ORNL/TM-11496

Engineering Physics and Mathematics Division

Mathematical Sciences Section

A COMPUTE-AHEAD IMPLEMENTATION OF THE FAN-IN SPARSE DISTRIBUTED FACTORIZATION SCHEME

C. Ashcraft [†]
S. C. Eisenstat [†]
J. W. H. Liu [‡]
B. W. Peyton [§]
A. H. Sherman [†]

[†] Department of Computer Science
Yale University
New Haven, CT 06520

[‡] Department of Computer Science
York University
North York, Ontario, Canada M3J 1P3

[§] Mathematical Sciences Section
Oak Ridge National Laboratory
P.O. Box 2009, Bldg. 9207-A
Oak Ridge, TN 37831-8083

ORNL/TM--11496

DE91 000531

Date Published: August, 1990

This research was supported by the Office of Naval Research under contracts N00014-86-K-0310 and N00014-89-J-1906, the National Science Foundation under grants DCR-85-21451 and ASC-86-11454, the Natural Sciences and Engineering Research Council of Canada under grant A5509, and the Applied Mathematical Sciences Research Program of the Office of Energy Research.

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-AC-05-84OR21400

eb

Contents

A COMPUTE-AHEAD IMPLEMENTATION OF THE FAN-IN SPARSE DISTRIBUTED FACTORIZATION SCHEME

C. Ashcraft
S. C. Eisenstat
J. W. H. Liu
B. W. Peyton
A. H. Sherman

Abstract

In this report, we consider a compute-ahead computational technique in the distributed factorization of large sparse matrices using the fan-in parallel scheme. Experimental results on an Intel iPSC/2 hypercube are provided to demonstrate the relevance and effectiveness of this technique. Fortran source code is also included in an appendix.

1. Introduction

A fan-in algorithm for distributed sparse numerical factorization of large symmetric positive definite matrices has been proposed in [1]. This report describes an implementation of this fan-in scheme that uses a "compute-ahead" technique to improve performance. We assume familiarity with the research area of parallel sparse matrix factorization and refer the reader to [6] for background material.

The basic idea behind the compute-ahead technique is simple, yet effective. In essence, when a processor is waiting for external information required by the current column on hand, it suspends this column task and proceeds with useful work on future columns of the matrix. The task of computing the original column is resumed as new external data for the column arrives.

An outline of the paper is as follows. Section 2 contains a high-level description of the *basic fan-in* sparse factorization scheme. Section 3 discusses how to incorporate compute-ahead updating into the basic fan-in scheme, and it deals with some of the issues that arise in implementing the resulting *compute-ahead fan-in* algorithm. Section 4 provides performance data for our Fortran implementation on an Intel iPSC/2 hypercube. It also compares the performance of our code with that of the fan-out code used in [6]. Section 5 contains a few closing remarks, and an appendix contains the source listing of our Fortran code.

2. The Fan-in Distributed Factorization Scheme

The *fan-in* distributed sparse factorization scheme is proposed in [1]. The scheme is best described by using the notion of *aggregate update columns.* For a column j, its *complete update column* is defined to be

$$\sum_{k<j,\ \ell_{jk}\neq 0} \ell_{jk} \begin{pmatrix} \ell_{jk} \\ \vdots \\ \ell_{nk} \end{pmatrix} .$$

An *aggregate update column* for column j is

$$\sum_{k\in K} \ell_{jk} \begin{pmatrix} \ell_{jk} \\ \vdots \\ \ell_{nk} \end{pmatrix} ,$$

where K is any subset of $\{k < j \mid \ell_{jk} \neq 0\}$, i.e. any subset of the nonzero off-diagonal locations in row j of L.

Assume that we are given a mapping of columns to processors, and let $map[j]$ denote the processor assigned to column j. For a given processor p and a column j, we define the set

$$K[p,j] := \{k < j \mid \ell_{jk} \neq 0 \text{ and } map[k] = p\} ,$$

which is simply the set of columns owned by processor p that update column j. The *internal update column* for column j from processor p is defined to be the aggregate

```
for column $j$ := 1 to $n$ do
    compute the internal update column $v$ for column $j$ ;
    if $map[j] \neq p$ then
        if $K[p,j] \neq \emptyset$ then send $v$ to processor $map[j]$ ;
    else
        while $v$ is not the complete update column for column $j$ do
            receive an external update column for column $j$ and add it to $v$ ;
        end while
        Form $L_{*j}$ from $A_{*j}$ and $v$ ;
    end if
end for
```

Figure 1: Fan-in distributed sparse Cholesky factorization on processor p.

update column where $K = K[p,j]$. If processor p owns column j, then each *external update column* for column j is an aggregate update column for column j that is internal to another processor $q \neq p$, and for which $K[q,j] \neq \emptyset$.

Using the notation and terminology introduced above, the fan-in algorithm for distributed sparse Cholesky factorization is stated in its simplest form in Figure 1. The fan-in scheme is driven by the columns, which are processed in increasing order. When processor p is computing a column j that it owns and the update column v is not yet complete, then it is forced to wait for external update column(s) for column j to arrive from other processor(s) before it can proceed with useful work on later columns of L. The next section presents an implementation of a *compute-ahead* strategy whose sole purpose is to help alleviate this problem.

3. Implementation of a Compute-ahead Fan-in Scheme

A processor need not remain idle while waiting for external data required by column j, as processor p does in Figure 1. Instead, the processor can perform so-called "compute-ahead" work on later columns $i > j$ of the matrix. The term "compute-ahead" has been used by Heath and Romine [8] (Page 564) in studying efficient distributed algorithms for triangular solution of dense linear systems. The algorithm in Figure 2 is a high-level description of how we incorporate the strategy into fan-in sparse Cholesky factorization.

