factorization algorithms. The 3D grids not only form a class of model problems but also have practical applications. One application of 3D grids is in the numerical solution of sparse least-square problems arising from the particle method for turbulent combustion [6, 23]. Test problems CN28CN, TPAS and TECT3 arise from numerical modelling of some biomechanical systems [4]. NASA4704 is from the NASA Ames collection of sparse matrices. The remaining test problems are from the Harwell-Boeing collection of sparse matrices [7].

All grid problems are ordered by the nested-dissection ordering [11]. CN28CN is ordered by the automatic nested-dissection ordering subroutine from SPARSPAK [5]. All other problems are ordered by minimum degree ordering [14]. The characteristics of the test problems are shown in Table 1, where N is the number of equation, $|L|$ is the number of nonzeros in the sparse Cholesky factor, "fact_flops" denotes number of mega flops for numerical factorization and "tri_flops" denotes number of mega flops for triangular solution including both forward elimination and backward substitution. A flop is either a multiplicative operation or an additive operation. The notation k-2D represents a $k \times k$ nine-point grid while k-3D represents a $k \times k \times k$ twenty-seven point grid.

Experiments are done on an Intel iPSC/860 with 32 RX nodes and 8 megabytes of memory per node, and an Intel iPSC/2 with 32 CX nodes and 4 megabytes of memory per node. All programs are written in C, and double precision arithmetic is used. On

```
j = 0;
while (j < m) do
    if column j is mapped to p then
        scale column j;
        send column j to processors in S \ {p} which require column j;
        update columns in {j + 2, ..., n − 1} mapped to p by column j;
    else
        if at least one column in {j + 1, ..., n − 1} is mapped to p then
            receive column j;
            if j + 1 < m then
                if column j + 1 is mapped to p then
                    update column j + 1 by column j;
                    scale column j + 1;
                    send column j + 1 to processors in S \ {p} which
                        require column j + 1;
                    update columns in {j + 3, ..., n − 1} mapped to p
                        by column j and column j + 1;
                    j = j + 1;
                else
                    update columns in {j + 2, ..., n − 1} mapped to p
                        by column j;
                end if
            else
                update columns in {j + 1, ..., n − 1} mapped to p
                    by column j;
            end if
        end if
    end if
    j = j + 1;
end while
```

FIG. 3. *Distributed partial factorization of a dense submatrix.*

TABLE 1
Characteristics of the test problems.

problems	N	$\|L\|$	fact_flops	tri_flops
100-2D	10,000	297,953	18.29	1.33
16-3D	4,096	487,600	95.93	2.03
21-3D	9,261	1,610,247	517.12	6.70
25-3D	15,625	3,359,861	1495.72	13.90
28-3D	21,952	5,445,708	2975.83	22.35
29-3D	24,389	6,345,879	3707.73	26.00
CN28CN	7,846	4,252,978	3386.75	17.18
TPAS	4,682	1,326,780	476.84	5.39
TECT3	3,584	778,275	213.46	3.18
NASA4704	4704	281472	35.73	1.19
BCSSTK13	2003	271671	59.72	1.14
BCSSTK15	3,948	651,222	167.39	2.72
BCSSTK18	11,948	662,725	143.69	2.85
BCSSTK23	3,134	420,311	121.19	1.77
BCSSTK25	15,439	1,416,568	288.98	6.01
BCSSTK33	8,738	2,546,802	1213.68	10.42

the iPSC/860 machine, compiler optimization level -O4 and the `blas` subroutines `daxpy`, `ddot` and `dscal` from the Basic Math Library are used. On the iPSC/2 machine, compiler optimization level -O is used. All programs run on the iPSC/2 are plain C programs.

For each problem we report performance results on 1, 2, 4, 8, 16 and 32 processors provided that memory space is sufficient. All running times are in seconds. The experimental results are shown in Table 2, Table 3 and Table 4, where "f_t" denotes numerical factorization time in the new implementation, "of_t" numerical factorization time in the original implementation, "fe_t" sparse forward elimination time, "bs_t" sparse backward substitution time, "ts_t" time for sparse triangular solution including both forward elimination and backward substitution, "ratio" the ratio of the triangular solution time to the numerical factorization time in the new implementation in percentage, "f_mf" mega flop rate achieved during the numerical factorization phase of the new implementation. The running time for each phase on a set of processors is the maximum time used on a processor in that set. The parallel sparse triangular solution will be described in Section 4. Note that "ts_t" is not equal to the sum of "fe_t" and "bs_t". Before the forward elimination begins, the right hand side vector b is permuted and put into appropriate data structure used in the triangular solution. After the backward substitution, the solution vector is extracted from the data structure used in the triangular solution. Since "fe_t" and "bs_t" represent maximum running times, they may not occur on the same processor. Therefore, "ts_t" could be smaller than the sum of "fe_t" and "bs_t" as shown in Table 3 for the problem BCSSTK13 when 32 processors are used.

