Sparse Cholesky Factorization on a Local-Memory Multiprocessor*

Alan George †
Michael T. Heath ††
Joseph Liu ‡
Esmond Ng ††

CS-86-02

January 1986

* Research supported in part by Canadian Natural Sciences and Engineering Research Council under grants A8111 and A5509, by the Applied Mathematical Sciences Research Program, Office of Energy Research, U.S. Department of Energy under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems Inc., and by the U.S. Air Force Office of Scientific Research under contract AFOSR-ISSA-85-00083.
† Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1
†† Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831
‡ Department of Computer Science, York University, Downsview, Ontario M3J 1P3

Sparse Cholesky Factorization on a Local-Memory Multiprocessor

Alan George

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

Michael T. Heath

Mathematical Sciences Section
Oak Ridge National Laboratory
Oak Ridge, Tennessee

Joseph Liu

Department of Computer Science
York University
Downsview, Ontario, Canada

Esmond Ng

Mathematical Sciences Section
Oak Ridge National Laboratory
Oak Ridge, Tennessee

*Research supported in part by Canadian Natural Sciences and Engineering Research Council under grants A8111 and A5509, by the Applied Mathematical Sciences Research Program, Office of Energy Research, U.S. Department of Energy under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems Inc., and by the U.S. Air Force Office of Scientific Research under contract AFOSR-ISSA-85-00083.

ABSTRACT

This article deals with the problem of factoring a large sparse positive definite matrix on a multiprocessor system. The processors are assumed to have substantial local memory but no globally shared memory. They communicate among themselves and with a host processor through message passing. Our primary interest is in designing an algorithm which exploits parallelism, rather than in exploiting features of the underlying topology of the hardware. However, part of our study is aimed at determining, for certain sparse matrix problems, whether hardware based on the binary hypercube topology adequately supports the communication requirements for such problems. Numerical results from experiments running on a multiprocessor simulator are included.

Table of Contents

1. Introduction

This article deals with the problem of factoring a large sparse positive definite matrix A on a multiprocessor system. It is assumed that the system supports message passing among individual processors, and that each processor has a substantial amount of local memory. We assume also that there is no globally shared memory. These assumptions are appropriate for a number of recent commercially available machines, such as the binary hypercube multiprocessors marketed by Ametek, Intel and NCUBE corporations. In [7], a parallel algorithm was developed for solving dense positive definite systems on such machines, so this article can be regarded as a sequel to that work, in which the sparsity of the problem is addressed and exploited.

The process of solving large sparse positive definite systems typically involves four distinct steps:

1 (*Ordering*) Find a good ordering P for A. That is, a permutation matrix P so that PAP^T has a sparse Cholesky factor L. This is usually referred to as the *ordering problem*.

2 (*Symbolic factorization*) Determine the structure of the Cholesky factor L of PAP^T, and set up a data structure for this factor.

3 (*Numerical factorization*) Place the elements of A into the data structure, and then compute L.

4 (*Triangular solution*) Using the computed L, solve the triangular systems $Ly=Pb$, $L^Tz=y$, and then set $x=P^Tz$.

The problems of implementing an ordering algorithm and performing the symbolic factorization procedure on a multiprocessor machine are major projects that will be considered in a subsequent article. In this paper we develop and test a parallel algorithm for step 3 only.

Before proceeding with the description and details of the algorithm, some general remarks about the design and implementation of parallel algorithms should be made. First, it should be kept in mind that the objective is *speed-up*. That is, given a p-processor machine, we would like to solve our problem in time that is as close as possible to a factor of p less than that needed to solve the same problem on a single processor version of the machine, using the best serial algorithm available. Of course in the latter case we assume that the single processor machine has adequate memory, presumably much more than the amount available to a single processor in the multiprocessor configuration.

There is a tendancy to focus on *processor utilization* in studying parallel algorithms. However, while high processor utilization is a *necessary* condition for good speed-up, it is clearly not sufficient; the processors have to be doing *useful work.* Thus, in order to achieve our objective, it is necessary to be able to distribute the computation approximately uniformly across the processors, and to identify sufficient parallelism so that most of the computations can be performed simultaneously.

Let us assume that we are able to achieve this distribution. Except in unusual circumstances, some communication among the processors will be required during the computation. This leads us to another important point about *communication traffic.*

Ideally, every processor in the system should be able to send a message *directly* to any other processor. However, for large p, economics make building machines with such a capability infeasible, so most local-memory multiprocessors provide actual physical communication links among only a few nearest neighbors in some geometric layout. (Common topologies include the ring, the two-dimensional regular grid and the binary hypercube.) A consequence is that a message to be sent from processor i to processor j may have to traverse several physical links, and be *forwarded* by processors along the transmission path.