As long as there are external update columns for column j that have not yet arrived, the modified algorithm alternates between processing available external update columns for column j and performing compute-ahead work on some column $i > j$ of the matrix. To ensure that the message buffer is checked regularly for incoming data required by column j, the algorithm works on only one column $i > j$ before again checking for incoming data required by column j. This permits prompt completion of the factor column L_{*j} once the last of its external update columns has arrived.

```
for column $j$ := 1 to $n$ do
    complete the internal update column $v$ for column $j$ ;
    if $map[j] \neq p$ then
        if $K[p,j] \neq \emptyset$ then send $v$ to processor $map[j]$ ;
    else
        add to $v$ all available external update columns for column $j$ ;
        while $v$ is not the complete update column for column $j$ do
            perform compute-ahead updating for some column $i > j$ ;
            add to $v$ all newly-available external update columns for column $j$ ;
        end while
        Form $L_{*j}$ from $A_{*j}$ and $v$ ;
    end if
end for
```

Figure 2: Compute-ahead fan-in sparse Cholesky factorization on processor p.

Compute-ahead work can be performed on any column $i > j$. Naturally, columns required earlier in the factorization should be given priority over those required later. Performing compute-ahead work on columns $j+1, j+2, \ldots, n$ in ascending order by column number is a reasonable choice.

3.1. Types of Compute-Ahead Updates

When waiting for external data required to complete column j, processor p will perform one of two distinct "tasks" as a unit of compute-ahead work before resuming efforts to complete column j.

- **[Compute-ahead external update.]** Receive and subtract an external update column from some column $i > j$.
- **[Compute-ahead internal updates.]** Choose some $i > j$ whose internal update column has at least one indexed SAXPY operation pending, and do the following: For every $k \in K[p,i]$ such that a) L_{*k} has been computed, and b) L_{*k} has not yet been applied to the internal update column for column i, add the appropriate multiple of L_{*k} to the internal update column for column i (an indexed SAXPY operation).

Note that the *compute-ahead internal updating task* is "greedy" in the sense that it performs every needed indexed SAXPY on column i's internal update column that it possibly can with the columns of L that are currently computed. Implementing a *compute-ahead external update* turns out to be simple and straightforward, as we shall see later in this section. But implemention issues connected with *compute-ahead internal updates* require more extended discussion.

One of the key issues is management of the work storage required to accumulate internal update columns. While the basic fan-in algorithm requires that each processor allocate only a single column of work storage to accumulate successive internal update columns (see Figure 1), the compute-ahead fan-in algorithm requires more work storage for this purpose. The compute-ahead internal-updating task cannot complete an internal update column that must be modified with a multiple of column j. As a result, the compute-ahead feature requires each processor to allocate a block of storage that can contain *incomplete* internal update columns for more than one column $i > j$. Thus, we must choose a mechanism to limit and manage work storage, while retaining sufficient access to compute-ahead work.

Our implementation does not permit the computation of all "available" compute-ahead updates. Compute-ahead internal updates are restricted to target columns within the currently active *supernode*, primarily to preserve a simple but effective overall implementation. A *supernode* is a block of contiguous columns in the Cholesky factor whose diagonal block is full triangular and whose off-block-diagonal column structure is the same for every column. Supernodes have been used in [3] to devise efficient vectorized sparse factorization schemes. They are also used in the domain-separator model [2] to study distributed sparse factorization schemes.

In our implementation, when a processor p begins work on the columns of supernode S, it has on hand work space sufficient to hold an internal update column for each column in S. For columns of S *not* owned by a processor p, the allocated work space is indispensible; for columns of S owned by processor p, the space is not required, but is merely a programming convenience. Also, because of the shared sparsity structure of columns within S, only a *single* indexing vector is required to map each entry of a column $k \in K[p,i]$ to the corresponding entry of column i for *any* column $i \in S$. Once computation on the columns of S has begun, compute-ahead internal updates for columns in S are simple and natural to perform because the required *initialized work space* and *indexing information* are already available.

We now discuss more explicitly the role of supernodes in identifying the types of compute-ahead updates actually available in our implementation. As before, let p be the processor that owns column j and assume that it is currently working on column j. Let S be the supernode containing column j and consider the situation where processor p is awaiting the arrival of some external update columns for column j. Relative to p and S, we can identify the following possible compute-ahead updates:

- internal updates for columns $i > j$, $i \in S$.
- external updates for columns $i > j$, $i \in S$.
- external updates for columns $i > j$, $i \notin S$.
- internal updates for columns $i > j$, $i \notin S$.

Compute-ahead external updates for any column $i > j$, whether inside or outside the current supernode S, will be included in our implementation. This has the desirable effect of clearing the message buffer, and moreover since we follow [1] in giving each external update column the same sparsity structure as its target column, our implementation can incorporate external update columns directly into factor column storage. Consequently, compute-ahead external updates require no additional overhead storage or computation to provide structural information, nor do they required additional work storage for their accumulation.

Compute-ahead internal updates for columns within supernode S are also included in our implementation. After initializing to zero the block of work storage large enough to contain all the columns in S, our implementation computes internal update columns for each column in S in the provided storage. All work, compute-ahead or otherwise, on the internal update column for a column $i \in S$ is applied to the corresponding vector in work storage. Computed just before the first column of S is processed, the single subscript indexing vector required by S is used to apply these internal updates to the appropriate column in working storage.

As noted before, the algorithm may have to toggle quite often between probing the message queue for external update columns for the current column j, and processing compute-ahead updates (internal or external) for a column $i > j$. It is therefore important to alternate between these tasks in a smooth and efficient manner. Compute-ahead external updates satisfy this requirement because no indexing information or additional work storage is required to apply an external update to its target column. Compute-ahead internal updates for columns within the current supernode S satisfy this requirement because the initialized work space and the necessary indexing information are already available. However, compute-ahead internal updates to columns *outside* S do not have these advantages; they require an extra block of intialized work storage and a new indexing vector before other useful computation can be resumed. Thus, we have excluded such internal compute-ahead updates from our implementation, and consequently, processors will generally become idle more often when processing the last few columns of a supernode. Indeed, when the last column of a supernode is being processed, no compute-ahead internal updates are possible.

