A short version won an **Outstanding Student Paper Award** in Supercomputing '94.

A Scalable Parallel Algorithm for Sparse Matrix Factorization*

Anshul Gupta and Vipin Kumar

Department of Computer Science
University of Minnesota
Minneapolis, MN - 55455

agupta@cs.umn.edu *kumar@cs.umn.edu*

TR 94-19, April 1994)

Abstract

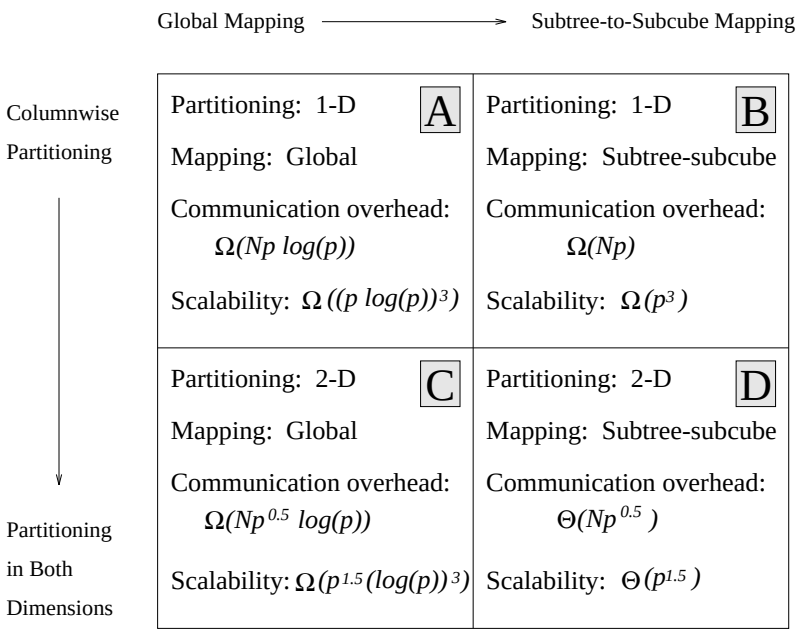
In this paper, we describe a scalable parallel algorithm for sparse matrix factorization, analyze its performance and scalability, and present experimental results of its implementation on a 1024-processor nCUBE2 parallel computer. Through our analysis and experimental results, we demonstrate that our algorithm improves the state of the art in parallel direct solution of sparse linear systems by an order of magnitude—both in terms of speedups and the number of processors that can be utilized effectively for a given problem size. An implementation of our algorithm on nCUBE2 delivers a speedup of up to 364 on 1024 processors for a relatively small problem. A variation of this algorithm [21] implemented on a 1024-processor Cray T3D delivers up to 20 GFLOPS on medium-size structural engineering and linear programming problems.

This algorithm incurs less communication overhead and is more scalable than any known parallel formulation of sparse matrix factorization. We show that, for a wide class of sparse matrices (including those arising from two- and three-dimensional finite element problems), our algorithm is optimally scalable on hypercube and mesh architectures and its asymptotic scalability is the same as that of dense matrix factorization. The key features of our algorithm are that it is based on the multifrontal technique, it uses subtree-to-subcube mapping scheme, and it partitions the matrix in two dimensions along both rows and columns. Although, in this paper, we discuss Cholesky factorization of symmetric positive definite matrices, the algorithm can be adapted for solving sparse linear least squares problems and for Gaussian elimination of diagonally dominant matrices that are almost symmetric in structure.

1 Introduction

This paper describes an optimally scalable parallel algorithm for factorization of sparse matrices. This algorithm incurs strictly less communication overhead than any known parallel formulation of sparse matrix factorization, and hence, can utilize a higher number of processors effectively. It is well known that dense matrix factorization can be implemented efficiently on distributed-memory parallel computers [10, 39, 12, 26]. However, despite inherent parallelism in sparse sparse direct methods, not much success has been achieved to date in developing their scalable parallel formulations [23, 47]. In this paper, we show that the parallel Cholesky factorization

*This work was supported by Army Research Office under contract # DA/DAAH04-93-G-0080 and by the University of Minnesota Army High Performance Computing Research Center under contract # DAAL03-89-C-0038. Access to the nCUBE2 at Sandia National Labs was provided via DeSRA consortium.



A: Column-wise 1-D partitioning of matrix.

B: Columns distributed via subtree-subcube mapping.

C: 2-D partitioning of matrix among processors.

D: Dense submatrices partitioned dynamically in two dimensions according to subtree-subcube mapping.

Figure 1: An overview of the performance and scalability of parallel algorithms for factorization of sparse matrices resulting from two-dimensional N-node grid graphs. Box D represents our algorithm, which is a significant improvement over other known classes of algorithms for this problem.

algorithm described here is as scalable as the best parallel formulation of dense matrix factorization on both mesh and hypercube architectures for a wide class of sparse matrices, including those arising in two- and three-dimensional finite element problems.

The performance and scalability analysis of our algorithm is supported by experimental results on up to 1024 processors of the nCUBE2 parallel computer. We have been able to achieve speedups of up to 364 on 1024 processors and 230 on 512 processors over a highly efficient sequential implementation for moderately sized problems from the Harwell-Boeing collection [7]. A variation of this algorithm [21] implemented on a 1024-processor Cray T3D delivers up to 20 GFLOPS on medium-size structural engineering and linear programming problems. To the best of our knowledge, this is the first parallel implementation of sparse Cholesky factorization that has delivered speedups of this magnitude and has been able to benefit from several hundred processors. In this paper, we focus on Cholesky factorization of symmetric positive definite matrices; however, the ideas presented here can be adapted for performing Gaussian elimination on diagonally dominant matrices that are almost symmetric in structure [9] and for solving sparse linear least squares problems [34].

It is difficult to derive analytical expressions for the number of arithmetic operations in factorization and for the size (in terms of number of nonzero entries) of the factor for general sparse matrices. This is because the computation and fill-in during the factorization of a sparse matrix is a function of the the number and position of nonzeros in the original matrix. In the context of the important class of sparse matrices that are adjacency matrices of graphs whose n-node subgraphs have $O(\sqrt{n})$-node separators (this class includes sparse matrices

arising out of all two-dimensional finite difference and finite element problems), the contribution of this work can be summarized by Figure 1. A simple fan-out algorithm [14] with column-wise partitioning of an $N \times N$ matrix of this type on p processors results in an $O(Np \log N)$ total communication volume [17] (box A). The communication volume of the column-based schemes represented in box A has been improved using smarter ways of mapping the matrix columns onto processors, such as, the subtree-to-subcube mapping [16] (box B). A number of column-based parallel factorization algorithms [32, 33, 4, 43, 44, 49, 14, 11, 24, 23, 47, 36] have a lower bound of $O(Np)$ on the total communication volume. Since the overall computation is only $O(N^{1.5})$ [15], the ratio of communication to computation of column-based schemes is quite high. As a result, these column-cased schemes scale very poorly as the number of processors is increased [47, 46]. In [2], Ashcraft proposes a fan-both family of parallel Cholesky factorization algorithms that have a total communication volume of $O(N\sqrt{p} \log N)$. A few schemes with two-dimensional partitioning of the matrix have been proposed [46, 45, 1, 37, 50, 18], and the total communication volume in the best of these schemes [46, 45] is $O(N\sqrt{p} \log p)$ (box C).

Most researchers so far have analyzed parallel sparse matrix factorization in terms of the total communication volume. It is noteworthy that, on any parallel architecture, the total communication volume is only a lower bound on the overall communication overhead. It is the total communication overhead that actually determines the overall efficiency and speedup, and is defined as the difference between the parallel processor-time product and the serial run time [22, 26]. The communication overhead can be asymptotically higher than the communication volume. For example, a one-to-all broadcast algorithm based on a binary tree communication pattern has a total communication volume of $m(p-1)$ for broadcasting m words of data among p processors. However, the broadcast takes $\log p$ steps of $O(m)$ time each; hence, the total communication overhead is $O(mp \log p)$ (on a hypercube). In the context of matrix factorization, the experimental study by Ashcraft [4] serves to demonstrate the importance of studying the total communication overhead rather than volume. In [4], the fan-in algorithm, which has a lower communication volume than the distributed multifrontal algorithm, has a higher overhead (and hence, a lower efficiency) than the multifrontal algorithm for the same distribution of the matrix among the processors.

In summary, the simple parallel algorithm with $O(Np \log p)$ communication volume (box A) has been improved along two directions—one by improving the mapping of matrix columns onto processors (box B) and the other by splitting the matrix along both rows and columns (box C). In this paper, we describe a parallel implementation of sparse matrix factorization that combines the benefits of improvements along both these lines. The total communication overhead of our algorithm is only $O(N\sqrt{p})$ for factoring an $N \times N$ matrix on p processors if it corresponds to a graph that satisfies the separator criterion. Our algorithm reduces the communication overhead by a factor of at least $O(\log p)$ over the best algorithm implemented to date. It is also significantly simpler in concept as well as in implementation, which helps in keeping the constant factors associated with the overhead term low. Furthermore, as we show in Section 5, this reduction in communication overhead by a factor of $O(\log p)$ results in an improvement in the scalability of the algorithm by a factor of $O((\log p)^3)$; i.e., the rate at which the problem size must increase with the number of processors to maintain a constant efficiency is lower by a factor of $O((\log p)^3)$. This can make the difference between the feasibility and non-feasibility of parallel sparse factorization on highly parallel ($p \geq 256$) computers.

The algorithm presented in this paper is based on the multifrontal principle [8, 31] and the computation is guided by an elimination tree. Independent subtrees of the elimination tree are initially assigned to individual processors. As the computation proceeds up the elimination tree, through pairwise exchanges of data, the matrix is progressively split both horizontally and vertically among the processors resulting in a two-dimensional partitioning. This strategy keeps the communication low in the initial stage of parallel factorization. In the later part, the increase in communication is balanced by a corresponding increase in computation per column, and at the same time the processors do not starve due to narrowing of the elimination tree.

The remainder of the paper is organized as follows. Section 2 describes the serial multifrontal algorithm for sparse matrix factorization. Section 3 describes our parallel algorithm based on multifrontal elimination. In Section 4, we derive expressions for the communication overhead of the parallel algorithm. In Section 5, we use the isoefficiency analysis [26, 28, 20] to determine the scalability of our algorithm and compare it with the scalability of other parallel algorithms for sparse matrix factorization. Section 6 contains some preliminary experimental results on an nCUBE2 parallel computer. Section 7 contains concluding remarks.

/*
A is the sparse $N \times N$ symmetric positive definite matrix to be factored. L is the lower triangular matrix such that $A = LL^T$ after factorization. $A = (a_{i,j})$ and $L = (l_{i,j})$, where $0 \leq i, j < N$. Initially, $l_{i,j} = 0$ for all i, j.
*/

1. **begin function** Factor(k)

2. $F^k := \begin{pmatrix} a_{k,k} & a_{k,q_1} & a_{k,q_2} & \cdots & a_{k,q_s} \\ a_{q_1,k} & 0 & 0 & \cdots & 0 \\ a_{q_2,k} & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{q_s,k} & 0 & 0 & \cdots & 0 \end{pmatrix}$;

3. **for** all i such that Parent(i) = k in the elimination tree of A, **do**
4. **begin**
5. Factor(i);
6. F^k := Extend_add(F^k, U^i);
7. **end**

/*
At this stage, F^k is a $(t+1) \times (t+1)$ matrix, where t is the number of nonzeros in the sub-diagonal part of column k of L. U^k is a $t \times t$ matrix. Assume that an index i of F^k or U^k corresponds to the index q_i of A and L.
*/

8. **for** $i := 0$ **to** t **do**
9. $l_{q_i,k} := F^k(i,0)/\sqrt{F^k(0,0)}$;
10. **for** $j := 1$ **to** t **do**
11. **for** $i := j$ **to** t **do**
12. $U^k(i,j) := F^k(i,j) - l_{q_i,k} \times l_{q_j,k}$;
13. **end function** Factor.

Figure 2: An elimination-tree guided recursive formulation of the serial multifrontal algorithm for Cholesky factorization of a sparse symmetric positive definite matrix A into LL^T, where L is a lower-triangular matrix. If r is the root of the postordered elimination tree of A, then a call to Factor(r) factors the matrix A.