It is therefore useful to distinguish between *logical* and *physical* data traffic. By the logical traffic from processor i to processor j, we mean the amount of data originated from processor i that must be received and utilized by processor j. On the other hand, we use physical traffic from i to j to refer to the total amount of data traffic that actually flows on the physical link (assuming it exists) from processor i to j in the multiprocessor network. If there is no direct link between processors i and j, the amount of physical traffic will always be zero even if there is some logical data traffic between them. In this case, data originated from processor i and required by processor j has to travel through one or more intermediate processors in some transmission path before reaching j.

It is clear that logical traffic is determined by the way in which the total computation has been distributed across the processors, and physical traffic further depends on the underlying hardware topology and routing strategies. Loosely speaking, logical traffic is a function of the algorithm only, while physical traffic is a function of both the algorithm and the hardware.

2. Sparse Cholesky Factorization

2.1. Dense Case: the Basic Algorithm

We begin by providing a column-oriented version of the basic Cholesky factorization algorithm, described in the following algorithmic form.

```
for j := 1 to n do
begin
    for k := 1 to j−1 do
        for i := j to n do
            a_ij := a_ij − a_ik *a_jk
    a_jj := √a_jj
    for k := j+1 to n do
        a_kj := a_kj/a_jj
end
```

It is shown in [8] that this form of Cholesky factorization, the so-called column-Cholesky formulation, is particularly well suited to medium- to coarse-grain parallel implementation. It was found to have the best combination of work-load balance and overlapped execution in the outer loop sub-tasks. This version is implemented for shared-memory multiprocessors in [8], and for various local-memory architectures supporting message passing in [7,12].

Following [8], we let $Tcol(j)$ be the *task* that computes the j-th column of the Cholesky factor. Each such task consists of the following two types of subtasks:

1. $cmod(j,k)$: modification of column j by column k $(k<j)$;
2. $cdiv(j)$: division of column j by a scalar.

Thus, in terms of these sub-tasks, the basic algorithm can be expressed in the following condensed form.

```
for j := 1 to n do
begin
    for k := 1 to j–1 do
        cmod(j,k)

    cdiv(j)
end
```

We now consider the potential for parallelism in the above formulation of the algorithm. We implicitly assume throughout this paper that the *cmod* and *cdiv* operations are atomic in the sense that we do not attempt to exploit parallelism within them, although such exploitation is clearly possible.

Note first that $cdiv(j)$ cannot begin until $cmod(j,k)$ has been completed for all $k < j$, and column j can be used to modify subsequent columns only after $cdiv(j)$ has been completed. However, there is no restriction on the order in which the *cmod* operations are executed, and *cmod* operations for different columns can be performed concurrently. For example, after $cdiv(1)$ has completed, $cmod(2,1)$ and $cmod(3,1)$ could execute in parallel. These precedence relations are depicted in Fig. 1.

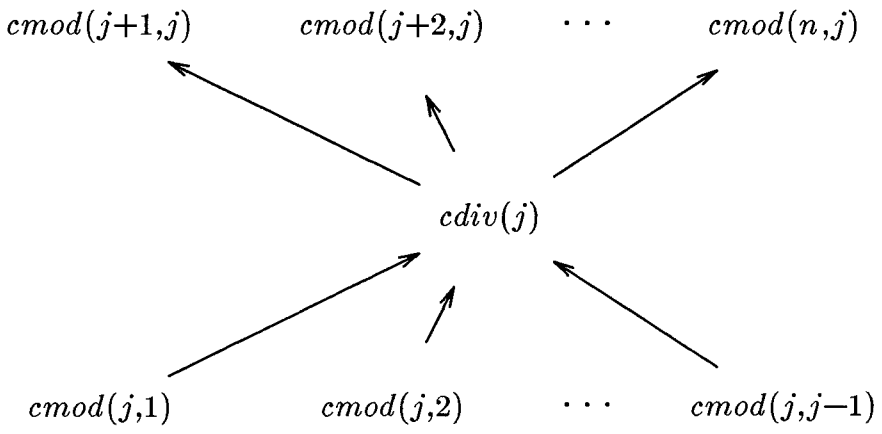

Fig. 1: Subtask precedence graph for column-Cholesky.

2.2. Parallel Sparse Column-Cholesky and the Effect of Ordering

The main difference between the sparse and dense versions of the algorithm stems from the fact that for sparse A, column j may no longer need to be modified by *all* columns $k < j$. Specifically, column j is modified *only* by columns k for which $l_{jk} \neq 0$, and after $cdiv(j)$ has been executed, column j needs to be made available only to tasks $Tcol(r)$ for which $l_{rj} \neq 0$. This can be understood easily by examining the basic form of the algorithm displayed at the beginning of section 2.1. If $a_{jk}=0$, it is obviously unnecessary to execute the loop on i, since it has no effect.