On the iPSC/860 machine, the new implementation works especially well for a large number of processors. For less than or equal to 4 processors, the original implementation

TABLE 2
Running times on an iPSC/860 (I).

problem	ρ	f_t	of_t	fe_t	bs_t	ts_t	ratio	f_mf
100-2D	1	3.833	3.833	0.323	0.376	0.716	18.7	4.77
	2	2.004	2.127	0.178	0.204	0.397	19.8	9.13
	4	1.124	1.202	0.109	0.122	0.244	21.7	16.28
	8	0.772	0.879	0.079	0.090	0.181	23.4	23.68
	16	0.577	0.829	0.080	0.094	0.186	32.3	31.73
	32	0.470	1.096	0.096	0.128	0.231	49.2	38.94
16-3D	1	11.047	11.047	0.245	0.207	0.458	4.1	8.68
	2	6.291	6.505	0.152	0.147	0.305	4.9	15.25
	4	3.831	4.065	0.112	0.108	0.226	5.9	25.04
	8	2.489	2.980	0.097	0.104	0.206	8.3	38.54
	16	1.781	3.131	0.135	0.158	0.296	16.6	53.87
	32	1.529	4.714	0.183	0.233	0.414	27.0	62.75
21-3D	4	15.304	15.858	0.263	0.242	0.518	3.4	33.79
	8	8.968	10.266	0.193	0.200	0.404	4.5	57.66
	16	6.160	9.545	0.242	0.273	0.526	8.5	83.95
	32	4.383	12.357	0.303	0.372	0.683	15.6	117.98
25-3D	16	14.844	21.301	0.370	0.408	0.797	5.4	100.77
	32	9.526	24.864	0.447	0.536	1.002	10.5	157.02
28-3D	32	17.666	42.584	0.593	0.695	1.318	7.5	168.45
29-3D	32	20.505	47.324	0.632	0.740	1.394	6.8	180.82
CN28CN	32	23.138	75.494	0.873	1.064	1.944	8.4	146.37
TPAS	8	13.898	16.327	0.294	0.421	0.722	5.2	34.31
	16	9.849	17.824	0.454	0.593	1.050	10.7	48.42
	32	7.524	25.630	0.690	0.893	1.586	21.1	63.37
TECT3	2	12.217	12.386	0.204	0.227	0.436	3.6	17.47
	4	8.931	9.450	0.173	0.165	0.342	3.8	23.90
	8	6.125	7.458	0.180	0.229	0.413	6.7	34.85
	16	5.067	9.181	0.284	0.355	0.643	12.7	42.13
	32	4.255	13.351	0.411	0.528	0.935	22.0	50.17

Table 3
Running times on an iPSC/860 (II).

problem	ρ	f_t	of_t	fe_t	bs_t	ts_t	ratio	f_mf
NASA4704	1	4.657	4.657	0.178	0.166	0.352	7.6	7.67
	2	3.193	3.269	0.139	0.145	0.291	9.1	11.19
	4	2.138	2.299	0.107	0.119	0.232	10.9	16.71
	8	1.617	2.191	0.110	0.136	0.252	15.6	22.09
	16	1.301	2.705	0.165	0.242	0.411	31.6	27.47
	32	1.186	4.378	0.211	0.299	0.515	43.4	30.12
BCSSTK13	1	6.748	6.748	0.135	0.108	0.246	3.6	8.85
	2	4.006	4.118	0.107	0.130	0.240	6.0	14.91
	4	3.385	3.677	0.113	0.125	0.240	7.1	17.64
	8	2.673	3.427	0.115	0.139	0.258	9.6	22.34
	16	2.321	4.204	0.179	0.237	0.418	18.0	25.73
	32	1.826	6.738	0.243	0.321	0.559	30.6	32.71
BCSSTK15	2	9.571	9.670	0.198	0.183	0.388	4.1	17.49
	4	6.966	7.360	0.187	0.195	0.389	5.6	24.03
	8	5.685	6.851	0.204	0.249	0.458	8.1	29.44
	16	4.566	7.637	0.279	0.346	0.628	13.8	36.66
	32	2.955	9.905	0.366	0.538	0.908	30.7	56.65
BCSSTK18	2	10.454	10.584	0.307	0.399	0.728	7.0	13.74
	4	8.279	8.869	0.285	0.351	0.655	7.9	17.36
	8	5.406	6.585	0.224	0.299	0.540	10.0	26.58
	16	4.194	7.364	0.294	0.384	0.696	16.6	34.26
	32	3.217	10.750	0.396	0.519	0.934	29.0	44.67
BCSSTK23	1	13.013	13.013	0.221	0.186	0.413	3.2	9.31
	2	7.821	8.020	0.168	0.212	0.386	4.9	15.50
	4	6.424	6.937	0.168	0.183	0.357	5.6	18.87
	8	4.716	5.959	0.150	0.196	0.351	7.4	25.70
	16	3.644	6.989	0.245	0.349	0.597	16.4	33.26
	32	3.017	10.801	0.321	0.482	0.803	26.6	40.17
BCSSTK25	16	11.015	15.625	0.680	1.024	1.731	15.7	26.23
	32	8.757	19.757	0.875	1.394	2.293	26.2	33.00
BCSSTK33	32	14.570	49.793	0.965	1.302	2.278	15.6	83.30

