PARALLEL DIRECT METHODS FOR SPARSE LINEAR SYSTEMS

Michael T. Heath

Department of Computer Science and NCSA
University of Illinois
Urbana, Illinois 61801

ABSTRACT

We present an overview of parallel direct methods for solving sparse systems of linear equations, focusing on symmetric positive definite systems. We examine the performance implications of the important differences between dense and sparse systems. Our main emphasis is on parallel implementation of the numerically intensive factorization process, but we also briefly consider the other major components of direct methods, such as parallel ordering.

Introduction

In this paper we present a brief overview of parallel direct methods for solving sparse linear systems. Paradoxically, sparse matrix factorization offers additional opportunities for exploiting parallelism beyond those available with dense matrices, yet it is often more difficult to attain good efficiency in the sparse case. We examine both sides of this paradox: the additional parallelism induced by sparsity, and the difficulty in achieving high efficiency in spite of it.

We focus on Cholesky factorization, primarily because this allows us to discuss parallelism in relative isolation, without the additional complications of pivoting for numerical stability. Most of the lessons learned are also applicable to other matrix factorizations, such as LU and QR. Our main point in the current discussion is to explain how the sparse case differs from the dense case, and examine the performance implications of those differences.

Consider a system of linear equations

$$Ax = b,$$

where A is an $n \times n$ symmetric positive definite (SPD) 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 with positive diagonal elements. Then the solution vector x can be computed by successive forward and back substitutions to solve the triangular systems

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

Cholesky Factorization Algorithm

The algorithm for Cholesky factorization is a variant of Gaussian elimination that takes advantage of symmetry to reduce both work and storage by about half. Like Gaussian elimination, the algorithm consists of a triple nested loop. One of the 3! ways of arranging that loop is shown in Figure 1.

```
for  j = 1, n
    for  k = 1, j − 1
        for  i = j, n
            a_ij = a_ij − a_ik · a_jk        {cmod(j, k)}
    a_jj = √a_jj
    for  k = j + 1, n
        a_kj = a_kj / a_jj                   {cdiv(j)}
```

Figure 1: Serial Cholesky factorization algorithm

We make the following important observations about this algorithm:

- Since A is SPD, the square roots are all of positive numbers, so the algorithm is well defined.
- Pivoting is not required for numerical stability.
- Only the lower triangular portion of A is accessed.
- The factor L is computed in place, overwriting the lower triangle of A.

- Each column j is modified by a multiple of each prior column k. We denote this operation by $cmod(j,k)$.
- If that multiple is zero (i.e., $a_{jk} = 0$), then the innermost loop has no effect and may as well be skipped.
- Elements of A that were initially zero may become nonzero due to *cmod* operations by nonzero elements from previous columns. Such new nonzeros are called *fill*.
- When all modifications to column j are complete, it is scaled by the square root of its diagonal element to produce column j of the factor. We denote this operation by $cdiv(j)$.

For further details on Cholesky factorization, see (Golub and Van Loan, 1989).

Three Forms of Cholesky Factorization

The three choices of index for the outer loop yield markedly different memory access patterns, as illustrated in Figure 2, and these have important performance implications in various architectural settings, such as effective cache utilization, vectorization, parallelization, or out-of-core solutions.

- *Row-Cholesky:* With i in the outer loop, the inner loops solve a triangular system for each new row in terms of the previously computed rows.
- *Column-Cholesky:* With j in the outer loop, the inner loops compute the matrix-vector product that gives the effect of previously computed columns on the column currently being computed.
- *Submatrix-Cholesky:* With k in the outer loop, the inner loops apply the current column as a rank-1 update to the remaining unreduced submatrix.

Although row-oriented algorithms can be effective in some contexts, column-oriented algorithms tend to be much more effective in practice for sparse problems, so we will restrict our attention to the latter.

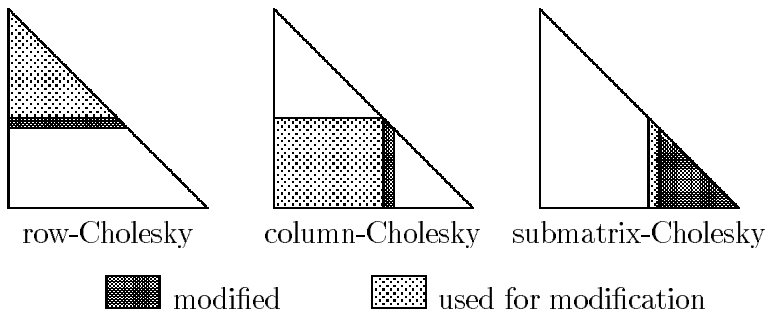

Figure 2: Three forms of Cholesky factorization.

Column-Cholesky is sometimes said to be a "left-looking" algorithm, since at each stage it accesses needed columns to the left of the current column in the matrix. It can also be viewed as a "demand-driven" algorithm, since the inner products that affect a given column are not accumulated until actually needed to modify and complete that column. For this reason, column-Cholesky is called a "delayed-update" algorithm. It is also referred to as a "fan-in" algorithm, since the basic operation is to combine the effects of multiple previous columns on a single target column.

In submatrix-Cholesky, as soon a column has been computed, its effects on all subsequent columns are computed immediately. Thus, submatrix-Cholesky is said to be a "right-looking" algorithm, since at each stage columns to the right of the current column are modified. It can also be viewed as a "data-driven" algorithm, since each new column is used as soon as it is completed to make all modifications to all the subsequent columns it affects. For this reason, submatrix-Cholesky is called an "immediate-update" algorithm. It is also referred to as a "fan-out" algorithm, since the basic operation is for a single column to affect multiple subsequent columns. We will see that these characterizations of the column-Cholesky and submatrix-Cholesky algorithms have important implications for parallel implementations.

For further details on the performance implications of the various rearrangements of Gaussian elimination in various architectural contexts, see, for example, (Dongarra, Gustavson, and Karp, 1984), (George, Heath, and Liu, 1986), (Ortega, 1988), (Robert, 1990),

(Dongarra, Duff, Sorensen, and van der Vorst, 1991), (Demmel, Heath, and van der Vorst, 1993).

Data Dependence

The data dependences in column-oriented Cholesky factorization are shown in Figure 3. All of the modifications to column k must be completed before the column division of column k can take place. Once the column division of column k is completed, then the updating can be done for all of the columns that it affects.

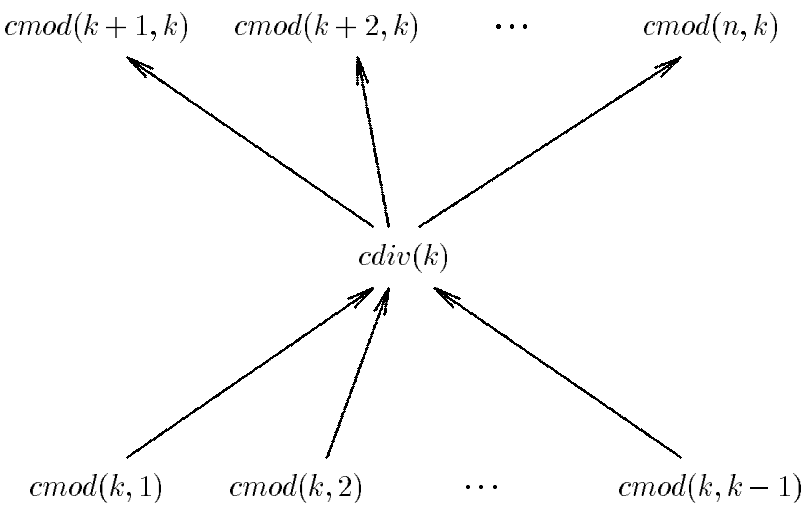

Figure 3: Data dependence in Cholesky factorization.

We make the following important observations:

- The $cmod(k,*)$ operations along the bottom can be done in any order, but they all have the same target column, so the updating must be coordinated to preserve data integrity.
- The $cmod(*,k)$ operations along the top can be done in any order, and they all have different target columns, so the updating can be done simultaneously.

- Performing *cmod*s concurrently is the most important source of parallelism in column-oriented factorization algorithms.
- For a dense matrix, each $cdiv(k)$ depends on the immediately preceding column, so only one *cdiv* can be done at a time.
- Sparse matrices potentially have an additional source of parallelism, since "missing" *cmod*s may permit multiple *cdiv*s to be done simultaneously.