To explore the effects of limiting compute-ahead internal updates to columns in the currently active supernode, we developed a second code that allows compute-ahead internal updating to cross at most one supernode boundary. Such a code has access to more compute-ahead internal updates, but at the expense of an increase in 1) the complexity of the program, 2) the work storage requirement, and 3) the bookkeeping overhead required to manage work storage. Preliminary results with that code revealed very little difference in performance between it and a much simpler code based on the algorithm given in the next subsection.

3.2. The Detailed Algorithm

We assume that the given sparse matrix has been properly ordered for parallel elimination and that the supernode blocks of the ordering have been determined. The detailed compute-ahead algorithm is given in Figure 3.

The compute-ahead section of the algorithm can be interpreted more informally in the following way: As long as there are external update columns for the current column j that have not yet been processed, obtain a task of highest priority and perform it,

```
for each supernode block $S$ do
    let $s, s+1, \ldots, s+k-1$ be the columns of the current supernode $S$ ;
    initialize to zero work space for internal update columns $v_s, v_{s+1}, \ldots, v_{s+k-1}$ ;
    compute the subscript indexing vector for $S$ ;
    for $j := s$ to $s+k-1$ do
        assume that initially $L_{*j} = A_{*j}$ ;
        complete the internal update column $v_j$ for column $j$ ;
        if $map[j] \neq p$ then
            if $K[p, j] \neq \emptyset$ then send $v_j$ to processor $map[j]$ ;
        else
            subtract from $L_{*j}$ every available external update for column $j$ ;
            while external update columns for column $j$ remain to be processed do
                if internal indexed SAXPY's are pending for some column $i \in S$ then
                    perform all pending internal indexed SAXPY's for the first such
                    column $i \in S$, $i > j$, accumulating the result in $v_i$ ;
                else
                    receive any available external update column and subtract
                    it from the target column $L_{*i}$, $i \geq j$.
                end if
                subtract from $L_{*j}$ every newly-available external update column
                for column $j$ ;
            end while
            subtract $v_j$ from $L_{*j}$ and scale the resulting vector to obtain column $j$ of $L$.
        end if
    end for
    free the work space for future use ;
end for
```

Figure 3: Detailed version of compute-ahead fan-in sparse Cholesky factorization on processor p.

where the tasks to be done are ranked in descending order of "urgency" as follows:

1. Receive and apply directly to column j every available external update column for column j (whenever at least one such update column is available in the message queue).

2. **[Compute-ahead internal updates.]** Perform all column updates, i.e. indexed SAXPY's, waiting to be incorporated into the internal update column for the first column $i \in S$, $i > j$ that has any such updates pending.

3. **[Compute-ahead external update.]** Receive and apply any available external update column to its target column $i \geq j$.

Thus, external data for column j is processed as long as such data are available in the message queue. When column j remains imcomplete and the message queue contains no external data for column j, the algorithm performs compute-ahead internal updates. When there are neither external update columns for column j nor internal updates for columns $i \in S$, $i > j$, then, and only then, does the algorithm process any available external updates. Note that after all pending internal indexed SAXPY's for columns in S are exhausted, only external updates for column $i \geq j$ are available, until finally column j is completed.

While processing an external update column requires little work, the compute-ahead internal-updating task may sometimes perform quite a few indexed SAXPY's before the message queue is again checked for data required by column j. The decision to allow the compute-ahead internal-updating task to perform *all* indexed SAXPY's pending for a single column $i > j$ merits further comment. While this appears to be a natural choice, we were concerned that it might not permit the program to check the message queue often enough for data required by column j. To investigate this question, we introduced into our program a parameter KTROL that limits the number of indexed SAXPY'S that may constitute a single compute-ahead internal-update task. We tried several widely-varying values of KTROL and never observed more than 2% difference in factorization time between the best and the worst case. The worst results were obtained with KTROL=1, which restricts the compute-ahead internal-update task to a single indexed SAXPY. This setting for KTROL evidently caused the code to waste a small amount of time on an excessive number of subroutine calls to perform the compute-ahead internal updates and on an excessive number of probes for for incoming external update columns for the current column j. We observed less than 1% variability in factorization time as long as KTROL was chosen to allow at least a few indexed SAXPY's. We consistently obtained our best timing results (by an extremely small margin) when KTROL was chosen large enough to allow the compute-ahead internal-updating task to compute all pending indexed SAXPY's for the target internal update column. Thus we incorporated into our algorithm a compute-ahead internal-update task that is as "complete" as possible, because it is marginally more efficient, appears to be the natural choice from the start, and helps preserve the simplicity of the algorithm.

grid problem	*np*	basic fan-out	basic fan-in	compute-ahead fan-in
50x50	1	22.470	13.911	13.910
	2	12.545	7.201	7.240
	4	7.509	4.000	3.747
	8	5.197	2.460	2.273
	16	3.619	1.564	1.364
	32	2.639	0.972	0.872
	64	2.020	0.684	0.659
75x75	1	80.447	48.388	48.419
	2	42.278	24.388	24.360
	4	23.291	13.118	12.380
	8	14.643	7.935	7.307
	16	9.733	4.815	4.222
	32	6.860	2.887	2.490
	64	4.976	1.748	1.561
100x100	1	—	115.341	115.350
	2	105.989	58.488	58.439
	4	57.539	31.660	30.064
	8	34.324	18.586	17.090
	16	21.042	11.191	9.484
	32	13.860	6.459	5.380
	64	9.529	3.781	3.198

Table 1: Parallel factorization time (in seconds) on an Intel iPSC/2.

4. Experimental Results

The *compute-ahead fan-in* algorithm for sparse Cholesky factorization was implemented in Fortran and run on an Intel iPSC/2 hypercube. The test problems were nine-point finite-difference operators on square grids. We used the nested dissection ordering [5] since it gives optimal-order fill and well-balanced elimination trees for these problems. We used the subtree-to-subcube mapping [7] to assign processors to columns since it gives good load balance and reduces communication. Our code is written so that when the parameter `KTROL`, discussed in the previous section, is set to zero, it becomes an implementation of the basic fan-in algorithm shown earlier in Figure 1. When `KTROL` is set to a sufficiently high value, our code becomes an implementation of the compute-ahead fan-in algorithm shown in Figure 3 in the previous section. Until recently, the best-known algorithm for distributed sparse Cholesky factorization was a *basic fan-out* algorithm reported in [6]. We include it in our numerical results. We refer to this version of the fan-out algorithm as *basic fan-out* in order to distinguish it from the more recent *domain fan-out* algorithm introduced independently in [2] and in [9]. Table 1 contains timing results for the three algorithms: basic fan-out, basic fan-in, and compute-ahead fan-in.

