PARALLEL SOLUTION OF SPARSE LINEAR SYSTEMS

Babak Nader M.S.
B.A Otterbein College , 1983

A thesis submitted to faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirement for the degree
Master of Science
in
Computer Science

May 1987

The thesis "Parallel Solution Of Sparse Linear Systems" by Babak Nader has been examined and approved by the following Examination Committee:

Cleve B. Moler
Adjunct Professor
Thesis Research Advisor

Robert Babb II
Associate Professor

Richard Kieburtz
Professor

Dan Hammerstrom
Associate Professor

ACKNOWLEDGEMENTS

The completion of this particular piece of work provides a welcome opportunity to acknowledge some long standing personal and intellectual debts.

It was Dr. Cleve B. Moler who first introduced me to the field of numerical analysis and multiprocessor computation. This work to a great extent reflects the results of his insight, criticism, and encouragement. I am deeply in debt and very grateful to Dr. Moler for his guidance and patience with this thesis and its author. This work would have never been completed with out his support and advice in the development of ideas.

In addition I would like to thank Douglas Pase for many valuable contribution and discussions which helped keep things moving.

Many thanks to Dr Robert Babb II, Dr Dan Hammerstrom and Dr Richard Kieburtz for their efforts as my thesis committee members.

To everyone, my thanks.

Babak Nader
Oregon Graduate Center
May 1987

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Parallel Solution Of Sparse Linear Systems

Babak Nader M.S.

Oregon Graduate Center, 1987

Supervising Professor: Cleve B. Moler

This paper deals with the problem of solving a system of sparse non-symmetric matrices on a distributed memory multiprocessor computer, the Intel iPSC (hypercube). The processors have substantial local memory but no global shared memory. They communicate among themselves and with a host processor through message passing. The primary interest is to design an algorithm which exploits parallelism, and which perform elimination and solution of large sparse matrices. Elimination is performed by LU-decomposition. The storage scheme is based on linked list data-structure defined for a given generated matrix. The matrix is distributed by columns in a "wrapped" fashion so that elimination in the natural order will be balanced, if the sparsity structure is equally distributed across the columns. Numerical results from experiments running on the hypercube are included along with performance analysis.

CHAPTER 1

INTRODUCTION

This paper explains the implementation of an algorithm for solving sparse systems of simultaneous linear equations on a distributed memory parallel computer, the Intel iPSC hypercube. The algorithm is designed so it can be used with any number of processors and non-symmetric matrices of any order; subject only to memory limitations. The following chapters explain the steps that are needed to be taken for implementing the program, as well as a detailed description of usage. A performance analysis of the program is also included.

1.1. Intel iPSC Concurrent Computer

The Intel Personal SuperComputer (iPSC) is one of the first commercially available parallel (or concurrent) computers. The iPSC is a true Multiple Instruction, Multiple Data (MIMD) machine. All processor nodes are identical and are connected by bidirectional links in a hypercube topology. In a 32 node hypercube, each node is directly connected to 5 nearest neighbors. For any hypercube, if d is the dimension of the cube, each processor will have d nearest neighbors, and the cube will have 2^d nodes. The average distance between any two nodes is $\frac{d}{2}$, and the maximum distance is d. Although the basic machine ($d=5$) consists of a single unit of the specifications just described, the architecture allows expansion to two or four units (64 or 128 nodes). The communication arrangement allows other topologies, such as meshes, rings, and trees, to be constructed in software by the user.

An iPSC system consists of one, two, or four basic computational units plus an Intel 286/310 computer, referred to as the *cube manager* or *host*. Each unit consists of 32 identical single-board microcomputers or *nodes*. There is a total of 16 Mbytes of memory distributed evenly among the 32 nodes. Each node has a copy of a small operating system (NX), an Intel 80286 CPU, and an Intel 80287 floating point co-processor. The 80286/80287 combination has a throughput rate of about 30 Kflops, or just under one Mflop per 32 node unit.

There are eight communication channels per node. The internode channels are implemented via seven Intel 82586 communication co-processor per node. In addition, an eighth 82586 implements a *global ethernet channel* for communication with the cube manager.

Hypercube interconnections for a 32 node machine are implemented via backplane connections. Machines consisting of two or four 32 node computational units (32 nodes) are interconnected via external cables. The collection of nodes is controlled by a system cube manager, which is an Intel 286/310 computer. This computer uses the same processors as the nodes, but has 2 Mbytes of memory, a 40 Mbyte Winchester disk, a floppy disk drive, and runs the XENIX operating system.

Processes communicate with other processes on the same or neighboring nodes by sending and receiving messages. Message passing is the only means available for internode communication and synchronization, since the iPSC has no shared memory. Message passing can be either *blocked* or *unblocked*. A *blocked send* delays execution until the message is sent. (Note that this does not mean that the message has been received). Although the use of unblocked message passing can decrease execution time, a check must be made to determine whether or not the message has been sent before modifying the contents of the message buffer. Another problem with the iPSC is that some programs can generate messages (blocked or unblocked) faster than destination nodes can receive them. There is no way of detecting whether the next message sent will cause the network hang.

All messages carry a type, which is a non-negative integer. Message types allow receiving nodes to accept only messages of a desired type. The time required for message passing depends on the number of 1 Kbyte packets (the basic unit that is sent) that must be formed and on the number of internode connections that must be traversed. If a message is sent using the blocked send routine, the process suspends until the message is sent. Similarly with a blocked receive will suspend a process until a message is received. With unblocked message passing, the program continues executing after the send (or receive) and the message is handled by the operating system according to its own priorities. In the latter case, a call to `status` can determine whether a particular message buffer is available for reuse. These message passing protocols allow users to construct correctly synchronized parallel applications and to avoid message flow problems.

CHAPTER 2

BACKGROUND

A square matrix A of order n consists of n^2 elements $a_{i,j}$. When only a few elements of $a_{i,j}$ are not zero, the matrix is *sparse*. Clearly it can, with appropriate coding, be represented by far fewer than n^2 real numbers since zero elements need not be stored. A matrix for which the majority of the elements are nonzero is a *dense* matrix. The word *density* is used to denote the proportion of nonzero elements.

Sometimes even though no element of a matrix is zero, the elements $a_{i,j}$ can be generated by a simple algorithm depending on the arguments i,j. Such a matrix is a *generated* matrix, and its element do not require n^2 real numbers of computer storage. If, on the other hand, elements of a matrix are represented as n^2 real numbers, it is a *stored* matrix. It does not matter whether some elements are zero or not since the zero will in any event be stored[FoM67].

One can easily make a trite definition of sparse matrices by defining quantitatively the ratio of nonzero to zero entries. However it is much better to say that a sparse matrix or system is one in which advantage can be taken of the percentage and/or distribution of zero elements, for example, systems with a high percentage of zero elements. There are several advantages that can be taken of sparse systems. The most evident is information storage and retrieval.

The basic problem considered in this paper is the solution of a system of simultaneous linear equations,

$$Ax=b.$$

Much of the work on sparse matrices involves symmetric, positive definite matrices A. However this paper is mainly concerned with general non-symmetric matrices A.

As Alan George *et al[GeL81].* point out, the numerical methods for solving such systems falls into two general classes, *iterative* and *direct.* A typical iterative method involves the initial selection of an approximation $x^{(1)}$ to the solution x, and the determination of a sequence $x^{(2)},x^{(3)},\cdots$ such that $\lim_{i\to\infty} x^{(i)}=x$. Usually the calculation of $x^{(i+1)}$ involves only A,b, and one or two of the previous iterates. In theory, when we use an iterative method we must perform an infinite number of arithmetic operations in order to obtain x, but in practice we stop the iteration when we believe our current approximation is acceptably close to x. On the other hand, in the absence of rounding errors, a direct method provides the solution after a fixed number of arithmetic operations have been performed.

When using the direct methods for solving sparse linear equations it is important to design the algorithm to preserve as much as possible the system's initial sparsity. As Ian Duff[DUF77] explains there are several direct methods:

i) LU decomposition (elimination form of inverse)
ii) Row Gauss elimination
iii) Product form of the inverse (Gauss-Jordan)
iv) Compact elimination (Crout reduction)

i) LU decomposition :

The LU decomposition of the system is

$$A = LU$$

where U is the upper triangular matrix and L is the lower triangular matrix of the A matrix. L can be considered as the sparse factors of the normally dense L^{-1} . The system

$$Ax = b$$

can be solved by using forward substitution

$$Ly = b$$

followed by back substitution

$$Ux = y$$

The k^{th} column of L is obtained from the first k columns of A. The reduced matrix $A^{(k)}$ is shown in Fig.1, where the L part of A has been reduced to zero (and factors of L are stored), the first k-1 rows of U have already been found, and $A^{(k)}$ has been modified by each previous step of elimination. The remainder of A is then modified in step k according to the equation.

$$a_{ij} = a_{ij} - a_{ik} \cdot \frac{a_{kj}}{a_{kk}}$$

and the algorithm proceeds to the next step.

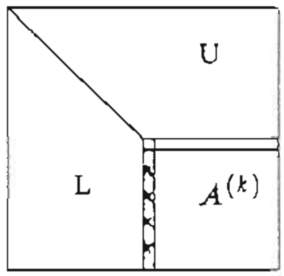


Fig.1 Partial LU decomposition

When performing elimination on sparse systems, a primary objective is to choose the pivot sequence to preserve sparsity, so that the number of arithmetic operations and the number of nonzero's added during the elimination is kept to minimum. But it is also important that the pivoting preserves numerical stability. Pivoting for sparsity does not necessarily depend on numerical values. An extreme example of the effects of pivoting on sparsity is given by

```
X X X X X
X X
X   X
X     X
X       X
```

If the pivots are chosen from the diagonal so that the element (1,1) is chosen last, there is no change in sparsity pattern and only (n-1) divides and (n-1) multiply-adds are required to effect the decomposition and no additional storage is required. However, should element (1,1) be chosen first, then fill-in is total and $\frac{n^3}{3}+O(n^2)$ multiply-adds and $\frac{n^2}{2}+O(n)$ divides are required and n^2 storage locations are needed. Pivoting for numerical stability is also important, since in equation

$$a_{ij}=a_{ij}-a_{ik}\cdot\frac{a_{kj}}{a_{kk}}$$

division by a small pivot a_{kk} will causes excessive magnification of round-off error.

Most techniques for pivot selection fall into one of two categories. In *a priori* methods, the column (or rows) are first ordered and then, at each stage of the elimination, the pivot is chosen from within the first column of the reduced submatrix (matrix $A^{(k)}$). In *local* strategies, the pivot is selected from among all the nonzero's in the reduced matrix using the knowledge of its actual updated structure at the stage of the elimination.

A priori ordering strategies are useful when the matrix is held on backing store and can be accessed only a column (row) at a time. They are however, not nearly as good as local methods at preserving sparsity or reducing the operation count. Common selection criteria are to order the columns in increasing order column count, or to order them in increasing total number of nonzero entry in the given column.

ii) Row Gauss elimination

In this method of elimination, at the k^{th} stage of row k, A is transformed to the appropriate parts of L and U by subtracting multiple of rows *1,2...,* *k-1* from it in turn. This routine can handle a sparse data structure but local ordering techniques are not possible. The *a priori* ordering is possible. Fig.2 illustrates the scheme of Row Gauss elimination.

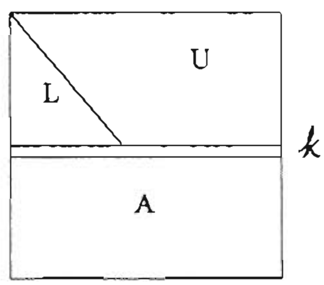


Fig.2 Row Gauss elimination

full matrices and is also used for sparse matrices. This algorithm again computes L and U from the identity

$$A = LU$$

in the order shown in Fig.4. Again the same problem is encountered as in the row-Gauss elimination. Local ordering is not possible, but priori ordering and partial pivoting are.

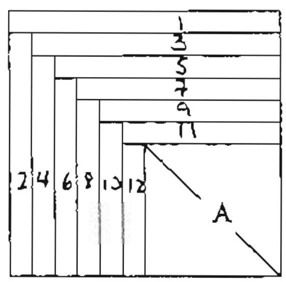


Fig.4 The Crout reduction

2.1. Other algorithms:

A) According to Dongarra *et al.* [DGK84] the basic algorithm for Gaussian elimination can be described as follows

Generic Gaussian elimination algorithm

for ______

for ______

for ______

$$a_{ij}=a_{ij}-\frac{a_{ik}\cdot a_{kj}}{a_{kk}}$$

end

end

end.

The indices and loop information are intentionally left blank since there are six different forms of Gaussian elimination possible depending on the order the indices i,j,k are placed in the above algorithm. For example, the form ijk and jik are variants of Crout reduction algorithm discussed before. The Crout reduction algorithm can be characterized by the use of inner products to accomplish the decomposition. In appendix B there are four Gaussian elimination algorithms which are column variants of the generic algorithm discussed above. Since Fortran is a column oriented language, the algorithm performance in a column variant algorithm is better than the corresponding row variant.

B) Research has been done at Oak Ridge National laboratory on systems of positive definite sparse matrices

$$Ax=b$$

on local-memory and shared memory multiprocessor systems. The basic algorithm used is parallel sparse Choloski decomposition

$$A=LL^T$$

where L is the lower triangular factor of matrix A and L^T is the transpose of L.

Pivoting in a positive definite system is done only for sparsity. For numerical stability, any diagonal pivots are acceptable. Therefore symmetric Gaussian elimination (Cholesky's method) applied to a symmetric positive definite matrix does not require interchangings (pivoting) to maintain numerical stability. Since PAP^T is also symmetric and positive definite for any permutation matrix P, this means we can chose to reorder A symmetrically *i)* without regard to numerical stability and *ii)* before the actual numerical factorization begins.

This has an important practical implication, since the ordering can be determined before the factorization begins, the locations of fill-in suffered during the factorization can also be determined. Thus the data structure used to store L can be constructed before the actual numerical factorization, and spaces for fill-in components can be reserved. The computation then proceeds with the storage structure remaining *static* (unaltered).

In the local memory case (hypercube) [GHL86] the ordering that is most suited is one that utilizes the parallelism, and allows distribution of the computation across the processors in a way so that there is not an inordinate amount of communication. The formulation use is to store the lower matrix L is an elimination tree for sparse Cholesky factors[DuR83][Liu86]. Therefore consider the structure of Cholesky factor L. For each column $j \leq n$, if column j has off-diagonal non-zeros, define by

$$\Upsilon[j]=\min\{i \mid l_{ij} \neq 0, i>j\}$$

that is, $\Upsilon[j]$ is the row subscript of the first off-diagonal in column j of L. If column j has no off-diagonal non-zeros, we set $\Upsilon[j]=j$, (Hence $\Upsilon[n]=n$.) The elimination tree has n nodes, labeled from 1 to n. For each j, then node $\Upsilon[j]$ is the parent of the node j in the elimination tree, and node j is one of possibly several different child nodes of the node $\Upsilon[j]$. In order to recognize the parallelism identified by the elimination tree, consider the 3 by 3 grid example shown in fig.5a from [GHL86] which can be represented by triangular matrices in fig.5b from [GHL86], and their elimination tree is represented in fig.6 from [GHL86]. The elimination tree on the left is the best, since it yields less fill-in and low operation count, and leads to better parallel load balancing. Therefore task of column 1,2,3,4 can start in parallel. Moreover, when 1,2,3,4 complete their execution then column 5,6 can start execution in parallel independently and so on.

