A High Performance Sparse Cholesky Factorization Algorithm For Scalable Parallel Computers

George Karypis and Vipin Kumar
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

Technical Report 94-41

Abstract

This paper presents a new parallel algorithm for sparse matrix factorization. This algorithm uses subforest-to-subcube mapping instead of the subtree-to-subcube mapping of another recently introduced scheme by Gupta and Kumar [13]. Asymptotically, both formulations are equally scalable on a wide range of architectures and a wide variety of problems. But the subtree-to-subcube mapping of the earlier formulation causes significant load imbalance among processors, limiting overall efficiency and speedup. The new mapping largely eliminates the load imbalance among processors. Furthermore, the algorithm has a number of enhancements to improve the overall performance substantially. This new algorithm achieves up to 6GFlops on a 256-processor Cray T3D for moderately large problems. To our knowledge, this is the highest performance ever obtained on an MPP for sparse Cholesky factorization.

1 Introduction

Direct methods for solving sparse linear systems are important because of their generality and robustness. For linear systems arising in certain applications, such as linear programming and some structural engineering applications, they are the only feasible methods for numerical factorization. It is well known that dense matrix factorization can be implemented efficiently on distributed-memory parallel computers [4, 27, 7, 22]. However, despite inherent parallelism in sparse sparse direct methods, not much success has been achieved to date in developing their scalable parallel formulations [15, 38], and for several years, it has been a challenge to implement efficient sparse linear system solvers using direct methods on even moderately parallel computers. In [38], 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.

A parallel formulation for sparse matrix factorization can be easily obtained by simply distributing rows to different processors [8]. Due to the sparsity of the matrix, communication overhead is a large fraction of the computation for this method, resulting in poor scalability. In particular, for sparse matrices arising out of planar finite element graphs, the isoefficiency of such a formulation is $O(p^3 \log^3 p)$, that is the problem size (in terms of total number of computation) should grow as $O(p^3 \log^3 p)$ to maintain a fixed efficiency. In a smarter parallel formulation [11], the rows of the matrix are allocated to processors using the subtree-to-subcube mapping. This localizes the communication among groups of processors, and thus improves the isoefficiency of the scheme to $O(p^3)$. Rothberg and Gupta [36, 35] used a different method to reduce the communication overhead. In their method, the entire sparse matrix is partitioned among processors using a two-dimensional block cyclic mapping. This reduces the communication overhead and improves the isoefficiency to $O(p^{1.5} \log^3 p)$.

Gupta and Kumar [13] recently developed a parallel formulation of sparse Cholesky factorization based on the multifrontal method. The *multifrontal* method [2, 23] is a form of submatrix Cholesky, in which single elimination steps are performed on a sequence of small, dense *frontal matrices*. One of the advantages of multifrontal methods is that the frontal matrices are dense, and therefore the elimination steps can be implemented efficiently using level three BLAS primitives. This algorithm has two key features. It uses the subtree-to-subcube mapping to localize communication among processors, and it uses the highly scalable two-dimensional grid partitioning for dense matrix factorization for each supernodal computation in the multifrontal algorithm. As a result, the communication overhead of this scheme is the lowest of all other known parallel formulations for sparse matrix factorization [24, 25, 1, 31, 32, 39, 8, 38, 17, 33, 37, 3, 6, 18, 15, 40, 26, 12, 36, 35]. In fact, asymptotically, the isoefficiency of this scheme is $O(p^{1.5})$ for sparse matrices arising out of two- and three-dimensional finite element problems on a wide variety of architectures such as hypercube, mesh, fat tree, and three-dimensional torus. Note that the isoefficiency of the best known parallel formulation of dense matrix factorization is also $O(p^{1.5})$ [22]. On a variety of problems, Gupta and Kumar report speedup of up to 364 on a 1024-processor nCUBE 2, which is a major improvement over the previously existing algorithms.

However, the subtree-to-subcube mapping results in gross imbalance of load among different processors, as elimination trees for most practical problems tend to be unbalanced. This load imbalance is responsible for a major portion of the efficiency loss of their scheme. Furthermore, the overall computation rate of their single processor multifrontal code on nCUBE 2 was only 0.7MFlops and the maximum overall performance on a 1028-processor nCUBE 2 was only 300MFlops. This was partly due to the slow processors of nCUBE 2 (3.5 MFlops peak), and partly due to inadequacies in the implementation.

This paper presents a new parallel algorithm for sparse matrix factorization. This algorithm uses subforest-to-subcube mapping instead of the subtree-to-subcube mapping of the old scheme. The new mapping largely eliminates the load imbalance among processors. Furthermore, the algorithm has a number of enhancements to improve the overall performance substantially. This new algorithm achieves up to 6GFlops on a 256-processor Cray T3D for moderately large problems (even the biggest problem we tried took less than two seconds on a 256-node T3D. For larger problems, even higher performance can be achieved). To our knowledge, this is the highest performance ever obtained on an MPP for sparse Cholesky factorization. Our new scheme, like the scheme of Gupta and Kumar [13], has an asymptotic isoefficiency of $O(p^{1.5})$ for matrices arising out of two- and three-dimensional finite element problems on a wide variety of architectures such as hypercube, mesh, fat tree, and three-dimensional torus.

The rest of the paper is organized as follows. Section 2 presents a general overview of the Cholesky factorization process and multifrontal methods. Section 3 provides a brief description of the algorithm in [13]. Section 4 describes our new algorithm. Section 5 describes some further enhancements of the algorithm that significantly improve the performance. Section 6 provides the experimental evaluation of our new algorithms on a Cray T3D. Section 7 contains concluding remarks.

Due to space limitations, many important topics, including the theoretical performance analysis of our algorithm have been moved to the appendices.

2 Cholesky Factorization

Consider a system of linear equations

$$Ax = b$$

where A is an $n \times n$ symmetric positive definite matrix, b is a known vector, and x is the unknown solution vector to be computed. One way to solve the linear system is first to compute the Cholesky factorization

$$A = LL^T,$$

where the Cholesky factor L is a lower triangular matrix. The solution vector x can be computed by successive forward and back substitutions to solve the triangular systems

$$Ly = b, \quad L^T x = y.$$

If A is sparse, then during the course of the factorization, some entries that are initially zero in the upper triangle of A may become nonzero entries in L. These newly created nonzero entries of L are known as *fill-in*. The amount of fill-in generated can be decreased by carefully reordering the rows and columns of A prior to factorization. More precisely, we can choose a permutation matrix P such that the Cholesky factors of PAP^T have minimal fill-in. The problem of finding the best ordering for M that minimizes the amount of fill-in is NP-complete [41], therefore a number of heuristic algorithms for ordering have been developed. In particular, minimum degree ordering [9, 14, 10] is found to have low fill-in.

For a given ordering of a matrix, there exists a corresponding elimination tree. Each node in this tree is a column of the matrix. Node j is the parent of node i ($j > i$) if $l_{i,j}$ is the first nonzero entry in column i. Elimination of rows in different subtrees can proceed concurrently. For a given matrix, elimination trees of smaller height usually have greater concurrency than trees of larger height. A desirable ordering for parallel computers must increase the amount of concurrency without increasing fill-in substantially. Spectral nested dissection [29, 30, 19] has been found to generate orderings that have both low fill-in and good parallelism. For the experiments presented in this paper we used spectral nested dissection. For a more extensive discussion on the effect of orderings to the performance of our algorithm refer to [21].

In the multifrontal method for Cholesky factorization, a frontal matrix F_k and an update matrix U_k is associated with each node k of the elimination tree. The rows and columns of F_k corresponds to $t+1$ indices 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. The tree is traversed in a postorder sequence. When the subtree rooted at a node k has been traversed, then F_k becomes a dense $(t+1) \times (t+1)$ matrix, where t is the number of off diagonal nonzeros in L_k.

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 matrix 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. After F_k has been assembled, a single step of the standard dense Cholesky factorization is performed with node k as the pivot. 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. Since matrices are symmetric, only the upper triangular part is stored. For further details on the multifrontal method, the reader should refer to Appendix A, and to the excellent tutorial by Liu [23].

If some consecutively numbered nodes form a chain in the elimination tree, and the corresponding rows of L have identical nonzero structure, then this chain is called a *supernode*. The *supernodal elimination tree* is similar to the elimination tree, but nodes forming a supernode are collapsed together. In the rest of this paper we use the supernodal multifrontal algorithm. Any reference to the elimination tree or a node of the elimination tree actually refers to a supernode and the supernodal elimination tree.