Sparse Matrices

Thus far we have not said what we mean by a "sparse" matrix. A good operational definition is that a matrix is sparse if it contains enough zero entries to be worth taking advantage of them to reduce both the storage and work required in solving a linear system. Ideally, we would like to store and operate on only the nonzero entries of the matrix, but such a policy is not necessarily a clear win in either storage or work. The difficulty is that sparse data structures include more overhead (to store indices as well as numerical values of nonzero matrix entries) than the simple arrays used for dense matrices, and arithmetic operations on the data stored in them usually cannot be performed as rapidly either (due to indirect addressing of operands).

There is therefore a tradeoff in memory requirements between sparse and dense representations and a tradeoff in performance between the algorithms that use them. For this reason, a practical requirement for a family of matrices to be "usefully" sparse is that they have only $O(n)$ nonzero entries, that is, a (small) constant number of nonzeros per row or column, independent of the matrix dimension. For example, most matrices arising from finite difference or finite element discretizations of PDEs satisfy this condition. In addition to the number of nonzeros, their particular locations, or pattern, in the matrix also has a major effect on how well sparsity can be exploited. Sparsity arising from physical problems usually exhibits some systematic pattern that can be exploited effectively, whereas the same number of nonzeros located randomly might offer relatively little advantage.

Many of the combinatorial aspects of sparse direct methods are most easily expressed and understood in graph-theoretic terms. The *graph* $G(A)$ of a symmetric $n \times n$ matrix A is an undirected graph

having n vertices, with an edge between two vertices i and j if the corresponding entry a_{ij} of the matrix is nonzero. An example of a matrix and its corresponding graph are shown in Figure 4.

The structural effect of the factorization process can be characterized by observing that the elimination of a variable adds fill edges to the corresponding graph so that the neighbors of the eliminated vertex become a clique (i.e., a fully connected subgraph). The resulting fill in the Cholesky factor for the previous example is shown in Figure 4. For further details on these and many other aspects of sparse matrix computations, see (George and Liu, 1981) or (Duff, Erisman, and Reid, 1986).

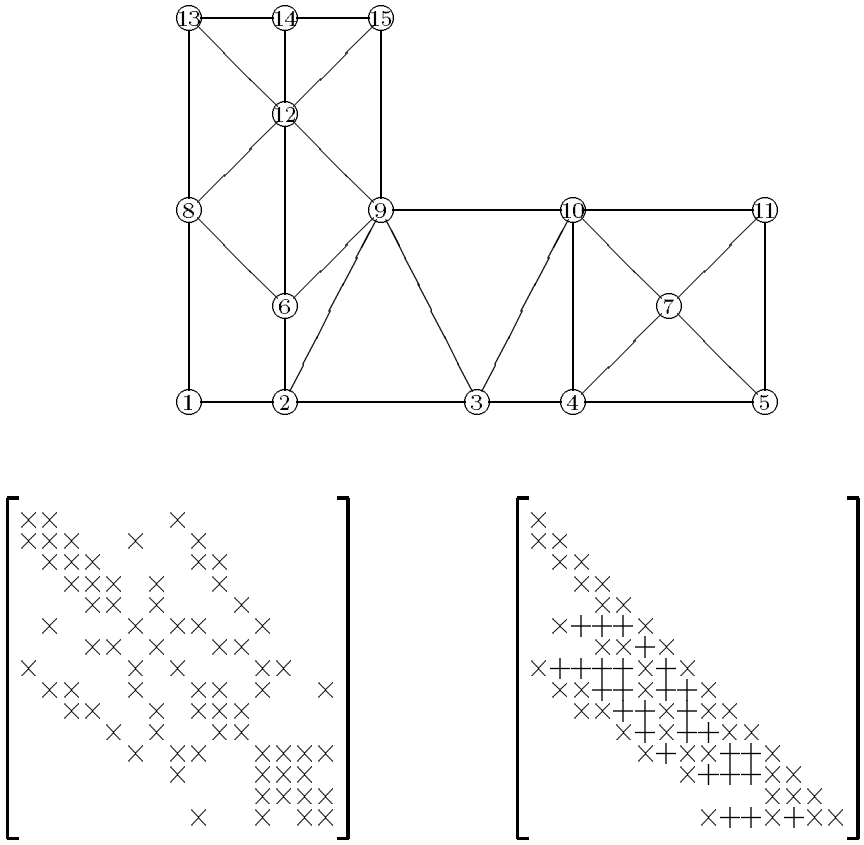

Figure 4: Example finite element graph (top) and the nonzero patterns of the corresponding sparse matrix (left) and its Cholesky factor (right), with fill indicated by +.

Matrix Ordering

The amount of fill (new nonzeros) caused by factorization depends critically on the order in which variables are eliminated. Consider, for example, an "arrow" matrix: if the first row and column are dense, then the factor fills in completely, but if the dense row and column are permuted symmetrically to become the last row and column, then they cause no fill at all.

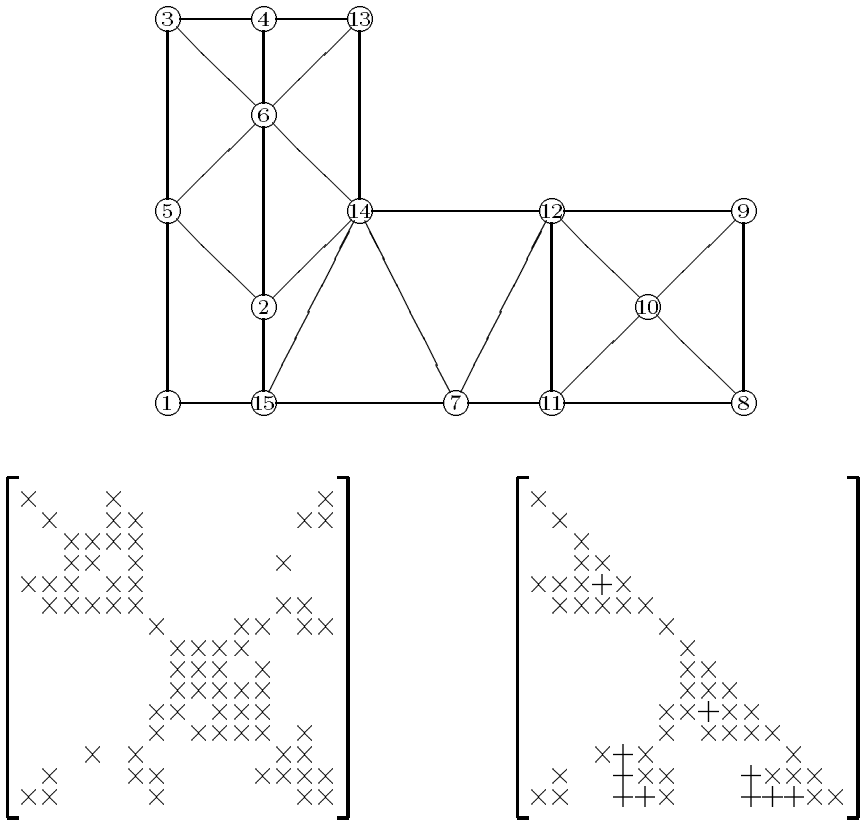

Figure 5: Finite element graph reordered by nested dissection (top) and the nonzero patterns of the corresponding sparse matrix (left) and its Cholesky factor (right), with fill indicated by +.

Thus, an important preliminary step in sparse factorization is to find an ordering for the equations and unknowns in which the factor suffers relatively little fill. The general problem of minimizing fill

exactly is NP-complete, but several relatively cheap heuristics are known that do a good job of limiting fill. These include minimum degree, nested dissection, and various schemes for reducing the bandwidth or profile of a matrix (George and Liu, 1981). A reordering of the previous example, shown in Figure 5, incurs significantly less fill than the original ordering.

One of the key advantages of SPD matrices is that such a fill reducing ordering can be selected in advance of the numeric factorization, independent of the particular values of the nonzero entries: only the pattern of the nonzeros matters, not their numerical values. This would not be the case, in general, if we also had to take into account pivoting for numerical stability, which obviously would require knowledge of the nonzero values, and would introduce a potential conflict between preserving sparsity and preserving stability.