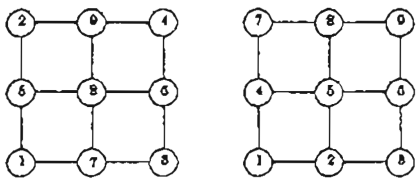


Fig 5a Two orderings of a 3 by 3 grid

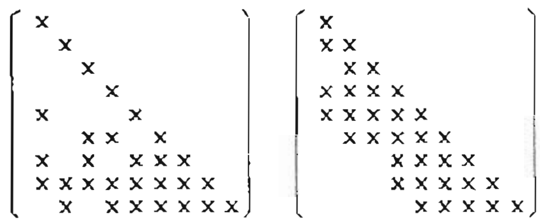


Fig 5b Structure of the Cholesky factors for the orderings of Fig 5a

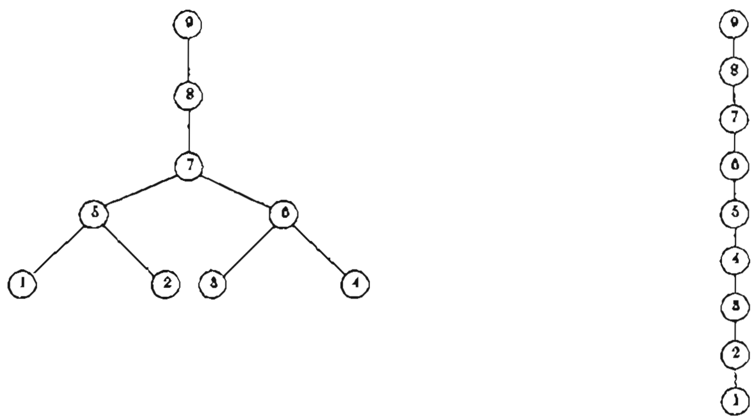


Fig. 6 The elimination trees associated with the matrices in Fig. 5b

In the shared memory case [GHL87] the formulation used to store the non-zero elements of the matrix is a linked list data-structure. Their formulation maintains a set of non-overlapping linked list, one for each column of matrix. Since they are non-overlapping, an n-vector link will be enough to implement them. When the term $link^m[j]$ is used it denotes the m-th element in the link list structure for column j; for example, the third element of column j can be denoted by $link^3[j]=link[link[link[j]]]$. The lists are assumed to be null-terminated, so that the j-th list is given by:

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

Where for some r, $link^{r+1}[j]=0$. Also an array *next(j,k)* is defined to be the row subscript of the next nonzero in column k of L immediately beneath L_{jk}. Hence *next(j,k)* will depend on both j and k. For more illustration see fig.7.

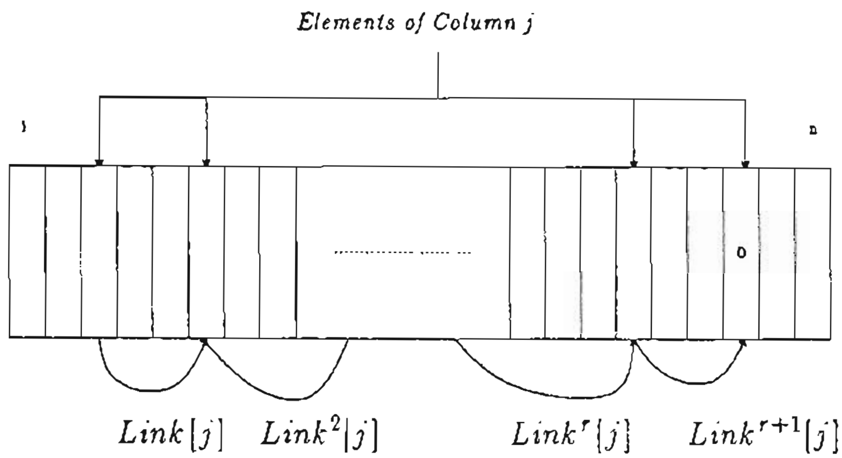


Fig. 7 ORNL Link list data-structure for columns of sparse matrix

Since a shared memory is used, the computing regime they use adopts a notion of a *pool of tasks* whose parallel execution is controlled by a self scheduling discipline[Jor84]. The tasks are those computations associated with columns of the coefficient matrix and hence have a well defined order associated with them. Since effective static load balancing among the processors requires that the distribution of work to be reasonably uniform, the self scheduling can be regarded as a mechanism for implementing dynamic load balancing; p processors are initiated to perform T tasks where p is less than or equal T. When a given processor completes a task, it checks to see if any unsigned tasks remain, if so it is assigned to the next one. So a processor with smaller task will be freed sooner than a processor with larger task. In this way, processors tend to be kept busy even if the tasks vary in their computational requirements. For more information and proof about positive definite systems see references [GeL81][GHL87].

C) For non-symmetric matrices, a sequential algorithm developed by Cleve Moler uses a sparse compact method to solve the system of sparse linear equation

$$Ax=b.$$

The non-zero elements are stored in n linked lists, rather than in a two dimensional array as in a dense matrix case.

A sparse *vector* is a linked list of triplets corresponding to the nonzero elements of a vector. Each triplet contains *val,* a floating point value; *row,* the index of that value; and *next,* a pointer to the next triplet. The last nonzero of each vector or column of a matrix is followed by an additional triplet called the "footer", containing a zero value, a row index equal to the largest machine integer, and a pointer to the footer itself. A sparse *matrix* is an array of pointers to sparse vectors, one for each of the column in the matrix A as shown in Fig.8.

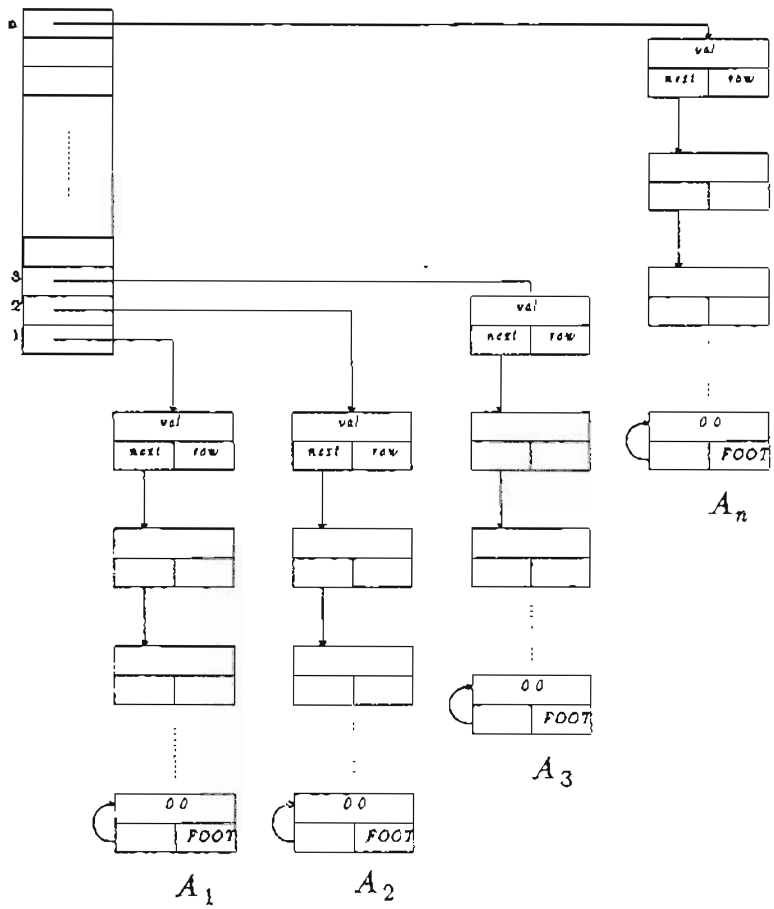


Fig. 8 Link list data-structure of sparse matrix

D) The parallel extension of Moler's algorithm uses the LU decomposition to solve the system of sparse linear equations on a distributed memory multiprocessor system, the Intel iPSC hypercube. The algorithm uses the same data structure as in Fig.8 but with the variation that the columns of the matrix are distributed across p processors. In this scheme column j of the matrix is generated and stored on the processor with identification number $(j-1) \bmod p$, as shown in Fig.9. The remainder of this paper will describe this algorithm in greater detail.

COLUMN j IN NODE $(j\text{-}1)$ mod p

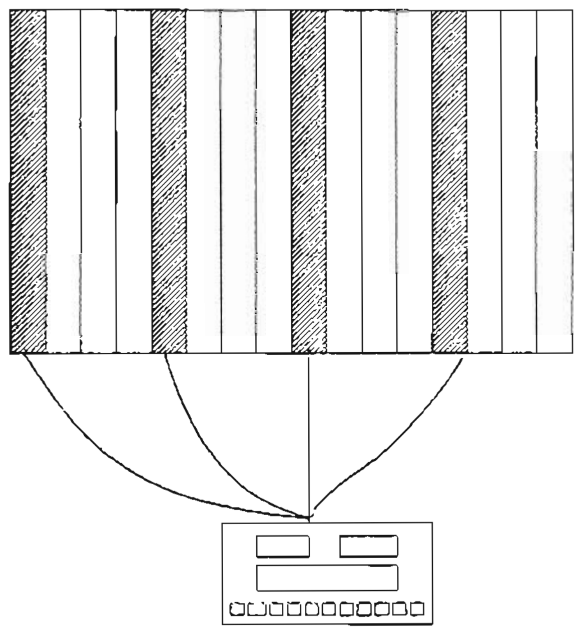


Fig.9 Distribution of columns of sparse matrix

CHAPTER 3

USAGE

Three double precision routines for sparse matrices which are included in the appendix A, PGSFA, PGSSL, and PGSMUL. The subroutine names follow the LINPACK [DMB79] naming conventions; PGS stands for parallel general sparse, FA for Factor and SL for solve. PGSFA is called once to factor a particular sparse matrix and then PGSSL is called to perform forward and backward substitution. The calling sequence for PGSFA is

CALL PGSFA(A,n,nm,p,cid,id,prat,krow,gkrow,pvt,L,U,D)

This computes LU decomposition of a sparse matrix.

The input arguments are:

A	integer (n) pointers to the columns of the sparse matrix
n	integer order of the sparse matrix
nm	integer number of column assigned to node.
p	integer number of processors.
cid	integer channel id
id	integer identification of node (mynode() returns)
prat	double precision ratio : minimum acceptable / maximum pivot
krow	integer (n) count of the number of nonzero elements in the rows

of the sparse matrix

The output arguments are:

pvt integer (n)
record of the row exchanges incurred in the LU decomposition

gkrow integer (n)
global count of the number of nonzero elements in the rows of the sparse matrix across p processors

L integer (n)
pointers to the columns of the sparse lower triangular factor (unit diagonal omitted)

U integer (n)
pointers to the columns of the sparse upper triangular factor (diagonal omitted)

D double precision (n)
diagonal elements of the upper triangular factor

Note that upon return A points at the columns of the upper triangular factor (diagonal omitted). The triplets of the original sparse matrix are overwritten by the those of the upper and lower triangular factors. Therefore A is same as U on output from PGSFA

PGSFA is usually called first to factor the sparse matrix. The actual factorization is done in the form of

$$A = L(D+U)$$

Which L is the lower triangle of the sparse matrix A, and U is the upper triangle of the sparse matrix excluding the diagonals which are kept in vector D.

PGSSL uses the L(D+U) factorization of the matrix A to solve the linear system of equation

$$Ax=b$$

The calling sequence is

CALL PGSSL(cid,L,U,D,n,nm,p,pvt,b,id)

which finds the solution of a system of linear equations whose sparse matrix is in the L(D+U) form provided by subroutine pgsfa.

The input arguments are the output arguments of PGSFA together with the right hand side b :

L	integer (n) pointers to the columns of the lower triangular factor (unit diagonal omitted), as returned by PGSFA
U	integer (n) pointers to the columns of the upper triangular factor (diagonal omitted), as returned by PGSFA
D	double precision (n) diagonal elements of the upper triangular factor as returned by pgsfa
n	integer order of the system
nm	integer number of the column on each node.
p	integer number of the processors
pvt	integer (n) record of exchanges returned by PGSFA
b	double precision (n) right-hand side of the system

The output arguments are:

b	double precision (n) solution of the system

PGSMUL is called to multiply a vector by a sparse matrix. This routine is used to compute the right hand side of the equation

$$Ax=b$$

The b is then used by PGSSL to solve for x in the above equation using L,U and D. The calling sequence is

CALL PGSMUL (A,n,m,p,cid,id,x,y,t)

This routine computes y = A*x.

The input arguments are:

A	integer distributed over p nodes
n	integer order of sparse matrix
m	integer number of column assign to node
p	integer number of processors
cid	integer channel identification
id	integer node id (returned by mynode())
x	double precision held on node 0
t	double precision dummy array for work

The output arguments are:

y	double precision result on node 0
x	double precision destroyed

There are two routines used for generation of a sparse matrix, INIT, and INSERT. INIT is called to initialize a vector. The calling sequence is

CALL INIT(v)

This creates a new sparse vector which is initially empty (consisting of a footer).

The input arguments are:

v	integer pointer to the first triplet of the new sparse vector

INSERT is called to insert new element in a sparse vector. The calling sequence is

CALL INSERT(p,alfa,i)

Inserts a new component in a sparse vector

The input arguments are:

p	integer pointer to the successor of the triplet to be inserted
alfa	double precision value of the component to be inserted
i	integer index of the component to be inserted

3.1. Other routines called

There are several routines called by the three routines PGSFA, PGSSL, PGSMUL, which are included in the appendix A. PIVIDX is the function called by PGSFA for each column of sparse matrix A to find the pivot index of that sparse vector. The calling sequence for PIVIDX is

CALL PIVIDX(aa,defalt,prat,krow)

This returns the index of the minimum component of a vector of integers subject to the corresponding component of a sparse vector being no less than a fraction of its maximum component (in magnitude).

The input argument are:

aa	integer pointer identifying the sparse vector
defalt	integer index to be returned if the sparse vector is empty
prat	double precision acceptable fraction of the maximum component of the sparse vector
krow	integer (*) vector of integers

Also PGSFA calls a routine called SCOLL to perform scaling of each of the columns of the lower triangular matrix L by the reciprocal of the pivot index. The calling sequence for SCOLL is

CALL SCOLL(xx,s,krow,cid)

This performs scaling of the sub-diagonal elements of a column of a sparse matrix in Gaussian elimination to form the corresponding column of the lower triangular factor, and update the record of nonzero elements in the rows (krow) by subtracting one from the krow of the row index of that column (krow(row(x))-1).

