NUMERICAL ANALYSIS PROJECT
MANUSCRIPT NA-92-10

AUGUST1992

A Parallel Row-Oriented Sparse Solution Method for Finite Element Structural Analysis

by

Kincho H. Law
and
David R. Mackay

NUMERICAL ANALYSIS PROJECT
COMPUTER SCIENCE DEPARTMENT
STANFORD UNIVERSITY
STANFORD, CALIFORNIA 94305

A Parallel Row-Oriented Sparse Solution Met hod for Finite Element Structural Analysis'

Kincho H. Law and David R Mackay
Department of Civil Engineering
Stanford University
Stanford, CA 94305-4020

Abstract

This paper describes a parallel implementation of LDL^T factorization on a distributed memory parallel computer. Specifically, the parallel LDL^T factorization procedure is based on a row-oriented sparse storage scheme. In addition, a strategy is proposed for the parallel solution of triangular system of equations. The strategy is to compute the inverses the dense principal diagonal block submatrices of the factor L, stored in a row-oriented structure. Experimental results for a number of finite element models are presented to illustrate the effectiveness of the parallel solution schemes.

[1]This work is sponsored by the National Science Foundation grant number ECS-9003107, and the Army Research Office grant number DAAL-03-91-G-0038.

Contents

List of Figures

List of Tables

1 Introduction

Displacement based structural finite element analysis often requires the solution of a linear system of equations:

$$h\text{-}x = f \tag{1}$$

where x and f are, respectively, the displacement and loading vectors. h' is the global stiffness matrix which is often symmetric, positive definite and sparse. To solve the system of linear equations, the symmetric matrix K is often factored into its matrix product LDL^T where L is a unit lower triangular matrix and D is a diagonal matrix. The displacement vector x is then computed by a forward solve, $Lz = f$ or $z = L^{-1} f$, followed by a backward solve, $DL^T x = z$ or $x = L^{-T} D^{-1} z$.

There are three basic forms of the LDL^T factorization, depending on how the lower triangular factor L is being stored and computed:

1. row-oriented form where L is computed and stored by rows;
2. column-oriented form where L is computed and stored by columns;
3. submatrix form where at each step. the remaining submatrix (Schur complement) is updated by the outer-product of a computed column of L.

In sparse matrix factorization, the column-oriented form is perhaps the most commonly used approach [6, 5]. The multifrontal method is an example of factorization in submatrix form [15]. Row-oriented form has traditionally been used primarily for variable bandwidth or profile method [3, 11, 10], and the approach has recently been extended for general sparse factorization [13, 14, 16].

In this paper, we describe a parallel implementation of LDL^T factorization on a distributed memory parallel computer. Current developments in parallel sparse factorization have been reviewed by Heath, Ng and Peyton [9]. Previous studies have been focused on the parallelization of column-oriented (fan-in) and submatrix (such as fan-out and multifrontal methods) forms of factorization which are based on a column-oriented storage scheme. In this paper, we describe a parallel implementation of LDL^T factorization based on a row-oriented storage scheme. The method is based on the recent development in sparse row-oriented factorization, exploiting all possible zeros in the matrix factor [2, 12, 13, 14, 16].

Efficient implementation of the forward and backward triangular solutions on parallel computers is quite difficult because of data dependencies [9]. Alvarado and Schreiber proposed a scheme to partition L into column panels and compute the inverses of the partitioned factors [1]. The idea is to eliminate the data dependencies in the forward and backward solutions and to exploit the parallelism inherent in matrix-vector multiplication. We propose a different scheme by computing the inverses of the principal diagonal block submatrices of L based on a row-oriented storage scheme.

This paper describes the development of a parallel row-oriented factorization method and its implementation on a distributed memory Intel's i860 hypercube computer. The paper is organized as follows: Section 2 reviews an approach for sparse row-oriented factorization [16]. Section 3 describes the load assignment of a sparse matrix on a multiple processing environment. A parallel implementation of the row-oriented solution scheme is described in Section 4. In Section 5, we introduce a strategy that computes the inverses of the principal diagonal block submatrix factors; this approach can significantly speed up the computation time for the forward and backward solution of triangular systems and is particularly useful for problems with multiple right-hand sides. Experimental results are presented in Section 6. Finally, this paper is concluded with a summary and discussion in Section 7.

2 A Row-Oriented Sparse LDL^T Factorization

This section describes the development of a row-oriented LDL^T factorization scheme. We first review the graph-theoretic representation of sparse matrices in Section 2.1. The notion of elimination tree and symbolic factorization is described in Section 2.2. Section 2.3 presents a sparse LDL^T factorization procedure based on a row-oriented storage scheme.

2.1 Graph Theoretic Notations of Sparse Matrices

This section provides a brief review on the graph-theoretic representation of sparse matrices. Detailed discussion on sparse matrix and graph theory can be found in References [4] and [6].

A finite undirected graph G = *(V, E)* consists of a finite set V of n elements, called nodes, and a set E of unordered pairs of distinct nodes (u, v), called edges. When each node is assigned an integer, ranging from 1 to n, which uniquely identifies the number or label of that node, the graph is said to be an ordered graph. A **subgraph** G' = *(V', E')* of G is a graph for which $V' \subseteq V$ and $E' \subseteq E$.

A (simple) path $\{u, \ldots, v\}$, or $Path\{u - v\}$, is a sequence of distinct nodes and contiguous edges leading from u to v such that there are no repeating edges. A cycle is a path that begins and ends at the same node. A tree T = (u, v) is a connected graph with no cycles, A rooted tree is a tree in which one node is distinguished as the "root". For a path {w, . . . , v, . . . , u} from a node w to the root node u via an intermediate node v, v is called an ancestor of w and w a descendant of v. If $(w$, v) is an edge in the rooted tree T, v is termed the parent of w, and w the child of v. A monotone ordering of a rooted tree is one for which each node is numbered before its parent. A **subtree** T(v) rooted at a node v is the **subgraph** consisting of v and all its descendants in the tree.

Given an n by n symmetric matrix $K(= L\ D\ L^t)$, there corresponds to it a finite ordered matrix **graph** $G(K) = (V, E)$,where a node $v_i \in V$ denotes the ith row (or column) of the matrix K and an edge (v;, v_j) $\in E$ symbolizes an off-diagonal **nonzero** entry $K_{i,j}(= K_{j,i})$. The filled graph

$G_F(K) = (V, E_F)$ represents the structure of the filled matrix $F(= L + L^t)$ of matrix K. The edge set E_F in the filled graph includes the edge set E of $G(K)$ and the set of edges corresponding to the fill-in entries created during the LDL^t factorization.

2.2 Elimination Tree and Symbolic Factorization

Sparse matrix factorization can be divided into two phases: symbolic factorization and numeric factorization. Symbolic factorization determines the structure of matrix factor L, i.e. the locations of the nonzero entries, from that of K. Numeric factorization then makes use of the data structure determined to compute efficiently the numeric values of L.

In graph-theoretic terms, the problem of symbolic factorization is to determine the filled graph $G_F(K)$ from the solution graph $G(K)$. Let's define a list array *PARENT:*

$$PARENT(j) = \min\{i \mid L_{i,j} \neq 0\} \tag{2}$$

The array *PARENT* represents the row subscript of the first nonzero entry in each column of the lower triangular matrix factor L. The definition of the list array *PARENT* results in a monotonically ordered (elimination) tree $T(K)$ of an n by n (non-decomposable) matrix K [13, 15, 19]. In $T(K)$, each node has its numbering higher than its descendants.

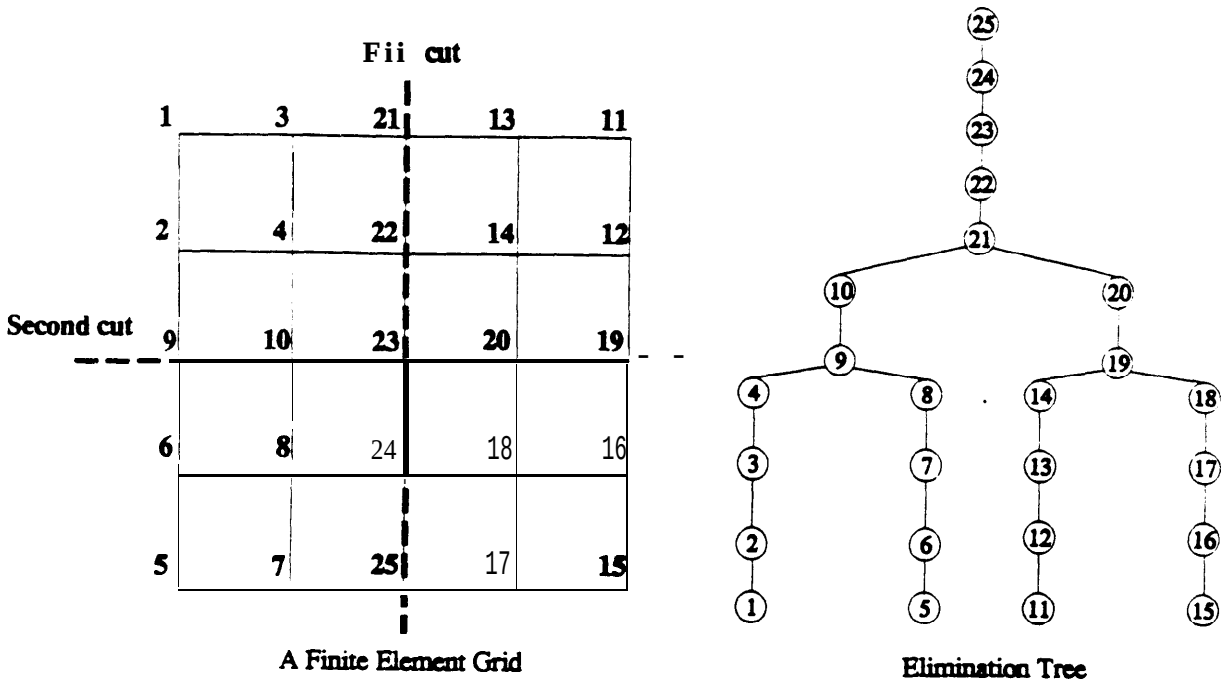
With the definition of the array *PARENT*, the nonzero entries induced by a nonzero entry $K_{i,j}$ or $L_{i,j}$ can be determined based on the following statement:

> Lemma 1: If $K_{i,j}$ (or $L_{i,j}$) $\neq 0$ then for each $k = PARENT(\ldots(PARENT(j)))$, $L_{i,k} \neq 0$ where $k < i$.

That is, the list array *PARENT* contains sufficient information for determining the nonzero pattern of any row in L [12, 13, 19]. This lemma forms the basis for establishing the row-oriented data structure for the sparse matrix factor L.

A topological post-ordering strategy can be used to re-order the nodes of the elimination tree T(K) so the nodes in any subtree are numbered consecutively [13]. This post-ordering process facilitates the partitioning of the matrix into block submatrices where the columns/rows of each block correspond to the node set of a branch in $T(K)$ [14, 16]. Figure 1 shows the matrix structure and its post-ordered elimination tree representation. As shall be discussed in Section 3, the parallel assignment of a sparse matrix em ployed in this study is based on the post-ordered elimination tree representation.

The data structure for the row-oriented factorization method builds a sparse storage scheme over a profile storage scheme. The post-ordered elimination tree plays an important role in defining the data structure for representing the matrix factor L. As noted earlier, we partition the matrix according to the node set along each branch of the elimination tree. Each branch of the elimination tree defines a column block. This partitioning divides a sparse matrix into two basic data sets: principal block submatrices and the row segments outside the diagonal blocks



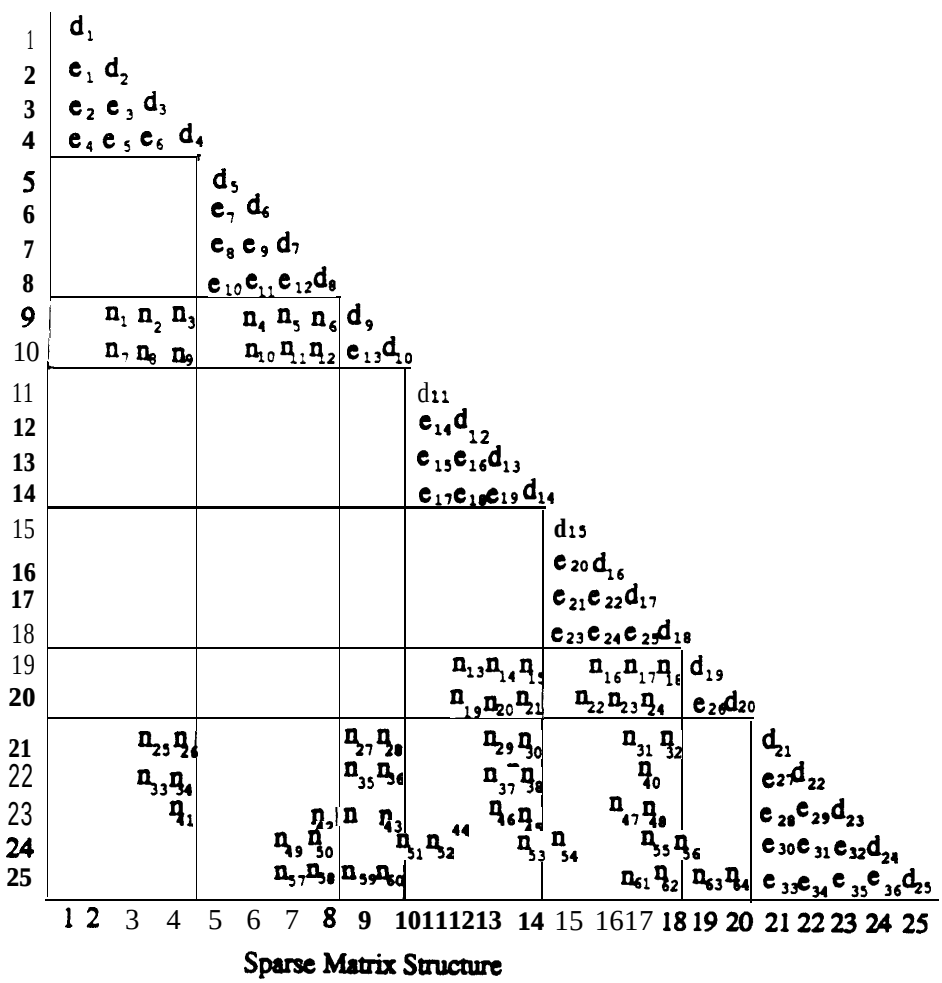


Figure 1: A Post-Ordered Elimination Tree and Sparse Matrix Structure

(see Figure 1). The storage scheme for the matrix factor is defined according to the principal diagonal block submatrices and the offdiagonal row segments.