3 Earlier Work on Parallel Multifrontal Cholesky Factorization

In this section we provide a brief description of the algorithm by Gupta and Kumar. For a more detailed description the reader should refer to [13].

Consider a p-processors hypercube-connected computer. Let A be the $n \times n$ matrix to be factored, and let T be its supernodal elimination tree. The algorithm requires the elimination tree to be binary for the first $\log p$ levels. Any elimination tree of arbitrary shape can be converted to a binary tree using a simple tree restructuring algorithm described in [19].

In this scheme, portions of the elimination tree are assigned to processors using the standard subtree-to-subcube assignment strategy [11, 14] illustrated in Figure 1. With subtree-to-subcube assignment, all p processors in the system cooperate to factor the frontal matrix associated with the root node of the elimination tree. The two subtrees of the root node 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 $\log p$ levels are each assigned to individual processors. Each processor can process this part of the tree completely independently without any communication overhead.

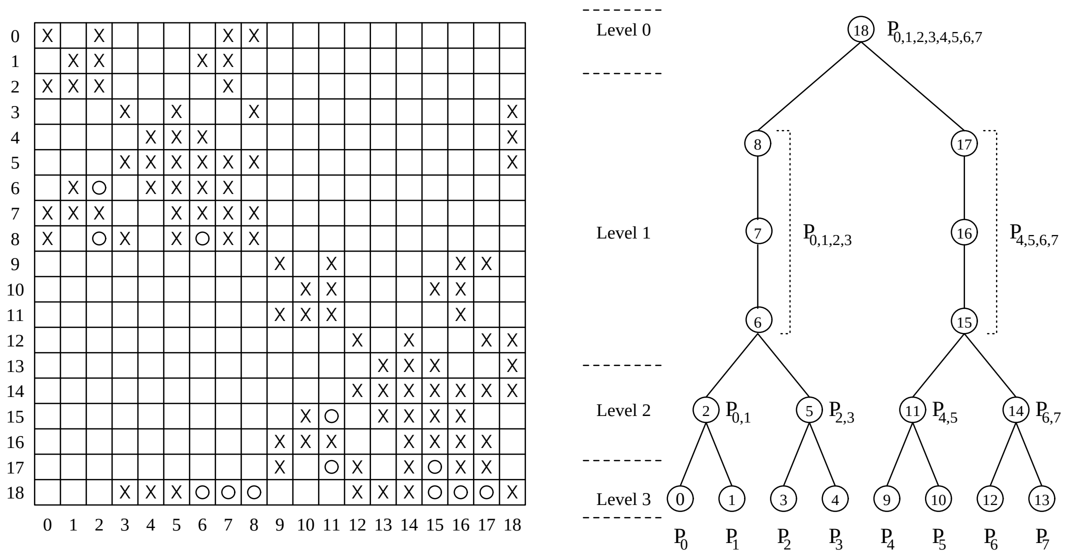


Figure 1: The elimination tree associated with a sparse matrix, and the subtree-to-subcube mapping of the tree onto eight processors.

Assume that the levels of the binary supernodal elimination tree are labeled from top starting with 0. In general, at level l of the elimination tree, $2^{\log p - l}$ processors work on a single frontal or update matrix. These processors form a logical $2^{\lceil(\log p - l)/2\rceil} \times 2^{\lfloor(\log p - l)/2\rfloor}$ grid. All update and frontal matrices at this level are distributed on this grid of processors. To ensure load balance during factorization, the rows and columns of these matrices are distributed in a cyclic fashion.

Between two successive extend-add operations, the parallel multifrontal algorithm performs a dense Cholesky factorization of the frontal matrix corresponding to the root of the subtree. Since the tree is supernodal, this step usually requires the factorization of several nodes. The communication taking place in this phase is the standard communication in grid-based dense Cholesky factorization.

Each processor participates in $\log p$ distributed extend-add operations, in which the update matrices from the factorization at level l are redistributed to perform the extend-add operation at level $l-1$ prior to factoring the frontal matrix. In the algorithm proposed in [13], each processor exchanges data with only one other processor during each one of these $\log p$ distributed extend-adds. The above is achieved by a careful embedding of the processor grids on the hypercube, and by carefully mapping rows and columns of each frontal matrix onto this grid. This mapping is described in [21], and is also given in Appendix B.

4 The New Algorithm

As mentioned in the introduction, the subtree-to-subcube mapping scheme used in [13] does not distribute the work equally among the processors. This load imbalance puts an upper bound on the achievable efficiency. For example, consider the supernodal elimination tree shown in Figure 2. This elimination tree is partitioned among 8 processors using the subtree-to-subcube allocation scheme. All eight processors factor the top node, processors zero through three are responsible for the subtree rooted at 24–27, and processors four through seven are responsible for the subtree rooted at 52–55. The subtree-to-subcube allocation proceeds recursively in each subcube resulting in the mapping shown in the figure. Note that the subtrees of the root node do not have the same amount of work. Thus, during the parallel multifrontal algorithm, processors zero through three will have to wait for processors four through seven to finish their work, before they can perform an extend-add operation and proceed to factor the top node. This idling puts an upper bound on the efficiency of this algorithm. We can compute this upper bound on the achievable efficiency due to load imbalance in the following way. The time required to factor a subtree of the elimination tree is equal to the time to factor the root plus the maximum of the time required to factor each of the two subtrees rooted at this root. By applying the above rule recursively we can compute the time required to perform the Cholesky factorization. Assume that the communication overhead is zero, and that each processor can perform an operation in a time unit, the time to factor each subtree of the elimination tree in Figure 2 is shown on the right of each node. For instance, node 9–11 requires $773 - 254 - 217 = 302$ operations, and since the computation is distributed over processors zero and one, it takes 151 time units. Now its subtree rooted at node 4–5 requires 254 time units, while its subtree rooted at node 8 requires 217 time units. Thus, this particular subtree is factored in $151 + \max\{254, 217\} = 405$ time units. The overall efficiency achievable by the above subtree-to-subcube mapping is

$$\frac{4302}{8 \times 812} = 0.66$$

which is significantly less than one. Furthermore, the final efficiency is even lower due to communication overheads.

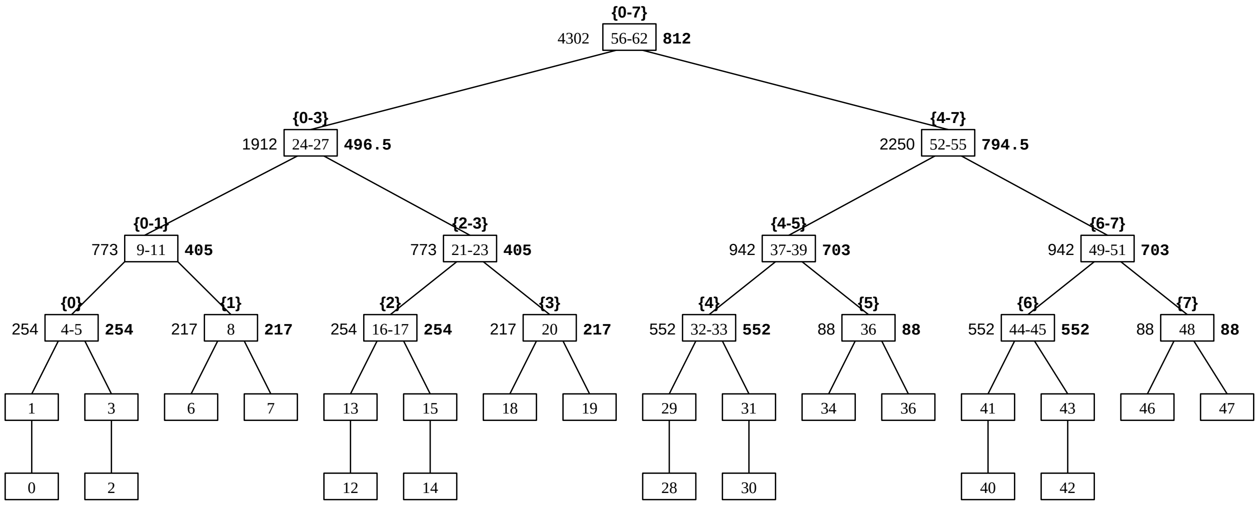


Figure 2: The supernodal elimination tree of a factorization problem and its mapping to eight processors via subtree-to-subcube mapping. Each node (*i.e.*, supernode) is labeled by the range of nodes belonging to it. The number on the left of each node is the number of operations required to factor the tree rooted at this node, the numbers above each node denotes the set of processors that this subtree is assigned to using subtree-to-subcube allocation, and the number on the right of each node is the time-units required to factor the subtree in parallel.