The input arguments are:

xx	integer pointer identifying the first sub-diagonal element of the column to be scaled
s	double precision scaling divisor (pivot of the elimination)
krow	integer (*) record of the numbers of nonzero elements in the rows of the matrix

The output arguments are:

krow updated record of the row counts of nonzero elements

Subroutine SWAP is used whenever it is necessary to interchange two elements in a sparse vector. The calling sequence for SWAP is

CALL SWAP(xx,kk,mm)

This exchanges the k-th and m-th components of a sparse vector

The input arguments are:

xx	integer pointer to the vector where the exchange takes place
kk	integer index of one of the components to be exchanged
mm	integer index of the other components to be exchanged

It is possible that either index, *kk* or *mm*, may refer to a zero component, for which there is no triplet in the linked list. The list is always ordered so that row indices increase as the list is traversed. So if exactly one of the indices , *kk* or *mm*, refers to a zero component, it is necessary to reorder the list. In this case, a routine called CANDP is used to perform the cut and paste operation. The routine CANDP moves one triplet to precede another. The calling sequence for CANDP is

CALL CANDP(p,q,i)

The input arguments are:

p	integer pointer identifying the triplet to be moved
q	integer

	pointer identifying the triplet chosen to become the successor of the moved triplet
i	integer component index to be assigned to the moved triplet

The effect of a call to candp is equivalent to the sequence

```
call insert(q,val(p),i)
call delete(p)
```

but without creating a "dead" element.

When both indices refer to nonzero components, it is necessary to exchange the contents of the triplets. SWAP calls routine SWREAL. The calling sequence for SWREAL is

```
CALL SWREAL(alfa,beta)
```

This swaps two double precision variables

The input arguments are:

alfa	double precision variable to be exchanged with beta
beta	double precision variable to be exchanged with alfa

The output arguments are:

alfa	the value entered as beta
beta	the value entered as alfa

There are two routine used for packaging the sparse vector for broadcasting to the other nodes PACK and DPACK. The PACK routine is used by PGSFA to pack the sparse scaled vector L and the pivot index of present iteration. Starting from the second element into the packed sparse vector y, the row index and the value of each triplet in the sparse vector L is stored in pair sequentially. The first element is saved to store the pivot index. The sequence of call for PACK is

CALL PACK(cid,y,xx,m,s)

The input arguments are:

xx	integer pointer identifying the sparse vector
m	integer k-th pivot

The output arguments are:

y	full vector (packed sparse vector)
s	integer determined size of packed sparse vector including FOOT, returns minimum size of 2.

The routine DPACK is used by PGSSL2 for packing the sparse vectors L, U and is similar to PACK except instead of setting the first element of the buffer y to the pivot index, it is set to the double precision value of the new b. The calling sequence is

CALL DPACK(cid,y,xx,m,s)

which fills y with column of pointer xx and the first
element of y with the value of b(k) and the last element
with index and value of FOOT.

The input arguments are:

xx	integer pointer identifying the sparse vector
m	double precision computed value of b(k)

The output arguments are:

y	full vector (packed sparse vector)
s	integer determined size of packed sparse vector including FOOT, returns minimum size of 2.

There are three routines used for elimination and multiplication of the sparse vectors YAXPY, FAXPY and BAXPY. All three of these routine perform the operation

$$y = y + \alpha \cdot x$$

But each routine performs this operation with different set of data-structures passed as their arguments, see table 1. and Fig.10 for illustration.

Table 1.
Data-structure of routines

Routine	Vector y	Vector x
YAXPY	sparse	packed sparse
FAXPY	full	sparse
BAXPY	full	packed sparse

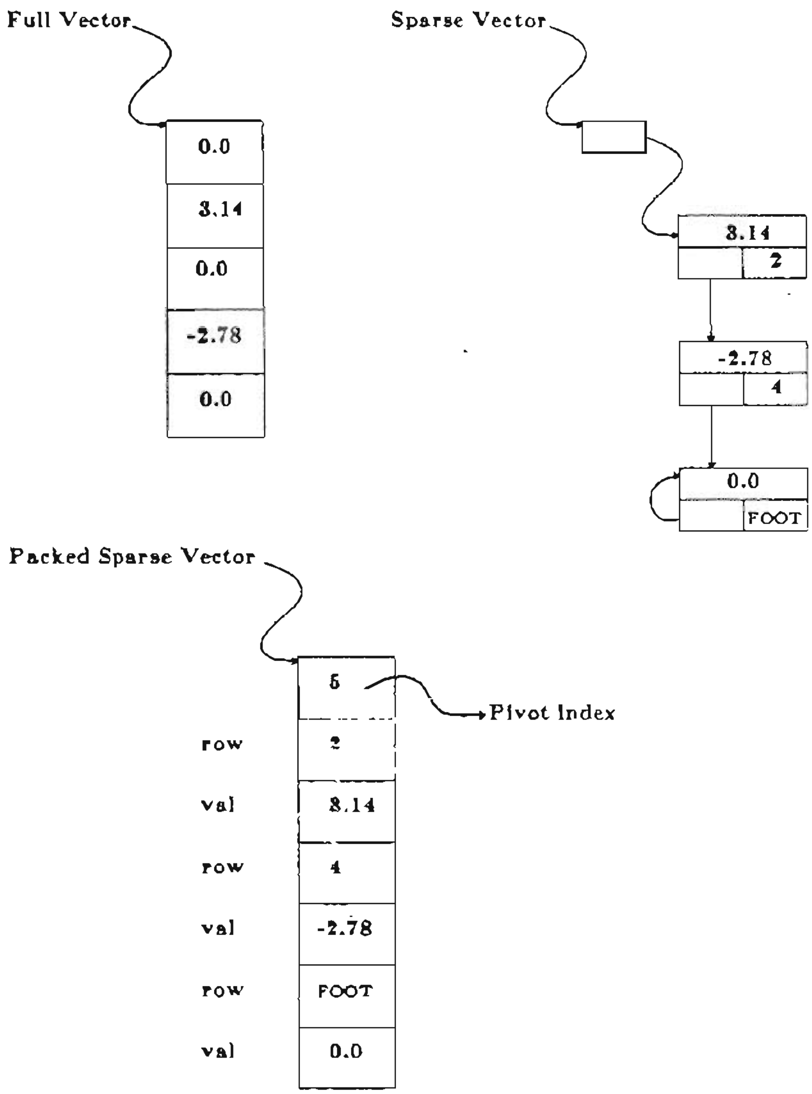


Fig. 10 Different data-structures for vector x, and y

The routine YAXPY is called by PGSFA to perform Gaussian elimination on each sparse vector below the pivot row, yy, by multiplying the packed sparse vector x containing the scaled values of lower matrix, L produced by routine SCOLL to the sparse vector yy, which results in producing the new vectors of the upper matrix U. Also *YAXPY* performs insert and delete operation if the elimination result in a zero element or a newly created element for the vector. The calling sequence for YAXPY is

CALL YAXPY(alfa,x,yy,krow)

This performs modification of a column of a sparse matrix in the Gaussian elimination of one variable

The input arguments are:

alfa	double precision element of the column with same row index as the pivot
x	packed sparse vector containing the value and row indices of lower matrices
yy	integer pointer to the first element of the column of the matrix below the pivot row
krow	integer (*) record of the numbers of nonzero elements in the rows of the sparse matrix

The output arguments are:

krow	updated record of the number of nontrivial elements in
yy	integer pointer to the new changed element of the column of the matrix below the pivot row

matrix

On the other hand the routine FAXPY is called upon by PGSMUL to perform multiplication operation of a sparse vector x to a full vector y using alfa as coefficient of the vector operation. The calling sequence for FAXPY is

CALL FAXPY(alfa,xx,y)

Which performs addition of a multiple of a sparse vector to a full vector

y <- y + alfa * sparse (xx)

The input arguments are:

alfa	double precision multiplier of the sparse vector
xx	integer pointer identifying the sparse vector
y	double precision (*) full vector

The output arguments are:

y	modified full vector

The routine BAXPY is called by routine PGSSL2 the second version of PGSSL to perform the vector operation of a packed sparse vector x to a full vector y. These packed sparse vector are of length of each sparse vectors L or U depending on the operation being performed by PGSSL2 when called. The calling sequence for BAXPY is

CALL BAXPY(alfa,x,y)

This performs addition of a multiple of a full vector to a packed sparse vector

y <- y + alfa * packed-sparse(x)

The input arguments are:

alfa	double precision multiplier of the packed sparse vector
x	double precision (*) packed sparse vector
y	double precision (*) full vector

The output arguments are:

y   modified full vector

3.2. Sequence of call

Since the input/output operations are performed by the *cube manager*, the input parameters $p, n, dens, prat, seed$ corresponding to number of processors used, the order of matrix, the off-diagonal density, the maximum allowable pivot ratio, and the initial seed for the random number generator are collected into a buffer by the host program and sent to nodes by the message passing routine *sendmsg* and result received from the node program by the routine *recvmsg*. Once the buffer is received by the root node of the spanning tree a copy of the buffer is sent to the other nodes for execution.

The node program in appendix A describes the sequence of calls used in the main node program, which is executed on each node. First the matrix of order n is generated and then factored to the form of

$$A = L(D+U)$$

and then solved by forward substitution and backward substitution.

Also the main node program when generating the sparse matrix A calls a routine called FSUM to compute the sum of absolute value of the sparse matrix A for residual calculation. The calling sequence for FSUM is

CALL FSUM (xx)

This sum up the absolute values of a sparse vector

$$t = t + abs(sparse (xx))$$

The input argument is ;

xx integer
pointer identifying the sparse vector

The output is :

fsum Sum of absolute values of sparse vector

3.3. Broadcast routines

The communication between nodes is done with broadcast routines, GSENDW, GRECVW and global operation such as addition, multiplication is performed by using the GOP routine. In a technical report by Intel a detailed information about the above communication utilities is given [MoS86].

3.4. Basic Linear Algebra Subroutines

There are two Basic Linear Algebra Subroutine used, DASUM, and DCOPY for double precision addition of full vector elements and copying elements of a vector to another vector. For detailed information about the above routines see reference[DMB79][Mol86].

CHAPTER 4

Programming detail

4.1. PGSFA

In the routine PGSFA the principal loop involves k, the index of the pivot row and column. The subroutine pividx is used to find L, the row index of the minimum component of a vector of integers subject to the corresponding component of a sparse vector being not less than a fraction of its maximum component in magnitude, below the diagonal in the k^{th} column. Pividx uses the global row count *gkrow* of the row counts *krow* across the p processors which is collected by a global operation across the spanning tree to find the pivot index. Once the pivot index is found, if the pivot index is not equal to the value of the k^{th} step of the iteration the value of the triplet contained at the k^{th} row index is swapped by the value of the row index of the pivot index. And if there is no triplet at either indices a triplet is created and deleted in accordance to the index(the routine *swap*). Since the value of the pivot index is changed, all the other elements of that row which are across the node in different columns should be swapped as well so that the pivoting is complete. Finally the krow count should be swapped as well so it will contain the correct row count. At the

R^{th} node the value of the pivot index is stored in a full vector D, and the rest of the sparse vector L is scaled by the value stored at that index of D(the routine *scoll)*. Since the rest of the nodes containing the rest of the column at this time are waiting for this information to perform elimination on there column to produce the upper triangular sparse matrix U, the scaled lower sparse vector $\frac{a_{kj}}{a_{kk}}$ is packed in a buffer array BUF and broadcasted across to the rest of the nodes. The rest of the nodes including the R^{th} node (root node) then perform the elimination in accordance to the equation

$$a_{ij}=a_{ij}-a_{ik}\frac{a_{kj}}{a_{kk}}$$

Which the result is the upper triangular matrix U. The routine used for performing the vector multiplication of the above equation and transformation of the vector of matrix A is *YAXPY*. Once all the iterations of index k is done the LU-decomposition is complete and the sparse vector of A is decomposed into two other sparse vectors L *and* U and full vector$D$.

4.2. PGSSL

For solving the system of linear equation there are two routines implemented since neither of the routines utilize the parallelism of the system, evaluation of performance difference is left for further research, and implementation of a parallel solve routine is left for further research as well.

4.2.1. PGSSL1 (first version of solve)

The routine *PGSSL1* uses the two sparse vectors L , U and a full vector D to perform forward substitution and backward substitution. Before any substitution applied to b vector the row index of b is checked against the pivot index which is stored in an array *pvt* by PGSFA and if they are not equal the row of index of b at present iteration is swapped with the index of the pivot column. Then forward substitution is first applied to the first column or columns of the matrix which resides at node one, in accordance to the equation

$$Ly=b.$$

Then the new value of b after application is sent to the successor node which is at the stage of receive wait for a message from the predecessor node. When the message received by the successor the same transformation is applied to the column it is assigned to and send b to its successor.

Once forward substitution is perform all through nodes with their columns then backward substitution is applied, but first the b is scaled by the pivot value which is stored in the diagonal since the transformation of A was to LDU rather than to

$$A=LU.$$

Then the backward substitution is applied in accordance to equation

$$Ux = y$$

Which y contains the vector with the estimated solution of the system.

4.2.2. PGSSL2(second version of solve)

The routine *PGSSL2* performs the same operation of substitutions as the *PGSSL1* with a difference that each node waits to receive the packed vector of the sparse vector L through the iterations of k and then applies the transformation to the its copy of the vector b. At the end of k^{th} iteration all the copies of b have gone through same double transformation (forward and backward) and contain same values.

4.3. PGSMUL

The routine *PGSMUL* perform multiplication of a sparse vector to a full vector. Since the vector x is initially stored at node zero therefore a copy of it is sent to all nodes. Then faxpy (sparse daxpy) operation is applied to the vector y with coefficient x. Since sparse vectors of A are spread across p nodes, then each of the row values of the y vector are spread across the p processes. Therefore global operation is applied and a copy of the value of all indices of the vector y is sent to the node 0 which is the root node in spanning tree.

.4. PIVIDX

The routine *PIVIDX* is called by *PGSFA* with each column of sparse matrix A. Each sparse vector is traversed through using *FOOT* as the indicator of end of vector, and the largest value (*val*) plus the row index of that value (*row*) are stored in a temporary variables *t and m*. If the vector is empty a default index which is the index of the k^{th} pivot is returned as the pivot index. On the other hand if the pivot ratio *prat* is 1.0d0 then the index of the largest value in that vector is returned as the pivot index. Otherwise we check to find the index of the triplet with the largest component value and the most row count *krow*.

4.5. SCOLL

The routine *SCOLL* is called by *PGSFA* with each column of triangular sparse matrix L. Each sparse vector is scaled (divided) by the pivot value. The row counts *krow* of each row is decremented since a subset of the matrix is now left for further elimination.

4.6. SWAP

The routine *SWAP* is called by *PGSFA* if the pivot index returned by the *PIVIDX* is not the index of the k^{th} iteration. It is called with a pointer to the first element of the sparse vector and the index of pivot and k^{th} iteration index. First row index of triplet bigger than or equal the minimum of indexes is found and then the index of the triplet bigger or equal than the maximum of the indexes is found. Once the two index are found there are four cases to consider, i) both components are zero which nothing is done, ii) one of the components of the indexes is zero which *candp* (cut and paste) routine is used for exchange, iii) both components are non-trivial which *swreal*(swap real value) is used to just swap the value in the triplets.