Following from Lemma 1, each principal block submatrix of L has a full envelope. Thus, it is most appropriate to use a profile storage scheme to store the entries in the principal diagonal blocks. For the entries that lie outside the principal diagonal blocks. we group them by rows as row segments within each column block. A row segment begins at the starting nonzero column subscript and extends to the ending column subscript of the same column block. Based on Lemma 1, it can be seen that in the matrix factor L, there would be no zero entries within each row segment . A symbolic factorization procedure to set up a row-oriented data structure of L has been developed and described in Reference [16].

To facilitate the discussion on the numerical procedures, we denote the bth column block as $\mathcal{C}^{(b)}$ which is partitioned into the principal block submatrix, $\mathcal{D}^{(b)}$, and the set of row segments, $\mathcal{R}^{(b)}$. Since the matrix K is symmetric, we store only the lower triangular part of the matrix. Furthermore, the diagonal entries are assumed to be stored in a separate vector. Subscripts are used to denote the entries in a matrix or vector:

1. A single subscript, i, denotes a specific entry, such as $K_{i,j}$ or f_i.

2. A range, j : k, denotes the entries from subscripts j to k; for example, $K_{i,j:k} = [K_{i,j}, K_{i,j+1}, \ldots, K_{i,k}]$ and $f_{j:k} = [f_j, f_{j+1}, \ldots, f_k]^T$.

3. The colon, :, is also used as a "wild card" denoting a range of nonzero entries in a row or column. For example,

   (a) $K^{(b)}_{i,:}$ denotes the ith row vector of K in column block b. If $K_{i,:} \in \mathcal{D}^{(b)}$, the row vector starts at the first nonzero column subscript in that column block b and extends to the column subscript of i − 1. If $K_{i,:} \in \mathcal{R}^{(b)}$, the row vector starts at the first nonzero column subscript and extends to the ending column subscript of the column block b.

   (b) $K^{(b)}_{i,j:}$ denotes the nonzero entries in the row vector starting from the jth column subscript to the ending subscript of that row vector in the column block b. $K^{(b)}_{i,:j}$ represents the row vector starts at the first nonzero column subscript of that row and extends to the j column subscript.

   (c) $K^{(b)}_{:,i}$ denotes the ith column vector of K in column block b.

   The same notation is used for vectors; for example, $f^{(b)}_{i:}$ ($f^{(b)}_{:i}$) indicates the range of entries starting from (ending at) subscript i in vector f corresponding to the column block b.

The above notations will be used to describe the numerical procedures in this paper.

2.3 Numerical Factorization

The sparse matrix factorization is basically a block column scheme with a row-oriented profile factorization for the principal diagonal block submatrices. The factorization of each column block involves three basic steps:

1. update the coefficients in the principal block submatrix and the row segments of the column block;
2. decompose the principal block submatrix by a row-based profile factorization scheme;
3. factor the row segments by a series of forward solves.

This procedure is depicted as shown in Figure 2 and is summarized in Figure 3.

Once the matrix factor L is obtained, the solution vector can be computed by a forward solve $z = L^{-1}f$, followed by a backward solve x = $L^{-t}D^{-1}z$.

1. In the forward solution procedure, the calculations proceed in a column block by column block manner. For each column block, we first perform a forward solve with the diagonal block factor. We then update the solution vector with the row segments within the column block.
2. In the backward solution procedure, the calculations proceed in a row block by row block fashion. For each row block of L, we first update the solution vector due to the row segments. We then update the solution with the diagonal block factors using a backward solve.

The forward and backward solution procedures are summarized in Figure 4. Again, the solution procedures for the triangular systems proceed in a column block by column block basis. A sequential implementation of the solution procedures discussed in this section has been described in detail in Reference [16].

3 Matrix Partitioning for Parallel Computations

The efficiency of a parallel algorithm depends on the proper distribution of data among the processors. In a distributed memory computer, it is desirable to confine data access to the local memory of a processor as much as possible, thus minimizing the number of messages that must be sent and received. Moreover, a good balance of data distribution is important to minimize the idle time of a processor waiting for other processors to complete their tasks. In this section, we describe a processor assignment strategy based on the matrix partitioning according to the post-ordered elimination tree [7].

The coefficients of a sparse matrix factor are distributively stored among the processors according to the column blocks. Figure 5 shows an example of the data assignment of a matrix on multiple processors. Essentially, the strategy is to assign the rows corresponding to the nodes

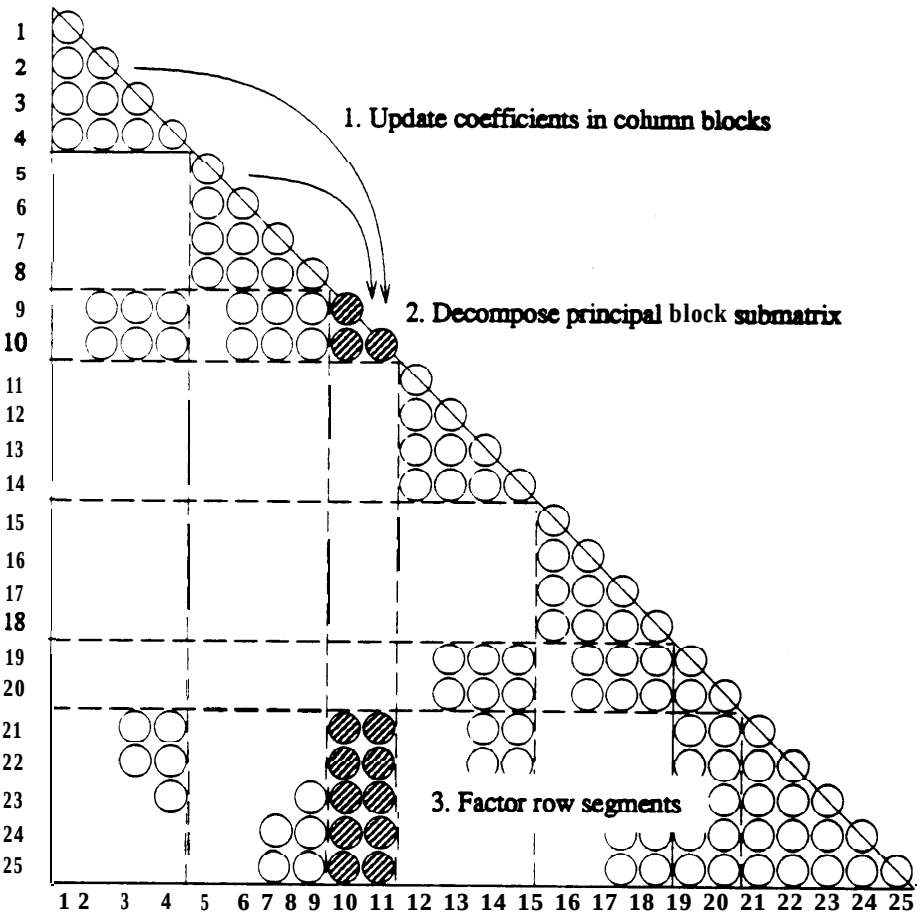


Figure 2: A Sequential Row-Oriented Factorization Scheme

```
Procedure: Sequential Factorization
/* nblock denotes the number of partitioned column blocks.*/
BEGIN
    FOR each column block b = 1 TO nblock, DO
    BEGIN      /* sequential factorization of each column block */
        seq_fact_column_block(b);
    END.
END.
```

```
Procedure: seq_fact_column_block(b)
/* Sequential Factorization of each column block C^(b) */
BEGIN
    /* 1. update column block by fanning in updated entries
            from preceding column blocks */
    FOR  each row j ∈ C^(b), DO
    BEGIN
        FOR each preceding block p = 1 TO b - 1, DO
        BEGIN
            FOR each row i ∈ D^(b) AND i < j, DO
            BEGIN      /* update coefficients */
                K_{j,i}^(b) = K_{j,i}^(b) - R̄_{j,:}^(p) * R_{i,:}^(p)T;      /* dot product ● /
            END:"
            IF j ∈ D^(b) THEN
                y ⟵ R̄_{j,:}^(p)T;
                R_{j,:}^(p) = y^T D^(p)-1;
                K_{j,j}^(b) = K_{j,j}^(b) - R_{j,:}^(p) * y;      /* dot product */
            ENDIF.
        END.
    END.
    /* 2. compute factor of principal submatrix K^(b,b) ∈ D^(b) */
    factor K^(b,b) = L^(b) D^(b) L^(b)T;      /* profile factorization */
    /* 3. factor row segments */
    FOR each row segment i ∈ R^(b), DO
    BEGIN      /* forward solve */
        R̄_{i,:}^(b)T = L^(b)-1 K_{i,:}^(b)T ;      /* note* R̄_{i,:}^(b) = R_{i,:}^(b) D^(b) */
    END.'
END.
```

Figure 3: A Sequential Procedure for Matrix Factorization

```
Procedure: Sequential Forward Solve Lz = f
BEGIN
    FOR each column block b = 1 TO nblock, DO
    BEGIN
        solve z^(b) = L^(b)-1 f^(b) ;      /* forward solve */
        FOR each row segment i ∈ R^(b), DO
        BEGIN      /* update soiution vector */
            f_i = f_i - R^(b)_i,: * z^(b);      /* dot product */
        END.
    END.
END.

Procedure: Sequential Backward Solve DL^T x = z
BEGIN
    z = D-'z ;
    FOR each column block b = nblock TO 1 STEP -1, DO
    BEGIN
        FOR each row segment i ∈ R^(b), DO
        BEGIN      /* update solution vector */
            z^(b) = z^(b) - x_i * R^(b)T_i,:;      /*axpyoperation */
        END.
        solve x^(b) = L^(b)-T z^(b);
    END.
END.
```

Figure 4: Sequential Solution Procedures for Triangular Systems

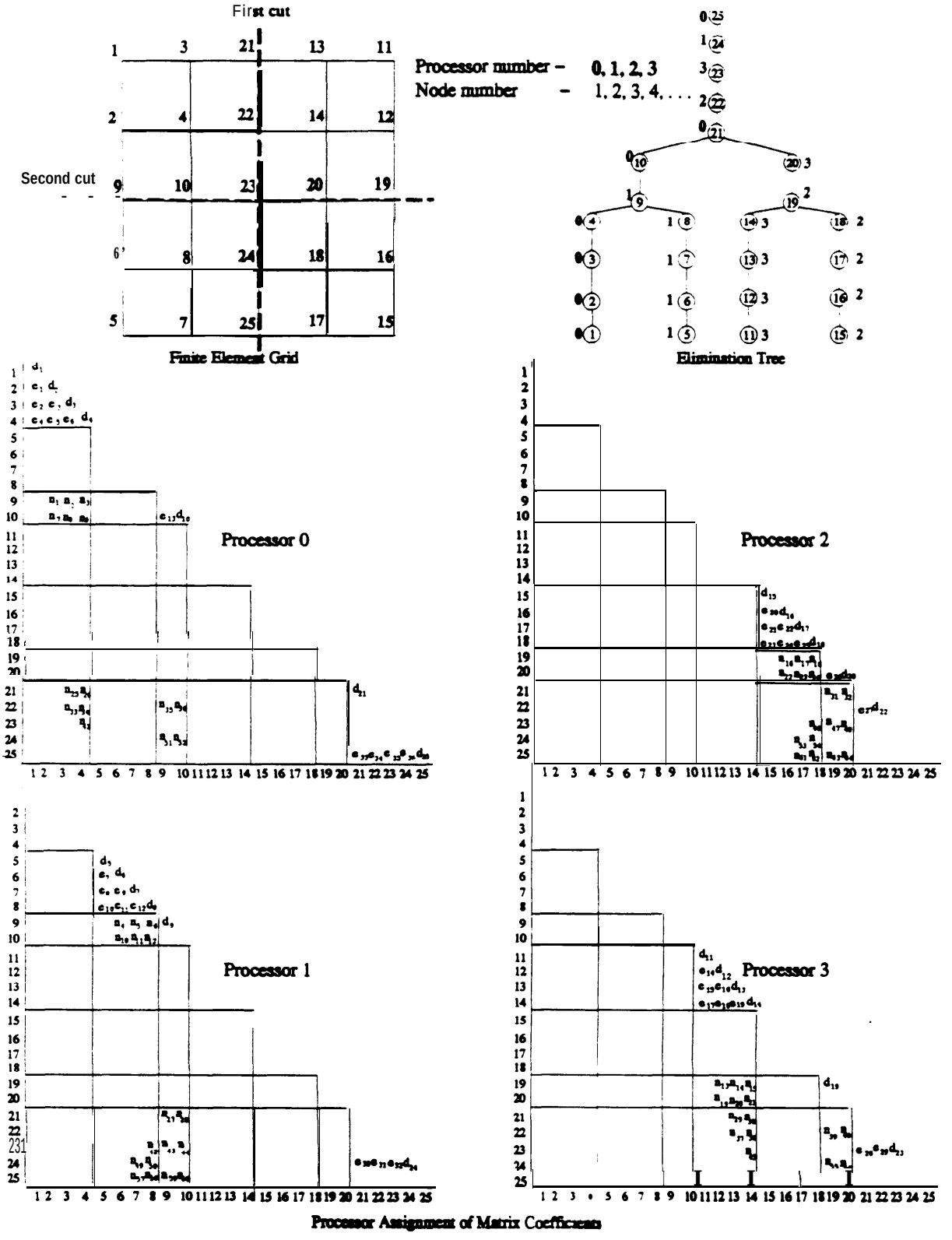


Figure 5: Assignment of Matrix Coefficients on Multiple Processors

along each branch (column block) of the elimination tree to a processor or a group of processors. Beginning at the root of the elimination tree, the nodes belonging to this branch of the tree are assigned among the available processors in a rotating block round robin fashion, or a block wrap mapping [8]. As we traverse down the elimination tree, at each fork of the elimination tree, the group of processors is divided to match the number and the size of the subtrees below the current branch. A separate group of processors is assigned to each branch at the fork and the process is repeated for each subtree. For a balanced elimination tree, the group of processors assigned to the branch is always a subcube or subring. Otherwise, the procedure is to follow as closely as possible the mapping of subcubes or subrings to subtrees. The number of processors assigned to each branch is weighted according to the number of nodes in the subtree of that branch. For example, five processors may be assigned to one subtree and three processors to the other, in this case neither branch has been assigned a subcube or subring. The process of assigning subcubes or groups of processors to each branch of the elimination tree continues until each subcube consists of only one processor, then all remaining nodes in the subtree are assigned to the single processor.