For the SPD case, once the ordering is selected, the locations of all fill elements in L can be anticipated prior to the numeric factorization, and thus an efficient static data structure can be set up in advance to accommodate them (this process is called *symbolic factorization*). This feature stands in contrast to general sparse linear systems, which usually require dynamic data structures to accommodate fill entries as they occur, since their locations depend on numerical information that becomes known only as the numeric factorization process unfolds. Thus, modern algorithms and software for solving sparse SPD systems include a symbolic preprocessing phase in which a sparsity preserving ordering is computed, and a static data structure is set up for storing the entries of L, before any floating-point computation takes place.

Solving Sparse SPD Systems

We now summarize the four basic steps in solving sparse SPD systems by Cholesky factorization:

1. **Ordering**, in which the rows and columns of the matrix are reordered so that the Cholesky factor suffers relatively little fill,
2. **Symbolic factorization**, in which all fill is anticipated and data structures are allocated in advance to accommodate it,
3. **Numeric factorization**, in which the numeric entries of the Cholesky factor are computed,

4. **Triangular solution**, in which the solution is computed by forward and back substitution.

Note that the first two steps involve no floating-point computation. The main purpose of these first two symbolic steps is to reduce the computational resources (time and memory) required by the next two steps, which are floating-point intensive. Indeed, the factorization step usually requires the dominant portion of the computing time for solving sparse problems on conventional computers. Consequently, the factorization step has received by far the most attention in developing parallel algorithms for sparse linear systems on multiprocessor architectures; see (Heath, Ng, and Peyton 1991) for a survey. That same emphasis is reflected in our presentation here, but we will also consider the other steps as well, since a complete suite of parallel algorithms is ultimately required for scalability of the overall solution process.

Sparse Cholesky Factorization Algorithms

To state sparse factorization algorithms precisely, we introduce the following notation. For a given sparse matrix M, we let M_{i*} denote its ith row, and M_{*j} its jth column, and define

$$Struct(M_{i*}) = \{k < i \mid m_{ik} \neq 0\},$$

$$Struct(M_{*j}) = \{k > j \mid m_{kj} \neq 0\}.$$

In other words, $Struct(M_{i*})$ is the sparsity structure of row i of the strict lower triangle of M, while $Struct(M_{*j})$ is the sparsity structure of column j of the strict lower triangle of M.

The sparse column-Cholesky algorithm is shown in Figure 6. A given column j of A remains unchanged until the outer loop index reaches that value of j. At that point, column j is updated by a multiple of each column $k < j$ of L for which $\ell_{jk} \neq 0$. After all column modifications have been applied to column j, the diagonal entry ℓ_{jj} is computed and used to scale the completely updated column to obtain the remaining nonzero entries of L_{*j}. The column-Cholesky algorithm is the most commonly used method in conventional sparse matrix packages.

The sparse submatrix-Cholesky algorithm is shown in Figure 7. As soon as column k has been computed, its effects on subsequent

columns are computed immediately, but only those columns j, $j > k$, such that $\ell_{jk} \neq 0$ are affected.

Sparse column-Cholesky factorization

```
for j = 1, n
    for k ∈ Struct(L_{j*})
        cmod(j, k)
    cdiv(j)
```

- left-looking
- delayed-update
- demand-driven
- fan-in

Figure 6: The column-Cholesky algorithm and its properties.

Sparse submatrix-Cholesky factorization

```
for k = 1, n
    cdiv(k)
    for j ∈ Struct(L_{*k})
        cmod(j, k)
```

- right-looking
- immediate-update
- data-driven
- fan-out

Figure 7: The submatrix-Cholesky algorithm and its properties.

Multifrontal methods, which we discuss below, can be viewed as a compromise between these two extremes. Update information from each column is accumulated in a series of dense submatrices, which are merged as necessary throughout the algorithm, before ultimately being incorporated into each affected target column. Before stating the multifrontal algorithm in detail, however, we need to introduce a bit more machinery. A different type of hybrid between fan-out and fan-in algorithms is given in (Ashcraft, 1992).

Elimination Tree

To help in analyzing the sparse factorization process, we introduce the concept of an *elimination tree*, which is defined by the fol-

lowing *parent* relationship:

$$parent(j) = \begin{cases} \min\{i \in Struct(L_{*j})\}, & \text{if } Struct(L_{*j}) \neq \emptyset, \\ j & \text{otherwise.} \end{cases}$$

Thus, $parent(j)$ is the row index of the first offdiagonal nonzero in column j of L, if any, and has the value j otherwise. The *elimination tree* $T(A)$ is a graph having n vertices, with an edge between vertices i and j, for $i > j$, if $i = parent(j)$. An example is shown in Figure 8.

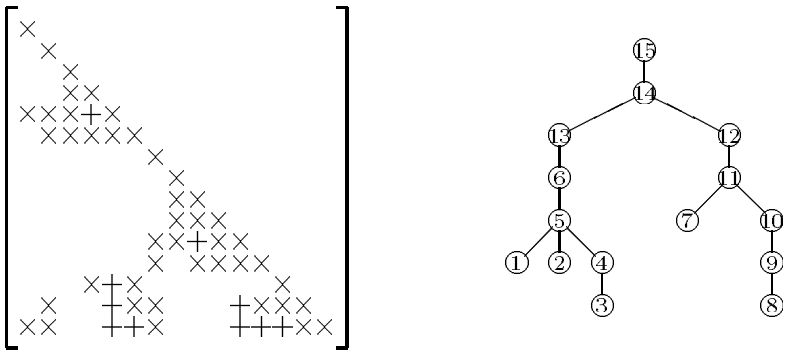

Figure 8: Cholesky factor matrix and corresponding elimination tree for previous (reordered) example.

If the matrix is irreducible, then the elimination tree is indeed a single tree with root at vertex n (otherwise it is more accurately termed an *elimination forest*). For a survey of the role of elimination trees in sparse factorization, see (Liu, 1990).

The significance of the elimination tree is that it displays the data dependences among the columns of the Cholesky factor, and hence reveals potential parallelism:

- Each column of L depends only on its descendants in the elimination tree.
- At any point in the factorization process, all factor columns corresponding to leaf nodes can be computed simultaneously.

- Roughly speaking, the height of the elimination tree determines the longest serial path through the computation, and hence the parallel completion time.
- Similarly, the width of the tree roughly determines the degree of parallelism available (i.e., the number of processors that can be used simultaneously).
- A short, wide, well-balanced elimination tree is desirable for parallel computation. These properties depend on the ordering chosen for the matrix.

The height and width of the elimination tree are only very rough measures of potential parallelism because the amount of computation involved in each branch must be taken into account. In general, however, a short, bushy, well-balanced elimination tree is more advantageous than one that is tall and slender or unbalanced. Just as the fill in the Cholesky factor is very sensitive to the ordering of the matrix, so is the structure of the elimination tree. This suggests that the ordering should be chosen to enhance parallelism, but such an objective may conflict to some degree with preservation of sparsity. Sparsity and parallelism are largely compatible, since the large-grain parallelism is due to sparsity in the first place, but these two criteria are by no means coincident, as we will soon see by example.

Multifrontal Method

In order to state the basic multifrontal algorithm formally, we introduce the frontal matrices F_j and update matrices U_j. Each dense frontal matrix F_j is initialized to zero, except for its first row and column, which contain the nonzero entries from the corresponding row and column of A. The frontal matrix F_j is then updated by performing *extend_add* operations with any previously computed update matrices from its children in the elimination tree. The *extend_add* operation simply merges two submatrices by taking the union of their subscript sets and summing the entries when any collisions occur. Finally, a dense partial Cholesky factorization is carried out on the fully updated F_j, producing the corresponding row and column of L as well as the dense submatrix U_j of update information to be propagated further. We state the multifrontal factorization algorithm formally in Figure 9 as a recursive procedure, so that the call

Factor(j), where j is the root of the elimination tree, factors the matrix A.

Factor(j)

Let $\{i_1, \ldots, i_r\} = \text{Struct}(L_{*j})$

Let $F_j = \begin{bmatrix} a_{j,j} & a_{j,i_1} & \ldots & a_{j,i_r} \\ a_{i_1,j} & 0 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{i_r,j} & 0 & \ldots & 0 \end{bmatrix}$

For each child i of j in the elimination tree

Factor(i)

$F_j = \mathit{extend\_add}(F_j, U_i)$

Perform one step of dense Cholesky factorization so that

$$F_j = \begin{bmatrix} \ell_{j,j} & 0 \\ \ell_{i_1,j} & \\ \vdots & I \\ \ell_{i_r,j} & \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & U_j \end{bmatrix} \begin{bmatrix} \ell_{j,j} & \ell_{i_1,j} & \ldots & \ell_{i_r,j} \\ 0 & & I & \end{bmatrix}.$$

Figure 9: Multifrontal factorization algorithm.

