Abstract

This paper compares three distributed algorithms for factoring a large sparse symmetric positive definite matrix on a local-memory parallel processor. The fan-out, fan-in, and distributed multifrontal schemes are presented in a unified framework which highlights their communication requirements and their innermost loops. Experimental results on an Intel iPSC/2 hypercube illustrate their relative performance.

A Comparison of Three Column-based Distributed Sparse Factorization Schemes

Cleve Ashcraft[†], *Stanley C. Eisenstat*[‡],
Joseph W. H. Liu[§], *and Andrew H. Sherman*[‡]

Research Report YALEU/DCS/RR-810
July 1990

Approved for public release; distribution is unlimited.

[†] Department of Computer Science, Yale University, New Haven, Connecticut 06520. The research of this author was supported in part by the Office of Naval Research under contract N00014-86-K-0310 and the National Science Foundation under grant DCR-85-21451.

[‡] Department of Computer Science and Research Center for Scientific Computation, Yale University, New Haven, Connecticut 06520. The research of these authors was supported in part by the Office of Naval Research under contract N00014-86-K-0310 and the National Science Foundation under grant DCR-85-21451.

[§] Department of Computer Science, York University, North York, Ontario, Canada M3J 1P3. The research of this author was supported in part by the Natural Sciences and Engineering Research Council of Canada under grant A5509.

1 Introduction

The solution of large sparse systems of linear equations is an important application of parallel computers. In this paper, we give a unified presentation and compare the performance of three distributed schemes to compute the Cholesky factor of a large sparse symmetric positive definite matrix on a local-memory parallel processor.

The three distributed schemes are column-based — we assume that each processor has been assigned a set of columns in the matrix and is responsible for computing the corresponding set of columns in the Cholesky factor. The *fan-out* method [11] is a sparse analog of the parallel outer-product algorithm for factoring dense matrices [10]. The *fan-in* method [1] is a parallel analog of the standard algorithm for factoring sparse matrices [12] which uses a distributed fan-in scheme (cf. [19]). The distributed *multifrontal* method [5, 6, 7, 16, 18] is a parallel implementation of the multifrontal algorithm [21, 8].

The performance of these schemes depends on the ordering of the variables and equations and the mapping of columns to processors, but we do not consider these issues in this paper. We simply assume that the matrix has been ordered by some fill-reducing ordering which is appropriate for parallel elimination, and that the columns of the permuted matrix have been assigned to processors in a manner which is consistent with efficiency.

In §2, we review Cholesky factorization by columns. In §3, we review elimination trees and introduce domains and separators, two partitions of the matrix problem based on the structure of the elimination tree and the mapping function. In §4, we describe the fan-out algorithm and an improved variant, the *domain fan-out* algorithm [3, 22], which offers substantial reductions in message traffic. In §5, we describe the fan-in algorithm. In §6, we describe the distributed multifrontal algorithm in the same terms and then relate this to the more standard description in terms of frontal matrices.

In §7, we compare the three distributed schemes on an Intel iPSC/2 hypercube. The fan-in and multifrontal schemes are shown to require significantly less message traffic than the domain fan-out scheme on a model problem. In terms of overall parallel factorization time for our *current* implementations, the distributed multifrontal method is better than the fan-in scheme, which is, in turn, better than the domain fan-out scheme.

Throughout this paper, we assume that there are p processors interconnected by some underlying network and communicating among themselves only through messages. We assume that the jth column is assigned to processor $map[j]$, which is also responsible for computing the jth column of the Cholesky factor. We use π to represent any of the p processors, and *myname* to denote the processor on which the code is running.

2 Column-Cholesky Factorization

Let A be an $n \times n$ symmetric positive definite matrix and let L be its Cholesky

Algorithm 1 Column-Cholesky Factorization

for column $j := 1$ **to** n **do**

begin

$$\begin{pmatrix} t_j \\ \vdots \\ t_n \end{pmatrix} := \begin{pmatrix} a_{jj} \\ \vdots \\ a_{nj} \end{pmatrix} - \sum_{\substack{k<j \\ l_{jk}\neq 0}} l_{jk} \begin{pmatrix} l_{jk} \\ \vdots \\ l_{nk} \end{pmatrix} \tag{1}$$

$$\begin{pmatrix} l_{jj} \\ \vdots \\ l_{nj} \end{pmatrix} := \frac{1}{\sqrt{t_j}} \begin{pmatrix} t_j \\ \vdots \\ t_n \end{pmatrix}$$

end

factor, with entries a_{ij} and l_{ij}, respectively. As shown in Algorithm 1, the column-Cholesky method computes L column by column. The temporary vector $(t_j, \cdots, t_n)^T$ is used only for clarity; its storage can overlap completely with that for $(l_{jj}, \cdots, l_{nj})^T$.

Our formulation is applicable to both dense and sparse matrices. In the sparse case, both column j of A and the updates $l_{jk}(l_{jk}, \cdots, l_{nk})^T$ are sparse so that the vector operations can be performed in a manner which takes advantage of sparsity. The columns contributing updates to the sum are given precisely by the nonzero structure of row j of L:

$$Struct(L_{j*}) = \{k \mid k < j,\ l_{jk} \neq 0\}\ .$$

There are many ways to implement equation (1) in a distributed manner; the algorithms we consider differ in the order in which and the processor on which updates are computed and applied. As an aid in describing these schemes, we now introduce four related notions [1]:

- *factor column* $L_{*j} = (l_{jj}, \cdots, l_{nj})^T$;
- *update column* $l_{jk}(l_{jk}, \cdots, l_{nk})^T$, where $l_{jk} \neq 0$;
- *complete update column* $\sum_{k \in Struct(L_{j*})} l_{jk}(l_{jk}, \cdots, l_{nk})^T$;
- *aggregate update column* $\sum_{k \in K} l_{jk}(l_{jk}, \cdots, l_{nk})^T$, where $K \subseteq Struct(L_{j*})$.