As noted in Section 2.2, a sparse matrix is partitioned into two basic sets: the principal diagonal block submatrices and the row segments outside the principal block submatrices. For the principal block submatrix, which has the profile structure, the processor assignment proceeds on a row group by row group basis [1]. In our implementation, we assign a row group corresponding to a node in the finite element model, grouping individual degrees of freedom per that node as a unit.

The row segments are assigned to the processors that share the column block. When the node set of a branch in the elimination tree is shared among a number of processors, the rows are assigned to the processors sharing the node set (column block) in an alternating round robin or wrap fashion. That is, for a subtree-to-subcube mapping, two successive rows are assigned to the neighboring processors in the subring. This can be determined easily using a simple procedure as follows:

```
Procedure: processor( row-number, #_of_shared_processors, processor-list)
BEGIN
   index = mod( row-number, # _o f shared-processors);
   processor = processor_list[index]
END.
```

where *processor-list* is a list of processors sharing the column block, *index* points to the position in the list where the processor number can be found, and *processor* is the processor to which the row segment is assigned. By this rule if the entire node set of a branch in the elimination tree is assigned to a single processor, all of the row segments in that column block are assigned to the

[1] It is more efficient to assign the rows in a row block fashion. The optimal block width will be determined by the relative speed of the processors and communication time. For the Intel/i860 hypercube we have found that block width to be about eight. For the Intel/iPSC2 we found the optimal block width to be about three or four.

same processor. Thus if a column block is not shared, no processor to processor communications are needed to factorize the column block.

4 Parallel Solution Procedures

4.1 Parallel Factorization

As in the sequential computing environment, the parallel numerical factorization procedure computes the matrix factor L on a column block by column block basis. The block factorization scheme consists of (1) a profile factorization for the principal diagonal block submatrices; and (2) a profile forward solve for the row segments per each column block. The matrix factorization is divided into two distinct phases. During the first phase, the column blocks assigned entirely to a single processor are factorized. During the second phase, the columns blocks shared by more than one processor are factorized. The operations involved in these two phases are described in the following subsections.

4.1.1 Phase One of Parallel Factorization

In the first phase, each processor independently factorizes the column blocks assigned to a single processor. There are two distinct stages in this first phase of factorization.

I.1 Factoring the column blocks entirely in the same processor using the same procedure as in the sequential factorization.

I.2 Forming dot products among the row segments for updating the coefficients in the column blocks shared by a number of processors. These dot products are then fanned-out to update the remaining matrix coefficients in the same processor or saved in the buffer to be fanned-in to another processor during the second phase of factorization.

This procedure is graphically illustrated as shown in Figure 6.

The procedure for the factorization of a column block stored entirely in a processor is described in Figure 7. The factorization of each column block in a processor follows the same procedure as in the sequential factorization. We first fan-in the dot products among the row segments located in the preceding column blocks to update the coefficients in the current column block. For the principal block submatrix which has a variable banded structure, a profile-oriented factorization scheme is used to decompose the submatrix. For the row segments which lie in the same column block, a profile forward solver is used to calculate the numerical coefficients in these segments. (This is the reason that all of the row segments of the column blocks are stored in the same processor as the diagonal block. It eliminates the need to send the row segments to another processor in the forward solve.)

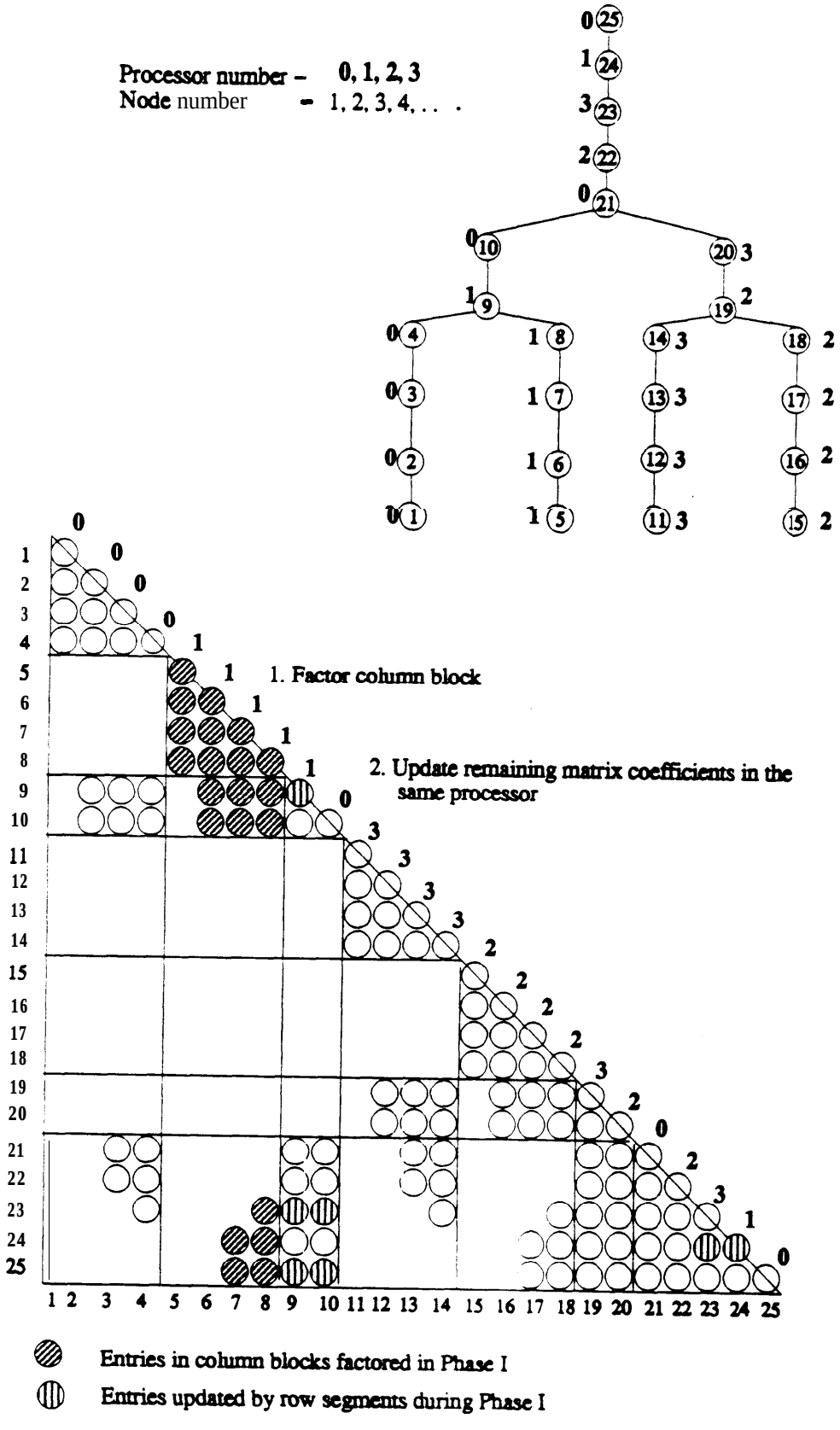


Figure 6: Phase I of Parallel Factorization Scheme

```
Procedure: Parallel Factorization - Phase I
BEGIN
   /* I.1 Factorization of column blocks entirely in my processor */
   FOR each column block b ∈ my_processor, DO
   BEGIN     /* sequential factorization of each column block */
      seq_fact_column_block(b);
   END.      /* Compiete factoring column block */
   /* I.2 Perform dot products between row segments */
   FOR each column block b ∈ my_processor, DO
      FOR each row segment j ∈ R^(b), DO
      BEGIN
         y = D^(b)-1 R̃^(b)T_j,:;
         FOR row i = j TO n, DO
         BEGIN
            s = R̃^(b)_j,: * y;      /* dot product */
            /* find what processor L_ij belongs to */
            proc = processor(i, #_shared_processor, processor_list);
            IF (proc = my_processor) THEN
               K_i,j = K_i,j - s;      /* fan_out update */
            ELSE
               accumulate update coefficient: s_i,j = s_i,j + s;
               save coefficient (s_i,j, i, j) in buffer for processor proc;
            ENDIF.
         R^(b)_j,: = y^T;
         END.
      END.
   END:
END.
```

Figure 7: A Parallel Factorization Procedure for Column Blocks Entirely in the Processor

At the end of this phase, each processor forms the dot products among the row segments in its column blocks which have been factored. These dot products are fanned out immediately to update the coefficients of the remaining matrix in the same processor or saved in a buffer for fanning into another processor for updating the column block during the second phase of parallel factorization. The strategy is to carry out as much computations as possible in the processor. When a processor sends the dot products to anot her processor, all dot products saved in the buffer for that processor are sent as a package.

4.1.2 Phase Two of Parallel Factorization

In the second phase of numerical factorization, the column blocks shared by more than one processor are factorized. The parallel factorization of a column block proceeds as follows:

II.1 Each processor fans-in the dot products saved previously in the buffers on the other processors sharing the column block. The dot products received are used to update the principal block submatrix and the row segments.

II.2 Perform a parallel profile factorization and factor the row segments.

II.3 Form dot products among row segments in the column block. This step consists of two basic operations:

11.3.1 Form dot products among the row segments stored in the processor.

11.3.2 Form dot products between the row segments stored in different processors.

The dot products are fanned-out to update the remaining matrix coefficients in the same processor or saved in the buffer to be fanned-in to another processor (see Step 11.1.)

This procedure is illustrated in Figure 8.

The parallel factorization of a column block shared by multiple processors is described in Figure 9. In this procedure, the factorization is performed in place; that is, the matrix factors L, R and D share the same storage as the original matrix K. We use the overbar, such as $\bar{R}$, to indicate the coefficients that have not been divided by the diagonal entries of D. Furthermore, we assume that every processor has a (temporary) copy of all the diagonal entries for the column block; those entries that do not belong to the processor will be discarded after the column block is factored.

After the profile submatrix has been factored and all of the row segments in the column block have been updated, each processor forms the dot products among the row segments in the same processor. Then the row segments of the column blocks are circulated among the processors sharing the column block. When a processor receives another processor's row segment, each processor forms the dot products between its own row segments and the row segments received from the neighboring processor. In this manner, each processor receives the other processors' row