The multifrontal approach benefits from a number of advantages over conventional sparse factorization methods:

- Multifrontal methods perform most arithmetic operations on dense matrices, thereby avoiding much of the high indexing overhead and indirect addressing typical of conventional sparse matrix codes.
- Multifrontal codes can take advantage of such techniques as loop unrolling, vectorization, and optimized BLAS to run at near peak speed on many types of processors.

- Multifrontal methods maintain excellent data locality, which is highly beneficial for effective utilization of memory hierarchies, such as cache, virtual memory with paging, or explicit out-of-core solvers.

- Multifrontal methods are naturally adaptable to parallel implementation by processing multiple independent fronts simultaneously on different processors. Parallelism can also be exploited in the dense matrix computations within each front.

The frontal method was originally developed by (Irons, 1970), motivated primarily as an out-of-core solver. Generalization to the multifrontal method seems to be due to (Speelpenning, 1978). The first widely available implementations were (Duff and Reid, 1983) and (Duff and Reid, 1984). For an excellent tutorial introduction to the multifrontal method, see (Liu, 1992).

Levels of Parallelism Available

We now examine in greater detail the opportunities for parallelism in sparse Cholesky factorization and various algorithms for exploiting it. One of the most important issues in designing any parallel algorithm is selecting an appropriate level of *granularity*, by which we mean the size of the computational subtasks that are assigned to individual processors. The optimal choice of task size depends on the tradeoff between communication costs and the load balance across processors. Three potential levels of granularity can be identified in a parallel implementation of Cholesky factorization (Liu, 1986):

- *fine-grain*, in which each task consists of only one or two floating-point operations, such as a multiply-add pair.

- *medium-grain*, in which each task is a single column operation, such as *cmod* or *cdiv*.

- *large-grain*, in which each task is the computation of an entire group of columns in a subtree of the elimination tree.

Fine-grain parallelism, at the level of individual floating-point operations, is available in either the dense or sparse case. It can be exploited effectively by a vector processing unit or a systolic array,

but would incur far too much communication overhead to be exploited profitably on most current generation parallel computers. In particular, the communication latency of these machines is too great for such frequent communication of small messages to be feasible.

Medium-grain parallelism, at the level of operations on entire columns, is also available in either the dense or the sparse case. This level of granularity accounts for essentially all of the parallel speedup in dense factorization on current generation parallel machines, and it is an extremely important source of parallelism for sparse factorization as well. This parallelism is due primarily to the fact that many *cmod* operations can be computed simultaneously by different processors.

For many problems, a medium level of granularity provides a good balance between communication and computation, but scaling up to very large problems and/or very large numbers of processors may necessitate that the tasks be further broken up based on a two-dimensional partitioning of the columns (Schreiber, 1993). One must keep in mind, however, that in the sparse case an entire column operation may require only a few floating-point operations involving the sparsely populated nonzero elements in the column. For a matrix of order n having a planar graph, for example, the largest embedded dense submatrix to be factored is roughly of order $\sqrt{n}$, and thus a sparse problem must be quite large before a two-dimensional partitioning becomes essential.

Large-grain parallelism, at the level of subtrees of the elimination tree, is available only in the sparse case. If T_i and T_j are disjoint subtrees of the elimination tree, with neither root node a descendant of the other, then all of the columns corresponding to nodes in T_i can be computed completely independently of the columns corresponding to nodes in T_j, and vice versa, and hence these computations can be done simultaneously by separate processors with no communication between them. For example, each leaf node of the elimination tree corresponds to a column of L that depends on no prior columns, and hence all of the leaf node columns can be completed immediately merely by performing the corresponding *cdiv* operation on each of them. Furthermore, all such *cdiv* operations can be performed simultaneously by separate processors (assuming enough processors are available). By contrast, in the dense case all *cdiv* operations must be performed sequentially (at least at this level of granularity), since there is never more than one leaf node at any given time.

Many of the foregoing concepts are illustrated in the following series of simple examples. Figure 10 shows a small one-dimensional mesh with a "natural" ordering of the nodes, the nonzero patterns of the corresponding tridiagonal matrix A and its Cholesky factor L, and the resulting elimination tree $T(A)$. On the positive side, the Cholesky factor suffers no fill at all and the total work required for the factorization is minimal. However, we see that the elimination tree is simply a chain, and therefore there is no large-grain parallelism available. Each column of L depends on the immediately preceding one, and thus they must be computed sequentially. This behavior is typical of orderings that minimize the bandwidth of a sparse matrix: they tend to inhibit rather than enhance large-grain parallelism in the factorization. There is in fact little parallelism of any kind to be exploited in solving a tridiagonal system in this natural order. The *cmod* operations involve only a couple of flops each, so that even the "medium-grain" tasks are actually rather small in this case.

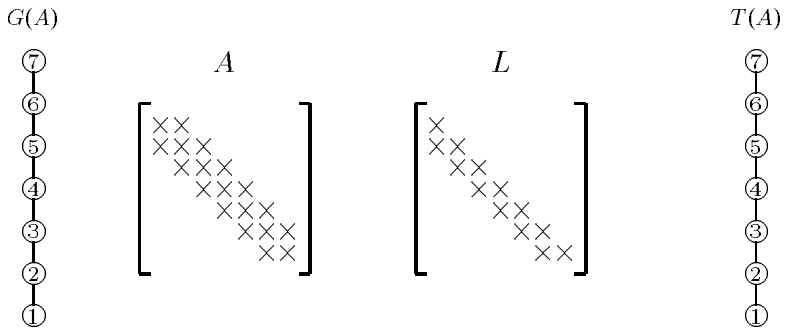

Figure 10: One-dimensional grid and corresponding tridiagonal matrix (left), with Cholesky factor and elimination tree (right).

Figure 11 shows the same one-dimensional mesh with the nodes reordered by a minimum degree algorithm. Minimum degree is the most effective general purpose heuristic known for limiting fill in sparse factorization (George and Liu, 1989). In its simplest form, this algorithm begins by selecting a node of minimum degree (i.e., one having fewest incident edges) in $G(A)$ and numbering it first. The selected node is then deleted and new edges are added, if necessary, to make its former neighbors into a clique. The process is then repeated on the updated graph, and so on, until all nodes have been

numbered. We see in Figure 11 that L suffers no fill in the new ordering, and the elimination tree now shows some large-grain parallelism. In particular, columns 1 and 2 can be computed simultaneously, then columns 3 and 4, and so on. This two-fold parallelism reduces the height (roughly the parallel completion time) by approximately a factor of two.

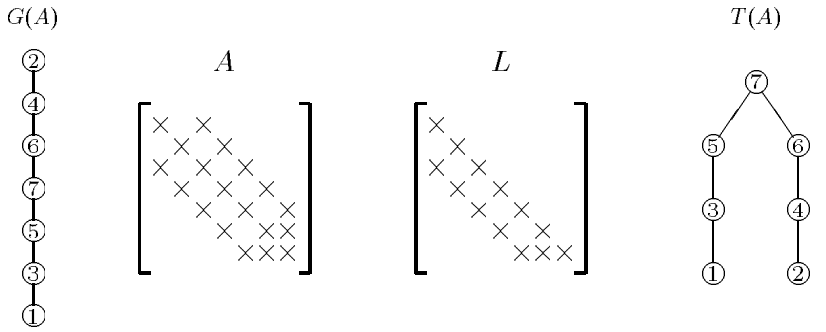

Figure 11: Graph and matrix reordered by minimum degree (left), with corresponding Cholesky factor and elimination tree (right).

At any stage of the minimum degree algorithm, there may be more than one node with the same minimum degree, and the quality of the ordering produced may be affected by the tie breaking strategy used. In the example of Figure 11, we have deliberately broken ties in the most favorable way (with respect to parallelism); the least favorable tie breaking would have reproduced the original ordering of Figure 10, resulting in no parallelism. Breaking ties randomly (which in general is about all one can do) could produce anything in between these two extremes, yielding an elimination tree that reveals some large-grain parallelism, but which is taller and less well balanced than our example in Figure 11. Again, this is typical of minimum degree orderings. In view of this property, a useful strategy is to order the matrix initially by minimum degree for low fill, and then modify the ordering to reduce the height of the elimination tree while preserving the fill (Jess and Kees, 1982), (Liu, 1989), (Lewis, Peyton, and Pothen, 1989).

