Highly Scalable Parallel Algorithms for Sparse Matrix Factorization*

Anshul Gupta

IBM T. J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
anshul@watson.ibm.com

George Karypis and Vipin Kumar

Department of Computer Science
University of Minnesota
Minneapolis, MN 55455
{karypis/kumar}@cs.umn.edu

Abstract

In this paper, we describe scalable parallel algorithms for sparse matrix factorization, analyze their performance and scalability, and present experimental results for up to 1024 processors on a Cray T3D parallel computer. Through our analysis and experimental results, we demonstrate that our algorithms substantially improve the state of the art in parallel direct solution of sparse linear systems—both in terms of scalability and overall performance. It is a well known fact that dense matrix factorization scales well and can be implemented efficiently on parallel computers. In this paper, we present the first algorithms to factor a wide class of sparse matrices (including those arising from two- and three-dimensional finite element problems) that are asymptotically as scalable as dense matrix factorization algorithms on a variety of parallel architectures. Our algorithms incur less communication overhead and are more scalable than any previously known parallel formulation of sparse matrix factorization. Although, in this paper, we discuss Cholesky factorization of symmetric positive definite matrices, the algorithms can be adapted for solving sparse linear least squares problems and for Gaussian elimination of diagonally dominant matrices that are almost symmetric in structure. An implementation of one of our sparse Cholesky factorization algorithms delivers up to 20 GFlops on a Cray T3D for medium-size structural engineering and linear programming problems. To the best of our knowledge, this is the highest performance ever obtained for sparse Cholesky factorization on any supercomputer.

*This work was supported by IST/BMDO through Army Research Office contract DA/DAAH04-93-G-0080, NSF grant NSG/1RI-9216941, and by Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. Access to computing facilities were provided by Minnesota Supercomputer Institute, Cray Research Inc. and by the Pittsburgh Supercomputing Center. Related papers are available via WWW at URL: http://www.cs.umn.edu/users/kumar/papers.html.

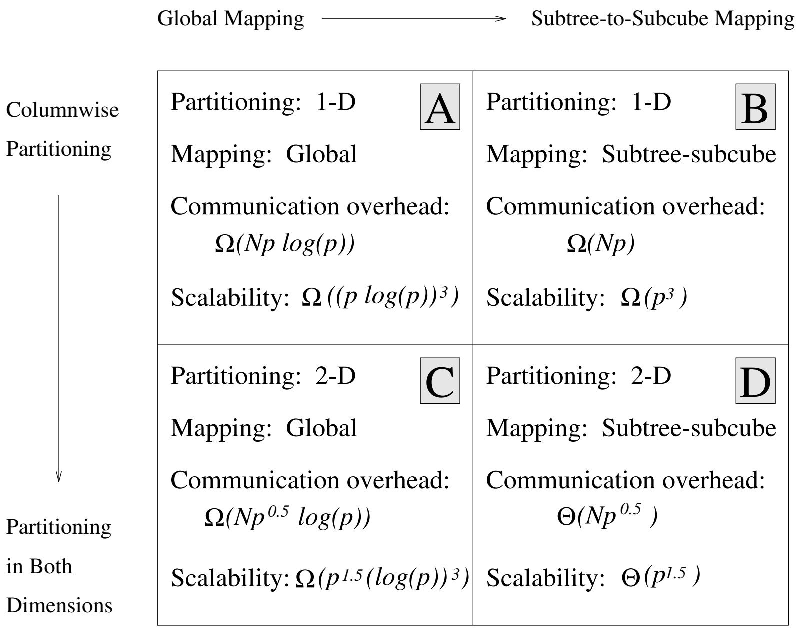
1 Introduction

Solving large sparse systems of linear equations is at the core of many problems in engineering and scientific computing. Such systems are typically solved by two different types of methods—*iterative methods* and *direct methods*. The nature of the problem at hand determines which method is more suitable. A direct method for solving a sparse linear system of the form $Ax = b$ involves explicit factorization of the sparse coefficient matrix A into the product of lower and upper triangular matrices L and U. This is a highly time and memory consuming step; nevertheless, direct methods are important because of their generality and robustness. For linear systems arising in certain applications, such as linear programming and structural engineering applications, they are the only feasible solution methods. In many other applications too, direct methods are often preferred because the effort involved in determining and computing a good preconditioner for an iterative solution may outweigh the cost of direct factorization. Furthermore, direct methods provide and effective means for solving multiple systems with the same coefficient matrix and different right-hand side vectors because the factorizations needs to be performed only once.

A wide class of sparse linear systems have a symmetric positive definite (SPD) coefficient matrix that is factored using Cholesky factorization. Although Cholesky factorization used extensively in practice, their use for solving large sparse systems has been mostly confined to big vector supercomputers due to its high time and memory requirements. As a result, parallelization of sparse Cholesky factorization has been the subject of intensive research [27, 59, 12, 15, 14, 18, 58, 41, 42, 3, 53, 54, 61, 9, 29, 27, 28, 55, 2, 1, 45, 62, 16, 59, 44, 34, 5, 43, 4, 63]. We have developed highly scalable formulations of sparse Cholesky factorization that substantially improve the state of the art in parallel direct solution of sparse linear systems—both in terms of scalability and overall performance. It is well known that dense matrix factorization can be implemented efficiently on distributed-memory parallel computers [8, 47, 10, 35]. We show that the parallel Cholesky factorization algorithms described here are 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. These algorithms incur less communication overhead than any known parallel formulation of sparse matrix factorization, and hence, can utilize a higher number of processors effectively. The algorithms presented here can deliver speedups in proportion to an increasing number of processors while requiring almost constant memory per processor.

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[1]. A simple fan-out algorithm [12] 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 [15] (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 [14] (box B). A number of column-based parallel factorization algorithms [41, 42, 3, 53, 54, 61, 12, 9, 29, 27, 59, 44, 5]

[1] In [48], Pan and Reif describe a parallel sparse matrix factorization algorithm for a PRAM type architecture. This algorithm is not cost-optimal (i.e., the processor-time product exceeds the serial complexity of sparse matrix factorization) and is not included in the classification given in Figure 1.



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.

have a lower bound of $O(Np)$ on the total communication volume [15]. Since the overall computation is only $O(N^{1.5})$ [13], 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 [59, 57]. In [2], Ashcraft proposes a fan-both family of parallel Cholesky factorization algorithms that have a total communication volume of $\Theta(N\sqrt{p}\log N)$. Although the communication volume is less than the other column-based partitioning schemes, the isoefficiency function of Ashcraft's algorithm is still $\Theta(p^3)$ due to concurrency constraints because the algorithm cannot effectively utilize more than $O(\sqrt{N})$ processors for matrices arising from two-dimensional constant node-degree graphs. Recently, a number of schemes with two-dimensional partitioning of the matrix have been proposed [18, 57, 56, 18, 58, 1, 45, 62, 16, 34, 43, 4, 63]. The least total communication volume in most of these schemes is $O(N\sqrt{p}\log p)$ (box C)[2].

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 [24, 35]. 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 et al. [3] serves to demonstrate the importance of studying the total communication overhead rather than volume. In [3], 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.

The performance and scalability analysis of our algorithm is supported by experimental results on up to 1024 processors of nCUBE2 [25] and Cray T3D parallel computers. 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 [6]. In [30], we have applied this algorithm to obtain a highly scalable parallel formulation of interior point algorithms and have observed significant speedups in solving linear programming problems. On the Cray T3D, we have been able to achieve 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 summary, researchers have improved the simple parallel algorithm with $O(Np\log p)$ communication volume (box A) 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 algorithms reduce the communication overhead by a factor of $O(\log p)$ over the best algorithm implemented to date. 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

[2][4] and [63] could be possible exceptions, but neither a detailed communication analysis, nor any experimental results are available.