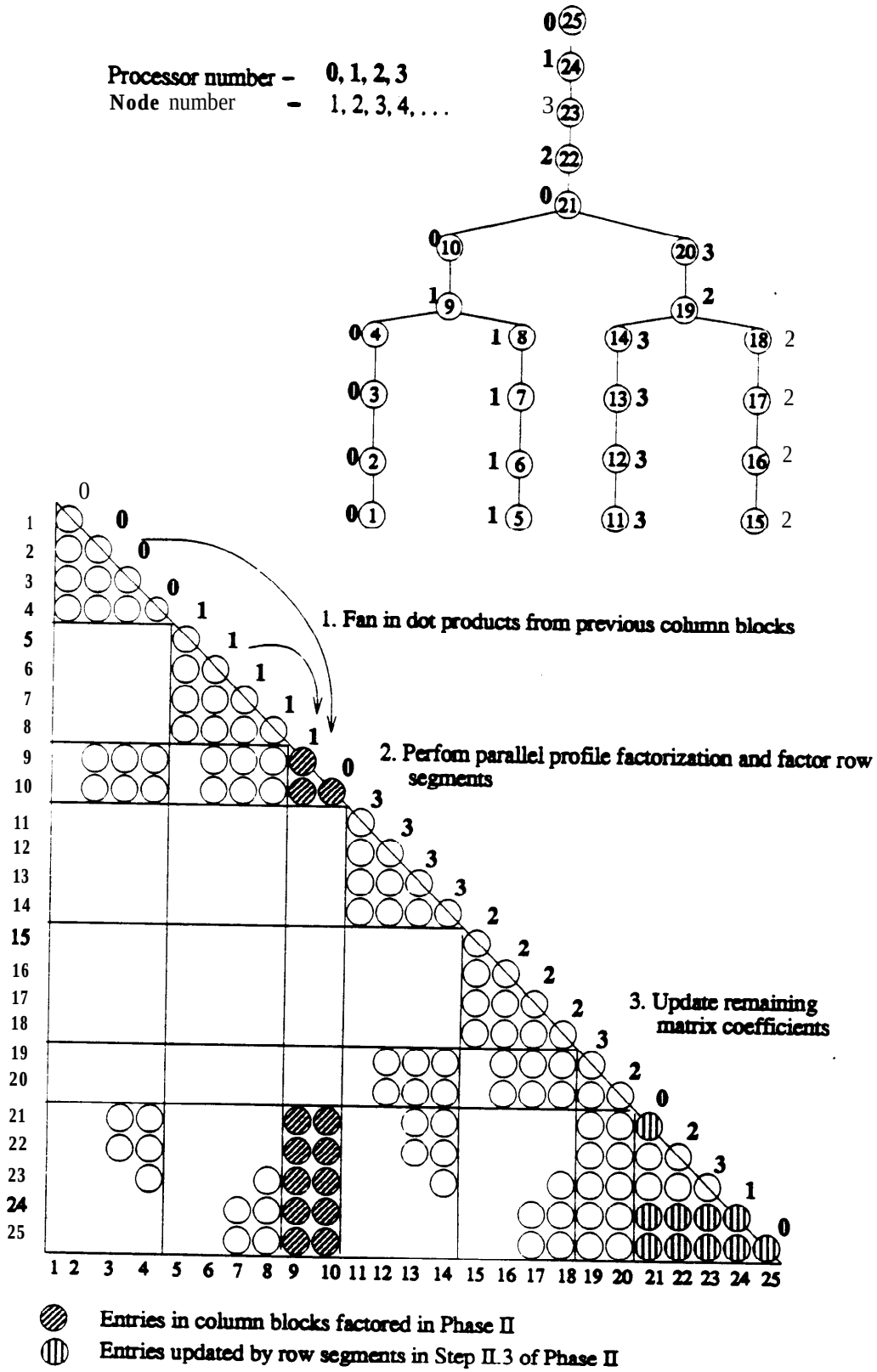


Figure 8: Phase II of Parallel Factorization Scheme

```
Procedure: Parallel Factorization - Phase II
BEGIN
  FOR each column block b ∈ my_processor, DO
  BEGIN
    /* matrices can share same storage in implementation */
    assign K^(b) to D^(b), L̄^(b) and R̄^(b);
    /* II.1 Update column block by fanning in updated entries from
            previous column blocks in other processors */
    receive update entries (s, i, j) from other processors sharing C^(b);
    FOR each entry (s, i, j) received, DO
    BEGIN
      IF (i = j) THEN D_{i,i} = D_{i,i} - s;
      IF '(i, j) ∈ D^(b) and i ≠ j THEN L̄^(b)_{i,j} = L̄^(b)_{i,j} - s;
      IF (i, j) ∈ R^(b) THEN R̄^(b)_{i,j} = R̄^(b)_{i,j} - s;
    END.
    /* II.2 parallel factorization of column block */
    FOR each row i ∈ D^(b), DO
    BEGIN
      IF row i ∈ my_processor, THEN
        /* form L^(b)_{i,:} and D_{i,i} */
        y ← L̄^(b)T_{i,:};
        L^(b)_{i,:} = [..., y_j/D_{j,j}, ..., y_{i-1}/D_{i-1,i-1}];
        D_{i,i} = D_{i,i} - L^(b)_{i,:} * y;      /* dot product */
        broadcast L^(b)_{i,:} and D_{ii} to other processors sharing C^(b);
      ELSE
        receive L^(b)_{i,:}, D_{ii} from other processors sharing C^(b);
      ENDIF.
      assign D_{ii} to D^(b);      /* every processor has entire copy of D^(b) */
      /* update remaining entries in column block */
      FOR each row j ∈ D^(b) and j > i + 1, DO
      BEGIN      /* dot product */
        IF row j ∈ my_processor THEN L̄^(b)_{j,i} = L̄^(b)_{j,i} - L̄^(b)T_{j,:i-1} * L^(b)_{i,:};
      END.
      FOR each row j ∈ R^(b), DO
      BEGIN      /* dot product */
        IF row j ∈ my_processor THEN R̄^(b)_{j,i} = R̄^(b)_{j,i} - R̄^(b)T_{j,:i-1} * L^(b)_{i,:};
      END.
    END.
```

Figure 9: A Parallel Factorization Procedure for Column Blocks Shared among Processors

```
/* II.3 Perform dot products between row segments */
/* 11.3.1 For the row segments in my processor */
send all row segments R̄(b) to next processor;
FOR each row segment j ∈ my_processor , DO
BEGIN
    y = D(b)-1 R̄(b)T j,:
    FOR each row segment i ∈ my_processor and i ≥ j , DO
    BEGIN
        s = R̄(b) j,: * y;      /* dot product ● /
        /* find what processor L_ij belongs to */
        proc = processor (i, #-of_shared_processor, processor_list);
        IF(proc = my_processor) THEN
            K_i,j = K_i,j - s;      /*fan_out update ● /
        ELSE
            accumulate update coefficient: s_i,j = s_i,j + s;
            save coefficient (s_i,j, i, j) in buffer for processor proc;
        ENDIF.
    END.
    R(b) 7,: = y^T;
END-:
/* II.3.2 Circulate row segments to other processors */
FOR k = 1 TO #_of_shared_processors - 1, DO
BEGIN
    receive row segments R̄(b) from preceding processor;
    FOR each row segment i ∈ my_processor, DO
    BEGIN
        FOR each row segment j < i received from other processor, DO
        BEGIN      /* dot product */
            s = R̄(b) i,: * R(b)T j,:;
            /* find what processor L_ij belongs to */
            proc = processor(i, #_of_shared_processor, processor_list);
            IF (proc = my_processor) THEN
                K_i,j = K_i,j - s;      /* fan-out update */
            ELSE
                accumulate update coefficient: s_i,j = s_i,j + s;
                save coefficient (s_i,j, i, j) in buffer for processor proc;
            ENDIF.
        END.
    END.
END.
/* circulate the received row segments to other processor */
IF (k ≠ #_of_shared_processors - 1) THEN
    send row segments R̄(b) to next processor;
END.
END.
```

Figure 9 (continued)

segments and computes the dot products only once. The row segments are then passed on to the next processor. It should be noted that, in actual implementation, memory buffers are allocated for sending and receiving data.

4.2 Parallel Forward Solve

The forward solution procedure can be viewed symbolically as the traversal of the elimination tree from the leaves to the root. Each processor begins to work on the column blocks or subtree that reside entirely in the processor. As work proceeds up the elimination tree, the processors begin to work on the column blocks corresponding to the branches that are assigned with more than one processor. That is, each processor begins working independently and then merges its work with other adjacent processors until all of the processors work together on the same (root) block at the end. For the forward solve, the processors may begin asynchronously but finish synchronized at the same time.

The parallel forward solution procedure is described in Figure 10. The solution procedure can be performed in place in that the solution vector z and the load vector f can share the same memory locations. Although the vectors z and f are divided into blocks just as the matrix is divided into column blocks, we assume that each processor contains an entire vector of length n; those entries that do not belong to the processor will be discarded after the solution is completed.

The forward solve is divided into two phases. In the first phase, each processor calculates the portion of the solution vector z corresponding to the column blocks which reside entirely within a single processor. Each processor also updates the shared portions of the solution vector based on the row segments in that column block.

In the second phase, the parallel forward solve for the shared portions of the vector is performed. This parallel procedure is carried out in a column block by column block basis. There are three basic operations for factoring a column block shared by multiple processors:

1. Send and receive updates for the solution vector corresponding to the current block.
2. Calculate the solution for the current block.
3. Use the solution computed to update the remaining coefficients using the row segments in the column block.

At the end of the procedure, each processor has the correct value of z_i assigned to the processor; other entries that do not belong to the processor can be discarded.

4.3 Parallel Backward Solve

The backward substitution solves the upper triangular system of equations $DL^T x = z$. The procedure is essentially a reverse of the forward solve. Symbolically, the procedure proceeds at the root of the elimination tree and traverses down to the leaves. In the backward solve, all of

```
Procedure: Parallel Forward Solve Lz = f - Phase I
BEGIN
   /* Initialize the entire vector f in the processor */
   FOR each row i = 1 TO n, DO
   BEGIN
      IF i ∈ my_processor THEN
         f_i = f_i;
      ELSE
         f_i = 0;
      ENDIF.
   END.
   FOR each column block b ∈ my_processor , DO
   BEGIN
      solve z^(b) = L^(b)-1 f^(b) ;      /* forward solve */
      FOR each row segment i ∈ R^(b), DO
      BEGIN    /* update solution vector ● /
         f_i = f_i - R^(b)_i,: * z^(b);
      END.
   END.      /* partial updates computed and stored */
END.

Procedure: Parallel Forward Solve Lz = f - Phase II
BEGIN
   FOR each column block b , DO
   BEGIN
      /* 1. update solution vector ● /
      broadcast f_i^(b), i ∈ D^(b), to other processors sharing C^(b);
      Receive f^(rec) and accumulate f^(b) = f^(b) + f^(rec) ;
         /* Note: every processor has a copy of f_i^(b) */
      /* 2. parallel forward solve */
      FOR each row i ∈ D^(b), DO
      BEGIN
         IF i ∈ my-processor THEN
            z_i ← f_i;
            broadcast z_i to other processors sharing C^(b);
         ELSE
            receive z_i;
         ENDIF.
         /* update solution vector */
         FOR each row j ∈ D^(b), j < i, DO
         BEGIN
            f_j = f_j - z_i * L^(b)_j,i;
         END.
      END.      /* Note: every processor has a copy of z^(b) */
      /* 3. update vector by row segments */
      FOR each row segment i ∈ R^(b) , DO
      BEGIN
         f_i = f_i - R^(b)_i,: * z^(b);      /* dot product */
      END.
   END.
END.
```

Figure 10: A Parallel Forward Solution Procedure

the processors begin working together on the same block and branch out into smaller groups of processors until each processor is working independently on data within its own local memory. During the backward substitution, the processors start the computations synchronized, but may finish asynchronously at different times.

The backward solution procedure is described in Figure 11. Similar to the forward solve and the factorization, the procedure can be divided into two phases. In phase one, the processors compute the portion of the solution vector shared by multiple processors. In the second phase, each processor calculates the portion of the solution vector corresponding to the column blocks residing within a single processor.

Phase one of parallel backward solve is performed as follows: For each shared column block, $\mathcal{C}^{(b)}$.

1. update the vector based on the values that have been resolved;
2. perform a backward substitution to solve for the portion of solution vector corresponding to the current block;
3. distribute the result and assign the solution to each processor.

Again, the solution procedure can be performed in place, in that the vectors x and z can share the same memory locations. We assume that each processor has an entire vector z of length n. We first initialize the solution vector z in each processor and divide it by the diagonal entries, $z = D^{-1}z$. We then update the vector $z^{(b)}$ based on the row segments. The updates are summed across all processors so that each processor has a copy of $z^{(b)}$.

The backward solve for the principal profile submatrix is performed by a procedure similar to the forward solve procedure described in Reference [8]. In this backward solve procedure, each processor updates the vector $z^{(b)}$ only as far as necessary to keep the other processors busy. It updates $z^{(b)}$ from the current row up to the next row that the processor is responsible for. It then sends off this portion of the vector and then proceeds to update the remaining coefficients of the vector based on the current row of L. To illustrate this process, let's assume that a processor is responsible for rows i and k, where k < i and both k, i $\in \mathcal{D}^{(b)}$. The processor receives a vector containing the updates for $z_i^{(b)} \ldots z_{k+1}^{(b)}$. The processor then adds this vector to z and then update part of the vector using row i of L:

$$z_{k+1:i-1}^{(b)} = z_{k+1:i-1}^{(b)} - z_i * L_{i,k+1:i-1}^{(b)T}$$

The processor immediately sends $z_{i-1}^{(b)} \ldots z_{k+1}^{(b)}$ off to the processor responsible for row $i - 1$ so that the receiving processor can proceed its own computations. After this message is sent off, the processor continues to update the remaining coefficients based on row i of L:

$$z_{:k}^{(b)} = z_{:k}^{(b)} - z_i * L_{i,:k}^{(b)T}$$

```
Procedure: Parallel Backward Solve DL^T x = z - Phase I
BEGIN
   /* Initialize solution vector in each processor */
   FOR each row i = 1, n, DO
   BEGIN
      IF i ∈ my_processor THEN
         z_i = z_i / D_{i,i};
      ELSE
         z_i = 0;
      ENDIF.
   END.
   FOR each column block b = duck,. . . STEP -1 DO
   BEGIN
      /* 1. update the solution vector z^(b) by row segments */
      FOR each row j ∈ R^(b) and j > i, DO
      BEGIN
         z^(b) = z^(b) - x_j * R^(b)T_{j,:};      /* axpy operation */
      END.
         /* sum the updates across processors */
      broadcast z^(b) to other processors sharing C^(b) ;
      receive z^(rec) and accumulate: z^(b) = z^(b) + z^(rec);
        /* Note: every processor has a copy of z^(b) ● */
      /* 2. parallel profile backward solve */
      /* k denotes the row preceding row i, both k, i ∈ D^(b) and my_processor */
      FOR each row i ∈ D^(b) STEP -1, DO
      BEGIN
         IF i ∈ my_processor THEN
            IF i ≠ last row of D^(b), THEN
               receive w_{k+1:i};
               z_{k+1:i} = z_{k+1:i} + w_{k+1:i};
            ENDIF.
            x_i ← z_i;
            IF i ≠ first row of D^(b), THEN
               w_{k+1:i-1} = z_{k+1:i-1} - x_i * L_{i,k+1:i-1};
               send w_{k+1:i-1} to processor responsible for row i - 1;
               z_{:k} = z_{:k} - x_i * L_{i,:k}
            ENDIF.
         ENDIF.
      END.
      /* 3. distribute the solution vector */
      broadcast x_i^(b), i ∈ my_processor, to other processors sharing C^(b);
      receive x_i^(rec) and assigned them to x^(b);
      /* Note: every processor has a copy of x^(b) */
   END.
END.
```

Figure 11: A Parallel Backward Solution Procedure

```
Procedure: Parallel Backward Solve DL^T x = z - Phase II
BEGIN
   FOR each block b ∈ my_processor, DO
   BEGIN
      /*1. update by row segments in column block b ●  /
      FOR each row segment j ∈ R^(b), DO
      BEGIN
         z^(b) = z^(b) - x_j * R^(b)T_j,:;
      END.
      /* 2. update by rows in principal block b */
      FOR each row j ∈ D^(b) STEP -1, DO
      BEGIN
         x_j ⟵ z_j^(b);
         z^(b)_:j-1 = z^(b)_:j-1 - x_j * L^(b)T_j,:j-1;
      END-.
   END.
END.
```

Figure 11 (cont inued)

The processor will then wait until it receives another vector containing updates from the processor responsible for row $k + 1$ and repeat this process for row k and so on until the block backward solve is complete. At the end of the profile backward solve, each processor has the solution vector for the rows it is responsible for. The solution values are distributed to all the processors sharing the column block.