Ideally, we would like to choose an ordering for the matrix A which achieves a number of objectives. First, just as in the use of serial machines, we would like to preserve sparsity and obtain a low arithmetic operation count. In addition, the ordering should allow a high degree of parallelism, and allow the distribution of the computation across the processors in a way that allows the parallelism to be exploited without requiring an inordinate amount of communication.

Fortunately, these objectives turn out to be mutually complementary. In order to gain insight into this problem, it is useful to introduce the notion of elimination trees for sparse Cholesky factors [3,15].

Consider the structure of the Cholesky factor L. For each column $j \leq n$, if column j has off-diagonal nonzeros, define $\gamma[j]$ by

$$\gamma[j] \;=\; \min \left\{ i \;\middle|\; l_{ij} \neq 0,\; i > j \right\} \quad ;$$

that is, $\gamma[j]$ is the row subscript of the first off-diagonal nonzero in column j of L. If column j has no off-diagonal nonzero, we set $\gamma[j]=j$. (Hence $\gamma[n]=n$.)

We now define an *elimination tree* corresponding to the structure of L. The tree has n nodes, labelled from 1 to n. For each j, if $\gamma[j]>j$, then node $\gamma[j]$ is the *parent* of node j in the elimination tree, and node j is one of possibly several *child* nodes of node $\gamma[j]$. We assume that the matrix A is *irreducible*, so that n is the only node with $\gamma[j]=j$ and it is the *root* of the tree. Thus, for $1 \leq j < n$, $\gamma[j] > j$. (If A is reducible, then the elimination tree defined above is actually a forest which consists of several trees.) There is exactly one *path* from each node to the root of the tree. If node i lies on the path from node j to the root, then node i is an *ancestor* of node j, and node j is a *descendant* of node i.

An example to illustrate the notion of elimination trees is provided by the structure of the Cholesky factor shown in Fig. 2, with the associated elimination tree being shown in Fig. 3. Elimination trees have been used either implicitly or explicitly in numerous articles dealing with sparse symmetric factorization [1,2,3,5,6,11,13,15,16,17,18,19]. In particular, the paper [17] uses the elimination tree

as a model to study the parallel sparse Cholesky factorization algorithm in a shared-memory multiprocessor. In addition, Duff [2] is exploring the use of elimination trees in the parallel implementation of multifrontal methods.

$$L = \begin{pmatrix} x & & & & & \\ & x & & & & \\ x & & x & & & \\ & & x & x & & \\ x & x & x & x & x & \\ & x & & x & x & x \end{pmatrix} .$$

Fig. 2: Structure of a Cholesky factor.

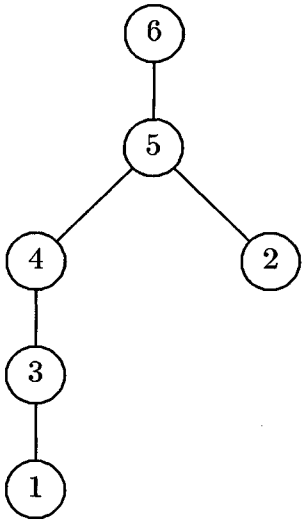

Fig. 3: The elimination tree associated with the Cholesky factor in Fig. 2.

The elimination tree provides precise information about the column dependencies. Specifically, $cdiv(i)$ cannot be executed until $cdiv(j)$ has completed for all descendant nodes j of node i.

The elimination tree has simple structure that can be economically represented using γ, as shown in Fig. 4. Thus, the representation requires only a single vector of size n.

j	1	2	3	4	5	6
$\gamma[j]$	3	5	4	5	6	6

Fig. 4: Computer representation of the tree of Fig. 3.

In order to see the role that elimination trees might play in identifying parallelism, we now consider two different orderings of the same problem, and study their corresponding elimination trees. Consider a 3 by 3 grid problem, where the 9 vertices of the grid are numbered in some manner, and the associated matrix A has the property that $a_{ij} \neq 0$ if and only if vertex i and vertex j are associated with the same small square in the grid. Two different orderings of the grid are given in Fig. 5, the associated Cholesky factors are displayed in Fig. 6, and their corresponding elimination trees are shown in Fig. 7.