4.7. CANDP

The routine *CANDP* moves one triplet to proceed another. When called index pointer of the two triplets and an index that is to be assigned to the triplet that is being moved to are passed as arguments. The cut and paste operation is performed with out deleting a triplet, rather it performs the exchange by first saving the contents of the triplet in a temporary variable and then uses the freed triplet to insert the other triplet values into it.

4.8. SWREAL

The routine *SWREAL* swap to double precision variables. When called the two variable to be swapped are passed as argument and the value are swapped and returned with new values in them.

4.9. YAXPY

The routine *YAXPY* performs the Gaussian elimination of one column of a sparse matrix A according to the equation

$$a_{ij}=a_{ij}-a_{ik}\frac{a_{kj}}{a_{kk}},i=k+1,\ldots,n$$

The above equation's coefficient $\frac{a_{kj}}{a_{kk}}$ is passed to the routine in a variable *alfa* and the values a_{ik} and their row indexes are passed on to the routine in a packed sparse matrix x. Also a pointer to the sparse vector *yy* containing the elements of sparse vector that contain the elements of a_{ij} is passed as well. Therefore inorder to apply the above equation we must first traverse the sparse vector against the each of the row indexes in the packed sparse matrix x until we reach to the triplet with index less than the index in x. If row index of triplet is less than the row index in x we that means a fill in operation must be performed containing the multiple of the *alfa* and the value stored in x. If the components contain coincident indices then the multiple of *alfa* and value are subtracted from the coincident triplet value *val*.

$$val(y)=val(y)+alfa*x(k)$$

The row index of y is equal to the row index stored at *x(k-1)* of the packed sparse vector x.

4.10. FAXPY

The routine *FAXPY* does vector operation by performing addition of a multiple of a sparse vector to a full vector

$$y := y + alfa * sparse(x)$$

Where in the routine, y is the full vector and *alfa* the multiplier and x is index of a pointer to the sparse matrix x. Therefore the sparse vector is traversed until the end of the list and the above vector operation is applied to the coincident indices of x and y.

4.11. BAXPY

The routine *BAXPY* does vector operation by performing addition of a multiple of a packed sparse vector to a full vector

$$y := y + alfa * x$$

Where in the routine, y is the full vector and *alfa* the multiplier and x is values stored in the sparse full vector x, which contain row index of packed sparse vector as well as the value. Therefore the sparse full vector is traversed using the row indexes stored until the end of the vector. Then the above vector operation is applied to the coincident indices of x and y.

4.12. FSUM

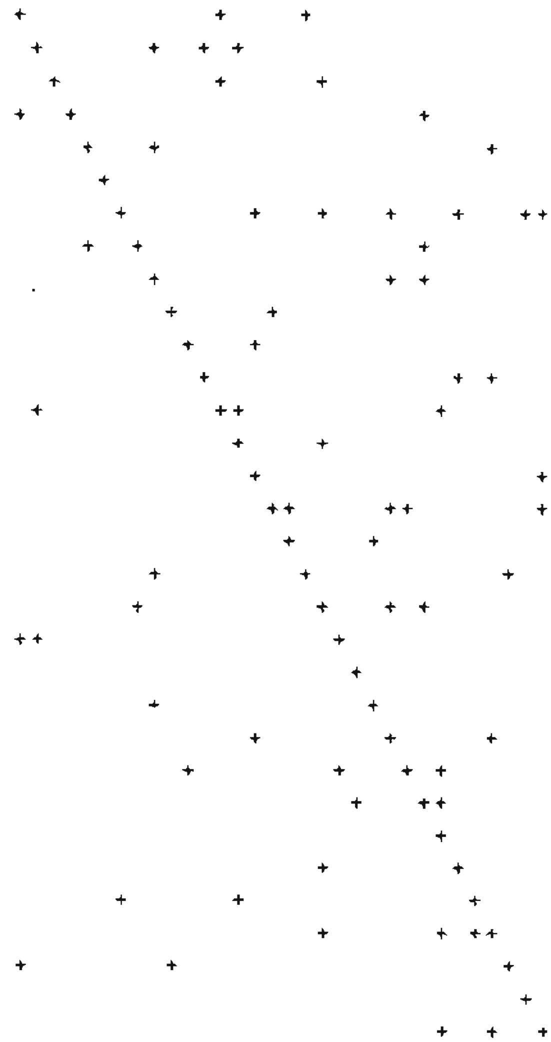
The routine *FSUM* is called by the main node program to calculate the sum of the absolute values of the sparse matrix A, by traversing each vector of the matrix and summing up the absolute values of *val* in the triplets of the vectors. The routine is called m times by each node, which is the total number of columns assign to that node. Then by global operation the value of all the sums are collected from all the participating nodes in the spanning tree.

CHAPTER 5

Performance Analysis

Numerical experiments were performed on the Intel iPSC hypercube of dimension $d=5$ with local memory and message passing routines for broadcasting and communication.

The parallel algorithm described in this paper is in Fortran and compiled by Ryan-McFarland compiler. The program has been tested on p processors, where $1 \leq p \leq 32$. The test problems used for these experiments are random sparse matrices of different density values. All the diagonal elements are nonzero and a fraction of off-diagonal elements are nonzero, the fraction is the experimental parameter *dens*. The locations of the off-diagonal elements are determined randomly. The data structure of the program consist of triplets *row*,*next*,*val*. So each nonzero element of the matrix is stored on a triplet which requires a total of twelve bytes of memory (2 bytes each for *row*,*next* and 8 bytes for the double precision value *val*). The figure on the following page is an example of a sparse matrix with off-diagonal density of 0.005 (one-half of a percent).



A random sparse matrix
n = 32, dens = .005

Our experiments may vary any of the five input parameters *p*,*n*,*dens*,*prat*,*seed* corresponding to the number of processors used, the order of matrix, the off-diagonal density, maximum allowable pivot ratio, and the initial seed for the random number generator. In the experiments reported here only the first three parameters are varied and their effects are noted. The other parameters are fixed at $prat$=.125 and $seed$=2.

The first experiment concerns sparse matrices and not parallel processing. It shows the effects of matrix order and density on execution time for a fixed number of processors. The number of processors was held at 32, n was varied from 100 to 1000 and dens was varied from .001 to .010. The results are shown in Figures 11 and 12. Figure 11 shows that, for fixed order, the execution time is roughly a linear function of density. Also the number of nonzero elements is a linear function of density, since as the density increases so does nonzero's. The circle points on the figure represent different values of density for fixed matrix order(n=1000). It is obvious from the figure that execution time is also dependent on number of nonzero's, since the slope of the line is much higher for larger order of matrices. This is because a change in density has greater effects on larger order of matrices. Figure 12 shows that, for fixed density, the execution time as a function of matrix order increases faster than linearly. A sparse matrix of density value of one (dens = 1) is a full matrix with no zero off-diagonal. For a full matrix the elimination requires n^3 operations. Therefore matrix order as a function of execution time for a full matrix would be proportional to n^3. On the other hand a diagonal matrix is a matrix with no off-diagonal elements which can be represented by a sparse matrix of density value zero (dens=0). For a diagonal matrix elimination requires n

operation. Therefore matrix order as function of execution time for a diagonal matrix is proportional to n. In Figure 12 curves for different values of the density lies between a full matrix and a diagonal matrix curves mentioned above.

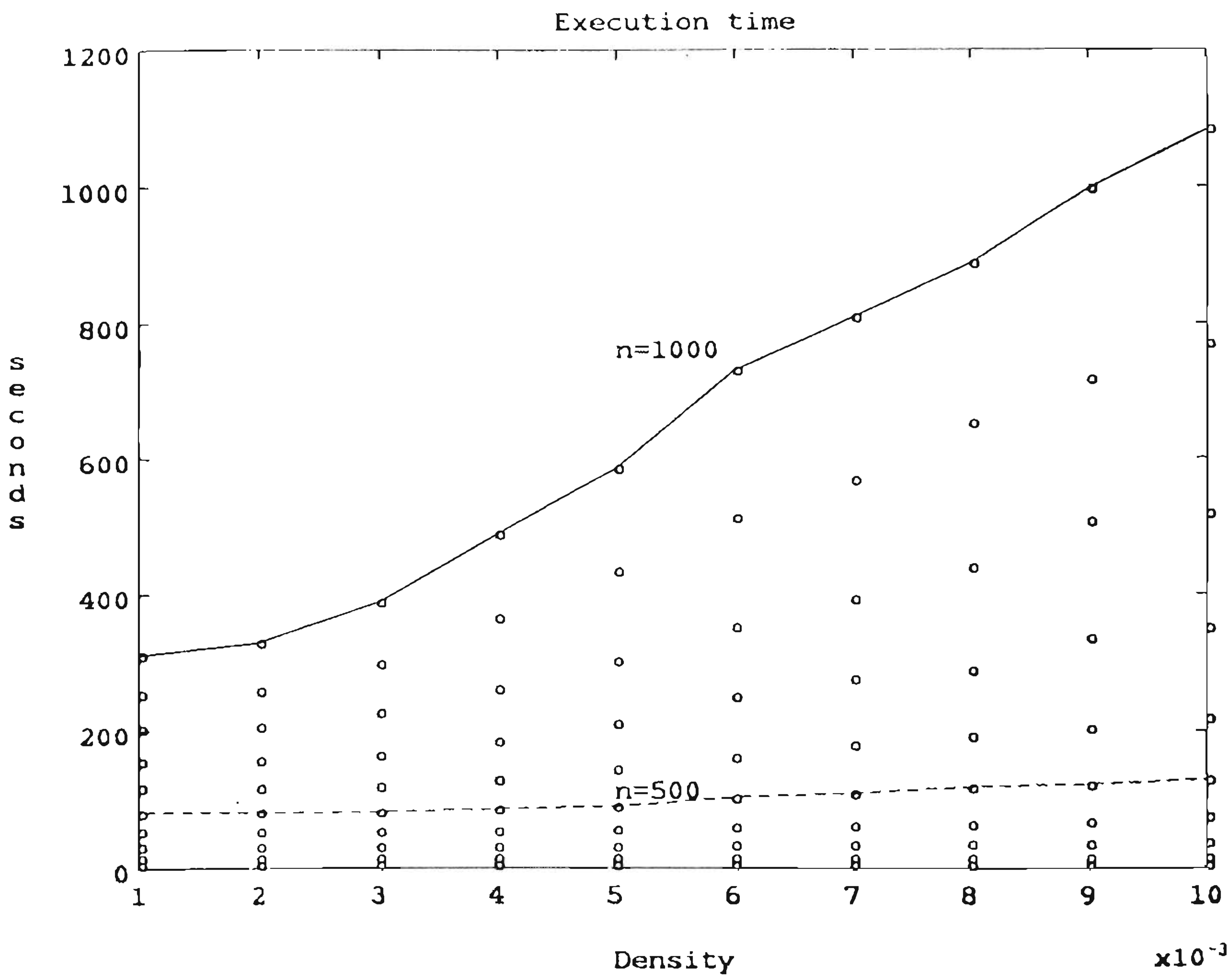
The overall effects of order and density and time appear to be difficult to model analytically. One possible component of such a model is shown in figure 13. Our experiments counted the number of nonzero elements present in the final LU data structure and the number of floating point operations -- additions, multiplications and divisions -- used during factorization. Figure 13 shows that the relation between these two quantities is nearly independent of density and can be fairly well modeled by the equation

$$ops = K \cdot (nz)^a$$

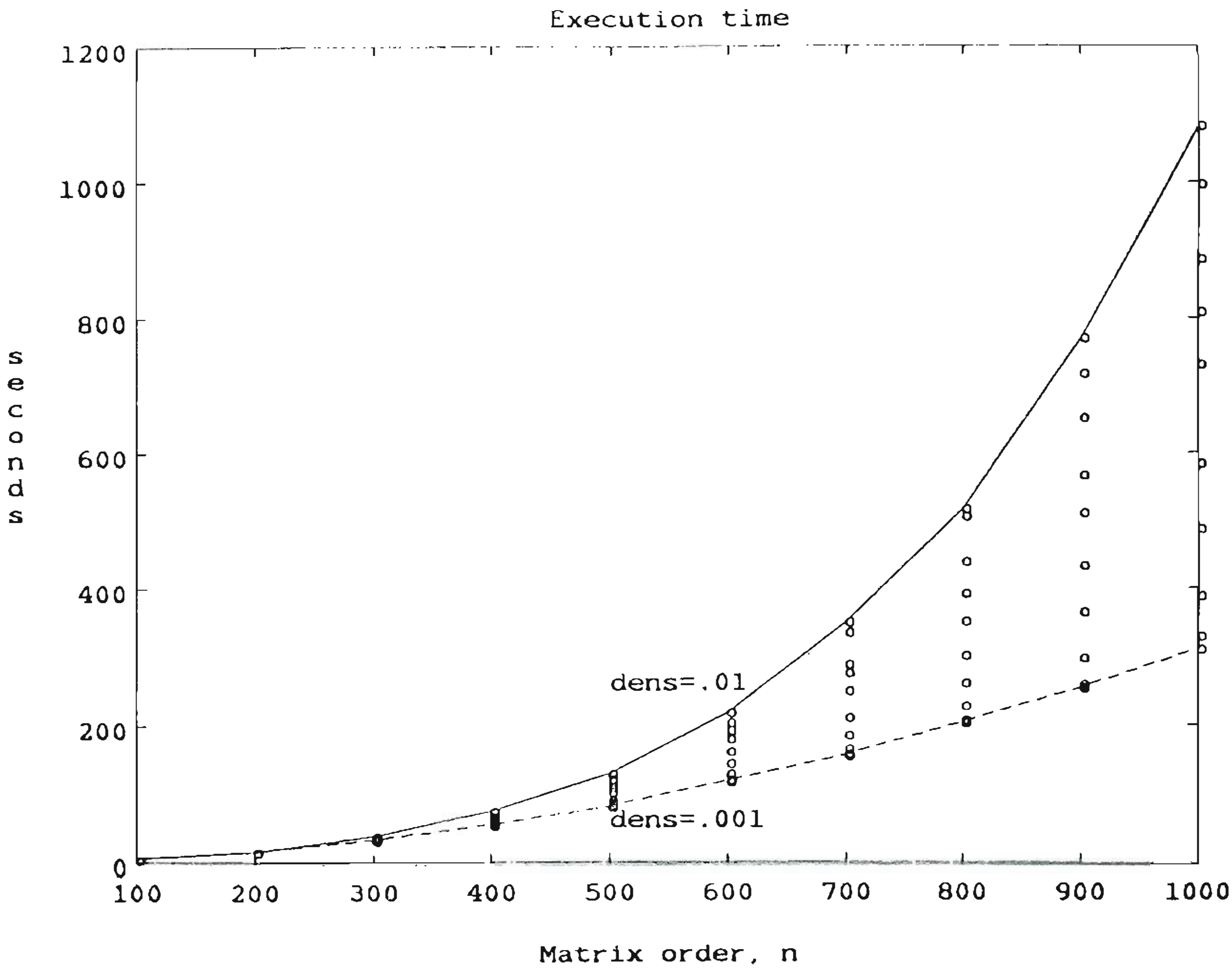
where nz is number of nonzeros in the final LU. A logarithmic least squares fit found $K = 0.0895$ and $a = 1.64$. Of course, the fit is best suited for large values of nz and ops because that is were the most operations occur. For a full matrix elimination requires $\frac{2}{3} \cdot n^3$ operations. If represented as a function of nonzero elements it requires $nz^{1.5}$ number of operations.