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 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 [35, 37, 17] to determine the scalability of our algorithm and compare it with the scalability of other parallel algorithms for sparse matrix factorization. In Section 6 we present a variation of the algorithm described in Section 3 that reduces the overhead due to load imbalance. Section 7 contains the preliminary experimental results on a Cray T3D parallel computer. Section 8 contains concluding remarks.

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 [60] and Duff and Reid [7], and later elucidated in a tutorial by Liu [40]. 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).

/*
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.
*/

begin function Factor(k)

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

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

    for $i := 0$ **to** t **do**
        $l_{q_i,k} := F^k(i,0)/\sqrt{F^k(0,0)}$;
    for $j := 1$ **to** t **do**
        for $i := j$ **to** t **do**
            $U^k(i,j) := F^k(i,j) - l_{q_i,k} \times l_{q_j,k}$;

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.

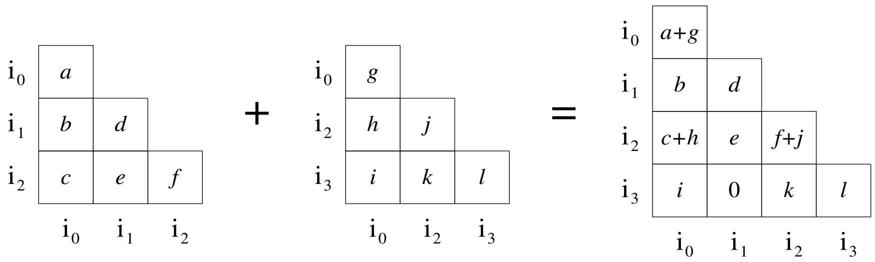


Figure 3: The extend-add operation on two 3×3 triangular matrices. It is assumed that $i_0 < i_1 < i_2 < i_3$.

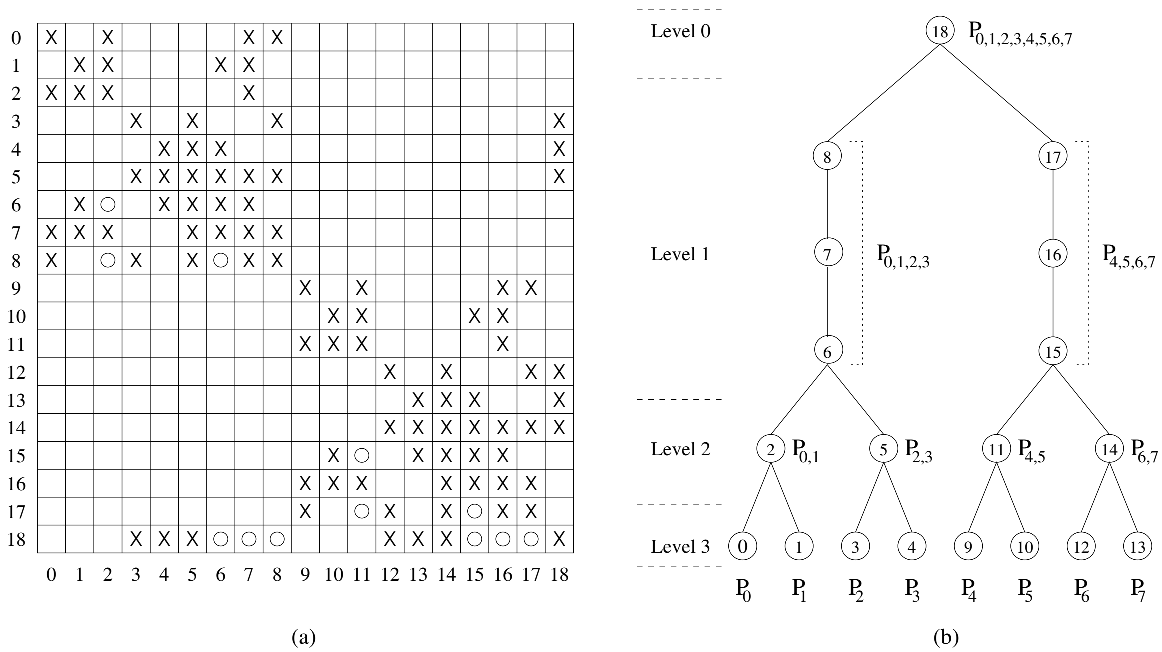


Figure 4: A symmetric sparse matrix and the associated elimination tree with subtree-to-subcube mapping onto 8 processors. The nonzeros in the original matrix are denoted by the symbol "$\times$" and fill-ins are denoted by the symbol "$\circ$".

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 described in detail in [25].

In order to factorize the sparse matrix in parallel, portions of the elimination tree are assigned to processors

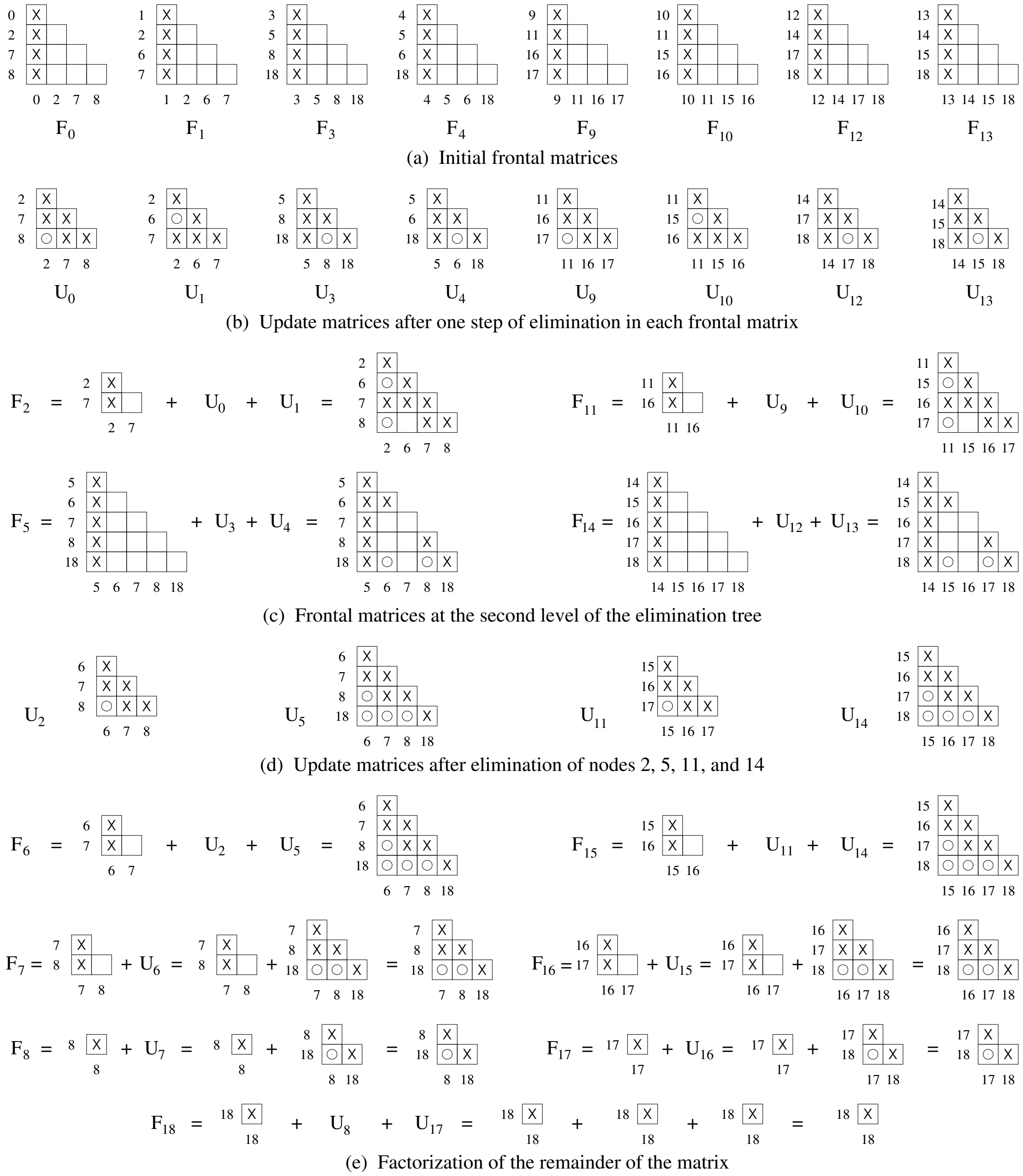


Figure 5: Steps in serial multifrontal Cholesky factorization of the matrix shown in Figure 4(a). The symbol "+" denotes an extend-add operation. The nonzeros in the original matrix are denoted by the symbol "×" and fill-ins are denoted by the symbol "○".