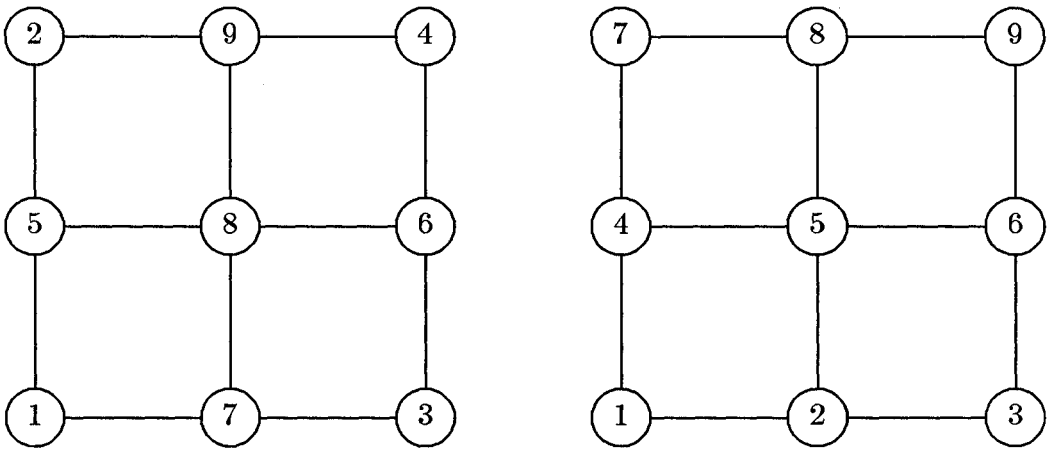

Fig. 5: Two orderings of a 3 by 3 grid.

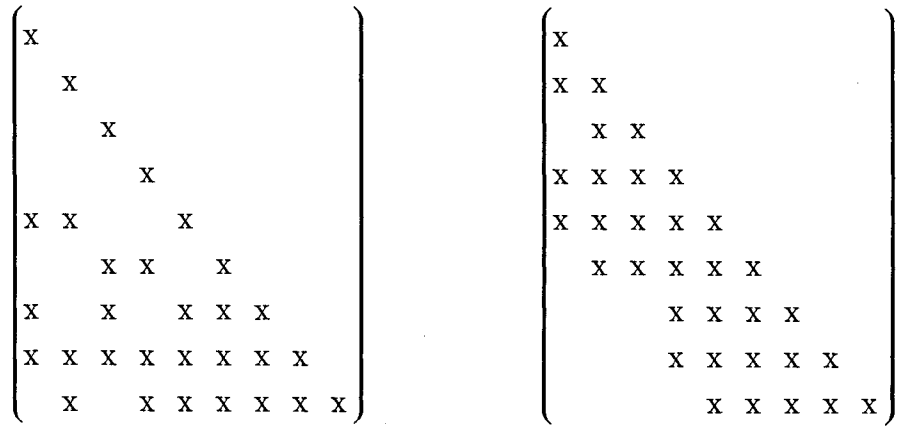

Fig. 6: Structure of the Cholesky factors for the orderings of Fig. 5.

