NASA Contractor ~~Report~~ 178422

ICASE REPORT NO. 87-75

ICASE

PARALLEL PIVOTING COMBINED WITH PARALLEL REDUCTION

Gita Alaghband

(NASA-CR-178422) PARALLEL PIVOTING COMBINED WITH PARALLEL REDUCTION Final Report (NASA) 41 p CSCL 09B

N88-14662

Unclas
G3/61 0118199

Contracts No. NAS1-17070 and NAS1-18107
December 1987

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Parallel Pivoting Combined with Parallel Reduction and Fill-in Control

Gita Alaghband

University of Colorado at Denver
Department of Electrical Engineering and Computer Science
1100 14th Street (Campus Box 104)
Denver, Colorado 80202*

Parallel algorithms for triangularization of large, sparse, and unsymmetric matrices are presented. The method combines the parallel reduction with a new parallel pivoting technique, control over generation of fill-ins and check for numerical stability, all done in parallel with the work being distributed over the active processes. The parallel pivoting technique uses the compatibility relation between pivots to identify parallel pivot candidates and uses the Markowitz number of pivots to minimize fill-in. This technique is not a preordering of the sparse matrix and is applied dynamically as the decomposition proceeds.

Research was supported by the National Aeronautics and Space Administration under NASA Contracts No. NAS1-17070 and NAS1-18107 and by the Air Force Office of Scientific Research under Contract AFOSR 85-0189 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665.

1. Introduction

In this paper we present multiprocessor algorithms for solving large systems of linear equations where the coefficient matrix is sparse and unsymmetric. VLSI circuit simulation, structural analysis, partial differential equations, and chemical analysis are few examples of applications requiring the solution of such systems of equations.

The algorithms described in the paper are designed for a shared-memory, MIMD model for parallel computation, in which the total memory address space is accessible uniformly to all parallel units. This computational model provides synchronization mechanisms to allow multiple updates. If multiple updates are aimed at the same memory cell, the penalty paid is a short delay in access time.

Given is a system of linear equations:

$$A\,x = b \tag{1.1}$$

where the coefficient matrix, A, is large and sparse. This paper concentrates on a direct parallel solution method for solving (1.1) by factoring A into lower (L) and upper (U) triangular matrices respectively.

$$A = LU \tag{1.2}$$

The solution is then obtained by forward and back substitution steps:

$$Ly = b \tag{1.3}$$

$$Ux = y \tag{1.4}$$

To solve (1.1), a sparse matrix technique based on the following principles is used:

a) Only the non-zero elements of A are stored.

b) Arithmetic operations are performed on non-zero elements only.

c) During the decomposition fill-ins are generated, i.e. new non-zero elements are created in the process of generating zeros below the diagonal. The number of fill-ins is kept small.

The three problems stated above are all related. Even though only non-zeros need to be stored, fill-ins must be stored in the matrix structure. Therefore, minimization of fill-in will result in minimization of the arithmetic operations and storage as well. One must find a permutation of the sparse matrix A to satisfy the above goal. The problem of finding an optimum permutation to minimize fill-in is NP-complete [1], and many heuristic algorithms have been developed to obtain near optimal solutions for this problem. Most of these heuristics find optimum permutations of the matrix which minimize fill-in in sequential solution process while they often minimize the amount of possible parallel work in parallel process. Therefore an ordering, or pivoting strategy to minimize some combination of fill-in and parallel execution time must be determined. The design of a heuristic algorithm which identifies a set of pivots to be processed in parallel while minimizing fill-ins is described in detail in [2], [3], and [4]. Other parallel pivoting strategies have also been suggested [5], [6], [7], [8], [9], [10], [11], [12], [13]. In this paper we concentrate on parallel implementation of sparse LU decomposition procedure using the parallel pivoting technique described in [2], [3], and [4]. In this implementation pivots are tested for numerical stability as well as for sparsity.

A brief description of the parallel pivoting algorithm is given in section 2. Section 3 describes the storage structure used in the implementation. In section 4 the various parallel procedures to perform steps of triangularization are described and analyzed. In section 5 we represent actual performance results from the parallel implementation of the sparse LU decomposition on the HEP computer.

Finally, in section 6 some concluding remarks are presented.

2. Parallel Pivoting Algorithm The Triangulation of an $n \times n$ matrix $A = [a_{ij}]$ can be described by the following procedure.

for $K = 1,2,\ldots,n-1$ and for *each* $a_{jk} \neq 0$

$$a_{jk} \leftarrow \frac{a_{jk}}{a_{kk}} \qquad j > k \tag{2.1}$$

For each pair $a_{ik} \cdot kj \neq 0$

$$a_{ij} \leftarrow a_{ij} - a_{ik} \times a_{kj} \qquad i > k,\ j > k \tag{2.2}$$

In (2.2) if $a_{ij} = 0$ but $a_{ik} \cdot a_{kj} \neq 0$, a fill-in is generated. It is obvious that if we have sufficient processors, the divide operations (2.1) for each column K can be done in parallel. Also, for each k the update operation (2.2) for all pairs $a_{ik} \cdot a_{kj} \neq 0$ can be done in parallel. Our experience in employing this approach has indicated that the sparsity of application matrices leaves parallel processes with little work to perform if only reduction for a single pivot is done in parallel [14]. During Sparse LU decomposition it is possible to perform computation on many diagonal elements simultaneously. In parallel LU decomposition of general unsymmetric sparse matrices several key issues must be considered:

a) Parallelism and fill-in are two competing issues and a balance between the two must be obtained. In other words minimizing fill-ins results in limited parallelism, and maximizing parallelism results in uncontrolled generation of fill-ins.

b) A test for numerical stability of pivots must be made to ensure the accuracy of the solution process.

c) In applications where the sparse linear system must be solved repeatedly, it must be possible to decompose structurally identical matrices using the information produced for the first decomposition of such matrix.

d) A storage structure suitable for parallel processing must be determined.

A heuristic algorithm has been designed in [2], [3], [4] which identifies parallel pivot candidates and allows the matrix to be reduced for multiple pivots simultaneously while it minimizes fill-ins. It is a dynamic algorithm which can be applied at any point in the decomposition phase and does not require a preordering of the input matrix. It allows pivots to be tested for numerical stability. Therefore at any point during the reduction, if numerically unstable pivots are encountered, unsymmetric permutations can be performed. The algorithm can then be applied to the remaining unreduced submatrix. This technique also allows structurally identical matrices to be decomposed using the information generated during the decomposition of the first matrix. In subsequent decompositions a test for numerical stability should be made. If the test is not satisfied, an off-diagonal permutation can be made and the parallel pivoting algorithm can be applied anew to the unreduced matrix only.

Here we will concentrate on parallel implementation of this algorithm and will not go into a detailed description of its design. A complete and detailed description and analysis is available in [2], [3], [4]. In what follows a brief description of the algorithm and the required steps is given. The procedure to implement each step is presented in detail in section 4.

Pivots that can be processed in parallel are related by a compatibility relation and are grouped in a compatible. In other words pivots P_{ii}, P_{jj}, P_{kk} are compatible and can be processed in parallel if and only if elements a_{ij}, a_{ji}, a_{ik}, a_{ki}, a_{jk}, a_{kj} are all zero. The collection of all maximal compatibles [15], [16] yields different maximum sized sets of pivots that can be processed in parallel. Several methods for generating maximal compatibles exist and they are all based on the construction of an implication (incompatible) table. The incompatible table gives information about pairs of incompatible pivots. Production of all maximal compatibles involves a binary tree search and is exponential in the order of the matrix. This problem is solved by a technique which generates an "*ordered incompatible table*" based on the Markowitz number [17] of the pivot candidates.

The Markowitz criterion is a heuristic for minimizing fill-ins in sparse matrices in sequential programming. It is based on the fact that at each step k, the maximum number of fill-ins generated by choosing a_{ij} as pivot is $(r_i-1)(c_j-1)$, where (r_i-1) and (r_j-1) are the number of nonzeros other than a_{ij} in row i and column j of the reduced matrix. Markowitz selects as pivot element at step k, the element which minimizes $(r_i-1)(c_j-1)$, which is called the Markowitz number of element a_{ij}.

An "*ordered Compatible*" can then be produced directly from the *ordered incompatible table* without the need to search the tree. The resulting set of compatible pivots has the property of generating few fills. The heuristic algorithm combines the idea of an *ordered compatible* with a limited binary tree search to generate several sets of compatible pivots in linear time. An *elimination set* to reduce the matrix is generated and selected on the basis of a minimum Markowitz sum number (sum of the Markowitz number of pivots in a compatible). Several parameters are introduced to trade off parallelism for fill-in which can be controlled by the program. In summary the algorithm requires the following steps:

1. An incompatible table is constructed by scanning the sparse matrix.

2. Pivots are ordered according to their Markowitz numbers.

3. A limited binary tree search produces several starting sets at a given level (ULEVEL) of the tree.

4. An *ordered compatible* is generated for each starting set at ULEVEL from the corresponding *ordered incompatible table*.