This example illustrates another difficulty associated with direct factorization. Even though both subtrees rooted at node 56–62 have 28 nodes, they require different amount of computation. Thus, balancing the computation cannot be done during the ordering phase by simply carefully selecting separators that split the graph into two roughly equal parts. The amount of load imbalance among different parts of the elimination tree can be significantly worse for general sparse matrices, for which it is not even possible to find good separators that can split the graph into two roughly equal parts. Table 1 shows the load imbalance at the top level of the elimination tree for some matrices from the Boeing-Harwell matrix set. These matrices were ordered using the spectral nested dissection [29, 30, 19]. Note that for all matrices the load imbalance in terms of operation count is substantially higher than the relative difference in the number of nodes in the left and right subtrees. Also, the upper bound on the efficiency shown in this table is due only to the the top level subtrees. Since subtree-to-subcube mapping is recursively applied in each subcube, the overall load imbalance will be higher, because it adds up as we go down in the tree.

For elimination trees of general sparse matrices, the load imbalance can be usually decreased by performing some simple elimination tree reorderings described in [19]. However, these techniques have two serious limitations. First, they increase the fill-in as they try to balance the elimination tree by adding extra dependencies. Thus, the total time required to perform the factorization increases. Second, these techniques are local heuristics that try to minimize the load imbalance at a given level of the tree. However, very often such

		Left Subtree		Right Subtree		
Name	Separator Size	Nodes	Remaining Work	Nodes	Remaining Work	Efficiency Bound
BCSSTK29	180	6912	45%	6695	55%	0.90
BCSSTK30	222	14946	59%	13745	41%	0.85
BCSSTK31	492	16728	40%	18332	60%	0.83
BCSSTK32	513	21713	45%	22364	55%	0.90

Table 1: Ordering and load imbalance statistics for some matrices from the Boeing-Harwell set. The matrices have been reordered using spectral nested dissection. For each matrix, the size of the top separator is shown, and for each subtree the number of nodes, and the percent of the remaining work is shown. Also, the last column shows the maximum achievable efficiency, if any subsequent levels of the elimination tree were perfectly balanced, or if only two processors were used for the factorization.

local improvements do not result in improving the overall load imbalance. For example, for a wide variety of problems from the Boeing-Harwell matrix set and linear programming (LP) matrices from NETLIB [5], even after applying the tree balancing heuristics, the efficiency bound due to load imbalance is still around 80% to 60% [13, 20, 19]. If the increased fill-in is taken into account, then the maximum achievable efficiency is even lower than that.

In the rest of this section we present a modification to the algorithm presented in Section 3 that uses a different scheme for mapping the elimination tree onto the processors. This modified mapping scheme significantly reduces the load imbalance.

4.1 Subforest-To-Subcube Mapping Scheme

In our new elimination tree mapping scheme, we assign many subtrees (subforest) 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 3. 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 3(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 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 efficiency due to load imbalance of this new assignment 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. Since it assigns subforests of the elimination tree to processor subcubes, we will refer to it as *subforest-to-subcube* mapping scheme. The general mapping algorithm is outlined in Program 4.1.

The tree partitioning algorithm uses a set Q that contains the unassigned nodes of the elimination tree. The algorithm inserts the root of the elimination tree into Q, and then it calls the routine *Elpart* that

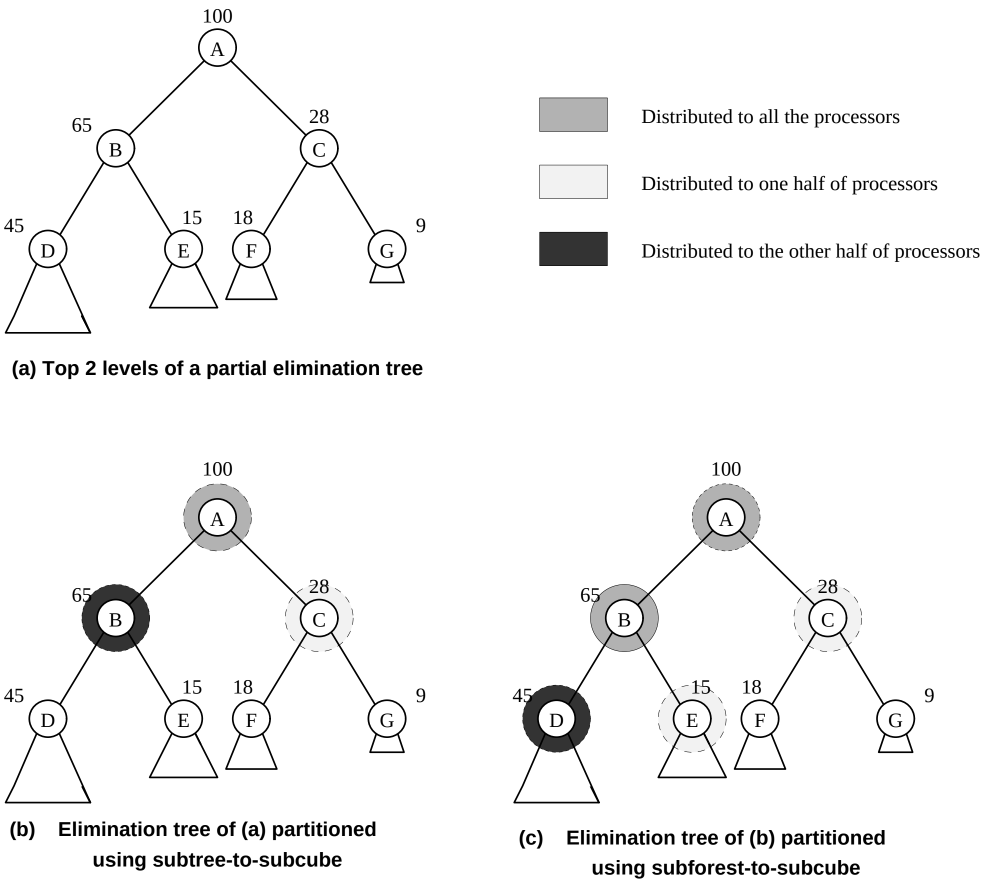


Figure 3: The top two levels of an elimination tree is shown in (a). The subtree-to-subcube mapping is shown in (b), the subforest-to-subcube mapping is shown in (c).

recursively partitions the elimination tree. *Elpart* partitions Q 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 Q (*i.e.*, *node = select(Q)*) is assigned to all the p processors, *node* is deleted from Q, and the children of *node* are inserted into the Q. 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 Q There are two different ways[1] of defining the procedure *select(Q)*.

- 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 Q we may get a good partitioning of Q 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.

[1] 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 elimination tree T, among p processors */
    Q = {}
    Add root(T) into Q
    Elpart(Q, T, p)
End Partition

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

Program 4.1: The subforest-to-subcube partitioning algorithm.

However, if the subtrees attached to this node are not large, then this may not lead to a good partitioning of Q in later steps. In particular, if the root of the subtree having most of the remaining work, requires little computation (*e.g.*, single 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 6.

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 Q In each step, the partitioning algorithm checks to see if it can split the set Q 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 [28]. 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 that 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 $n^2/\sqrt{p}$ [22]. 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 6, we used $\epsilon = 0.05$.

4.2 Other Modifications

In this section we describe the necessary modifications of the algorithm presented in Section 3 to accommodate this new mapping scheme.

Initially each processor factors the subtrees of the elimination tree assigned to itself. This represents local computation and requires no communication just like the earlier algorithm.

However, since each processor is assigned more than one subtree of the elimination tree, at the end of this local computation, its stack will contain one update matrix for each tree assigned to it. At this point, it needs to perform a distributed extend-add operation with its neighbor processor at the first level of the virtual binary tree. During this step, each processor splits the update matrices, and sends the part that is not local, to the other processor. This is similar to the parallel extend-add operation required by the algorithm described in Section 3, except that more than one update matrix is split and sent. Note, that these pieces from different update matrices can all be packed in a single message, as the communication happens with only one other processor. Furthermore, as shown in Appendix D, the amount of data being transmitted in each parallel extend-add step is no more than it is in the earlier algorithm [13]. The reason is that even though more update matrices are being transmitted, these update matrices correspond to nodes that are deeper in the elimination tree and the size of these matrices is much smaller.