Update columns and complete update columns are special cases of aggregate update columns where $K = \{k\}$ and $K = Struct(L_{j*})$, respectively.

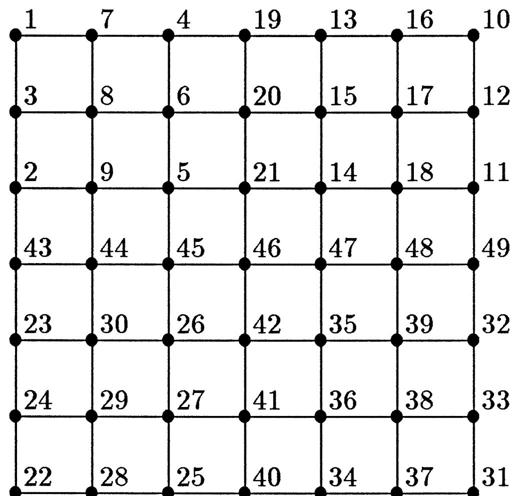

Figure 1: A nested dissection ordering of a 7-by-7 grid.

3 Elimination Trees, Domains, and Separators

3.1 Elimination Trees

If $k < j$ and $l_{jk} \neq 0$ (i.e., $k \in Struct(L_{j*})$),, then column j depends on column k. The graph which represents this precedence relation is precisely the directed graph of the matrix L^T. Taking the transitive reduction yields the *elimination tree* [14, 15, 20].

More directly, the elimination tree of the matrix A is a tree[1] with n nodes $\{1, \cdots, n\}$ such that node j is the parent of node k if and only if

$$j = \min\{i \mid k < i, \; l_{ik} \neq 0\}.$$

We will use j to refer to both a column of the matrix and the corresponding node in the elimination tree. Figure 1 shows a nested dissection ordering of a 9-point finite difference operator on a 7×7 grid. Figure 2 shows the corresponding elimination tree.

Without loss of generality, we will assume that the nodes are ordered in what corresponds to a postorder traversal of the elimination tree (i.e., each of the subtrees of a node is numbered and then the node itself is numbered as in Figure 2). Such an ordering is equivalent[2] to any other ordering which generates the same elimination tree [15].

3.2 Domains

Let $T[x]$ denote the subtree of the elimination tree T rooted at the node x (that is, the node x together with all of its descendants in T), and let $parent[x]$ denote the

[1] If A is reducible, then the elimination "tree" is actually a forest.

[2] Here equivalence is in terms of the number of nonzeroes in L, the amount of work to factor A, and the operations required to compute the factor.

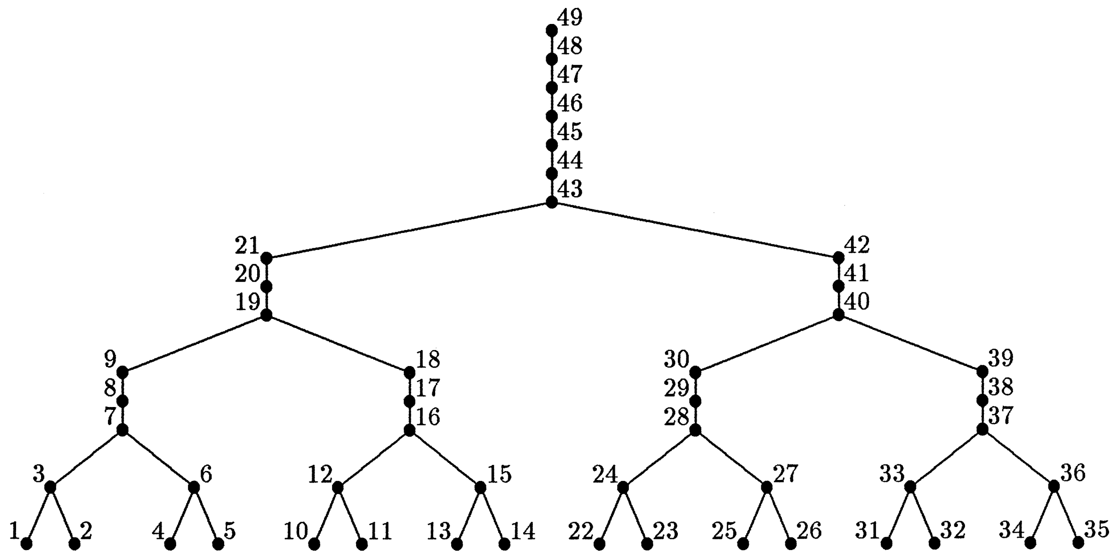

Figure 2: The elimination tree for the ordering in Figure 1.

parent of node x in the tree. If every node y in $T[x]$ is mapped to the same processor as x (i.e., $map[y] = map[x]$ for all $y \in T[x]$), and this property is not satisfied by $T[parent[x]]$, then the nodes in $T[x]$ are said to form a *domain.*[3]

In Figure 3 the tree structure of Figure 2 has been annotated to illustrate features corresponding to solution on a four-processor hypercube. The labels "P_i" show a possible mapping of the nodes/columns to the processors, and the domains are indicated by heavy outlines.