Figure 12 shows the same mesh again, this time ordered by nested dissection, a divide-and-conquer strategy (George, 1973). Let S be a set of nodes, called a *separator*, whose removal, along with all edges

incident upon nodes in S, disconnects $G(A)$ into two remaining subgraphs. The nodes in each of the two remaining subgraphs are numbered contiguously and the nodes in the separator S are numbered last. This procedure is then applied recursively to split each of the remaining subgraphs, and so on, until all nodes have been numbered. If sufficiently small separators can be found, then nested dissection tends to do a good job of limiting fill, and if the pieces into which the graph is split are of about the same size, then the elimination tree tends to be well balanced. We see in Figure 12 that for our example, with this ordering, the Cholesky factor L suffers fill in two matrix entries (indicated by +), but the elimination tree now shows a four-fold large-grain parallelism, and its height has been reduced further. This behavior is typical of nested dissection orderings: they tend to be somewhat less successful at limiting fill than minimum degree, but their divide-and-conquer nature tends to identify parallelism more systematically and produce better balanced elimination trees.

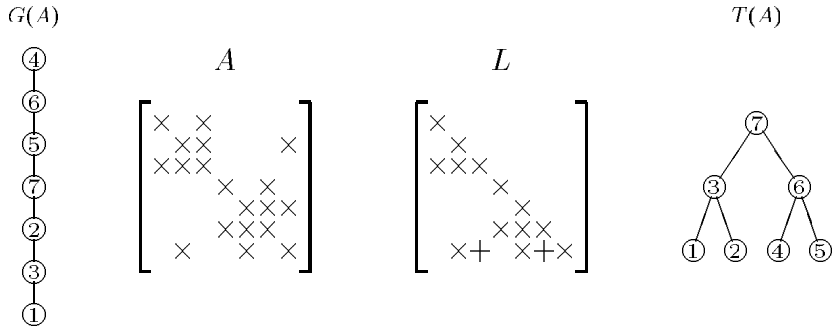

Figure 12: Graph and matrix reordered by nested dissection (left), with corresponding Cholesky factor and elimination tree (right).

Finally, Figure 13 shows the same problem reordered by odd-even reduction. This is not a general purpose strategy for sparse matrices, but it is often used to enhance parallelism in tridiagonal and related systems — see (Duff and Reid, 1986) or (Golub and Van Loan, 1989) — so we illustrate it for the sake of comparison with more general purpose methods. In odd-even reduction, odd node numbers come before even node numbers, and then this same renumbering is applied recursively within each resulting subset, and so on until

all nodes are numbered. Although the resulting nonzero pattern of A looks superficially different, we can see from the elimination tree that this method is essentially equivalent to nested dissection for this type of problem.

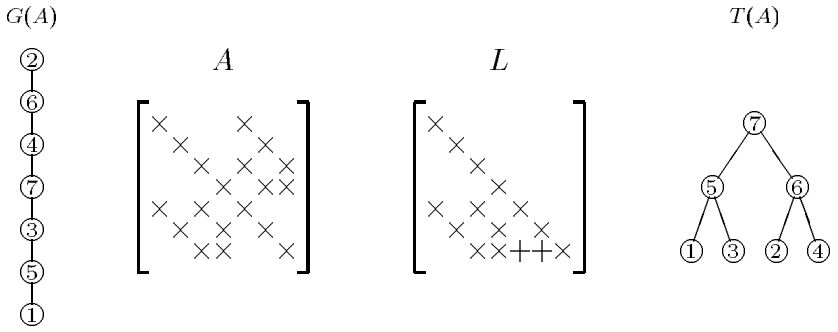

Figure 13: Graph and matrix reordered by odd-even reduction (left), with corresponding Cholesky factor and elimination tree (right).

Distributed Sparse Factorization

Having developed some understanding of the sources of parallelism in sparse Cholesky factorization, we now consider some algorithms for exploiting it. In designing any parallel algorithm, one of the most important decisions is how tasks are to be assigned to processors. In a shared memory parallel architecture, the tasks can easily be assigned to processors dynamically by maintaining a common pool of tasks from which available processors claim work to do. This approach has the additional advantage of providing automatic load balancing to whatever degree is permitted by the chosen task granularity. An implementation of this approach for parallel sparse factorization is given in (George, Heath, Liu, and Ng, 1986).

In a distributed memory environment, communication costs often prohibit dynamic task assignment or load balancing, and thus we seek a static mapping of tasks to processors. In the case of column-oriented factorization algorithms, this amounts to assigning the columns of the matrix to processors according to some mapping procedure determined in advance. Such an assignment could be made using the block or cyclic mappings, or combinations thereof, often used for dense matrices. However, such simple mappings risk

wasting much of the large-grain parallelism identified by means of the elimination tree, and may also incur unnecessary communication. For example, the leaf nodes of the elimination tree can be processed in parallel if they are assigned to different processors, but the latter is not necessarily ensured by a simple block or cyclic mapping.

A better approach for sparse factorization is to preserve locality by assigning subtrees of the elimination tree to contiguous subsets of neighboring processors. A good example of this technique is the "subtree-to-subcube" mapping often used with hypercube multicomputers (George, Heath, Liu, and Ng, 1989). Of course, the same idea applies to other network topologies, such as submeshes of a larger mesh. We will assume that some such mapping is used, and we will comment further on its implications below. Whatever the mapping, we will denote the processor containing column j by $map[j]$, or, more generally, if J is a set of column numbers, $map[J]$ will denote the set of processors containing the given columns.

One of the earliest and simplest parallel algorithms for sparse Cholesky factorization is the version of submatrix-Cholesky shown in Figure 14 due to (George, Heath, Liu, and Ng, 1988). The given algorithm runs on each processor, with each responsible for its own subset, *mycols*, of columns. Any processor that owns a column of L corresponding to a leaf node of the elimination tree can complete it immediately merely by performing the necessary *cdiv* operation, since such a column depends on no prior columns. The resulting factor columns are then broadcast (fanned-out) to all other processors that will need them to update columns that they own. The remainder of the algorithm is then driven by the arrival of factor columns, as each processor goes into a loop in which it receives and applies successive factor columns, in whatever order they may arrive, to whatever columns remain to be processed. When the modifications of a given column have been completed, then the *cdiv* operation is done, the resulting factor column is broadcast as before, and the process continues until all columns of L have been computed.

The fan-out algorithm potentially exploits both the large-grain parallelism characterized by concurrent *cdiv*s and the medium-grain parallelism characterized by concurrent *cmod*s, but its data-driven approach also has a number of drawbacks that severely limit its efficiency. In particular, performing the column updates one at a time by the receiving processors results in unnecessarily high communica-

```
for j ∈ mycols
    if j is a leaf node in T(A)
        cdiv(j)
        send L_{*j} to processors in map(Struct(L_{*j}));
        mycols = mycols − {j}
while mycols ≠ ∅
    receive any column of L, say L_{*k};
    for j ∈ mycols ∩ Struct(L_{*k})
        cmod(j, k)
        if column j requires no more cmods
            cdiv[j]
            send L_{*j} to processors in map(Struct(L_{*j}));
            mycols = mycols − {j}
```

Figure 14: Distributed fan-out sparse Cholesky factorization.

tion frequency and volume, and in a relatively inefficient computational inner loop. The communication requirements can be reduced by careful mapping and by aggregating updating information over subtrees (see, e.g., (George, Liu, and Ng, 1989), (Zmijewski, 1989), (Mu and Rice, 1992)), but even with this improvement, the fan-out algorithm is usually not competitive with more recent algorithms.