The factorization times reported in Table 1 demonstrate the large advantage of the fan-in scheme over the fan-out scheme, thus confirming results reported in [2]. But the primary objective of these tests is to confirm whether or not the compute-ahead technique significantly improves the efficiency of the basic fan-in algorithm. The usefulness of the technique is adequately demonstrated by these timing results, particularly by the factorization times obtained for the largest problem on 16, 32 and 64 processors. On the 100x100 grid, basic fan-in is respectively 18.0%, 20.1% and 18.2% slower than compute-ahead fan-in on 16, 32, and 64 processors.

We would like to point out that the problem set used in Table 1 includes the problems used by Zmijewski in [9] to compare the domain fan-out algorithm with the basic fan-out algorithm. Though he also made his runs on an iPSC/2, his timings and ours cannot be compared directly because his machine differs from ours and/or he selected different options when compiling his Fortran code. Because he used the same basic fan-out code that we used, one can, with caution, make a rough comparison of our results with his by normalizing all times against those obtained for the common basic fan-out runs.

5. Concluding Remarks

We have described an implementation of the fan-in distributed sparse factorization scheme that uses a compute-ahead technique to improve performance over the basic fan-in scheme. We have detailed how to use supernodes to limit the amount of additional work storage required by the compute-ahead fan-in algorithm, and to organize the computation in a way that enables clean and efficient access to the compute-ahead internal updates. We have indicated how providing access to compute-ahead internal updates across supernode boundaries increases the amount of work storage required and makes the code more complex and difficult to write. While the improvement in the factorization times of either fan-in scheme over the basic fan-out scheme is by far the most significant demonstrated in our testing, we have shown that incorporating compute-ahead updates into the basic fan-in algorithm significantly improves its performance, at least under the ideal circumstances used in our tests. The source code is included in the appendix to show our implementation.

6. References

[1] C. Ashcraft, S. Eisenstat, and J. Liu. *A fan-in algorithm for distributed sparse numerical factorization.* Technical Report CS-89-03, Department of Computer Science, York University, 1989. (to appear in SIAM J. Sci. Statist. Comput.).

[2] C. Ashcraft, S. Eisenstat, J. Liu, and A. Sherman. *The comparison of three column-based distributed sparse factorization schemes.* Technical Report, Department of Computer Science, York University, 1990. (in preparation).

[3] C. Ashcraft, R. Grimes, J. Lewis, B. Peyton, and H. Simon. Progress in sparse matrix methods for large linear systems on vector supercomputers. *Intern. J. Supercomputer Applic.*, 1(4):10–29, 1987.

[4] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A machine-independent communication library. In John L. Gustafson, editor, *Hypercube Concurrent Computers and Applications 1989*, 1990. (to appear).

[5] J. A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, 10:345–363, 1973.

[6] J. A. George, M. Heath, J. W. H. Liu, and E. Ng. Sparse Cholesky factorization on a local-memory multiprocessor. *SIAM J. Sci. Statist. Comput.*, 9:327–340, 1988.

[7] J. A. George, J. W. H. Liu, and E. Ng. Communication results for parallel sparse Cholesky factorization on a hypercube. *Parallel Computing*, 10:287–298, 1989.

[8] M. T. Heath and C. H. Romine. Parallel solution of triangular systems on distributed-memory multiprocessors. *SIAM J. Sci. Statist. Comput.*, 9:558–588, 1988.

[9] E. Zmijewski. *Limiting communication in parallel sparse Cholesky factorization.* Technical Report TRCS89-18, Department of Computer Science, University of California at Santa Barbara, California, 1989.

Appendix: Fortran Source Listing

Our routines call four routines from the Portable Instrumented Communication Library [4] (PICL), which is designed to provide a portable syntax for the message-passing routines used on typical distributed-memory MIMD machines. A brief description of these four routines is given below.

```
subroutine send0 ( buf, bytes, type, node )
character*(*) buf
integer bytes, type, node
```

The subroutine `send0` sends a message of length `bytes` stored in `buf` to processor `node`. The variable `type` is used by the receiving processor to distinguish one "type" of message from another. The contents of `buf` need not be character data; `buf` can contain data of any valid Fortran data type. This applies to `buf` in subroutine `recv0` below, also.

```
subroutine recv0 ( buf, bytes, type )
character*(*) buf
integer bytes, type
```

The subroutine `recv0` receives a message with type field `type` into a buffer `buf`. The variable `bytes` contains the length of the buffer (in bytes). When `type` is -1, any incoming message will satisfy the request. This applies to `type` in `probe0` below, also.

```
integer function probe0 ( type )
integer type
```

The integer function probe0 returns the value 1 if the processor has received a message of the specified type; otherwise it returns the value 0.

```
     subroutine recvinfo0 ( bytes, type, node )
     integer bytes, type, node
```