There are four domains, given by the subtrees rooted at nodes 9, 18, 30, and 39, respectively. Figure 4 shows the same 7×7 grid as Figure 1, with the four domains identified by heavy outlines.

Since the columns associated with a domain depend only on columns within the same domain, these factor columns can be computed by the processor to which the domain is mapped without any interprocessor communication. Mu and Rice [17] have examined such mappings and shown that the total communication requirements are lower than with other subtree-to-subcube mappings for fan-out type methods.

Since we have assumed that the nodes in the elimination tree are ordered by a postorder traversal, the nodes within a domain are numbered consecutively; i.e., each domain corresponds to a block of contiguous columns in the matrix.

---

[3] An alternate definition is that a domain is a connected subset R of nodes in the graph of A satisfying:

- if y and z are in R, then $map[y] = map[z]$;
- if y and u are adjacent and y is in R but u is not in R, then y is ordered before u;
- no superset of R satisfies these properties.

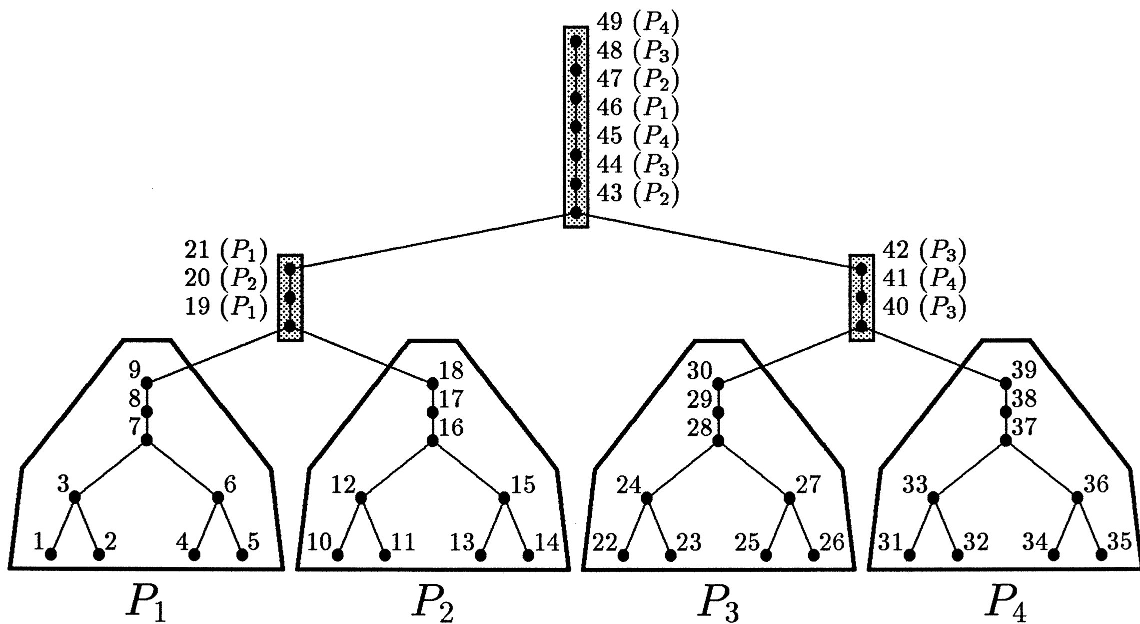

Figure 3: The elimination tree showing processor mapping, domains (heavy outlines), and separators (shaded rectangles).

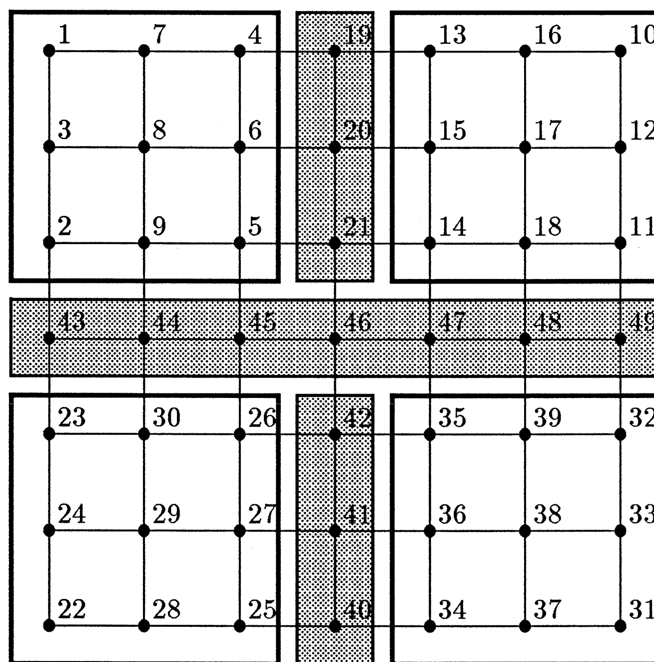

Figure 4: The 7×7 grid showing domains (heavy outlines) and separators (shaded rectangles).