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 $0 \le j < p/2$) perform a parallel extend-add on their update matrices, say C and D, respectively. At the end of this parallel extend-add operation, P_{2j} and P_{2j+1} roughly equally share $C + D$. 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 $C + D$ go to P_{2j} and all odd columns of $C + D$ 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 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 C and D perform a parallel extend-add, then the final result at P_i will be the add-extension of the white portion of C and the grey portion of D. Similarly, the final result at P_j will be the add-extension of the grey portion of C and the white portion of D. Figure 7 further illustrates the 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

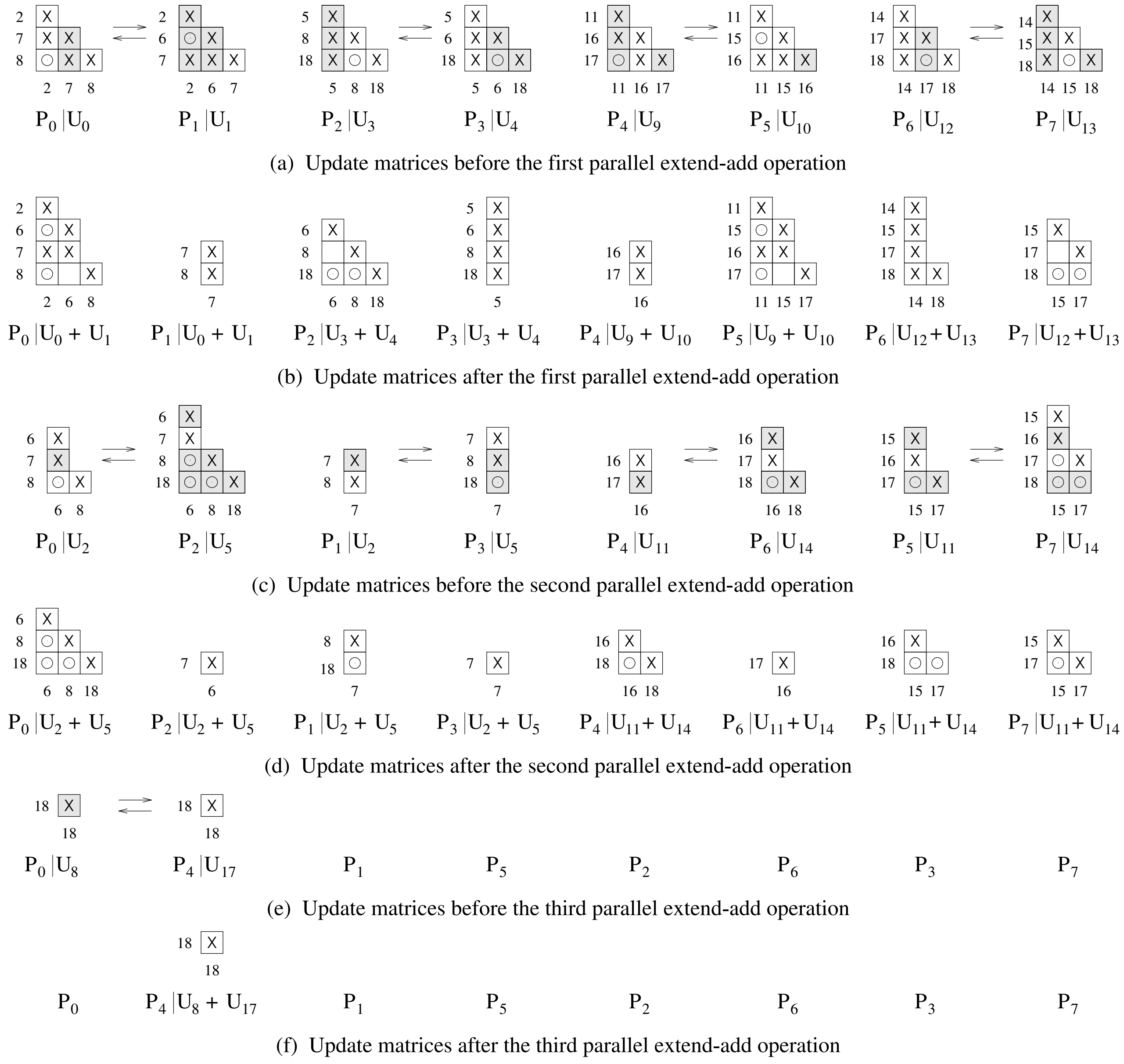
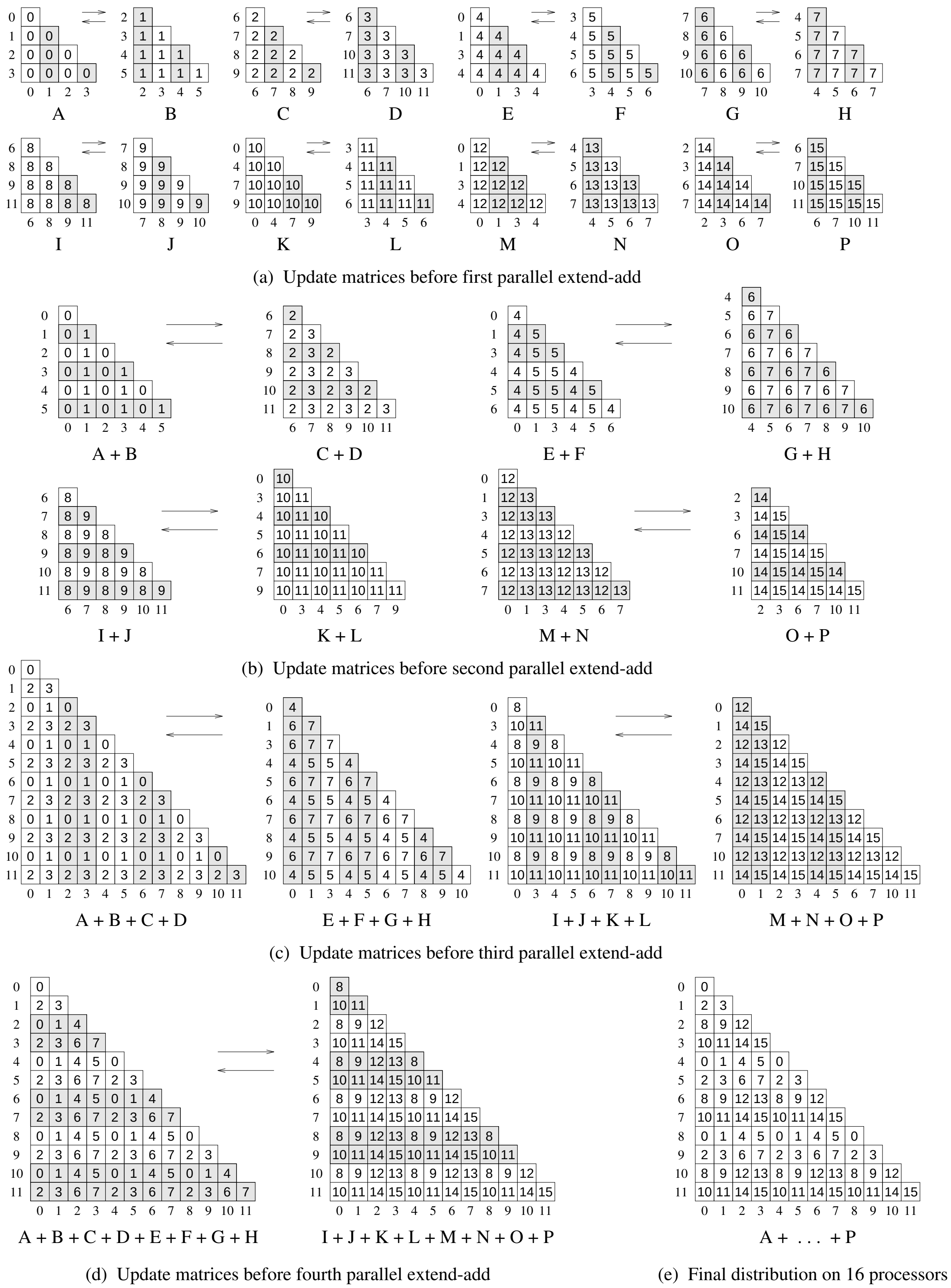