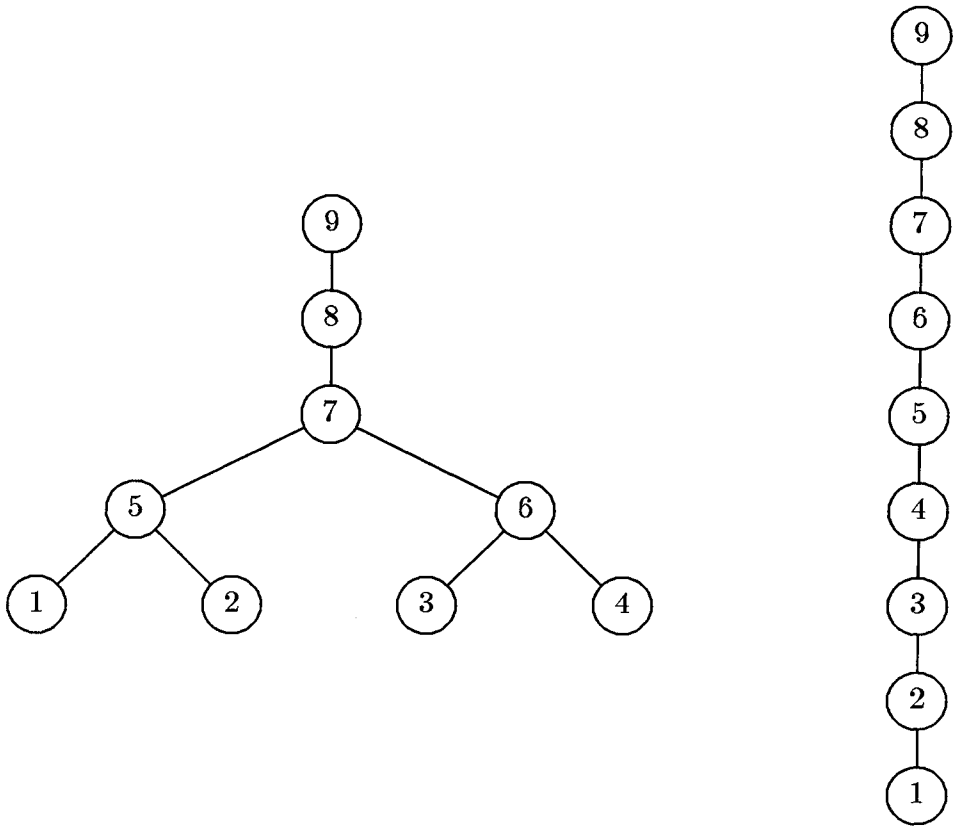

Fig. 7: The elimination trees associated with the matrices in Fig. 6.

The elimination tree on the left is typical of those generated by orderings that are good in the sense of yielding low fill and low operation counts. Its tree structure is short and wide, and such trees and their associated orderings lend themselves well to parallel computation. For example, it should be clear that $Tcol(1)$, $Tcol(2)$, $Tcol(3)$, and $Tcol(4)$ can start immediately in parallel. Moreover, when they have completed execution, $Tcol(5)$ and $Tcol(6)$ may proceed independently. The remaining tasks are no different than those for a dense matrix, and the findings in [7] apply equally well here.

On the other hand, the band-oriented ordering shown above is undesirable because it imposes the same serial execution on the *cdiv* operations that is imposed in the dense case. Moreover, the operation counts and fill-in are inferior to that of the first ordering.

In the elimination tree, if node i and node j belong to the same level of the tree, it is clear that the tasks $Tcol(i)$ and $Tcol(j)$ can be performed independently so long as the tasks associated with their descendant nodes have all been completed. In order to gain high processor utilization, it is therefore desirable to assign, if possible, nodes on the same level of the tree to different processors. An overall task assignment scheme will then correspond to assigning the $Tcol(i)$ tasks to successive processors in a breadth-first bottom-up manner from nodes of the elimination tree.

It should be pointed out that some of the practical fill-reducing orderings will already order the nodes of the elimination tree in this desirable sequence. They include the recent implementation of the minimum degree ordering using multiple elimination [14] and some version of the nested dissection ordering [9]. In such cases, the task assignment scheme corresponds to the straightforward wrap-around assignment, where task $Tcol(i)$ will be assigned to the processor s, given by $s = (i-1) \ mod \ p$.

3. Design and Implementation

In this section, we consider the design and implementation of a sparse Cholesky factorization algorithm appropriate for a parallel multiprocessor with local memory. Let A be the given n by n sparse symmetric positive definite matrix with Cholesky factor L. We assume that the matrix has already been permuted by some fill-reducing ordering appropriate for parallel elimination.

As before, we let $Tcol(j)$ be the task of computing the j-th column of the sparse Cholesky factor L. This task consists of the two types of subtasks: $cmod(j,k)$ and $cdiv(j)$.

In the sparse case, the task $Tcol(j)$ can be expressed in the following algorithmic form:

```
for each k with nonzero l_jk and j>k do
    cmod(j,k)
cdiv(j)
```

It should be clear that the number of *cmod* operations required in the task $Tcol(j)$ is given by the number of off-diagonal nonzeros in the j-th row of L. To facilitate our discussion, we introduce the vector $nmod[*]$, where the value $nmod[j]$ is the number of column modifications *cmod* required in the execution of $Tcol(j)$. This vector can be obtained by simply counting the number of off-diagonal nonzeros in each row of L.

Consider the symmetric factorization of A in a given parallel message-passing multiprocessor environment. Let p be the number of processors in the parallel machine. We assume that an assignment of the column tasks $Tcol(*)$ to the computational nodes of the multiprocessor has been given. For definiteness, let $map[*]$ be the mapping of these n tasks into the p processors. That is, $map[j]$ will be the processor that is responsible for the performance of the task $Tcol(j)$, and hence the computation of column j of L. It should be pointed out that the effect of task-to-processor assignment on load balancing and communication cost can be studied by choosing different $map[*]$ functions.

In the parallel environment, we further assume that there are two primitives: *send* and *await*. Execution of a *send* does not cause the sending process to wait for a reply. On the other hand, execution of an *await* causes the process executing it to be suspended until the message is received. Messages that arrive at the destination process before the execution of the receiving *await* are placed in a queue until needed.

We shall now describe, in an algorithmic form, the work to be performed by the host and node processors. Each node processor uses a *multisend* routine, which will be discussed later in detail.