3.3 Separators

While domains group nodes based on the processor to which they are assigned, it is also useful to group nodes based on the structure of their factor columns.

A maximal subset S of nodes/columns which are pairwise connected in the graph of L and which have identical Cholesky column structures outside S is said to form a (*supernodal*) *separator* [4]. In Figure 4 the top-level separators are identified by shading. In terms of the elimination tree, each separator is a chain of nodes (although not all chains are separators). In Figure 3, the top-level separators are identified by shading.

Just as we defined the parent of a node in the elimination tree, we can define the parent $parent[S]$ of a separator S as the separator S' containing the parent of the highest node in S.

Since we have assumed that the nodes have been ordered by a postorder traversal, a separator corresponds to a block of contiguous columns in the Cholesky factor, where the diagonal block is dense triangular and the off-block-diagonal column structures are identical.

4 The Fan-out Distributed Algorithm

The fan-out algorithm [11] is a distributed sparse factorization scheme in which all communication is through factor columns and all updates to column j are computed and applied by processor $map[j]$. The implementation in Algorithm 2 works for any mapping and any network topology. For simplicity, we have allowed processors to send factor columns to themselves; in practice, the corresponding updates are computed and applied as soon as the factor column has been sent to other processors.

Most of the floating-point operations are performed in the assignment (2). This is a sparse AXPY, adding a multiple ($\alpha = -l_{ik}$) of a sparse vector ($x = (l_{ik}, ..., l_{nk})^T$) to another sparse vector ($y = L_{*i}$). It can be implemented efficiently by expanding y into a dense vector t, sparsely adding αx (which is computed with a scalar-dense-vector multiply), and then compressing the result back into y. The overhead (the expansion and compression) can be reduced significantly by saving the factor columns so that all of the updates to a column are computed and applied at the same time. Unfortunately, this also increases the storage requirements.

If one processor owns all of the columns in some domain R, then that processor can compute the factor columns corresponding to R without any interprocessor communication. There are two ways to send the resulting updates to the other processors:

- send factor columns, as above;
- send aggregate update columns associated with the domain R (each aggregate update is sent to only one processor).

We will refer to this second variant of the fan-out algorithm as the *domain fan-out* algorithm (cf. [3], [22]).

Algorithm 2 Fan-out Distributed Factorization

```
mycols = {i | map[i] = myname} ;
for j ∈ mycols do
   L_{*j} := (a_{jj}, ..., a_{nj})^T ;
for column j := 1 to n do
   if j ∈ mycols then
     begin
       while not all updates to column j have been applied do
         begin
           Receive a factor column L_{*k} ;
           for each i > k with l_{ik} ≠ 0 and map[i] = myname do
              L_{*i} := L_{*i} − l_{ik}(l_{ik}, ..., l_{nk})^T ;          (2)
         end ;
       L_{*j} := (1/√l_{jj}) L_{*j} ;
       Send L_{*j} to each processor π for which there exists i > j
         such that l_{ij} ≠ 0 and map[i] = π ;
     end
```

The computation proceeds in two phases. In the first phase, each domain is factored as before, except that no factor columns are sent to other processors. When the root j of the defining subtree for a domain is reached, the processor computes an aggregate update column corresponding to each column $i > j$ for which $l_{ij} \neq 0$, and sends it to processor $map[i]$ (again, for simplicity we have allowed processors to send aggregate updates to themselves). Finally all of the updates are applied. In the second phase, the algorithm proceeds as before.

Note that aggregate update columns are used to communicate between domain and non-domain nodes, and factor columns are used to communicate among non-domain nodes. As we will see in §7, the amount of communication is significantly reduced.

5 The Fan-in Distributed Algorithm

The fan-in algorithm [1] is a distributed scheme for sparse factorization in which all communication is through aggregate update columns. In essence, the complete update column for column j is written as a sum of aggregate update columns (each corresponding to a different processor π):

$$\sum_{k \in Struct(L_{j*})} l_{jk}(l_{jk}, \cdots, l_{nk})^T = \sum_{\substack{\pi \\ row[j,\pi] \neq \emptyset}} u[j,\pi] \,,$$

Algorithm 3 Fan-in Distributed Factorization

```
mycols = { j | map[j] = myname } ;
for column j := 1 to n do
   if row[j, myname] ≠ ∅ or j ∈ mycols then
    begin
     t := 0 ;
     for k ∈ row[j, myname] do
        t := t + l_jk (l_jk, ···, l_nk)^T ;                       (3)
     if j ∉ mycols then
        Send aggregate update column t to processor map[j]
     else
       begin
        L_*j := (a_jj, ···, a_nj)^T − t ;
        while not all aggregate updates have been received do
          begin
           Receive aggregate update column u[j, π] for column j ;
           L_*j := L_*j − u[j, π] ;                                (4)
          end ;
        L_*j := L_*j / sqrt(l_jj)
       end
    end
```

where

$$u[j,\pi] = \sum_{k \in row[j,\pi]} l_{jk}(l_{jk},\cdots,l_{nk})^T$$

and

$$row[j,\pi] = \{k \in Struct(L_{j*}) \mid map[k] = \pi\} .$$

All of the update columns appearing in $u[j,\pi]$ come from factor columns that are mapped to processor π, and thus $u[j,\pi]$ can be computed without any interprocessor communication. These aggregate updates are then sent to the processor $map[j]$ to which column j is assigned, and that processor subtracts the aggregate updates from A_{*j} and scales to obtain L_{*j}. Our implementation of fan-in, Algorithm 3, works for any mapping and network topology. Since only aggregate update columns are sent between processors, the incorporation of domains is implicit.