Figure 6: Extend-add operations on the update matrices during parallel multifrontal factorization of the matrix shown in Figure 4(a) on eight processors. $P_i|M$ denotes the part of the matrix M that resides on processor number i. M may be an update matrix or the result of performing an extend-add on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



(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.

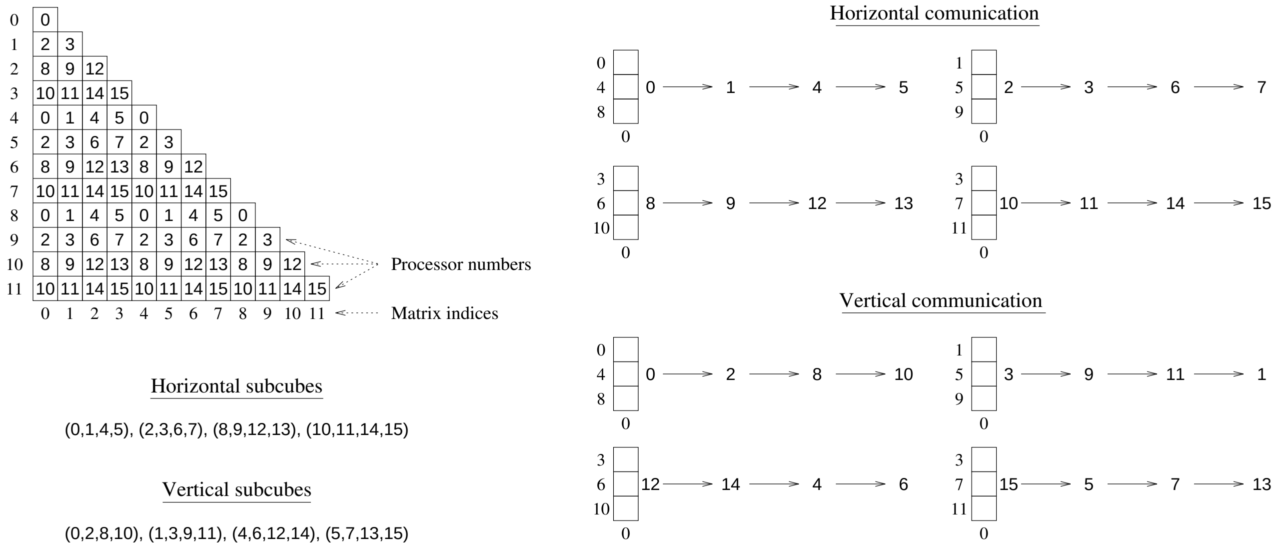


Figure 8: The two communication operations involved in a single elimination step (index of pivot = 0 here) of Cholesky factorization on a 12×12 frontal matrix distributed over 16 processors.

supernode being processed. The communication that takes place in this phase is the standard communication in pipelined grid-based dense Cholesky factorization [47, 35]. 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 [36] 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 [46, 35] 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.

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

(a) A logical mesh of 16 processors

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

Processor numbers

Matrix indices

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

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

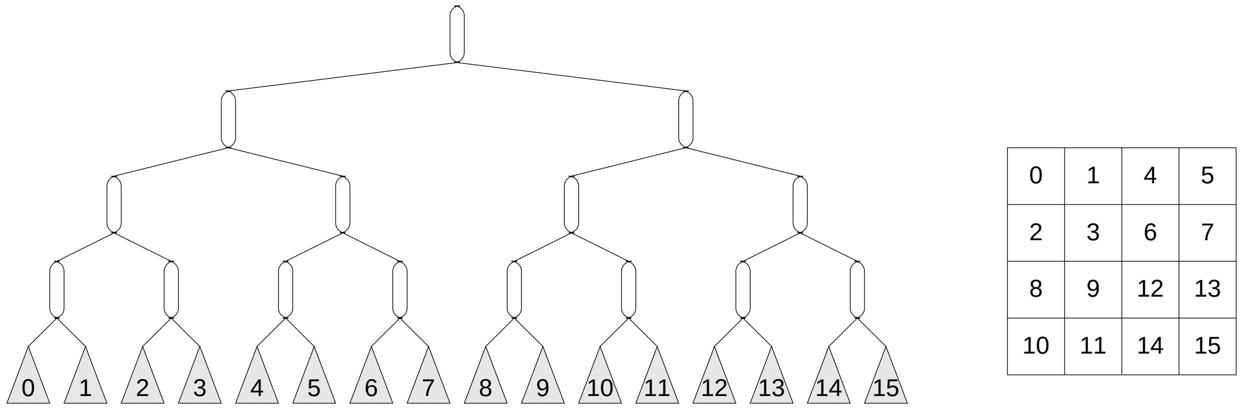
Furthermore, on cache-based processors, the use of BLAS-3 for eliminating multiple columns simultaneously yields much higher performance than the use of BLAS-2 for eliminating one column at a time. Figure 9(b) shows a variation of the cyclic mapping, called block-cyclic mapping [35], that can alleviate 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 addition, increasing the block size too much may cause too much load imbalance during the dense Cholesky steps and may offset the advantage of using BLAS-3.

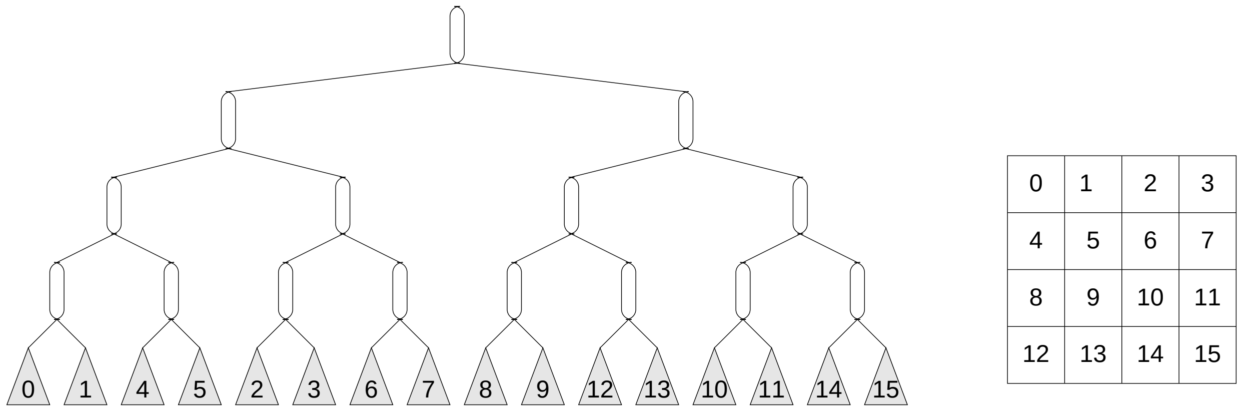
3.2 Subtree-to-submesh mapping for the 2-D mesh architecture

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 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



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



(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.