The shortcomings of the distributed fan-out algorithm motivated the formulation of the distributed fan-in algorithm given in Figure 15, due to (Ashcraft, Eisenstat, and Liu, 1990). This algorithm takes a demand-driven approach: the updates for a given column j are not computed until needed to complete that column, and they are computed by the sending processors rather than the receiving processor. As a result, all of a given processor's contributions to the updating of the column in question can be combined into a single aggregate update column, which is then transmitted in a single message to the processor containing the target column. This approach not only decreases communication frequency and volume, but it also facilitates a more efficient computational inner loop. In particular, no communication is required to complete the columns corresponding to any subtree that is assigned entirely to a single processor. Thus, with an appropriate locality-preserving and load-balanced subtree mapping,

the fan-in algorithm has a perfectly parallel, communication-free initial phase that is followed by a second phase in which communication takes place over increasingly larger subsets of processors as the computation proceeds up the elimination tree, encountering larger subtrees. The perfectly parallel phase, which is due entirely to sparsity, tends to constitute a larger proportion of the overall computation as the size of the problem grows for a fixed number of processors, and thus the algorithm enjoys relatively high efficiency for sufficiently large problems.

```
for j = 1, n
    if j ∈ mycols or mycols ∩ Struct(L_{j*}) ≠ ∅
        u = 0
        for k ∈ mycols ∩ Struct(L_{j*})
            u = u + ℓ_{jk} * L_{*k}      {aggregate column updates}
        if j ∈ mycols
            incorporate u into the factor column j;
            while any aggregated update column for column
                j remains, receive in u another aggregated
                update column for column j;
                incorporate u into the factor column j;
            cdiv(j)
        else
            send u to processor map[j].
```

Figure 15: Distributed fan-in sparse Cholesky factorization.

The effectiveness of any distributed parallel algorithm depends on a number of factors, including

- communication overhead: both the number of messages and the total volume sent should be kept to a minimum,
- computational overhead: any extra work in the parallel algorithm beyond that required by the best serial algorithm should be minimized,
- load balance: the amount of work assigned to each processor

should be as evenly distributed as possible,

- concurrency: the work should take place simultaneously on all processors (as opposed to being staggered),
- single-node speed: the code running on each processor should attain a reasonable fraction of the peak speed of which the processor is capable,
- scalability: the efficiency of the algorithm should be bounded away from zero as the number of processors grows, assuming that the size problem being solved grows at a reasonable rate.

Applying these criteria to sparse factorization algorithms, the fan-in algorithm improves on the fan-out algorithm in communication cost by amalgamating column updates rather than sending each one individually, and also improves the single-node speed by enabling a more efficient inner loop. However, the simple sparse fan-in algorithm still falls short of the single-node speed of a multifrontal method. As we have seen, in the multifrontal approach update information is propagated through a series of dense submatrices, so that the computations can utilize dense matrix kernels with much more favorable single-node speed, due to lower indexing overhead and good cache utilization.

In addition, it is easy to show on theoretical grounds that any parallel factorization based on one-dimensional matrix partitioning cannot be scalable to very large numbers of processors (Schreiber, 1993). Thus, a further advantage of a multifrontal approach is that it easily inherits favorable scalability properties if two-dimensional matrix partitioning is exploited in the underlying dense kernels. Moreover, the multifrontal method also lends itself readily to the exploitation of large-grain, subtree-level parallelism, and takes full advantage of the data locality induced by a subtree-to-subcube style mapping. For these reasons, the most effective parallel sparse factorization algorithms known today are based on a multifrontal approach. For a unified description and comparison of parallel fan-in, fan-out, and multifrontal method for sparse factorization, see (Ashcraft, Eisenstat, Liu, and Sherman, 1990).

We will not formally state a parallel multifrontal algorithm here, as a high-level description does not differ significantly from Figure 9. Conceptually, it is a tree-structured, divide-and-conquer parallel algorithm that can exploit both large-grain parallelism in the

elimination tree and medium-grain parallelism in the dense frontal and update matrices. Parallel multifrontal algorithms have been developed for both shared-memory and distributed-memory architectures. Early parallel implementations included those of (Duff, 1986), (Ashcraft, Grimes, Lewis, Peyton, and Simon, 1987), (Benner, Montry, and Weigand, 1987), and (Lucas, Blank, and Tieman, 1987). More recent parallel implementations include those of (Gilbert and Schreiber, 1992), (Heath and Raghavan, 1994a), (Conroy, Kratzer, and Lucas, 1994), (Rothberg, 1994), (Gupta and Kumar, 1994), and (Karypis and Kumar, 1994).

To summarize, at the current state of the art, the principal ingredients in an efficient parallel algorithm for sparse Cholesky factorization are

- an ordering that yields a short and well balanced elimination tree while also limiting fill,
- a multifrontal approach to exploit dense subproblems effectively,
- a subtree-to-subcube mapping (or equivalent) for reduced communication,
- a block-cyclic mapping of subproblems for load balancing,
- a two-dimensional partitioning for scalability.

The most recent implementations of parallel sparse factorization mentioned above share some or all of these properties, and have attained both parallel efficiency and absolute performance that is comparable to dense factorization on the same machines. Reported execution rates for some of these sparse codes on selected parallel machines are shown in Table 1. For comparison, the maximum execution rate reported for the Linpack benchmark (for a dense matrix of unlimited size) is shown. The execution rates for the sparse codes are roughly between one-third and one-half of the speed for solving dense systems on the same machines, which is a substantial improvement over previous experience, and a remarkable achievement considering the greater complexity and inherently higher overhead in sparse codes. Figures for speedup and parallel efficiency are a little harder to come by, since large problems usually cannot be solved on a single processor, but the code of (Gupta and Kumar, 1994)

has been reported to achieve a speedup of 364 on an nCUBE-2 with 1024 processors. The multifrontal algorithms for sparse factorization of Kumar and co-workers have been shown to be as scalable as the underlying algorithm for dense factorization.

Table 1: Execution rate in Gflops for sparse and dense factorization.

implementation	machine	proc.	sparse	dense
(Conroy, *et al.*, 1994)	MasPar MP-2	16K	.62	1.6
(Rothberg, 1994)	Intel Paragon	128	1.7	4.0
(Kapyris & Kumar, 1994)	Cray T3D	256	6.0	21.4

Parallel Ordering

In addition to the factorization, a complete algorithm for direct solution of sparse linear systems also requires ordering, symbolic factorization, and triangular solution steps. These other steps, although usually of much lower cost than the factorization serially, must be done in parallel in order to have a scalable overall algorithm.

Many popular ordering algorithms, such as multiple minimum degree and automatic nested dissection, are very efficient serially, but do not parallelize well. Most alternative approaches to ordering that can be effectively parallelized are based on nested dissection, which requires the computation of small separators for the graph of the matrix. Methods for choosing good separators that lend themselves to effective parallel implementation include

- coordinate bisection (Heath and Raghavan, 1992),
- inertial (Williams, 1991),
- geometric (Miller, Teng, and Vavasis, 1991),
- spectral (Pothen, Simon, and Liou, 1990).

Several of these methods are implemented (though not in parallel) in the Chaco package for graph partitioning (Hendrickson and Leland, 1993), which allows for convenient comparison among them. Chaco also includes a "smoother," based on the Kernighan-Lin algorithm, for iteratively improving graph partitions computed by any of the

methods. For additional recent work on improving graph partitions, see (Ashcraft and Liu, 1994).

These parallel ordering techniques differ in the amount of computation required and in the quality of orderings they provide, but specific results are problem dependent. These parallel methods generally require more total computation than good serial ordering algorithms, and some are much more expensive. Thus, in a parallel setting, the ordering step may no longer require only a small fraction of the overall solution time.

Parallel Triangular Solution

Algorithmically, parallel implementation of the triangular solution step is similar to that of the factorization, but there is far less computation over which to amortize the necessary communication, so that the relative efficiency of the triangular solution tends to be much lower than that of the factorization, whether dense or sparse. For a review of parallel algorithms for triangular solution with dense matrices, see (Heath and Romine, 1988).

A lack of efficiency in solving triangular systems is also an impediment to the effective parallel implementation of iterative methods that use a triangular preconditioner, such as an incomplete Cholesky factorization, and therefore require the solution of a sparse triangular system at each iteration. Indeed, much of the work on parallel sparse triangular solvers has been motivated by their use in iterative, rather than direct, methods, for example (Greenbaum, 1986), (Anderson and Saad, 1989), (Saltz, 1990), (Rothberg and Gupta, 1992). All of these algorithms are based on the familiar substitution method. A recently developed alternative parallel algorithm for solving sparse triangular systems (Alvardo and Schreiber, 1993) is based instead on the product form of the inverse, originally popularized in linear programming.

A Fully Parallel Sparse Solver

A first attempt at producing a fully parallel sparse solver is presented in (Heath and Raghavan, 1994b), where each of the four basic *steps* (ordering, symbolic factorization, numeric factorization, and triangular solution) is divided into two distinct *phases*:

- a local phase, in which the processors work independently and

require no interprocessor communication, and

- a distributed phase, in which the processors collaborate and consequently require interprocessor communication.