Most of the floating-point computations are in the assignments (3), which is a sparse AXPY, and (4), which is a sparse vector add. But most importantly, each is part of a sequence of sparse vector operations to the same destination vector, so that the expansion need be done only once for each j. Thus we would expect the kernels for fan-in to be more efficient than the kernel for fan-out. Moreover, the fan-in algorithm can be implemented in terms of supernodal updates, in which case

most of the assignments (3) can be implemented as dense AXPY's, further improving performance (see [4]).

Algorithm 3 proceeds one column at a time. If a processor working on column j has to wait for some aggregate update column from another processor, then this formulation keeps it idle even though it could be doing useful work on other columns. This bottleneck can be avoided by allowing blocked processors to compute updates to later columns while they wait. This *compute-ahead* strategy further enhances the performance (see [2]).

6 The Distributed Multifrontal Algorithm

The multifrontal method [21, 8] has been implemented on both shared memory [5, 6, 7] and distributed memory [16, 18] machines. Indeed, the notion of multiple fronts conveys the idea of performing independent elimination from many different "frontiers" in the associated graph. In this section we express this algorithm in the same terms that were used to describe the fan-out and fan-in schemes, and then relate this to the more standard description in terms of frontal matrices.

If S is a separator and $j \in S$, then we can write the complete update for column j as

$$v[j,S] \equiv \sum_{k \in Struct(L_{j*})} l_{jk}(l_{jk}, \cdots, l_{nk})^T = \sum_{S' \text{ child of } S} v[j,S'] + \sum_{\substack{k<j \\ k \in S}} l_{jk}(l_{jk}, \cdots, l_{nk})^T$$

where

$$v[j,S'] \equiv \sum_{k \in T[S']} l_{jk}(l_{jk}, \cdots, l_{nk})^T$$

and $T[S']$ denotes the subtree of the elimination tree rooted at the highest node in the separator S'. The first term is a sum of aggregate updates; the second is a sum of updates computed from factor columns within the separator. Moreover, we can express the aggregate update $v[j,S']$ recursively as

$$v[j,S'] = \sum_{S'' \text{ child of } S'} v[j,S''] + \sum_{\substack{k<j \\ k \in S'}} l_{jk}(l_{jk}, \cdots, l_{nk})^T \tag{5}$$

which is again a sum of aggregate updates and updates computed from factor columns within a separator. Finally, we see that if $j \in S$, then $v[j,S]$ also satisfies equation (5).

Algorithm 4 is our implementation of the distributed multifrontal method; it works for any mapping and network topology. Note that if $v[j,S] \neq 0$, then the processor $map[j,S]$ is assigned to compute $v[j,S]$. Thus $map[j,S] = map[j]$ for $j \in S$.

The multifrontal method is normally expressed in terms of the frontal matrix associated with each separator S. The aggregate updates $v[j,S]$ all have the same nonzero structure as $(l_{jk}, \cdots, l_{nk})^T$ for any $k \in S'$, $k < j$. Thus, if we delete the zero rows, then we can gather these aggregate updates into a single lower triangular update

Algorithm 4 Distributed Multifrontal Factorization

$mycols = \{ j \mid map[j, S'] = myname$ for some separator $S' \}$;
for $j \in mycols$ **do**
  $L_{*j} := (a_{jj}, ..., a_{nj})^T$;
for column $j := 1$ **to** n **do**
  if $j \in mycols$ **then**
   begin
    while not all updates to some $v[j, S']$ for which
      $map[j, S'] = myname$ have been applied **do**
     begin
      Receive aggregate update $v[i, S'']$ or factor column L_{*k} ;
      if received $v[i, S'']$ **then**
       begin
        Let $S' = parent[S'']$;
        $v[i, S'] := v[i, S'] + v[i, S'']$; (6)
        if all updates to $v[i, S']$ have been applied **then**
          Send $v[i, S']$ to $map[i, parent[S']]$;
       end
      else
        for $i \in S'$ with $l_{ik} \neq 0$ and $map[i, S'] = myname$ **do**
         begin
          $v[i, S'] := v[i, S'] + l_{ik}(l_{ik}, \cdots, l_{nk})^T$; (7)
          if all updates to $v[i, S']$ have been applied **then**
            Send $v[i, S']$ to $map[i, parent[S']]$;
         end
     end
    Let S be the separator to which j belongs ;
    if $map[j, S] = myname$ **then**
     begin
      $L_{*j} = L_{*j} - v[j, S]$;
      $L_{*j} := \frac{1}{\sqrt{l_{jj}}} L_{*j}$;
      Send L_{*j} to every processor π for which there exists $i > j$
        such that $l_{ij} \neq 0$ and $map[i, S] = \pi$;
     end
   end

From	To	Pure Fan-out	Domain Fan-out	Fan-in	Domain Multifrontal
Domain	Parent	factor	update	update	update
Domain	Ancestor	factor	update	update	——
Separator	Parent	factor	factor	update	update
Separator	Ancestor	factor	factor	update	——
Separator	Self	factor	factor	update	factor

Table 1: Message types based on domain-separator model

matrix. This update matrix corresponds to the frontal matrix for S and represents the sum of all update contributions from columns in the separator together with descendant columns of the separator in the elimination tree.