5. The *ordered compatible* of maximum size and minimum Markowitz sum is selected as the *elimination set* to reduce the matrix.

6. A set of program parameters can be applied to the resulting *elimination set* to further minimize fill-in.

3. Storage Structure The basic global data structure used in the parallel LU decomposition program is described below. Each element of the matrix structure consists of five fields: the real numerical value, the row index, the column index, a pointer to the next element in the row, and a pointer to the next element in the column. The incompatible table is represented by an array of dimension n, order of the matrix, with elements of the array *imptbl* being sets of n elements each. Each set corresponds to a column of the table. Column i of the table, $imptbl_i$, holds the incompatible information for pivot i of the matrix. Note that the parallel pivoting algorithm considers only the diagonal elements as pivot candidates. Unsymmetric permutations are possible in between parallel pivoting steps. *compst* holds the resulting *elimination set*.

```
Type Definition:
  ptr= matpac;  pointer type to a matrix element.
  matpac = record
        val    : real;     real value.
        row    : integer;  row index.
        column : integer;  column index.
         nc    : ptr;      pointer to next element in row.
         nr    : ptr;      pointer to next element in column.
        end;
  rocl=(r,c);   row and column list.
  sets=set of 1..n;  set type

Variables:
  A: array(rocl,1..n)of ptr;   matrix structure.
  nofr,nofc: array(1..n)of integer; number of nonzeros
                                in row and column.
  imptbl: array(1..n)of sets;  incompatible table.
  compst: sets;               elimination set.
```

4. Parallel LU Decomposition In order to write efficient parallel programs one must consider the underlying parallel architecture to which the program is to be applied. In an MIMD environment parallelism must be applied at the highest possible level in the program in order to effectively exploit the underlying parallel hardware. In our design and implementation we have used the idea of universal parallelism due to Jordan [18], [19] which is based on writing parallel programs assuming that all the parallelism needed by the programmer exists throughout the program execution. A set of parallel programming constructs known as "the Force" implemented for several shared-memory MIMD computers [18], [19] are used in the implementation of the algorithms presented in this paper.

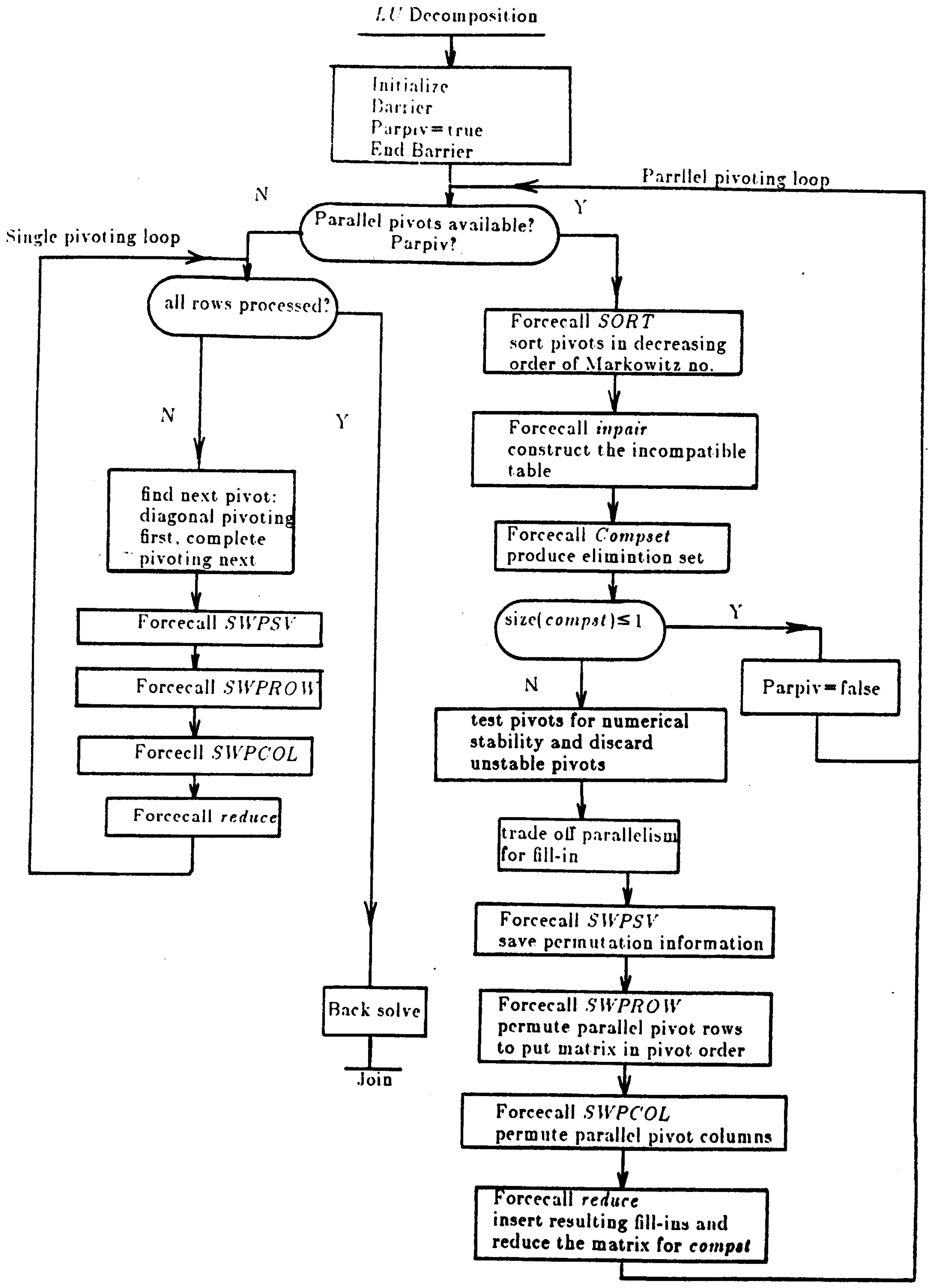
A high level block diagram of the program is given below. The entire *LU Decomposition* program is executed by *NPROC* processes. These processes can be created by a driver routine. The parallel routines are specified by a *Force-call* followed by a brief description of their function on each box. Therefore the body of each Force subroutine is executed by *NPROC* processes in parallel. After the program is completely executed, the parallel processes are joined in the driver.

The flowchart consists of two major loops, parallel pivoting loop and single pivoting loop. The parallel pivoting loop is executed as long as the program can find compatibles of more than one pivot, otherwise the single pivoting loop is executed. During parallel pivoting steps only diagonal elements are considered as pivots and unsymmetric permutations are not permitted. In single pivoting steps unsymmetric permutations are allowed and hence any matrix element can be considered as pivot.

In the remainder of this section we describe the parallel algorithms involved in the *LU Decomposition* program by stepping through the flowchart in the given order.

4.1. Parallel Sort A sorting routine is required to sort the pivots in decreasing order of Markowitz number. The ordered list of pivots is used at several points in the parallel pivoting algorithm: in the construction of an *ordered incompatible table*, in the construction of a partial binary tree search, and finally it is used to trade off parallelism for fill-in by discarding a fraction of compatible pivots with Markowitz number higher than a given threshold value in the ordered list.

The sorting algorithm used for this purpose is the Batcher sort [20], [21]. Batcher's sorting scheme is somewhat like Shell's sort but, the comparisons are made in a novel way so that no propagation of exchanges is necessary. The amount of bookkeeping needed to control the sequence of comparisons is rather large. All comparisons/exchanges specified by a given iteration can be done simultaneously. In the procedure below processes are prescheduled over a range of indices and they perform the comparison/exchange operations in parallel. As can be seen from the algorithm at each iteration we must compute the range of


LU Decomposition
Initialize
Barrier
Parpiv=true
End Barrier
Parrllel pivoting loop
N
Y
Parallel pivots available? Parpiv?
Single pivoting loop
all rows processed?
N
Y
find next pivot: diagonal pivoting first, complete pivoting next
Forcecall SWPSV
Forcecall SWPROW
Forcecll SWPCOL
Forcecall reduce
Back solve
Join
Forcecall SORT sort pivots in decreasing order of Markowitz no.
Forcecall inpair construct the incompatible table
Forcecall Compset produce elimintion set
size(compst)≤1
Y
N
Parpiv=false
test pivots for numerical stability and discard unstable pivots
trade off parallelism for fill-in
Forcecall SWPSV save permutation information
Forcecall SWPROW permute parallel pivot rows to put matrix in pivot order
Forcecall SWPCOL permute parallel pivot columns
Forcecall reduce insert resulting fill-ins and reduce the matrix for compst

nonadjacent pairs for comparison. This is a rather large overhead but is performed in parallel by processes In between the iterations however, a large section of sequential code is needed to adjust the range of indices for the next iteration.

```
Procedure Batcher sort
Global $t,p,q,r,d$

Barrier [1]
$t= \lceil \log n \rceil$

$p= 2^{t-1}$
$q= 2^{t-1}$ , $r=0$, $d=p$;
End barrier
while ($p > 0$) do
  begin
  Presched DO [2] $i=1, n-d$
   (compute correct index)
   $q= (i-1)/p$
   $j= p \cdot q + r + i$
   if($j \leq (n-d)$) then
      compare and exchange;
  End Presched DO
  Barrier
  if( $p \neq q$ ) then
     $d= q-p$
     $q= q/2$
     $r= p$
  else
     $p= \lfloor p/2 \rfloor$
  endif
  End barrier
  endwhile
```

It has been shown that with enough parallel operations, sorting is completed in $1/2 \lceil \log n \rceil (\lceil \log n \rceil + 1)$ steps. The sequential work and the small critical section used in implementation of the barrier construct will dominate the parallel work unless n is very large.