A simple framework was developed into which techniques of varying levels of sophistication can be incorporated for the individual modules. Some of the design decisions made in the interest of keeping the implementation task to a manageable size were:

- the ordering technique exploits an embedding of the problem in Euclidean space to compute small line or plane separators in parallel, subject to a balance constraint,
- the symbolic and numeric factorization algorithms are based on the separator tree arising from nested dissection, foregoing more precise characterizations of fill,
- in the numeric factorization, a column-oriented one-dimensional matrix partitioning is used rather than a theoretically more scalable two-dimensional partitioning,
- in assigning columns to processors, an equal weighting of the subtrees in the separator tree is used, which may induce a load imbalance.

Thus, in the interest of simplicity, some compromises were made in potential performance and scalability. The result is a research prototype, not a highly optimized production code, but it nevertheless allows experimentation with the relative costs of the various steps in a fully parallel solver, and also provides a basis for incremental improvement of each module.

The ordering step is based on Cartesian nested dissection (Heath and Raghavan, 1992), which makes use of coordinate information for the underlying graph of the matrix. Determining a line or plane separator requires counting and searching operations in each coordinate dimension for a series of trial coordinate values, and these operations are carried out in a distributed parallel manner using global communication operations analogous to parallel prefix. The coordinate-based separator algorithm is applied repeatedly to further subdivide the resulting subgraphs until there are as many subgraphs as processors. At each level of nested dissection, the algorithm chooses

the smallest possible separator consistent with a balance constraint. CND has been shown, both theoretically and empirically, to be an effective ordering for most sparse problems that arise in practice (Raghavan, 1993a).

Parallel ordering incurs a bootstrapping problem in that determining an appropriate locality-preserving mapping of the problem onto the processors is essentially equivalent to the ordering problem itself. Thus one cannot assume any data locality at the outset of the ordering step. Initially, the problem is mapped onto the distributed processor memories essentially arbitrarily. Once the dissection process has produced as many subgraphs as there are processors, the problem data can then be redistributed so that each subgraph is assigned entirely to a separate processor. After such redistribution, then the ordering process can continue, with no further interprocessor communication, until all nodes have been numbered. This local phase can continue to employ the CND ordering algorithm or can use any sequential ordering algorithm desired.

In conventional sparse matrix codes, the symbolic factorization step is designed to anticipate all fill and allocate the necessary data structures to accommodate it. In a sequential setting, it is relatively easy to compute the fill exactly, but in a distributed parallel setting a significant amount of communication would be required, so instead we use the simpler structure given by the separator tree that results naturally from the nested dissection process. In effect, we assume that the submatrix corresponding to each separator is dense, which may result in an overestimate of the fill, but in practice this is seldom excessive. Columns corresponding to the nodes in a given local subgraph are assigned to the processor owning that subgraph, while columns corresponding to separator nodes are mapped cyclically among the subset of processors owning columns corresponding to nodes that are connected to the given separator.

The numeric factorization step is a multifrontal implementation of sparse Cholesky factorization (Heath and Raghavan, 1994a), which starts at the leaves of the separator tree, merging portions of the corresponding dense submatrices and propagating the resulting update information upward to higher levels in the tree. Initially the computation is entirely local, as each processor factors the matrix columns corresponding to the internal nodes of the subgraph that it has been assigned, with no dependence on any columns owned by other processors. Eventually interprocessor communication is required in order

to factor matrix columns corresponding to separator nodes, which depend on data from multiple processors. Update information from different processors is incorporated as the computation moves up the separator tree, involving an expanding hierarchy of processor subsets until reaching the root (i.e., the highest level separator), at which point all processors cooperate in the computation. This framework has been further generalized to include both LU and QR factorization for nonsymmetric and rectangular sparse systems (Raghavan, 1993b).

Like numeric factorization, the forward substitution begins with purely local computation on the local subgraphs, and then proceeds up the separator tree from leaves to root. The back substitution process is just the opposite, beginning at the root and proceeding downward toward the leaves of the separator tree. Thus, for the back substitution, the distributed phase precedes the final local phase that completes the computation of the solution.

The scalability of the overall algorithm, and relative costs of the individual steps, are illustrated in the Figure 16, which shows the execution times on a CM-5 for a series of square grid problems whose size was chosen so that the amount of work per processor in the factorization is approximately constant as the number of processors varies. Under these conditions, the total execution time should remain constant if the proportion of parallel overhead does not grow with the number of processors.

The execution time is broken down into the incremental costs of the various steps and phases (some steps are too brief to show the separate phases), with the total given by the upper curve. The steps and phases are plotted in the same order in which they occur in the algorithm, so each curve shows the cumulative execution time through the given phase for a given problem. We see in Figure 16 that the overall execution time (upper curve) is fairly flat, but appears to be trending upward as we reach 128 processors, so the solver apparently falls short of being scalable in this sense.

By considering the individual phases, we can see some of the reasons for this behavior. The execution time degrades significantly in going from 1 to 2 processors, primarily due to the redistribution phase that is required in the parallel algorithm. We observe further that as the number of processors increases, execution times for the distributed phases grow at a slightly faster rate than those for the corresponding local phases decline, yielding an overall rise in

execution time. More detailed examination reveals that both load imbalance and communication overhead grow with the number of processors, leading to the observed rise in overall execution time. Both of these sources of inefficiency could be alleviated by the use of known techniques for improved partitioning and mapping of submatrices to processors.

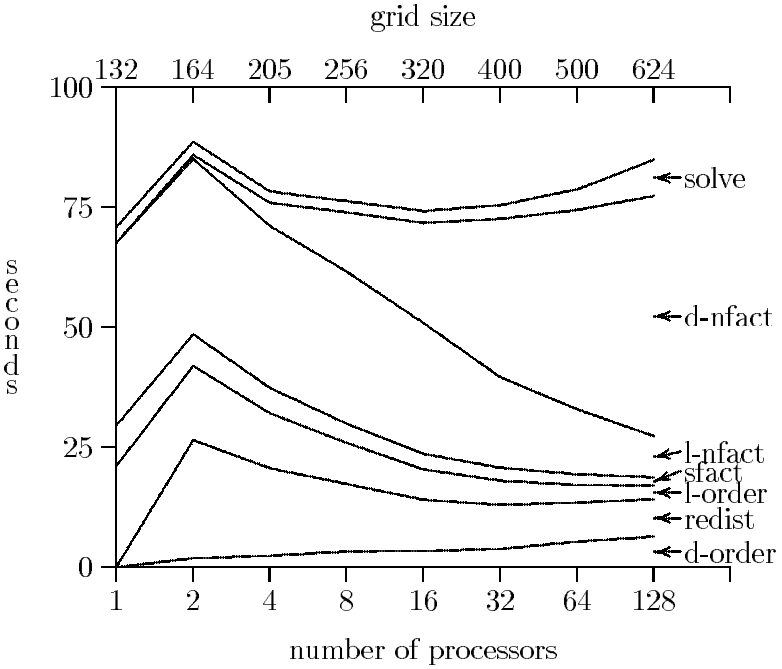

Figure 16: Execution time for series of grid problems on CM-5.

References

Alvardo, F., and Schreiber, R., 1993. "Optimal parallel solution of sparse triangular systems," *SIAM J. Sci. Comput.* **14**, pp. 446–460.

Anderson, E., and Saad, Y., 1989. "Solving sparse triangular linear systems on parallel computers," *Internat. J. High Speed Comput.* **1**, pp. 73–95.

Ashcraft, C., 1992. "The fan-both family of column-based distributed Cholesky factorization algorithms," Tech. Rept. MEA-

TR-208, Boeing Computer Services, Seattle, WA.

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

Ashcraft, C., Eisenstat, S., Liu, J., and Sherman, A., 1990. "A comparison of three column-based distributed sparse factorization schemes," Tech. Rept. YALEU/DCS/RR-810, Dept. of Computer Science, Yale University, New Haven, CT.

Ashcraft, C., Grimes, R., Lewis, J., Peyton, B., and Simon, H., 1987. "Progress in sparse matrix methods for large linear systems on vector supercomputers," *Internat. J. Supercomp. Appl.* **1**, pp. 10–30.

Ashcraft, C., and Liu, J., 1994. "Generalized nested dissection: some recent progress," *Proc. Fifth SIAM Conf. Appl. Linear Algebra*, SIAM Publications, Philadelphia, PA, pp. 130–134.

Benner, R., Montry, G., and Weigand, G., 1987. "Concurrent multifrontal methods: shared memory, cache, and frontwidth issues," *Internat. J. Supercomp. Appl.* **1**, pp. 26–44.