Most of the floating-point arithmetic is in the assignment (7), which can be implemented as a dense `AXPY` since $v[i, S']$ and $l_{ik}(l_{ik}, \cdots, l_{nk})^T$ have the same nonzero structure. The assignment (6), which is a sparse `AXPY`, represents a lower order term. Thus we would expect the kernels for multifrontal to be more efficient than the kernels for either fan-out or fan-in (without supernodal updates).

It is straight-forward to incorporate domains into the multifrontal algorithm. Let S denote the root separator for a domain. Then, after the domain has been factored, the aggregate updates $v[j, S]$, $j \notin S$, can be computed and sent to processors $map[j, parent[S]]$. We shall refer to this variant as the *domain multifrontal algorithm.*

In order to minimize communication, the processor $map[j, S]$ assigned to compute $v[j, S]$ is always chosen from the set $Q = \{map[k] | k \in S\}$ in such a manner as to balance the load among the processors in Q.

7 Comparisons of Distributed Algorithms

We have described several distributed factorization schemes in the last three sections. In the pure fan-out scheme, processors communicate using only factor columns. In the domain fan-out scheme, aggregate update columns are used to pass information from domain to non-domain nodes, while factor columns are used to pass information among non-domain nodes. In the fan-in scheme, processors communicate using only aggregate update columns. In the domain multifrontal method, the processors use a mixture of factor columns and aggregate update columns.

In Table 1, we summarize the communication aspects of these schemes. Here "Parent" refers to the separator immediately above the domain or separator in the elimination tree, and "Ancestor" refers to a non-parent separator lying on the path from the domain or separator to the root of the elimination tree. Note that in the multifrontal method, column information is never forwarded from a domain or separator directly to an ancestor separator; it must be passed to the parent separator, then to the grand-parent separator, etc., until it reaches the destination.

No.of Proc.	Distributed Algorithm	Total Messages			Send Mesgs		Recv Mesgs	
		max	min	avg	max	min	max	min
16	Domain Fan-out	711	650	681	372	307	353	323
	Fan-in	238	220	234	119	115	123	101
	Multifrontal	329	298	318	175	142	172	137
32	Domain Fan-out	997	825	906	513	367	484	379
	Fan-in	295	253	272	138	133	159	117
	Multifrontal	348	307	332	186	145	184	145
64	Domain Fan-out	916	614	808	460	344	487	248
	Fan-in	348	246	298	151	144	200	96
	Multifrontal	353	252	309	193	130	186	118

Table 2: Average message counts per processor for the 63-by-63 grid problem

We now present experimental results for a model problem, the sparse symmetric positive definite system associated with a 9-point difference operator on a k-by-k regular grid. The multiprocessor is an Intel iPSC/2 hypercube with Weitek 1167 floating-point chips. The three distributed schemes were implemented in *C*. Results are not presented for the pure fan-out method because it was uniformly and substantially worse than domain fan-out.

The nested dissection ordering was used to order the variables, since it gives optimal-order fill and a well-balanced elimination tree [9]. A subtree-to-subcube mapping [13] was used to assign columns to processors. We chose equal-size domains (one per processor) of largest possible size, since that choice is known to give good load balance and reduced communication [17]. Thus the columns/nodes in the last grid dissector are assigned to the 2^d processors of the entire hypercube, and the nodes associated with each of the two remaining subgrids are mapped to one of the two $(d-1)$-dimensional subcubes in a recursive manner.

For each of the three distributed factorization schemes, the amount of message traffic and the amount of parallel numerical computation depends on the mapping function. Table 2 shows the actual message counts for a 63×63 grid. Table 3 shows the corresponding message volumes. Figure 5 shows the average message volume for a 63-by-63 grid problem, Figure 6 the average message counts, and Figure 7 the efficiency of the domain fan-out, fan-in, and domain multifrontal methods. Table 4 gives speedups for these schemes, relative to a state-of-the-art serial code, also written in *C*.

It is clear from these experimental results that domain fan-out is inferior to the fan-in and the multifrontal schemes for our *current* implementations. The fan-in scheme requires marginally less message traffic than the distributed multifrontal method, but it is not as efficient.

No.of Proc.	Distributed Algorithm	Total Volume			Send Volume		Recv Volume	
		max	min	avg	max	min	max	min
16	Domain Fan-out	604.8	546.6	576.3	317.6	257.3	292.1	284.2
	Fan-in	109.7	104.7	106.2	55.1	51.1	57.6	49.9
	Multifrontal	152.4	137.5	144.6	80.5	63.4	75.7	67.5
32	Domain Fan-out	823.5	675.0	750.7	436.4	306.9	399.0	316.8
	Fan-in	130.0	120.3	126.0	67.0	59.0	70.1	55.4
	Multifrontal	163.2	144.8	154.6	83.9	70.0	81.3	70.2
64	Domain Fan-out	756.5	477.1	647.7	385.6	260.8	402.4	194.2
	Fan-in	155.6	123.3	138.2	73.7	64.3	88.8	52.1
	Multifrontal	163.1	127.4	148.8	90.8	59.2	83.5	58.8

Table 3: Average message volumes per processor (kilobytes) for the 63-by-63 grid problem

