ORNL/TM-10260

ornl

OAK RIDGE
NATIONAL
LABORATORY

MARTIN MARIETTA

Solution of Sparse Positive Definite Systems on a Shared-Memory Multiprocessor

Alan George
Michael T. Heath
Joseph Liu
Esmond Ng

OPERATED BY
MARTIN MARIETTA ENERGY SYSTEMS, INC.
FOR THE UNITED STATES
DEPARTMENT OF ENERGY

Printed in the United States of America. Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road, Springfield, Virginia 22161
NTIS price codes—Printed Copy: A03; Microfiche A01

ORNL/TM-10260

Engineering Physics and Mathematics Division

Mathematical Sciences Section

SOLUTION OF SPARSE POSITIVE DEFINITE SYSTEMS ON A SHARED-MEMORY MULTIPROCESSOR

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

† Also a member of the Departments of Computer Science and Mathematics
The University of Tennessee
Knoxville, Tennessee 37996-1300

†† Department of Computer Science
York University
Downsview, Ontario, Canada M3J 1P3

Date Published: January 1987

Research was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy, by the U.S. Air Force Office of Scientific Research under contract AFOSR-ISSA-85-00083, and by the Canadian Natural Sciences and Engineering Research Council under grants A8111 and A5509.

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
operated by
Martin Marietta Energy Systems, Inc.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-84OR21400

Table of Contents

SOLUTION OF SPARSE POSITIVE DEFINITE SYSTEMS ON A SHARED-MEMORY MULTIPROCESSOR *

Alan George †

Michael T. Heath †

Joseph Liu ††

Esmond Ng †

ABSTRACT

Algorithms and software for performing sparse Cholesky decomposition and using the Cholesky factors in the solution of sparse symmetric positive definite systems on serial computers have reached a high state of development. In this paper we present algorithms for performing these two phases on a shared-memory multiprocessor computer, along with some numerical experiments demonstrating their performance on a Sequent Balance 8000 system.

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

† Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, Tennessee. The first author is also a member of the Departments of Computer Science and Mathematics, The University of Tennessee, Knoxville, Tennessee.

†† Department of Computer Science, York University, Downsview, Ontario, Canada.

1. Introduction

This article deals with the problem of solving a large sparse positive definite system of equations $Ax = b$ on a shared memory multiprocessor system. In [3], a parallel algorithm was developed for solving dense positive definite systems in such an environment, so this article can be regarded as a sequel to that work, in which the sparsity of the problem is addressed and exploited.

The solution of large sparse positive definite systems typically involves four distinct steps [6]:

1. *Ordering*
   Find a good ordering P for A. That is, find a permutation matrix P so that PAP^T has a sparse Cholesky factor L (i.e., $PAP^T = LL^T$).
2. *Symbolic factorization*
   Determine the structure of the Cholesky factor of PAP^T, and set up a data structure that exploits the sparsity of L.
3. *Numerical factorization*
   Place the elements of PAP^T into the data structure, and then compute L.
4. *Triangular solution*
   Using the computed L, solve the triangular systems $Ly = Pb$, $L^T z = y$, and then set $x = P^T z$.

The problems of implementing an ordering algorithm and a symbolic factorization algorithm on a multiprocessor machine are major projects and will be considered elsewhere. In this paper we develop and test parallel algorithms for steps 3 and 4.

The computing regime we adopt employs the notion of a *pool of tasks* whose parallel execution is controlled by a *self-scheduling discipline* [7]. In our context, the tasks are those computations associated with columns of the coefficient matrix, and thus have a well-defined order associated with them.

In some parallel algorithms, specific tasks are mapped onto specific processors in advance of initiating the computation. In this situation, effective (static) load balancing among the processors requires that the distribution of work be reasonably uniform. Self-scheduling can be regarded as a mechanism for implementing dynamic load balancing; p processes are initiated to perform T tasks ($p \leqslant T$). When a given process completes a task, it checks to see if any unassigned tasks remain, and if so it is assigned the next one. Thus, if a process happens to have drawn a relatively small task, it will become free to perform another one sooner than a process occupied by a larger task. In this way, processors tend to be kept busy even if the tasks vary in their computational requirements.

This self-scheduled pool-of-tasks approach is flexible in that it is not very strongly dependent on the number of processors available. However, its effective use depends upon certain computational and hardware characteristics.

Since the pool of tasks must be made available to each processor, there must be either a significant amount of shared global memory, or very high communication bandwidth among processors. This is particularly important if the tasks involve a substantial amount of data and/or if the computation involved with each task is small compared to the time required for the processor to acquire the task.

This latter issue of granularity is complicated in the context of using the self-scheduling pool-of-tasks approach. On one hand, unless the overhead associated with the initiation of a process is small, one would prefer to have the computation associated with each task quite substantial, so as to amortize the initiation overhead over a large amount of computation. On the other hand, large tasks imply that there will be fewer of them, and the problem of allocating them to processors in a way that will keep them all gainfully employed over time will be more difficult.

In light of the remarks above, we will assume that $T \gg p$, and that the balance between hardware speed and task size is such that the time associated with assigning a task to a processor is low compared to the time it takes to perform that task. Computers for which such assumptions are reasonable include the Cray X-MP, Elxsi, Encore, Flex, and the Sequent, each of which has a moderate number of processors (4-30) and considerable memory, all attached to a very fast bus.

An outline of the paper is as follows. In Section 2 we briefly review the ideas in [3] and some basic results about sparse Cholesky factorization. We then describe a parallel version of the algorithm. In Section 3 we consider the parallel solution of sparse triangular systems. In Section 4 we present numerical experiments which demonstrate the performance characteristics of the algorithms. These were performed on a Sequent multiprocessor, which has a shared-memory architecture. Finally, Section 5 contains our concluding remarks.

Recent work on parallel sparse Cholesky factorization includes the following. In [1], Duff considers the implementation of the multifrontal approach for solving sparse symmetric systems on a shared-memory multiprocessor, and in [4], the authors of this paper have proposed a parallel sparse Cholesky factorization algorithm for a distributed-memory multiprocessor.