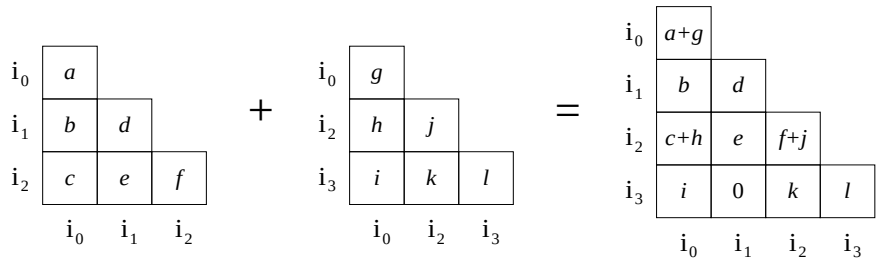


Figure 3: The extend-add operation on two 3×3 triangular matrices. It is assumed that $i_0 < i_1 < i_2 < i_3$.

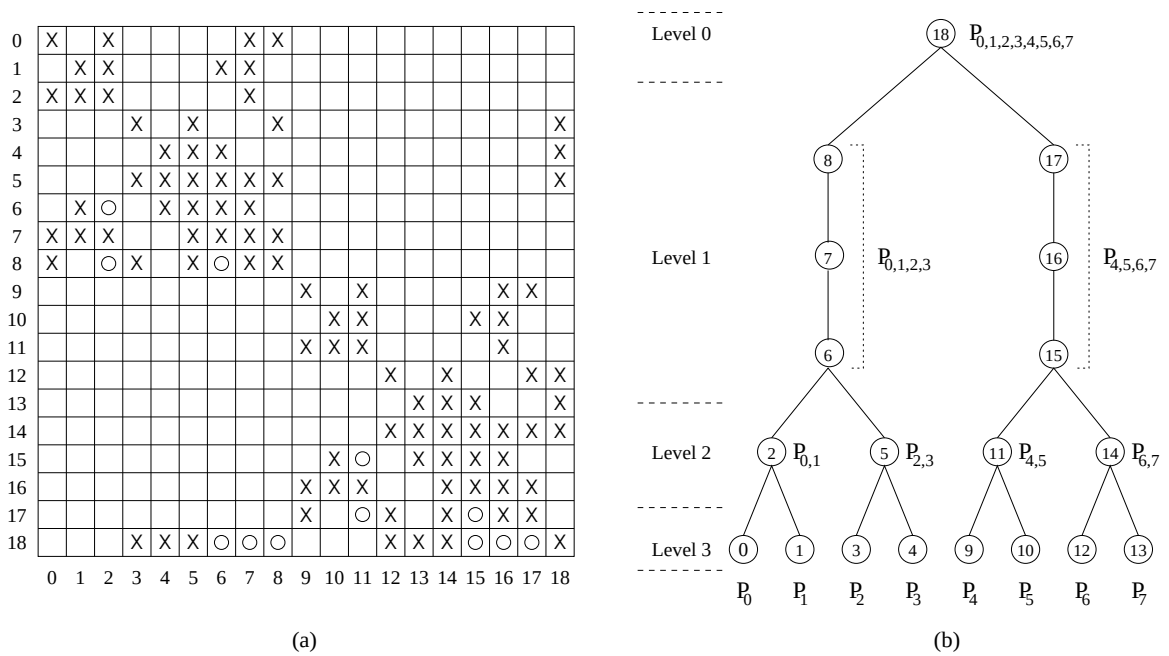


Figure 4: A symmetric sparse matrix and the associated elimination tree with subtree-to-subcube mapping onto 8 processors. The nonzeros in the original matrix are denoted by the symbol "×" and fill-ins are denoted by the symbol "∘".

2 The Multifrontal Algorithm for Sparse Matrix Factorization

The multifrontal algorithm for sparse matrix factorization was proposed independently, and in somewhat different forms, by Speelpening [48] and Duff and Reid [8], and later elucidated in a tutorial by Liu [31]. In this section, we briefly describe a condensed version of multifrontal sparse Cholesky factorization.

Given a sparse matrix and the associated elimination tree, the multifrontal algorithm can be recursively formulated as shown in Figure 2. Consider the Cholesky factorization of an $N \times N$ sparse symmetric positive definite matrix A into LL^T, where L is a lower triangular matrix. The algorithm performs a postorder traversal of the elimination tree associated with A. There is a frontal matrix F^k and an update matrix U^k associated with any node k. The row and column indices of F^k correspond to the indices of row and column k of L in increasing order.

In the beginning, F^k is initialized to an $(s+1) \times (s+1)$ matrix, where $s+1$ is the number of nonzeros in the lower triangular part of column k of A. The first row and column of this initial F^k is simply the upper triangular part of row k and the lower triangular part of column k of A. The remainder of F^k is initialized to all zeros. Line 2 of Figure 2 illustrates the initial F^k.

After the algorithm has traversed all the subtrees rooted at a node k, it ends up with a $(t+1) \times (t+1)$ frontal matrix F^k, where t is the number of nonzeros in the strictly lower triangular part of column k in L. The row and column indices of the final assembled F^k correspond to $t+1$ (possibly) noncontiguous indices of row

and column k of L in increasing order. If k is a leaf in the elimination tree of A, then the final F^k is the same as the initial F^k. Otherwise, the final F^k for eliminating node k is obtained by merging the initial F^k with the update matrices obtained from all the subtrees rooted at k via an extend-add operation. The extend-add is an associative and commutative operator on two update matrices such the index set of the result is the union of the index sets of the original update matrices. Each entry in the original update matrices is mapped onto some location in the accumulated matrix. If entries from both matrices overlap on a location, they are added. Empty entries are assigned a value of zero. Figure 3 illustrates the extend-add operation.

After F^k has been assembled through the steps of lines 3–7 of Figure 2, a single step of the standard dense Cholesky factorization is performed with node k as the pivot (lines 8–12). At the end of the elimination step, the column with index k is removed from F^k and forms the column k of L. The remaining $t \times t$ matrix is called the update matrix U^k and is passed on to the parent of k in the elimination tree.

The multifrontal algorithm is further illustrated in a step-by-step fashion in Figure 5 for factoring the matrix of Figure 4(a).

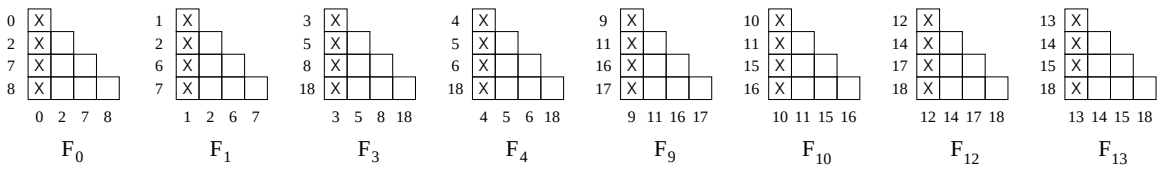
3 A Parallel Multifrontal Algorithm

In this section we describe the parallel multifrontal algorithm. We assume a hypercube interconnection network; however, the algorithm also can be adapted for a mesh topology (Section 3.2) without any increase in the asymptotic communication overhead. On other architectures as well, such as those of the CM-5, Cray T3D, and IBM SP-2, the asymptotic expression for the communication overhead remains the same. In this paper, we use the term *relaxed supernode* for a group of consecutive nodes in the elimination tree with one child. Henceforth, any reference to the height, depth, or levels of the tree will be with respect to the relaxed supernodal tree. For the sake of simplicity, we assume that the relaxed supernodal elimination tree is a binary tree up to the top $\log p$ relaxed supernodal levels. Any elimination tree can be converted to a binary relaxed supernodal tree suitable for parallel multifrontal elimination by a simple preprocessing step.

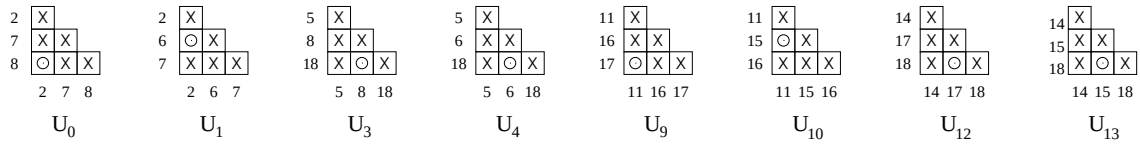
In order to factorize the sparse matrix in parallel, portions of the elimination tree are assigned to processors using the standard subtree-to-subcube assignment strategy. This assignment is illustrated in Figure 4(b) for eight processors. With subtree-to-subcube assignment, all p processors in the system cooperate to factorize the frontal matrix associated with the topmost relaxed supernode of the elimination tree. The two subtrees of the root are assigned to subcubes of $p/2$ processors each. Each subtree is further partitioned recursively using the same strategy. Thus, the p subtrees at a depth of $\log p$ relaxed supernodal levels are each assigned to individual processors. Each processor can work on this part of the tree completely independently without any communication overhead. A call to the function *Factor* given in Figure 2 with the root of a subtree as the argument generates the update matrix associated with that subtree. This update matrix contains all the information that needs to be communicated from the subtree in question to other columns of the matrix.

After the independent factorization phase, pairs of processors (P_{2j} and P_{2j+1} for $0 \leq j < p/2$) perform a parallel extend-add on their update matrices, say Q and R, respectively. At the end of this parallel extend-add operation, P_{2j} and P_{2j+1} roughly equally share $Q + R$. Here, and in the remainder of this paper, the sign "+" in the context of matrices denotes an extend-add operation. More precisely, all even columns of $Q + R$ go to P_{2j} and all odd columns of $Q + R$ go to P_{2j+1}. At the next level, subcubes of two processors each perform a parallel extend-add. Each subcube initially has one update matrix. The matrix resulting from the extend-add on these two update matrices is now merged and split among four processors. To effect this split, all even rows are moved to the subcube with the lower processor labels, and all odd rows are moved to the subcube with the higher processor labels. During this process, each processor needs to communicate only once with its counterpart in the other subcube. After this (second) parallel extend-add each of the processors has a block of the update matrix roughly one-fourth the size of the whole update matrix. Note that, both the rows and the columns of the update matrix are distributed among the processors in a cyclic fashion. Similarly, in subsequent parallel extend-add operations, the update matrices are alternatingly split along the columns and rows.

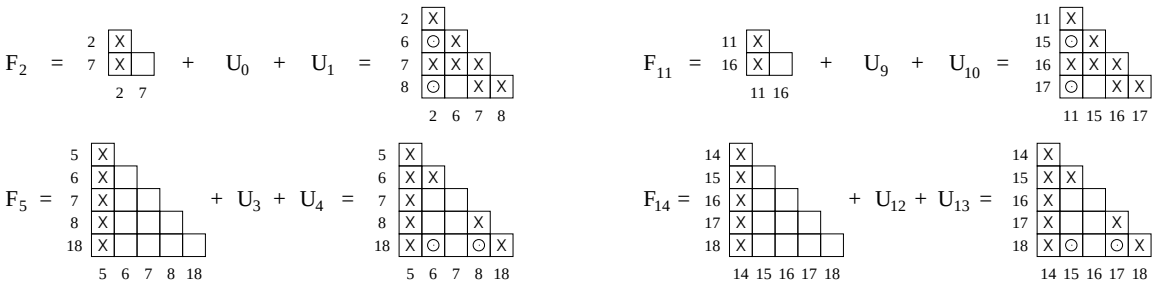
Assume that the levels of the binary relaxed supernodal elimination tree are labeled starting with 0 at the top as shown in Figure 4(b). In general, at level l of the relaxed supernodal elimination tree, $2^{\log p - l}$ processors work on a single frontal or update matrix. These processors form a logical $2^{\lfloor (\log p - l)/2 \rfloor} \times 2^{\lceil (\log p - l)/2 \rceil}$ mesh. All update and frontal matrices at this level are distributed on this mesh of processors. The cyclic distribution of