Conroy, J., Kratzer, S., and Lucas, R., 1994. "Data-parallel sparse matrix factorization," *Proc. Fifth SIAM Conf. Appl. Linear Algebra*, SIAM Publications, Philadelphia, PA, pp. 377–381.

Demmel, J., Heath, M., and van der Vorst, H., 1993. "Parallel numerical linear algebra," *Acta Numerica* **2**, pp. 111–197.

Dongarra, J., Duff, I., Sorensen, D., and van der Vorst, H., 1991. *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM Publications, Philadelphia, PA.

Dongarra, J., Gustavson, F., and Karp, A., 1984. "Implementing linear algebra algorithms for dense matrices on a vector pipeline machine," *SIAM Review* **26**, pp. 91–112.

Duff, I., 1986. "Parallel implementation of multifrontal schemes," *Parallel Computing* **3**, pp. 193–204.

Duff, I., Erisman, A., and Reid, J., 1986. *Direct Methods for Sparse Matrices*, Oxford University Press, Oxford, England.

Duff, I., and Reid, J., 1983. "The multifrontal solution of indefinite sparse symmetric linear equations," *ACM Trans. Math. Software* **9**, pp. 302–325.

Duff, I., and Reid, J., 1984. "The multifrontal solution of unsymmetric sets of linear equations," *SIAM J. Sci. Stat. Comput.* **5**, pp. 633–641.

George, A., 1973. "Nested dissection of a regular finite element mesh," *SIAM J. Numer. Anal.* **10**, pp. 345–363.

George, A., Heath, M., and Liu, J., 1986. "Parallel Cholesky factorization on a shared-memory multiprocessor," *Linear Algebra Appl.* **77**, pp. 165–187.

George, A., Heath, M., Liu, J., and Ng, E., 1986. "Solution of sparse positive definite systems on a shared memory multiprocessor," *Internat. J. Parallel Programming* **15**, pp. 309–325.

George, A., Heath, M., Liu, J., and Ng, E., 1988. "Sparse Cholesky factorization on a local-memory multiprocessor," *SIAM J. Sci. Stat. Comput.* **9**, pp. 327–340.

George, A., Heath, M., Liu, J., and Ng, E., 1989. "Solution of sparse positive definite systems on a hypercube," *J. Comp. Appl. Math.* **27**, pp. 129–156.

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

George, A., and Liu, J., 1989. "The evolution of the minimum degree ordering algorithm," *SIAM Review* **31**, pp. 1–19.

George, A., Liu, J., and Ng, E., 1989. "Communication results for parallel sparse Cholesky factorization on a hypercube," *Parallel Computing* **10**, pp. 287–298.

Gilbert, J., and Schreiber, R., 1992. "Highly parallel sparse Cholesky factorization," *SIAM J. Sci. Stat. Comput.* **13**, pp. 1151–1172.

Golub, G., and Van Loan, C., 1989. *Matrix Computations*, 2nd edition, Johns Hopkins University Press, Baltimore, MD.

Greenbaum, A., 1986. "Solving sparse triangular linear systems using Fortran with extensions on the NYU Ultracomputer prototype," Tech. Rept. 99, NYU Ultracomputer Note, New York University.

Gupta, A., and Kumar, V., 1994. "A scalable parallel algorithm for sparse matrix factorization," Tech. Rept. 94-19, Dept. of Computer Science, University of Minnesota, Minneapolis, MN.

Heath, M., Ng, E., and Peyton, B., 1991. "Parallel algorithms for sparse linear systems," *SIAM Review* **33**, pp. 420–460.

Heath, M., and Raghavan, P., 1992. "A Cartesian nested dissection algorithm," Tech. Rept. UIUCDCS-R-92-1772, Dept. of Computer Science, University of Illinois, Urbana, IL (to appear in *SIAM J. Matrix Anal. Appl.*).

Heath, M., and Raghavan, P., 1994a. "Distributed solution of sparse linear systems," *Proc. Scalable Parallel Libraries Conf.*, IEEE Computer Society Press, Los Alamitos, CA, pp. 114–122.

Heath, M., and Raghavan, P., 1994b. "Performance of a fully parallel sparse solver," *Proc. Scalable High-Performance Computing Conf.*, IEEE Computer Society Press, Los Alamitos, CA, pp. 334–341.

Heath, M., and Romine, C., 1988. "Parallel solution of triangular systems on distributed-memory multiprocessors," *SIAM J. Sci. Stat. Comput.* **9**, pp. 558–588.

Hendrickson, B., and Leland, R., 1993. "The Chaco user's guide, version 1.0," Tech. Rept. SAND93-2339, Sandia National Laboratories, Albuquerque, NM.

Irons, B., 1970. "A frontal solution program for finite element analysis," *Internat. J. Numer. Meth. Engrg.* **2**, pp. 5–32.

Karypis, G., and Kumar, V., 1994. "A high performance sparse Cholesky factorization algorithm for scalable parallel computers," Tech. Rept. 94-41, Dept. of Computer Science, University of Minnesota, Minneapolis, MN.

Jess, J., and Kees, H., 1982. "A data structure for parallel L/U decomposition," *IEEE Trans. Comput.* **C-31**, pp. 231–239.

Lewis, J., Peyton, B., and Pothen, A., 1989. "A fast algorithm for reordering sparse matrices for parallel factorization," *SIAM J. Sci. Stat. Comput.* **10**, pp. 1156–1173.

Liu, J., 1986. "Computational models and task scheduling for parallel sparse Cholesky factorization," *Parallel Computing* **3**, pp. 327–342.

Liu, J., 1989. "Reordering sparse matrices for parallel elimination," *Parallel Computing* **11**, pp. 73–91.

Liu, J., 1990. "The role of elimination trees in sparse factorization," *SIAM J. Matrix Anal. Appl.* **11**, pp. 134–172.

Liu, J., 1992. "The multifrontal method for sparse matrix solution: theory and practice," *SIAM Review* **34**, pp. 82–109.

Lucas, R., Blank, W., and Tieman, J., 1987. "A parallel solution method for large sparse systems of equations," *IEEE Trans. Computer Aided Design* **CAD-6**, pp. 981–991.

Miller, G., Teng, S., and Vavasis, S., 1991. "A unified geometric approach to graph separators," *Proc. 32nd Ann. Symp. Foundations of Computer Science*, IEEE Computer Society, Los Alamitos, CA, pp. 538–547.

Mu, M., and Rice, J., 1992. "A grid based subtree-subcube assignment strategy for solving partial differential equations on hypercubes," *SIAM J. Sci. Stat. Comput.* **13**, pp. 826–839.

Ortega, J., 1988. *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum Press, New York.

Pothen, A., Simon, H., and Liou, K., 1990. "Partitioning sparse matrices with eigenvectors of graphs," *SIAM J. Matrix Anal. Appl.* **11**, pp. 430–452.

Raghavan, P., 1993a. "Line and plane separators," Tech. Rept. UIUCDCS-R-93-1794, Dept. of Computer Science, University of Illinois, Urbana, IL.

Raghavan, P., 1993b. "Distributed sparse Gaussian elimination and orthogonal factorization," Tech. Rept. UIUCDCS-R-93-1818, Dept. of Computer Science, University of Illinois, Urbana, IL.

Robert, Y., 1990. *The Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm*, John Wiley and Sons, New York.

Rothberg, E., 1994. "Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon multicomputers," *Proc. Scalable High-Performance Computing Conf.*, IEEE Computer Society Press, Los Alamitos, CA, pp. 324–333.

Rothberg, E., and Gupta, A., 1992. "Parallel ICCG on a hierarchical memory multiprocessor — addressing the triangular solve bottleneck," *Parallel Computing* **18**, pp. 719–741.

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

Schreiber, R., 1993. "Scalability of sparse direct solvers," in *Graph Theory and Sparse Matrix Computation*, Springer-Verlag, pp. 191–209.

Speelpenning, B., 1978. "The generalized element method," Tech. Rept. UIUCDCS-R-78-946, Dept. of Computer Science, University of Illinois, Urbana, IL.

Williams, R., 1991. "Performance of dynamic load balancing algorithms for unstructured mesh calculations," *Concurrency: Practice and Experience* **3**, pp. 457–481.

Zmijewski, E., 1989. "Limiting communication in parallel sparse Cholesky factorization," Tech. Rept. TRCS89-18, Dept. of Computer Science, University of California, Santa Barbara, CA.