The second experiment measures parallel speedups. The density was kept fixed at .005. The dimension of the hypercube was varied from 0 to 5, so the number of processors varied from 1 to 32 in powers of 2. For a given number of processors, memory size limits the maximum order that can be handled. These limits are seen in figure 14 which shows the values of matrix order n that were used for various p . It can be seen that matrices of orders up to 400 and off-diagonal sparsity of 0.005 (one-half of a percent) can be stored on one processor. With 32 processors, orders up to 1000 can be stored.

Figure 15 shows the aggregate megaflop rate (millions of floating point operations per second for the total multiprocessor system) measured during the LU factorization of these matrices. It can be seen for a fixed problem size, the aggregate megaflop rate increases with the number of processors, thereby showing parallel speedup. The obvious question, "Is the speedup linear?", is hard to answer. Problems which are small enough to run on one processor are inefficient on 32 processors and so do not show good speedup. The curves at the bottom of the figure demonstrates the deterioration as more processors are used. On the other hand, problems which are large enough to efficiently utilize many processors will not fit on one processor. This is demonstrated by the curves associated with the larger order of matrices. The straight line in the figure is an attempt to provide a speedup guide. It is a constant, tau, times the number of processors. In Moler's experiments with dense matrices [Mol86] tau is the maximum megaflop rate for the inner loop, DAXPY, and has the value .030. The corresponding quantity for our sparse matrix experiments would be the megaflop rate for YAXPY, but this depends upon density and resulting fill-in. So we have somewhat arbitrarily set tau = .006.

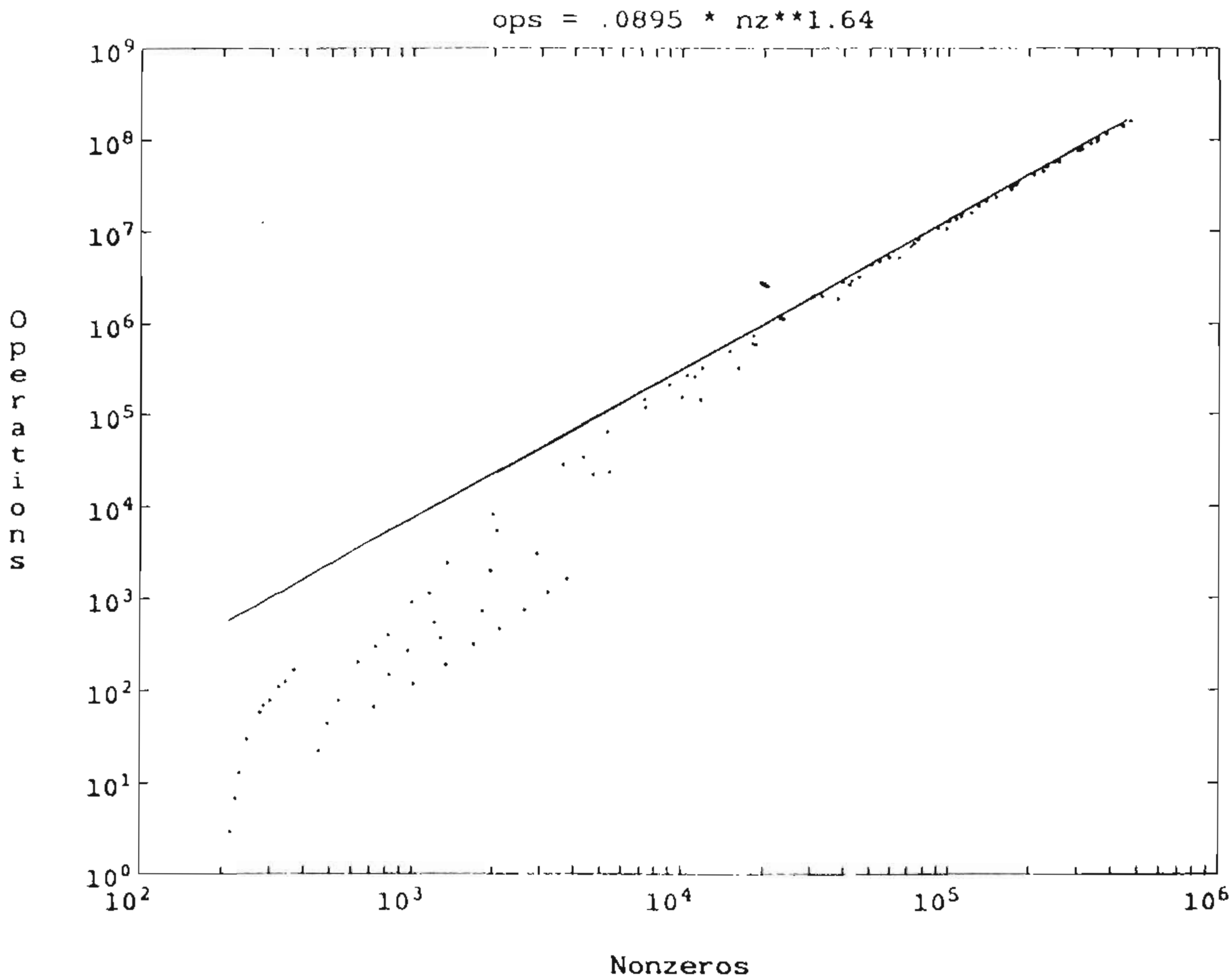


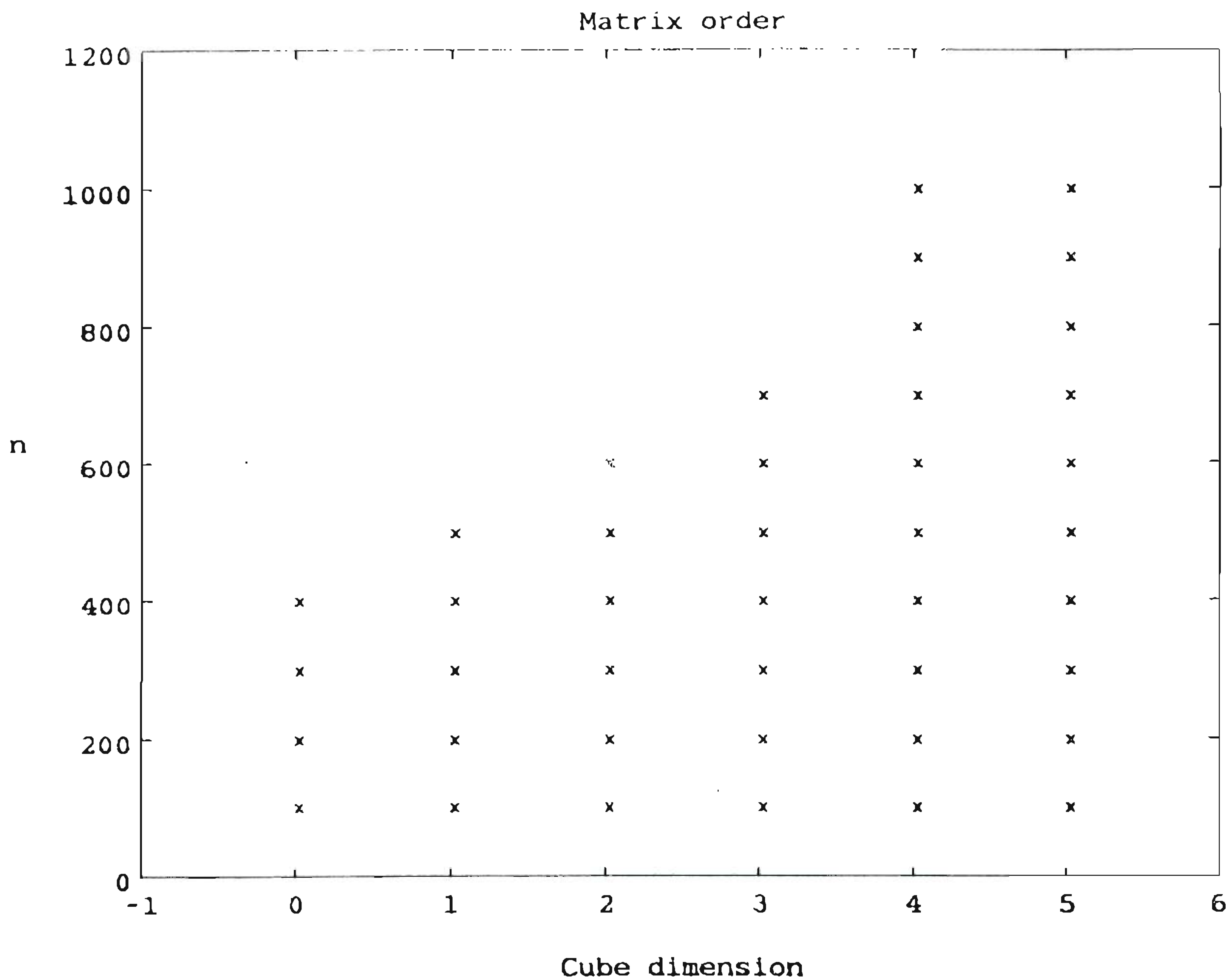
Execution time
1200
1000
800
600
400
200
0
seconds
n=1000
n=500
1
2
3
4
5
6
7
8
9
10
Density
x10^-3

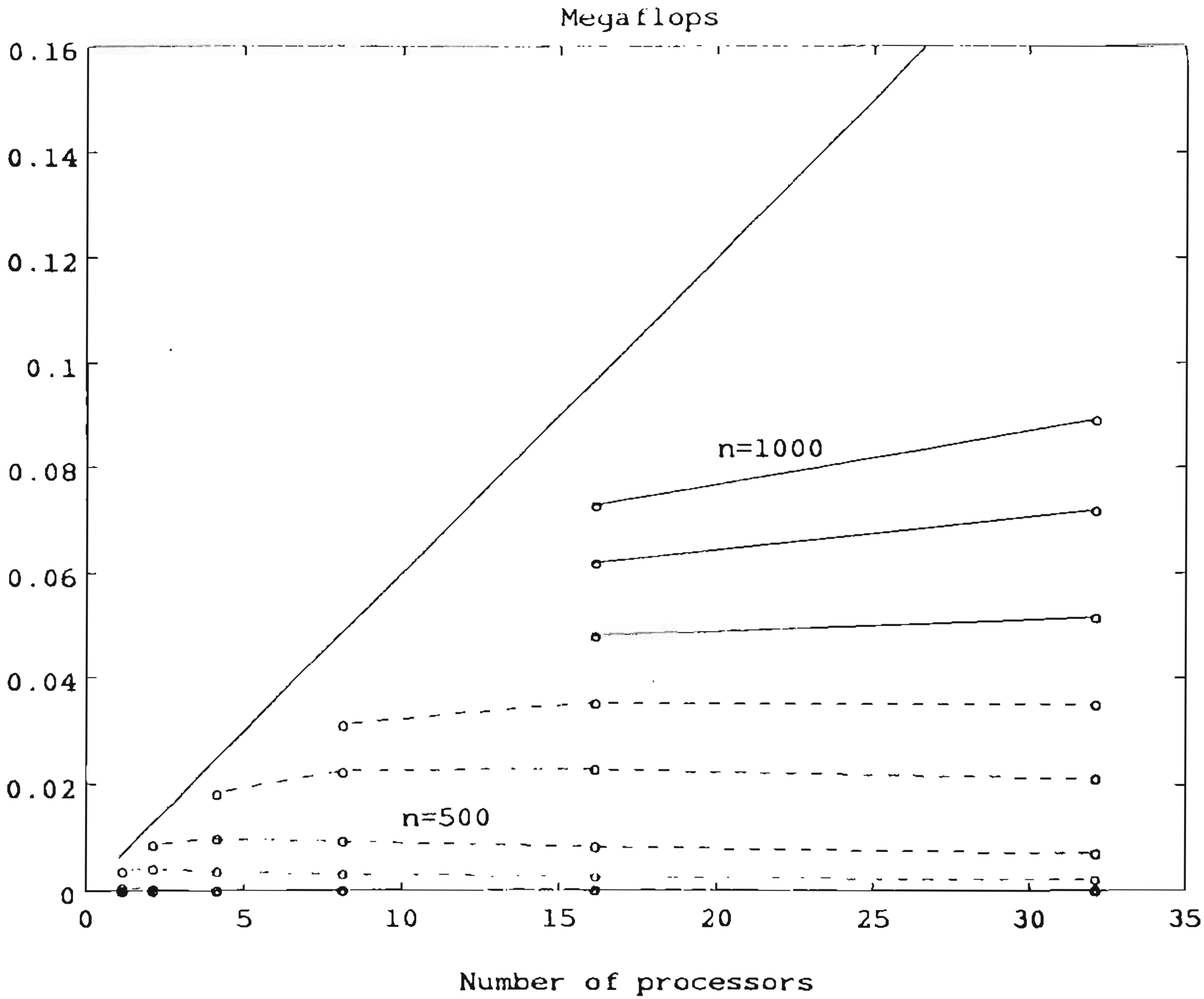


Execution time
seconds
Matrix order, n
dens=.01
dens=.001
0
200
400
600
800
1000
1200
100
200
300
400
500
600
700
800
900
1000

Fig. 13 Nonzero elements versus operation count




Matrix order
n
0
200
400
600
800
1000
1200
-1
0
1
2
3
4
5
6
Cube dimension


Megaflops
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
n=1000
n=500
0
5
10
15
20
25
30
35
Number of processors

CHAPTER 6

CONCLUSIONS

This thesis has developed an algorithm for sparse LU-decomposition and solutions that are suitable for multiprocessor system with local memory. The data structure of the algorithm consist of triplets *row*,*next*,*val*. Each nonzero element of the matrix is stored on a triplet which requires total of twelve bytes of memory (2 bytes each for *row*,*next* and 8 bytes for the double precision value *val*). An efficient load balancing was achieved by distributing the columns of the matrix across the nodes using the wrap fashion so elimination in the natural order will be balanced.

Numerical experiments performed on an iPSC $d=5$ system have been presented which demonstrate the behavior of the algorithm. The result indicate that for matrices of large order the speedup is much better than for smaller order of matrices. Also when using random sparse matrices it is difficult to be precise about speed because several variables must be considered such as density and maximum allowable pivot ratio.

Moreover, matrices with random sparsity pattern are probably not representative of problems encountered in practice. For example in civil engineering connectivity of structures and circuits leads into non-random sparsity. However for specific use of the algorithm it is advised that more extensive performance analysis be carried out before using on real problems.

BIBLIOGRAPHY

[DUF77]
DUFF, IAIN S., A Survey of Sparse Matrix Research, *Proceedings of the IEEE 65*, 4 (April 1977), 500-535.

[DMB79]
Dongarra, J. J., Cleve B. Moler, J. R. Bunch and G. W. Stewart, *Linpack Users' Guide*, SIAM, Philadelphia, PA, 1979.

[DGK84]
Dongarra, J. J., F. G. Gustavson and A. Karp, Implementing Linear Algebra Algorithms For Dense Matrices on a Vector Pipelined Machine, *SIAM Review 26*, 1 (January, 1984), 91-112.