(a) Initial frontal matrices

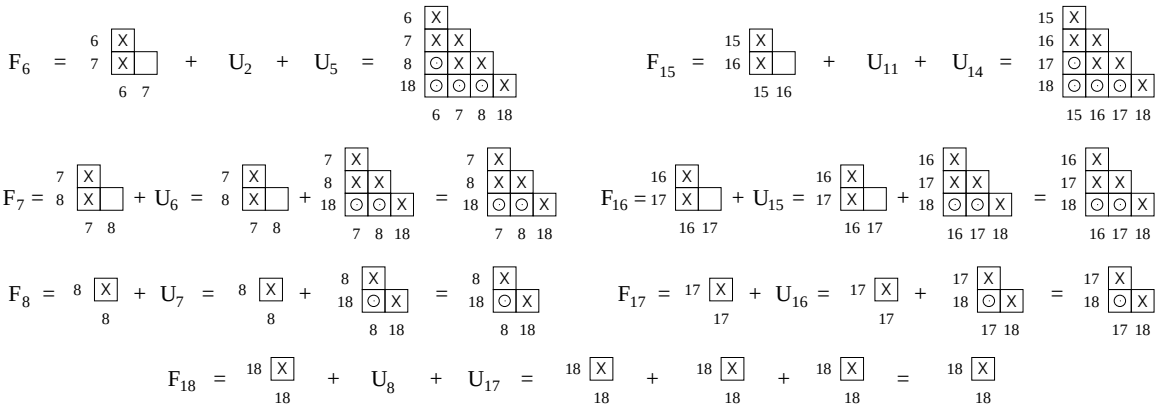


(b) Update matrices after one step of elimination in each frontal matrix



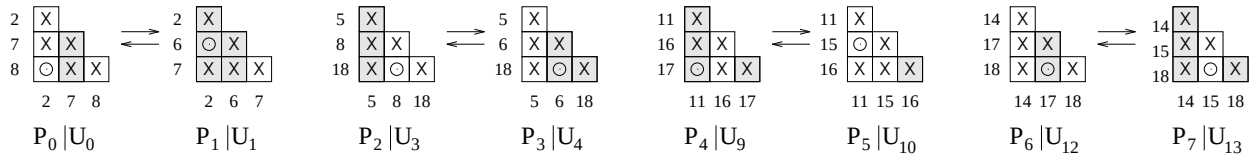
(c) Frontal matrices at the second level of the elimination tree

(d) Update matrices after elimination of nodes 2, 5, 11, and 14

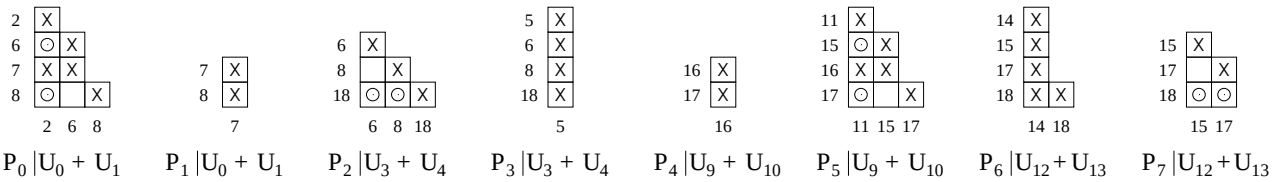


(e) Factorization of the remainder of the matrix

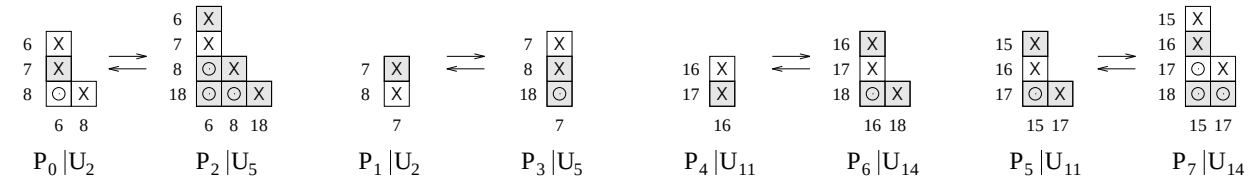
Figure 5: Steps in serial multifrontal Cholesky factorization of the matrix shown in Figure 4(a). The symbol "+" denotes an extend-add operation. The nonzeros in the original matrix are denoted by the symbol "×" and fill-ins are denoted by the symbol "∘".



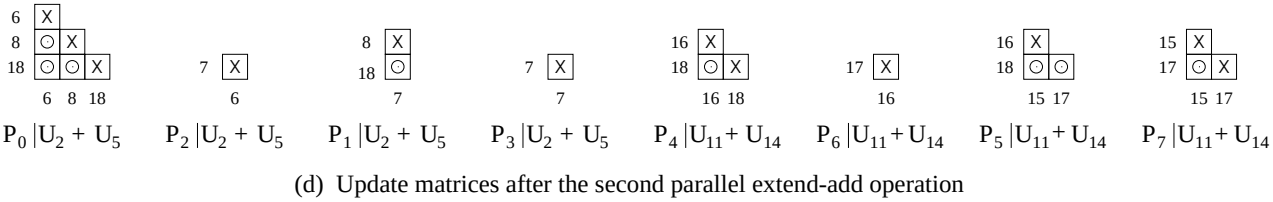
(a) Update matrices before the first parallel extend-add operation



(b) Update matrices after the first parallel extend-add operation



(c) Update matrices before the second parallel extend-add operation



(d) Update matrices after the second parallel extend-add operation

(e) Update matrices before the third parallel extend-add operation

(f) Update matrices after the third parallel extend-add operation

Figure 6: Extend-add operations on the update matrices during parallel multifrontal factorization of the matrix shown in Figure 4(a) on eight processors. $P_i|M$ denotes the part of the matrix M that resides on processor number i. M may be an update matrix or the result of performing an extend-add on two update matrices. The shaded portions of a matrix are sent out by a processor to its communication partner in that step.

rows and columns of these matrices among the processors helps maintain load-balance. The distribution also ensures that a parallel extend-add operation can be performed with each processor exchanging roughly half of its data only with its counterpart processor in the other subcube. This distribution is fairly straightforward to maintain. For example, during the first two parallel extend-add operations, columns and rows of the update matrices are distributed depending on whether their least significant bit (LSB) is 0 or 1. Indices with LSB = 0 go to the lower subcube and those with LSB = 1 go to the higher subcube. Similarly, in the next two parallel extend-add operations, columns and rows of the update matrices are exchanged among the processors depending on the second LSB of their indices.

Figure 6 illustrates all the parallel extend-add operations that take place during parallel multifrontal factorization of the matrix shown in Figure 4. The portion of an update matrix that is sent out by its original owner processor is shown in grey. Hence, if processors P_i and P_j with respective update matrices Q and R perform a parallel extend-add, then the final result at P_i will be the add-extension of the white portion of Q and the grey portion of R. Similarly, the final result at P_j will be the add-extension of the grey portion of Q and the white portion of R. Figure 7 further illustrates this processes by showing four consecutive extend-add operations on hypothetical update matrices to distribute the result among 16 processors.

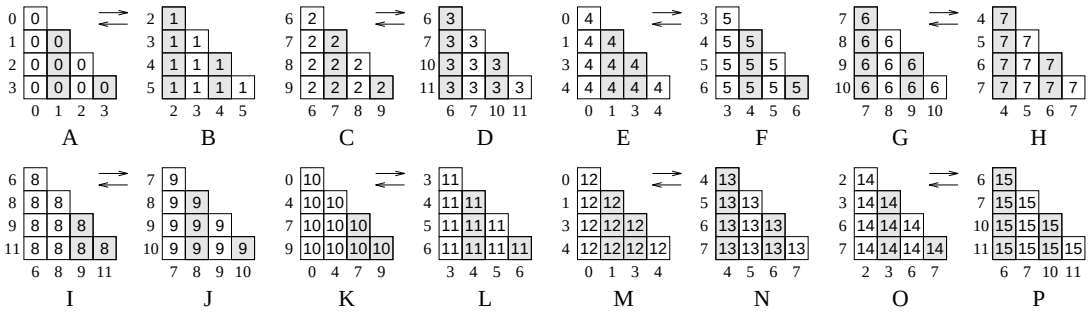
Between two successive parallel extend-add operations, several steps of dense Cholesky elimination may be performed. The number of such successive elimination steps is equal to the number of nodes in the relaxed supernode being processed. The communication that takes place in this phase is the standard communication in pipelined grid-based dense Cholesky factorization [39, 26]. If the average size of the frontal matrices is $t \times t$ during the processing of a relaxed supernode with m nodes on a q-processor subcube, then $O(m)$ messages of size $O(t/\sqrt{q})$ are passed through the grid in a pipelined fashion. Figure 8 shows the communication for one step of dense Cholesky factorization of a hypothetical frontal matrix for $q = 16$. It is shown in [27] that although this communication does not take place between the nearest neighbors on a subcube, the paths of all communications on any subcube are conflict free with e-cube routing [38, 26] and cut-through or worm-hole flow control. This is a direct consequence of the fact that a circular shift is conflict free on a hypercube with e-cube routing. Thus, a communication pipeline can be maintained among the processors of a subcube during the dense Cholesky factorization of frontal matrices.

3.1 Block-cyclic mapping of matrices onto processors

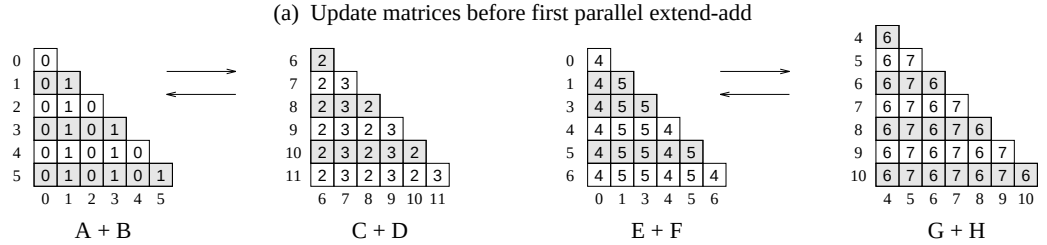
In the parallel multifrontal algorithm described in this section, the rows and columns of frontal and update matrices are distributed among the processors of a subcube in a cyclic manner. For example, the distribution of a matrix with indices from 0 to 11 on a 16-processor subcube is shown in Figure 7(e). The 16 processors form a logical mesh. The arrangement of the processors in the logical mesh is shown in Figure 9(a). In the distribution of Figure 7(e), consecutive rows and columns of the matrix are mapped onto neighboring processors of the logical mesh. If there are more rows and columns in the matrix than the number of processors in a row or column of the processor mesh, then the rows and columns of the matrix are wrapped around on the mesh.

Although the mapping shown in Figure 7(e) results in a very good load balance among the processors, it has a disadvantage. Notice that while performing the steps of Cholesky factorization on the matrix shown in Figure 7(e), the computation corresponding to consecutive pivots starts on different processors. For example, pivot 0 on processor 0, pivot 1 on processor 3, pivot 2 on processor 12, pivot 3 on processor 15, and so on. If the message startup time is high, this may lead to significant delays between the stages of the pipeline. Furthermore, on cache-based processors, the use of BLAS-3 for eliminating multiple columns simultaneously yields much higher performance than the use of BLAS-2 for eliminating one column at a time. Figure 9(b) shows a variation of the cyclic mapping, called block-cyclic mapping [26], that can alleviate these problems at the cost of some added load imbalance.

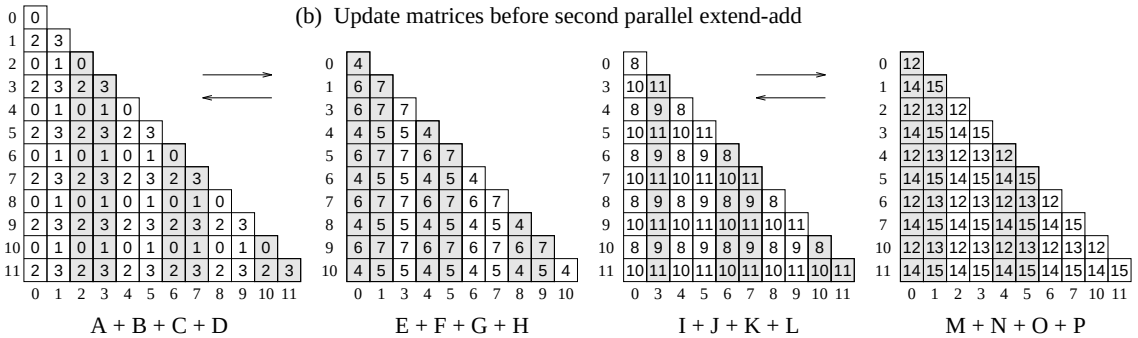
Recall that in the mapping of Figure 7(e), the least significant $\lceil \log p/2 \rceil$ bits of a row or column index of the matrix determine the processor to which that row or column belongs. Now if we disregard the least significant bit, and determine the distribution of rows and columns by the $\lceil \log p/2 \rceil$ bits starting with the second least significant bit, then the mapping of Figure 9(b) will result. In general, we can disregard the first k least significant bits, and arrive at a block-cyclic mapping with a block size of $2^k \times 2^k$. The optimal value of k depends on the ratio of computation time and the communication latency of the parallel computer in use and may vary from one computer to another for the same problem. In addtion, increasing the block size too much



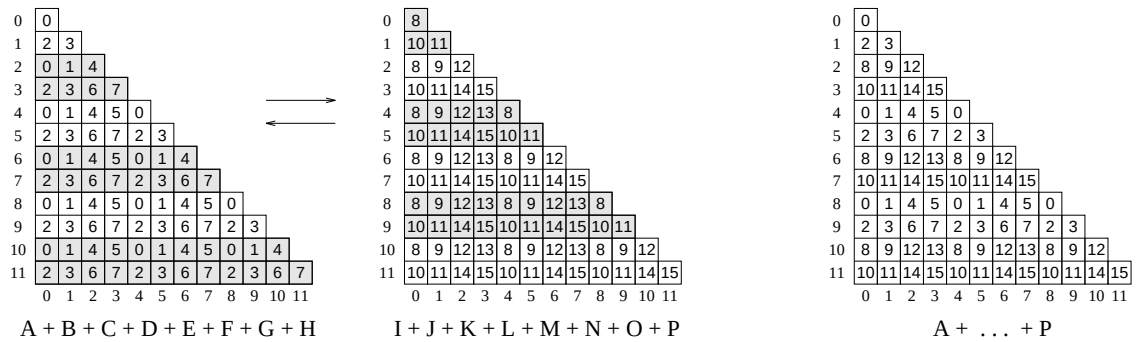
(a) Update matrices before first parallel extend-add



(b) Update matrices before second parallel extend-add



(c) Update matrices before third parallel extend-add



(d) Update matrices before fourth parallel extend-add

(e) Final distribution on 16 processors

Figure 7: Four successive parallel extend-add operations (denoted by "+") on hypothetical update matrices for multifrontal factorization on 16 processors, numbered from 0 to 15. The number inside a box denotes the number of the processor that owns the matrix element represented by the box.