TABLE 4
Running times on an iPSC/2.

problem	ρ	f_t	of_t	fe_t	bs_t	ts_t	ratio	f_mf
100-2D	2	72.621	72.718	3.121	2.981	6.373	8.8	0.25
	4	36.977	37.207	1.638	1.569	3.443	9.3	0.49
	8	21.105	21.557	0.931	0.914	2.073	9.8	0.87
	16	11.578	12.394	0.615	0.636	1.462	12.6	1.58
	32	6.483	7.913	0.629	0.710	1.533	23.6	2.82
NASA4704	2	157.335	158.572	2.837	2.759	5.716	3.6	0.23
	4	86.943	88.152	1.722	1.690	3.519	4.0	0.41
	8	40.640	41.987	0.957	1.007	2.073	5.1	0.88
	16	25.111	27.530	0.848	0.919	1.872	7.5	1.42
	32	10.526	16.554	0.926	1.082	2.080	19.8	3.39
BCSSTK13	2	238.822	240.565	2.405	2.448	4.908	2.1	0.25
	4	139.994	142.672	1.670	1.649	3.370	2.4	0.43
	8	71.945	75.156	1.101	1.112	2.264	3.1	0.83
	16	38.031	42.865	0.949	1.034	2.035	5.4	1.57
	32	20.687	27.089	1.001	1.164	2.193	10.6	2.89

and the new implementation show small difference; however, for more than 4 processors, the new implementation is significantly better. For 32 processors, the new implementation is almost three times as fast as the original one.

We experiment with three smaller problems on the iPSC/2. The new implementation is better as shown in Table 4. However, the performance improvement is not as dramatic as that on iPSC/860. The *speedup* for ρ processors is defined to be the ratio of the running time on a single processor to the running time on ρ processors. The speedups of the new implementation for problems 100-2D, NASA4704, BCSSTK13 on 32 processors are 22.4, 29.9 and 23.1, respectively, where the single processor time is estimated as twice the running time on two processors.

In general, the more the arithmetic work, the better the performance on both iPSC/860 and iPSC/2. When more arithmetic work is involved, the impact of the communication overhead on the overall computation is reduced. Dunigan [9] has studied the performance of Intel iPSC/2 and Intel iPSC/860. The ratio of communication speed to computation speed is calculated using the 8-byte transfer and multiply times, where the 8-byte transfer time is based on the 8-byte one-hop echo time. This communication to computation ratio is 59 on iPSC/2 and 1000 on iPSC/860. This suggests that communication overhead on iPSC/860 is much more significant than that on iPSC/2.

The numerical factorization on each processor consists of two phases. In the first phase, a processor deals with cliques mapped entirely to it and no communication is involved. In the second phase, a group of processors cooperate to perform the distributed assembly and factorization of a frontal matrix by communication. Since the communication to computation ratio on iPSC/860 is much higher, the communication overhead in the second phase takes a larger share of the overall running time. Therefore, the speedup on the iPSC/2 is much higher than that on the iPSC/860. For the same reason, the distributed phase plays much more significant role on iPSC/860 than on iPSC/2. Since the new algorithm only improves the performance of the distributed phase, the performance improvement on iPSC/860 is much more dramatic than that on iPSC/2.

The efficiency of our approach lies in the fact that the overall sparse factorization is decomposed into a sequence of operations on a set of dense submatrices. The vectorization capability on a RX node of an Intel iPSC/860 is fully exploited. However, considerable amount of temporary storage space is needed during the numerical factorization by the multifrontal method for storing the temporary update matrices. The memory space used by CN28CN and 29-3D on 32 processors during numerical factorization is given in Table 5, where "min_val" denotes the minimum number of values of L allocated to a processor, "max_val" the maximum number of values of L allocated to a processor, "min_s" the minimum number of temporary floating-point spaces used on a processor, and "max_s" the maximum number of temporary floating-point spaces used on a processor.

TABLE 5
Storage used in numerical factorization.

problems	min_val	max_val	min_s	max_s
$29 \times 29 \times 29$	158,071	259,778	95,408	218,934
CN28CN	93,978	226,297	75,657	361,408

Evaluation of a sparse matrix algorithm is different from that of a dense matrix algorithm. The mega flop rate doesn't tell the whole story. The number of nonzero entries in L which largely determines the arithmetic work required must be taken into consideration. For example, if problem BCSSTK33 is ordered by the automatic nested-dissection ordering from SPARSPAK [5], the number of nonzeros in L is 3,188,240, and the number of mega flops required by the numerical factorization is about 1909.67. When it is run on 32 processors, the time for numerical factorization is 18.187 seconds, and the mega flop rate is about 105.00. Even though the mega flop rate is higher than that of using the minimum degree ordering, the running time actually gets worse since more fills in L are created and the number of flops required is more than that required by the minimum degree ordering.