Phase two of the backward solve computes the solution vector based on the column blocks entirely in the processor. The two basic steps of the backward solve for each column block $\mathcal{C}^{(b)}$ are:

1. Update $z^{(b)}$ based on the row segments in the column block.

2. Execute a profile backward solve using the principal block submatrix factor $L^{(b)}$.

The processors perform the calculations independently without any processor communications and may complete the solution at different times.

5 Partial Matrix Factor Inversion

While the parallel matrix factorization procedure described in the previous section performs well, the parallel forward and backward solves do not exhibit similar efficiency. Heath et. al. concluded that there is little that can be done to improve the performance of the parallel triangular solvers [9]. Our procedures suffer in a similar way. When examining closely the performance of the forward and backward solves, most of the parallelism come from assigning many column blocks to a single processor so that the processors can work independently. Good speed up also seems to occur when working with the distributed row segments. The main deficiency lies on the parallel solutions of the triangular systems for the dense principal block submatrix factors; the triangular solution procedures have a significant number of communication overhead. Additional messages are also required before and after the profile submatrix solve for the backward solution phase.

In this section, we describe an alternative method that can expedite the solution of triangular systems. The strategy is to invert the dense principal block submatrix factors that are shared by multiple processors. This strategy can significantly improve the performance of the direct solution methods for problems that involve multiple right-hand side vectors. Section 5.1 describes a method that computes the inverse of a dense matrix factor without extra processor communications. Section 5.2 presents the forward and backward solves based on the matrix factors with partial inverses.

5.1 Parallel Computation of the Inverse of a Dense Matrix Factor

A faster method to solve for z in $Lz = f$ in parallel would be to form L^{-1}, the inverse of the matrix factor. By distributing the coefficients of the factor inverse L^{-1} over all the processors, each processor can form the partial matrix-vector product from the entries in its processor and

sum the partial products from each processor globally to form the solution. This reduces the communication overhead down to one set of messages, which is to sum the partial products among the processors sharing the principal block submatrix $\mathcal{D}^{(b)}$. The problem is to directly compute the inverse of a dense matrix factor L.

First, let's define L_i to represent the triangular matrix factor that contains the ith row of L :

$$L_i = \begin{bmatrix} 1 & & & & & & & \\ & 1 & & & & & & \\ & & \ddots & & & & & \\ & & & 1 & & & & \\ L_{i,1} & \cdots & L_{i,j} & \cdots & L_{i,i-1} & 1 & & \\ & & & & & & 1 & \\ & & & & & & & \ddots \\ & & & & & & & 1 \end{bmatrix} \tag{3}$$

The triangular matrix factor L can now be written as:

$$L = L_1 \ldots L_i L_{i+1} \ldots L_n = L^{(i)} L_{i+1} \ldots L_n \tag{4}$$

where $L^{(i)} = L_1 \ldots L_i$ contains the first i rows of the triangular factor L. The inverse of the triangular factor becomes:

$$L^{-1} = L_n^{-1} \ldots L_{i+1}^{-1} L_i^{-1} \ldots L_1^{-1} = L_n^{-1} \ldots, L_{i+1}^{-1} L^{(i)-1} \tag{5}$$

where

$$L_i^{-1} = \begin{bmatrix} 1 & & & & & & & \\ & 1 & & & & & & \\ & & \ddots & & & & & \\ & & & 1 & & & & \\ -L_{i,1} & \cdots & -L_{i,j} & \cdots & -L_{i,i-1} & 1 & & \\ & & & & & & 1 & \\ & & & & & & & \ddots \\ & & & & & & & 1 \end{bmatrix} \tag{6}$$

In other words, L_i^{-1} can be obtained by negating the offdiagonal values in the ith row of L. Note that, each row of L can be computed as:

$$L_{i,:i-1}^T = L^{(i-1)-1} K_{i,:i-1}^T \tag{7}$$

Once the ith row of L is computed, the inverse of the matrix factor $L^{(i)}$ is given by

$$L^{(i)-1} = L^{(i-1)-1} L_i^{-1} \tag{8}$$

Based on the equations above, the procedure for inverting a lower triangular factor on a row by row basis can be summarized as follows:

1. Compute $L_{i,:}$ by Equation 7.

2. Negate the entries of $L_{i,:}$ to form L_i^{-1} as in. Equation 6.

3. Compute $L^{(i)-1}$ by multiplying L_i^{-1} with $L^{(i-1)-1}$ as in Equation 8.

The multiplication shown in Equation 8 only affects the entries on row i of $L^{(i)-1}$. Therefore, no additional processor communications are needed when $L^{(i)-1}$ is formed in the processor responsible for row i. We apply this procedure to directly compute the inverses of the dense principal block submatrix factors.

Figure 12 summarizes the procedure for parallel factorization that includes inverting the dense block submatrix factors that are shared by multiple processors. The procedure is the same as the direct parallel LDL^T factorization described in Figure 9 except in the factorization of column block (step II.2). We assume that there is a duplicate temporary copy of the coefficients for the column block that is being factorized. In our implementation, we use the buffer reserved for the messages to hold the temporary column block. The factor inverses are formed mainly by matrix-vector multiplication instead of the forward triangular solve. The number of processor communications are the same for both the direct $L\ D\ L^T$ factorization and the factorization with partial factor inverses.

5.2 Forward and Backward Solvers for Partially Inverted Matrix Factors

As noted earlier, it has been well recognized that efficient implementation of parallel forward and backwards solves is quite difficult because of the data dependencies in the solution of triangular systems. One approach is to transform the triangular solves into matrix-vector multiplication by inverting portions of the matrix factors. An approach of using the inverses of partitioned factors for parallel forward and backward solution has been proposed by Alvarado and Schreiber [1]. Their approach is to partition the matrix factor L into column panels (blocks) and compute the inverses of the partitioned factors. Our approach, however, differs from the approach of Alvarado and Schreiber in a number of manners. First, we store the column block using a row oriented scheme rather than a column storage scheme. Since we do not invert the portion of L stored in row segments, we do not need to worry about the extra fills that may occur during the matrix inversion. We also do not need to reorder or permute any rows or columns of the matrix factor in order to minimize the number of partitions that can be inverted. In our approach we look for any shared principal block submatrices that are dense. This is often the case for the submatrix blocks corresponding to the branches near the root of the elimination tree.

We now introduce the forward and backward solvers for the partially inverted matrix factor introduced in the previous section. Since the column blocks assigned to a single processor are not inverted, no changes are needed for the first phase of the forward solve and the second phase of the backward solve. The routine for the second phase of forward solve is summarized in Figure 13. Notice that the only difference is to replace the parallel forward solution procedure on the

Procedure: **Parallel Factorization (Partial Factor Inverses) - Phase II**
BEGIN
FOR each **column block** $b \in my\_processor$, **DO**
BEGIN
/* matrices **can** share same storage in implementation */
assign $K^{(b)}$ **to** $D^{(b)}$, $\bar{L}^{(b)}$ **and** $\bar{R}^{(b)}$;
/* II.1 Update column block by fanning in updated entries from previous column blocks in other processors ● /
{ **same as in Parallel Factorization shown in Figure 8** }
/* Duplicate a copy of column block ● /
assign $K^{(b)}$ **to** $\bar{K}^{(b)}$
/* 11.2 parallel factorization of column block ● /
FOR each row $i \in \mathcal{C}^{(b)}$, DO
BEGIN
IF row $i \in$ *my-processor,* **THEN**
broadcast ($L^{(b)})^{-}_{i,:}$ **and** $D_{i,i}$ **to other processors sharing** $\mathcal{C}^{(b)}$
ELSE
receive $(L^{(b)})^{-1}_{i,:}$, $D_{i,i}$ **from other processors**
END.
assign D_{ii} **to** $D^{(b)}$; / * every processor has entire copy of $D^{(b)}$ */
/* update remaining entries in column block */
FOR each row $j \in \mathcal{D}^{(b)}$ **and** $j > i+1$, **DO**
BEGIN
IF row $j \in$ *my-processor* **THEN**
$\bar{L}^{(b)}_{j,i} = K^{(b)}_{j,i} - (L^{(b)})^{-1}_{i,:} \bar{K}^{(b)T}_{j,:i-1}$; /* dot product */
$L^{(b)}_{j,i} = -\bar{L}^{(b)}_{j,i} / D_{i,i}$;
$D_{j,j} = D_{j,j} + L^{(b)}_{j,i} * \bar{L}^{(b)}_{j,i}$;
$(L^{(b)})^{-1}_{j,:i-1} = (L^{(b)})^{-1}_{j,:i-1} + L^{(b)}_{j,i} * (L^{(b)})^{-1}_{i,:}$; /* dot product */
END.
FOR each row j $\in \mathcal{R}^{(b)}$, **DO**
BEGIN /* dot product */
IF row $j \in my\_processor$ **THEN** $\bar{R}^{(b)}_{j,:i-1} = \bar{R}^{(b)}_{j,:i-1} + (L^{(b)})^{-1}_{i,:} \bar{K}^{(b)}_{j,:i-1}$;
END.
END.
/* II.3 **Perform** dot products between row segments */
{ **same as in Parallel Factorization shown in Figure 8** }
END.
END.

Figure 12: Phase II of Parallel Factorization with Partial Factor Inverses

principal block submatrices with a matrix-vector multiplication and a summation of the resulting product over all processors sharing the column block. The communication is reduced to two global summations among the processors shared by the column block. By working from the bottom and up to the top of the column block, the procedure can be performed in place such that the solution vector can share the same memory locations as the original vector.

Similarly, the first phase of the parallel backward solution procedure requires very little changes. The new backward solve routine is summarized as shown in Figure 14. Like the forward solve, after the matrix product is calculated in each processor, the vector is summed across all processors sharing the block to form the entire matrix vector product. When multiplying the transpose of a submatrix by a vector, we begin the computations at the top of the block and work down to the bottom of the block so that the solution vector can overwrite the original vector. Again, we reduce the communication to two global summations among the processors shared by the column block.

6 Experimental Results

The procedures described in this paper have been implemented in a finite element program written in the C programming language and run on an Intel iPSC/i860 hypercube computer. Version 2.0 of the compiler and optimized level 1 BLAS routines were used. The level 1 BLAS routines include vector operations, such as dot products and axpy procedures. It is interesting to note that on the RISC i860 processor, the dot product operation is generally more efficient than the axpy operation. The row oriented factorization scheme discussed here includes mainly vector dot products.

In this section, we present the results on three different finite element models that we have used to evaluate the parallel sparse factorization and solution procedures. The three finite element models include a set of square finite element grids, structural domes and a high speed civil transport model. These models are ordered using various nested dissection schemes to re-number the sparse systems of equations.

6.1 Solution of Square Finite Element Grid Models

Our first experiment deals with the solution of a number of square finite element grid models of sizes ranging from 80 by 80 to 240 by 240 elements. The square grid problems are ordered using four levels of coordinate nested dissection which recursively partitions the grid into smaller square subgrids. The coordinate nested dissection of the square grid problem provides a very regular and well balanced work load distribution to run on a parallel computer. Each processor is responsible for approximately the same number of elements and equations. The good load balance is intended to minimize the synchronization and processor idling overheads so that we can examine the procedures in detail. The number of equations, the number of nonzero entries

```
Procedure: Parallel Forward Solve Lz = f (Partial Factor Inverses) - Phase II
BEGIN
   FOR each column block b , DO
   BEGIN
      /* 1. update solution vector */
      broadcast f_i^(b), i ∈ D^(b) to other processors sharing C^(b);
      Receive f^(rec) and accumulate f^(b) = f^(b) + f^(rec) ;
         /* Note: every processor has a copy of f_i^(b) ● /
      /* 2. parallel forward solve - matrix-vector multiply */
      FOR each row i ∈ D^(b) STEP -1, DO
      BEGIN
         IF i ∈ my_processor THEN z_i = (L^(b))^-1_i,: * f_:i;     /* dot product */
      END.
      broadcast z^(b) to other processors sharing C^(b);
      receive z^(rec) and accumulate: z^(b) = z^(b) + z^(rec);
         /* Note: every processor has a copy of z^(b) ● /
      /* 3. update vector by row segments ● /
      FOR each row segment i ∈ R^(b), DO
      BEGIN
         f_i = f_i - R^(b)_i,: * z^(b);
      END. END.
END.
```

Figure 13: Phase II of Parallel Forward Solve with Partial Factor Inverses

```
Procedure: Parallel Backward Solve DL^T x = z (Partial Factor Inverses) - Phase I
BEGIN
   /* Initialize solution vector in each processor */
   FOR each row i = 1 TO n, DO
   BEGIN
      IF i ∈ my_processor THEN
         z_i = z_i / D_{i,i};
      ELSE
         z_i = 0;
      ENDIF.
   END.
   FOR each column block b = nblock, . . . STEP - 1 DO
   BEGIN
      /* 1. update the solution vector z^(b) by row segments */
      FOR each row j ∈ R^(b) and j > i, DO
      BEGIN
         z^(b) = z^(b) - x_j * R^(b)T_{j,:};      /* axpy operation ● /
      END.
      broadcast z^(b) to other processors sharing C^(b) ;
      receive z^(rec) and accumulate: z^(b) = z^(b) + z^(rec); •
         /* Note: every processor has a copy of z^(b) */
      /* 2. parallel profile backward solve : matrix-vector multiply*/
      FOR each row i ∈ D^(b), DO
      BEGIN
         IF i ∈ my_processor THEN
            a ⟵ z_i;
            z^(b) = z^(b) + α * (L^(b))^-T_{i,:}      /* axpy operation */
         ENDIF.
      END.
      /* 3. distribute the solution vector */
      broadcast z^(b) to other processors sharing C^(b);
      receive z^(rec) and accumulate: x^(b) = z^(b) + z^(rec);
      /* Note: every processor has a copy of x^(b) */
   END.
END.
```

Figure 14: Phase I of Parallel Backward Solve with Partial Factor Inverses

in the matrix factors and the square grid models are shown in Figure 15.