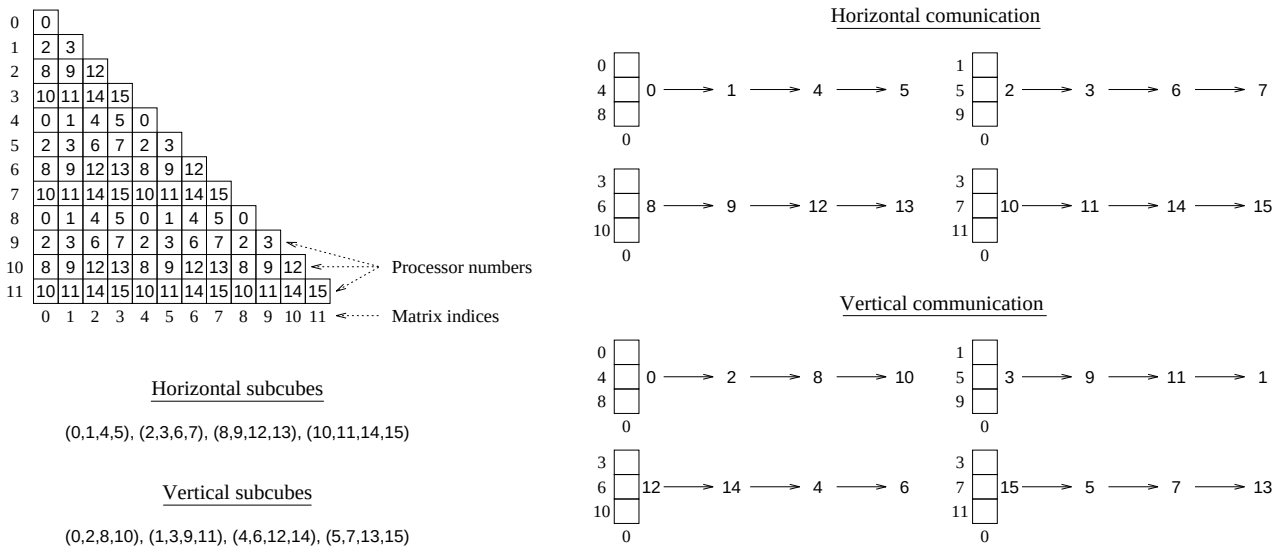
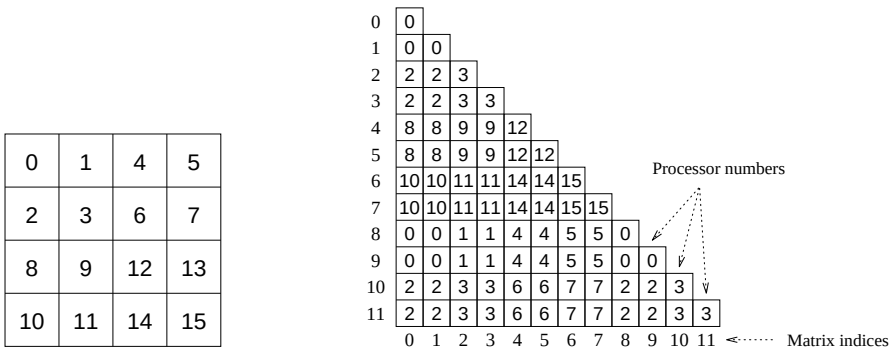


Figure 8: The two communication operations involved in a single elimination step (index of pivot = 0 here) of Cholesky factorization on a 12×12 frontal matrix distributed over 16 processors.



(a) A logical mesh of 16 processors

(b) Block-cyclic mapping with 2 x 2 blocks

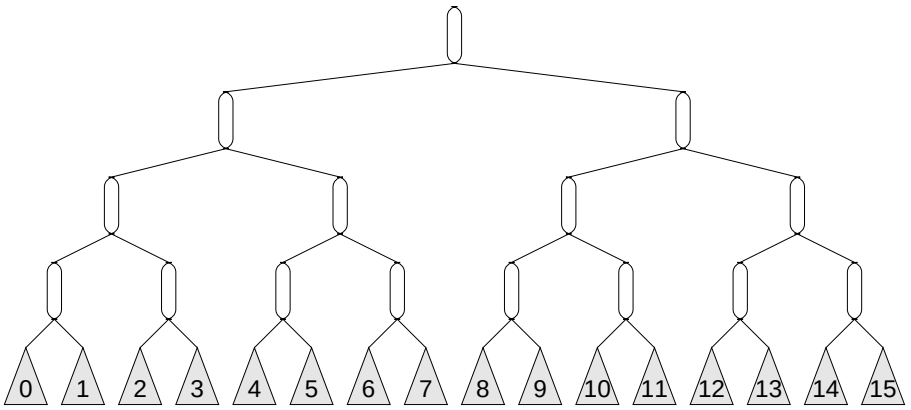
Figure 9: Block-cyclic mapping of a 12×12 matrix on a logical processor mesh of 16 processors.

may cause too much load imbalance during the dense Cholesky steps and may offset the advantage of using BLAS-3.

3.2 Subtree-to-submesh mapping for the 2-D mesh architecture

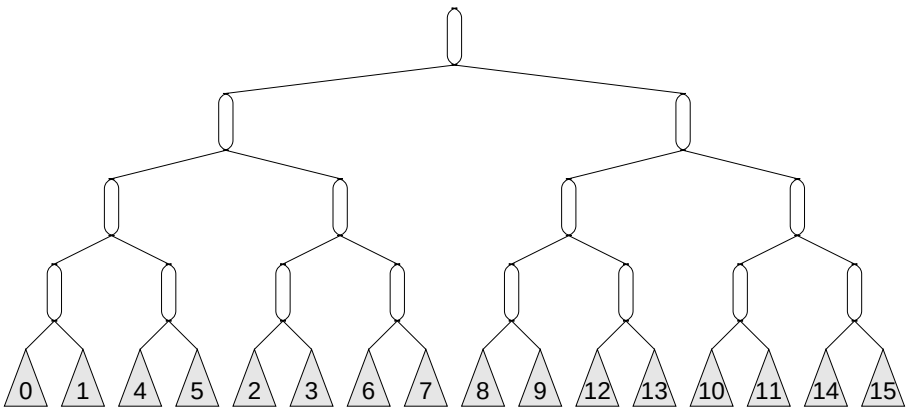
The mapping of rows and columns described so far works fine for the hypercube network. At each level, the update and frontal matrices are distributed on a logical mesh of processors (e.g., Figure 9(a)) such that each row and column of this mesh is a subcube of the hypercube. However, if the underlying architecture is a mesh, then a row or a column of the logical mesh may not correspond to a row or a column of the physical mesh. This will lead to contention for communication channels during the pipelined dense Cholesky steps of Figure 8 on a physical mesh. To avoid this contention for communication channels, we define a subtree-to-submesh mapping in this subsection. The subtree-to-subcube mapping described in Figure 4(b) ensures that any subtree of the relaxed supernodal elimination tree is mapped onto a subcube of the physical hypercube. This helps in localizing communication at each stage of factorization among groups of as few processors as possible. Similarly, the subtree-to-submesh mapping ensures that a subtree is mapped entirely within a submesh of the physical mesh.

Note that in subtree-to-subcube mapping for a 2^d-processor hypercube, all level-d subtrees of the relaxed



0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

(a) Subtree-subcube assignment of level-4 subtrees and the corresponding logical mesh for level-0 supernode



0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

(b) Subtree-submesh assignment of level-4 subtrees and the corresponding logical mesh for level-0 supernode

Figure 10: Labeling of subtrees in subtree-to-subcube (a) and subtree-to-submesh (b) mappings.

supernodal elimination tree are numbered in increasing order from left to right and a subtree labeled i is mapped onto processor i. For example, the subtree labeling of Figure 10(a) results in the update and frontal matrices for the supernodes in the topmost (level-0) relaxed supernode to be distributed among 16 processors as shown in Figure9(a). The subtree-to-submesh mapping starts with a different initial labeling of the level-d subtrees. Figure10(b) shows this labeling for 16 processors, which will result in the update and frontal matrices of the topmost relaxed supernode being partitioned on a 4×4 array of processors labeled in a row-major fashion.

We now define a function map such that replacing every reference to processor i in subtree-to-subcube mapping by a reference to processor $map(i, m, n)$ results in a subtree-to-submesh mapping on an $m \times n$ mesh. We assume that both m and n are powers of two. We also assume that either $m = n$ or $m = n/2$ (this configuration maximizes the cross-section width and minimizes the diameter of an mn-processor mesh). The function $map(i, m, n)$ is given by the following recurrence:

$$
\begin{aligned}
map(i,m,n) &= i, && \text{if } i < 2.\\
map(i,m,n) &= map(i, \tfrac{m}{2}, n), && \text{if } m = n,\ i < \tfrac{mn}{2}.\\
map(i,m,n) &= \tfrac{mn}{2} + map(i - \tfrac{mn}{2}, \tfrac{m}{2}, n), && \text{if } m = n,\ i \geq \tfrac{mn}{2}.\\
map(i,m,n) &= m\lfloor map(i,m,\tfrac{n}{2})/m \rfloor + map(i,m,\tfrac{n}{2}), && \text{if } m = \tfrac{n}{2},\ i < \tfrac{mn}{2}.\\
map(i,m,n) &= m(\lfloor map(i - \tfrac{mn}{2}, m, \tfrac{n}{2})/m \rfloor + 1) + map(i - \tfrac{mn}{2}, m, \tfrac{n}{2}), && \text{if } m = \tfrac{n}{2},\ i \geq \tfrac{mn}{2}.
\end{aligned}
$$

The above recurrence always maps a level-l relaxed supernode of a binary relaxed supernodal elimination tree onto an $(mn/2^l)$-processor submesh of the mn-processor two-dimensional mesh.

3.3 Applications to Gaussian elimination and QR factorization

Although we are focusing on Cholesky factorization in this paper, the serial algorithm of Figure 2 can be generalized to Gaussian elimination without pivoting for nearly structurally symmetric sparse matrices [9] and for solving sparse linear least squares problems [34].

Gaussian elimination without pivoting is numerically stable for diagonally dominant matrices; i.e., the matrices in which the sum of the absolute values of all the non-diagonal elements of a row or a column is less than the absolute value of the diagonal element of the same row or column. If the matrix A to be factored is not perfectly symmetric in structure, it can be treated as a symmetric structure matrix by explicitly storing a zero element $a_{j,i}$ if $a_{i,j}$ is nonzero. The multifrontal factorization can then be guided by an elimination tree constructed from this symmetric pattern matrix. For such matrices, the dense Cholesky factorization steps of F^k (lines 8–12) are replaced by steps of dense Gaussian elimination. The parallel algorithm also works similarly, except that the frontal and update matrices are now full square matrices rather than triangular matrices as in the case of Cholesky factorization.