2. Parallel Sparse Cholesky Factorization

2.1. Parallel Dense Column-Cholesky

In [3], the column-Cholesky formulation is recommended for parallel Cholesky factorization of dense symmetric positive definite linear systems on a shared-memory multiprocessor. Following [3], 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.

We first review the version of dense column-Cholesky in [3]. The self-scheduling of the tasks $Tcol(j)$ is implemented by maintaining a vector of flags *ready*. They can be viewed as a vector of semaphores to ensure synchronization among these column tasks. The value $ready[j]$ indicates whether column j is ready to be used for modification of subsequent columns. The following gives an algorithmic description of the implementation.

```
for j := 1 to n
    ready[j] := 0;
for j := 1 to n
    schedule Tcol(j);
```

The task $Tcol(j)$ can then be implemented as follows.

```
for k := 1 to j-1
begin
    wait until ready[k] = 1;
    do cmod(j,k);
end
do cdiv(j);
ready[j] := 1;
```

It should be clear that the columns of the matrix become *ready* in order of the column subscripts, so that the tasks are completed in the sequence:

$$Tcol(1)\ ,\ Tcol(2)\ ,\ \cdots\ ,\ Tcol(n)\ .$$

We now introduce a different version of parallel dense factorization. It is the basis for the parallel sparse column-Cholesky algorithm to be described in Section 2.2, but is easier to understand. Its purpose here is to facilitate a better

appreciation of the underlying idea for the sparse case.

The new formulation maintains a set of n non-overlapping linked lists, one for each column of the matrix. Since they are non-overlapping, an n-vector *link* will be enough to implement them. In the following discussion, $link^m[j]$ denotes the m-th element in the linked list for column j; for example, $link^3[j] = link[link[link[j]]]$. We assume that the lists are null-terminated, so that the j-th list is given by:

$$link[j], link^2[j], \cdots, link^r[j],$$

where for some r, $link^{r+1}[j] = 0$. These linked lists are often used in the sequential algorithms for sparse Cholesky factorization [10]. Furthermore, we define $next(j,k)$ to be the row subscript of the next nonzero in column k of L immediately beneath L_{jk}. That is, $next(j,k) = j + 1$. (In the dense case, $next(j,k)$ is independent of k, but $next(j,k)$ will depend on both j and k in the sparse case.) The new algorithm is shown below.

```
for j := 1 to n
begin
    link[j] := 0;
    nmod[j] := j-1;
end
for j := 1 to n
    schedule Tcol(j);
```

Each task $Tcol(j)$ is as follows.

```
while nmod[j] > 0 do
begin
    wait until link[j] > 0;
    k := link[j];        /* remove first column k from j-th list */
    link[j] := link[k];

    do cmod(j, k);

    nmod[j] := nmod[j] - 1;
    nextnz := next(j, k);
    if nextnz ≤ n then
    begin        /* add k to next(j, k)-th list */
        link[k] := link[nextnz];
        link[nextnz] := k;
    end
end

do cdiv(j);

nextnz := next(j, j);
if nextnz ≤ n then
begin        /* add j to next(j, j)-th list */
    link[j] := link[nextnz];
    link[nextnz] := j;
end
```

In this version, $nmod[j]$ is used to keep track of the number of column modifications that remain to be performed on column j; and it is initialized to $j-1$. For column j, the linked list:

$$link[j]\ ,\ link^2[j]\ ,\ \cdots$$

gives the columns that are currently ready to modify column j. As each (say, column k) is used for the column modification, it is removed from the linked list for j and passed onto that for $next(j,k)$ (namely, column $j+1$ in the dense case). Moreover, after the $cdiv(j)$ operation, column j is now ready to modify subsequent columns, and it is placed in the linked list for $next(j,j)$ (which is again equal to $j+1$).

Compared to the previous algorithm (using the *ready* vector), this version has several drawbacks. Two n-vectors are required in the current version. More importantly, some form of critical section has to be set up during the update of the *link* vector. This will avoid simultaneous update of the *link* vector by different *Tool* tasks. Such a mechanism is not necessary in the *ready*-version. However, the

changes are important in order to take advantage of the parallelism derived from the sparsity of L.

2.2. Parallel Sparse Column-Cholesky

As we have pointed out in the previous subsection, general sparse matrix packages (on serial machines) often use a variant of the *link* -version described in Section 2.1 for sparse Cholesky factorization; examples are the Yale sparse matrix package [2] and SPARSPAK [5]. One important difference from the dense case is that $next(j,k)$ is no longer always equal to $j+1$. Recall that $next(j,k)$ is the row subscript of the next nonzero in column k of L immediately beneath L_{jk}. Hence, $next(j,k)$ will depend on both j and k in the sparse case. More precisely, $next(j,k)$ depends on the structure of L. Thus column k (after a *cmod*) or j (after a *cdiv*) of L will not be passed to the $(j+1)$-st list in general, but instead will be passed to the list determined by the structure of the Cholesky factor. For simplicity, if L_{jk} is the last nonzero in column k, we put $next(j,k)=n+1$.

The *link* -version for the sparse factorization is similar to that for the dense factorization in Section 2.1. One difference is that $nmod[j]$ should be initialized to be $\eta(L_{j*})$, where $\eta(L_{j*})$ is the number of offdiagonal nonzeros in the j-th row of L. This information can be obtained easily from the structure of L (which is computed in the symbolic factorization phase). Also the function *next* can be evaluated easily if the nonzeros of a column are stored in ascending order of the row subscripts.

Another implementational difference from the dense case is in the execution of $cmod(j,k)$. In the sparse case, the columns L_{j*} and L_{k*} are stored in a compact form. To perform $cmod(j,k)$, we need to unpack the compact form of the column L_{j*} so that modifications from other columns can be done efficiently. This implies that the processor executing the task $Tcol(j)$ requires a local working array of size n to facilitate the column update operations on column j. However the data structure used for storing L in the parallel sparse Cholesky algorithm is the same as that in the sequential sparse Cholesky algorithm, since it is stored in a global memory and is accessible by all processors.

In terms of behavior, the dense and the sparse cases differ in the order the *cdiv*'s are performed. In the dense case, the *cdiv*'s are performed in a sequential order. However, in the sparse case, only a few *cmod*'s have to be applied to column j and $cdiv(j)$ can be performed once all the necessary *cmod*'s have been applied. Thus not only can the *cmod*'s be carried out in parallel in the sparse case, some of the *cdiv*'s may also be completed simultaneously.