Tables 1 and 2 summarize the results of the factorization and solution procedures for the direct LDL^T factorization and the factorization with partial factor inverses, respectively. We first examine the results on the factorization of the sparse matrices. The results are plotted in Figure 16, showing the computing time verses the number of equations for different number of processors. It is interesting to note that, for these square grid problems, the computing times for the factorization schemes with and without the factor inverses are in general comparable. When sixteen or thirty-two processors are utilized, it actually takes slightly less time for the factorization with partial factor inverses than the direct $LD\ L^T$ factorization. This rather unexpected result may be due to the following reasons in our implementation:

1. When the normal matrix factor L is formed? the principal block submatrix is assumed to be a variable banded structure. Thus, the length of each row is checked every time it is used or updated. For the factor inverses, since it is known that the principal block submatrices are dense, the row lengths are not checked.

2. Another contributing factor may concern with the on-chip cache. When the column block is distributed over a large number of processors, each processor stores less amount of data so that large portion of the column block can be kept in the cache. The extra operations to form the factor inverses can all work out of the cache and take advantage of data locality. This may explain why the better performance of the factorization with partial factor inverses occurs only for larger number of processors.

For such well balanced square grid problems, it appears that the factor inverses of the principal block submatrices can be formed with very little costs.

Figures 17 and 18 summarize the performance of the triangular solvers for, respectively, the forward and the backward solutions. The forward solution procedures are generally more efficient than the backward solution procedures. This result may be due to the following reasons:

1. The forward solves have less data dependencies, permit more computations to be done in parallel, and require less communication.

2. The forward solution procedure is mainly composed of vector dot products rather than axpy operations.

It is also clear that the factor inverses significantly decrease the solution time for the triangular systems. In fact, for moderate size problems, only the triangular solves with factor inverses show any speed up as the number of processors increases.

6.2 Solution of Structural Dome Problems

The second experiment deals with the solution of a number of structural dome models. The models are ordered by a spectral nested dissection scheme which generally works well on partitioning

elements per side	number of elements	number of equations	number of nonzeros (in matrix factor)
80	**6,400**	13,114	041,951
100	**10,000**	20,394	**1,450,027**
120	14,400	29,274	**2,221,911**
150	**22,500**	45,594	**3,736,351**
180	32,400	65,514	**5,698,375**
200	**40,000**	80,794	**7,157,575**
212	**44,944**	90,730	**8,194,811**
240	57,600	116,154	10.951~75

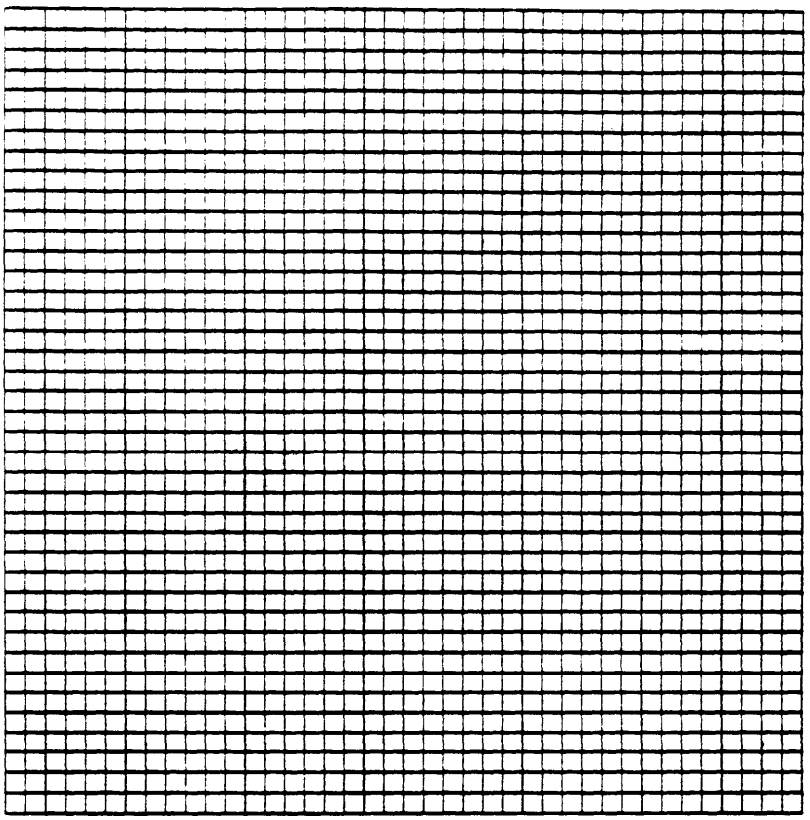


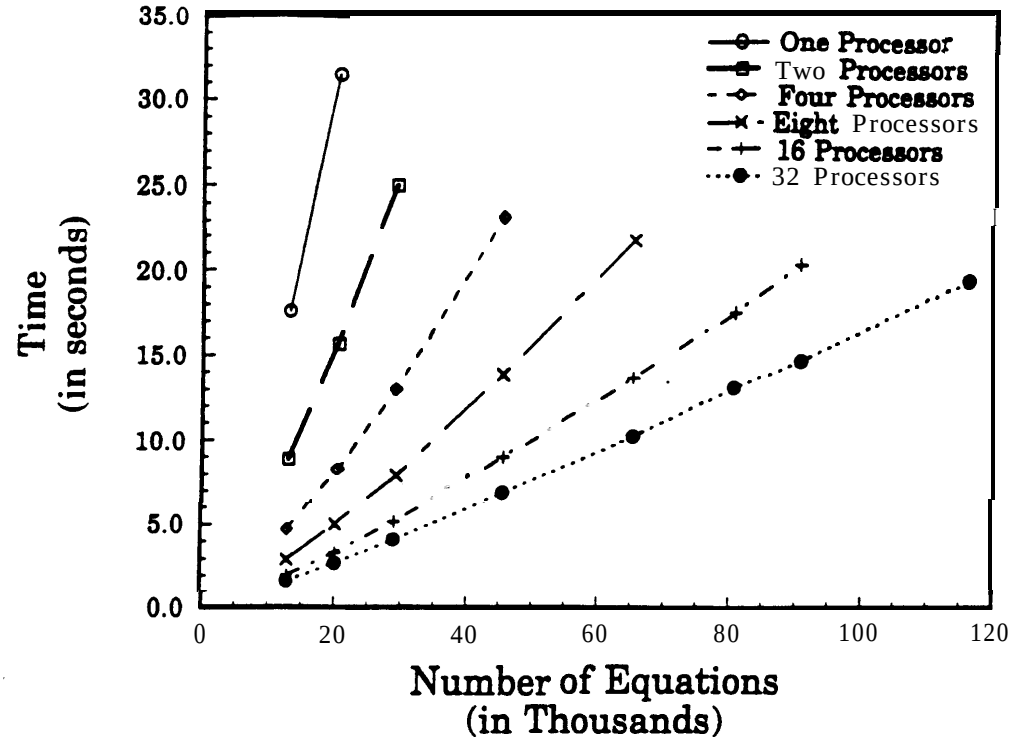
Figure 15: Square Plane Stress Finite Element Grid Model

Table 1: Square FEM Grid Models: Direct LDL^T Factorization (Time in seconds)

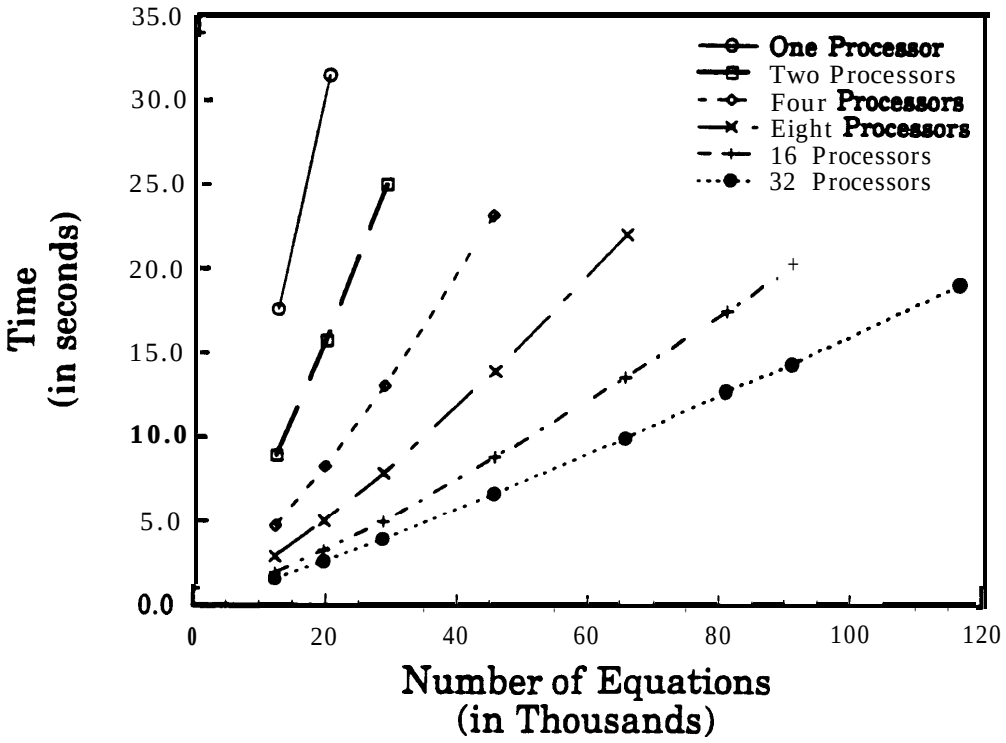
Number of processors	LDL^T Factorization	Forward solve	Back solve
	80 by 80 mesh		
1 PROCESSOR	17.579	0.228	0.448
2 PROCESSORS	8.840	0.127	0.232
4 PROCESSORS	4.709	0.072	0.131
8 PROCESSORS	2.939	0.051	0.090
16 PROCESSORS	2.022	0.044	0.080
32 PROCESSORS	1.667	0.048	0.088
	100 by 100 mesh		
1 PROCESSOR	31.473	0.358	0.745
2 PROCESSORS	15.582	0.195	0.383
4 PROCESSORS	8.189	0.109	0.210
8 PROCESSORS	5.021	0.073	0.137
16 PROCESSORS	3.358	0.059	0.114
32 PROCESSORS	2.726	0.064	0.122
	120 by 120 mesh		
2 PROCESSORS	24.789	0.284	0.570
4 PROCESSORS	12.941	0.158	0.306
8 PROCESSORS	7.821	0.100	0.191
16 PROCESSORS	5.149	0.077	0.150
32 PROCESSORS	4.118	0.079	0.155
	150 by 150 mesh		
4 PROCESSORS	22.963	0.251	0.496
8 PROCESSORS	13.805	0.154	0.298
16 PROCESSORS	8.865	0.111	0.219
32 PROCESSORS	6.744	0.105	0.212
	180 by 180 mesh		
8 PROCESSORS	21.555	0.229	0.410
16 PROCESSORS	13.524	0.160	0.296
32 PROCESSORS	10.086	0.159	0.285
	200 by 200 mesh		
16 PROCESSORS	17.379	0.193	0.356
32 PROCESSORS	12.943	0.186	0.331
	212 by 212 mesh		
16 PROCESSORS	20.167	0.212	0.397
32 PROCESSORS	14.506	0.189	0.354
	240 by 240 mesh		
32 PROCESSORS	19.160	0.243	0.434

Table 2: Square FEM Grid Models: Factorization with Partial Factor Inverses (Time in seconds)