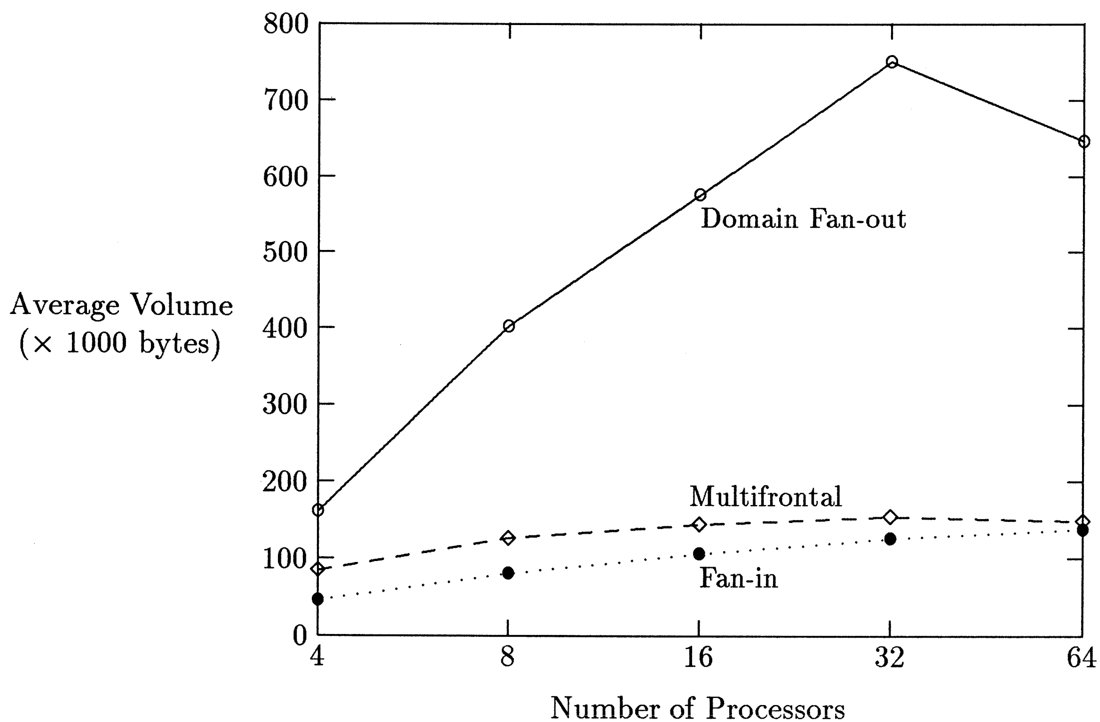

Figure 5: Average message volumes per processor for the 63-by-63 grid problem

No. of Processors	Domain Fan-out	Fan-in	Distributed Multifrontal
8	5.2	6.0	6.5
16	6.8	9.7	10.6
32	6.8	14.9	17.5
64	8.3	20.6	25.6