In the discussion above, we have not addressed the problem of scheduling the tasks. We have assumed that the column tasks are scheduled according to the ordering of the columns. Recall from Section 1 that the columns of A are ordered at the beginning so as to reduce fill in the Cholesky factor. (In our experiments,

we have used a variant of the minimum degree algorithm [8].) Thus we implicitly assume that such an ordering is appropriate for parallel computation on a shared-memory multiprocessor. In general, because the Cholesky factor is sparse, the completion time of parallel sparse Cholesky factorization will depend on how the column tasks are scheduled. It is possible to reorder the columns of the permuted matrix PAP^T so that the Cholesky factor has the same amount of fill, but when the new ordering is used to schedule the column tasks, the completion time on a shared-memory multiprocessor will be reduced. See [9] for details.

3. Parallel Sparse Triangular Solutions

In this section we consider the parallel solutions of the triangular systems

$$Lv = u \qquad \text{and} \qquad L^T w = v \quad ,$$

where L and u are either given or computed elsewhere. It is assumed that the elements of L are stored column by column. We shall first describe a parallel algorithm for the backward solve and then present two parallel algorithms for the forward solve.

3.1. Parallel Backward Solve

Let $Tw(j)$ denote the task that computes w_j. A parallel algorithm for computing w is given below and it makes use of the vector *ready* described in the previous section.

```
for j := 1 to n
    ready[j] := 0;
for j := n down to 1
    schedule Tw(j);
```

Here each task $Tw(j)$ is given as follows.

```
for each offdiagonal nonzero L_kj in the j-th column, do
begin
    wait until ready[k] = 1;
    do wmod(j, k);
end
do wdiv(j);
ready[j] := 1;
```

The functions $wmod(j,k)$ and $wdiv(j)$ are similar to *cmod* and *cdiv* in the Cholesky factorization respectively. More precisely, $wmod(j,k)$ modifies v_j by w_k using L_{kj}, and $wdiv(j)$ divides v_j by L_{jj} to yield the result w_j. Note that once a component w_j is computed, it can be used to modify any v_i, $i < j$. The flag

ready[j] is used to signal when w_j is available. The parallel algorithm described above is efficient in terms of accessing the elements of L, since L is stored column by column.

When L is dense, the *wmod*'s can be performed in parallel, but the *wdiv*'s are completed sequentially. However, when L is sparse, *wmod*(j,k) is performed only when L_{kj} is nonzero. As a result, not only can the *wmod*'s be carried out in parallel, some of the *wdiv*'s may also be completed simultaneously. Thus the algorithm described above exploits the parallelism inherent in the backward solve and that provided by the sparsity of L.

3.2. Parallel Forward Solve

Now we consider the problem of solving the lower triangular system $Lv = u$ in parallel. We first recall that the elements of L are assumed to be stored column by column. Let $Tv(j)$ denote the task that computes v_j. In the following discussion, *vmod*(j,k) modifies u_j by v_k using L_{jk}, and *vdiv*(j) divides u_j by L_{jj} to yield the result v_j.

We shall consider the dense case first. We note that v_j can be computed only when u_j has been modified by the product of v_i and L_{ji}, $1 \leqslant i \leqslant j-1$. This requires accessing the elements of L by rows. Note that our ultimate goal is to develop a parallel algorithm for sparse forward solve. Since we assume that, when L is sparse, the nonzeros are stored column by column in a compact data structure, it is awkward to access the elements by rows. This difficulty can be alleviated by using the ideas in the *link*-version of the parallel algorithm for Cholesky factorization given in Section 2. That is, the nonzeros in row j of L that will be used to modify u_j are put in a linked list and the linked lists are updated after these nonzeros are used. However, it should be noted that the amount of computing is small in forward solve compared to the overhead (for synchronization purposes and updating the linked lists) involved. Indeed, preliminary experiments on a Sequent multiprocessor have shown that the performance of this approach is poor.

We now describe a second approach for parallel forward solve. It makes use of only *nmod* and it is column-oriented. The algorithm is shown below. (Recall that $\eta(L_{j*})$ denotes the number of offdiagonal nonzeros in row j of L.)

```
for j := 1 to n
    nmod[j] := η(L_{j*});
for j := 1 to n
    schedule Tv(j);
```

Here each task $Tv(j)$ is defined as follows.

```
wait until nmod[j] = 0;
do vdiv(j);
for each offdiagonal nonzero L_kj in column j, do
begin
    do vmod(k,j);
    nmod[k] := nmod[k] - 1;
end
```

In this version, once v_j is computed, it can be used, together with column j of L, to modify the right hand side vector. Thus accessing the elements of L is very efficient.

Obviously, some form of critical section must be set up when performing the *vmod*'s and decrementing *nmod*. This is particularly important in the sparse case since the sparsity of L may cause a component of u to be modified simultaneously by several components of v. Thus, in general, there must be some form of synchronization lock for each component of u. The second algorithm therefore requires the same amount of storage as the *link* version (assuming a synchronization lock and an integer location both require the same amount of space), but the overhead is smaller (since no linked lists have to be maintained). However, the second algorithm suffers from the drawback that it requires n synchronization locks (n is the order of L). There will be a difficulty if the multiprocessor system provides only a small number of such locks.

4. Numerical Results

The numerical experiments were performed on a Sequent Balance 8000, which is a multiprocessor system with shared memory. On our system, there are 8 processors attached to a high speed bus, and these processors share 8M bytes of global memory. The operating system running on the Sequent multiprocessor is Dynix, which is a variant of the Unix operating system. Special library functions are provided for creating multiple cooperating processes and for synchronizing these processes.