The subroutine recvinfo0 returns information about the most recently received or "probed for" message: bytes contains the length of the message (in bytes), type contains the integer "type" of the message, and node contains the processor ID number of the processor that sent the message.

```
C***********************************************************************
C***********************************************************************
C*****     FANIN ..... PARALLEL SPARSE FAN-IN FACTORIZATION      ********
C***********************************************************************
C***********************************************************************
C
C       PURPOSE:
C          THIS SUBROUTINE PERFORMS A SPARSE FAN-IN DISTRIBUTED
C          CHOLESKY DECOMPOSITION, WITH AN OPTIONAL COMPUTE-AHEAD
C          TECHNIQUE TO IMPROVE PERFORMANCE.
C
C       INPUT PARAMETERS:
C          KTROL        - CONTROLS NUMBER OF COMPUTE-AHEAD UPDATES:
C                            < 1, PERFORM NO COMPUTE-AHEAD UPDATES.
C                           >= 1, PERFORM NO MORE THAN KTROL UNINTERUPTED
C                                 COMPUTE-AHEAD INTERNAL UPDATES.
C          ME           - NODE NUMBER OF THIS NODE PROCESSOR.
C          NEQNS        - NUMBER OF EQUATIONS.
C          MAP          - MAPS EACH COLUMN TO THE PROCESSOR THAT OWNS IT.
C          XBLK         - SUPERNODE PARTITION.  XBLK(I) POINTS TO THE
C                         FIRST COLUMN OF THE I-TH SUPERNODE.
C          NBLKS        - NUMBER OF SUPERNODES.
C          MSGCNT       - MSGCNT(J) CONTAINS THE NUMBER OF EXTERNAL
C                         MESSAGE UPDATES REQUIRED BY COLUMN J.
C          XLNZ         - SPARSPAK'S LNZ POINTER ARRAY; USED TO
C                         OBTAIN COLUMN LENGTHS.
C          XNZSUB,NZSUB - ROW SUBSCRIPT ARRAY; SAME AS SPARSPAK.
C          MAXWS        - MAXIMUM SIZE OF WS.
C          MYET         - LOCAL ELIMINATION TREE. MYET(I) IS 1 (TRUE)
C                         IF NODE I HAS A DESCENDANT WHICH BELONGS TO
C                         THIS PROCESSOR.  OTHERWISE, MYET(I) IS 0
C                         (FALSE).
C
C       OUTPUT PARAMETERS:
C          ERROR        - ERROR CODE.  (ERROR = 180 IF MATRIX IS NOT
C                         POSITIVE DEFINITE.)
C
C       UPDATED PARAMETERS:
C          XMYLNZ,MYLNZ - ON INPUT, MY COLUMNS OF A.
C                         ON OUTPUT, MY COLUMNS OF L.
C
C       WORKING PARAMETERS:
C          WS           - WORK SPACE FOR COLUMNS OF A SUPERNODE.
C          LINK         - AT STEP J, CONTAINS LINKED LIST OF MY
C                         COLUMNS THAT WILL UPDATE COLUMN J.
C          FIRST        - FIRST(I) POINTS TO THE TOP OF THE 'ACTIVE'
C                         PORTION OF COLUMN I.
C          UPDINX       - UPDATE INDEX VECTOR.
C          MSGUPD       - BUFFER INTO WHICH EXTERNAL UPDATES ARE
C                         RECEIVED.
C
C       PROGRAM SUBROUTINES:
C          SENDO, INTUPD, EXTUPD.
C
C***********************************************************************
C
        SUBROUTINE  FANIN ( KTROL , ME    , NEQNS , MAP   , XBLK  ,
     &                      NBLKS , MSGCNT, XMYLNZ, MYLNZ , XLNZ  ,
     &                      XNZSUB, NZSUB , MAXWS , WS    , LINK  ,
     &                      FIRST , MYET  , UPDINX, MSGUPD, ERROR  )
C
C***********************************************************************
C
C    -----------
```

```
C   PARAMETERS.
C   -----------
       INTEGER           ERROR , KTROL , MAXWS , ME     , NEQNS ,
     &                   NBLKS
       INTEGER           FIRST(*) , LINK(*)  , MAP(*)    , MSGCNT(*),
     &                   MYET(*)  , NZSUB(*) , UPDINX(*), XBLK(*)  ,
     &                   XLNZ(*)  , XMYLNZ(*), XNZSUB(*)
       REAL              MSGUPD(*), MYLNZ(*) , WS(*)
C
C   ----------------
C   LOCAL VARIABLES.
C   ----------------
       INTEGER           BLKSZE, FSTLNK, I      , II     , ISTOP ,
     &                   ISTRT , ISUB  , J      , JSIZE  , JSTOP ,
     &                   JSTRT , JXWS  , K      , KBLK   , KSTOP ,
     &                   KSTRT , KSUB  , WSSIZE, NEQNS4
       REAL              DIAGJ
C
C***********************************************************************
C
       NEQNS4 = NEQNS * 4
       DO  100  J = 1, NEQNS
           LINK(J) = 0
  100  CONTINUE
C      ---------------------------
C      FOR EACH SUPERNODE KBLK ...
C      ---------------------------
       DO  700  KBLK = 1, NBLKS
           JSTOP = XBLK(KBLK+1) - 1
           IF  ( MYET(JSTOP) .NE. 0 )  THEN
C              ---------------------------------------------
C              ... THAT INTERSECTS MY ELIMINATION TREE, FIND
C                  THE FIRST COLUMN IN MY ELIMINATION TREE.
C              ---------------------------------------------
               JSTRT = XBLK(KBLK)
  200          IF ( MYET(JSTRT) .EQ. 0 )  THEN
                  JSTRT = JSTRT + 1
                  GOTO 200
               ENDIF
C              ------------------------------------------------
C              INITIALIZE WORK SPACE FOR CURRENT SUPERNODE KBLK.
C              **NOTE** EACH COLUMN IN WS INCLUDES THE DIAGONAL.
C              ------------------------------------------------
               JSIZE = XLNZ(JSTRT+1) - XLNZ(JSTRT) + 1
               BLKSZE = JSTOP - JSTRT + 1
               WSSIZE = JSIZE*BLKSZE - BLKSZE*(BLKSZE-1)/2
               DO  300  II = 1, WSSIZE
                   WS(II) = 0.0
  300          CONTINUE
C              ---------------------------------------------