[1] The semantics for Barrier construct are such that all processes pause when they reach the Barrier. After all have arrived, one process executes the section of code enclosed by Barrier-End barrier pair. After the singly executed code section is complete, all processes will resume execution after the End barrier.

[2] Presched DO loop causes the body of the loop enclosed between it and the matching End Presched DO to be executed in parallel for different values of i. Instances of the loop body must be independent for different values of i.

4.2. Parallel Inpair The *incompatible table* is constructed in this routine. Each column of this table corresponds to a pivot of the matrix and contains the list of pivots incompatible with the pivot under consideration. This information is used in the construction of the partial binary tree search described in the next section. Assume pivots are numbered 1 to n corresponding to diagonal elements of rows 1 through n of the matrix ordered with decreasing Markowitz number. Column i of the incompatible table corresponds to pivot number i of the matrix. Each column of the table can be constructed independently by a parallel process. Parallel processes are prescheduled over a loop of indices (i) corresponding to diagonal pivots of the matrix. Each process, say i, scans the row-column pair corresponding to pivot i. If a nonzero element a_{ij} or a_{ji} is encountered a mark for pivot j is entered in row j of column i of the incompatible table, indicating pivot j is incompatible with pivot i. No process synchronization is required since each process is responsible for scanning row-column pairs of different diagonal elements and updating the corresponding columns of the table. Figure 4.1.b shows the *ordered incompatible table* for the sparse matrix of Figure 4.1.a. The algorithm can be described as:

```
Procedure Incompatible table
Global imptbl[1..n] of set 1..n;
Global n,nrem ;
Presched DO $i = nrem,n$
  scan row $i$ for any nonzero $a_{ij}$
     if $P_j$ not in imptbl($i$) then
     add $P_j$ to imptbl($i$)
  scan col $i$ for any nonzero $a_{ji}$
     if $P_j$ not in imptbl($i$) then
     add $P_j$ to imptbl($i$)
End Presched DO
```

The construction of the incompatible table requires scanning NZ nonzeros of the matrix. As can be seen from the procedure the only set operations required are addition of a new element to a set and a test for membership. These

	1	2	3	4	5	6	7	8	9	10	11
1	x										
2		x									
3			x					x			x
4				x		x					
5	x				x					x	
6		x				x					
7				x			x		x		x
8			x		x			x	x		
9						x	x		x		x
10				x		x		x	x	x	
11	x					x		x	x		x

Matrix A1

Pivot	Markowitz Number	Order
1	0	9
2	0	11
3	2	8
4	2	6
5	2	10
6	4	7
7	3	3
8	9	4
9	12	5
10	4	1
11	12	2

Pivots Ordered with Markowitz Number

Figure 4.1 a

	9	11	8	6	10	7	3	4	5	1
11	x									
8	x	x								
6	x	x								
10	x		x	x						
7	x	x								
3		x	x							
4				x	x	x				
5			x		x					
1		x							x	
2				x						

Figure 4.1.b Ordered Incompatible Table

operations are $O(1)$, therefore the incompatible table can be constructed in $O(NZ/NPROC)$ time with $NPROC$ parallel processes.

4.3. Parallel Compset The procedure that produces the ***ordered compatibles*** has two major parts. The first part generates several starting sets at a given level (ULEVEL) of the binary search tree. The second part produces an *ordered compatible* for each of the starting sets from the incompatible table.

The binary tree search is a systematic approach for extracting the maximal compatibles. Initially, it is assumed that all pivots are compatible. They are grouped in one set consisting of all pivot (diagonal) elements. This set, S, will be at the root of a binary tree, level zero. Next, the set of pivots incompatible with the pivot of minimum Markowitz number, P_j, obtained from the incompatible table, $imptbl_j$ is used to split S into a left S_1 and a right S_2 set, constituting level one. S_1 consists of all elements of its parent S except those incompatible with P_j. S_2 consists of the same elements as S except P_j itself. At each level of the binary tree sets are produced by splitting the parent set into left and right sets, taking pivots in increasing order of Markowitz number from the ordered list of pivots to split the sets. This process continues until we have produced all starting sets,

$S_1^{ULEVEL-1}$ through $S_2^{ULEVEL-1}$ at level = ULEVEL. The partial binary tree search for the example matrix of Figure 4.1 and for ULEVEL=3 is shown in Figure 4.2. As We can see eight starting sets are produced for this level of the tree. Note that set 5 and 6 are the same as their parent set simply because the parent set could not have been split for pivot number 10. Different orderings of pivots for splitting the nodes of the binary tree are considered in [2], [3], and [4].

In the second part of this procedure an ***ordered compatible*** is generated for each of the starting sets. This is done by scanning the incompatible table corresponding to each starting set in decreasing order of Markowitz number of pivots in the starting set.

The incompatible table for a given starting set, S_i, is the original table with those rows and columns corresponding to pivots absent from S_i eliminated.

For each starting set, S_i, its corresponding incompatible table is scanned. Any pivot P_j whose corresponding column in the incompatible table, $imptbl_{P_j}$, is null is added to the *ordered compatible*, $compset_i$. In addition any pivot P_j for which $imptbl_{P_j} \bigcap compset_i = empty$ is also added to $compset_i$ since $compset_i$ does not contain any pivots incompatible with P_j. Finally the ***ordered compatible*** of maximum size and minimum Markowitz sum is selected as the ***elimination set*** to reduce the matrix. The ***ordered compatibles*** corresponding to the eight starting sets above are given in Figure 4.3. Any of these sets can be selected to reduce the matrix in parallel. Among these ***ordered compatibles*** $compset_5$, $compset_6$, and $compset_8$ are of maximum size (5). The set with minimum Markowitz sum will tend to generate fewer fill-ins. Therefore $compset_5$ or $compset_6$, which ever is produced first, will be selected as the ***elimination set***.