The parallel algorithms described in this paper were implemented in FORTRAN and the programs were compiled using the Sequent FORTRAN compiler. The programs have been tested on p processors, where $1 \leqslant p \leqslant 7$ (the eighth processor was reserved for operating system functions). The test problems used in the experiments were matrices defined on a sequence of L-shaped finite element meshes, and these matrices have been reordered using a variant of the minimum degree algorithm [8]. For comparison purposes, we have run SPARSPAK, which is a software package for solving large sparse symmetric positive definite systems on serial machines [5], on a single processor in order to obtain the "sequential" times. The results are provided in Table 4.1.

n	$\|A\|$	Sequential Algorithms		
		Factorization	Forward Solve	Backward Solve
265	1753	1.350	.150	.150
406	2716	2.950	.267	.267
577	3889	4.917	.400	.417
778	5272	8.350	.600	.600
1009	6865	11.700	.817	.817
1270	8668	16.433	1.083	1.083
1561	10681	23.733	1.433	1.400
1882	12904	32.950	1.800	1.783
2233	15337	42.450	2.217	2.167
2614	17980	55.367	2.733	2.650
3025	20833	67.817	3.233	3.117
3466	23896	92.583	3.917	3.800

Table 4.1 : Performance of the sequential algorithms.
(Times are in seconds.)

The results of the multiple-processor experiments are presented in Tables 4.2-4.7. In order to gain some insights into the performance of the parallel algorithms, we have computed the *speedup* ratios σ, which are defined to be

$$\sigma = \frac{\text{time required by the sequential algorithm}}{\text{time required by the parallel algorithm on } p \text{ processors}} .$$

Thus a parallel algorithm on p processors has a good performance if the speedup ratios are close to p.

Table 4.2 contains the execution times (in seconds) and speedup ratios of the parallel sparse numeric factorization algorithm on p processors, for $2 \leqslant p \leqslant 7$. It is worthwhile to point out that the speedup ratios on a fixed number of processors improve as the problem size increases. When $p = 2$, the speedup ratios approach 1.8 for our set of test problems. Thus the efficiency (which is defined to be σ / p) approaches 90%. On 7 processors, the efficiency approaches 77% and we expect it to improve as the problem size increases. Table 4.3 contains the performance statistics of running the parallel sparse numeric factorization algorithm on only 1 processor. This provides an indication of the amount of synchronization overhead in the parallel algorithm. As one can see, the overhead is fairly small. From these results we conclude that our parallel sparse numeric factorization algorithm is capable of producing good efficiencies for large enough problems.

n	Numerical Factorization											
	$p=2$		$p=3$		$p=4$		$p=5$		$p=6$		$p=7$	
	time	σ	time	σ	time	σ	time	σ	time	σ	time	σ
265	.883	1.53	.667	2.02	.567	2.38	.533	2.53	.500	2.70	.500	2.70
406	1.850	1.59	1.317	2.24	1.067	2.76	.933	3.16	.867	3.40	.800	3.69
577	3.017	1.63	2.150	2.29	1.700	2.89	1.500	3.28	1.317	3.73	1.233	3.99
778	5.017	1.66	3.517	2.37	2.767	3.02	2.367	3.53	2.117	3.94	1.950	4.28
1009	7.000	1.67	4.917	2.38	3.867	3.03	3.267	3.58	2.900	4.03	2.667	4.39
1270	9.767	1.68	6.783	2.42	5.350	3.07	4.500	3.65	3.950	4.16	3.600	4.56
1561	13.850	1.71	9.550	2.49	7.500	3.16	6.300	3.77	5.483	4.33	4.950	4.79
1882	18.967	1.74	13.067	2.52	10.217	3.23	8.567	3.85	7.383	4.46	6.700	4.92
2233	24.383	1.74	16.750	2.53	13.017	3.26	10.833	3.92	9.417	4.51	8.467	5.01
2614	31.383	1.76	21.600	2.56	16.650	3.33	13.817	4.01	11.950	4.63	10.717	5.17
3025	38.383	1.77	26.300	2.58	20.367	3.33	16.850	4.02	14.533	4.67	12.983	5.22
3466	51.817	1.79	35.333	2.62	27.200	3.40	22.417	4.13	19.300	4.80	17.167	5.39

Table 4.2 : Performance of the parallel sparse numeric factorization algorithm on p processors. (Times are in seconds).

n	Numerical Factorization	
	$p=1$	
	time	σ
265	1.633	.83
406	3.483	.85
577	5.733	.86
778	9.583	.87
1009	13.400	.87
1270	18.650	.88
1561	26.683	.89
1882	36.667	.90
2233	47.033	.90
2614	60.950	.91
3025	74.467	.91
3466	100.800	.92

Table 4.3 : Performance of the parallel sparse numeric factorization algorithm on 1 processor. (Times are in seconds).

The results for the parallel sparse forward solve algorithm (column-oriented version) are given in Tables 4.4-4.5. Note that the performance of the column-oriented algorithm is fairly good despite the fact that updating a component of the right hand side vector must be done in a synchronized manner. The efficiency is about 45%.

n	Forward Solve											
	$p=2$		$p=3$		$p=4$		$p=5$		$p=6$		$p=7$	
	time	σ	time	σ	time	σ	time	σ	time	σ	time	σ
265	.183	.82	.133	1.13	.117	1.28	.083	1.81	.067	2.24	.083	1.81
406	.317	.84	.217	1.23	.183	1.46	.150	1.78	.133	2.01	.117	2.28
577	.500	.80	.350	1.14	.267	1.50	.217	1.84	.200	2.00	.183	2.19
778	.733	.82	.500	1.20	.400	1.50	.333	1.80	.267	2.25	.233	2.58
1009	1.000	.82	.683	1.20	.533	1.53	.433	1.89	.367	2.23	.333	2.45
1270	1.317	.82	.917	1.18	.700	1.55	.567	1.91	.483	2.24	.433	2.50
1561	1.733	.83	1.183	1.21	.900	1.59	.767	1.87	.633	2.26	.550	2.61
1882	2.200	.82	1.500	1.20	1.150	1.57	.950	1.89	.817	2.20	.717	2.51
2233	2.717	.82	1.850	1.20	1.417	1.56	1.150	1.93	.983	2.26	.850	2.61
2614	3.317	.82	2.250	1.21	1.717	1.59	1.400	1.95	1.200	2.28	1.050	2.60
3025	3.950	.82	2.683	1.20	2.050	1.58	1.683	1.92	1.400	2.31	1.233	2.62
3466	4.767	.82	3.250	1.21	2.483	1.58	2.017	1.94	1.717	2.28	1.483	2.64

Table 4.4 : Performance of the parallel sparse forward solve algorithm (column-oriented version) on p processors. (Times are in seconds).

n	Forward Solve	
	$p=1$	
	time	σ
265	.333	.45
406	.617	.43
577	.950	.42
778	1.433	.42
1009	1.933	.42
1270	2.550	.42
1561	3.350	.43
1882	4.283	.42
2233	5.267	.42
2614	6.433	.42
3025	7.600	.43
3466	9.267	.42

Table 4.5 : Performance of the parallel sparse forward solve algorithm (column-oriented version) on 1 processor. (Times are in seconds).