4. Parallel sparse triangular solution.

4.1. Introduction. We propose new algorithms for solving the two sparse triangular systems $Lu = Pb$ and $L^T v = u$ on distributed-memory multiprocessors. Solving dense triangular systems efficiently on distributed-memory machines is a challenging task because of the sequential data dependency and the small amount of work to be distributed among processors. It is even more challenging to solve sparse triangular systems efficiently on distributed-memory machines. The number of flops involved in solving a sparse triangular system $Lu = Pb$ or $L^T v = u$ is relatively low — about twice the number of nonzeros in L. The sparsity consideration makes the complicated techniques used in the distributed dense triangular solution algorithms difficult to be applied to solving sparse triangular systems. Solving sparse triangular systems with reasonable efficiency on distributed-memory multiprocessors is necessary for the following reasons. First, the entire L can not be stored on a single processor for large problems. Second, a sparse system $Ax = b$ may have multiple right hand sides. Therefore, repeated sparse triangular solutions are needed. Third, inefficient sparse triangular solution will significantly degrade the performance of the overall solution process, especially for a large number of processors.

After the numerical factorization phase, the Cholesky factor L is distributed among all processors. The algorithmic structure of the triangular solution is largely dictated by the storage scheme used for storing L. Geroge, Heath, Liu and Ng [12] discussed the distributed sparse triangular solution in the context of using a fan-out algorithm to compute sparse Cholesky factorization. The columns of L are mapped to processors by using subtree-to-subcube mapping. Zmijewski [27] considered the use of cyclic algorithms employed in the dense triangular solution algorithms for solving sparse triangular systems. Other parallel sparse triangular solution algorithms [1, 16, 21, 26] have been proposed in the context of preconditioned conjugate gradient method. We present a *parallel multifrontal triangular solution* algorithm for computing the sparse forward elimination and sparse backward substitution in the context of solving $Ax = b$. In our approach, a sparse forward elimination or backward substitution is decomposed into a sequence of solutions of smaller dense trapezoidal systems. The solutions of these dense trapezoidal systems are done efficiently in a multifrontal fashion. The vectorization capability of node processors on distributed-memory machines such as the Intel iPSC/860 machine is exploited in solving the dense trapezoidal systems.

Let a clique $K = \{i_0, \cdots, i_{m-1}; i_m, \cdots, i_{n-1}\}$. Variables $x_{i_0}, x_{i_1}, \cdots, x_{i_{n-1}}$ are referred to as the variables of clique K. Variables $x_{i_0}, \cdots, x_{i_{m-1}}$ are referred to as the *factor* variables of clique K, and variables $x_{i_m}, \cdots, x_{i_{n-1}}$ the *update* variables of clique K. Columns of K are columns of the frontal matrix of K. The right hand side vector b in $Ax = b$ is permuted according to the fill-reducing ordering. Every processor has a copy of the vector b. The mapping of cliques in the triangular solution is the same as the mapping of cliques in the numerical factorization. In another word, Cholesky factor L in triangular solution is stored in exactly the same way as in numerical factorization.

An *index sequence* is defined as a subset of $\{0, 1, ..., N-1\}$ in increasing order, where N is the order of matrix A. Let $I = \{i_0, \cdots, i_{k-1}\}$ and $J = \{j_0, \cdots, j_{n-1}\}$ are index sequences, and I is a subset of J. The *relative index sequence* of I with respect to J is defined as an index sequence $R(I, J) = \{r_0, \cdots, r_{k-1}\}$, such that $r_s = t$ if i_s is the t-th element in J for $0 \leq s < k$. A *sparse vector* of length k consists of a dense vector of length k and an index sequence of length k such that the j-th component of the dense vector has the j-th component of the index sequence as its index. Let a be a sparse vector of length k with index sequence I. Let c be a sparse vector of length n with index sequence J. I and J are defined as above. The *scatter and add* operation of vector a with respect to vector c is defined as

$$c = c + scatter(a, J),$$

where $c[r_j] = c[r_j] + a[j]$ for $0 \leq j < k$, and $c[l]$ is unchanged if $l \notin R(I, J)$. The *gather* operation of vector a with respect to vector c is defined as

$$a = gather(c, I),$$

where $a[j] = c[r_j]$ for $0 \leq j < k$.