[DuR83]
Duff, I. S. and J. K. Reid, The Multifrontal Solutions of Indefinite Sparese Symmetric Linear Equations, *ACM. Transactions on Mathematical Software 9*, 3 (September 1983), 302-325, Association for computing Machinery.

[FoM67]
Forsythe, George and Cleve B. Moler, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1967.

[GeL81]
George, J. A. and j. W-H Liu, *Computer Solution of Large Sparse Positive Definite System*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[GHL86]
George, J. A., M. T. Heath, J. W-H. Liu and E. G-Y. Ng, Sparse Cholesky Factorization on a Local-memory Multiprocessor, ORNL/TM-9962, Oak Ridge National Laboratory, Oak Ridge, Tennessee, April 1986.

[GHL87]
George, J. A., M. T. Heath, J. W-H. Liu and E. G-Y. Ng, Solution of Sparse Positive Definite systems on a Shared-Memory Multiprocessor, ORNL/TM-10260, Oak Ridge National Laboratory, Oak Ridge, Tennessee, January 1987.

[Jor84]
Jordan, H. F., Experience with pipelined multiple instruction stream, *Proceedings of the IEEE 72*(1984), 113-123.

[Liu86]

Liu, W. H., A Compact Row Storage Scheme for Cholesky factors using Elimination Trees, *ACM. Transactions on Mathematical Software 12*, 2 (June 1986), 127-148, Association for Computing Machinery.

[MoS86]

Moler, Cleve B. and David S. Scott, Communication Utilities For The iPSC, No. 2, Intel Corporation, Beaverton, OR, August 1986.

[Mol86]

Moler, Cleve B., *Matrix Computation on the Intel Hypercube*, Intel Corporation, Beaverton, OR, August 1986.

APPENDIX A

HOST PROGRAM

```
      include 'sparse.h'
      integer i, j, cubedim, dim, pmax, n, p, maxn, method
      integer pid, nid, cid, copen, type, cnt, ldble, lmesg
      integer ktol, NZERO
      double precision dens,eps,t,prat, mesg(5), secs, pfloat
      double precision en, res, mflps, mesg2(5)
      double precision x(NMAX), xx(NMAX), b(NMAX), D(NMAX)
      character*10 OK
      data ldble/8/ , lmesg/40/
c
      open (unit=10,status='unknown',file ='result')
      dim = cubedim()
      pmax = 2**dim
      eps = 0.5d0**52
c
  110 print *
      print *,'Enter number of processors, order of matrix : '
      read (*,*,end=99,err=98) p, n
      if ( p .gt. pmax) go to 98
      pfloat = p
      maxn = dsqrt(pfloat*MAXMEM + (1.5*pfloat)**2) - (1.5* pfloat)
      if (n .le. 0 .or. n .gt. maxn) then
            n = maxn
            write(*,'(" n reset to ",i5)') n
      endif
      print '(2(i4,2x))', p, n
      if ( p. eq. 0 ) go to 99
      print * 'Off-diagonal density:  '
      read (*,*,end=99,err=98) dens
c (the density applies to the off-diagonal elements. the diagonal is
c  always full)
      if (dens.lt.0d0 .or. dens.gt.1.0d0) then
            print *, 'Density must be less than 1 and positive: '
```

```
        read (*,*,end=99,err=98) dens
      endif
      print '( f10.6)', dens
c
      prat=0.125d0
      print *,'Enter the ratio : minimum acceptable / maximum pivot'
      print *,'      (a nonpositive value is replaced by 1/8) : '
      read *,t
      if (t.ge.0d0 .and. t.le.1d0) prat=t
      print '(f9.6)',prat
      print *
      print * ' Method of Solve (1,2) : '
      read (*,*,end=99,err=98) method
      if ( method .lt. 1 .or. method .gt. 2) then
            print *,' Method  must be  (1 or 2 ) so default 1 taken : '
        method = 1
      endif
      print '('' Method using for Solve is: '',i4)', method
      print *
      write(*,9)
    9 format(3x,'Task',7x,'p',4x,'n',6x,'dens',7x,'secs',9x,'mflps',
     >       6x,'residual', 6x, 'OK?')
c
c     Send problem specification to node 0
c
      mesg(1) = p
      mesg(2) = n
      mesg(3) = dens
      mesg(4) = prat
      mesg(5) = method
      type = 101
      pid = 0
      cid = copen(pid)
      call sendmsg(cid,type,mesg,lmesg,0,pid)
      if (p .le. 0) go to 99
c
c     Generate matrix
c
      call recvmsg(cid,type,secs,ldble,cnt,nid,pid)
      write(*,20) p, n, dens, secs
   20 format(' Generate: ',2(i4,2x), (f8.3,2x), (f12.6,2x))

c
```

```
c    Matrix - matrix multiplication.
c
      call recvmsg(cid,type,secs,ldble,cnt,nid,pid)
      mflps = 2.0d0*en**2/secs * 1.d-6
      write(*,30) p, n, dens, secs, mflps
   30 format(' Multiply: ',2(i4,2x), (f8.3,2x), (f12.6,2x),(f8.3,2x))
c
c    LU - factorization of A
c
      call recvmsg(cid,type,secs,ldble,cnt,nid,pid)
      en = n
      mflps = 0.666667d0*en**3/secs * 1.d-6
      write(*,40) p, n, dens, secs, mflps
   40 format(' Factor  : ',2(i4,2x), (f8.3,2x), (f12.6,2x),(f8.3,2x))

c
c    Solve linear system.
c
      call recvmsg(cid,type,secs,ldble,cnt,nid,pid)
      mflps = 2.0d0*en**2/secs * 1.d-6
      write(*,50) p, n, dens, secs, mflps
   50 format(' Solve   : ',2(i4,2x), (f8.3,2x), (f12.6,2x),(f8.3,2x))

c
c     Residual -- calclated
c
      if ( n .le. lprint) then
            call recvmsg(cid,type,D,NMAX*ldble,cnt,nid,pid)
            call recvmsg(cid,type,b,NMAX*ldble,cnt,nid,pid)
            call recvmsg(cid,type,x,NMAX*ldble,cnt,nid,pid)
            call recvmsg(cid,type,xx,NMAX*ldble,cnt,nid,pid)
      endif
      call recvmsg(cid,type,res,ldble,cnt,nid,pid)
      if ( res .lt. n*eps) then
                    OK = 'OK        '
      elseif ( res .lt. 1000.0*n*eps) then
                    OK = 'Suspicious'
      else
                    OK = 'TROUBLE!!!'
      endif
      secs = 0.0d0
      mflps = 0.0d0
      write(*,60) p, n, dens, secs, mflps, res, OK
```

```
 60 format(' Residual: ',2(i4,2x), (f8.3,2x), (f12.6,2x),
   > (f8.3,2x),1pd13.3,1x,a10)

      if ( n .le. lprint ) then
             write (10,*)
             write (10,*)' ------------------------------------'
             write (10,*)
             write(10 , *) ' The Diagonal D is : '
             write (10,*)
             call vwrite(D,n)
             write (10,*)
             write(10 , *) ' The exact x is : '
             write (10,*)
             call vwrite(xx,n)
             write (10,*)
             write(10 ,*) ' The computed x is : '
             write (10,*)
             call vwrite(x,n)
             write (10,*)
             write(10, *) ' The exact b is : '
             write (10,*)
             call vwrite(b,n)
      endif

      print *
c
c            Operation and storage count
c
      call recvmsg(cid,type,mesg2,5*ldble,cnt,nid,pid)
      kadd = mesg2(1)
      kmul = mesg2(2)
      kdiv = mesg2(3)
      memptr = mesg2(4)
      NZERO = mesg2(5)
      print *,' fl-pt addition, multiplications, division, and total'
      ktot = kadd + kmul + kdiv
      print '(3i15)', kadd, kmul, kdiv, ktot
      print *' Total number of triplets stored'
      print '(i15)', memptr
      print *' Total number of NON zero elements'
      print '(i15)', NZERO
      print *'---------------------------------------------'
c
```

```
c
      go to 110
c
   98 write(*, '(" Something wrong with input, try again")')
      go to 110
   99 stop
      end
```

NODE PROGRAM

```
c     main node program for Parrallel Sparse Computation of Linear
c     Equations.
c
c
      include 'sparse.h'
      character *50 string
      character *5 case
      integer A(NMAX),L(NMAX),U(NMAX),pvt(NMAX),krow(NMAX)
      integer cid, p, n, m, cnt, copen, type, pid, method
      integer lmesg, cubedim, dpsize, gkrow(NMAX)
      integer id, i, j, h, root, hid, mynode, dim, dimcube
      integer click, clock
      integer iy, iysave , R, NZERO, acmemptr
      double precision D(NMAX),b(NMAX),x(NMAX), xx(NMAX)
      double precision acx(NMAX),z(NMAX), res
      double precision dens, randum, urand, prat, fsum, dasum
      double precision secs, t, mesg(5),mesg2(5)
      double precision normx, norma
      data lmesg/40/, hid/-32768/, root/0/, dpsize /8/
c
c Open one channel
c
      pid = 0
      cid = copen(pid)
c
c Reieve problem size information from host.
c
      iy = 0
   1  type = 101
      iysave = iy
      NZERO = 0
      call grecvw (cid, type, mesg, lmesg, cnt, cubedim())
c
c  p = number of processors
c  n = order of the matrix
c  dens = density of the matrix
c  prat = ratio : minimum acceptable/ minimum pivot.
c  method = method of Solve
c
      p = mesg(1)
      n = mesg(2)
```

```
      dens = mesg(3)
      prat = mesg(4)
      method = mesg(5)
      if (p .le. 0) go to 200
c
      dim = dimcube(p)
      id  = mynode()
      if (id .ge. 2**dim) go to 10
c
c     m = number of columns in this process.
c
      m = n/p
      if ( id .lt. MOD(n,p)) m = m + 1
      if ( id .ge. p) m = 0
      click = clock()
c
c
c Initialize the row counts of nonzero elements
      do 120 i=1,n
            b(i) = 0.0d0
            D(i) = 0.0d0
            x(i) = 0.0d0
            xx(i) = 0.0d0
            krow(i)=0
 120 continue
c
c initialize the flop counts
      kadd=0
      kmul=0
      kdiv=0
c
c generation of a "random sparse matrix" with integer coefficients ...
      memptr = 0
      norma = 0.0d0
      h = id + 1
      do 140 j = 1, m
        call init(A(j),cid)
        do 130 i = n,1,-1
          if (i .eq. h .or.  urand(iy) .le. dens) then
            call insert(A(j),randum(iy),i,cid)
                NZERO = NZERO + 1
                krow(i)=krow(i)+1
          endif
```

```
 130    continue
            norma  = dmax1( norma, fsum(A(j)) )
            h = h + p
 140 continue
c
c ... and a random solution with integer components ...
c
      if ( id .eq. root )  then
          do 150 j =  1, n
                  x(j) = randum(iy)
 150       continue
      endif
c
      secs = (clock() - click)/1000.d0
      call gop(cid. type, secs, 1, 'M', hid, dim, t)
      call gop(cid, 2*type+1, norma, 1, 'M', root, dim, t)
      if ( id .eq. root) call dcopy(n, x, 1, xx, 1)
c
c  Matrix - matrix multiply
c
      click = clock()
      call pgsmul(A, n, m, p, cid, id, x, b,z)
      secs = (clock() - click )/1000.d0
      call gop(cid, type, secs, 1, 'M', hid, dim, t)
      call dcopy(n, b, 1, x, 1)
c
c  LU - factorization of A
c
      click = clock()
      call pgsfa(A,n,m,p,cid,id,prat,krow,gkrow,pvt,L,U,D)
      secs = (clock() - click )/1000.d0
      call gop(cid, type, secs, 1, 'M', hid, dim, t)

c
c  Solve linear system.
c
      if ( method .eq. 1) then
          click = clock()
          call pgssl1(cid,L, U, D, n, m, p, pvt, x, id)
          secs = (clock() - click )/1000.d0
          call gop(cid, type, secs, 1, 'M', hid, dim, t)
      endif
```

```
      if ( method .eq. 2) then
            click = clock()
            call pgasl2(cid,L, U, D, n, m, p, pvt, x, id)
            secs = (clock() - click )/1000.d0
            call gop(cid, type, secs, 1, 'M', hid, dim, t)
      endif

      call igop(cid,,type+4, memptr, 1, '+', root, dim, t)
      if (id .eq. root) then
              acmemptr = memptr
      endif

c Regeneration of a "random sparse matrix" with integer coefficients ...
c
      iy = iysave
      memptr = 0
      h = id + 1
      do 170 j = 1, m
        call init(A(j),cid)
        do 160 i = n,1,-1
          if (i .eq. h .or.  urand(iy) .le. dens) then
            call insert(A(j),randum(iy),i,cid)
                krow(i)=krow(i)+1
          endif
  160   continue
            h = h + p
  170 continue
c
c
c
c  Residual - Check the relative residual of
c

      call gop(cid, 8*type+1, D, n, '+', root, dim, z)
      if ( id .eq. root) then
              normx = dasum(n,x,1)
              kadd = kadd + n
         if ( n .le. lprint ) then
                        call sendw(cid, type, D, n*dpsize, hid, pid)
                        call sendw(cid, type, b, n*dpsize, hid, pid)
                        call sendw(cid, type, x, n*dpsize, hid, pid)
                        call sendw(cid, type, xx, n*dpsize, hid, pid)
         endif
```

```
      endif
      call pgsmul(A, n, m, p, cid, id, x, acx, 2)
      if( id .eq. root) then
         res = 0.0d0
         do 190 j = 1, n
               res = res + dabs(b(j) - acx(j))
 190     continue
         res = res/(norma * normx)
         call sendw(cid, type, res, dpsize, hid, pid)
      endif
c Send operation counts
c
         call igop(cid, type+1, kadd, 1, '+', root, dim, t)
         call igop(cid, type+2, kmul, 1, '+', root, dim, t)
         call igop(cid, type+3, kdiv, 1, '+', root, dim, t)
         call igop(cid, type+5, NZERO, 1, '+', root, dim, t)
         if (id .eq. root ) then
               mesg2(1) = kadd
               mesg2(2) = kmul
               mesg2(3) = kdiv
               mesg2(4) = acmemptr
               mesg2(5) = NZERO
               call sendw(cid, type, mesg2, 5*dpsize, hid, pid)
         endif
c
c
      go to 10
c
c  Quitely terminate
c
 200  continue
      end
```

PGSMUL

```
      subroutine pgsmul (A,n,m,p,cid,id,x,y,t)
      integer n,m,p,cid,id
      integer A(n)
      double precision x(n),y(n), t(n)
      character *100 string
c
c     y = A*x
c
c     Input..
c          A   distributed over p nodes
c          x   node 0
c
c     Output..
c          y   node 0
c          x   destroyed
c
      integer i,j,l,root,dimcube,cnt,tmul
      double precision s
      data root/0/, tmul/7001/
c
      if (id .eq. root) then
        call gsendw(cid, tmul, x, 8*n, dimcube(p))
      else
        call grecvw(cid, tmul, x, 8*n, cnt, dimcube(p))
      endif
c
      do 50 i = 1, n
        y(i) = 0.0d0
   50 continue
      l = id+1
      do 51 j = 1, m
        s = x(l)
        call faxpy (s,A(j),y)
        l = l+p
   51 continue
c
      call gop(cid, tmul+1 , y, n, '+', root, dimcube(p), t)
      end
```