Tables 4.6 and 4.7 contain the results of the parallel sparse backward solve algorithm. The performance is between those of the factorization and column-oriented forward solve algorithms. We can achieve an efficiency of about 60%. As we can see from Tables 4.3, 4.5 and 4.7, the synchronization overhead in the parallel forward and backward solves is higher than that in parallel numeric factorization. This is because the forward and backward solve algorithms require relatively little computing.

n	Backward Solve											
	$p=2$		$p=3$		$p=4$		$p=5$		$p=6$		$p=7$	
	time	σ	time	σ	time	σ	time	σ	time	σ	time	σ
265	.117	1.28	.083	1.81	.067	2.24	.050	3.00	.050	3.00	.050	3.00
406	.217	1.23	.150	1.78	.117	2.28	.100	2.67	.067	3.99	.083	3.22
577	.300	1.39	.217	1.92	.167	2.50	.133	3.14	.117	3.56	.117	3.56
778	.467	1.28	.333	1.80	.267	2.25	.217	2.76	.183	3.28	.167	3.59
1009	.600	1.36	.433	1.89	.333	2.45	.283	2.89	.250	3.27	.200	4.08
1270	.817	1.33	.567	1.91	.433	2.50	.367	2.95	.333	3.25	.283	3.83
1561	1.033	1.36	.717	1.95	.583	2.40	.483	2.90	.417	3.36	.367	3.81
1882	1.333	1.34	.917	1.94	.717	2.49	.583	3.06	.517	3.45	.450	3.96
2233	1.633	1.33	1.133	1.91	.883	2.45	.733	2.96	.617	3.51	.567	3.82
2614	1.983	1.34	1.367	1.94	1.067	2.48	.900	2.94	.767	3.46	.667	3.97
3025	2.333	1.34	1.617	1.93	1.250	2.49	1.033	3.02	.917	3.40	.817	3.82
3466	2.817	1.35	1.967	1.93	1.517	2.50	1.250	3.04	1.067	3.56	.950	4.00

Table 4.6 : Performance of the parallel sparse backward solve algorithm on p processors. (Times are in seconds).

n	Backward Solve	
	$p=1$	
	time	σ
265	.217	.69
406	.383	.70
577	.583	.72
778	.867	.69
1009	1.150	.71
1270	1.533	.71
1561	2.000	.70
1882	2.550	.70
2233	3.100	.70
2614	3.783	.70
3025	4.467	.70
3466	5.417	.70

Table 4.7 : Performance of the parallel sparse backward solve algorithm on 1 processor. (Times are in seconds).

To summarize our results, we have plotted the speedup ratios versus the problem sizes in Figures 4.1-4.3.

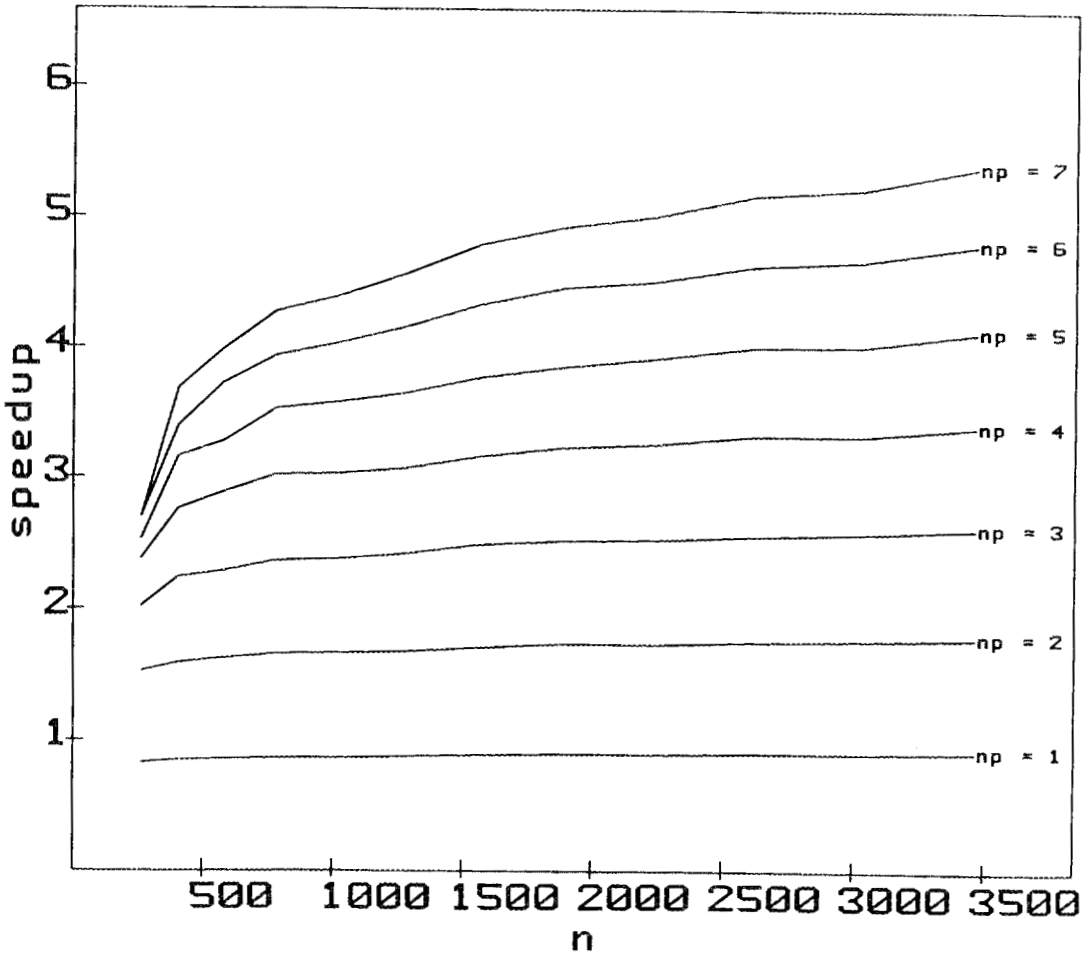

Figure 4.1 : Speedup graph for parallel sparse numeric factorization.