Now, after pairs of processors have performed the extend-add operation, they cooperate to factor the nodes of the elimination tree assigned to them. The nodes are eliminated in a postorder order. Next, groups of four processors exchange the update matrices that are stored in their stack to perform the extend-add operation for the next level. This process continues until all the nodes have been factored. The new parallel multifrontal algorithm is outlined in Appendix C.

5 Improving Performance

We have added a number of modifications to the algorithm described in Section 4 that greatly improve its performance. In the following sections we briefly describe these modifications. For a more detailed description of these enhancements the reader should refer to [21].

5.1 Block Cyclic Mapping

As discussed in Appendix D, for the factorization of a supernode, we use the pipelined variant of the grid-based dense Cholesky algorithm [22]. In this algorithm, successive rows of the frontal matrix are factored one after the other, and the communication and computation proceeds in a pipelined fashion.

Even though this scheme is simple, it has two major limitations. Since the rows and columns of a frontal matrix among the processor grid in a cyclic fashion, information for only one row is transmitted at any given time. Hence, on architectures in which the message startup time is relatively high compared to the transfer time, the communication overhead is dominated by the startup time. For example, consider a $\sqrt{q} \times \sqrt{q}$ processor grid, and a k-node supernode that has a frontal matrix of size $m \times m$. While performing k elimination steps on an $m \times m$ frontal matrix, on average, a message of size $(2m - k)/(2\sqrt{q}))$ needs to be send in each step along each direction of the grid. If the message startup time is 100 times higher than the per word transfer time, then for $q = 256$, as long as $2m - k < 3200$ the startup time will dominate the data transfer time. Note, that the above translates to $m > 1600$. For most sparse matrices, the size of the frontal matrices tends to be much less than 1600.

The second limitation of the cyclic mapping has to do with the implementation efficiency of the computation phase of the factorization. Since, at each step, only one row is eliminated, the factorization algorithm must perform a rank-one update. On systems with BLAS level routines, this can be done using either level one BLAS (DAXPY), or level two BLAS (DGER, DGEMV). On most microprocessors, including high performance RISC processors such as the Dec Alpha AXP, the peak performance achievable by these primitives is usually significantly less than that achieved by level three BLAS primitives, such as matrix-matrix multiply (DGEMM). The reason is that for level one and level two BLAS routines, the amount of computation is of the same order as the amount of data movement between CPU and memory. In contrast, for level three BLAS operations, the amount of computation is much higher than the amount of data required from memory. Hence, level three BLAS operations can better exploit the multiple functional units, and deep pipelines available in these processors.

However, by distributing the frontal matrices using a block cyclic mapping [22], we are able to eliminate both of the above limitations and greatly improve the performance of our algorithm. In the block cyclic mapping, the rows and columns of the matrix are divided into groups, each of size b, and these groups are assigned to the processors in a cyclic fashion. As a result, diagonal processors now store blocks of b consecutive pivots. Instead of performing a single elimination step, they now perform b elimination steps, and send data corresponding to b rows in a single message. Note that the overall volume of data transferred remains the same. For sufficiently large values of b, the startup time becomes a small fraction of the data transmission time. This result in a significant improvements on architectures with high startup time. In each phase now, each processor receives b rows and columns and has to perform a rank-b update on the not yet factored part of its frontal matrix. The rank-b update can now be implemented using matrix-matrix multiply, leading to higher computational rate.

There are a number of design issues in selecting the proper value for b. Clearly, the block size should be large enough so that the rank-b update achieves high performance and the startup time becomes a small fraction of the data transfer time. On the other hand a very large value of b leads to a number of problems. First, processors storing the current set of b rows to be eliminated, have to construct the rank-b update by performing b rank-1 updates. If b is large, performing these rank-1 updates takes considerable amount of time, and stalls the pipeline. Also, a large value of b leads to load imbalances on the number of elements of the frontal matrix assigned to each processor, because there are fewer blocks to distribute in a cyclic fashion. Note that this load imbalance within the dense factorization phase is different from the load imbalance associated with distributing the elimination tree among the processors described in Section 4.

A number of other design issues involved in using block cyclic mapping and ways to further improve the performance are described in [21].

5.2 Pipelined Cholesky Factorization

In the parallel portion of our multifrontal algorithm, each frontal matrix is factored using a grid based pipelined Cholesky factorization algorithm. This pipelined algorithms works as follows [22]. Assume that the processor grid stores the upper triangular part of the frontal matrix, and that the processor grid is square. The diagonal processor that stores the current pivot, divides the elements of the pivot row it stores by the pivot and sends the pivot to its neighbor on the right, and the scaled pivot row to its neighbor down. Each processor upon receiving the pivot, scales its part of the pivot row and sends the pivot to the right and its scaled pivot row down. When a diagonal processor receives a scaled pivot row from its up processor, it forwards this down along its column, and also to its right neighbor. Every other processor, upon receiving a scaled pivot row either from the left or from the top, stores it locally and then forwards it to the processor at the opposite end. For simplicity, assume that data is taken out from the pipeline by the processor who initiated the transmission. Each processor performs a rank-1 update of its local part of the frontal matrix as soon as it receives the necessary elements from the top and the left. The processor storing the next pivot element starts eliminating the next row as soon as it has finished computation for the previous iteration. The process continues until all the rows have been eliminated.

Even though this algorithm is correct, and its asymptotic performance is as described in Appendix D, it requires buffers for storing messages that have arrived and cannot yet be processed. This is because certain processors receive the two sets of data they need to perform a rank-1 update at different times. Consider for example a 4×4 processor grid, and assume that processor $(0,0)$ has the first pivot. Even though processor $(1,0)$ receives data from the top almost right away, the data from the left must come from processor $(0,1)$ via $(1,1), (1,2), \text{and}, (1,3)$. Now if the processor $(0,0)$, also had the second pivot (due to a greater than one block size), then the message buffer on processor $(1,0)$ might contain the message from processor $(0,0)$ corresponding to the second pivot, before the message from $(0,1)$ corresponding to the first pivot had arrived. The source of this problem is that the processors along the row act as the sources for both type of messages (those circulating along the rows and those circulating along the columns). When a similar algorithm is used for Gaussian elimination, the problem doesn't arise because data start from a column and a row of processors, and messages from these rows and columns arrive at each processor at roughly the same time [22].

On machines with very high bandwidth, the overhead involved in managing buffers significantly reduces the percentage of the obtainable bandwidth. This effect is even more pronounced for small messages. For this reason, we decided to implement our algorithm with only a single message buffer per neighbor. As mentioned in Section 6, this communication protocol enable us to utilize most of the theoretical bandwidth on a Cray T3D.

However, under the restrictions of limited message buffer space, the above Cholesky algorithm spends a significant amount of time idling. This is due to the following requirement imposed by the single communication buffer requirement. Consider processors $P_{k,k}$, and $P_{k+1,k}$. Before processor $P_{k,k}$ can start the $(i+1)$th iteration, it must wait until processor $P_{k+1,k}$ has started performing the rank-1 update for the ith iteration (so that processor $P_{k,k}$ can go ahead and reuse the buffers for iteration $(i+1)$). However, since processor $P_{k+1,k}$ receives data from the left much later than it does from the top, processor $P_{k,k}$ must wait until this latter data transmission has taken place. Essentially during this time processor $P_{k,k}$ sits idle. Because, it sits idle, the $i+1$ iteration will start late, and data will arrive at processor $P_{k+1,k}$ even later. Thus, at each step, certain processors spend certain amount of time being idle. This time is proportional to the time it takes for a message to travel along an entire row of processors, which increases substantially with the number of processors.

To solve this problem, we slightly modified the communication pattern of the pipelined Cholesky algorithm as follows. As soon as a processor that contains elements of the pivot row has finished scaling it, it sends it both down and also to the transposed processor along the main diagonal. This latter processor upon receiving the scaled row, starts moving it along the rows. Now the diagonal processors do not anymore forward the data they receive from the top to the right. The reason for this modification is to mimic the behavior of the algorithm that performs Gaussian elimination. On architectures with cut-through routing, the overhead of this communication step is comparable to that of a nearest neighbor transmission for sufficiently large messages. Furthermore, because these *transposed* messages are initiated at different times cause little or no contention. As a result, each diagonal processor now will have to sit idle for a very small amount of time [21].