PGSFA

```
      subroutine pgsfa(A,n,nm,p,cid,id,prat,krow,gkrow,pvt,L,U,D)
c
c LU decomposition of a sparse matrix by compact elimination
c
c on entry
c
c     A      integer (n)
c            pointers to the columns of the sparse matrix
c     n      integer
c            order of the sparse matrix
c     prat   double precision
c            ratio : minimum acceptable / maximum pivot
c     krow   integer (n)
c            record of the number of nonzero elements in the rows
c            of the sparse matrix
c
c on return
c
c     pvt    integer (n)
c            record of the row exchanges incurred in the LU
c            decomposition
c     L      integer (n)
c            pointers to the columns of the sparse lower triangular
c            factor (unit diagonal omitted)
c     U      integer (n)
c            pointers to the columns of the sparse upper triangular
c            factor (diagonal omitted)
c     D      double precision (n)
c            diagonal elements of the upper triangular factor
c     Note that upon return A points at the columns of the upper
c     triangular factor (diagonal omitted). The triplets of the
c     original sparse matrix are overwritten by the those of the
c     upper and lower triangular factors.
c
      include 'sparse.h'
      parameter(NMAX2 - NMAX + NMAX)
      integer A(n),pvt(n),L(n),U(n),krow(n)
      integer gkrow(n), tt(NMAX)
      integer n, p, cid, id, cnt, f, trow,root
      double precision D(n),prat,t, BUF(NMAX2)
      integer j,k,h,m,pividx,R, LDBLE, dimcube
```

```
      integer nm, nz, dim
      character*70 string
      character*5 case
      data LDBLE /8/ , trow/8001/, root/0/

      dim = dimcube(p)
c
c make sure that prat is valid
      prat=dmin1(1d0, dabs(prat))
c
c define U
      do 10 k = 1, nm
        U(k) = A(k)
   10 continue
c
      h = 1
      do 30 k = 1, n
c
c Process R (for root)
      R = MOD(k-1,p)
c
c Find global krow values
c
                    call icopy(n, krow, 1, gkrow,1)
                    call igop(cid,trow,gkrow,n,'+',R,dim,tt)
c
c if Process R (for root)
c
      if (R .eq. id) then
c
c choose the k-th pivot
c
            m = pividx(A(h),k,prat,gkrow)
            pvt(h) = m
            if (m .ne. k) call swap(A(h),k,m)
c
c save the pivot in D and flag the end of the k-th column of U
            D(k) = val(A(h))
            if (D(k) .eq. 0.0d0) then
                      write(string,'("zero pivot on column : ",i2)') k
                      call syslog(cid,string)
                    endif
            row(A(h)) = FOOT
```

```
c
c compute the elimination multipliers
          L(h) = next(A(h))
          if (D(k) .ne. 0.0d0) call scoll(L(h),D(k),krow,cid)
                    call pack(cid,BUF,L(h),m,nz)
                    call gsendw(cid, k, BUF, nz*LDBLE, dimcube(p))
                    h = h + 1
c
c  Wait for elimination information other than R
c
     else
                    call grecvw(cid, k, BUF, NMAX2*LDBLE, cnt, dimcube(p))
                    m = BUF(1)
     endif
c
c Exchange krows if pivot column is not k-th
c
          if(m .ne. k) then
                    f = krow(k)
                    krow(k) = krow(m)
                    krow(m) = f
          endif
c
c apply all transformations to (k+1)st column
             do 20 j = h , nm
                    if (m .ne. k) call swap(A(j), k, m)
                    if ( row(A(j)) .eq. k) then
                            t= -val(A(j))
                            A(j)= next(A(j))
                            call yaxpy(t,BUF,A(j),krow,cid)
                    endif
  20      continue
  30 continue
    return
    end
```

PGSSL1

```
      subroutine pgssl1(cid,L,U,D,n,mm,p,pvt,b,id)
c
c Solution of a system of linear equations whose sparse matrix is in
c the LU form provided by subroutine pgsfa.
c
c on entry
c
c     L      integer (n)
c            pointers to the columns of the lower triangular factor
c            (unit diagonal omitted), as returned by dgsfa
c     U      integer (n)
c            pointers to the columns of the upper triangular factor
c            (diagonal omitted), as returned by pgsf1
c     D      double precision (n)
c            diagonal elements of the upper triangular factor as
c            returned by dgsfa
c     n      integer
c            order of the system
c     mm     integer
c            number of the cloumn on each node.
c     p      integer
c            number of the proccessors
c     pvt    integer (n)
c            record of exchanges returned by pgsfa
c     b      double precision (n)
c            right-hand side of the system
c
c on return
c
c     b      double precision (n)
c            solution of the system
c
      include 'sparse.h'
      integer n,mm,L(n),U(n),pvt(n)
      integer p, type, pid, pred, succ, BYTES
      double precision D(n),b(n)
      double precision  t
      integer k,m, cid, id, h, R
      character *60 string
      character *5 case
      data type /4/
```

```
      pred = MOD(id-1+p, p)
      succ = MOD(id+1+p, p)
      BYTES = 8 * n
      pid = 0
c
c forward substitution L*y = b

      k = id + 1
      do 10 h = 1, mm
         if ( k .gt. 1 .and. pred .ne. id)
     >      call recvw(cid, type, b, BYTES, BYTES, pred, pid)
      m = pvt(h)
      if ( m .ne. k) call swreal(b(k) , b(m))
      if ( b(k) .ne. 0.0d0) call faxpy(-b(k), L(h), b)
      if ( k .lt. n .and. succ .ne. id)
     >      call send(cid, type, b, BYTES, succ, pid)
      k = k + p
   10 continue
c
c backward substitution U*x = y

      do 20 h = mm, 1, -1
         k = k - p
         if( k .lt. n .and. succ .ne. id)
     >      call recvw(cid, type, b, BYTES, BYTES, succ, pid)
         if (D(k) .ne. 0.0d0) b(k) = b(k)/D(k)
         kdiv=kdiv+1
         t = b(k)
       if (t .ne. 0.0d0) call faxpy(-t,U(h),b)
         if (k .gt. 1 .and. pred .ne. id)
     >      call send(cid, type, b, BYTES, pred, pid)
   20 continue
      return
      end
```

PGSSL2

```
      subroutine pgssl2(cid,L,U,D,n,mm,p,pvt,b,id)
c
c Solution of a system of linear equations whose sparse matrix is in
c the LU form provided by subroutine dgsfa.
c
c on entry
c
c     L        integer (n)
c              pointers to the columns of the lower triangular factor
c              (unit diagonal omitted), as returned by dgsfa
c     U        integer (n)
c              pointers to the columns of the upper triangular factor
c              (diagonal omitted), as returned by dgsfl
c     D        double precision (n)
c              diagonal elements of the upper triangular factor as
c              returned by dgsfa
c     n        integer
c              order of the system
c     mm       integer
c              number of the cloumn on each node.
c     p        integer
c              number of the proccessors
c     pvt      integer (n)
c              record of exchanges returned by dgsfa
c     b        double precision (n)
c              right-hand side of the system
c
c on return
c
c     b        double precision (n)
c              solution of the system
c
      include 'sparse.h'
      parameter ( NMAX2 = NMAX + NMAX )
      integer n,mm,L(n),U(n),pvt(n)
      integer p, dimcube, dim, cnt , type, DPSIZE
      integer k,m, cid, id, h, R, nz
      double precision D(n),b(n), BUF(NMAX2)
      double precision t
      character *60 string
      data type /512/, DPSIZE /8/
```

```
c
c forward substitution L*y = b

      dim = dimcube(p)
      nz = 0
      h = 1
      do 10 k = 1, n
         R = MOD( k-1, p)
         if ( R .eq. id) then
        m = pvt(h)
            call pack(cid,BUF, L(h), m, nz)
            call gsendw(cid, 10*type+k , BUF, nz*DPSIZE, dim)
            h = h + 1
         else
            call grecvw(cid, 10*type+k, BUF, NMAX2*DPSIZE, cnt, dim)
            m = BUF(1)
         endif
       if (m .ne. k) call swreal(b(k),b(m))
       if (b(k) .ne. 0.0d0) call baxpy(-b(k),BUF,b)
   10 continue
c
c backward substitution. U*x = y

      h = mm
      do 20 k = n, 1, -1
         R = MOD( k-1, p)
         if ( R .eq. id) then
       if (D(k) .ne. 0.0d0) b(k) = b(k)/D(k)
           kdiv=kdiv+1
           call dpack(cid,BUF, U(h), b(k), nz)
           call gsendw(cid, 20*type+k , BUF, nz*DPSIZE, dim)
           h = h - 1
         else
           call grecvw(cid, 20*type+k, BUF,  NMAX2*DPSIZE, cnt, dim)
           b(k) =  BUF(1)
         endif
       if (b(k) .ne. 0.0d0) call baxpy(-b(k),BUF,b)
   20 continue
      return
      end
```

BLAS functions

A.1. FAXPY

```
      subroutine faxpy(alfa,xx,y)
c
c Addition of a multiple of a sparse vector to a full vector
c            y <- y + alfa * sparse (xx)
c
c on entry
c
c     alfa   double precision
c            multiplier of the sparse vector
c     xx     integer
c            pointer identifying the sparse vector
c     y      double precision (*)
c            full vector
c on return
c
c     y      modified full vector
c
      double precision alfa,y(1)
      integer x,xx ,d
      include 'sparse.h'
      character*100 string
c
      x=xx
10    if(row(x).eq.FOOT) return
        y(row(x))=y(row(x))+alfa*val(x)
            kadd=kadd+1
            kmul=kmul+1
        x=next(x)
      go to 10
      end
c
```

A.2. BAXPY

```
c
      subroutine baxpy(alfa,x,y)
c
c Addition of a multiple of a full vector to a full vector
c              y <- y + alfa * x
c
c on entry
c
c     alfa    double precision
c             multiplier of the sparse vector
c     x       double precision (*)
c             full vector
c     y       double precision (*)
c             full vector
c on return
c
c     y       modified full vector
c
      integer d, k, ix
      double precision alfa,y(1),x(*)
      include 'sparse.h'
      character*100 string
c
      k=2
10    ix = x(k+1)
      if(ix .eq. FOOT) return
        y(ix) = y(ix) + alfa * x(k)
              kadd=kadd+1
              kmul=kmul+1
        k = k + 2
      go to 10
      end
c
```

A.3. FSUM

```
c
      double precision function fsum (xx)
c
c Sum up the values of a sparse vector
c             t = t + sparse (xx)
c
c on entry
c
c     xx     integer
c            pointer identifying the sparse vector
c on return
c
c     fsum      sum of the sparse vector
c
      double precision t
      integer x,xx
      include 'sparse.h'
      character*100 string
c
      x=xx
      t = 0

10    if(row(x).eq.FOOT) then
            fsum = t
            return
      endif

      t = t + dabs(val(x))
      kadd = kadd + 1
      x=next(x)

      go to 10
      end
c
```

A.4. DCOPY

```
c
      subroutine  dcopy(n,dx,incx,dy,incy)
c
c     copies a vector, x, to a vector, y.
c     uses unrolled loops for increments equal to one.
c     jack dongarra, linpack, 6/17/77.
c
      double precision dx(1),dy(1)
      integer i,incx,incy,ix,iy,m,mp1,n
c
      if(n.le.0)return
      if(incx.eq.1.and.incy.eq.1)goto 20
c
c        code for unequal increments or equal increments
c          not equal to 1
c
      ix = 1
      iy = 1
      if(incx.lt.0)ix = (-n+1)*incx + 1
      if(incy.lt.0)iy = (-n+1)*incy + 1
      do 10 i = 1,n
        dy(iy) = dx(ix)
        ix = ix + incx
        iy = iy + incy
   10 continue
      return
c
c        code for both increments equal to 1
c
c
c        clean-up loop
c
   20 m = mod(n,7)
      if( m .eq. 0 ) go to 40
      do 30 i = 1,m
        dy(i) = dx(i)
   30 continue
      if( n .lt. 7 ) return
   40 mp1 = m + 1
      do 50 i = mp1,n,7
        dy(i) = dx(i)
        dy(i + 1) = dx(i + 1)
        dy(i + 2) = dx(i + 2)
        dy(i + 3) = dx(i + 3)
        dy(i + 4) = dx(i + 4)
        dy(i + 5) = dx(i + 5)
        dy(i + 6) = dx(i + 6)
   50 continue
      return
      end
```

A.5. DASUM

```
c
      double precision function dasum(n,dx,incx)
c
c     takes the sum of the absolute values.
c     jack dongarra, linpack, 6/17/77.
c
      double precision dx(1),dtemp
      integer i,incx,m,mp1,n,nincx
c
      dasum = 0.0d0
      dtemp = 0.0d0
      if(n.le.0)return
      if(incx.eq.1)goto 20
c
c        code for increment not equal to 1
c
      nincx = n*incx
      do 10 i = 1,nincx,incx
        dtemp = dtemp + dabs(dx(i))
   10 continue
      dasum = dtemp
      return
c
c        code for increment equal to 1
c
c
c        clean-up loop
c
   20 m = mod(n,6)
      if( m .eq. 0 ) go to 40
      do 30 i = 1,m
        dtemp = dtemp + dabs(dx(i))
   30 continue
      if( n .lt. 6 ) go to 60
   40 mp1 = m + 1
      do 50 i = mp1,n,6
        dtemp = dtemp + dabs(dx(i)) + dabs(dx(i + 1)) + dabs(dx(i + 2))
     *  + dabs(dx(i + 3)) + dabs(dx(i + 4)) + dabs(dx(i + 5))
   50 continue
   60 dasum = dtemp
      return
      end
c
c
c
```

A.6. ICOPY

```
c
      subroutine  icopy(n,dx,incx,dy,incy)
c
c     copies a vector, x, to a vector, y.
c     uses unrolled loops for increments equal to one
c
      integer dx(1),dy(1)
      integer i,incx,incy,ix,iy,m,mp1,n
c
      if(n.le.0)return
      if(incx.eq.1.and.incy.eq.1)goto 20
c
c        code for unequal increments or equal increments
c          not equal to 1
c
      ix = 1
      iy = 1
      if(incx.lt.0)ix = (-n+1)*incx + 1
      if(incy.lt.0)iy = (-n+1)*incy + 1
      do 10 i = 1,n
        dy(iy) = dx(ix)
        ix = ix + incx
        iy = iy + incy
   10 continue
      return
c
c        code for both increments equal to 1
c
c
c        clean-up loop
c
   20 m = mod(n,7)
      if( m .eq. 0 ) go to 40
      do 30 i = 1,m
        dy(i) = dx(i)
   30 continue
      if( n .lt. 7 ) return
   40 mp1 = m + 1
      do 50 i = mp1,n,7
        dy(i) = dx(i)
        dy(i + 1) = dx(i + 1)
        dy(i + 2) = dx(i + 2)
        dy(i + 3) = dx(i + 3)
        dy(i + 4) = dx(i + 4)
        dy(i + 5) = dx(i + 5)
        dy(i + 6) = dx(i + 6)
   50 continue
      return
      end
```