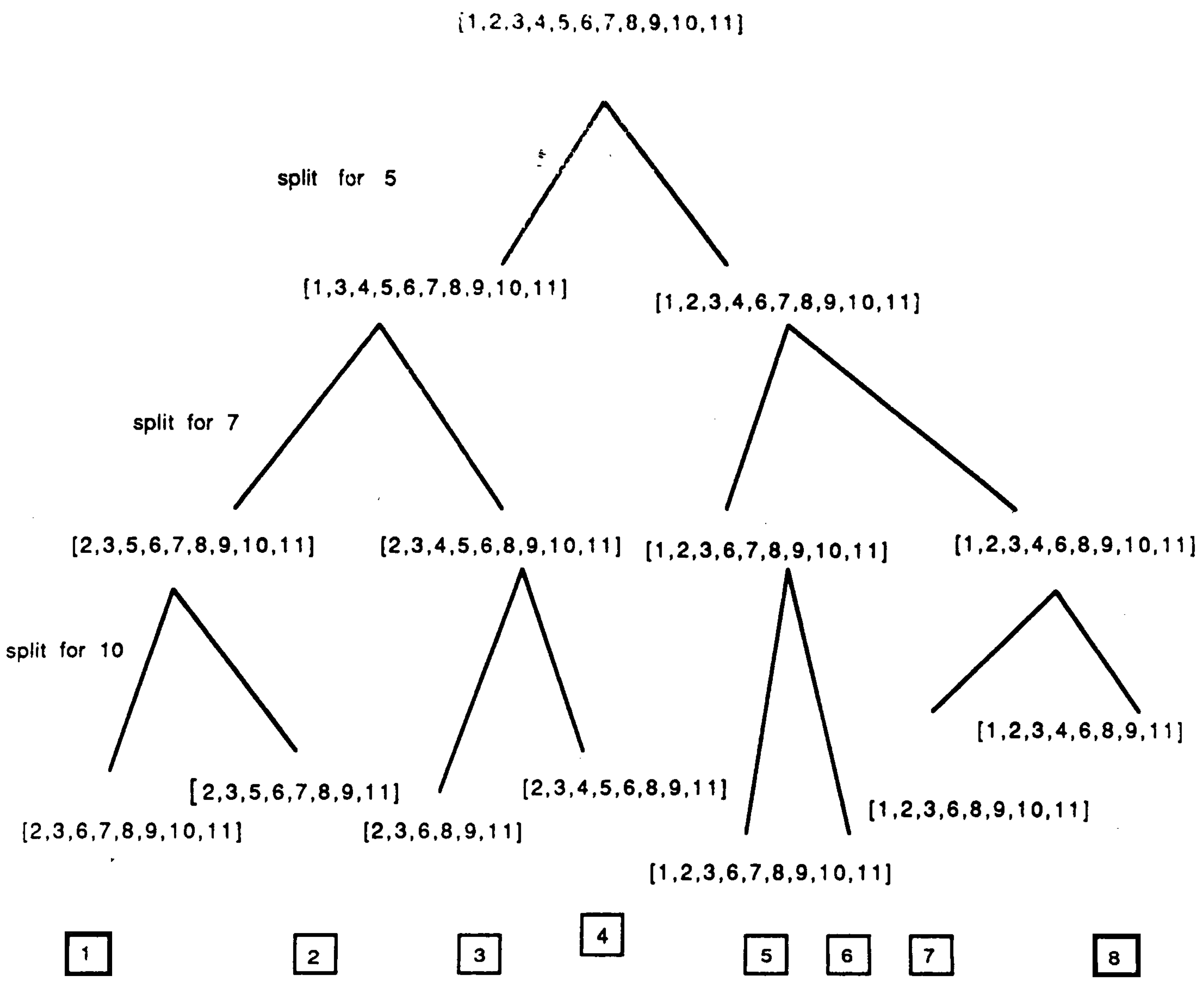


Figure 4.2 Partial Binary Tree Search

$compset_1$ = [2,3,7,10],	Markowitz sum = 9
$compset_2$ = [2,3,5,7],	Markowitz sum = 7
$compset_3$ = [2,3,9],	Markowitz sum = 14
$compset_4$ = [2,3,4,5],	Markowitz sum = 6
$compset_5$ = [1,2,3,7,10],	Markowitz sum = 9
$compset_6$ = [1,2,3,7,10],	Markowitz sum = 9
$compset_7$ = [1,2,3,10],	Markowitz sum = 6
$compset_8$ = [1,2,3,4,9],	Markowitz sum = 14

The *Ordered Compatibles* and Their Markowitz Sum Number
Figure 4.3

To produce the starting sets at ULEVEL, processes are assigned to the nodes of the partial binary tree from the root to level ULEVEL-1. A process cannot start to split a set until the set is produced by the parent process. To accomplish this, a lock is assigned to each set from the root to ULEVEL-1. The lock is initialized to false except for the root set. As soon as a process has completed generation of a child set, it sets the lock for the child set to true allowing the next process to proceed. This is done by selfscheduling processes over the work. Production of starting sets as described above is embodied in the first self-scheduled loop in the algorithm below.

```
Procedure Compset
Global lock(nset);    nset :number of sets from root to $ULEVEL-1$.
Global imptbl(1,n) set of 1..n;
Global $S(1..2\times nset)$;
Global compst(1,nset) set of 1..n;
Local less set of 1..n;
Local tempset set of 1..n ;

Selfsched DO [3] $i=1,(2^{(ULEVEL-1)}-1)$
  wait until lock(i) true;
  take the next pivot, $P_j$, with lowest
  Markowitz number to split $set_i$ :
    produce left set, set $lock(2\times i)$ to true;
    produce right set, set $lock(2\times i+1)$ to true;
End Selfsched DO
Barrier
End barrier
For each starting set, $S_i$, produce an ordered compatible, $compset_i$


Presched DO $i= 2^{ULEVEL-1},2^{ULEVEL}-1$
  $compset_i$ = empty
  less = $S-S_i$
  for $j=$ n down to 1 do
    begin
    if ( $P_j \in S_i$ ) then
      begin
      tempset = $imptbl_j$ - less
      tempset = tempset $\bigcap$ $compset_i$
      if ( tempset= empty ) then
        $compset_i$ = $compset_i$ + $[P_j]$
      end
   end
   find a local maximum.
 End Presched DO

 Critical [4] max
 find a global maximum
 End critical
```

Generation of *ordered compatibles* is done by prescheduling processes over the sets at ULEVEL. Each process is responsible for keeping an updated copy of the *ordered compatible* of maximum size and minimum Markowitz sum it pro-

[3] A process takes the next unassigned value of i as soon as it is free. This tends to even the work load over processes when the execution time of the loop can vary significantly for different i values.

[4] Mutual exclusion is accomplished by critical sections, begun by a Critical statement and ended by End critical.

duces. In order to do this, processes execute a section of code to obtain a local maximum. After processes have completed the execution of the prescheduled loop body, they execute a critical section to obtain a global maximum (*elimination set*).

Production of K starting sets for a given ULEVEL takes a constant time. For ULEVEL small and constant compared to n, generation of *ordered compatibles* from starting sets is of order n set intersection and difference operations. Assuming efficient implementation of the set operations is available, $O(setop)$, the heuristic algorithm has a complexity of $O(K \cdot n \cdot setop)$, where *setop* can be assumed to be constant. Employing *NPROC* processes will reduce the execution time of the second prescheduled loop by 1/*NPROC*. Of course, the complete execution time cannot be improved by 1/*NPROC* because of the synchronization code used in waiting for locks to become true in the barrier code and in the critical section to find a global maximum. As ULEVEL is increased the number of parallel processes that can be effectively used increases but at the same time the complexity of the algorithm will increase. It is important to choose ULEVEL such that the amount of parallelism provided by the underlying hardware is effectively exploited to speed up the execution time and not to add to its size.

4.4. Numerical Stability and Trade off Parameters

A test for numerical stability is done by prescheduling processes over the parallel pivot candidates in the *elimination set*. Each process searches its pivot column for the maximum entry, *Vmax*. A pivot is numerically stable if:

$$\textit{pivot tolerance} < |\textit{pivot value}| \quad \text{and}$$
$$u \times |Vmax| < |\textit{pivot value}|$$

pivot tolerance and *u* are user defined values which define the desired accuracy. If a pivot does not satisfy the test, it is discarded from the *elimination set*. When no more parallel pivots exist, i.e. the single pivoting loop, if unstable pivots are still present an acceptable pivot is obtained by a complete pivoting strategy. Thus an unsymmetric permutation is performed to put the matrix in the new pivot order. Note that for matrices in which diagonal pivoting becomes impossible at an early stage during the decomposition, it is possible to switch to single pivoting code for a few steps. During single pivoting steps unsymmetric permutations are possible and will change the matrix structure. So parallel steps may become possible again.

In previous papers we have shown that it is possible to minimize generation of fill-ins significantly by reducing the amount of parallel work slightly according to some criteria [2], [3], [4]. Trading off parallelism for fill-in is done according to the size of the *elimination set* and a number of parameters:

1. Shrinkage parameter: By allowing a small percentage of the *elimination set* to be discarded we can control the number of compatible pivots to a degree that does not limit our parallel work by too much.

2. Upper limit parameter: This limit would allow just enough parallel work to keep our parallel processes busy.

3. Threshold parameter: In shrinking the size of an *elimination set* only pivots with Markowitz number higher than a threshold value in the ordered list of pivots may be discarded. Pivots with low Markowitz numbers do not tend to generate many fills and need not be discarded.

If trade off is possible then pivots are discarded from the *elimination set* asynchronously by parallel processes. Of course it is not necessary to use a very tight synchronization. It is possible to calculate the number of pivots with the highest Markowitz number to shrink the *elimination set* and to let parallel processes to discard these pivots without synchronization. This approach is more parallel but less flexible on the size of the resulting *elimination set*.

4.5. Parallel SWPSV Once parallel pivots are determined the matrix is permuted to the new pivot order. The permutation information is saved in procedure *SWPSV*. The destination addresses of parallel pivot candidates are stored such that at each parallel step rows and columns are swapped only once. At each step indices of pivots in the *elimination set* are checked against the upper left hand corner of the unreduced matrix. Let *nrem* be the index of the next row to be reduced in the remaining unreduced matrix. Let *npiv* be the number of pivots in the *elimination set*. If i is the index of a pivot in the *elimination set*, then if

$$i < nrem + npiv - 1$$

there is no need to permute row and column i. Therefore pivot i can be marked to avoid unnecessary permutations. This is accomplished in a prescheduled do loop over the pivots in the *elimination set*. Having identified the necessary permutations, the required information is stored in permutation vectors in a self-scheduled loop. Each process obtains the index of a row to be swapped and updates the corresponding entries in the permutation vectors. This is a simple routine and is parallelized over the compatible pivots in the *elimination set*(*npiv*). The order of the sequential routine is $O(npiv)$, since it only involves exchanges of entries in the permutation vectors for parallel pivot candidates.

4.6. Parallel SWPROW The actual permutation of rows and columns is performed by routines *SWPROW* and *SWPCOL*. Parallelism could be most effective if rows of the parallel pivots are completely permuted first followed by column permutations. A single step row-column permutation would involve many changes in the row and column lists of the matrix structure and would require tremendous amount of synchronizing code Thus *SWPROW* permutes all the pivot rows in parallel, and *SWPCOL* permutes all pivot columns. Barrier synchronization is necessary in between the calls to the two routines. These routines are symmetric in the function they perform, so only *SWPROW* is described.