The least square problem (LSP) $min_x||Ax - b||_2$ is commonly solved [19] by factoring the $m \times n$ matrix A ($m \geq n$) into the product QR, where Q is an $m \times n$ orthogonal matrix and R is an $n \times n$ upper triangular matrix. Matstoms [34, 35] has recently developed a multifrontal algorithm for QR factorization for sparse A. Matstoms' approach avoids storing Q explicitly and is based on the observation that the matrix R is a Cholesky factor of the $n \times n$ symmetric positive definite matrix A^TA. The LSP is solved from the semi-normal equation $R^TRx = A^Tb$ with a few steps of iterative refinement. The elimination tree and symbolic factorization of A^TA are used to guide the multifrontal QR factorization. The frontal matrices corresponding to the leaves of the elimination tree are derived from the original matrix as in the algorithm of Figure 2. The steps of dense QR factorization are then performed on these frontal matrices to obtain the corresponding update matrices. The process of forming the frontal matrix corresponding to a node higher up in the tree involves assembling the contributions from the update matrices of the node's children in the tree and nonzeros from the row and column corresponding to the node in the matrix A via extend-add operations. The difference between the multifrontal QR factorization and the algorithm of Figure 2 is that the frontal matrices can be square or rectangular and steps of dense QR factorization are performed in lines 8–12. It is easy to see that the parallel algorithm described in this section can be used by incorporating these changes.

4 Analysis of Communication Overhead

In this section, we derive expressions for the communication overhead of our algorithm for sparse matrices resulting from a finite difference operator on regular two- and three-dimensional grids. Within constant factors, these expressions can be generalized to all sparse matrices that are adjacency matrices of graphs whose n-node subgraphs have $O(\sqrt{n})$-node and $O(n^{2/3})$-node separators, respectively. This is because the properties of separators can be generalized from grids to all such graphs within the same order of magnitude bounds [30, 29, 15]. We derive these expressions for both hypercube and mesh architectures, and also extend the results to sparse matrices resulting from three-dimensional graphs whose n-node subgraphs have $O(n^{2/3})$-node separators.

The parallel multifrontal algorithm described in Section 3 incurs two types of communication overhead: one during parallel extend-add operations (Figure 7) and the other during the steps of dense Cholesky factorization while processing the supernodes (Figure 8). Crucial to estimating the communication overhead is estimating the sizes of frontal and update matrices at any level of the supernodal elimination tree.

Consider a $\sqrt{N} \times \sqrt{N}$ regular finite difference grid. We analyze the communication overhead for factorizing the $N \times N$ sparse matrix associated with this grid on p processors. In order to simplify the analysis, we assume a somewhat different form of nested-dissection than the one used in the actual implementation. This method of analyzing the communication complexity of sparse Cholesky factorization has been used in [17] in the context of a column-based subtree-to-subcube scheme. Within very small constant factors, the analysis holds for the standard nested dissection [13] of grid graphs. We consider a cross-shaped separator (described in [17]) consisting of $2\sqrt{N} - 1$ nodes that partitions the N-node square grid into four square subgrids of size $(\sqrt{N} - 1)/2 \times (\sqrt{N} - 1)/2$. We call this the level-0 separator that partitions the original grid (or the level-0 grid) into four level-1 grids. The nodes in the separator are numbered after the nodes in each subgrid have

been numbered. To number the nodes in the subgrids, they are further partitioned in the same way, and the process is applied recursively until all nodes of the original grid are numbered. The supernodal elimination tree corresponding to this ordering is such that each non-leaf supernode has four children. The topmost supernode has $2\sqrt{N}-1$ ($\approx 2\sqrt{N}$) nodes, and the size of the supernodes at each subsequent level of the tree is half of the supernode size at the previous level. Clearly, the number of supernodes increases by a factor of four at each level, starting with one at the top (level 0).

The nested dissection scheme described above has the following properties: (1) the size of level-l subgrids is approximately $\sqrt{N}/2^l \times \sqrt{N}/2^l$, (2) the number of nodes in a level-l separator is approximately $2\sqrt{N}/2^l$, and hence, the length of a supernode at level l of the supernodal elimination tree is approximately $2\sqrt{N}/2^l$. It has been proved in [17] that the number of nonzeros that an $i \times i$ subgrid can contribute to the nodes of its bordering separators is bounded by ki^2, where $k = 341/12$. Hence, a level-l subgrid can contribute at most $kN/4^l$ nonzeros to its bordering nodes. These nonzeros are in the form of the triangular update matrix that is passed along from the root of the subtree corresponding to the subgrid to its parent in the elimination tree. The dimensions of a matrix with a dense triangular part containing $kN/4^l$ entries is roughly $\sqrt{2kN}/2^l \times \sqrt{2kN}/2^l$. Thus, the size of an update matrix passed on to level $l-1$ of the supernodal elimination tree from level l is roughly upper-bounded by $\sqrt{2kN}/2^l \times \sqrt{2kN}/2^l$ for $l \geq 1$.

The size of a level-l supernode is $2\sqrt{N}/2^l$; hence, a total of $2\sqrt{N}/2^l$ elimination steps take place while the computation proceeds from the bottom of a level-l supernode to its top. A single elimination step on a frontal matrix of size $(t+1) \times (t+1)$ produces an update matrix of size $t \times t$. Since the size of an update matrix at the top of a level-l supernode is at most $\sqrt{2kN}/2^l \times \sqrt{2kN}/2^l$, the size of the frontal matrix at the bottom of the same supernode is upper-bounded by $(\sqrt{2k}+2)\sqrt{N}/2^l \times (\sqrt{2k}+2)\sqrt{N}/2^l$. Hence, the average size of a frontal matrix at level l of the supernodal elimination tree is upper-bounded by $(\sqrt{2k}+1)\sqrt{N}/2^l \times (\sqrt{2k}+1)\sqrt{N}/2^l$. Let $\sqrt{2k}-1 = \alpha$. Then $\alpha\sqrt{N}/2^l \times \alpha\sqrt{N}/2^l$ is an upper bound on the average size of a frontal matrix at level l.

We are now ready to derive expressions for the communication overhead due to the parallel extend-add operations and the elimination steps of dense Cholesky on the frontal matrices.

4.1 Overhead in parallel extend-add

Before the computation corresponding to level $l-1$ of the supernodal elimination tree starts, a parallel extend-add operation is performed on lower triangular portions of the update matrices of size $\sqrt{2kN}/2^l \times \sqrt{2kN}/2^l$, each of which is distributed on a $\sqrt{p}/2^l \times \sqrt{p}/2^l$ logical mesh of processors. Thus, each processor holds roughly $(kN/4^l) \div (p/4^l) = kN/p$ elements of an update matrix. Assuming that each processor exchanges roughly half of its data with the corresponding processor of another subcube, $t_s + t_w kN/(2p)$ time is spent in communication, where t_s is the message startup time and t_w is the per-word transfer time. Note that this time is independent of l. Since there are $(\log p)/2$ levels at which parallel extend-add operations take place, the total communication time for these operations is $O(N/p)\log p$ on a hypercube. The total communication overhead due to the parallel extend-add operations is $O(N \log p)$ on a hypercube.

4.2 Overhead in factorization steps

We have shown earlier that the average size of a frontal matrix at level l of the supernodal elimination tree is bounded by $\alpha\sqrt{N}/2^l \times \alpha\sqrt{N}/2^l$, where $\alpha = \sqrt{341/6} - 1$. This matrix is distributed on a $\sqrt{p}/2^l \times \sqrt{p}/2^l$ logical mesh of processors. As shown in Figure 8, there are two communication operations involved with each elimination step of dense Cholesky. The average size of a message is $(\alpha\sqrt{N}/2^l) \div (\sqrt{p}/2^l) = \alpha\sqrt{N/p}$. It can be shown [39, 26] that in a pipelined implementation on a $\sqrt{q} \times \sqrt{q}$ mesh of processors, the communication time for s elimination steps with an average message size of m is $O(ms)$. The reason is that although each message must go to $O(\sqrt{q})$ processors, messages corresponding to $O(\sqrt{q})$ elimination steps are active simultaneously in different parts of the mesh. Hence, each message effectively contributes only $O(m)$ to the total communication time. In our case, at level l of the supernodal elimination tree, the number of steps of dense Cholesky is $2\sqrt{N}/2^l$. Thus the total communication time at level l is $\alpha\sqrt{N/p} \times 2\sqrt{N}/2^l = O((N/\sqrt{p})(1/2^l))$. The total communication time for the elimination steps at the top $(\log p)/2$ levels of the supernodal elimination tree is $O((N/\sqrt{p})\Sigma_{l=0}^{\log_4 p - 1}(1/2^l))$. This has an upper bound of $O(N/\sqrt{p})$. Hence, the total communication overhead due to the elimination steps is $O(p \times N/\sqrt{p}) = O(N\sqrt{p})$.

The parallel multifrontal algorithm incurs an additional overhead of emptying the pipeline $\log p$ times (once before each parallel extend-add) and then refilling it. It can be easily shown that this overhead is $O(N)$ each time the pipeline restarts. Hence, the overall overhead due to restarting the pipeline $\log p$ time is $O(N \log p)$, which is smaller in magnitude than the $O(N\sqrt{p})$ communication overhead of the dense Cholesky elimination steps.

4.3 Communication overhead for 3-D problems

The analysis of the communication complexity for the sparse matrices arising out of three-dimensional finite element problems can be performed along the lines of the analysis for the case of two-dimensional grids. Consider an $N^{1/3} \times N^{1/3} \times N^{1/3}$ grid that is recursively partitioned into eight subgrids by a separator that consists of three orthogonal $N^{1/3} \times N^{1/3}$ planes. The number of nonzeros that an $i \times i \times i$ subgrid contributes to the nodes of its bordering separators is $O(i^4)$ [17]. As a result, an update or a frontal matrix at level l of the supernodal elimination tree will contain $O(N^{4/3}/2^{4l})$ entries distributed among $p/8^l$ processors. Thus, the communication time for the parallel extend-add operation at level l is $O(N^{4/3}/(2^l p))$. The total communication time for all parallel extend-add operations is $O((N^{4/3}/p)\Sigma_{l=0}^{\log_8 p-1}(1/2^l))$, which is $O(N^{4/3}/p)$. For the dense Cholesky elimination steps at any level, the message size is $O(N^{2/3}/\sqrt{p})$. Since there are $3N^{2/3}/4^l$ nodes in a level-l separator, the total communication time for the elimination steps is $O((N^{4/3}/\sqrt{p})\Sigma_{l=0}^{\log_8 p-1}(1/4^l))$, which is $O(N^{4/3}/\sqrt{p})$.

Hence, the total communication overhead due to parallel extend-add operations is $O(N^{4/3})$ and that due to the dense Cholesky elimination steps is $O(N^{4/3}\sqrt{p})$. As in the 2-D case, these asymptotic expressions can be generalized to sparse matrices resulting from three-dimensional graphs whose n-node subgraphs have $O(n^{2/3})$-node separators. This class includes the linear systems arising out of three-dimensional finite element problems.

4.4 Communication overhead on a mesh

The communication overhead due the dense Cholesky elimination steps is the same on both the mesh and the hypercube architectures because the frontal matrices are distributed on a logical mesh of processors. However, the parallel extend operations use the entire cross-section bandwidth of a hypercube, and the communication overhead due to them will increase on a mesh due to channel contention.

Recall from Section 4.1 that the communication time for parallel extend-add at any level is $O(N/p)$ on a hypercube. The extend-add is performed among groups of $p/4^l$ processors at level l of the supernodal elimination tree. Therefore, at level l, the communication time for parallel extend-add on a $\sqrt{p}/2^l \times \sqrt{p}/2^l$ submesh is $O(N/(2^l\sqrt{p}))$. The total communication time for all the levels is $O((N/\sqrt{p})\Sigma_{l=0}^{\log_4 p-1}(1/2^l))$. This has an upper bound of $O(N/\sqrt{p})$, and the upper bound on the corresponding communication overhead term is $O(N\sqrt{p})$. This is the same as the total communication overhead for the elimination steps. Hence, for two-dimensional problems, the overall asymptotic communication overhead is the same for both mesh and hypercube architectures.

The communication time on a hypercube for the parallel extend-add operation at level l is $O(N^{4/3}/(2^l p))$ for three-dimensional problems (Section 4.3). The corresponding communication time on a mesh would be $O(N^{4/3}/(4^l\sqrt{p})$. The total communication time for all the parallel extend-add operations is $O((N^{4/3}/\sqrt{p})\Sigma_{l=0}^{\log_8 p-1}(1/4^l))$, which is $O(N^{4/3}/\sqrt{p})$. As in the case of two-dimensional problems, this is asymptotically equal to the communication time for the elimination steps.