C              SET UP THE UPDATE INDEX VECTOR FOR XBLK.
C              **NOTE** NZSUB DOES NOT INCLUDE THE DIAGONAL.
C              ---------------------------------------------
               KSTRT = XNZSUB(JSTRT)
               KSTOP = KSTRT + JSIZE - 2
               UPDINX(JSTRT) = 1
               ISUB = 1
               DO  400  K = KSTRT, KSTOP
                   KSUB = NZSUB(K)
                   ISUB = ISUB + 1
                   UPDINX(KSUB) = ISUB
  400          CONTINUE
C              ----------------------------------------------
C              FOR EACH COLUMN J IN CURRENT SUPERNODE, DO ...
```

```
C              ----------------------------------------------
               JXWS = 1
               DO  600  J = JSTRT, JSTOP
C                  ----------------------------------
C                  FORM INTERNAL UPDATE FOR COLUMN J.
C                  ----------------------------------
                   FSTLNK = LINK(J)
                   IF  ( FSTLNK .GT. 0 )  THEN
                       CALL INTUPD ( J, JSIZE, WS(JXWS), UPDINX, LINK,
     &                        FIRST, XNZSUB, NZSUB, XMYLNZ, MYLNZ, J )
                   ENDIF
C                  -----------------------------------------------
C                  IF J IS NOT MINE, SEND NON-ZERO INTERNAL UPDATE
C                  TO OWNER OF J.
C                  -----------------------------------------------
                   IF  ( MAP(J) .NE. ME )  THEN
                       IF  ( FSTLNK .GT. 0 )  THEN
                           CALL SENDO ( WS(JXWS), 4*JSIZE, J, MAP(J) )
                       ENDIF
                   ELSE
C                      ------------------------------------------
C                      IF J IS MINE AND ITS UPDATE IS INCOMPLETE,
C                      RECEIVE AND APPLY EXTERNAL UPDATES ...
C                      ------------------------------------------
                       IF  ( MSGCNT(J) .GT. 0 )  THEN
                           IF  ( KTROL .GT. 0 )  THEN
C                              --------------------------------
C                              ... WITH COMPUTE-AHEAD UPDATING.
C                              --------------------------------
                               CALL EXUPCA ( KTROL, NEQNS4, J,
     &                             JSIZE, JXWS, WS, JSTOP, MSGCNT,
     &                             UPDINX, LINK, FIRST, XNZSUB, NZSUB,
     &                             XMYLNZ, MYLNZ, MSGUPD )
                           ELSE
C                              -----------------------------------
C                              ... WITH NO COMPUTE-AHEAD UPDATING.
C                                  (PURE FAN-IN)
C                              -----------------------------------
                               CALL EXTUPD ( NEQNS4, J, JSIZE,
     &                             MSGCNT(J), MYLNZ(XMYLNZ(J)),
     &                             MSGUPD )
                           ENDIF
                       ENDIF
C                      -------------------------------------------
C                      APPLY INTERNAL UPDATES ACCUMULATED IN WS TO
C                      COLUMN L(*,J). MODIFY LINK(*) AND FIRST(*).
C                      -------------------------------------------
                       ISTRT = XMYLNZ(J)
                       ISTOP = XMYLNZ(J+1) - 1
                       DIAGJ = MYLNZ(ISTRT) - WS(JXWS)
                       IF ( DIAGJ .LE. 0.0 )  GOTO 800
                       DIAGJ = SQRT(DIAGJ)
                       MYLNZ(ISTRT) = DIAGJ
                       IF  ( JSIZE .GT. 1 )  THEN
                           ISTRT = ISTRT + 1
                           FIRST(J) = ISTRT
                           I = XNZSUB(J)
                           ISUB = NZSUB(I)
                           LINK(J) = LINK(ISUB)
                           LINK(ISUB) = J
                           ISUB = JXWS
                           DO  500  II = ISTRT, ISTOP
                               ISUB = ISUB + 1
                               MYLNZ(II) = (MYLNZ(II)-WS(ISUB))/DIAGJ
```

```
  500                     CONTINUE
                      ENDIF
                  ENDIF
C                 -------------------------------------------
C                 PROCEED WITH NEXT COLUMN IN SUPERNODE KBLK.
C                 -------------------------------------------
                  JXWS = JXWS + JSIZE
                  JSIZE = JSIZE - 1
  600         CONTINUE
          ENDIF
C         ----------------------------
C         PROCEED WITH NEXT SUPERNODE.
C         ----------------------------
  700 CONTINUE
      RETURN
C     -----------
C     ERROR EXIT.
C     -----------
  800 ERROR = 180
      RETURN
      END
```

```
C****************************************************************
C****************************************************************
C******     INTUPD ..... FAN-IN: INTERNAL COLUMN UPDATES   ******
C****************************************************************
C****************************************************************
C
C      PURPOSE:
C         THIS ROUTINE PERFORMS A CONTROLLED NUMBER OF INTERNAL
C         UPDATES ON A GIVEN COLUMN.
C
C      INPUT PARAMETERS:
C         J               - COLUMN TO WHICH INTERNAL UPDATES ARE
C                           TO BE APPLIED.
C         JSIZE           - NUMBER OF NONZEROS IN COLUMN J: L(*,J).
C         UPDINX          - UPDATE INDEX VECTOR FOR COLUMN J.
C         XNZSUB,NZSUB - ROW SUBSCRIPT ARRAY; SAME AS SPARSPAK.
C         XMYLNZ,MYLNZ - MY COLUMNS OF L.