HOST processor:

Determine the mapping function $map[*]$
for $s := 1$ to p do /* broadcast $map[*]$ */
send $map[*]$ to processor s

Determine the $nmod[*]$ function
for $j := 1$ to n do
send column j of A and $nmod[j]$ to processor $map[j]$

repeat n times do
await a column of L and store it into the data structure

NODE processor s:

```
await map[*] from the host
compute ncol (using map), the number of columns to be processed by processor s

/* obtain columns from the host and eliminate if possible */
repeat ncol times do
begin
   await a column j of A and nmod[j] from the host
   if nmod[j] = 0 then
   begin
      cdiv(j)
      multisend(j, L*j)
   end
end

ncol := ncol– number of columns received with zero nmod

/* main loop: driven by the incoming columns */
while ncol > 0 do
begin
   await a column of L, say L*k
   for each offdiagonal nonzero ljk with map[j] = s do
   begin
      cmod(j,k)
      nmod[j] := nmod[j]–1
      if nmod[j] = 0 then
      begin
         cdiv(j)
         multisend(j, L*j)
         ncol := ncol–1
      end
   end
end
```

It is clear that the host processor is merely responsible for the initiation of the tasks by sending the relevant information to each node processor, and then for the collection of the computed columns of the factor matrix L. In each node processor, a routine called *multisend* is used. Its function is to send the column L_{*j} to the host processor and also to all the node processors that require this column for performing modifications.

Specifically, this routine can be formulated as follows.

Subroutine *multisend*(j, L_{*j}):

for each processor d such that for some $i > j$, $l_{ij} \neq 0$ and $map[i] = d$ do
send L_{*j} to processor d

send L_{*j} to the host

It should be emphasized that the routine *multisend* should only send one copy of the column L_{*j} to a processor even though the processor may use this column to modify more than one column in this processor. Furthermore, the routing strategy in the distribution of the column L_{*j} to the processors concerned can be changed by simply coding a new version of *multisend*.

There are a few points worth mentioning in the scheme for each node processor. As soon as a column L_{*j} of L is completely formed, it is immediately sent to the other processors that need this column. This allows an overlapping of column elimination and column input from the host in the *repeat* loop in the algorithm. More importantly, by making columns of L immediately available, this will reduce wait time on node processors.

Note also that the main loop is driven by the incoming columns of L. This implies that the parallel algorithm is working at the granularity level of the subtasks $cmod(j,k)$ and $cdiv(j)$, rather than at the level of the tasks $Tcol(j)$. This is in direct contrast to the serial implementation of the sparse Cholesky method (for example, SPARSPAK[10] or YSMP[4]), where each $Tcol(j)$ is executed and completed in succession.

Another important characteristic of this formulation is that it is independent of the interconnection network topology. In other words, the parallel algorithm as formulated is applicable to any parallel multiprocessor in a message-passing environment. For different processor interconnections, it may be desirable to choose a different task-to-processor mapping function $map[*]$ or a different message routing strategy. But the basic algorithm remains unchanged.

4. Experiments and Conclusions

In the previous sections our discussion has been independent of the interconnection topology of the multiprocessor. Our objective has been to distribute the workload uniformly and to reduce the amount of communication that must be performed. In this section we report some experimental results obtained from an implementation of our algorithm running on a binary hypercube multiprocessor. For background information about hypercube multiprocessors, see [7] and the references contained therein.

In order to test our implementation, and to gain some information on communication traffic, we solved some finite element problems derived from a sequence of L-shaped triangular meshes described in [9]. The ordering used for these problems was an automatic nested dissection ordering produced by the algorithm described in [9]. The $Tcol(i)$ tasks were assigned to the processors in a simple serial wrap-around manner, with no account whatsoever being taken of the underlying topology of the hypercube multiprocessor. Both the ordering and the symbolic factorization phases were done in serial mode. Parallel versions of these algorithms are under development.

Our experiments were conducted using a binary hypercube simulator written by T. H. Dunigan of the Oak Ridge National Laboratory. For details about the simulator, see [7].

Statistics on both the logical and physical communication for one of the problems were collected, as shown in the tables that follow. The results reported are typical of those found in experiments for other problems in the set of nine problems in [9]. The entry in row r and column c of each table is the amount of data traffic from the processor corresponding to row r to the one corresponding to column c. Thus, the entries in the last row of the tables represent traffic from the host processor to the individual node processors.

We have included both communication counts and volume in the statistics. Communication count simply refers to the number of messages sent. Note that a message associated with the nonzeros of a column includes the number of nonzeros, the subscript information and the actual nonzero values. The numbers reported are the total number of bytes transmitted. In the experiments, an integer requires 4 bytes, and a floating point number requires 8 bytes.