6 Experimental Results

We implemented our new parallel sparse multifrontal algorithm on a 256-processors Cray T3D parallel computer. Each processor on the T3D is a 150Mhz Dec Alpha chip, with peak performance of 150MFlops for 64-bit operations (double precision). The processors are interconnected via a three dimensional torus network that has a peak unidirectional bandwidth of 150Bytes 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 70Mbytes per second. This is significantly higher than the 35MBytes 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 (BCSSTK40, 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. In all of our experiments, we used spectral nested dissection [29, 30] to order the matrices.

The performance obtained by our multifrontal 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 (it does not include the operations involved in extend-add). Some of these problems could not be run on 32 processors due to memory constraints (in our T3D, each processor had only 2Mwords of memory).

Figure 4 graphically represents the data shown in Table 2. Figure 4(a) shows the overall performance obtained versus the number of processors, and is similar in nature to a speedup curve. Figure 4(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	$\|A\|$	$\|L\|$	Operation Count	32	64	128	256
MAROS-R7	3136	330472	1345241	720M	0.83	1.41	2.18	3.08
BCSSTK30	28924	1007284	5796797	2400M		1.48	2.45	3.59
BCSSTK31	35588	572914	6415883	3100M		1.47	2.42	3.87
BCSSTK32	44609	985046	8582414	4200M		1.51	2.63	4.12
BCSSTK33	8738	291583	2295377	1000M	0.78	1.23	1.92	2.86
COPTER2	55476	352238	12681357	9200M		1.92	3.17	5.51
CUBE35	42875	124950	11427033	10300M		2.23	3.75	6.06

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, number of operations required to factor the nodes, and the performance in gigaflops for different number of processors.

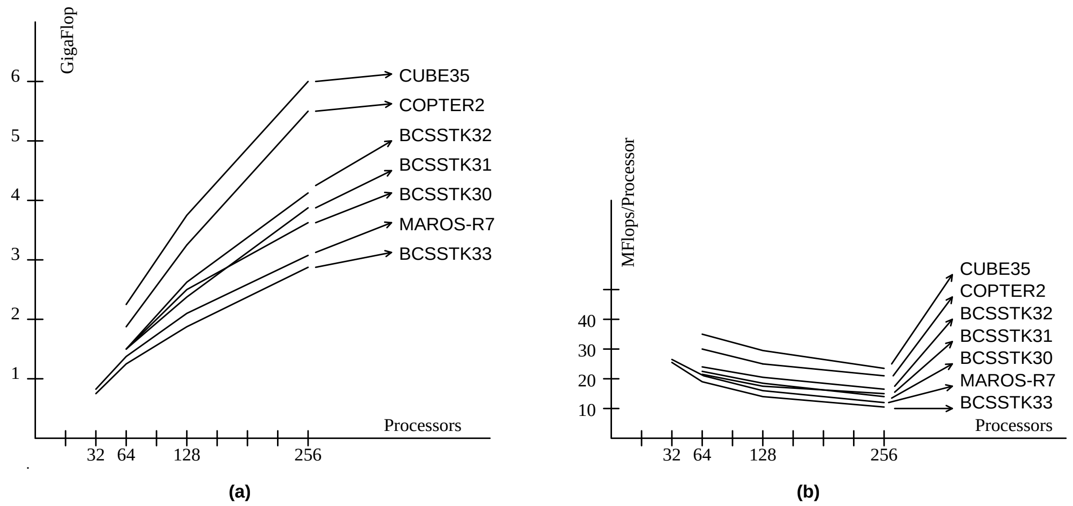


Figure 4: Plot of the performance of the parallel sparse multifrontal algorithm for various problems on Cray T3D. (a) Total Gigaflops obtained; (b) Megaflops per processor.

The highest performance of 6GFlops was obtained for CUBE35, which is a regular three-dimensional problem. Nearly as high performance (5.51GFlops) 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 to the other problems shown in Table 2, we see that even on smaller problems, our algorithm performs quite well. For BCSSTK33, it was able to achieve 2.86GFlops on 256 processors, while for BCSSTK30, it achieved 3.59GFlops.

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 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 Appendix D, since the overhead of distributed extend-add is $O((n \log p)/p)$. Since the figure 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 5.1, 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) However, 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.

	BCSSTK31			CUBE35		
		Distributed Computation			Distributed Computation	
p	Local Computation	Factorization	Extend-Add	Local Computation	Factorization	Extend-Add
64	0.17	1.34	0.58	0.15	3.74	0.71
128	0.06	0.90	0.32	0.06	2.25	0.43
256	0.02	0.61	0.18	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 1.18GFlops when $\epsilon = 0.4$, and 1.37GFlops when $\epsilon = 0.0001$. In either case, the performance is worse than the 2.42GFlops 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.

7 Conclusion

Experimental results clearly show that our new scheme is capable of using a large number of processor efficiently. On a single processor of a state of the art vector supercomputer such as Cray C90, sparse Cholesky factorization can be done at the rate of roughly 500MFlops for the larger problems studied in Section 6. Even a 32-processor Cray T3D clearly outperforms a single node C-90 for these problems.

Our algorithm as presented (and implemented) works for Cholesky factorization of symmetric positive definite matrices. With little modifications, it is also applicable to LU factorization of other sparse matrices, as long as no pivoting is required (*e.g.*, sparse matrices arising out of structural engineering problems).

With highly parallel formulation available, the factorization step is no longer the most time consuming step in the solution of sparse systems of equations. Another step that is quite time consuming, and has not been parallelized effectively is that of ordering. In our current research we are investigating ordering algorithms that can be implemented fast on parallel computers [16, 34].

Acknowledgment

We would like to thank the Minnesota Supercomputing Center for providing access to a 64-processor Cray T3D, and Cray Research Inc. for providing access to a 256-processor Cray T3D. Finally we wish to thank Dr. Alex Pothen for his guidance with spectral nested dissection ordering.

References

[1] 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.

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

[3] Kalluri Eswar, Ponnuswamy Sadayappan, and V. Visvanathan. *Supernodal Sparse Cholesky factorization on distributed-memory multiprocessors.* In *International Conference on Parallel Processing*, pages 18–22 (vol. 3), 1993.

[4] 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.

[5] D. M. Gay. *Electronic Mail Distribution of Linear Programming Test Problems. ¡mathematical Programming Society COAL Newsletter*, December 1985.

[6] 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.

[7] 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.

[8] 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.

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

[10] A. George and J. W.-H. Liu. *The evolution of the minimum degree ordering algorithm. SIAM Review*, 31(1):1–19, March 1989.

[11] 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.

[12] John R. Gilbert and Robert Schreiber. *Highly Parallel Sparse Cholesky Factorization. SIAM Journal on Scientific and Statistical Computing*, 13:1151–1172, 1992.

[13] 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 shorter version appears in Supercomputing '94. TR available in *users/kumar/sparse-cholesky.ps* at anonymous FTP site *ftp.cs.umn.edu.*

[14] M. T. Heath, E. Ng, and B. W. Payton. *Parallel Algorithms for Sparse Linear Systems. SIAM Review*, 33(3):420–460, 1991.

[15] 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.

[16] M. T. Heath and P. Raghavan. *A Cartesian nested dissection algorithm.* Technical Report UIUCDCS-R-92-1772, Department of Computer Science, University of Illinois, Urbana, IL 61801, October 1992. to appear in SIMAX.

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

[18] 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.

[19] George Karypis, Anshul Gupta, and Vipin Kumar. *Ordering and load balancing for parallel factorization of sparse matrices.* Technical Report (in preparation), Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994.

[20] George Karypis, Anshul Gupta, and Vipin Kumar. *A Parallel Formulation of Interior Point Algorithms.* In *Supercomputing 94*, 1994. Also available as Technical Report TR 94-20, Department of Computer Science, University of Minnesota, Minneapolis, MN.

[21] George Karypis and Vipin Kumar. *A High Performance Sparse Cholesky Factorization Algorithm For Scalabale Parallel Computers.* Technical Report 94-41, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994.

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

[23] Joseph W. H. Liu. *The Multifrontal Method for Sparse Matrix Solution: Theory and Practice. SIAM Review*, 34(1):82–109, 1992.

[24] Robert F. Lucas. *Solving planar systems of equations on distributed-memory multiprocessors.* PhD thesis, Department of Electrical Engineering, Stanford University, Palo Alto, CA, 1987. Also see *IEEE Transactions on Computer Aided Design*, 6:981–991, 1987.

[25] 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.

[26] 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.