5 Scalability Analysis

The scalability of a parallel algorithm on a parallel architecture refers to the capacity of the algorithm-architecture combination to effectively utilize an increasing number of processors. In this section we use the isoefficiency metric [26, 28, 20] to characterize the scalability of our algorithm. The isoefficiency function of a combination of a parallel algorithm and a parallel architecture relates the problem size to the number of processors necessary to maintain a fixed efficiency or to deliver speedups increasing proportionally with increasing number of processors.

5.1 The isoefficiency metric of scalability analysis

Let W be the size of a problem in terms of the total number of basic operations required to solve a problem on a serial computer. For example, $W = O(N^2)$ for multiplying a dense $N \times N$ matrix with an N-vector. The serial run time of a problem of size W is given by $T_S = t_c W$, where t_c is the time to perform a single basic computation step. If T_P is the parallel run time of the same problem on p processors, then we define an overhead function T_o as $pT_P - T_S$. Both T_P and T_o are functions of W and p, and we often write them as $T_P(W, p)$ and $T_o(W, p)$, respectively. The efficiency of a parallel system with p processors is given by $E = T_S/(T_S + T_o(W, p))$. If a parallel system is used to solve a problem instance of a fixed size W, then the efficiency decreases as p increases. This is because the total overhead $T_o(W, p)$ increases with p. For many parallel systems, for a fixed p, if the problem size W is increased, then the efficiency increases because for a given p, $T_o(W, p)$ grows slower than $O(W)$. For these parallel systems, the efficiency can be maintained at a desired value (between 0 and 1) for increasing p, provided W is also increased. We call such systems **scalable** parallel systems. Note that for a given parallel algorithm, for different parallel architectures, W may have to increase at different rates with respect to p in order to maintain a fixed efficiency. As the number of processors are increased, the smaller the growth rate of problem size required to maintain a fixed efficiency, the more scalable the parallel system is.

Given that $E = 1/(1 + T_o(W, p)/(t_c W))$, in order to maintain a fixed efficiency, W should be proportional to $T_o(W, p)$. In other words, the following relation must be satisfied in order to maintain a fixed efficiency:

$$W = \frac{e}{t_c} T_o(W, p), \tag{1}$$

where $e = E/(1 - E)$ is a constant depending on the efficiency to be maintained. Equation (1) is the central relation that is used to determine the isoefficiency function of a parallel algorithm-architecture combination. This is accomplished by abstracting W as a function of p through algebraic manipulations on Equation (1). If the problem size needs to grow as fast as $f_E(p)$ to maintain an efficiency E, then $f_E(p)$ is defined as the isoefficiency function of the parallel algorithm-architecture combination for efficiency E.

5.2 Scalability of the parallel multifrontal algorithm

It is well known [15] that the total work involved in factoring the adjacency matrix of an N-node graph with an $O(\sqrt{N})$-node separator using nested dissection ordering of nodes is $O(N^{1.5})$. We have shown in Section 4 that the overall communication overhead of our scheme is $O(N\sqrt{p})$. From Equation 1, a fixed efficiency can be maintained if and only if $N^{1.5} \propto N\sqrt{p}$, or $\sqrt{N} \propto \sqrt{p}$, or $N^{1.5} = W \propto p^{1.5}$. In other words, the problem size must be increased as $O(p^{1.5})$ to maintain a constant efficiency as p is increased. In comparison, a lower bound on the isoefficiency function of Rothberg and Gupta's scheme [46] with a communication overhead of at least $O(N\sqrt{p}\log p)$ is $O(p^{1.5}(\log p)^3)$. The isoefficiency function of any column-based scheme is at least $O(p^3)$ because the total communication overhead has a lower bound of $O(Np)$. Thus, the scalability of our algorithm is superior to that of the other schemes.

It is easy to show that the scalability of our algorithm is $O(p^{1.5})$ even for the sparse matrices arising out of three-dimensional finite element grids. The problem size in the case of an $N \times N$ sparse matrix resulting from a three-dimensional grid is $O(N^2)$ [17]. We have shown in Section 4.3 that the overall communication overhead in this case is $O(N^{4/3}\sqrt{p})$. To maintain a fixed efficiency, $N^2 \propto N^{4/3}\sqrt{p}$, or $N^{2/3} \propto \sqrt{p}$, or $N^2 = W \propto p^{1/5}$.

A lower bound on the isoefficiency function for dense matrix factorization is $O(p^{1.5})$ [26, 27]. The factorization of a sparse matrix involves factoring a dense matrix and the amount of computation required to factor this dense matrix is of the same order as the amount of computation required to factor the entire sparse matrix [15, 17]. Hence, $O(p^{1.5})$ is the lower bound on the isoefficiency function of sparse matrix factorization, and our algorithm achieves this lower bound.

6 Experimental Results

We implemented the parallel multifrontal algorithm described in this paper on the nCUBE2 parallel computer. We have gathered some preliminary speedup results for two classes of sparse matrices, which are summarized in Tables 1 and 2. The entire code is written in the C programming language augmented with the message passing

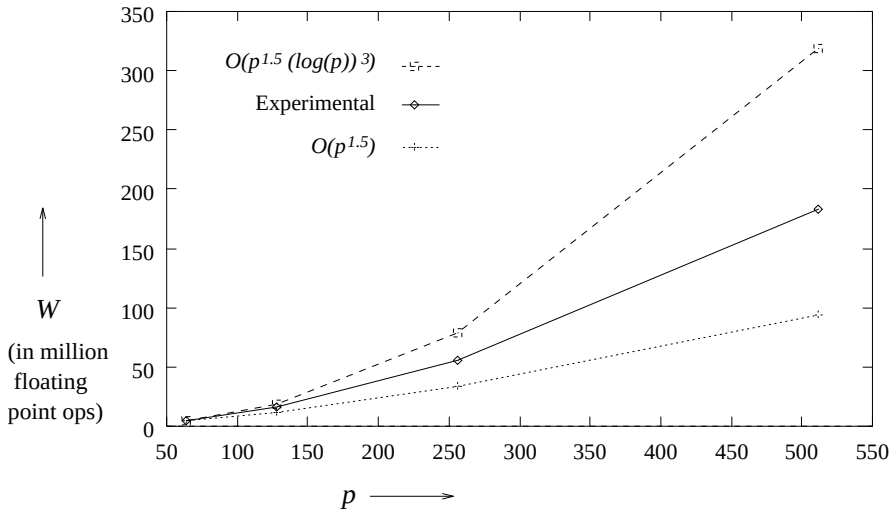


Figure 11: Comparison of our experimental isoefficiency curves with $O(p^{1.5})$ curve (theoretical asymptotic isoefficiency function of our algorithm due to communication overhead on a hypercube) and with $O(p^{1.5}(\log p)^3)$ curve (the lower bound on the isoefficiency function of the best known parallel sparse factorization algorithm until now). The four data points on the curves correspond to the matrices GRID63x63, GRID103x95, GRID175x127, and GRID223x207 (see Table 1).

primitives for nCUBE and compiled using the *ncc* compiler with the *-O* optimization option. The results for 64 processors or less were obtained on the 64-processor nCUBE2 (with 16 MB memory per node) at the CIS department of University of Florida at Gainsville. The results for 128 to 1024 processors were obtained on the 1024-processor nCUBE2 (with 4 MB memory per node) at Sandia National Labs. All timings are for numerical factorization with single precision arithmetic. From an independent set of experiments, we determined the time for a floating point operation to be roughly 0.5μs, the message startup time to be roughly 180μs, and the communication rate with no channel conflicts to be roughly 2 bytes per μs.

We conducted one set of experiments on sparse matrices associated with a 9-point difference operator on rectangular grids. The purpose of these experiments was to compare their results with the scalability analysis in Section 5. The dimensions of the grids were chosen such that the elimination trees were as balanced as possible. The standard nested dissection ordering [13] was used for these matrices. Nested dissection has been shown to have optimal fill-in in the case of regular grids [15]. The results of our implementation for some of these grids are summarized in Table 1. Matrix GRIDixj in the table refers to the sparse matrix obtained from an $i \times j$ 9-point finite difference grid.

From our experiments on the 2-D grids, we selected a few points of equal efficiency and plotted the W versus p curve, which is shown by the solid line in Figure 11 for $E \approx 0.31$. The problem size W is measured in terms of the total number of floating point arithmetic operations performed during factorization; for example, for GRID63x63, $W = 4.1684 \times 10^6$. The dotted curve below the experimental curve corresponds to an isoefficiency function of $O(p^{1.5})$ and the dashed curve at the top corresponds to an isoefficiency function of $O(p^{1.5}(\log p)^3)$. The first point on each curve is that for the matrix GRID63x63 on 64 processors. To obtain the dotted curve, W was increased by a factor of $2\sqrt{2}$ every time the number of processors p was doubled. To obtain the dashed curve, a constant was defined to be $4.1684 \times 10^6/(64 \times 8 \times 6^3) \approx 37.7$. Then, for any $p > 64$, $W = 37.7p(\log p)^3$. Given that for $p = 64$, an efficiency of roughly 31% is obtained for problem size 4.1684×10^6, the dotted and dashed curves indicate the problem sizes that will yield the same efficiency for $p = 128$, 256, and 512 if the isoefficiency function is $O(p^{1.5})$ and $O(p^{1.5}(\log p)^3)$, respectively.

Figure 11 shows that the experimental isoefficiency curve is considerable better than $O(p^{1.5}(\log p)^3)$, which is a lower bound on the isoefficiency function of the previously best known (in terms of total communication volume) parallel algorithm [46] for sparse matrix factorization. However, it is worse than $O(p^{1.5})$, which is the asymptotic isoefficiency function derived in Section 5. There are two main reasons for this. First, the $O(p^{1.5})$ isoefficiency function does not take load imbalance into account. It has been shown in [33] that even

Matrix: GRID63x63; N = 3969; NNZ = 99.45 thousand; FLOP = 4.1684 million								
p	1	2	4	8	16	32	64	128
Time	8.27	4.22	2.22	1.41	.904	.594	.411	.315
Speedup	1.00	1.96	3.73	5.87	9.15	13.9	20.1	26.2
Efficiency	100.0%	98.0%	93.1%	73.3%	57.2%	43.5%	31.4%	20.5%
Matrix: GRID103x95; N = 9785; NNZ = 288.04 thousand; FLOP = 16.599 million								
p	1	4	8	16	32	64	128	256
Time	28.92	7.500	4.493	2.699	1.596	1.017	.7301	.5516
Speedup	1.00	3.86	6.44	10.7	18.1	28.4	39.6	52.4
Efficiency	100.0%	96.4%	80.5%	66.9%	56.6%	44.4%	31.0%	20.5%
Matrix: GRID127x127; N = 16129; NNZ = 518.58 thousand; FLOP = 36.682 million								
p	1	4	8	16	32	64	128	256
Time	58.86	15.07	8.878	5.155	2.971	1.783	1.171	.8546
Speedup	1.00	3.90	6.63	11.4	19.8	33.0	50.3	68.9
Efficiency	100.0%	97.6%	82.8%	71.0%	61.9%	51.6%	39.3%	26.9%
Matrix: GRID175x127; N = 22225; NNZ = 731.86 thousand; FLOP = 56.125 million								
p	1	8	16	32	64	128	256	512
Time	87.38	12.74	7.420	4.164	2.468	1.570	1.092	.8529
Speedup	1.00	6.86	11.8	21.0	35.4	55.7	80.0	102.4
Efficiency	100.0%	85.7%	73.6%	65.6%	55.3%	43.5%	31.3%	20.0%
Matrix: GRID255x127; N = 32385; NNZ = 1140.6 thousand; FLOP = 100.55 million								
p	1	16	32	64	128	256	512	1024
Time	149.15	12.22	6.651	3.861	2.349	1.557	1.091	.8357
Speedup	1.00	12.2	22.4	38.6	63.5	95.8	136.7	178.5
Efficiency	100.0%	76.3%	70.1%	60.4%	49.6%	37.4%	26.7%	17.4%
Matrix: GRID223x207; N = 46161; NNZ = 1498.7 thousand; FLOP = 179.87 million								
p	1	16	32	64	128	256	512	1024
Time	254.70	20.21	10.85	6.161	3.751	2.380	1.634	1.237
Speedup	1.00	12.6	23.5	41.3	67.9	107.0	155.9	205.9
Efficiency	100.0%	78.8%	73.4%	64.6%	53.0%	41.8%	30.5%	20.1%

Table 1: Experimental results for factoring sparse symmetric positive definite matrices associated with a 9-point difference operator on rectangular grids. All times are in seconds.

if a grid graph is perfectly partitioned in terms of the number of nodes, the work load associated with each partition varies. The partitions closer to the center of the grid require more computation than the ones on or closer to the periphery. Another reason that the experimental isoefficiency function appears worse than the prediction is that the efficiencies of the parallel implementation are computed with respect to a very efficient serial implementation. This can be judged from the run times on a single nCUBE2 processor whose peak performance is rated at roughly 2.5 MFLOPS (double precision) and 8 MIPS. In our implementation, the computation associated with the subtrees below level $\log p$ in the relaxed supernodal elimination tree is handled by the serial code. However, the computation above this level is handled by a separate code. In our preliminary implementation, this part of the code is less efficient than the serial code (disregarding communication) due to additional bookkeeping, which has a potential for optimization. For example, the total time spent by all the processors participating in a parallel extend-add operation besides message passing is more than the time taken to perform extend-add on the same update matrices on a single processor. The same is true for the dense factorization steps too. However, despite these inefficiencies, our implementation is more scalable than a hypothetical ideal implementation (with perfect load balance) of the previously best known parallel algorithm for sparse Cholesky factorization.