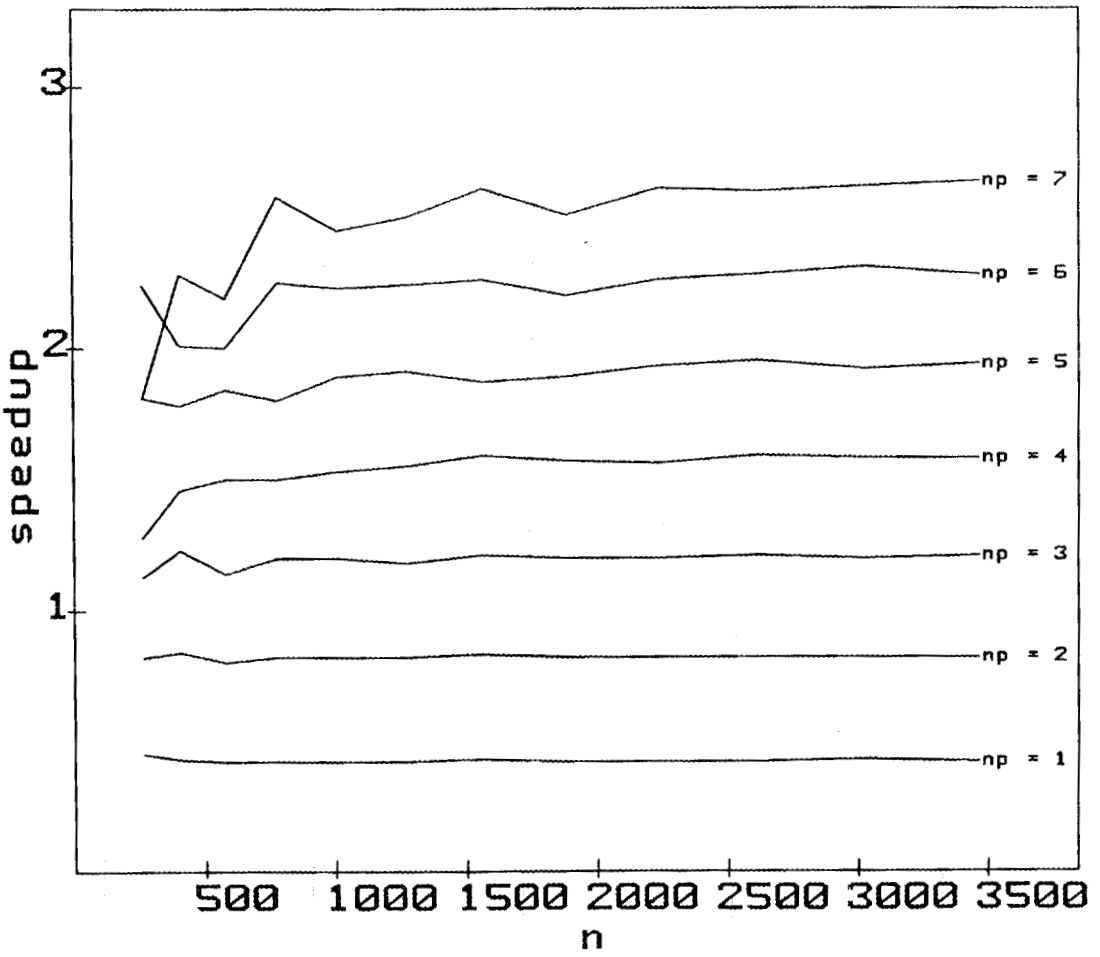

Figure 4.2 : Speedup graph for parallel sparse forward solve.
(Column-oriented version.)