The rows and columns of the pivots in the ***elimination set*** are to be permuted with others in the remaining matrix such that the matrix is in the pivot order with any ordering of elements within an ***eliminationset***. In permuting parallel pivot rows the next row pointer and the row index fields must be updated. So after all parallel pivot rows are swapped with their destination rows, each column of the matrix will be in increasing order of row indices obtained from the permutation vector. This suggests that we can sort the columns of the matrix according to the new ordering given by the permutation vector. Of course not every matrix column has to be sorted. Only columns having a nonzero element in any of the rows involved in permutation must be sorted. Therefore by constructing a bit vector which is the result of the union of the boolean vectors corresponding to the permuting rows (parallel pivot rows and their destination rows), we gather the indices of columns to be sorted. Construction of the boolean vectors for parallel pivot rows and their destination rows is done in parallel be prescheduling processes over these rows. The union operation is then performed sequentially. Note that the union could be done in parallel using a parallel tree sum computation method. This would involve much storage for the intermediate results but would speed up the operation.
Next, every column having bit position in the resulting bit vector set must be sorted. Each column consists of very few nonzero elements due to the sparsity of the matrix and hence a simple bubble sort can be used efficiently to sort the columns. The sorting of columns are independent operations and can be done simultaneously by parallel processes. This is done by self-scheduling the processes

over the work. Each process executes a small critical section to obtain the index of the next column to be sorted. The algorithm description follows:

```
    Procedure $SWPROW$

    Global $brow$: $array[1..2\times npiv]$ of sets;
                       bit vectors of rows to be permuted.
    Global $colindex$: sets;       bit vector of column indices
                       to be sorted.

    Presched DO $i=1,\ 2\times npiv$
     (initialize the boolean vectors)
     $brow(i)=0$
    End Presched DO
    Barrier
    End barrier
    Presched DO $i=1,\ 2\times npiv$
     obtain index, $j$, of the row to be permuted.
     scan row $j$ and for each nonzero $a_{jk}$
         add $k$ to $brow(i)$
    End Presched DO
    Barrier
    $colindex = brow(1)\bigcup brow(2)\bigcup \cdots \bigcup brow(2\times npiv)$
    End barrier
  sortnext:
    Critical $nextcol$
    get a local column index, $j$, from $colindex$ to be sorted.
    End critical
    if ($j$ is a valid index) then
      bubble sort column $j$ using the information
      from the permutation vector.
      go to sortnext
    endif
```

The sort is $O(nz^2)$, where nz is an average number of nonzeros per row or column. The sort must be done for all columns in the *colindex* vector. This number is usually a multiple of nz, say Knz, and in the worst case could be n, the order of the matrix. It also involves the set operations union and the next element from *colindex* which is implementation dependent. The next element operation is performed within the loop and can be done in $O(1)$. Thus on the average the number of operations in this routine is of order of:

$$O(K\cdot nz^3)$$

If $K \cdot nz$ parallel processes exist, *SWPROW* can be done in time $O(nz)^2$.

4.7. Parallel reduce The numerical decomposition and insertion of fill-ins for the ***elimination set*** is parallelized by self-scheduling processes as follows:

The reduction of each row for a pivot in the ***elimination set*** is performed by a parallel process. If no more rows are left to be reduced for this pivot, the reduction process for the next row can be started by parallel processes looking for more work. The logic to do this is contained within a critical section of code. A process obtains a local pointer to the next row having a nonzero in the pivot column it is processing, advances the global pointer and exits the critical section. If no more rows are left to be updated for the pivot under consideration, the process advances a global pointer to the next pivot in the ***elimination set*** and obtains the next row pointer for the new pivot in the same manner and exits the critical section. Thus processes work in parallel over rows of a single pivot first and over the parallel pivot candidates in the ***elimination set*** next.

Each process is responsible for dividing the nonzero element in the pivot column by the pivot and subtracting a multiple of the pivot row from the row, j, being updated. The process must also check for a possible fill-in and insert it if necessary. The search for a possible fill-in must be done atomically so that parallel processes do not try to insert the same element in the same position with different values. Means must be provided to allow only one process to insert a fill-in a_{jk} and others to update the element after it is inserted. This can be done by locking the row j and column k of the matrix such that it can only be searched by one process at a time. The locking of a row and column is done by a critical section on elements of two asynchronous arrays, one for rows and one for columns. Any two shared arrays, for example *nofr* and *nofc*, that are not used throughout this process can be used for this purpose. Of course only one element can be inserted in the matrix structure at a time since the insertion causes changes in the pointer structure for rows and columns other than j and k alone. It is important to note that the probability of searching for the same element by more than one process at the same time is very low. The synchronization described above is necessary for correct solution and does not increase the execution time by much. The updating of an element must also be done atomically by processes. Again the probability of more than one process trying to simultaneously update the same element is low. This synchronization can be done by simple scoreboarding which must be available in machines matching our computational model.

```
 Procedure reduce

 Barrier
 $ii \leftarrow$ index of the first pivot in the ordered matrix.
 End barrier
getwork:
 Critical next
   (get next parallel work)
   $i = ii$ ;        local pivot index
   (get a local row index to be reduced)
local:
   $j =$ nextrow in pivot column $ii$
   if($j$ not valid) and (more pivots) then
   begin
     $i = ii + 1$ ; advance to the next pivot and update
     $ii = i$ , the global pivot pointer.
     goto local; get the next row.
   endif
   update global nextrow pointer information.
 End critical
 if( $j$ valid) then
 begin
   $a_{ji} = a_{ji}/a_{ii}$;    divide by pivot
   scan pivot row $i$ and for each $a_{ik}$:
     (check for a possible fill-in)
     (lock row $j$ and column $k$ )
     Critical $nofr(j)$
     Critical $nofc(k)$
       if($a_{jk}$ not in matrix) then
         Critical insert
         insert the element
         End critical
     End critical
     End critical
   atomically update $a_{jk}$
   $a_{jk} = a_{jk} - a_{ji} \times a_{ik}$
   goto getwork
 endif
```

Reduction of the matrix for a single pivot requires a complete scan over the matrix which can be done in time NZ. For *npiv* pivots the time would be proportional to *npiv*$\cdot NZ$. NZ changes as fill-ins are encountered or as the matrix gets smaller due to the reduction. Duff [4] reports that over a wide variety of matrices, the number of arithmetic operations performed has been empirically observed to be about $\tau^2/4n$ where τ is the number of nonzeros in the decomposed form. The value of τ is generally not known a priori. Experience has shown that a value

$5/2NZ$ is a satisfactory estimate for τ although an estimate of order $n\log n$ is more realistic for problems arising from PDEs in two dimensions.

5. Implementation Results The program is implemented for the HEP (Heterogeneous Element Processor) pipelined shared-memory computer built by Denelcor, Inc. [22]. The results presented here are on a single PEM (Process Execution Module). The execution pipeline on the HEP has eight steps. In the HEP the degree of simultaneous execution is limited by the length of the various pipelines and may be characterized by an average pipeline length. Thus on a single PEM a program may be expected to speed up by no more than 7.5 to 9.5 over single stream execution. The *LU Decomposition* program has also been simulated on a Vax 11/780 and tested on many application matrices arising from electronic circuits and structural analysis producing successful results [2], [3], [4]. Here we represent the timing results of running the program over a 144 by 144 matrix from the circuit of an 8-bit full adder and employing different values for trade off parameters. Figure 5.1 represents the execution time of *LU Decomposition* program for different numbers of processes from 1 to 25. The result is for the case when maximum parallelism is used. For $NPROC=1$, the matrix is completely reduced in 10 parallel steps. The number of compatible pivots at each step is 72, 25, 16, 11, 6, 5, 3, 2, 2, and 1 respectively. Note that in the first step half of the matrix is reduced in parallel. The execution time decreases with an increase in the number of processes up to $NPROC=11$. In fact there is $1/NPROC$ reduction in execution time for small values of $NPROC$ as new processes make efficient use of the execution pipeline. This decrease in execution time bottoms out as the pipeline becomes full. The slope of the linearly rising tail of the curve indicates the length of time spent in critical sections in various points in the program. A com-