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:

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

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.

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 [39, 38, 13]. 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 [15] in the context of a column-based subtree-to-subcube scheme. Within very small constant factors, the analysis holds for the standard nested dissection [11] of grid graphs. We consider a cross-shaped separator (described in [15]) 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 [15] 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 [47, 35] 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 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 $O(i^4)$ [15]. At level l, due to l bisections, i is no more than $N^{1/3}/2^l$. 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 [35, 37, 17] 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 [13] 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 [57, 18] 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 $O(N^2)$ [15]. 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 $\Theta(p^{1.5})$ [35, 36] if the number of rank-1 updates performed by the serial algorithm is proportional to the rank of the matrix. The factorization of a sparse matrix derived from an N-node graph with an $S(N)$-node separator involves a dense $S(N) \times S(N)$ matrix factorization. $S(N)$ is $\Theta(\sqrt{N})$ and $\Theta(N^{2/3})$ for two- and three-dimensional constant node-degree graphs, respectively. Thus, the complexity of the dense portion of factorization for these two types of matrices is $\Theta(N^{1.5})$ and $\Theta(N^2)$, respectively, which is of the same order as the computation required to factor the entire sparse matrix [13, 15]. Therefore, the isoefficiency function of sparse factorization of such matrices is bounded from below by the isoefficiency function of dense matrix factorization, which is $\Theta(p^{1.5})$. As we have shown earlier in this section, our algorithm achieves this lower bound for both two- and three-dimensional cases.

5.3 Scalability with respect to memory requirement

We have shown that the problem size must increase in proportion to $p^{1.5}$ for our algorithm to achieve a fixed efficiency. As the overall problem size increases, so does the overall memory requirement. For an N-node two-dimensional constant node-degree graphs, the size of the lower triangular factor L is $\Theta(N \log N)$ [13]. For a fixed efficiency, $W = N^{1.5} \propto p^{1.5}$, which implies $N \propto p$ and $N \log N \propto p \log p$. As a result, if we increase the number of processors while solving bigger problems to maintain a fixed efficiency, the overall memory requirement increases at the rate of $\Theta(p \log p)$ and the memory requirement per processor increase logarithmically with respect to the number of processors.

In the three-dimensional case, size of the lower triangular factor L is $\Theta(N^{4/3})$ [13]. For a fixed efficiency, $W = N^2 \propto p^{1.5}$, which implies $N \propto p^{3/4}$ and $N^{4/3} \propto p$. Hence, in this case the overall memory requirement increases linearly with the number of processors and the per-processor memory requirement is constant for maintaining a fixed efficiency. It can be easily shown that for the three-dimensional case, the isoefficiency function should not be of a higher order than $\Theta(p^{1.5})$ if speedups proportional to the number of processors are desired without increasing the memory requirement per processor. To the best of our knowledge, the algorithm described in Section 3 is the only parallel algorithm for sparse Cholesky factorization with an isoefficiency function of $\Theta(p^{1.5})$.

6 An Improvement for Better Load Balance

We implemented the parallel multifrontal algorithm described in Section 3 on the nCUBE 2 parallel computer. The detailed experimental performance and scalability results of this implementation have been presented in [25]. Table 1 shows the results of factoring some matrices from the Harwell-Boeing collection of sparse matrices [6]. These results show that our algorithm can deliver good speedups on hundreds of processors for practical problems. Spectral nested dissection [50, 51, 52] was used to order these matrices.

The algorithm presented in Section 3 relies on the ordering algorithm to yield a balanced elimination tree. Imbalances in the elimination tree result in a loss in the efficiency of the parallel implementation. For example, the two subtrees of the top level relaxed supernode might require different amount of computation. Therefore, one half of the processors working on the smaller subtree will be idle after processing their subtree until the other half finishes the bigger subtree. Since subtree-to-subcube mapping is recursively applied in each subcube, the load imbalance at each level accumulates. In fact, in the results shown in Table 1, nearly half of the efficiency

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%
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%

Table 1: Experimental results on an nCUBE-2 for factoring some sparse symmetric positive definite matrices resulting from 3-D problems in structural engineering. All times are in seconds. The suffix "*" indicates run time estimated by timing the computation on two processors.

loss is due to load imbalance and the rest due to communication. We have experimentally shown in [25] that the overhead due do load imbalance tends to saturate as the number of processors increase and, therefore, does not affect the asymptotic scalability of the algorithm. However, load imbalance puts an upper bound on the achievable efficiency and results in a significant performance penalty.

In this section, we describe an algorithm that minimizes this drawback of a subtree-to-subcube mapping. This mapping assigns groups of subtrees to processor subcubes; hence, we call it subforest-to-subcube mapping.

6.1 Subforest-to-subcube mapping

In subforest-to-subcube mapping, we assign many subtrees of the elimination tree to each processor subcube. These trees are chosen in such a way that the total amount of work assigned to each subcube is as equal as possible. The best way to describe this partitioning scheme is via an example. Consider the elimination tree shown in Figure 11. Assume that it takes a total of 100 time units to factor the entire sparse matrix. Each node in the tree is marked with the number of time units required to factor the subtree rooted at this particular node (including the time required to factor the node itself). For instance, the subtree rooted at node B requires 65 units of time, while the subtree rooted at node F requires only 18.

As shown in Figure 11(b), the subtree-to-subcube mapping scheme will assign the computation associated with the top supernode A to all the processors, the subtree rooted at B to half the processors, and the subtree rooted at C to the remaining half of the processors. Since, these subtrees require different amount of computation, this particular partition will lead to load imbalances. Since 7 time units of work (corresponding to the node A) is distributed among all the processors, this factorization takes at least $7/p$ units of time. Now each subcube of $p/2$ processors independently works on each subtree. The time required for these subcubes to finish is lower-bounded by the time to perform the computation for the larger subtree (the one rooted at node B). Even if we assume that all subtrees of B are perfectly balanced, computation of the subtree rooted at B by $p/2$ processors will take at least $65/(p/2)$ time units. Thus, an upper bound on the efficiency of this mapping is only $100/(p(7/p + 65/(p/2))) \approx .73$. Now consider the following mapping scheme: The computation associated with supernodes A and B is assigned to all the processors. The subtrees rooted at E and C are assigned to half of the processors, while the subtree rooted at D is assigned to the remaining processors. In

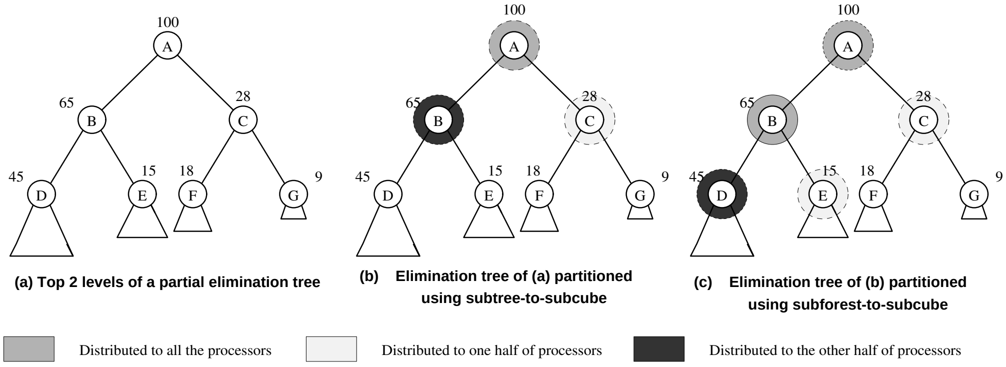


Figure 11: The top two levels of an elimination tree is shown in (a). The subtree-to-subcube mapping is shown in (b), the subforest-to-subcube mapping is shown in (c).

this mapping scheme, the first half of the processors are assigned 43 time units of work, while the other half is assigned 45 time units. The upper bound on the is $100/(p(12/p + 45/(p/2))) \approx 0.98$, which is a significant improvement over the earlier bound of .73.