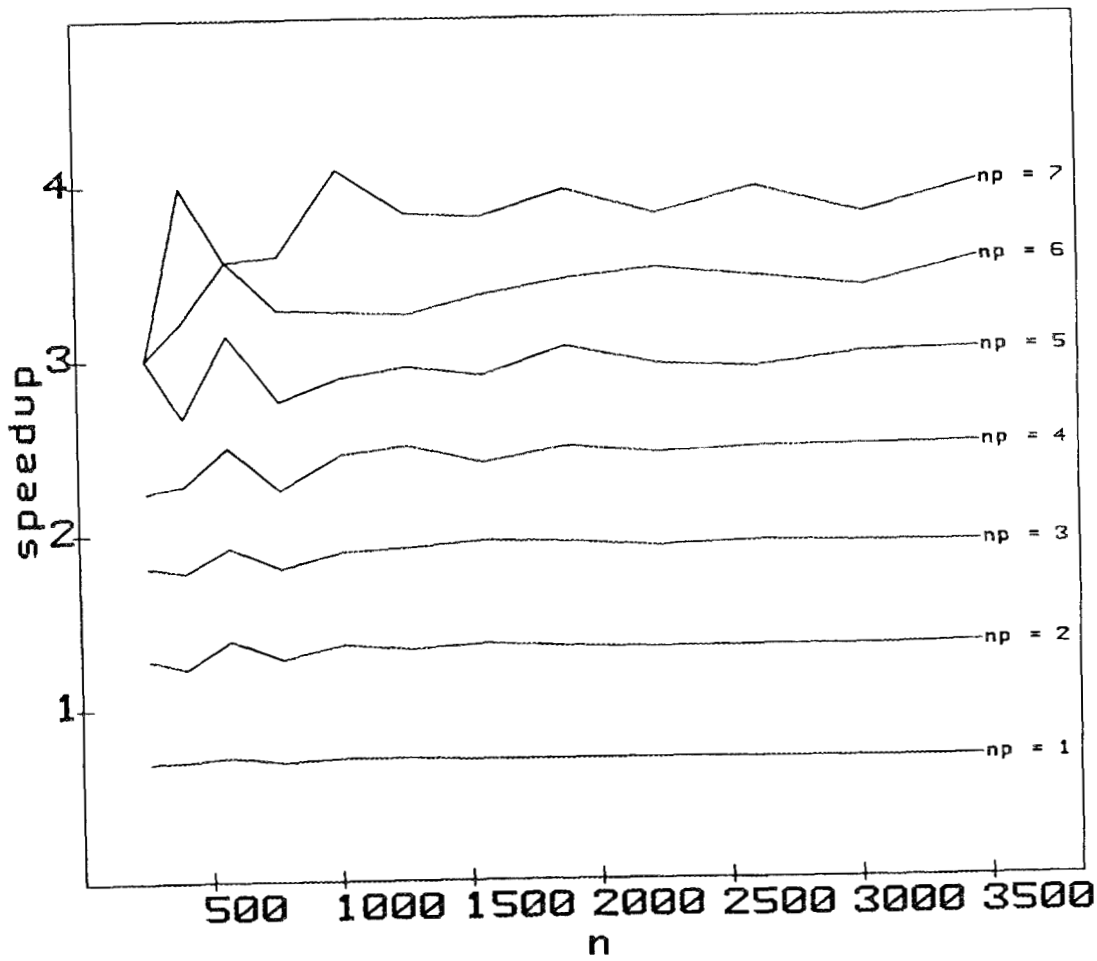

Figure 4.3 : Speedup graph for parallel sparse backward solve.

5. Concluding Remarks

In this paper, we have developed algorithms for sparse Cholesky factorization and triangular solutions that are suitable for multiprocessor systems with shared memory. We have presented numerical experiments performed on a Sequent multiprocessor system to demonstrate the efficiencies of our algorithms. The results indicate that inherent parallelism in the problem and parallelism provided by sparsity can be exploited on shared-memory multiprocessor systems. Good efficiency is achieved in the sparse factorization because the synchronization overhead is small compared to the amount of computing required. However, the efficiencies in the triangular solutions are relatively poor since the amount of computing is small and is approximately the same as the synchronization overhead.

Even though the programs were written for a particular multiprocessor system with shared memory, they can be used with minor changes on any multiprocessor system with shared memory that provides synchronization primitives. Parallel algorithms for ordering and symbolic factorization are under

investigation and the results will be described elsewhere.

6. References

[1] I. S. Duff, "Parallel implementation of multifrontal schemes", *Parallel Computing*, 3 (1986), pp.193-204.

[2] 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.*, 181 (1982), pp.1145-1151.

[3] J. A. George, M. T. Heath, and J. W-H. Liu, "Parallel Cholesky Factorization on a Shared-Memory Multiprocessor", *Linear Algebra and its Appl.*, 77 (1986), pp.165-187.

[4] J. A. George, M. T. Heath, J. W-H. Liu, E. G-Y. Ng, *Sparse Cholesky factorization on a local-memory multiprocessor*, Technical report ORNL/TM-9962, Oak Ridge National Laboratory, Oak Ridge, Tennessee (1986).

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

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

[7] H. F. Jordan, "Experience with pipelined multiple instruction streams", *Proc. IEEE*, 72 (1984), pp.113-123.

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

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

[10] A. H. Sherman, *On the efficient solution of sparse systems of linear and nonlinear equations*, Research Report #46, Dept. of Computer Science, Yale University (1975).

INTERNAL DISTRIBUTION

1. M. V. Denson
2. J. B. Drake
3. E. L. Frome
4. G. A. Geist
5-9. J. A. George
10. L. J. Gray
11-12. R. F. Harbison
13-17. M. T. Heath
18. J. K. Ingersoll
19. F. C. Maienschein
20. T. J. Mitchell
21-25. E. G. Ng
26. G. Ostrouchov
27. C. H. Romine
28. S. Thompson
29. R. C. Ward
30. M. A. Williams
31. D. G. Wilson
32. A. Zucker
33. P. W. Dickson (Consultant)
34. G. H. Golub (Consultant)
35. R. M. Haralick (Consultant)
36. D. Steiner (Consultant)
37. Central Research Library
38. K-25 Plant Library
39. ORNL Patent Office
40. Y-12 Technical Library /Document Reference Station
41. Laboratory Records - RC
42-43. Laboratory Records Department

EXTERNAL DISTRIBUTION

44. Dr. Donald M. Austin, Office of Scientific Computing, Office of Energy Research, ER-7, Germantown Building, U.S. Department of Energy, Washington, DC 20545
45. Dr. Robert G. Babb, Department of Computer Science and Engineering, Oregon Graduate Center, 19600 N.W. Walker Road, Beaverton, OR 97006
46. Dr. Jesse L. Barlow, Department of Computer Science, Pennsylvania State University, University Park, PA 16802
47. Prof. Ake Bjorck, Department of Mathematics, Linkoping University, Linkoping 58183, Sweden
48. Dr. James C. Browne, Department of Computer Sciences, University of Texas, Austin, TX 78712
49. Dr. Bill L. Buzbee, C-3, Applications Support & Research, Los Alamos National Laboratory, P.O. Box 1663, Los Alamos, NM 87545
50. Dr. Donald A. Calahan, Department of Electrical and Computer Engineering, University of Michigan, Ann Arbor, MI 48109

51. Dr. Tony Chan, Department of Computer Science, Yale University, P.O. Box 2158 Yale Station, New Haven, CT 06520

52. Dr. Jagdish Chandra, Army Research Office, P.O. Box 12211, Research Triangle Park, North Carolina 27709

53. Dr. Paul Concus, Mathematics and Computing, Lawrence Berkeley Laboratory, Berkeley, CA 94720

54. Dr. Jane K. Cullum, IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598

55. Dr. George Cybenko, Department of Computer Science, Tufts University, Medford, MA 02155

56. Dr. George J. Davis, Department of Mathematics, Georgia State University, Atlanta, GA 30303

57. Dr. Jack J. Dongarra, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439

58. Dr. Stanley Eisenstat, Department of Computer Science, Yale University, P.O. Box 2158 Yale Station, New Haven, CT 06520