4.2. Sparse forward elimination. Let a clique $K = \{i_0, \cdots, i_{m-1}; i_m, \cdots, i_{n-1}\}$. Let $first[K]$ be the processor to which column 0 of K is mapped and $last[K]$ be the processor to which column $m-1$ of K is mapped. The lower trapezoidal system associated with clique K is described as

$$Tz = d,$$

where T is the factor matrix of K, z is a vector of length m to be determined, and d is a vector of length n. Partition T and d as

$$T = \begin{pmatrix} T_1 \\ T_2 \end{pmatrix} \text{ and } d = \begin{pmatrix} d_1 \\ d_2 \end{pmatrix},$$

where T_1 is an $m \times m$ triangular matrix and d_1 contains the first m components of vector d. The vector d_1 is initialized with the appropriate values from the right hand side vector b—i.e., $d[j] = b[i_j]$ for $0 \leq j < m$. The vector d_2 is initialized to zero. The forward elimination of the trapezoidal system $Tz = d$ can be conveniently described as

$$T_1 z = d_1 \text{ and } d_2 = d_2 - T_2 z.$$

The vector d_2 is referred to as the *effect vector* of clique K and is denoted by $e[K]$. The effect vector represents the accumulated effects on the update variables of K by eliminating all factor variables in the subtree rooted at clique K. If K is an interior clique, the effect vectors from all its children are processed before the associated trapezoidal system of clique K is solved. Let C be a child of K. The effect vector from the child C is added to vector d by a scatter and add operation defined earlier:

$$d = d + scatter(e[C], K).$$

The sparse forward elimination deals with all cliques on the clique tree in postorder ordering. For each clique K the factor matrix of K is stored as in the numerical factorization. The right hand side vector of clique K is initialized by the appropriate components of the vector b. For each child clique C of K the effect vector of C is scattered and added to the right hand side vector of clique K. Then the lower trapezoidal system associated with clique K is solved, and the effect vector of K will be used to update the right hand side vector of the parent clique of K. The details of the parallel multifrontal sparse forward elimination algorithm is described in Fig. 4, where M is the number of cliques in the clique tree of L and $myname$ is a processor under consideration.

The most important task in the parallel forward elimination algorithm is to perform the forward elimination of the trapezoidal system of a clique in parallel. Consider the solution of a trapezoidal system $Tz = d$ associated with a clique K. The factor matrix T is wrapped around a set of processors. The algorithm is a modified version of the cyclic algorithm proposed by Li and Coleman [19] for solving dense triangular systems on distributed-memory multiprocessors. In our context, we need to consider the following constraining factors. (i). In general the set of processors on which T is wrapped doesn't form a subcube. (ii). Solution of a trapezoidal system of a clique K is related to the solution of the trapezoidal systems of the parent of clique K and the children of clique K.

The algorithm for the parallel solution of lower trapezoidal system $Tz = d$ associated with a clique K is described in Fig. 5. The notation $v(i : j)$ represents a portion of vector v consisting of the components $v[i], \cdots, v[j]$. Vector addition and vector-scalar multiplication are easily recognizable in the algorithm. We use n to denote the number of rows in the factor matrix T, m number of columns in T. The columns of T are numbered as $0, \cdots, m-1$. They are wrapped around a set of processors $S = \{low, \cdots, high\}$—i.e., column j of T is mapped to processor $low + j \bmod q$, where $q = high - low + 1$. Let $first$ be the processor to which column 0 of T is mapped. Let $last$ be the processor to which column $m-1$ of T is mapped. For a column $j > 0$, $prev$ is the processor to which column $j-1$ is mapped. For a column $j < m-1$, $next$ is the processor to which column $j+1$ is mapped. The processors $low, \cdots, high$ are considered as a ring of processors. The vector $sum(0 : q-1)$ is cycled through the ring to pass information among processors. The vector $psum(0 : n-1)$ maintains the partial sum on the variables of clique K. The one-dimensional array $a(0 : size-1)$ on a processor stores the columns of T mapped to the processor column by column, where $size$ is the total number of elements in T mapped to the processor. The array $b(0 : m-1)$ stores the right hand side vector d. The array $temp(0 : n-m-1)$ is a temporary vector for receiving incoming messages. The update on the vector $psum$ is implemented as a `daxpy` operation. On input, the right hand side vector d is stored in $b(0 : m-1)$ on processor low. On output, a processor stores z_j in $b[j]$ if column j is mapped to the processor for $0 \leq j < m$. The effect vector of K is stored in $psum(m : n-1)$ on processor $last$.