The above example illustrates the basic ideas behind the new mapping scheme. The general mapping algorithm is outlined in Figure 12.

The tree partitioning algorithm uses a set C that contains the unassigned nodes of the elimination tree. The algorithm inserts the root of the elimination tree into C, and then it calls the routine *Elpart* that recursively partitions the elimination tree. *Elpart* partitions C into two parts, L and R and checks if this partitioning is acceptable. If yes, then it assigns L to half of the processors, and R to the remaining half, and recursively calls *Elpart* to perform the partitioning in each of these halves. If the partitioning is not acceptable, then one node of C (i.e., *node* = *select(C)*) is assigned to all the p processors, *node* is deleted from C, and the children of *node* are inserted into the C. The algorithm then continues by repeating the whole process. The above description provides a high level overview of the subforest-to-subcube partitioning scheme. However, a number of details need to be clarified. In particular, we need to specify how the *select*, *halfsplit*, and *acceptable* procedures work.

Selection of a node from C There are two different ways[3] of defining the procedure *select(C)*.

- One way is to select a node whose subtree requires the largest number of operations to be factored.
- The second way is to select a node that requires the largest number of operations to factor it.

The first method favors nodes whose subtrees require significant amount of computation. Thus, by selecting such a node and inserting its children in C we may get a good partitioning of C into two halves. However, this approach can assign nodes with relatively small computation to all the processors, causing poor efficiency in the factorization of these nodes. The second method guarantees that the selected node has more work, and thus its factorization can achieve higher efficiency when it is factored by all p processors. However, if the subtrees attached to this node are not large, then this may not lead to a good partitioning of C in later steps. In particular, if the root of the subtree having most of the remaining work, requires little computation (e.g., single

[3]Note, that the information required by these methods (the amount of computation to eliminate a node, or the total amount of computation associated with a subtree), can be easily obtained during the symbolic factorization phase.

```
Partition(T, p) /* Partition the tree T, among p processors. */
  C = { }
  Add root(T) into C
  Elpart(C, T, p)
End Partition

Elpart(C, T, p)
  if (p == 1) return
  done = false
  while (done == false)
    halfsplit(C, L, R)
    if (acceptable(L, R))
      Elpart(L, T, p/2)
      Elpart(R, T, p/2)
      done = true
    else
      node = select(C)
      delete(C, node)
      node => p /* Assign node to all p processors */
      Insert into C the children of node in T
  end while
End Elpart
```

Figure 12: The subforest-to-subcube partitioning algorithm.

node supernode), then the root of this subtree will not be selected for expansion until very late, leading to too many nodes being assigned at all the processors.

Another possibility is to combine the above two schemes and apply each one in alternate steps. This combined approach eliminates most of the limitations of the above schemes while retaining their advantages. This is the scheme we used in the experiments described in Section 7.

So far we considered only the floating point operations when we were referring to the number of operations required to factor a subtree. On systems where the cost of each memory access relative to a floating point operation is relatively high, a more accurate cost model will also take the cost of each extend-add operation into account. The total number of memory accesses required for extend-add can be easily computed from the symbolic factorization of the matrix.

Splitting The Set C In each step, the partitioning algorithm checks to see if it can split the set C into two roughly equal halves. The ability of the *halfsplit* procedure to find a partition of the nodes (and consequently create two subforests) is crucial to the overall ability of this partitioning algorithm to balance the computation. Fortunately, this is a typical bin-packing problem, and even though, bin-packing is NP complete, a number of good approximate algorithms exist [49]. The use of bin-packing makes it possible to balance the computation and to significantly reduce the load imbalance.

Acceptable Partitions A partition is acceptable if the percentage difference in the amount of work in the two parts is less than a small constant ϵ. If ϵ is chosen to be high (e.g., $\epsilon \geq 0.2$), then the subforest-to-subcube mapping becomes similar to the subtree-to-subcube mapping scheme. If ϵ is chosen to be too small, then most of the nodes of the elimination tree will be processed by all the processors, and the communication overhead during the dense Cholesky factorization will become too high. For example, consider the task of factoring two $n \times n$ matrices A and B on p-processor square mesh or a hypercube using a standard algorithm that uses two-dimensional partitioning and pipelining. If each of the matrices is factored by all the processors, then the total communication time for factoring the two matrices is $n^2/\sqrt{p}$ [35]. If A and B are factored concurrently by $p/2$ processors each, then the communication time is $n^2/(2\sqrt{p/2})$ which is smaller. Thus the value of ϵ has to be chosen to strike a good balance between these two conflicting goals of minimizing load imbalance and the communication overhead in individual factorization steps. For the experiments reported in Section 7, we used $\epsilon = 0.05$.

Impact on Communication Overhead Note that the communication overhead of subforest-to-subcube mapping is somewhat higher than that of subtree-to-subcube mapping. This is is mainly because subforest-to-subcube mapping results in smaller frontal matrices being mapped onto larger groups of processor. However, we have proved in [23] that the asymptotic bounds on the communication overhead of subforest-to-subcube mapping are the same as those of subtree-to-subcube mapping. Therefore, the algorithm described in this section is equally scalable as the one discussed in 3. The actual impact on the performance depends on the ratio of communication and computation speeds of the parallel computer being used. Faster communication relative to computation will permit a smaller value of ϵ to be used, resulting in a finer load balance.

7 Experimental Results

We implemented our new parallel sparse multifrontal algorithm on a 1024-processor Cray T3D parallel computer. Each processor on the T3D is a 150 Mhz Dec Alpha chip, with peak performance of 150 MFlops for 64-bit operations (double precision). However, the peak performance of most level three BLAS routines is around 50 MFlops. The processors are interconnected via a three dimensional torus network that has a peak unidirectional bandwidth of 150 MBytes per second, and a very small latency. Even though the memory on T3D is physically distributed, it can be addressed globally. That is, processors can directly access (read and/or write) other processor's memory. T3D provides a library interface to this capability called SHMEM. We used SHMEM to develop a lightweight message passing system. Using this system we were able to achieve unidirectional data transfer rates up to 70 MBytes per second. This is significantly higher than the 35 MBytes channel bandwidth usually obtained when using T3D's PVM.

For the computation performed during the dense Cholesky factorization, we used single-processor implementation of BLAS primitives. These routines are part of the standard scientific library on T3D, and they have been fine tuned for the Alpha chip. The new algorithm was tested on matrices from a variety of sources. Four matrices (BCSSTK30, BCSSTK31, BCSSTK32, and BCSSTK33) come from the Boeing-Harwell matrix set. MAROS-R7 is from a linear programming problem taken from NETLIB. COPTER2 comes from a model of a helicopter rotor. CUBE35 is a $35 \times 35 \times 35$ regular three-dimensional grid. NUG15 is from a linear programming problem derived from a quadratic assignment problem obtained from AT&T. In all of our experiments, we used spectral nested dissection [50] to order the matrices. The factorization algorithms described in this paper will work well with any type of nested dissection. In [21, 22, 20, 32, 31], we show that nested dissection orderings with proper selection of separators can yield better quality orderings that traditional heuristics, such as, the multiple minimum degree heuristic.

The performance obtained by this algorithm in some of these matrices is shown in Table 2. The operation count shows only the number of operations required to factor the nodes of the elimination tree.