59. Dr. Howard C. Elman, Computer Science Department, University of Maryland, College Park, MD 20742

60. Dr. Albert M. Erisman, Boeing Computer Services, 565 Andover Park West, Tukwila, WA 98188

61. Dr. Geoffrey C. Fox, Booth Computing Center 158-79, California Institute of Technology, Pasadena, CA 91125

62. Dr. Paul O. Frederickson, Computing Division, Los Alamos National Laboratory, Los Alamos, NM 87545

63. Dr. Fred N. Fritsch, L-300, Mathematics and Statistics Division, Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94550

64. Dr. Robert E. Funderlic, Department of Computer Science, North Carolina State University, Raleigh, NC 27650

65. Dr. Dennis B. Gannon, Computer Science Department, Indiana University, Bloomington, IN 47405

66. Dr. David M. Gay, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974

67. Dr. C. William Gear, Computer Science Department, University of Illinois, Urbana, Illinois 61801

68. Dr. W. Morven Gentleman, Division of Electrical Engineering, National Research Council, Building M-50, Room 344, Montreal Road, Ottawa, Ontario, Canada k1A 0R8

69. Prof. Gene H. Golub, Department of Computer Science, Stanford University, Stanford, CA 94305
70. Dr. Joseph F. Grcar, Division 8331, Sandia National Laboratories, Livermore, CA 94550
71. Dr. Don E. Heller, Physics and Computer Science Department, Shell Development Co., P.O. Box 481, Houston, TX 77001
72. Dr. Robert E. Huddleston, Computation Department, Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94550
73. Dr. Ilse Ipsen, Department of Computer Science, Yale University, P.O. Box 2158 Yale Station, New Haven, CT 06520
74. Dr. Harry Jordan, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO 80309
75. Dr. Linda Kaufman, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974
76. Dr. Robert J. Kee, Applied Mathematics Division 8331, Sandia National Laboratories, Livermore, CA 94550
77. Ms. Virginia Klema, Statistics Center, E40-131, MIT, Cambridge, MA 02139
78. Dr. Richard Lau, Office of Naval Research, 1030 E. Green Street, Pasadena, CA 91101
79. Dr. Alan J. Laub, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106
80. Dr. Robert L. Launer, Army Research Office, P.O. Box 12211, Research Triangle Park, North Carolina 27709
81. Prof. Peter D. Lax, Director, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012
82. Dr. Michael R. Leuze, Computer Science Department, Box 1679 Station B, Vanderbilt University, Nashville, TN 37235
83. Dr. Joseph Liu, Department of Computer Science, York University, 4700 Keele Street, Downsview, Ontario, Canada M3J 1P3
84. Dr. Franklin Luk, Electrical Engineering Department, Cornell University, Ithaca, NY 14853
85. Dr. Thomas A. Manteuffel, Computing Division, Los Alamos National Laboratory, Los Alamos, NM 87545
86. Dr. Paul C. Messina, Applied Mathematics Division, Argonne National Laboratory, Argonne, IL 60439
87. Dr. Cleve Moler, Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006

88. Dr. Dianne P. O'Leary, Computer Science Department, University of Maryland, College Park, MD 20742

89. Maj. C. E. Oliver, Office of the Chief Scientist, Air Force Weapons Laboratory, Kirtland Air Force Base, Albuquerque, NM 87115

90. Dr. James M. Ortega, Department of Applied Mathematics, University of Virginia, Charlottesville, VA 22903

91. Prof. Chris Paige, Basser Department of Computer Science, Madsen Building F09, University of Sydney, N.S.W., Sydney, Australia 2006

92. Dr. John F. Palmer, NCUBE Corporation, 915 E. LaVieve Lane, Tempe, AZ 85284

93. Prof. Beresford N. Parlett, Department of Mathematics, University of California, Berkeley, CA 94720

94. Prof. Merrell Patrick, Department of Computer Science, Duke University, Durham, NC 27706

95. Dr. Robert J. Plemmons, Departments of Mathematics and Computer Science, North Carolina State University, Raleigh, NC 27650

96. Dr. John K. Reid, CSS Division, Building 8.9, AERE Harwell, Didcot, Oxon, England OX11 0RA

97. Dr. John R. Rice, Computer Science Department, Purdue University, West Lafayette, IN 47907

98. Dr. Garry Rodrigue, Numerical Mathematics Group, Lawrence Livermore Laboratory, Livermore, CA 94550

99. Dr. Donald J. Rose, Department of Computer Science, Duke University, Durham, NC 27706

100. Dr. Ahmed H. Sameh, Computer Science Department, University of Illinois, Urbana, IL 61801

101. Dr. Michael Saunders, Systems Optimization Laboratory, Operations Research Department, Stanford University, Stanford, CA 94305

102. Dr. Robert Schreiber, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180

103. Dr. Martin H. Schultz, Department of Computer Science, Yale University, P.O. Box 2158 Yale Station, New Haven, CT 06520

104. Dr. David S. Scott, Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006

105. Dr. Lawrence F. Shampine, Numerical Mathematics Division 5642, Sandia National Laboratories, P.O. Box 5800, Albuquerque, NM 87115

106. Dr. Danny C. Sorensen, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439

107. Prof. G. W. Stewart, Computer Science Department, University of Maryland, College Park, MD 20742

108. Capt. John P. Thomas, Air Force Office of Scientific Research, Building 410, Bolling Air Force Base, Washington, DC 20332

109. Prof. Charles Van Loan, Department of Computer Science, Cornell University, Ithaca, NY 14853

110. Dr. Robert G. Voigt, ICASE, MS 132-C, NASA Langley Research Center, Hampton, VA 23665

111. Dr. Andrew B. White, Computing Division, Los Alamos National Laboratory, Los Alamos, NM 87545

112. Mr. Patrick H. Worley, Computer Science Department, Stanford University, Stanford, CA 94305

113. Dr. Arthur Wouk, Army Research Office, P.O. Box 12211, Research Triangle Park, North Carolina 27709

114. Dr. Margaret Wright, Systems Optimization Laboratory, Operations Research Department, Stanford University, Stanford, CA 94305

115. Office of Assistant Manager for Energy Research and Development, Department of Energy, Oak Ridge Operations Office, Oak Ridge, TN 37830

116-146. Technical Information Center