4.3. Sparse backward substitution. Let clique K, $first[K]$ and $last[K]$ be defined as before. The upper trapezoidal system associated with clique K is described as

$$T^t y = z,$$

where T is the factor matrix of K, y is the solution vector of clique K with length n, and z is the right hand side vector of length m. The last $n-m$ elements of y are computed after the upper trapezoidal system associated with the parent clique of K is solved. The first m elements of y are to be determined. The vector z is the solution vector computed during the forward elimination of the lower trapezoidal system of clique K. If K is not a root clique, the solutions to the update variables of clique K are obtained from the solution vector of clique $J = parent[K]$ before the associated upper trapezoidal system is solved. This is accomplished by a gather operation introduced earlier:

$$y_2 = gather(y_J, K),$$

where y_2 is a vector containing the last $n-m$ components of y and y_J is the solution vector of clique J. The sparse backward substitution deals with the cliques on the clique tree in the reverse order of the postorder ordering. For a clique K under consideration, gather the solutions to the update variables of clique K from the solutions to the variables of the parent clique of clique K, and then solve the upper trapezoidal system of clique K, and obtain the solutions to the factor variables of clique K. The details of the parallel multifrontal sparse backward substitution algorithm are described in Fig. 6.

```
for K = 0 to M − 1 do
    n = |K| and m = |K^f|;
    initialize the first m elements of the right hand side vector d of K
        by the appropriate elements of vector b;
    initialize the last n − m elements of d to zero;
    if K is not partitioned then
        if K is mapped to myname then
            for each child clique C of K do
                scatter and add the effect vector of C to d;
            solve the lower trapezoidal system of K and obtain the effect vector of K;
            if K is not a root clique then
                J = parent[K];
                if myname ≠ first[J] then send the effect vector of K to first[J];
            end if
        end if
    else
        if at least one column of the factor matrix of K is mapped to myname then
            if myname = first[K] then
                for each child clique C of clique K do
                    if myname ≠ last[C] then receive the effect vector of C;
                    scatter and add the effect vector of C to d;
                end for
            end if
            solve the lower trapezoidal system of K in parallel,
                and store the effect vector of K in last[K];
            if myname = last[K] then
                if K is not a root clique then
                    J = parent[K];
                    if myname ≠ first[J] then send the effect vector of K to first[J];
                end if
            end if
        end if
    end if
end for
```

FIG. 4. *The parallel multifrontal sparse forward elimination algorithm.*

```
if myname = low then
    sum(0 : q − 2) = b(0 : q − 2) + psum(0 : q − 2);
    sum[q − 1] = 0.0;
    psum(0 : q − 2) = 0.0;
    psum(q − 1, m − 1) = b(q − 1 : m − 1) + psum(q − 1 : m − 1);
else
    sum(0 : q − 2) = 0.0;
    psum(0 : n − 1) = 0.0;
end if

i = 0;
for j = myname − low to m − 1 step q do
    if j > 0 receive a message into sum(0 : q − 1) from prev;
    b[j] = (sum[0] + psum[j])/a[i];
    if j + q > m then k = m − (j + 1) else k = q − 1;
    sum(0 : k − 1) = sum(1 : k) − a(i + 1 : i + k) ∗ b[j] + psum(j + 1 : j + k);
    if j < m − 1 then send sum(0 : q − 1) to next;
    if j + q < m then
        psum(j + q : n − 1) = psum(j + q : n − 1) − a(i + q : i + n − j − 1) ∗ b[j];
    else
        psum(m : n − 1) = psum(m : n − 1) − a(i + m − j : i + n − j − 1) ∗ b[j];
    end if
    i = i + n − j;
end for

if myname is not last then
    send psum(m : n − 1) to last;
else
    for each processor j in S \ {last} do
        receive part of psum vector into temp(0 : n − m − 1) from j;
        psum(m : n − 1) = psum(m : n − 1) + temp(0 : n − m − 1);
    end for
end if
```

FIG. 5. *Parallel solution of a lower trapezoidal system associated with a clique.*

The most important task in the parallel sparse backward substitution algorithm is to perform the backward substitution of the upper trapezoidal system of a clique in parallel. Consider the solution of the trapezoidal system $T^t y = z$, where the factor matrix T of clique K is mapped on a set of processors $\{low, \cdots, high\}$. The columns of T is wrapped around the q processors, where $q = high - low + 1$. Column j of T is mapped to processor $low + j \bmod q$. The algorithm for solving the upper trapezoidal system is a modified version of the cyclic algorithm proposed by Li and Coleman [19]. Again, we need to consider the constraining factors mentioned in the forward elimination.

The algorithm for the parallel solution of upper trapezoidal system $T^t y = z$ associated with a clique K is described in Fig. 7. The notations adopted in Fig. 5 are also used here. The notation $v(i:j) \cdot u(k:l)$ with $j - i = l = k$ represents the inner product of two vectors. The vector $buf(0 : q-1)$ is cycled through the ring of processors to pass information among processors. The one-dimensional array $a(0 : size - 1)$ on a processor stores the columns of T mapped to the processor column by column, where $size$ is the total number of elements in T mapped to the processor. The array $temp(0 : n-1)$ is a temporary vector for receiving incoming messages, and l is the last column of the factor matrix mapped to a processor.