In Table 2 we summarize the results of factoring some matrices from the Harwell-Boeing collection of sparse matrices [7]. The purpose of these experiments was to to demonstrate that our algorithm can deliver good speedups on hundreds of processors for practical problems. Spectral nested dissection (SND) [40, 41, 42] was used to order the matrices in Table 2. This choice of the ordering scheme was prompted by two factors. First, there is increasing evidence that spectral orderings offer a good balance between generality of application and the quality of ordering—both in terms of load balance and fill reduction [6]. Second, the SND algorithm itself can be parallelized efficiently, whereas most other ordering schemes do not appear to be as well-suited for parallelization. Although, at the present time we compute the ordering on a serial computer, SND is our ordering algorithm of choice in a prospective completely parallel implementation of a sparse linear system solver based on our parallel multifrontal algorithm. A drawback of using a serial implementation of SND is that its run time is too high. However, variations of SND such as multilevel SND [5, 42] run much faster without compromising on the quality of ordering.

From the experimental results in Tables 1 and 2, we can infer that our algorithm can deliver substantial speedups, even on moderate problem sizes. These speedups are computed with respect to a very efficient serial implementation of the multifrontal algorithm. To lend credibility to our speedup figures, we compared the run times of our program on a single processor with the single processor run times given for iPSC/2 in [43] and [49]. The nCUBE2 processors are about 2 to 3 times faster than iPSC/2 processors and our serial implementation, with respect to which the speedups are computed, is 4 to 5 times faster than the one in [43] and [49]. Our single processor run times are four times less than the single processor run times on iPSC/2 reported in [3]. We also found that for some matrices (e.g., that from a 127×127 9-point finite difference grid), our implementation on eight nCUBE2 processors (8.9 seconds) is faster than the 16-processor iPSC/860 implementation (9.7 seconds) reported in [51], although iPSC/860 has much higher computation speeds.

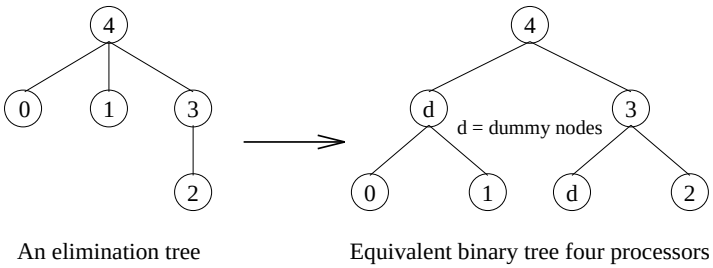
6.1 Load balancing for factorization

The factorization algorithm as described in this paper requires a binary relaxed supernodal elimination trees that are fairly balanced. After obtaining the ordered matrix and the corresponding elimination tree, we run the elimination tree through a very fast tree balancing heuristic. This heuristic performs a single pass of depth-first search on the elimination tree and accomplishes two tasks. First, it converts a general tree into a binary tree required by the subtree to subcube mapping. Second, it performs limited reordering of the subtrees within the elimination tree to ensure that the load imbalance at a given level does not exceed a predefined tolerance. The functionality of this algorithm is briefly described in Figure 12. If elimination of a column is regarded as a basic subtask in Cholesky factorization, then the elimination tree gives a partial ordering of these subtask for correct factorization [31]. Our tree balancing algorithm is based on the fact that a modified elimination tree that does not violate the partial order specified by the original tree still leads to correct factorization. This algorithm generally improves the load-balance significantly in parallel factorization at the cost of very small increase in fill.

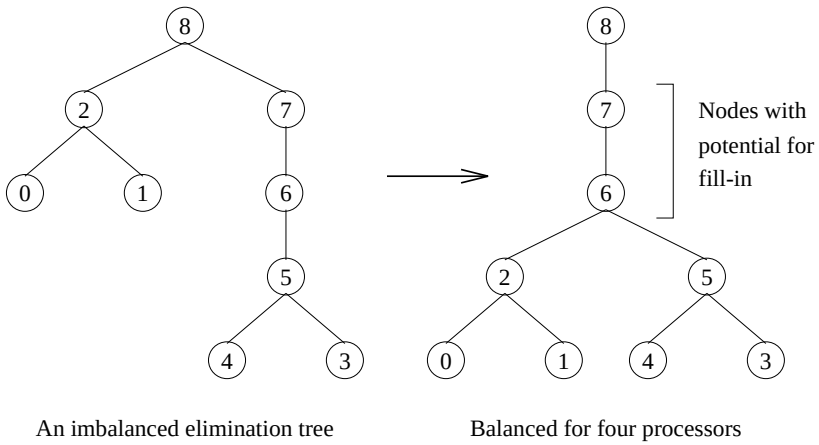
Table 2 also gives the percentage load balance for different values of p for BCSSTK15, BCSSTK25, and

Matrix: BCSSTK15; N = 3948; NNZ = 488.8 thousand; FLOP = 85.55 million											
p	1	2	4	8	16	32	64	128	256	512	1024
Time	103.73	52.63	26.66	14.88	8.29	4.98	3.20	2.156	1.530		
Speedup	1.00	1.97	3.89	6.97	12.5	20.8	32.4	48.1	67.8		
Efficiency	100.0%	98.5%	97.3%	87.1%	78.2%	65.1%	50.7%	37.6%	26.5%		
Load balance	100%	99%	98%	91%	91%	87%	87%	84%	84%		
Matrix: BCSSTK25; N = 15439; NNZ = 1940.3 thousand; FLOP = 512.88 million											
p	1	2	4	8	16	32	64	128	256	512	1024
Time	588.5	301.23	184.84	74.71	52.29	30.01	16.66	10.38	6.64	4.53	
Speedup	1.00	1.95	3.18	6.21	11.3	19.6	35.3	56.7	88.6	129.9	
Efficiency	100.0%	97.7%	79.6%	77.7%	70.3%	61.3%	57.0%	44.3%	34.6%	25.4%	
Load balance	100%	98%	80%	78%	71%	63%	62%	62%	62%	61%	
Matrix: BCSSTK29; N = 13992; NNZ = 2174.46 thousand; FLOP = 609.08 million											
p	1	2	4	8	16	32	64	128	256	512	1024
Time	704.0	359.7	212.9	110.45	55.06	31.36	19.22	12.17	7.667	4.631	3.119
Speedup	1.00	1.96	3.31	6.37	12.8	22.5	36.6	57.9	91.8	152.6	225.6
Efficiency	100.0%	97.9%	82.7%	79.7%	79.9%	70.2%	57.2%	45.2%	35.9%	29.8%	22.0%
Load balance	100%	98%	83%	82%	84%	82%	77%	72%	68%	72%	72%
Matrix: BCSSTK30; N = 28924; NNZ = 5893.59 thousand; FLOP = 2246.0 million											
p	1	2	4	8	16	32	64	128	256	512	1024
Time	2599.0*	1493.5	1050.8	537.4	256.6	134.5	79.93	43.67	24.73	14.616	11.078
Speedup	1.00	1.74	2.47	4.84	10.1	19.3	32.5	59.5	105.0	177.8	234.6
Efficiency	100.0%	87.0%	61.8%	60.5%	63.3%	60.4%	50.8%	46.5%	41.0%	34.7%	22.9%
Matrix: BCSSTK31; N = 35588; NNZ = 6458.34 thousand; FLOP = 2583.6 million											
p	1	2	4	8	16	32	64	128	256	512	1024
Time	3358.0*	1690.7	924.6	503.0	262.0	134.3	73.57	42.02	24.58	14.627	9.226
Speedup	1.00	1.99	3.63	6.68	12.8	25.0	45.6	79.9	136.6	229.6	364.2
Efficiency	100.0%	99.3%	90.8%	83.4%	80.1%	78.1%	71.3%	62.4%	53.4%	44.8%	35.6%
Matrix: BCSSTK32; N = 44609; NNZ = 8943.91 thousand; FLOP = 4209.0 million											
p	1	2	4	8	16	32	64	128	256	512	1024
Time	5215.0#					276.17	153.23		46.24	27.25	16.40
Speedup	1.00					18.9	34.0		112.8	191.4	318.0
Efficiency	100.0%					59.0%	53.2%		44.1%	37.4%	31.1%

Table 2: Experimental results for factoring some sparse symmetric positive definite matrices resulting from 3-D problems in structural engineering. All times are in seconds. The single processor run times suffixed by "*" were estimated by timing different parts of factorization on two processors. The factorization corresponding to the two subtrees of the topmost relaxed supernode is done completely independently on two processors. For example, in the case of BCSSTK31, the two processors spent 1669.4 and 1634.9 seconds respectively on the two subtrees of the topmost relaxed supernode. In terms of total number of operations, this was 98.4% of the total work. Hence, the single processor run time was estimated as $(1669.4 + 1634.9)/.984 = 3358.0$. Similarly, the time suffixed by a "#" was estimated using 32 processors.



(a) Converting an elimination tree to a binary tree



(b) Balancing an elimination tree

Figure 12: The two functions performed by the tree balancing algorithm.

BCSSTK29 matrices. To evaluate the load balance, we wrote a sequential program that simulates the parallel multifrontal algorithm and reports the maximum achievable efficiency for a given number of processors in the absence of communication overhead. The simulator takes as input the same binary relaxed supernodal elimination tree as the the parallel algorithm. Now the parallel run time to factorize the part of the matrix corresponding to a subtree rooted at level l of the binary relaxed supernodal elimination tree is given by the sum of the parallel run time to process the root of the subtree and the parallel run time to eliminate the subtrees of the root. The parallel run time to processes the root is computed by dividing the serial run time to processes the root by $p/2^m$, where $m = l$ if $0 \leq l < \log p$ and $m = \log p$ if $l \geq \log p$. The parallel run time to eliminate the subtrees is computed as the maximum of the parallel run times for the two subtrees if $0 \leq l < \log p$ and as the sum of the run times of the subtrees if $l \geq \log p$. Thus, by timing the work associated with individual subtrees in the recursive serial implementation of the multifrontal algorithm, the simulator can easily estimate the parallel run time in the absence of communication overhead for a given p. The load balance or the maximum achievable efficiency is then estimated by dividing the serial run time with the product of p and the estimated parallel run time. Note that the inefficiency due to load imbalance of subtree-to-subcube mapping does not continue to increase with the number of processors, but tends to saturate at 64–128 processors.

It is evident from the load balance values given in Table 2 that a combination of spectral nested dissection with our tree balancing algorithm results in very respectable load balances for up to 1024 processors. The number of nonzeros in the triangular factor (NNZ) and the number of floating point operations (FLOP) reported in Table 2 are for the single processor case. As the number of processors is increased, the tree balancing algorithm is applied to more levels ($\log p$) of the relaxed supernodal elimination tree, and consequently, the total NNZ and FLOP increase. Thus, an efficiency of $x\%$ in Table 2 indicates that there is a $(100 - x)\%$ loss, which includes three factors: communication, load imbalance, and extra work. For example, the efficiency for BCSSTK25 on

64 processors is 57%; i.e., there is a 43% overhead. However, only 5% overhead is due to communication and extra work. The remaining 38% overhead is due to load imbalance.

Although we have observed through our experiments that the upper bound on efficiency due to load imbalance does not fall below 60–70% for hundreds of processors, even this bound can be improved further. In [21], Karypis and Kumar relax the subtree-to-subcube mapping to a subforest-to-subcube mapping, which significantly reduces load imbalances at the cost of a little increase in communication.