[27] Dianne P. O'Leary and G. W. Stewart. *Assignment and Scheduling in Parallel Matrix Factorization. Linear Algebra and its Applications*, 77:275–299, 1986.

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

[29] A. Pothen and C-J. Fan. *Computing the block triangular form of a sparse matrix. ACM Transactions on Mathematical Software*, 1990.

[30] Alex Pothen, Horst D. Simon, and Kang-Pu Liou. *Partitioning Sparse Matrices With Eigenvectors of Graphs. SIAM J. on Matrix Analysis and Applications*, 11(3):430–452, 1990.

[31] 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.

[32] 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.

[33] P. Raghavan. *Distributed sparse Gaussian elimination and orthogonal factorization.* Technical Report 93-1818, Department of Computer Science, University of Illinois, Urbana, IL, 1993.

[34] P. Raghavan. *Line and plane separators.* Technical Report UIUCDCS-R-93-1794, Department of Computer Science, University of Illinois, Urbana, IL 61801, February 1993.

[35] Edward Rothberg. *Performance of Panel and Block Approaches to Sparse Cholesky Factorization on the iPSC/860 and Paragon Multicomputers.* In *Proceedings of the 1994 Scalable High Performance Computing Conference*, May 1994.

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

[37] P. Sadayappan and Sailesh K. Rao. *Communication Reduction for Distributed Sparse Matrix Factorization on a Processors Mesh.* In *Supercomputing '89 Proceedings*, pages 371–379, 1989.

[38] 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.

[39] Chunguang Sun. *Efficient parallel solutions of large sparse SPD systems on distributed-memory multiprocessors.* Technical report, Department of Computer Science, Cornell University, Ithaca, NY, 1993.

[40] 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.

[41] M. Yannakakis. *Computing the minimum fill-in is NP-complete. SIAM J. Algebraic Discrete Methods*, 2:77–79, 1981.

Appendix A Multifrontal Method

Let A be an $n \times n$ symmetric positive definite matrix and L be its Cholesky factor. Let T be its elimination tree and define $T[i]$ to represent the set of descendants of the node i in the elimination tree T. Consider the jth column of L. Let $i_0, i_1, \ldots, i_r$ be the row subscripts of the nonzeros in $L_{\bullet,j}$ with $i_0 = j$ (*i.e.*, column j has r off-diagonal nonzeros).

The *subtree update matrix* at column j, W_j for A is defined as

$$W_j = - \sum_{k \in T[j]-\{j\}} \begin{pmatrix} l_{j,k} \\ l_{i_1,k} \\ \vdots \\ l_{i_r,k} \end{pmatrix} (l_{j,k}, l_{i_1,k}, \ldots, l_{i_r,k}). \tag{1}$$

Note that W_j contains outer product contributions from those previously eliminated columns that are descendants of j in the elimination tree. The jth *frontal matrix* F_j is defined to be

$$F_j = \begin{pmatrix} a_{j,j} & a_{j,i_1} & \cdots & a_{j,i_r} \\ a_{i_1,j} & & & \\ \vdots & & 0 & \\ a_{i_r,j} & & & \end{pmatrix} + W_j. \tag{2}$$

Thus, the first row/column of F_j is formed from $A_{\bullet,j}$ and the subtree update matrix at column j. Having formed the frontal matrix F_j, the algorithm proceeds to perform one step of elimination on F_j that gives the nonzero entries of the factor of $L_{\bullet,j}$. In particular, this elimination can be written in matrix notation as

$$F_j = \begin{pmatrix} l_{j,j} & 0 \\ l_{i_1,j} & \\ \vdots & I \\ l_{i_r,j} & \end{pmatrix} \begin{pmatrix} 1 & 0 \\ & \\ 0 & U_j \end{pmatrix} \begin{pmatrix} l_{j,j} & l_{i_1,j} & \cdots & l_{i_r,j} \\ & & & \\ 0 & & I & \end{pmatrix}. \tag{3}$$

where $l_{\bullet,j}$ are the nonzero elements of the Cholesky factor of column j. The matrix U_j is called *update matrix* for column j and is formed as part of the elimination step.

In practice, W_j is never computed using Equation 1, but is constructed from the update matrices as follows. Let $c_1, \ldots, c_s$ be the children of j in the elimination tree, then

$$F_j = \begin{pmatrix} a_{j,j} & a_{j,i_1} & \cdots & a_{j,i_r} \\ a_{i_1,j} & & & \\ \vdots & & 0 & \\ a_{i_r,j} & & & \end{pmatrix} \mathbin{\updownarrow\!\!\!\!+} U_{c_1} \mathbin{\updownarrow\!\!\!\!+} \cdots \mathbin{\updownarrow\!\!\!\!+} U_{c_s} \tag{4}$$

where $\mathbin{\updownarrow\!\!\!\!+}$ is called the *extend-add operator*, and is a generalized matrix addition. The extend-add operation is illustrated in the following example. Consider the following two update matrices of A:

$$R = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad S = \begin{pmatrix} p & q & r \\ s & t & x \\ y & z & w \end{pmatrix}$$

where $\{2, 5\}$ is the index set of R (*i.e.*, the first row/column of R corresponds to the second row/column of M, and the second row/column of R corresponds to the fifth row/column of M), and $\{1, 3, 5\}$ is the index set of S. Then

$$R \mathbin{\updownarrow\!\!\!\!+} S = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & a & 0 & b \\ 0 & 0 & 0 & 0 \\ 0 & c & 0 & d \end{pmatrix} + \begin{pmatrix} p & 0 & q & r \\ 0 & 0 & 0 & 0 \\ s & 0 & t & x \\ y & 0 & z & w \end{pmatrix} = \begin{pmatrix} p & 0 & q & r \\ 0 & a & 0 & b \\ s & 0 & t & x \\ y & c & z & d+w \end{pmatrix}.$$

Note that the submatrix $U_{c_1} \mathbin{\updownarrow\!\!\!\!+} \cdots \mathbin{\updownarrow\!\!\!\!+} U_{c_s}$ may have fewer rows/columns than W_j, but if it is properly extended by the index set of F_j, it becomes the subtree update matrix W_j.

The process of forming F_j from the nonzero structure elements of column j of A and the updates matrices is called *frontal matrix assembly operation.* Thus, in the multifrontal method, the elimination of each column of M involves the assembly of a frontal matrix and one step of elimination.

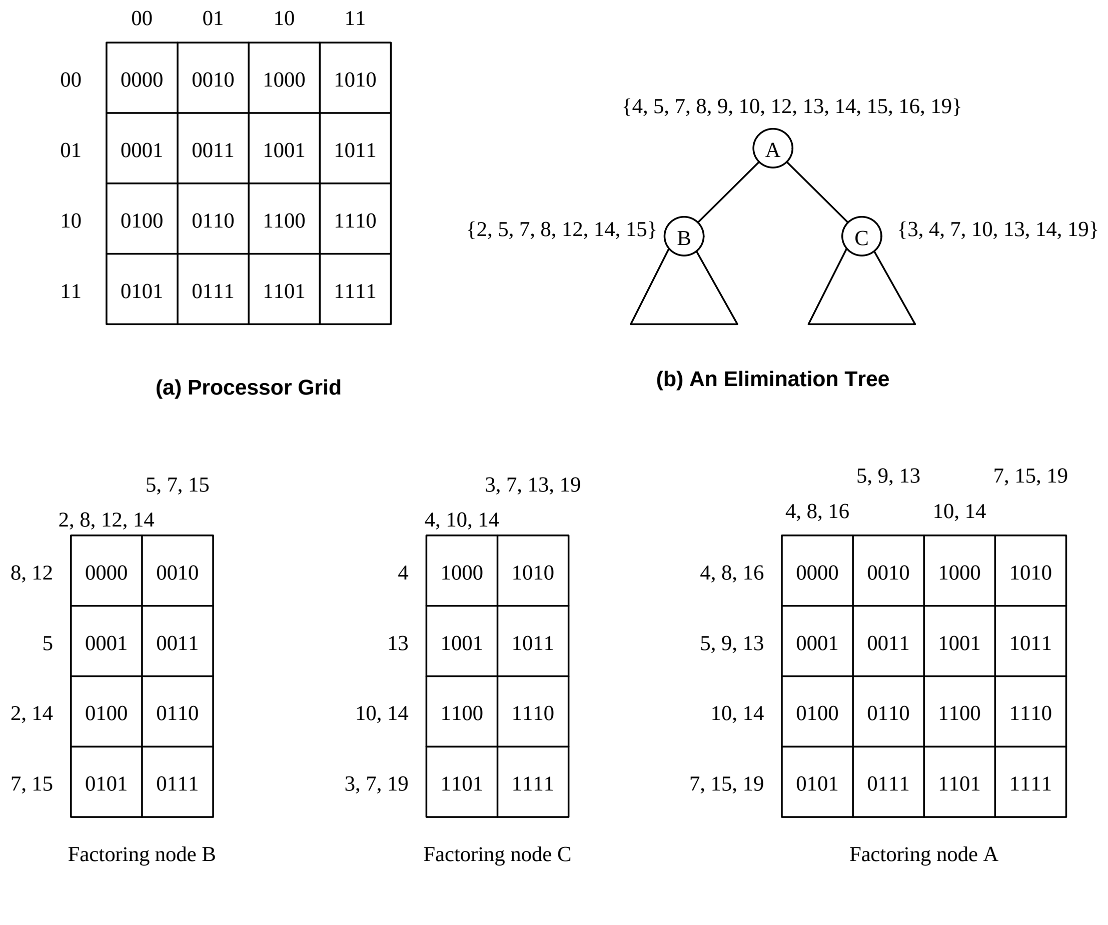
The supernodal multifrontal algorithm proceeds similarly to the multifrontal algorithm, but when a supernode of the elimination tree gets eliminated, this corresponds to multiple elimination steps on the same frontal matrix. The number of elimination steps is equal to the number of nodes in the supernode.