Since the right hand side vector z is the solution vector to the lower trapezoidal system of clique K, it is distributed among the set of q processors. On input, the entry z_j is stored on a processor if column j of T is mapped to the processor for $0 \leq j < m$. The solutions to the update variables of clique K is stored on $last$. On output, a processor stores y_j in $b[j]$ if column j of T is mapped to the processor for $0 \leq j < m$. Processor $first$ has a complete copy of the solutions to the variables of clique K. The inner product is implemented as a `ddot` operation.

4.4. Experimental results. The performance results on the iPSC/860 machine are shown in Table 2 and Table 3. For a grid problem the best time is achieved when about 8 processors are used. For a general problem the best time is achieved when 4 or 8 processors are used. When the number of processors is large, the arithmetic work on each processor is low. The communication overhead degrades the performance significantly. The impact of the triangular solution phase on the overall solution process can be assessed by considering the ratio of the triangular solution time to the numerical factorization time. This ratio is below 30% on 32 processors and below 20% on smaller number of processors for large problems.

The experimental results on the iPSC/2 machine are shown in Table 4. The best time is achieved when 16 processors are used. The ratio of triangular solution time to the numerical factorization time is below or around 20% on 32 processors and below 10% on smaller number of processors. The performance of the triangular solution on iPSC/2 is much better than that on iPSC/860. This again can be explained by the fact that the communication to computation ratio on iPSC/2 is smaller.

5. Concluding remarks. We have described a new algorithm for computing partial Cholesky factorization of a frontal matrix on distributed-memory multiprocessors in the context of multifrontal Cholesky factorization. This algorithm performs significantly better than the algorithm described in [25]. Impressive mega flop rates have been obtained for a collection of model and practical problems on an Intel iPSC/860 machine with 32 processors.

```
for K = M − 1 to 0 do
    if K is not partitioned then
        if K is mapped to myname then
            if K is not a root clique then
                J = parent[K];
                if myname ≠ first[J] then receive the solution vector of clique J;
                gather the solutions to the update variables of clique K
                    from solution vector of clique J;
            end if
            solve the upper trapezoidal system of K,
                and obtain the solution vector of K;
        end if
    else
        if at least one column of the factor matrix of K is mapped to myname then
            if myname = last[K] then
                if myname is not a root clique then
                    J = parent[K];
                    if myname ≠ first[K] then receive the solution vector of J;
                    gather the solutions to the update variables of clique K
                        from the solution vector of clique J;
                end if
            end if
            solve the upper trapezoidal system of K in parallel,
                and obtain the solution vector of K;
            if myname = first[K] then
                for each child clique C of K do
                    if myname ≠ last[C] then send the solution vector of K to last[C];
            end if
        end if
    end if
end for
```

FIG. 6. *The parallel multifrontal sparse backward substitution algorithm.*

for $i = 0$ **to** $m - 1$ **do if** $i \bmod q + low \neq myname$ **then** $b[i] = 0.0$;
if $myname = last$ **then**
    for each processor j in $S \setminus \{last\}$ **do** send $b(m : n-1)$ to j;
else
    receive the second part of b into $b(m : n-1)$ from $last$;
$h = 0$;
for $i = myname - low$ **to** l step q **do**
    $b[i] = a(h + m - i : h + n - i - 1) \cdot b(m : n-1)$, and $h = h + n - i$;
$buf(0 : q-1) = 0.0$;

$r = size$;
for $i = l$ **to** 0 **step** $-q$ **do**
    if $i < m - 1$ receive a message into $buf(0 : q-1)$ from $next$;
    if $i + 1 > m - q$ **then** $k = m - (i+1)$ **else** $k = q - 1$;
    $r = r - n - i$;
    $v = a(r + 1 : r + k) \cdot buf(0 : k-1)$;
    $b[i] = (b[i] - v)/a[r]$;
    for $j = q - 1$ **to** 0 **step** -1 **do** $buf[j] = buf[j-1]$;
    $buf[0] = b[i]$;
    if $i > 0$ **then** send $buf(0 : q-1)$ to $prev$;
    $k = k + 1$, and $h = 0$;
    for $j = myname - low$ **to** $i - q$ **step** q **do**
        $b[j] = b[j] - a(h + i - j : h + i - j + k - 1) \cdot buf(0 : k-1)$, and $h = h + n - j$;
end for

if $myname = low$ **then**
    for $k = low + 1$ **to** $high$ **do**
        if $k = last$ **then**
            receive b into $temp(0 : n-1)$ from $last$;
            $b(0 : m-1) = b(0 : m-1) + temp(0 : m-1)$;
            $b(m : n-1) = temp(m : n-1)$;
        else
            receive b into $temp(0 : m-1)$ from processor k;
            $b(0 : m-1) = b(0 : m-1) + temp(0 : m-1)$;
        end if
else
    if $myname = last$ **then** send $b(0 : n-1)$ to low **else** send $b(0 : m-1)$ to low;

FIG. 7. *Parallel solution of an upper trapezoidal system associated with a clique.*

However, the amount of temporary space required is considerablely high. Techniques for reducing temporary storage will be investigated.