	0	1	2	3	4	5	6	7	Host
0	125	121	115	107	111	106	106	100	126
1	108	126	120	108	107	100	100	101	127
2	108	102	125	118	112	105	100	98	126
3	105	105	104	125	120	112	109	104	126
4	99	103	102	99	125	120	113	103	126
5	110	106	98	97	90	125	118	106	126
6	116	112	107	102	100	97	125	117	126
7	118	116	113	101	110	103	98	125	126
Host	127	128	127	127	127	127	127	127	0

Table 1: Logical communication counts for 8 processors and n=1009.

	0	1	2	3	4	5	6	7	Host
0	34068	33576	32916	31656	32412	31776	31716	30972	34116
1	30504	32688	32040	30408	30540	29748	29724	29748	32724
2	30864	30048	32976	32088	31524	30552	29868	29676	33096
3	30768	30828	30720	33312	32640	31500	31260	30528	33420
4	29664	30180	30108	29628	32832	32160	31248	29988	32928
5	30552	30060	29076	28944	27996	32280	31464	30024	32364
6	32028	31656	31032	30252	30216	29688	33168	32088	33240
7	32496	32244	31848	30336	31536	30600	29832	33312	33372
Host	42212	40820	41192	41516	41024	40460	41336	41468	0

Table 2: Logical communication volume for 8 processors and n=1009.

	0	1	2	3	4	5	6	7	Host
0	125	438	423	0	423	0	0	0	126
1	441	126	0	423	0	408	0	0	127
2	438	0	125	426	0	0	415	0	126
3	0	444	435	125	0	0	0	445	126
4	403	0	0	0	125	428	422	0	126
5	0	411	0	0	427	125	0	425	126
6	0	0	437	0	414	0	125	418	126
7	0	0	0	448	0	445	425	125	126
Host	127	128	127	127	127	127	127	127	0

Table 3: Physical communication counts for 8 processors and n=1009.

	0	1	2	3	4	5	6	7	Host
0	34068	125460	124308	0	126876	0	0	0	34116
1	124320	32688	0	120468	0	119760	0	0	32724
2	124596	0	32976	123624	0	0	121620	0	33096
3	0	126336	123684	33312	0	0	0	125928	33420
4	119580	0	0	0	32832	124176	123924	0	32928
5	0	118632	0	0	122712	32280	0	120960	32364
6	0	0	124968	0	121980	0	33168	122724	33240
7	0	0	0	126924	0	126276	122280	33312	33372
Host	42212	40820	41192	41516	41024	40460	41336	41468	0

Table 4: Physical communication volume for 8 processors and n=1009.

There are several noteworthy aspects of the numbers in Tables 1-4. First, observe that the logical communication is quite evenly distributed among all the processors. That is, the algorithm generates about the same amount of traffic between any and every pair of processors.

Entries in the logical communication tables associated with the processor nodes are all nonzero. However, there are a number of zero entries in the physical communication table. Indeed, each zero in the tables (except for the "Host" row and column) means that a physical link does not exist between the two associated processors. For example, there is no direct link between processors 0 and 3. The messages from processor 0 to 3 must be directed through an intermediate processor, processor 1. This will have the effect of increasing the physical traffic from processor 0 to 1 and from processor 1 to 3. This explains why the nonzero entries in the physical communication tables are much larger than the corresponding entries in the logical communication tables.

Furthermore, it is interesting to observe that the actual physical links in the hypercube topology all carry about the same amount of traffic. Thus, it would appear that this particular topology adequately supports the actual (logical) traffic generated by the algorithm, at least for this class of sparse problems.

In order to determine what our implementation achieved in actual speed-up, we ran our code using one processor and eight processors, and in addition we ran the *best* serial code we have available.

A comparison of the times for the serial code and the parallel code with one processor was done to assess the cost incurred in the parallel implementation per se. It is noteworthy that the penalty is quite substantial, in the neighborhood of 25 percent. This is different from experience with solving dense systems on multiprocessors, where the performance of the best serial code and the parallel code running on one processor are comparable. This is to be expected for the dense case, since the parts of the codes where the majority of the computation is done are *identical.* However, serial codes for sparse Cholesky factorization gain important performance advantages through heavy use of *context.* For example, efficient processing and storage of a column depend on rapid and direct access to information about certain selected previous columns. This context is inevitably lost in a parallel implementation, since the columns are distributed among many processors, and the use of such context would almost certainly require prohibitive amounts of communication. Thus, the data structures and computational schemes used in the serial and parallel implementations are quite different.