Appendix B Data Distribution

Figure 5(a) shows a 4×4 grid of processors embedded in the hypercube. This embedding uses the shuffle mapping. In particular grid position with coordinate (y, x) is mapped onto processor whose address is made by interleaving the bits in the binary representation of x and y. For instance, the grid position with binary coordinates (ab, cd) is mapped onto processor whose address in binary is $cadb$, for example grid $(10, 11)$ is mapped onto 1110. Note that when we split this 4×4 processor grid into two 4×2 subgrids, these subgrids correspond to distinct subcubes of the original hypercube. This property is maintained in subsequent splits, for instance each 2×2 grid of a 4×2 grid is again a subcube. This is important, because the algorithm uses subtree-to-subcube partitioning scheme, and by using shuffling mapping, each subcube is simply half of the processor grid.

Consider next, the elimination tree shown in Figure 5(b). In this figure only the top two levels are shown, and at each node i, the nonzero elements of L_i for this particular node is also shown. Using the subtree-to-subcube partitioning scheme, the elimination tree is partitioned among the 16-processors as follows. Node A is assigned to all the 16 processors, node B is assigned to half the processors, while node C is assigned to the other half of the processors.

In Figure 5(c), we show how the frontal matrices corresponding to nodes B, C, and A will be distributed among the processors. Consider node B. The frontal matrix F_B is a 7×7 triangular matrix, that corresponds to L_B. The distribution of the rows and columns of this frontal matrix on the 4×2 processor grid is shown in part (c). Row i of F_B is mapped on row $i \oplus 2^2$ of the processor grid, while column j of F_B is mapped on column $j \oplus 2^1$ of the processor grid. In this paper $\oplus$ is used to denote bitwise exclusive-or.



(c) Distributed Factorization and Extend-Add

Figure 5: Mapping frontal matrices to the processor grid.

An alternate way of describing this mapping is to consider the coordinates of the processors in each processor grid. Each processor in the 4×2 grid has an (y, x) coordinate such that $0 \le y < 4$, and $0 \le x < 2$. Note that this coordinate is local with respect to the current grid, and is not the same with the grid coordinates of the processor in the entire 4×4 grid. For the rest of the paper, when we discuss grid coordinates we will always assume that they are local, unless specified otherwise. Row i of F_B will be mapped on the processors $(y, *)$ such that the two least significant bits of i is equal to y. Similarly, column j of F_B will be mapped on the processors $(*, x)$, such that the one least significant bit of j is equal to x.

In general, in a $2^k \times 2^l$ processor grid, $f_{i,j}$ is mapped onto the processor with grid coordinates $(i \oplus 2^k, j \oplus 2^l)$. Or looking it from the processor's point of view, processor (y, x) will get all elements $f_{i,j}$ such that the k least significant bits of i are equal to y, and the l least significant bits of j are equal to x. ($i \odot y = y$, and $j \odot x = x$, where $\odot$ is used to denote bitwise logical and).

The frontal matrix for node C is mapped on the other 4×2 grid in a similar fashion. Note that the rows of both F_B and F_C are mapped on processors along the same row of the 4×4 grid. This is because, both grids have the same number of rows. Also, because the final grid also has 4 rows, the rows of F_A that are similar to either rows of F_B and F_C, are mapped on the same row of processors. For example row 7 of both F_A, F_B, and F_C is mapped on the fourth row of the three grids in question. This is important, because when the processors need to perform the extend-add operation $U_B \mathbin{+\!\!\!\!\updownarrow} U_C$, no data movement between rows of the grid is required.

However, data needs to be communicated along the columns of the grid. This is because, the grids that are storing the update matrices have fewer columns than the grid of processors that is going to factor F_A.

Data needs to be communicated so that the columns of the update matrices U_B and U_C matches that of the frontal matrix F_A. Each processor can easily determine which columns of the update matrix it already stores needs to keep and which needs to send away. It keeps the columns whose 2 least significant bits match its x coordinate in the 4×4 grid, and it sends the other away. Let $u_{i,j}$ be a rejected element. This element needs to be send to processor along the same row in the processor grid, but whose x coordinate is $j \oplus 2^2$. However, since element $u_{i,j}$ resides in this processor during the factorization involving the 4×2 grid, then $j \oplus 2^1$ must be the x coordinate of this processor. Therefore, the rejected columns need to be send to the processor along the same row whose x grid coordinate differs in the most significant bit. Thus, during this extend-add operation, processors that are neighbors in the hypercube need to exchange data.

Appendix C New Parallel Multifrontal Algorithm

porder[1..k] /* The nodes of the elimination tree at each processor numbered in a level-postorder */
lvl = $\log p$
for (j=1; j$\leq$ k; j++) {
    i = porder[j]
    if (level[i] != lvl) {
        Split_Stacks(lvl, level[i])
        lvl = level[i]
    }
    Let F_i be the frontal matrix for node i
    Let $c_1, c_2, \ldots, c_s$ be the children of i in the elimination tree
    $F_i = F_i \ddagger U_{c_1} \ddagger U_{c_2} \ddagger \cdots \ddagger U_{c_s}$
    factor(F_i) /* Dense factorization using $2^{\log p - lvl}$ processors */
    push(U_i)
}

The parallel multifrontal algorithm using the subforest-to-subcube partitioning scheme. The function *Split_Stacks*, performs $lvl - level[i]$ parallel extend-add operations. In each of these extend-adds each processor splits its local stack and sends data to the corresponding processors of the other group of processors. Note that when $lvl = \log p$, the function *factor* performs the factorization of F_i locally.

Appendix D Analysis

In this section we analyze the performance of the parallel multifrontal algorithm described in Section 4. In our new parallel multifrontal algorithm and that of Gupta and Kumar [13], there are two types of overheads due to the parallelization: (a) load imbalances due to the work partitioning; (b) communication overhead due to the parallel extend-add operation and due to the factorization of the frontal matrices.

As our experiments show in Section 6, the subforest-to-subcube mapping used in our new scheme essentially eliminates the load imbalance. In contrast, for the subtree-to-subcube scheme used in [13], the overhead due to load imbalance is quite high.

Now the question is whether the subforest-to-subcube mapping used in our new scheme results in higher communication overhead during the extend-add and the dense factorization phase. This analysis is difficult due to the heuristic nature of our new mapping scheme. To keep the analysis simple, we present analysis for regular two-dimensional grids, in which the number of subtrees mapped onto each subcube is four. The analysis also holds for any small constant number of subtrees. Our experiments have shown that the number of subtrees mapped onto each subcube is indeed small.