We have described a multifrontal method for computing sparse forward elimination and sparse backward substitution on distributed-memory multiprocessors. The performance of our sparse triangular solution algorithm depends on the efficiency of the parallel solution of the dense trapezoidal systems of the cliques. In our current implementation, ring topology is assumed in solving the dense trapezoidal systems. Whether the more sophisticated techniques developed by Li and Coleman in [20] for dense triangular solutions can be incorporated into our sparse triangular solution algorithm will be investigated.

We believe that evaluation of a parallel sparse system solver need to take both the performance of the numerical factorization phase and the performance of the triangular solution phase into consideration. Sparse triangular solution is difficult to be parallelized due to the small amount of work and the sparsity consideration. However, the performance of the parallel sparse triangular solution is important in a distributed computing environment, and especially so for a large number of processors.

Acknowledgements. This research was partially supported by the Cornell Theory Center, which receives major funding from the National Science Foundation and the IBM corporation, with additional support from New York State and members of its Corporate Research Institute. Numerical experiments were done on an iPSC/860 with 32 nodes and an iPSC/2 with 32 nodes at the Cornell Theory Center. I would like to thank Professor Thomas F. Coleman for many discussions relating to this work and for his helpful comments and suggestions on the manuscript.

REFERENCES

[1] E. Anderson and Y. Saad, *Solving sparse triangular linear systems on parallel computers*, Int. J. High Speed Comput., 1 (1989), pp. 73–95.

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

[3] C. Ashcraft, S. C. Eisenstat, J. W. Liu, and A. H. Sherman, *A comparison of three column-based distributed sparse factorization schemes*, Tech. Report CS-90-09, Department of Computer Science, York University, North York, Ontario, Canada, August 1990.

[4] S. Chinchalker, *Sparse matrix test problems.* Private communication, 1992.

[5] E. C. Chu, J. A. George, J. W. Liu, and E. G. Ng, *User's guide for sparspak-a: Waterloo sparse linear equations package*, Tech. Report CS-84-36, Computer Science, University of Waterloo, Ontario, Canada, 1984.

[6] T. F. Coleman and C. Sun, *Large-scale box-constrained least squares calculations for turbulent combustion.* In preparation, 1992.

[7] I. S. Duff, R. Grimes, and J. Lewis, *Sparse matrix test problems*, ACM Trans. on Math. Software, 15 (1989), pp. 1–14.

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

[9] T. H. Dunigan, *Performance of the Intel iPSC/860 hypercube*, Tech. Report TM-11491, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831, 1990.

[10] G. A. Geist and E. G. Ng, *Task scheduling for parallel sparse Cholesky factorization*, International Journal of Parallel Programming, 18 (1989), pp. 291–314.

[11] J. A. George, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363.

[12] J. A. George, M. T. Heath, J. W. Liu, and E. G. Ng, *Solution of sparse positive definite systems on a hypercube*, J. Comp. Appl. Math, 27 (1989), pp. 129–156.

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

[14] ———, *The evolution of the minimum degree algorithm*, SIAM Review, 31 (1989), pp. 1–19.

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

[16] A. Greenbaum, *Solving sparse triangular systems using fortran with extensions on the nyu ultracomputer prototype*, Tech. Report 99, New York University, New York, NY, April 1986.

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

[18] J. G. Lewis, B. W. Peyton, and A. Pothen, *A fast algorithm for reordering sparse matrices for parallel factorization*, SIAM J. Sci. Stat. Comput., 10 (1989), pp. 1146–1173.

[19] G. Li and T. F. Coleman, *A parallel triangular solver for a hypercube multiprocessor*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 485–502.

[20] ———, *A new method for solving triangular systems on distributed-memory message-passing multiprocessors*, SIAM J. Sci. Stat. Comput., 10 (1989), pp. 382–396.

[21] R. Melhem, *Parallel solution of linear systems with striped sparse matrices*, Parallel Computing, 6 (1988), pp. 165–184.

[22] B. W. Peyton, *Some Applications of Clique Trees to the Solution of Sparse Linear Systems*, PhD thesis, Clemson University, Clemson, SC, 1986.

[23] S. B. Pope, *Particle methods for turbulent combustion*. Private communication, 1992.

[24] A. Pothen and C. Sun, *Compact clique tree data structures in sparse matrix factorizations*, in Large-Scale Numerical Optimization, T. F. Coleman and Y. Li, eds., SIAM, Philadelphia, 1990, pp. 180–204.

[25] ———, *A distributed multifrontal algorithm using clique trees*, Tech. Report CTC91TR72, Cornell Theory Center, Cornell University, Ithaca, NY, August 1991.

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

[27] E. Zmijewski, *Sparse Cholesky Factorization on a Multiprocessor*, PhD thesis, Cornell University, Ithaca, NY, 1987.