Number of processors	LDL^T Factorization	Forward solve	Back solve
	80 by 80 mesh		
1 PROCESSOR	17.579	0.228	0.448
2 PROCESSORS	8.906	0.116	0.228
4 PROCESSORS	4.733	0.064	0.122
8 PROCESSORS	2.942	0.043	0.073
16 PROCESSORS	1.996	0.036	0.052
32 PROCESSORS	1.639	0.038	0.049
	100 by 100 mesh		
1 PROCESSOR	31.473	0.358	0.745
2 PROCESSORS	15.742	0.183	0.378
4 PROCESSORS	8.257	0.099	0.199
8 PROCESSORS	5.056	0.063	0.116
16 PROCESSORS	3.324	0.049	0.077
32 PROCESSORS	2.670	0.050	0.065
	120 by 120 mesh		
2 PROCESSORS	25.063	0.268	0.564
4 PROCESSORS	13.081	0.142	0.294
8 PROCESSORS	7.877	0.086	0.166
16 PROCESSORS	5.106	0.063	0.105
32 PROCESSORS	4.012	0.060	0.084
	150 by 150 mesh		
4 PROCESSORS	23.241	0.231	0.480
8 PROCESSORS	13.884	0.136	0.266
16 PROCESSORS	8.760	0.091	0.163
32 PROCESSORS	6.610	0.077	0.119
	180 by 180 mesh		
8 PROCESSORS	22.004	0.189	0.390
16 PROCESSORS	13.548	0.120	0.229
32 PROCESSORS	9.915	0.097	0.158
	200 by 200 mesh		
16 PROCESSORS	17.504	0.149	0.280
32 PROCESSORS	12.732	0.115	0.192
	212 by 212 mesh		
16 PROCESSORS	20.164	0.163	0.318
32 PROCESSORS	14.336	0.122	0.206
	240 by 240 mesh		
32 PROCESSORS	19.061	0.147	0.257

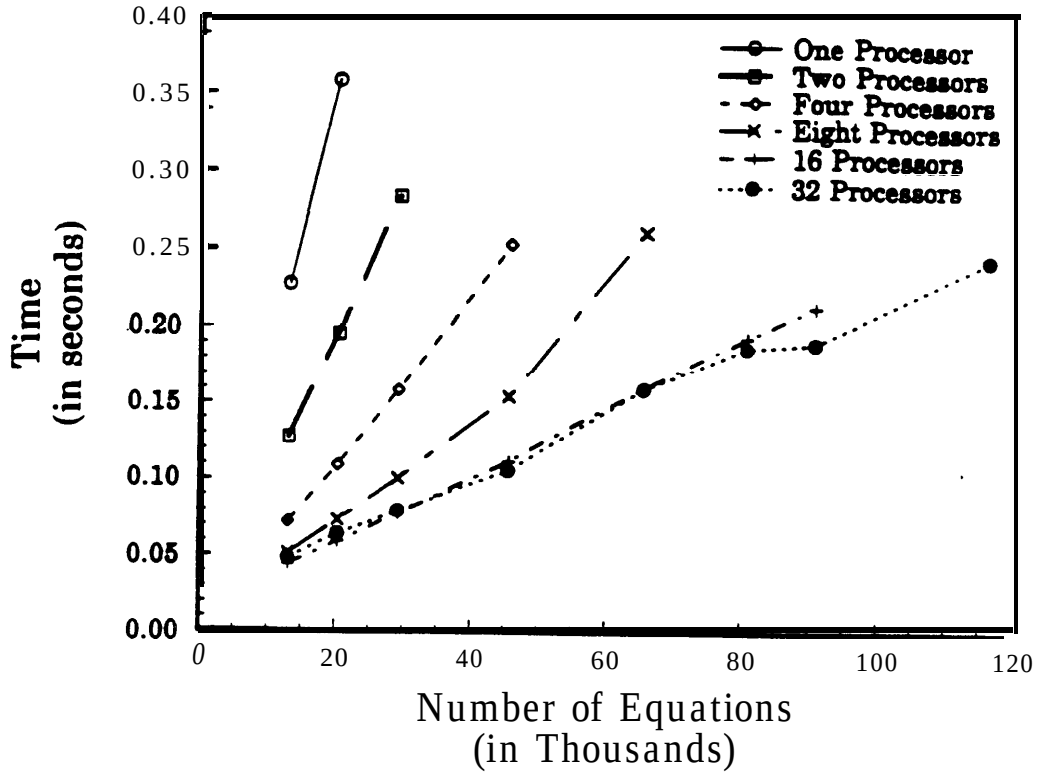


(a) Direct LDL^T Factorization

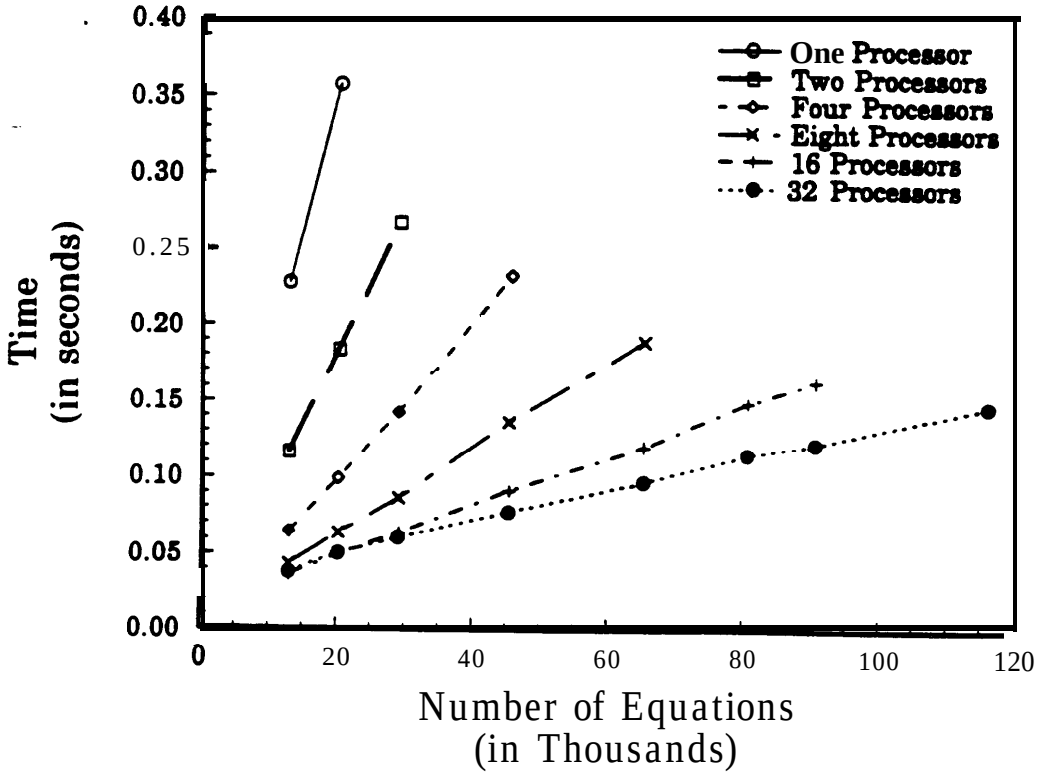


(b) Factorization with Partial Factor Inverses

Figure 16: Timings for parallel factorization

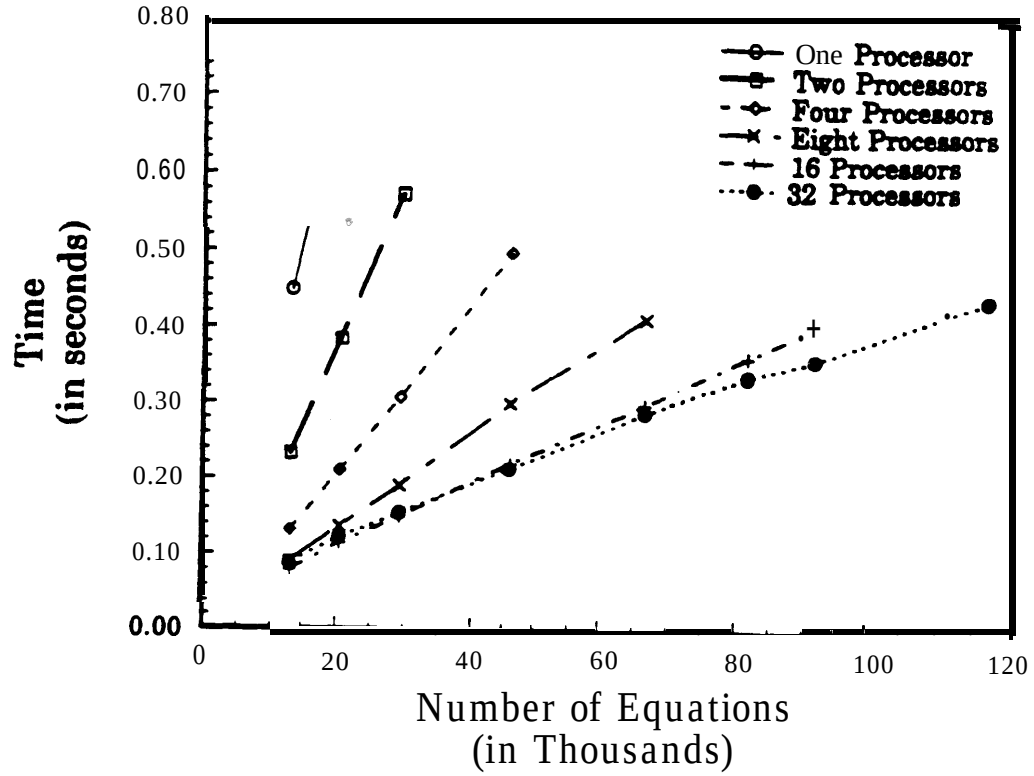


(a) Foward **Solve with LDL^T Factorization**

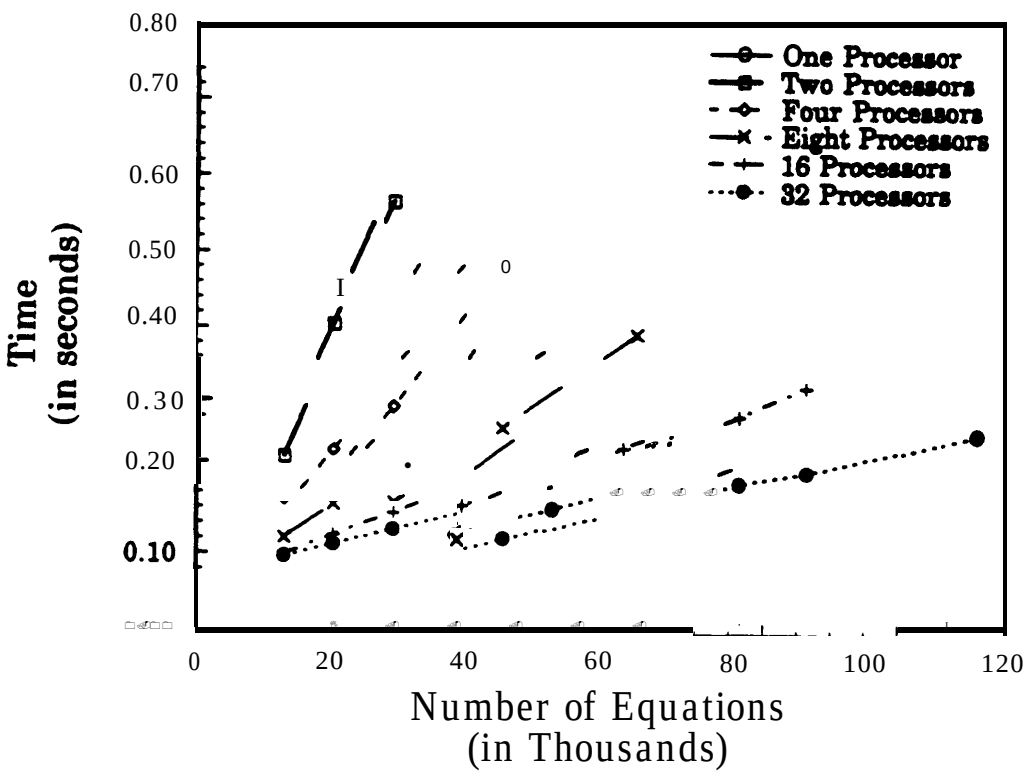


(b) Forward Solve with Partial Factor Inverses

Figure 17: Timings for Parallel Forward Solves



(a) Backward Solve with LDL^T **Factorization**



(b) Backward Solve with Partial Factor Inverses

Figure 18: Timings for Parallel Backward Solves

irregular models [18]. The matrix size and the number of nonzero entries for the dome problems are summarized in Figure 19. Tables 3 and 4 summarize the results of the factorization and solution procedures for, respectively, the direct LDL^T factorization and the factorization with fact or inverses.

For this problem, the load balance is not as good as the square grid problems discussed above. One indication is the time for the different processors to complete the backward solves. In the backward solve column of Tables 3 and 4, the results within the parenthesis show the time for the first processor to complete the backward solve while the results without the parenthesis show the time for the last processor to complete the computations. The two results show a slight difference in the completion time for the backward solve among the processors.

As the results indicate, the efficiency for forming the partial factor inverses depends on the number of processors used, the problem size and the number of right-hand side vectors. For example, the solution of 2 to 4 right-hand side vectors would compensate for the extra cost involved in forming the partial factor inverses when sixteen or thirty-two processors are used. However, when only two processors are used, it may require up to fifty-two right-hand side vectors to compensate for the extra cost of forming the factor inverses for the same problem. In general, the benefit in forming the factor inverses increases as both the number of right- hand side vectors and the number of processors increase.

6.3 Solution of High Speed Civil Transport Model

The third experiment deals with the solution of a High Speed Civil Transport plane model. This model does not yield to good load balance for a number of re-ordering schemes that we have experimented with. Here, we show the results based on an incomplete nested dissection ordering [6]. Figure 20 shows the model, the number of equations and the number of nonzero entries in the matrix factor. Tables 5 and 6 summarize the results of the factorization and solution procedures for, respectively, the direct LDL^T factorization and the factorization with partial factor inverses. As noted in the timings for the backward solves, there is a relatively large difference in the timings between the first and the last processors to complete the solution. This relative difference indicates the poor load balance of the model based on the ordering scheme used. It is again evident that the factorization with partial factor inverses provides significant improvements on the forward and backward solution of triangular systems. This result is particularly important for problems that require the solution of multiple right-hand sides vectors.