D.1 Communication Overhead

Consider a $\sqrt{n} \times \sqrt{n}$ regular finite difference grid, and a p-processor hypercube-connected parallel computer. To simplify the analysis, we assume that the grid has been ordered using a nested dissection algorithm, that selects cross-shaped separators [11]. For an n node square grid, this scheme selects a separator of size

$2\sqrt{n}-1 \approx 2\sqrt{n}$ that partitions the grid into four subgrids each of size $(\sqrt{n}-1)/2 \times (\sqrt{n}-1)/2 \approx \sqrt{n}/2 \times \sqrt{n}/2$. Each of these subgrids is recursively dissected in a similar fashion. The resulting supernodal elimination tree, is a quadtree. If the root of the elimination tree is at level zero, then a node at level l corresponds to a grid of size $\sqrt{n}/2^l \times \sqrt{n}/2^l$, and the separator of such a grid is of size $2\sqrt{n}/4^l$. Also, it is proven in [11, 13], that the size of the update matrix of a node at level l is bounded by $2kn/4^l$, where $k = 341/2$.

We assume that the subforest-to-subcube partitioning scheme partitions the elimination tree in the following way. It assigns all the nodes in the first two levels (the zeroth and first level) to all the processors. It then, splits the nodes at the second level into four equal parts and assigns each a quarter of the processors. The children of the nodes of each of these four groups are split into four equal parts and each is assigned to a quarter of the processors of each quarter. This processes continues, until the nodes at level $\log p$ have been assigned to individual processors. Figure 6 shows this type of subforest-to-subcube partitioning scheme. This scheme assigns to each subcube of processors, four nodes of the same level of the elimination tree. In particular, for $i > 0$, a subcube of size $\sqrt{p}/2^i \times \sqrt{p}/2^i$ is assigned four nodes of level $i+1$ of the tree. Thus, each subcube is assigned a forest consisting of four different trees.

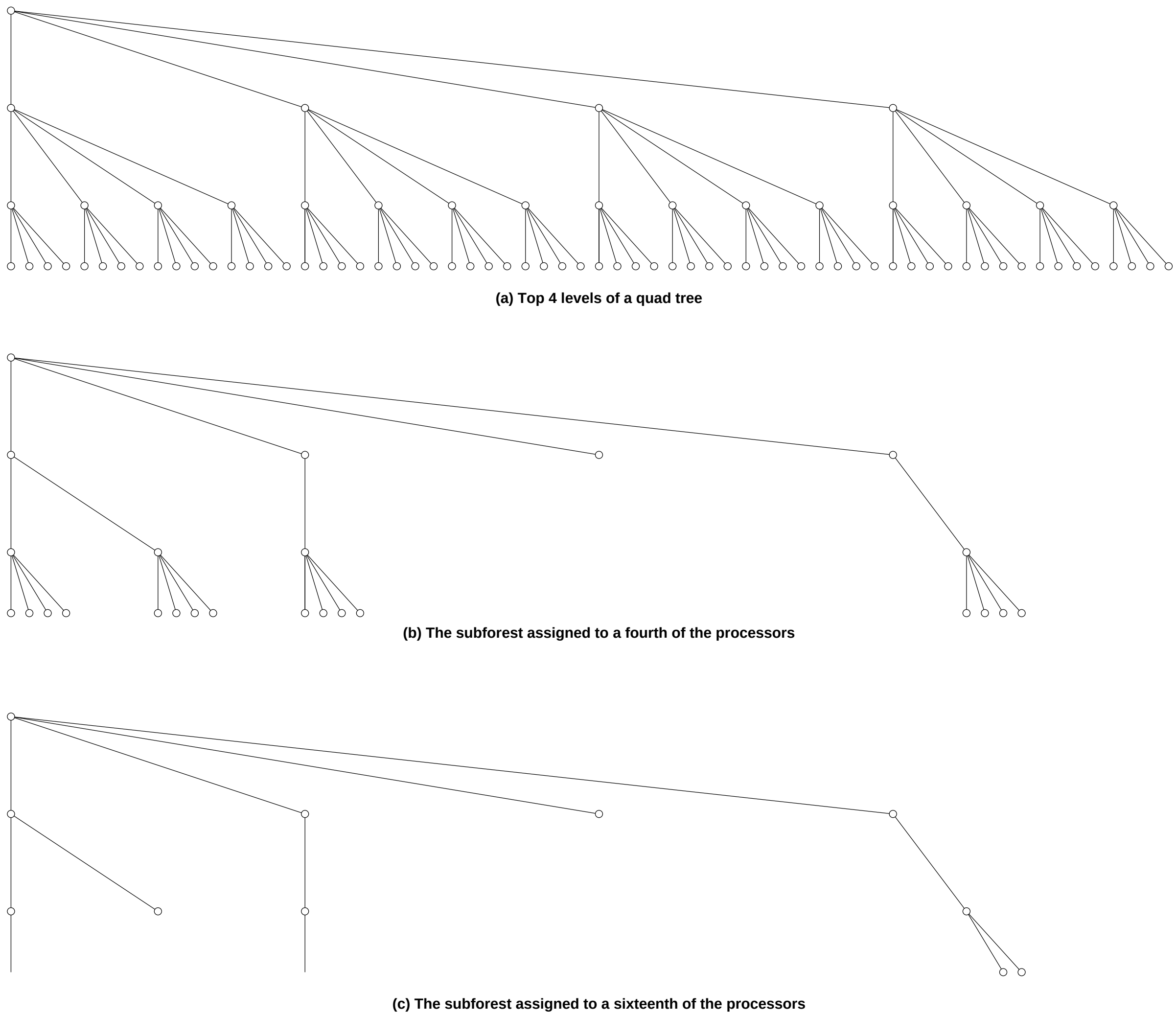


Figure 6: A quadtree and the subforest-to-subcube allocation scheme. (a) Shows the top four levels of a quadtree. (b) Shows the subforest assigned to a quarter of the processors. (c) Shows the subforest assigned to a quarter of a processors of the quarter of part (b).

Consider a subcube of size $\sqrt{p}/2^i \times \sqrt{p}/2^i$. After it has finished factoring the nodes at level $i+1$ of the elimination tree assigned to it, it needs to perform an extend-add with its corresponding subcube, so the new formed subcube of size $\sqrt{p}/2^{i-1} \times \sqrt{p}/2^{i-1}$, can go ahead and factor the nodes of the ith level of the elimination tree. During this extend-add operation, each subcube needs to split the roots of each one of the subtrees being assigned to it. Since, each subcube is assigned four subtrees, each subcube needs to split and send elements from four update matrices. Since each node at level $i+1$ has an update matrix of size $2kn/4^{i+1}$, distributed over $\sqrt{p}/2^i \times \sqrt{p}/2^i$ processors, each processor needs to exchange with the corresponding processor of the other subcube $kn/4p$ elements. Since, each processor has data from four update matrices, the total amount of data being exchanged is kn/p. The are a total of $\log p$ extend-add phases; thus, the total number of data that need to be exchanged is $O(n \log p/p)$. Note that this communication overhead is identical to that required by the subtree-to-subcube partitioning scheme [13].

For the dense Cholesky factorization we use the pipeline implementation of the algorithm on a two-dimensional processor grid using checkerboard partitioning. It can be shown [27, 22] that the communication overhead to perform d factorization steps of an $m \times m$ matrix, in a pipelined implementation on a $\sqrt{q} \times \sqrt{q}$ mesh of processors, is $O(dm/q)$. Since, each node of level $i+1$ of the elimination tree is assigned to a grid of $\sqrt{p}/2^i \times \sqrt{p}/2^i$, and the frontal matrix of each node is bounded by $\sqrt{2kn}/2^{i+1} \times \sqrt{2kn}/2^{i+1}$, and we perform $\sqrt{n}/2^{i+1}$ factorization steps, the communication overhead is

$$\frac{n}{2^{i+1}} \times \frac{\sqrt{2kn}}{2^{i+1}} \frac{2^i}{\sqrt{p}} = \frac{2kn}{42^i\sqrt{p}}.$$

Since, each such grid of processor has four such nodes, the communication overhead of each level is $2kn/(2^i\sqrt{p})$. Thus, the communication overhead over the $\log p$ levels is

$$O\left(\frac{n}{\sqrt{p}}\right) \sum_{i=0}^{\log p} \frac{1}{2^i} = O\left(\frac{n}{\sqrt{p}}\right).$$

Therefore, the communication overhead summed over all the processor due to the parallel extend-add and the dense Cholesky factorization is $O(n\sqrt{p})$, which is of the same order as the scheme presented in [13]. Since the overall communication overhead of our new subforest-to-subcube mapping scheme is of the same order as that for the subtree-to-subcube, the isoefficiency function for both schemes is the same. The analysis just presented can be extended similarly to three-dimensional grid problems and to architectures other than hypercube. In particular, the analysis applies directly to architectures such as CM-5, and Cray T3D.