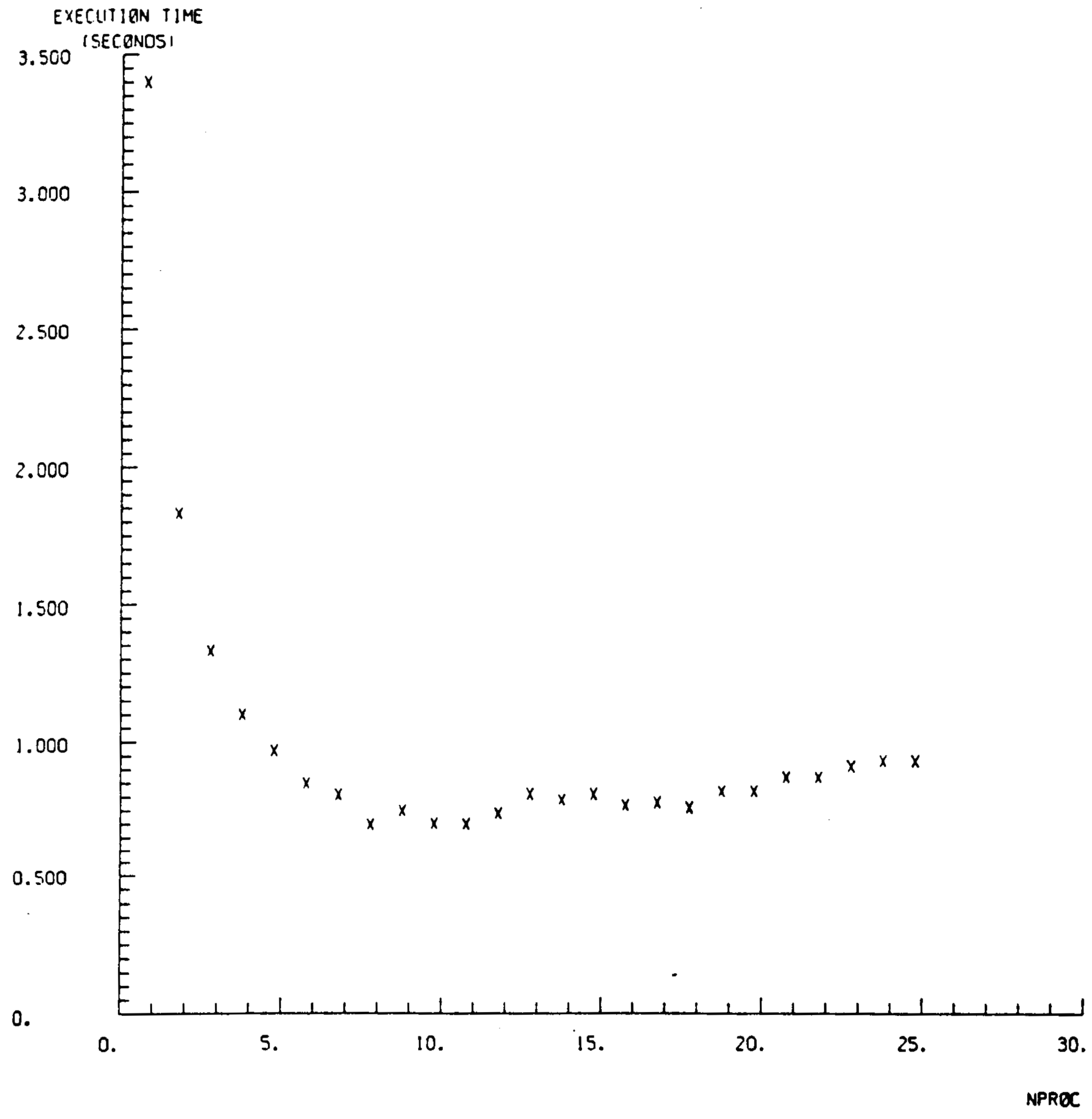


Figure 5.1 Execution Time vs. Number of Processes
No Trade off
144×144, NZ = 616, 8-Bit Full Adder

plete model for analysis of parallel programs can be found in [23]. Defining the speed up to be:

$$S = \frac{T(1)}{T(NPROC)}$$

where $T(1)$ is the time to execute the program with one process and $T(NPROC)$ is the same time using $NPROC$ processes. Then a speed up of 4.82 is obtained for 11 processes. Note that this is not speed up measured with respect to the best sequential algorithm, but only gives insight to the parallelism in this program.

For a small number of processes, execution time versus number $NPROC$ of processes can be represented as:

$$T(NPROC) = C_1 + \frac{C_2}{NPROC}$$

where C_1 represents the sequential portion of the work and C_2 the parallel portion. A simple least squares fit to determine C_1 and C_2 is applied to a linear portion of the execution time versus $NPROC$ curve to estimate the degree of parallelism. This analysis shows that the code is 87% parallel. Figure 5.2 shows the execution time versus $NPROC$ for individual routines. As can be seen there is a sharp increase in the slope of the sort curve for large $NPROC$ which indicates parallel processes spend more time in the critical section than doing useful parallel work. Of course a reason for this behavior is the small value of N, order of the matrix. As N increases the slope becomes smaller. A speed up of 3.3 for 8 processes is obtained for the sort. The degree of parallelism for this routine is 58%.

For the ***Compset*** routine speed up is 5.7 for 16 processes, and the code is 93% parallel. Here the value of ULEVEL is 4, so there are 16 starting sets for which an ***ordered compatible*** must be produced in parallel. Therefore every time number of active processes divides 16 evenly a sharp decrease in execution time is observed. Speed ups for ***reduce*** and $SWPROW$ are 4.4 for 11 processes and 6.5 for 12

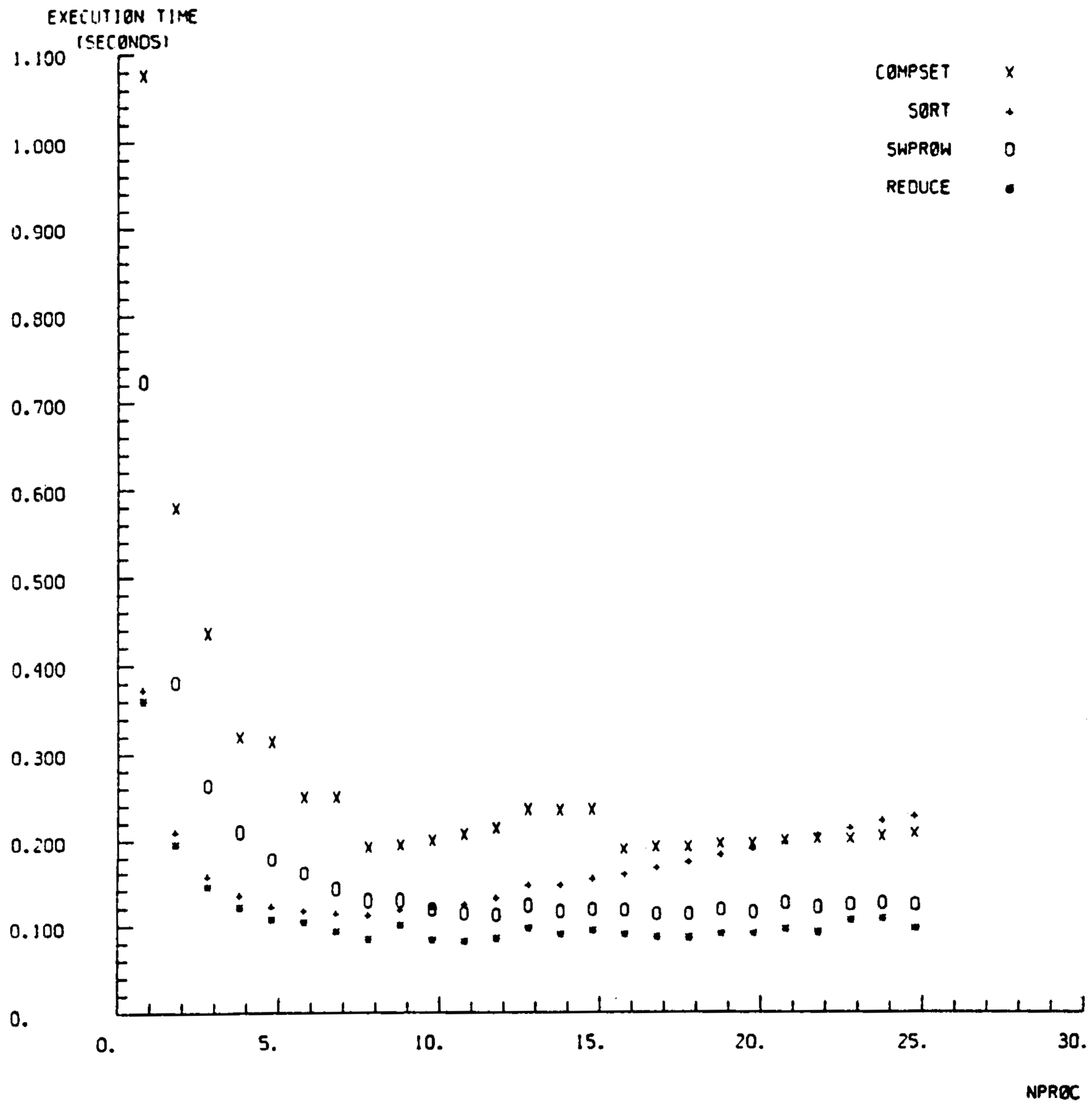


Figure 5.2 Execution Time vs. Number of Processes for Individual Routines, No Trade off 144× 144, NZ=616, 8-Bit Full Adder

processes respectively. The degree of parallelism for these two routines are 81% and 93% respectively. Our numerical results about number of available parallel pivots indicate the existence of many parallel operations and it is clear that by moving to computers with more parallel units or PEMs in the case of HEP a higher degree of parallelism can be achieved.