7 Summary and Discussion

In this paper, we have introduced a row-oriented sparse matrix factorization scheme for distributed memory parallel computers. This row-oriented scheme is formulated to take advantage of the RISC i860 processor architecture which performs vector dot products more efficiently than the

Dome number	number of elements	number of equations	number of nonzeros (in matrix factor)
1	1,250	3,606	338,925
2	1,800	5,226	538,707
3	5,400	16,026	2,028,495

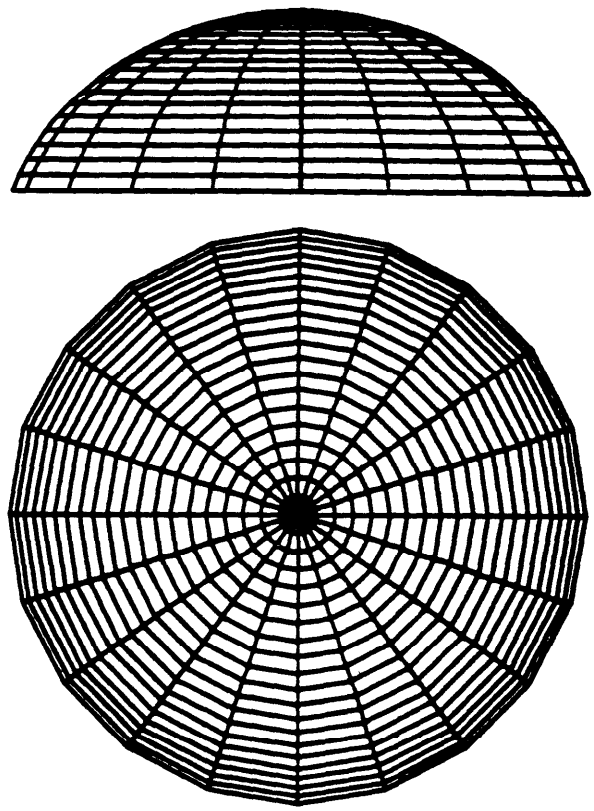


Figure 19: Structural Dome Models

Table 3: Structural Dome Models: Direct LDL^T Factorization (Time in seconds)

Number of processors	LDL^T Factorization	Forward solve	Back solve
Dome 1 (3,606 equations)			
1 PROCESSOR	7.358	0.076	0.166
2 PROCESSORS	4.248	0.047	0.097 (0.080)
4 PROCESSORS	2.775	0.034	0.062 (0.048)
8 PROCESSORS	1.992	0.030	0.055 (0.045)
16 PROCESSORS	1.594	0.031	0.061 (0.052)
Dome 2 (5,226 equations)			
1 PROCESSOR	12.449	0.115	0.253
2 PROCESSORS	6.880	0.068	0.142 (0.122)
4 PROCESSORS	4.423	0.048	0.088 (0.066)
8 PROCESSORS	3.393	0.039	0.074 (0.055)
16 PROCESSORS	2.754	0.040	0.079 (0.063)
32 PROCESSORS	2.303	0.049	0.072 (0.091)
Dome 3 (16,026 equations)			
2 PROCESSORS	26.065	0.205	0.474 (0.452)
4 PROCESSORS	14.887	0.118	0.264 (0.220)
8 PROCESSORS	9.118	0.078	0.169 (0.123)
16 PROCESSORS	6.261	0.065	0.136 (0.104)
32 PROCESSORS	5.026	0.069	0.133 (0.112)

Table 4: Structural Dome Models: Factorization with Partial Factor Inverses (Time in seconds)

Number of processors	LDL^T Factorization	Forward solve	Back solve
Dome 1 (3,606 equations)			
1 PROCESSOR	7.358	0.076	0.166
2 PROCESSORS	4.362	0.044	0.094 (0.078)
4 PROCESSORS	2.946	0.029	0.054 (0.042)
8 PROCESSORS	2.067	0.027	**0.041 (0.031)**
16 PROCESSORS	1.606	0.029	0.040 (0.031)
Dome 2 (5,226 equations)			
1 PROCESSOR	12.449	0.115	0.253
2 PROCESSORS	7.065	0.064	0.140 (0.120)
4 PROCESSORS	4.651	0.041	0.079 (0.057)
8 PROCESSORS	3.460	0.035	0.056 (0.038)
16 PROCESSORS	2.823	0.035	0.048 (0.035)
32 PROCESSORS	2.329	**0.044**	0.052 (0.040)
Dome 3 (16,026 equations)			
2 PROCESSORS	26.320	0.202	0.472 (0.449)
4 PROCESSORS	15.147	0.112	0.256 (0.212)
8 PROCESSORS	9.337	0.071	0.150 (0.104)
16 PROCESSORS	6.398	0.057	0.101 (0.071)
32 PROCESSORS	5.240	0.057	0.083 (0.065)

number of **equation8**	number of nonzeros **(in matrix factor)**
16,146	**3,783,784**

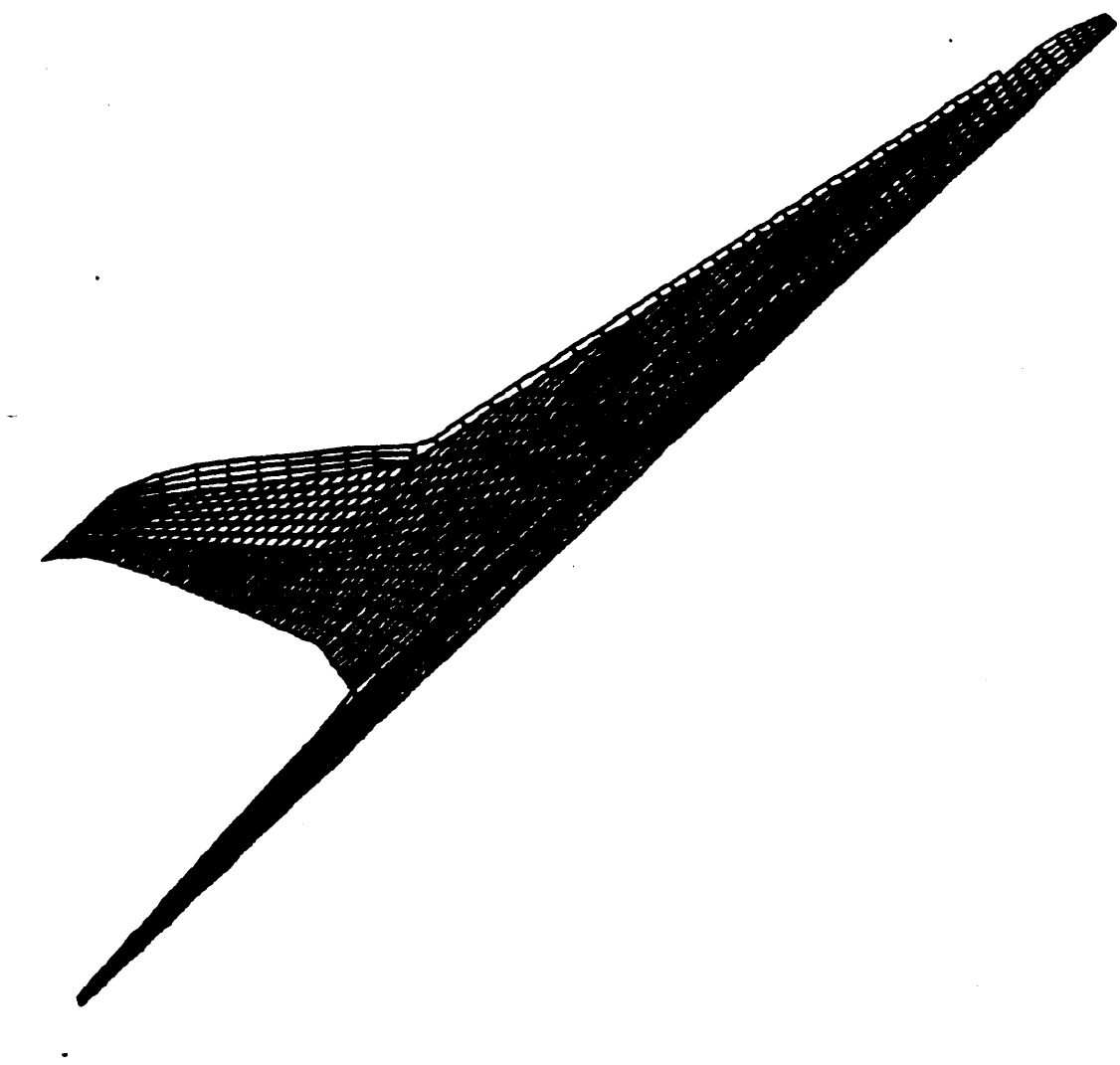


Figure 20: A High Speed Civil Transport Model (Courtesy of Dr. Olaf Storaasli of NASA Langley Research Center)

Table 5: High Speed Civil Transport Model: Direct LDL^T Factorization (Time in seconds)

Number of processors	LDL^T Factorization	Forward solve	Back solve
4 PROCESSORS	37.694	0.157	0.488 (0.332)
8 PROCESSORS	22.463	0.106	0.306 (0.175)
16 PROCESSORS	15.436	0.093	0.268 (0.150)
32 PROCESSORS	12.262	0.166	0.285 (0.190)

Table 6: High Speed Civil Transport Model: Factorization with Partial Factor Inverses (Time in seconds)

Number of processors	LDL^T Factorization	Forward solve	Back solve
4 PROCESSORS	39.084	0.134	0.474 (0.317)
8 PROCESSORS	23.201	0.086	0.265 (0.140)
16 PROCESSORS	15.801	0.080	0.194 (0.078)
32 PROCESSORS	12.392	0.146	0.155 (0.078)

vector axpy operations. We have shown that it is possible to obtain good speed up using a row-oriented factorization scheme. In our current implementation. the reordering of the finite element model and the matrix partitioning are performed on a workstation as a pre-processing step. The objective is to test the efficiencies of the direct solution procedures.

Based on our experimental results, a relatively good speed up for parallel factorization can be achieved when there is a good load balance. Furthermore, we need a sufficient size problem in order to utilize the parallel computer effectively. For the Intel's iPSC/i860 hypercube, we need at least 1500 equations per processor in order to use each individual processor efficiently. For the models that we have used in the experiments, for example, it is not optimal to assign more than eight processors to the last column block (with only the principal block submatrix) on the iPSC/i860 hypercube. As the number of processors increases, the problem size also need to increase to fully utilize the multiple processors. As processor speed changes and message communication costs improve, the number of equations required to efficiently use the processors and the number of processors to assign a column block will change.

We have developed a strategy to invert the dense submatrix factors that are shared among multiple processors. Although the number of operations required for the factorization increases slightly, the number of communications remain the same with or without the factor inverses. With the dense factor inverses of the principal block submatrices, the parallel solution of triangular systems can be made more efficiently, and higher parallelism for the triangular solution can be recognized. The benefit of this factorization with partial factor inverses can be exemplified with problems, such as generalized eigenvalue problems, that involve the solution of multiple right-hand side vectors [17].

References

[1] F. L. Alvarado and R. Schreiber. Optimal parallel solution of sparse triangular systems. Technical Report 90.36, Research Institute for Advanced Computer Science, NASA Ames Research Center, Mountain View, CA, 1990.

[2] R.E. Bank and R. K. Smith. General sparse elimination requires no permanent integer storage. *SIAM J. Sci. Statis. Comput.*, 8:574–584, 1987.

[3] K. J. Bathe. *Finite Element Procedures* in Engineering *Analysis.* Prentice Hall, Englewood Cliffs, NJ, 1982.

[4] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct* Methods for *Sparse Matrices.* Oxford Science Publications, Oxford, 1986.

[5] S.C. Eisenstat, M.C. Gursky, M.H. Schultz, and A.H. Sherman. The Yale sparse matrix package, 1. the symmetric code. *Int. J. Numer. Meth. Engrg.*, 18:1145–1151, 1982.

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

[7] J. A. George, J. W. H. Liu., and E. G. -Y. Ng Communication results for parallel sparse Cholesky factorization. *Parallel Computing,* 10:287-298, 1989.

[8] G. H. Golub and C. F. Van Loan. *Matrix Computations. The* Johns Hopkins University Press, Baltimore and London, 1989.

[9] M. T. Heath, E. Ng, and B. Peyton. Parallel algorithms for sparse linear systems. *SIAM* Review, 33(3):420–460, September 1991.

[10] T.J.R. Hughes. *The Finite Element Method Linear Static and Dynamic* Finite *Element Analysis.* Prentice-Hall, Englewood Cliffs, NJ, 1987.

[11] A. Jennings. A compact storage scheme for the solution of symmetric linear simultaneous equations. *Computer J.,* 8:351–361, 1966.

[12] K. H. Law and S. J. Fenves. A node-addition model for symbolic factorization. *ACM Trans. Math. Software,* 12:37–50, 1986.

[13] J. W. H. Liu. A compact row storage scheme for Cholesky factors using elimination trees. *ACM Tmn. Math. Software*, 12:127–148, 1986.

[14] J. W. H. Liu. A generalized envelope method for sparse factorization by rows. Technical Report CS-88-09, Department of Computer Science, York University, Canada, 1988.

[15] J. W. H. Liu. The multifrontal method for sparse matrix solution, theory and practice. Technical Report CS90-04, Department of Computer Science, York University, Canada, 1990.

[16] D. R. Mackay and K. H. Law. An implementation of a generalized sparse/profile finite element solution method. *Computers and Structures,* 41:723-737, 1991.

[17] D. R. Mackay and K.H. Law. A parallel implementation of Lanczos algorithm for structural dynamic analysis on distributed memory computers. 1992.

[18] A. Pothen, H. Simon, and L. Wang. Spectral nested dissection Technical Report RNR-92-003, NASA Ames Research Center, Moffett Field, CA, 1992.

[19] R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM* Tmns. Math. *Software,* 8:256–276, 1982.