Table 4: Speedups of distributed schemes on the 63-by-63 grid problem

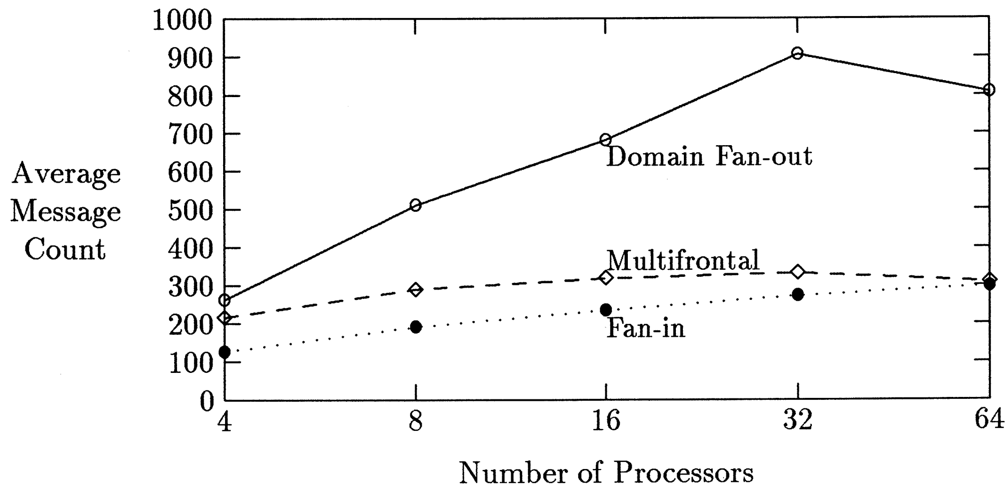

Figure 6: Average message counts per processor for the 63-by-63 grid problem

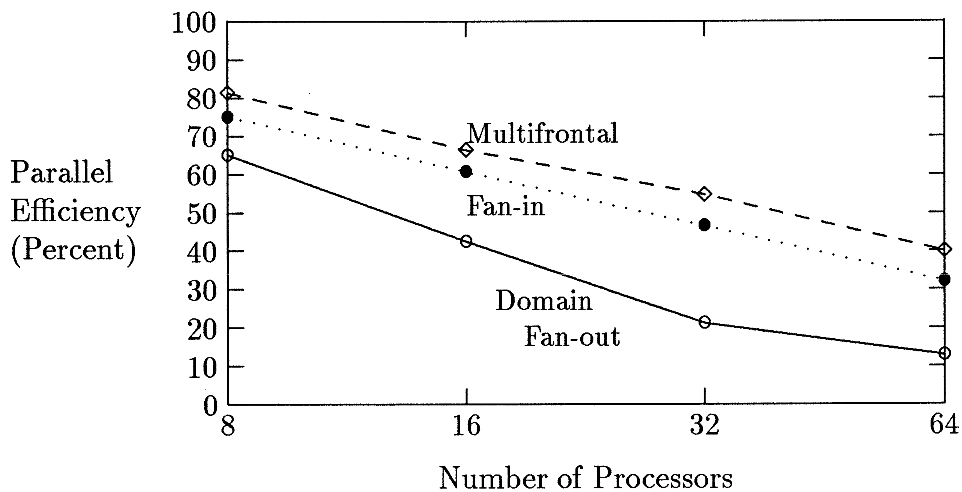

Figure 7: Parallel efficiency for the 63-by-63 grid problem

References

[1] Cleve Ashcraft, Stanley C. Eisenstat, and Joseph W. H. Liu, "A fan-in algorithm for distributed sparse numerical factorization," *SIAM J. Sci. Statist. Comput.*, 11(3):593–599, May 1990.

[2] C. Ashcraft, S. C. Eisenstat, J. W. H. Liu, B. W. Peyton, and A. H. Sherman, "A Compute-Ahead Fan-in Scheme for Parallel Sparse Matrix Factorization," in D. Pelletier, Editor, *4th Canadian Supercomputing Symposium (1990)*, pp. 351–361, June 1990.

[3] Cleve Ashcraft, Stan Eisenstat, Joseph Liu, and Andy Sherman, *A comparison of three distributed sparse factorization schemes*, presented at the SIAM Symposium on Sparse Matrices, Salishan Resort, Gleneden Beach, OR, May 1989.

[4] C. Cleveland Ashcraft, Roger G. Grimes, John G. Lewis, Barry W. Peyton, and Horst D. Simon, "Progress in sparse matrix methods for large linear systems on vector supercomputers," *Internat. J. Supercomputer Appl.*, 1(4):10–30, Winter 1987.

[5] R. E. Benner, G. R. Montry, and G. G. Weigand, "Concurrent multifrontal methods: Shared memory, cache, and frontwidth issues," *Internat. J. Supercomputer Appl.*, 1(3):26–44, Fall 1987.

[6] Iain S. Duff, "Parallel implementation of multifrontal schemes," *Parallel Comput.*, 3(3):193–204, July 1986.

[7] I. S. Duff, N. I. M. Gould, M. Lescrenier, and J. K. Reid, *The Multifrontal Method in a Parallel Environment*, Technical Report CSS 211, Harwell Laboratory, Oxfordshire, England, 1987.

[8] I. S. Duff and J. K. Reid, "The multifrontal solution of indefinite sparse symmetric linear equations," *ACM Trans. Math. Software*, 9(3):302–325, September 1983.

[9] Alan George, "Nested dissection of a regular finite element mesh," *SIAM J. Numer. Anal.*, 10(2):345–363, April 1973.

[10] Alan George, Michael T. Heath, and Joseph Liu, "Parallel Cholesky factorization on a shared-memory multiprocessor," *Linear Alg. and its Applications*, 77:165–187, May 1986.

[11] Alan George, Michael T. Heath, Joseph Liu, and Esmond Ng, "Sparse Cholesky factorization on a local-memory multiprocessor," *SIAM J. Sci. Statist. Comput.*, 9(2):327–340, March 1988.

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

[13] Alan George, Joseph W. H. Liu, and Esmond Ng, "Communication results for parallel sparse Cholesky factorization on a hypercube," *Parallel Comput.*, 10(3):287–298, May 1989.

[14] Jochen A. G. Jess and H. G. M. Kees, "A data structure for parallel L/U decomposition," *IEEE Trans. Comput.*, C-31(3):231–239, March 1982.

[15] J. W. H. Liu, *The Role of Elimination Trees in Sparse Factorization*, Technical Report CS-87-12, Deptartment of Computer Science, York University, 1987. (to appear in *SIAM J. Matrix Anal. and Applic.*).

[16] Robert Francis Lucas, *Solving Planar Systems of Equations on Distributed-Memory Multiprocessor*, Ph.D. thesis, Department of Electrical Engineering, Stanford University, Stanford, California, 1987.

[17] Mo Mu and J. R. Rice, *A Grid Based Subtree-Subcube Assignment Strategy for Solving PDEs on Hypercubes*, Technical Report CSD-TR-869, Computer Sciences Department, Purdue University, February 1989.

[18] Frans J. Peters, "Parallel pivoting algorithms for sparse symmetric matrices," *Parallel Comput.*, 1(1):99–110, January 1984.

[19] Charles H. Romine and James M. Ortega, "Parallel solution of triangular systems of equations," *Parallel Comput.*, 6(1):109–114, January 1988.

[20] Robert Schreiber, "A new implementation of sparse Gaussian elimination," *ACM Trans. Math Software*, 8(3):256–276, September 1982.

[21] B. Speelpenning, *The Generalized Element Method*, Technical Report UIUCDCS-R-78-946, Department of Computer Science, University of Illinois at Urbana-Champaign, November 1978.

[22] Earl Zmijewski, *Limiting Communication in Parallel Sparse Cholesky Factorization*, Technical Report TRCS89-18, Department of Computer Science, University of California at Santa Barbara, June 1989.