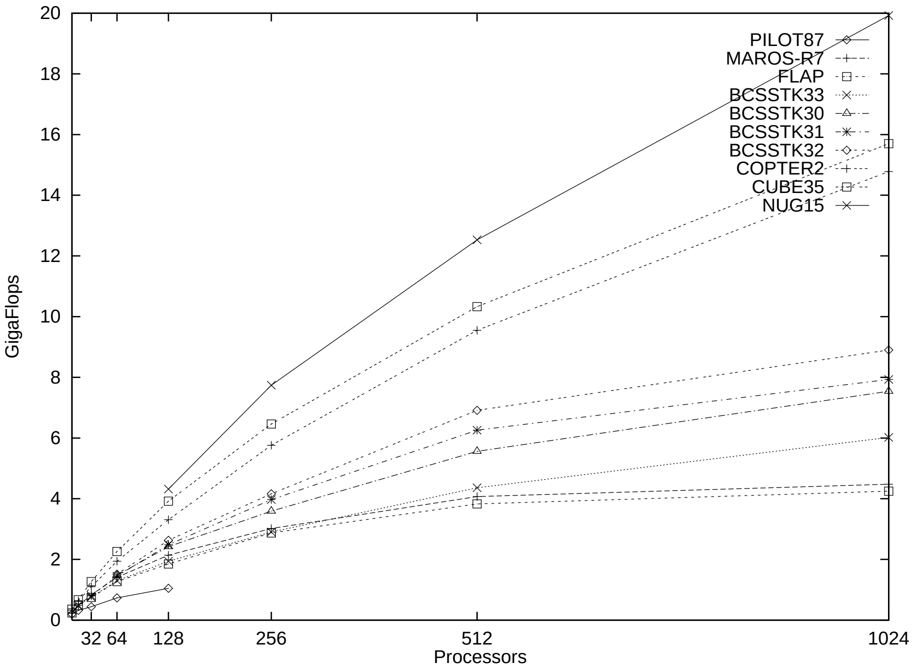
Figure 13 graphically represents the data shown in Table 2. Figure 13(a) shows the overall performance obtained versus the number of processors, and is similar in nature to a speedup curve. Figure 13(b) shows the per processor performance versus the number of processors, and reflects reduction in efficiency as p increases. Since all these problems run out of memory on one processor, the standard speedup and efficiency could not be computed experimentally.

					Number of Processors						
Problem	n	$\|A\|$	$\|L\|$	OPC	16	32	64	128	256	512	1024
PILOT87	2030	122550	504060	240M	0.32	0.44	0.73	1.05			
MAROS-R7	3136	330472	1345241	720M	0.48	0.83	1.41	2.14	3.02	4.07	4.48
FLAP	51537	479620	4192304	940M	0.48	0.75	1.27	1.85	2.87	3.83	4.25
BCSSTK33	8738	291583	2295377	1000M	0.49	0.76	1.30	1.94	2.90	4.36	6.02
BCSSTK30	28924	1007284	5796797	2400M			1.48	2.42	3.59	5.56	7.54
BCSSTK31	35588	572914	6415883	3100M		0.80	1.45	2.48	3.97	6.26	7.93
BCSSTK32	44609	985046	8582414	4200M			1.51	2.63	4.16	6.91	8.90
COPTER2	55476	352238	12681357	9200M	0.64	1.10	1.94	3.31	5.76	9.55	14.78
CUBE35	42875	124950	11427033	10300M	0.67	1.27	2.26	3.92	6.46	10.33	15.70
NUG15	6330	186075	10771554	29670M				4.32	7.54	12.53	19.92

Table 2: The performance of sparse direct factorization on Cray T3D. For each problem the table contains the number of equations n of the matrix A, the original number of nonzeros in A, the nonzeros in the Cholesky factor L, the number of operations required to factor the nodes, and the performance in gigaflops for different number of processors.

The highest performance of 19.9 GFlops was obtained for NUG15, which is a fairly dense problem. Among the sparse problems, a performance of 15.7 GFlops was obtained for CUBE35, which is a regular three-dimensional problem. Nearly as high performance (14.78 GFlops) was also obtained for COPTER2 which is irregular. Since both problems have similar operation count, this shows that our algorithm performs equally well in factoring matrices arising in irregular problems. Focusing our attention on the other problems shown in Table 2, we see that even on smaller problems, our algorithm performs quite well. For example, BCSSTK33 was able to achieve 2.90 GFlops on 256 processors and BCSSTK30 achieved 3.59 GFlops.

To further illustrate how various components of our algorithm work, we have included a breakdown of the various phases for BCSSTK31 and CUBE35 in Table 3. This table shows the average time spent by all the processors in the local computation and in the distributed computation. Furthermore, we break down the time taken by distributed computation into two major phases, (a) dense Cholesky factorization, (b) extend-add overhead. The latter includes the cost of performing the extend-add operation, splitting the stacks, transferring



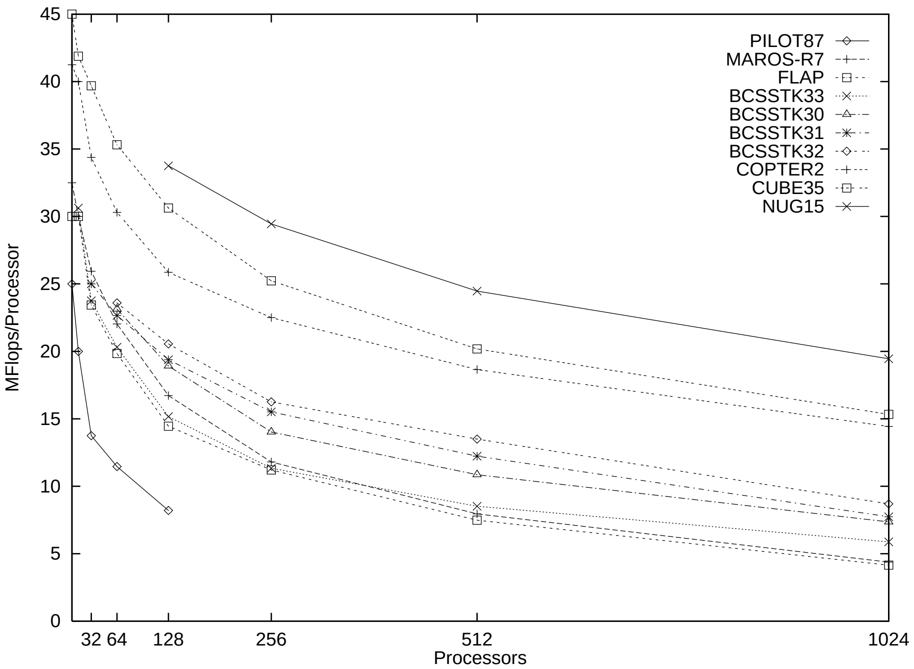


Figure 13: Plot of the performance of the parallel sparse multifrontal algorithm for various problems on Cray T3D. The first plot shows total Gigaflops obtained and the second one shows Megaflops per processor.

the stacks, and idling due to load imbalances in the subforest-to-subcube partitioning. Note that the figures in this table are averages over all processors, and they should be used only as an approximate indication of the time required for each phase.

A number of interesting observations can be made from this table. First, as the number of processors increases, the time spent processing the local tree in each processor decreases substantially because the subforest assigned to each processor becomes smaller. This trend is more pronounced for three-dimensional problems, because they tend to have fairly shallow trees. The cost of the distributed extend-add phase decreases almost linearly as the number of processors increases. This is consistent with the analysis presented in 4, since the overhead of distributed extend-add is $O((n \log p)/p)$. Since the expression for the time spent during the extend-add steps also includes the idling due to load imbalance, the almost linear decrease also shows that the load imbalance is quite small.

The time spent in distributed dense Cholesky factorization decreases as the number of processors increases. This reduction is not linear with respect to the number of processors for two reasons: (a) the ratio of communication to computation during the dense Cholesky factorization steps increases, and (b) for a fixed size problem load imbalances due to the block cyclic mapping becomes worse as p increases.