The program is nondeterministic when executed in parallel. In many cases there are several sets of equal maximum size and minimum Markowitz sum. Depending on the number of processes and their relative speeds, one of the candidate ***ordered compatibles*** will be selected as the ***elimination set***. Thus different results are produced. The number of fill-ins generated for single stream execution is 280. For different values of *NPROC* this number is in the range of 89% to 103% of the fill-ins produced sequentially.

Table 5.1 shows the result of running the program on the same matrix when trade off parameters are used. The values of parameters for this case are given below:

Threshold	1/3
Shrinkage Parameter	30%
Upper Limit	25
ULEVEL	4

Table 5.1

Routine	*NPROC*	Speed up
LU Decompose	9	5.81
Batcher	7	3.5
Compset	18	6.1
SWPROW	14	6.54
reduce	9	4.15

The higher speed up indicates that by employing the above parameters a better balance between number of compatible pivots generated at different steps is achieved. A reduction of 23% in fill is obtained as the result of the above parameter variations which compares reasonably with results from the best sequential program (166). The fill-in can further be decreased by assigning different values to trade off parameters.

The results of running the program on a 505 by 505 matrix produced from SPAR, a structural analysis program [24] is given in Figure 5.3 and 5.4. In the HEP, for every new process a local stack area is allocated. This area depends on the amount of local storage and some other system parameters. Due to the limited size of the available memory the program could only be run on this matrix for up to 7 processes. The trade off parameters for this run have the following values:

Threshold	2/3
Shrinkage Parameter	40%
Upper Limit	60
ULEVEL	4

Using a least squares fit the degree of parallelism for each routine is summarized in Table 5.2.

Table 5.2

Routine	Degree of Parallelism
LU Decompose	95%
Batcher	93.37%
Compset	91.45%
SWPROW	92.22%
reduce	85.44%

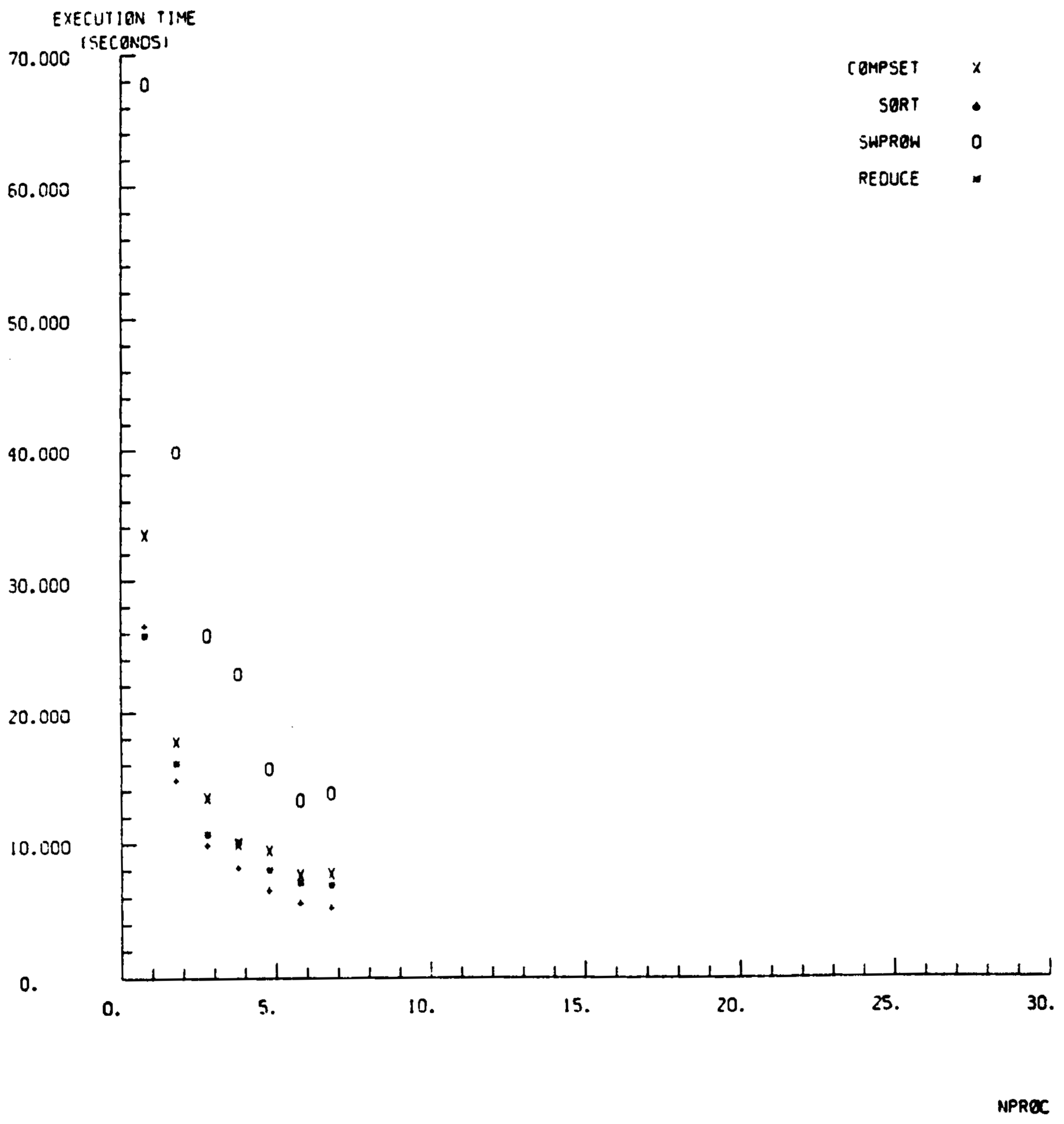


Figure 5.3 Execution Time vs. Number of Processes for Individual Routines
ULEVEL = 4, Shrinkage = 40%, Upper Limit = 60, Threshold = 2/3
505 × 505, NZ = 5889, from SPAR Program