Another aspect of parallel sparse matrix computations that tend to make them less efficient than their dense counterparts is that the associated messages in sparse parallel implementations tend to be shorter. Since the time required to transmit a message from one processor to another typically involves a fixed startup time plus a cost proportional to the message length, it is desirable for an algorithm to generate a few large messages rather than many small ones. This is much less easy to achieve for

sparse matrix computations than for the corresponding dense problems.

The results of our experiments are contained in Table 5. Note that the "time" reported is artificial. The simulator measures time simply as the number of machine instructions executed, with no distinction being made between the relative cost of executing instructions of different types.

RESULTS ON SPEED-UP					
n	serial	one processor		eight processors	
	time	time	speed-up	time	speed-up
265	719606	1027215	.70	285614	2.52
406	1462056	2005731	.73	484443	3.02
577	2567430	3454271	.74	776278	3.31
778	4022592	5357658	.75	1120536	3.59
1009	6112334	8060091	.76	1591583	3.84

Table 5: Speed-up for one processor and 8-processor configurations.

5. References

[1] I. S. DUFF, "Full matrix techniques in sparse Gaussian elimination", in *Lecture Notes in Mathematics (912)*, ed. G. A. Watson, Springer-Verlag (1982).

[2] I. S. DUFF, "Parallel implementation of multifrontal schemes", Technical Memorandum No. 49, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL (March 1985).

[3] I. S. DUFF AND J. K. REID, "The multifrontal solution of indefinite sparse symmetric linear equations", *ACM Trans. on Math. Software* **9**, pp. 302-325 (1983).

[4] S. C. EISENSTAT, M. C. GURSKY, M. H. SCHULTZ, AND A. H. SHERMAN, "The Yale sparse matrix package, I. the symmetric codes", *Internat. J. Numer. Meth. Engrg.* **18**, pp. 1145-1151 (1982).

[5] S. C. EISENSTAT, M. H. SCHULTZ, AND A. H. SHERMAN, "Applications of an element model for Gaussian elimination", in *Sparse Matrix Computations*, ed. J.E. Bunch and D.J. Rose, Academic Press, pp. 85-96 (1976).

[6] S. C. EISENSTAT, M. H. SCHULTZ, AND A. H. SHERMAN, "Software for sparse Gaussian elimination with limited core storage", in *Sparse Matrix Proceedings*, ed. I.S. Duff and G.W. Stewart, SIAM Press, pp. 135-153 (1979).

[7] G. A. GEIST AND M. T. HEATH, "Parallel Cholesky factorization on a hypercube multiprocessor", Technical Report 6190, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831 (1985).

[8] J. A. GEORGE, M. T. HEATH, AND J. W-H. LIU, "Parallel Cholesky factorization on a multiprocessor", Research Report CS-84-49, Department of Computer Science, University of Waterloo (1984).

[9] J. A. GEORGE AND J. W-H. LIU, "An automatic nested dissection algorithm for irregular finite element problems", *SIAM J. Numer. Anal.* **15**, pp. 1053-1069 (1978).

[10] J. A. GEORGE AND J. W-H. LIU, "The design of a user interface for a sparse matrix package", *ACM Trans. on Math. Software* **5**, pp. 134-162 (1979).

[11] J. A. GEORGE AND J. W-H. LIU, "An optimal algorithm for symbolic factorization of symmetric matrices", *SIAM J. Comput.* **9**, pp. 583-593 (1980).

[12] M. T. HEATH, "Parallel Cholesky factorization in message passing multiprocessor environments", Technical Report ORNL-6150, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831 (1985).

[13] J. A. G. JESS AND H. G. M. KEES, "A data structure for parallel L/U decomposition", *IEEE Trans. Comput.* **C-31**, pp. 231-239 (1982).

[14] J. W-H. LIU, "Modification of the minimum degree algorithm by multiple elimination", *ACM Trans. on Math. Software* **11**, pp. 141-153 (1985).

[15] J. W-H. LIU, "A compact row storage scheme for sparse Cholesky factors using elimination trees", *ACM Trans. on Math. Software*, (1985). (To appear)

[16] J. W-H. LIU, "On general row merging schemes for sparse Givens transformations", *SIAM J. Sci. Stat. Comput.*, (1986). (to appear)

[17] J. W-H. LIU, "Computational models and task scheduling for parallel sparse Cholesky factorization", *Parallel Computing*, (1986). (to appear)

[18] F. J. PETERS, "Sparse matrices and substructures ", Mathematical Centre Tracts 119, Mathematisch Centrum, Amsterdam, The Netherlands (1980).

[19] R. SCHREIBER, "A new implementation of sparse Gaussian elimination", *ACM Trans. on Math Software* **8**, pp. 256-276 (1982).