C         KTROL           - CONTROLS THE MAXIMUM NUMBER OF COLUMN
C                           UPDATES.  (>=J PERFORMS ALL UPDATES ON
C                           COL J)
C
C      UPDATED PARAMETERS:
C         U               - STORAGE FOR UPDATE VECTOR OF J.
C         LINK            - CONTAINS LINKED LIST OF MY COLUMNS
C                           THAT WILL UPDATE COLUMN J.
C         FIRST           - FIRST(I) POINTS TO THE TOP OF THE
C                           'ACTIVE' PORTION OF COLUMN I.
C
C****************************************************************
C
      SUBROUTINE  INTUPD ( J      , JSIZE , U      , UPDINX, LINK   ,
     &                     FIRST  , XNZSUB, NZSUB  , XMYLNZ, MYLNZ  ,
     &                     KTROL    )
C
C****************************************************************
C
C      -----------
C      PARAMETERS.
C      -----------
       INTEGER             J      , JSIZE , KTROL
       INTEGER             FIRST(*) , LINK(*)  , NZSUB(*) , UPDINX(*),
     &                     XMYLNZ(*), XNZSUB(*)
       REAL                U(*)     , MYLNZ(*)
C
C      ----------------
C      LOCAL VARIABLES.
C      ----------------
       INTEGER             I      , II     , ISTOP , ISTRT , ISUB   ,
     &                     K      , NMOD   , OFFSET
       REAL                LJK
C
C****************************************************************
C
       NMOD = 1
       OFFSET = UPDINX(J) - 1
C      ---------------------------------------------
C      FOR EACH COLUMN K IN THE LINK, APPLY CMOD(J,K)
C      ---------------------------------------------
  100  K = LINK(J)
       IF  ( K .GT. 0 .AND. NMOD .LE. KTROL )  THEN
           LINK(J) = LINK(K)
           NMOD = NMOD + 1
C
           ISTRT = FIRST(K)
```

```
          ISTOP = XMYLNZ(K+1) - 1
          LJK = MYLNZ(ISTRT)
          I = XNZSUB(K) + ISTRT - XMYLNZ(K)
C         ---------------------------------------------------
C         UPDATE FIRST/LINK FOR FUTURE MODIFICATION STEPS.
C         [**NOTE** XMYLNZ POINTS TO DIAG ENTRY
C         XNZSUB POINTS TO SUB-DIAG ENTRY]
C         ---------------------------------------------------
          IF  ( ISTOP .GT. ISTRT )  THEN
              FIRST(K) = ISTRT + 1
              ISUB = NZSUB(I)
              LINK(K) = LINK(ISUB)
              LINK(ISUB) = K
          ENDIF
C         ------------------------------------------
C         IF THE UPDATING AND UPDATED COLUMN HAVE THE
C         SAME NUMBER OF NONZERO ENTRIES ...
C         ------------------------------------------
          IF  ( ISTOP-ISTRT+1 .LT. JSIZE )  THEN
C             ------------------------------------------
C             PERFORM SPARSE (INDIRECT) COLUMN UPDATE.
C             [**NOTE** I=I-1 TO INCLUDE DIAG UPDATE.]
C             ------------------------------------------
              I = I - 1
              DO  200  II = ISTRT, ISTOP
                  ISUB = NZSUB(I)
                  ISUB = UPDINX(ISUB) - OFFSET
                  U(ISUB) = U(ISUB) + MYLNZ(II)*LJK
                  I = I + 1
  200         CONTINUE
          ELSE
C             ---------------------------------------------------
C             OTHERWISE, PERFORM DENSE (DIRECT) COLUMN UPDATE.
C             ---------------------------------------------------
              ISUB = 1
              DO  300  II = ISTRT, ISTOP
                  U(ISUB) = U(ISUB) + MYLNZ(II)*LJK
                  ISUB = ISUB + 1
  300         CONTINUE
          ENDIF
          GOTO 100
      ENDIF
C
      RETURN
      END
```

```
C***************************************************************
C***************************************************************
C******     EXUPCA ..... FAN-IN: EXTERNAL UPDATES W/CA     ******
C***************************************************************
C***************************************************************
C
C     PURPOSE:
C        THIS ROUTINE PERFORMS EXTERNAL UPDATES ON A GIVEN
C        COLUMN. A CONTROLLED AMOUNT OF COMPUTE-AHEAD UPDATING
C        WILL BE PERFORMED WHEN THE PROCESSOR IS WAITING FOR
C        EXTERNAL UPDATE COLUMNS.
C
C     INPUT PARAMETERS:
C        KTROL         - CONTROLS THE MAXIMUM NUMBER OF
C                        UNINTERRUPTED INTERNAL COLUMN UPDATES.
C        NEQNS4        - NUMBER OF EQUATIONS TIMES 4
C        J             - COLUMN TO WHICH EXTERNAL UPDATES ARE
C                        TO BE APPLIED.
C        JSIZE         - NO OF NONZEROS IN COLUMN J: L(*,J)
C        JXWS          - INDEX TO WS, POINTS TO THE START OF
C                        UPDATE FOR J
C        LASTJ         - LAST COLUMN IN THE SUPERNODE WITH
C                        COLUMN J.
C        UPDINX        - UPDATE INDEX VECTOR FOR CURRENT
C                        SUPERNODE
C        XNZSUB,NZSUB - ROW SUBSCRIPT ARRAY; SAME AS SPARSPAK.
C
C     UPDATED PARAMETERS:
C        WS            - WORKSPACE FOR COLUMNS OF J'S SUPERNODE.
C        MSGCNT        - MSGCNT(I) CONTAINS NUMBER OF EXTERNAL
C                        UPDATES REMAINING FOR COLUMN I. IT IS
C                        DECREMENTED TO REFLECT ANY APPLIED
C                        EXTERNAL UPDATES.
C        LINK          - CONTAINS LINKED LIST OF 'MY' COLUMNS
C                        THAT WILL UPDATE COLUMN J.
C        FIRST         - FIRST(I) POINTS TO THE TOP OF THE
C                        'ACTIVE' PORTION OF COLUMN I.
C        XMYLNZ,MYLNZ - MY COLUMNS OF L.
C
C     WORK PARAMETERS:
C        MSGUPD        - STORAGE FOR INCOMING EXTERNAL UPDATE
C                        COLUMNS FOR J.
C
C     PROGRAM SUBROUTINES:
C        PROBEO, RECVO, RECVINFOO
C
C***************************************************************
C
      SUBROUTINE  EXUPCA ( KTROL , NEQNS4, J      , JSIZE , JXWS   ,
     &                     WS     , LASTJ , MSGCNT, UPDINX, LINK   ,
     &                     FIRST  , XNZSUB, NZSUB , XMYLNZ, MYLNZ  ,
     &                     MSGUPD  )
C
C***************************************************************
C
C     ------------
C     PARAMETERS.
C     -----------
      INTEGER    J      , JSIZE , JXWS   , LASTJ , KTROL , NEQNS4
      INTEGER    FIRST(*) , LINK(*)   , MSGCNT(*), NZSUB(*) ,
     &           UPDINX(*), XMYLNZ(*), XNZSUB(*)
      REAL       MSGUPD(*), MYLNZ(*) , WS(*)
C
C     ----------------
```

```
C       LOCAL VARIABLES.
C       ----------------
        INTEGER     BYTES , I      , II     , ISIZE , ISUB   , JSUB   ,
     &              K     , KSIZE  , KXWS   , NODE
C
C       --------------------
C       EXTERNAL FUNCTIONS.
C       --------------------
        INTEGER             PROBEO, RECVINFOO
C
C*****************************************************************
C
        K = J + 1
        KXWS = JXWS + JSIZE
        KSIZE = JSIZE - 1
        JSUB = XMYLNZ(J)
C       ------------------------------------------------
C       WHILE THERE IS MESSAGE FOR COLUMN J,
C       RECEIVE IT AND APPLY EXTERNAL UPDATE TO L(*,J).
C       ------------------------------------------------
  100   IF  ( PROBEO(J) .GT. 0 )  THEN
            CALL RECVO ( MSGUPD, NEQNS4, J )
            ISUB = JSUB
            DO  200 II = 1, JSIZE
                MYLNZ(ISUB) = MYLNZ(ISUB) - MSGUPD(II)
                ISUB = ISUB + 1
  200       CONTINUE
            MSGCNT(J) = MSGCNT(J) - 1
            IF ( MSGCNT(J) .LE. 0 )  RETURN
            GO TO 100
        ENDIF
C       --------------------------------------------
C       PERFORM COMPUTE-AHEAD INTERNAL UPDATES ON
C       REMAINING COLUMNS OF THE CURRENT SUPERNODE.
C       --------------------------------------------
  300   IF  ( K .LE. LASTJ )  THEN
            IF  ( LINK(K) .GT. O )  THEN
                CALL  INTUPD ( K, KSIZE, WS(KXWS), UPDINX, LINK,
     &                         FIRST, XNZSUB, NZSUB, XMYLNZ,
     &                         MYLNZ, KTROL )
            ELSE
                K = K + 1
                KXWS = KXWS + KSIZE
                KSIZE = KSIZE - 1
                GO TO 300
            ENDIF
            GO TO 100
        ENDIF
C       ----------------------------------------------
C       PERFORM COMPUTE-AHEAD EXTERNAL UPDATES WITH
C       INCOMING MESSAGES, GIVING PRIORITY TO UPDATES
C       FOR COLUMN J.
C       ----------------------------------------------
  400   CONTINUE
            IF  ( PROBEO(J) .EQ. 1 )  THEN
                CALL RECVO ( MSGUPD, NEQNS4, J )
                I = J
            ELSE
                CALL RECVO ( MSGUPD, NEQNS4, -1 )
                CALL RECVINFOO ( BYTES, I, NODE )
            ENDIF
            ISUB = XMYLNZ(I)
            ISIZE = XMYLNZ(I+1) - ISUB
            DO  500 II = 1, ISIZE
```

```
                MYLNZ(ISUB) = MYLNZ(ISUB) - MSGUPD(II)
                ISUB = ISUB + 1
  500       CONTINUE
            MSGCNT(I) = MSGCNT(I) - 1
            IF ( MSGCNT(J) .LE. 0 )  RETURN
        GO TO 400
C
        END
```

```
C****************************************************************
C****************************************************************
C******     EXTUPD ..... FAN-IN: EXTERNAL COLUMN UPDATES   ******
C****************************************************************
C****************************************************************
C
C      PURPOSE:
C         THIS ROUTINE PERFORMS EXTERNAL UPDATES ON A GIVEN
C         COLUMN WITH NO COMPUTE-AHEAD UPDATING.
C
C      INPUT PARAMETERS:
C         NEQNS4        - NUMBER OF EQUATIONS TIMES 4.
C         J             - COLUMN TO WHICH EXTERNAL UPDATES ARE
C                         TO BE APPLIED.
C         JSIZE         - NUMBER OF NONZEROS IN COLUMN J: L(*,J).
C
C      UPDATED PARAMETERS:
C         MSGCNT        - MSGCNT CONTAINS NUMBER OF EXTERNAL
C                         UPDATES REMAINING FOR COLUMN J.
C         JNZ           - ON OUTPUT, ALL UPDATES HAVE BEEN
C                         APPLIED TO COLUMN J OF L.
C
C      WORK PARAMETERS:
C         MSGUPD        - STORAGE FOR INCOMING EXTERNAL UPDATE
C                         COLUMN OF J.
C
C      PROGRAM SUBROUTINES:
C         RECVO         - RECEIVE A MESSAGE.
C
C****************************************************************
C
       SUBROUTINE  EXTUPD ( NEQNS4, J      , JSIZE , MSGCNT,
     &                      JNZ    , MSGUPD  )
C
C****************************************************************
C
C      -----------
C      PARAMETERS.
C      -----------
       INTEGER     J      , JSIZE , NEQNS4, MSGCNT
       REAL        MSGUPD(*), JNZ(*)
C
C      ----------------
C      LOCAL VARIABLES.
C      ----------------
       INTEGER     II
C
C****************************************************************
C
C      ---------------------------------------------
C      RECEIVE AND APPLY EXTERNAL UPDATES TO L(*,J),
C      UNTIL ALL SUCH UPDATES HAVE BEEN APPLIED.
C      ---------------------------------------------
  100  CONTINUE
           CALL RECVO ( MSGUPD, NEQNS4, J )
           DO  200 II = 1, JSIZE
               JNZ(II) = JNZ(II) - MSGUPD(II)
  200      CONTINUE
           MSGCNT = MSGCNT - 1
           IF ( MSGCNT .LE. 0 )  RETURN
       GO TO 100
       END
```

END

DATE FILMED

11 / 08 / 90