7 Concluding Remarks and Future Work

In this paper, we analytically and experimentally demonstrate that scalable parallel implementations of direct methods for solving large sparse systems are possible. We describe a parallel algorithm whose implementation on a 1024-processor nCUBE2 delivers an order of magnitude higher speedups than any previously known implementation. We use the isoefficiency metric [26, 28, 20] to characterize the scalability of our algorithm. We show that the isoefficiency function of our algorithm is $O(p^{1.5})$ on hypercube and mesh architectures for sparse matrices arising out of both two- and three-dimensional problems. We also show that $O(p^{1.5})$ is asymptotically the best possible isoefficiency function for a parallel implementation of any direct method for solving a system of linear equations, either sparse or dense. In [47], Schreiber concludes that it is not yet clear whether sparse direct solvers can be made competitive at all for highly ($p \geq 256$) and massively ($p \geq 4096$) parallel computers. We hope that, through this paper, we have given an affirmative answer to at least a part of the query.

Our algorithm is a significant improvement over other parallel algorithms for sparse matrix factorization. This improvement has been realized by reducing the order of the communication overhead. An efficient parallel sparse matrix factorization algorithm requires the following: (1) the matrix must be partitioned among the processors along both rows and columns because pure column-based partitioning schemes have some fundamental limitations [47, 46], (2) the distribution of rows and columns of the matrix among the processors must follow some sort of cyclic order to ensure proper load balance, and (3) communication must be localized among as few processors as possible at every stage of elimination by following a subtree-to-subcube type work distribution strategy [16]. Ours is the only implementation we know of that satisfies all of the above conditions.

Through a preliminary implementation on nCUBE2, we have demonstrated the feasibility of using highly parallel computers for numerical factorization of sparse matrices. In [25], we have applied our algorithm to obtain a parallel formulation of interior point algorithms and have observed significant speedups in solving linear programming problems. Currently, an implementation of our parallel multifrontal algorithm is being developed for Cray T3D and IBM SP-2 parallel computers. A preliminary implementation of a variation of this scheme with improved load balancing currently yields up to 20 GFLOPS on medium-size problems on a 1024-processor Cray T3D [21]. We are also working on parallel formulations of spectral nested dissection ordering, symbolic factorization, and a triangular system solver. We plan to implement these algorithms on the IBM SP-2 and Cray T3D parallel computers to obtain high overall computation speeds for the complete solution of sparse symmetric positive definite linear system of equations.

Acknowledgments

We wish to thank Dr. Alex Pothen for his guidance with spectral nested dissection ordering, which was used for the BCSSTK matrices. George Karypis implemented the spectral nested dissection algorithm and gave several useful comments and suggestions in the development of the parallel multifrontal algorithm. We are also grateful to Prof. Sartaj Sahni and the systems staff at the CIS Department, University of Florida for providing access to a 64-processor nCUBE2. Sandia National Labs provided access to their 1024-processor nCUBE2.

References

[1] Cleve Ashcraft. The domain/segment partition for the factorization of sparse symmetric positive definite matrices. Technical Report ECA-TR-148, Boeing Computer Services, Seattle, WA, 1990.

[2] Cleve Ashcraft. The fan-both family of column-based distributed cholesky factorization algorithms. In A. George, John R. Gilbert, and J. W.-H. Liu, editors, *Graph Theory and Sparse Matrix Computations*. Springer-Verlag, New York, NY, 1993.

[3] Cleve Ashcraft, S. C. Eisenstat, and J. W.-H. Liu. A fan-in algorithm for distributed sparse numerical factorization. *SIAM Journal on Scientific and Statistical Computing*, 11:593–599, 1990.

[4] Cleve Ashcraft, S. C. Eisenstat, J. W.-H. Liu, and A. H. Sherman. A comparison of three column based distributed sparse factorization schemes. Technical Report YALEU/DCS/RR-810, Yale University, New Haven, CT, 1990. Also appears in *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, 1991.

[5] Stephen T. Barnard and Horst D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. Technical Report RNR-92-033, NASA Ames Research Center, Moffett Field, CA, 1992.

[6] W. J. Camp, S. J. Plimpton, B. A. Hendrickson, and R. W. Leland. Massively parallel methods for engineering and science problems. *Communications of the ACM*, 37(4):31–41, April 1994.

[7] Iain S. Duff, R. G. Grimes, and J. G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (release I). Technical Report TR/PA/92/86, Research and Technology Division, Boeing Computer Services, Seattle, WA, 1992.

[8] Iain S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[9] Iain S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear equations. *SIAM Journal on Scientific and Statistical Computing*, 5(3):633–641, 1984.

[10] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1):54–135, March 1990. Also appears in K. A. Gallivan et al. *Parallel Algorithms for Matrix Computations*. SIAM, Philadelphia, PA, 1990.

[11] G. A. Geist and E. G.-Y. Ng. Task scheduling for parallel sparse Cholesky factorization. *International Journal of Parallel Programming*, 18(4):291–314, 1989.

[12] G. A. Geist and C. H. Romine. LU factorization algorithms on distributed-memory multiprocessor architectures. *SIAM Journal on Scientific and Statistical Computing*, 9(4):639–649, 1988. Also available as Technical Report ORNL/TM-10383, Oak Ridge National Laboratory, Oak Ridge, TN, 1987.

[13] A. George. Nested dissection of a regular finite-element mesh. *SIAM Journal on Numerical Ananlysis*, 10:345–363, 1973.

[14] A. George, M. T. Heath, J. W.-H. Liu, and E. G.-Y. Ng. Sparse Cholesky factorization on a local memory multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9:327–340, 1988.

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

[16] A. George, J. W.-H. Liu, and E. G.-Y. Ng. Communication reduction in parallel sparse Cholesky factorization on a hypercube. In M. T. Heath, editor, *Hypercube Multiprocessors 1987*, pages 576–586. SIAM, Philadelphia, PA, 1987.

[17] A. George, J. W.-H. Liu, and E. G.-Y. Ng. Communication results for parallel sparse Cholesky factorization on a hypercube. *Parallel Computing*, 10(3):287–298, May 1989.

[18] John R. Gilbert and Robert Schreiber. Highly parallel sparse Cholesky factorization. *SIAM Journal on Scientific and Statistical Computing*, 13:1151–1172, 1992.

[19] Gene H. Golub and Charles Van Loan. *Matrix Computations: Second Edition*. The Johns Hopkins University Press, Baltimore, MD, 1989.

[20] Ananth Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distributed Technology*, 1(3):12–21, August, 1993. Also available as Technical Report TR 93-24, Department of Computer Science, University of Minnesota, Minneapolis, MN.

[21] Anshul Gupta, George Karypis, and Vipin Kumar. Highly scalable parallel algorithms for sparse matrix factorization. Technical Report 94-63, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994. To appear in *IEEE Transactions on Parallel and Distributed Systems*, 1997. Postscript file available via anonymous FTP from the site *ftp://ftp.cs.umn.edu/users/kumar*.

[22] Anshul Gupta and Vipin Kumar. Performance properties of large scale parallel systems. *Journal of Parallel and Distributed Computing*, 19:234–244, 1993. Also available as Technical Report TR 92-32, Department of Computer Science, University of Minnesota, Minneapolis, MN.

[23] M. T. Heath, E. G.-Y. Ng, and Barry W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991. Also appears in K. A. Gallivan et al. *Parallel Algorithms for Matrix Computations*. SIAM, Philadelphia, PA, 1990.

[24] Laurie Hulbert and Earl Zmijewski. Limiting communication in parallel sparse Cholesky factorization. *SIAM Journal on Scientific and Statistical Computing*, 12(5):1184–1197, September 1991.

[25] George Karypis, Anshul Gupta, and Vipin Kumar. Parallel formulation of interior point algorithms. Technical Report 94-20, Department of Computer Science, University of Minnesota, Minneapolis, MN, April 1994. A short version appears in *Supercomputing '94 Proceedings*.

[26] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings, Redwood City, CA, 1994.

[27] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Solutions Manual for Introduction to Parallel Computing*. Benjamin/Cummings, Redwood City, CA, 1994.

[28] Vipin Kumar and Anshul Gupta. Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 22(3):379–391, 1994. Also available as Technical Report TR 91-18, Department of Computer Science Department, University of Minnesota, Minneapolis, MN.

[29] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16:346–358, 1979.

[30] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36:177–189, 1979.

[31] J. W.-H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. Technical Report CS-90-04, York University, Ontario, Canada, 1990. Also appears in *SIAM Review*, 34:82–109, 1992.

[32] Robert F. Lucas. *Solving planar systems of equations on distributed-memory multiprocessors.* PhD thesis, Department of Electrical Engineering, Stanford University, Palo Alto, CA, 1987.

[33] Robert F. Lucas, Tom Blank, and Jerome J. Tiemann. A parallel solution method for large sparse systems of equations. *IEEE Transactions on Computer Aided Design*, CAD-6(6):981–991, November 1987.

[34] Pontus Matstoms. *The multifrontal solution of sparse linear least squares problems.* PhD thesis, Department of Mathematics, Linkoping University, S-581 83 Linkoping, Sweden, March 1992.

[35] Pontus Matstoms. Sparse QR factorization in MATLAB. Technical Report LiTH-MAT-R-1992-05, Department of Mathematics, Linkoping University, S-581 83 Linkoping, Sweden, March 1993.

[36] Mo Mu and John R. Rice. A grid-based subtree-subcube assignment strategy for solving partial differential equations on hypercubes. *SIAM Journal on Scientific and Statistical Computing*, 13(3):826–839, May 1992.

[37] Vijay K. Naik and M. Patrick. Data traffic reduction schemes Cholesky factorization on aynchronous multiprocessor systems. In *Supercomputing '89 Proceedings*, 1989. Also available as Technical Report RC 14500, IBM T. J. Watson Research Center, Yorktown Heights, NY.

[38] S. F. Nugent. The iPSC/2 direct-connect communications technology. In *Proceedings of the Third Conference on Hypercubes, Concurrent Computers, and Applications*, pages 51–60, 1988.

[39] Dianne P. O'Leary and G. W. Stewart. Assignment and scheduling in parallel matrix factorization. *Linear Algebra and its Applications*, 77:275–299, 1986.

[40] Alex Pothen, H. D. Simon, and K.-P. Liou. Partioning sparce matrices with eigenvectors of graphs. *SIAM Journal of Mathematical Analysis and Applications*, 11(3):430–452, 1990.

[41] Alex Pothen, H. D. Simon, and Lie Wang. Spectral nested dissection. Technical Report 92-01, Computer Science Department, Pennsylvania State University, University Park, PA, 1992.

[42] Alex Pothen, H. D. Simon, Lie Wang, and Stephen T. Bernard. Towards a fast implementation of spectral nested dissection. In *Supercomputing '92 Proceedings*, pages 42–51, 1992.

[43] Alex Pothen and Chunguang Sun. Distributed multifrontal factorization using clique trees. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 34–40, 1991.

[44] Roland Pozo and Sharon L. Smith. Performance evaluation of the parallel multifrontal method in a distributed-memory environment. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 453–456, 1993.

[45] Edward Rothberg. Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon systems. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, May 1994.

[46] Edward Rothberg and Anoop Gupta. An efficient block-oriented approach to parallel sparse Cholesky factorization. In *Supercomputing '92 Proceedings*, 1992.

[47] Robert Schreiber. Scalability of sparse direct solvers. Technical Report RIACS TR 92.13, NASA Ames Research Center, Moffet Field, CA, May 1992. Also appears in A. George, John R. Gilbert, and J. W.-H. Liu, editors, *Sparse Matrix Computations: Graph Theory Issues and Algorithms* (An IMA Workshop Volume). Springer-Verlag, New York, NY, 1993.

[48] B. Speelpening. The generalized element method. Technical Report UIUCDCS-R-78-946, Department of Computer Science, University of Illinois, Urbana, IL, November 1978.

[49] Chunguang Sun. Efficient parallel solutions of large sparse SPD systems on distributed-memory multiprocessors. Technical Report CTC92TR102, Advanced Computing Research Institute, Center for Theory and Simulation in Science and Engineering, Cornell University, Ithaca, NY, August 1992.

[50] Sesh Venugopal and Vijay K. Naik. Effects of partitioning and scheduling sparse matrix factorization on communication and load balance. In *Supercomputing '91 Proceedings*, pages 866–875, 1991.

[51] Sesh Venugopal and Vijay K. Naik. SHAPE: A parallelization tool for sparse matrix computations. Technical Report DCS-TR-290, Department of Computer Science, Rutgers University, New Brunswick, NJ, June 1992.