For reasons discussed in Section 3.1, we distributed the frontal matrices in a block-cyclic fashion. To get good performance on Cray T3D out of level three BLAS routines, we used a block size of sixteen (block sizes of less than sixteen result in degradation of level 3 BLAS performance on Cray T3D) For small problems, such a large block size results in a significant load imbalance within the dense factorization phase. This load imbalance becomes worse as the number of processors increases. However, as the size of the problem increases, both the communication overhead during dense Cholesky and the load imbalance due to the block cyclic mapping becomes less significant. The reason is that larger problems usually have larger frontal matrices at the top levels of the elimination tree, so even large processor grids can be effectively utilized to factor them. This is illustrated by comparing how the various overheads decrease for BCSSTK31 and CUBE35. For example, for BCSSTK31, the factorization on 128 processors is only 48% faster compared to 64 processors, while for CUBE35, the factorization on 128 processors is 66% faster compared to 64 processors.

			Distributed Computation	
	p	Local Comp.	Factorization	Extend-Add
BCSSTK31	64	0.17	1.34	0.58
	128	0.06	0.90	0.32
	256	0.02	0.61	0.18
CUBE35	64	0.15	3.74	0.71
	128	0.06	2.25	0.43
	256	0.01	1.44	0.24

Table 3: A break-down of the various phases of the sparse multifrontal algorithm for BCSSTK31 and CUBE35. Each number represents time in seconds.

To see the effect of the choice of ϵ in the overall performance of the sparse factorization algorithm we factored BCSSTK31 on 128 processors using $\epsilon = 0.4$ and $\epsilon = 0.0001$. Using these values for ϵ we obtained a performance of 1.18 GFlops when $\epsilon = 0.4$, and 1.37 GFlops when $\epsilon = 0.0001$. In either case, the performance is worse than the 2.48 GFlops obtained for $\epsilon = 0.05$. When $\epsilon = 0.4$, the mapping of the elimination tree to the processors resembles that of the subtree-to-subcube allocation. Thus, the performance degradation is due to the elimination tree load imbalance. When $\epsilon = 0.0001$, the elimination tree mapping assigns a large number of nodes to all the processors, leading to poor performance during the dense Cholesky factorization.

8 Concluding Remarks

In this paper, we analytically and experimentally demonstrate that scalable parallel implementations of direct methods for solving large sparse systems are possible. We describe an implementation on Cray T3D that yields up to 20 GFlops on medium-size problems. We use the isoefficiency metric [35, 37, 17] to characterize the scalability of our algorithms. We show that the isoefficiency function of our algorithms 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 [59], 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.

The process of obtaining a direct solution of a sparse system of linear equations of the form $Ax = b$ consists of the following four phases: **Ordering**, which determines permutation of the coefficient matrix A such that the factorization incurs low fill-in and is numerically stable; **Symbolic Factorization**, which determines the structure of the triangular matrices that would result from factorizing the coefficient matrix resulting from the ordering step; **Numerical Factorization**, which is the actual factorization step that performs arithmetic operations on the coefficient matrix A to performs arithmetic operations on the coefficient matrix A to produce a lower triangular matrix L and an upper triangular matrix U; and **Solution of Triangular Systems**, which produces the solution vector x by performing forward and backward eliminations on the triangular matrices resulting from numerical factorization. Numerical factorization is the most time-consuming of these four phases. However, in order to maintain the scalability of the entire solution process and to get around single-processor memory constraints, the other three phases need to be parallelized as well. We have developed parallel algorithms for the other phases that are tailored to work in conjunction with the numerical factorization algorithm. In [33], we describe an efficient parallel algorithm for determining fill-reducing orderings for parallel factorization of sparse matrices. This algorithm, while performing the ordering in parallel, also distributes the data among the processors in way that the remaining steps can be carried out with minimum data-movement. At the end of the parallel ordering step, the parallel symbolic factorization algorithm described in [19] can proceed without any redistribution. In [19, 26], we present efficient parallel algorithms for solving the upper and lower triangular systems. The experimental results in [19, 26] show that the data mapping scheme described in Section 3 works well for triangular solutions. We hope that the work presented in this paper, along with [19, 26, 33] will enable the development of efficient practical parallel solvers for a broad range of scientific computing problems.

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, 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.

[4] J. M. Conroy. Parallel nested dissection. *Parallel Computing*, 16:139–156, 1990.

[5] J. M. Conroy, S. G. Kratzer, and R. F. Lucas. Multifrontal sparse solvers in message passing and data parallel environments - a comparitive study. In *Proceedings of PARCO*, 1993.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] Anoop Gupta and Edward Rothberg. An efficient block-oriented approach to parallel sparse Cholesky factorization. In *Supercomputing '93 Proceedings*, 1993.

[19] Anshul Gupta. *Analysis and Design of Scalable Parallel Algorithms for Scientific Computing*. PhD thesis, University of Minnesota, Minneapolis, MN, 1995.

[20] Anshul Gupta. Fast and effective algorithms for graph partitioning and sparse matrix reordering. Technical Report (Number to be assigned), IBM T. J. Watson Research Center, Yorktown Heights, NY, 1996.

[21] Anshul Gupta. Graph partitioning based sparse matrix ordering algorithms for interior-point methods. Technical Report RC 20467 (90480), IBM T. J. Watson Research Center, Yorktown Heights, NY, May 21, 1996. Submitted for publication in *SIAM Journal on Optimization*.

[22] Anshul Gupta. WGPP: Watson graph partitioning (and sparse matrix ordering) package: Users manual. Technical Report RC 20453 (90427), IBM T. J. Watson Research Center, Yorktown Heights, NY, May 6, 1996. Also available at *http://www.cs.umn.edu/Research/ibm-cluster/refs/WGPP_Users_Manual.ps.Z.*

[23] 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*.

[24] 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.

[25] Anshul Gupta and Vipin Kumar. A scalable parallel algorithm for sparse matrix factorization. Technical Report 94-19, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994. A short version appears in *Supercomputing '94 Proceedings*. TR available in *users/kumar* at anonymous FTP site *ftp.cs.umn.edu*.

[26] Anshul Gupta and Vipin Kumar. Parallel algorithms for forward and back substitution in direct solution of sparse linear systems. In *Supercomputing '95 Proceedings*, December 1995.

[27] 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.

[28] M. T. Heath and Padma Raghavan. Distributed solution of sparse linear systems. Technical Report 93-1793, Department of Computer Science, University of Illinois, Urbana, IL, 1993.

[29] 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.

[30] 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*.

[31] George Karypis and Vipin Kumar. Analysis of multilevel graph partitioning. Technical Report TR 95-037, Department of Computer Science, University of Minnesota, 1995.

[32] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, Department of Computer Science, University of Minnesota, 1995.

[33] George Karypis and Vipin Kumar. Parallel multilevel graph partitioning. Technical Report TR 95-036, Department of Computer Science, University of Minnesota, 1995.

[34] S. G. Kratzer and A. J. Cleary. Sparse matrix factorization on simd parallel computers. In A. George, John R. Gilbert, and J. W.-H. Liu, editors, *Graph Theory and Sparse Matrix Computations*. Springer-Verlag, New York, NY, 1993.

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

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

[37] 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.

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

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

[40] 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.

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

[42] 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.

[43] F. Manne. *Load Balancing in Parallel Sparse Matrix Computations*. PhD thesis, University of Bergen, Norway, 1993.

[44] 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.

[45] 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.

[46] 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.

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

[48] V. Pan and J. H. Reif. Efficient parallel solution of linear systems. In *17th Annual ACM Symposium on Theory of Computing*, pages 143–152, 1985.

[49] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization, Algorithms and Complexity*. Prentice Hall, 1982.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] Padma Raghavan. *Distributed sparse matrix factorization: QR and Cholesky factorizations*. PhD thesis, Pennsylvania State University, University Park, PA, 1991.

[56] 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.

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

[58] Edward Rothberg and Robert Schreiber. Improved load distribution in parallel sparse Cholesky factorization. In *Supercomputing '94 Proceedings*, 1994.

[59] 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.

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

[61] 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.

[62] 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.

[63] P. H. Worley and Robert Schreiber. Nested dissection on a mesh connected processor array. In Arthur Wouk, editor, *New Computing Environments: Parallel, Vector, and Systolic*, pages 8–38. SIAM, Philadelphia, PA, 1986.