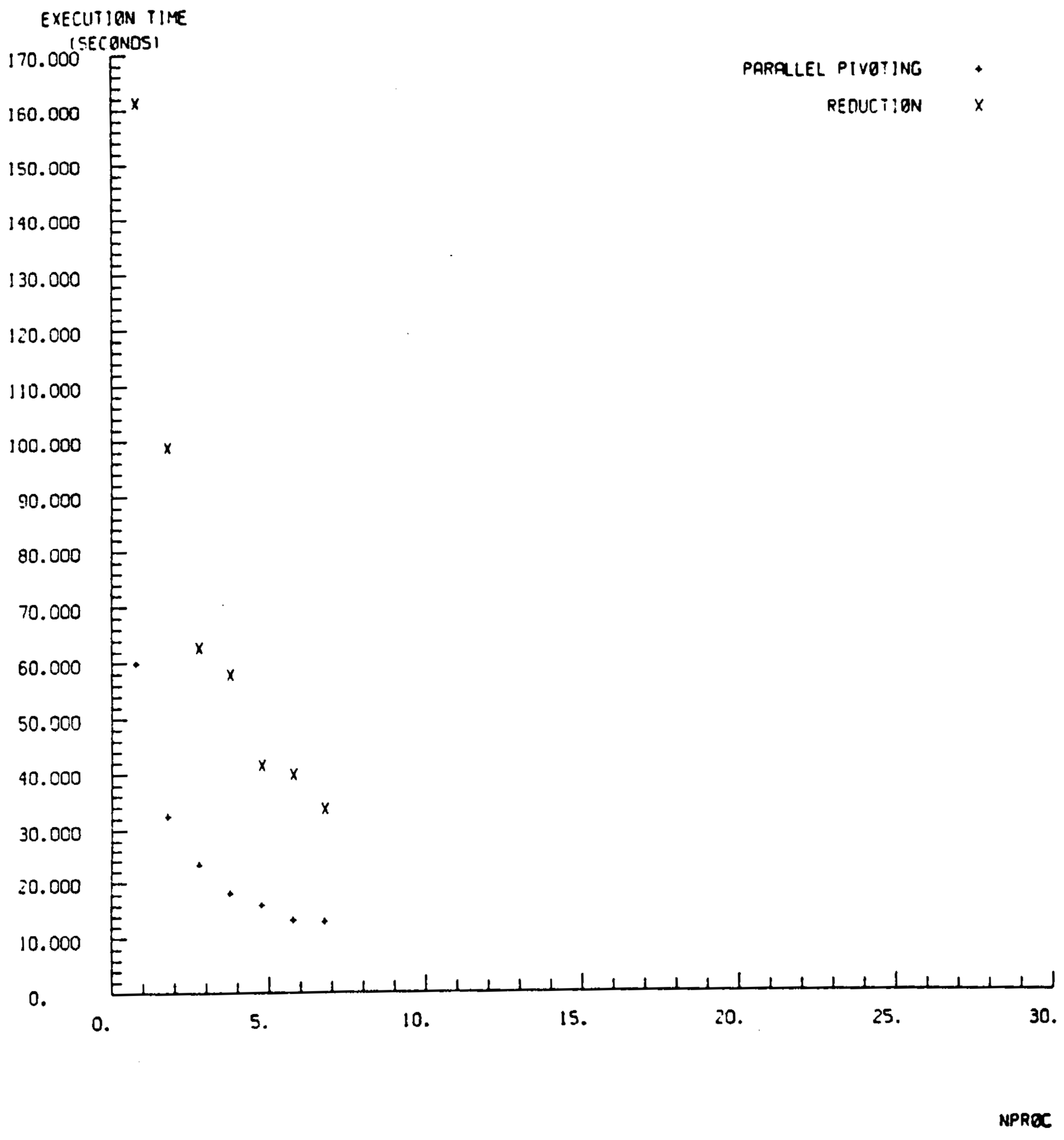


Figure 5.4 Execution Time vs. Number of Processes
Comparison of Parallel Pivoting and Reduction
ULEVEL = 4, Shrinkage = 40%, Upper Limit = 60, Threshold = 2/3
505 × 505, NZ = 5889, from SPAR Program

The 93% parallelism from the sort indicates that, for large values of N, parallel processes spend more time performing parallel operations than in the critical section.

Figure 5.4 compares the total execution time spent in routines to find the parallel pivoting candidates and execution time of the rest of the program for LU decomposition. As can be verified from this figure the time spent to find the parallel pivots is much less than the time to perform the decomposition. This difference increases as the size of the matrix becomes larger, verifying the advantage of parallel pivoting.

6. Conclusion A set of parallel algorithms for performing LU decomposition of general unsymmetric sparse matrices for shared-memory MIMD computers has been presented. The sparse LU decomposition technique employs a parallel pivoting strategy to solve the problem of having enough parallelism in sparse matrices. The main features of the heuristic algorithm can be summarized as follows:

-It can identify a good set of parallel pivots in linear time.

-It is a stepwise algorithm and can be applied to any submatrix of the original matrix. Thus it is not a preordering of the sparse matrix and is applied dynamically as the decomposition proceeds.

-Pivots can be tested for numerical stability and unsymmetric permutations can be performed if necessary.

-Trade off between parallelism and fill-in is possible under several program controlled parameters.

-The information produced by the algorithm can be stored to decompose structurally identical matrices.

We have presented the parallel reduction combined with parallel pivoting technique, control over the generation of fills and check for numerical stability, all done in parallel with work being distributed over the active processes. The program verifies that it is actually possible to do parallel pivoting in sparse matrices on multiprocessors and take advantage of the existing parallelism in the problem and in the hardware. The timing analysis of the routines indicate that every routine has been effectively parallelized. The small slope in the execution time versus number of processes of *LU Decomposition* program which represents the amount of synchronization overhead verifies the effectiveness of parallelization and machine utilization.

Acknowledgment The author would like to express her sincere appreciation to Dr. H. F. Jordan for his support and constructive comments and advice.

References

[1] M. Yannakakis "Computing the Minimum Fill-in is NP-complete," *SIAM J. Alg. DISC. Math.* 2, pp. 77-79, 1981.

[2] G. Alaghband "*Multiprocessor Sparse LU Decomposition with Controlled Fill-in*," Ph.D. thesis, Department of Electrical and Computer Engineering, University of Colorado, Boulder, May 1986.

[3] G. Alaghband, H. F. Jordan "Sparse Gaussian Elimination with Controlled Fill-in on a Shared Memory Multiprocessor," *ECSE Technical Report 86-1-5,* Electrical and Computer Engineering Department, University of Colorado, Boulder, November 1986.

[4] G. Alaghband and H. F. Jordan "Multiprocessor Sparse L/U Decomposition with Controlled fill-in," *ICASE Report No. 85-48,* NASA Langley Research Center, Hampton, Virginia 23665, 1985.

[5] D. A. Calahan, "Parallel Solution of Sparse Simultaneous Linear Equations," *Proc. 11-th Annual Allerton Conf. Circuits and System Theory,* pp. 729-735, Oct. 1973.

[6] J. W. Huang and O. Wing "Optimal Parallel Triangulation of a Sparse Matrix," *IEEE Trans. on Circuits and Systems,* vol. CAS-26, No. 9, pp. 726-732, Sept. 1976.

[7] O. Wing and J. W. Huang "A computation Model of Parallel Solution of Linear Equations," *IEEE Trans. on Computers,* vol. C-29, No. 9, pp. 632-638, July 1980.

[8] Y. F. Zhou "Optimal Parallel Triangulation of a Sparse Matrix- A Graphical Approach," *IEEE 1981 Symp. on Circuits and Systems.*

[9] K. Nakajima "A Graph Theoretical Approach to Parallel Triangulation of a Sparse Asymmetric Matrix," *Proceedings of 1984 Conf. on Information Science and Systems.*

[10] J. A. G. Jess and H. G. M. Kees "A Data Structure for Parallel LU Decomposition," *IEEE Trans. on Computers,* vol. C-31, no. 3, pp. 231-239, March 1982.

[11] F. J. Peters "Parallel Pivoting Algorithms for Sparse Symmetric Matrices," *Parallel Computing 1,* pp. 99-110, 1984.

[12] M. R. Leuze "Parallel Triangularization of Substructured Finite Element Problems," *ICASE Report no. 84-47,* Sept. 1984.

[13] I. S. Duff "Parallel Implementation of Multifrontal Scheme," Argonne National Laboratory, Mathematics and Computer Science Division, *Technical Memorandum no. 49,* March 1985.

[14] G. Alaghband and H. F. Jordan "Parallelizing a Sparse Matrix Package," *Report CSDG 83-3,* Computer System Design Group, Electrical and Computer Engineering Department, University of Colorado, Boulder, June 1983.

[15] Z. Kohavi "*Switching and Finite Automata Theory,*" Computer Science Series, Second Edition, McGraw Hill Book Company, 1978.

[16] D. Lewin "*Logical Design of Switching Circuits,*" American Elsevier Publishing, New York, 1974.

[17] H. M. Markowitz "The Elimination Form of the Inverse and its Application to Linear Programming," *Management Science, 3,* pp. 255-269, 1957.

[18] H. F. Jordan "Parallel Computation with the Force," *ICASE Report no. 85-45,* NASA Langley Research Center, Hampton, VA, October 1985.

[19] H. F. Jordan "Structuring Parallel Algorithms in an MIMD, Shared Memory Environment," *Parallel Computing,* May 1986.

[20] K. E. Batcher *Proc. AFIPS Spring joint Computer Conference, 32,* pp. 307-314, 1968.

[21] D. E. Knuth "*The Art of Computer Programming, Sorting and Searching,*" vol. 3, Addison-Wesley, 1973.

[22] J. S. Kowalik *The HEP Supercomputer and Its Applications,* Ed., MIT Press, 1985.

[23] H. F. Jordan "Interpreting Parallel Processor Performance Measurements," *Report CSDG 85-1,* Computer System Design Group, Electrical and Computer Engineering Department, University of Colorado, Boulder, November 1985.

[24] "SPAR," *NASA CR 158970-1,* Engineering Information Systems Inc., San Jose, CA, Dec. 1978.

National Aeronautics and Space Administration

Report Documentation Page

1. Report No. NASA CR-178422 ICASE Report No. 87-75	2. Government Accession No.	3. Recipient's Catalog No.

4. Title and Subtitle PARALLEL PIVOTING COMBINED WITH PARALLEL REDUCTION	5. Report Date December 1987
	6. Performing Organization Code
7. Author(s) Gita Alaghband	8. Performing Organization Report No. 87-75
	10. Work Unit No. 505-90-21-01
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	11. Contract or Grant No. NAS1-17070, NAS1-18107
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	13. Type of Report and Period Covered Contractor Report
	14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor: Richard W. Barnwell

Submitted to Parallel Computing Journal

Final Report

16. Abstract

Parallel algorithms for triangularization of large, sparse, and unsymmetric matrices are presented. The method combines the parallel reduction with a new parallel pivoting technique, control over generations of fill-ins and check for numerical stability, all done in parallel with the work being distributed over the active processes. The parallel technique uses the compatibility relation between pivots to identify parallel pivot candidates and uses the Markowitz number of pivots to minimize fill-in. This technique is not a preordering of the sparse matrix and is applied dynamically as the decomposition proceeds.

17. Key Words (Suggested by Author(s)) multiprocessor, Gaussian elimination, parallel pivoting	18. Distribution Statement 61 - Computer Programming and Software; 64 - Numerical Analysis; Unclassified - unlimited

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of pages 40	22. Price A03