CANDP

```
      subroutine candp(p,q,i)
c
c "Cut and paste" : move one triplet to precede another
c
c on entry
c
c     p       integer
c             pointer identifying the triplet to be moved
c     q       integer
c             pointer identifying the triplet chosen to become the
c             successor of the moved triplet
c     i       integer
c             component index to be assigned to the moved triplet
c
c the effect of a call to candp is equivalent to the sequence
c        call insert(q,val(p),i)
c        call delete(p)
c     but without creating a "dead" element.
c
      integer p,q,i
      include 'sparse.h'
      double precision alfa
      integer t
c
      if (next(p) .eq. q) then
        row(p) = i
      else
c
c remember val(p)
        alfa = val(p)
c
c delete(p), but do not dispose of the (freed) triplet
        t = next(p)
        val(p) = val(t)
        row(p) = row(t)
        next(p) = next(t)
c
c insert(q,alfa,i), using the free triplet
        val(t) = val(q)
        row(t) = row(q)
        next(t) = next(q)
        val(q) = alfa
        row(q) = i
        next(q) = t
      endif
      end
```

INIT

```
      subroutine init(v,cid)
c
c Creation of a new sparse vector initially empty (consisting of a
c footer).
c
c on entry
c
c     v       integer
c             pointer to the first triplet of the new sparse vector
c
      integer v, cid
      include 'sparse.h'
      character*80 string
c
      if (memptr .eq. MAXMEM) then
        write(string,'(" OUT OF MEMORY")')
            call syslog(cid,string)
        STOP
      endif
      memptr = memptr + 1
      v = memptr
      row(v) = FOOT
      val(v) = 0.0d0
      next(v) = v
      end
```

INSERT

```
      subroutine insert(p,alfa,i,cid)
c
c Insertion of a new component in a sparse vector
c
c on entry
c
c     p       integer
c             pointer to the successor of the triplet to be inserted
c     alfa    double precision
c             value of the component to be inserted
c     i       integer
c             index of the component to be inserted
c
      integer p,i, cid
      double precision alfa
      include 'sparse.h'
      character *60 string
c
      if (memptr .eq. MAXMEM) then
        write(string,'(" OUT OF MEMORY")')
            call syslog(cid,string)
        STOP
      endif
      memptr = memptr + 1
      val(memptr) = val(p)
      row(memptr) = row(p)
      next(memptr) = next(p)
      val(p) = alfa
      row(p) = i
      next(p) = memptr
      end
```

PIVIDX

```
      integer function pividx(aa,defalt,prat,krow)
c
c Index of the minimum component of a vector of integers subject to
c the corresponding component of a sparse vector being no less than
c a fraction of its maximum component (in magnitude).
c
c on entry
c
c     aa      integer
c             pointer identifying the sparse vector
c     defalt  integer
c             index to be returned if the sparse vector is empty
c     prat    double precision
c             acceptable fraction of the maximum component of the
c             sparse vector
c     krow    integer (*)
c             vector of integers
c
      integer aa, defalt,krow(1)
      double precision prat
      integer a,m
      double precision t
      include 'sparse.h'
c
      a = aa
      m = defalt
      t = 0.0d0
   10 if (row(a) .eq. FOOT) goto 20
        if (dabs(val(a)) .gt. t) then
          m = row(a)
          t = dabs(val(a))
        endif
        a = next(a)
      go to 10
   20 if(t.eq.0d0 .or. prat.eq. 1.d0) go to 100
      a=aa
      t=prat*t
   30 if (row(a) .eq. FOOT) go to 100
            if (dabs(val(a)).ge.t .and. krow(row(a)).lt.krow(m)) m=row(a)
        a=next(a)
      go to 30
  100 pividx = m
      end
```

SCOLL

```
      subroutine scoll(xx,s,krow,cid)
c
c Scaling of the subdiagonal elements of a column of a sparse matrix
c in Gaussian elimination to form the corresponding column of the
c lower triangular factor, and update the record of nonzero elements in
c the rows.
c
c on entry
c
c    xx     integer
c           pointer identifying the first subdiagonal element of
c           the column to be scaled
c    s      double precision
c           scaling divisor (pivot of the elimination)
c    krow   integer (*)
c           record of the numbers of nonzero elements in the rows
c           of the matrix
c on return
c
c    krow   updated record of the row counts of nonzero elements
c
      integer x,xx,krow(1), cid
      double precision s
      include 'sparse.h'
      character *60 string
c
      x = xx
   10 if (row(x) .eq. FOOT) return
         val(x) = val(x)/s
              krow(row(x))=krow(row(x))-1
              kdiv=kdiv+1
         x = next(x)
      go to 10
      end
```

RANDUM & URAND

```
c
      double precision function randum(iy)
      integer iy
c
c Generation of uniformly distributed random flints in a fixed range
c symmetric about the origin (compiler parameter).
c
      double precision r,range,urand
      parameter (range = 10.0d0)
   10 r = dnint(range*(2*urand(iy)-1))
      if (r .eq. 0.0d0) go to 10
      randum = r
      end

      real function urand(iy)
c
      integer*2 iy,ia,ic,m2
```

```
real s
data m2/16384/,ia/12869/,ic/6925/,s/3.051758e-5/
iy = iy*ia + ic
if (iy .lt. 0) iy = (iy + m2) + m2
urand = float(iy)*s
return
end
```

YAXPY

```
      subroutine yaxpy(alfa,x,yy,krow,cid)
c
c Modification of a column of a sparse matrix in the Gaussian
c elimination of one variable
c
c on entry
c
c     alfa    double precision
c             element of the column with same row index as the pivot
c     x       packed sparse vector
c             containig the value and row indecies of lower matrices
c     yy      integer
c             pointer to the first element of the column of the matrix
c             below the pivot row
c     krow    integer (*)
c             record of the numbers of nonzero elements in the rows
c             of the sparse matrix
c on return
c
c     krow    updated record of the number of nontrivial elements in
c             the rows of the matrix
c
      integer s , cid
      double precision alfa
      integer y,yy,krow(1), ix
      double precision x(*)
      include 'sparse.h'
      character*60 string
c
      y = yy
c
c check see if alfa is zero
c
      if ( alfa .eq. 0) return
c
c loop through nonzero elements of x
c
      k = 2
   10 ix = x(k+1)
      if ( ix .eq. FOOT) return
   20   if (row(y) .lt. ix) then
           y = next(y)
           go to 20
        endif
c
c insertion of the new elements of y created by x
        if (row(y) .gt. ix) then
          call insert(y, alfa*x(k), ix, cid)
              krow(ix)=krow(ix)+1
              kmul=kmul+1
c
c operation for the components of x and y with coincident indices
        else
          val(y) = val(y) + alfa*x(k)
              kadd=kadd+1
```

```
            kmul=kmul+1
c
c deletion of the components of y annihilated in the process
         if (val(y) .eq. 0.0d0)then
               krow(row(y))=krow(row(y))-1
            call delete(y)
         endif
      endif
      k = k + 2
    go to 10
    end
```

DELETE

```
      subroutine delete(p)
c
c Deletion of one component of a sparse vector
c
c on entry
c
c     p        integer
c              pointer identifying the triplet to be deleted
c
      integer p
      integer t
      include 'sparse.h'
c
      t = next(p)
      val(p) = val(t)
      row(p) = row(t)
      next(p) = next(t)
      end
```

DIMCUBE

```
      integer function dimcube(p)
      integer p
c
c     Dimension of a hypercube containing at least p nodes
c        = ceil(log2(p))
c
      dimcube = ifix(1.44*alog(float(p)) + 0.99)
      return
      end
```

PACK functions

```
      subroutine pack(cid,y,xx,m,s)
c
c Fill Buffer y with cloumn of pointer xx
c
c on entry
c
c    xx     integer
c          pointer identifying the sparse vector
c    m          integer
c               k-th pivot
c on return
c
c    y      full vector ( packed sparse vector )
c
c    s      integer
c          determined size of packed sparse vector including FOOT,
c               returns minimum size of 2.
c
      integer s
      character*100 string
      double precision y(*)
      integer x,xx, cid , m
      include 'sparse.h'

c
c First element is k-th pivot
c
c Places FOOT in first then checks against it.
c This  routine packs the least FOOT.
c
      x=xx
      s = 1
      y(s) = m
   10 y(s+1) = val(x)
      y(s+2) = row(x)
      s = s + 2
      if(row(x).eq.FOOT) return
      x=next(x)
      go to 10
      end
```

```
      subroutine dpack(cid,y,xx,m,s)
c
c Fill Buffer y with cloumn of pointer xx
c
c on entry
c
c     xx      integer
c             pointer identifying the sparse vector
c     m           double percision
c                 k-th pivot
c on return
c
c     y       full vector ( packed sparse vector )
c
c     s       integer
c             determined size of packed sparse vector including FOOT,
c                 returns minimum size of 2.
c
      integer s
      character*100 string
      double precision y(*), m
      integer x,xx, cid
      include 'sparse.h'

c
c First element contains
c
c Places FOOT in first then checks against it.
c This  routine packs the least FOOT.
c
      x=xx
      s = 1
      y(s) = m
   10 y(s+1) = val(x)
      y(s+2) = row(x)
      s = s + 2
      if(row(x).eq.FOOT) return
      x=next(x)
      go to 10
      end
```

Broadcast routines

B.1. GOP

```
      SUBROUTINE GOP (CI, TYPE, X, N, OP, ROOT, DIM, WORK)
      INTEGER CI, TYPE, N, ROOT, DIM
      CHARACTER*1 OP
      DOUBLE PRECISION X(N), WORK(N)
c
c Global vector commutative operation using spanning tree.
c
c All participating processes must have the same process id (PID).
c
c Input..
c
c   CI       channel number (previously opened).
c   TYPE     message type.  Must be the same for all participating
c            processes.  There must be no other messages of this type
c            in the system.
c   X        the input vector to be used in the operation.
c   N        the length of the vector.
c   OP       '+' sum
c            '*' product
c            'M' maximum
c            'm' minimum
c   ROOT     Node id of root process (which will get the final message).
c            (if root is negative, then the smallest node number in the active
c            subcube acts as root and then forwards the message to the root,
c            which should be the host, or, in release 3+, a subcube.)
c   DIM      the size of the subcube participating.
c
c Output..
c
c   X        for the root process, X contains the desired result.
c            for all other processes, X contains the partial result
c            for their subtrees.
c
c Workspace
c
c   WORK     used to receive other contributions.
c
c Errors Conditions
c
c      If called by a nonparticipating node, an error message is
c      syslogged and then the subroutine exits.
c
c      If a message longer than N elements is received, only the first N
c      elements are saved, an error message is syslogged,
c      and then the computation continues with the truncated results.
c
c      If a message shorter than N elements is received, then an error
c      message is syslogged and the computation continues.
c
c Calls: MYNODE, MYPID, RECVW, SENDW, SYSLOG, XOR
c
      INTEGER BIT, BYTES, CNT, DIFF, DPSIZE, I, IGNORE, ME, MYNODE,
```

```
     * MYPID, P, PARENT, PID, TROOT, XOR
      PARAMETER (DPSIZE = 8)
c
      ME = MYNODE()
      P = 2**DIM
c
c     Find temporary root (either the real root, or the lowest
c     numbered node in the active subcube--found by zeroing the
c     DIM lowest bits in mynode).
c
      TROOT = MAX0((ME/P)*P, ROOT)
c
      PID = MYPID()
      DIFF = XOR(ME,TROOT)
      IF (DIFF .GE. P) THEN
        CALL SYSLOG(MYPID(),'GOP: CALLED BY NON PARTICIPANT')
        RETURN
      ENDIF
c
c     Accumulate contributions from children, if any
c
      BIT = 1
      IF (DIFF .EQ. 0) DIFF = P
      BYTES = DPSIZE*N
   10 IF (XOR(BIT,DIFF) .LT. DIFF) GO TO 30
        CALL RECVW(CI,TYPE,WORK,BYTES,CNT,IGNORE,PID)
        IF (CNT .GT. BYTES) CALL SYSLOG(TYPE,'GOP: LONG MESSAGE')
        IF (CNT .LT. BYTES) CALL SYSLOG(TYPE,'GOP: SHORT MESSAGE')
        DO 20 I = 1, N
          IF (OP .EQ. '+') X(I) = X(I) + WORK(I)
          IF (OP .EQ. '*') X(I) = X(I) * WORK(I)
          IF (OP .EQ. 'M') X(I) = DMAX1(X(I),WORK(I))
          IF (OP .EQ. 'm') X(I) = DMIN1(X(I),WORK(I))
   20   CONTINUE
        BIT = 2*BIT
      GO TO 10
c
c     Pass result back to parent, if any
c
   30 CONTINUE
      IF (ME .NE. ROOT) THEN
        PARENT = XOR(ME, BIT)
        IF (ME .EQ. TROOT) PARENT = ROOT
        CALL SENDW(CI,TYPE,X,BYTES,PARENT,PID)
      ENDIF
      RETURN
      END
```

B.2. IGOP

```
      SUBROUTINE IGOP (CI, TYPE, X, N, OP, ROOT, DIM, WORK)
      INTEGER CI, TYPE, N, ROOT, DIM
      CHARACTER*1 OP
      INTEGER X(N), WORK(N)
c
```

```
c Global vector commutative operation using spanning tree.
c
c All participating processes must have the same process id (PID)
c
c Input..
c
c   CI       channel number (previously opened).
c   TYPE      message type. Must be the same for all participating
c            processes. There must be no other messages of this type
c            in the system.
c   X        the input vector to be used in the operation.
c   N        the length of the vector.
c   OP        '+' sum
c            '*' product
c            'M' maximum
c            'm' minimum
c   ROOT      Node id of root process (which will get the final message).
c            (if -32768, then the smallest node number in the active
c            subcube acts as root and then forwards the message to the host)
c   DIM       the size of the subcube participating.
c
c Output..
c
c   X        for the root process, X contains the desired result.
c            for all other processes, X contains the partial result
c            for their subtrees.
c
c Workspace
c
c   WORK      used to receive other contributions.
c
c Errors Conditions
c
c       If called by a nonparticipating node, an error message is
c       syslogged and then the subroutine exits.
c
c       If a message longer than N elements is received, only the first N
c       elements are saved, an error message is syslogged,
c       and then the computation continues with the truncated results.
c
c       If a message shorter than N elements is received, then an error
c       message is syslogged and the computation continues.
c
c Calls: MYNODE, MYPID, RECVW, SENDW, SYSLOG, XOR
c
      INTEGER BIT, BYTES, CNT, DIFF, ISIZE, I, IGNORE, ME, MYNODE,
     *   MYPID, P, PARENT, PID, TROOT, XOR
      PARAMETER (ISIZE = 8)
c
      ME = MYNODE()
      P = 2**DIM
c
c     Find temporary root (either the real root, or the lowest
c     numbered node in the active subcube--found by zeroing the
c     DIM lowest bits in mynode).
c
      TROOT = MAX0((ME/P)*P, ROOT)
```

```
c
      PID = MYPID()
      DIFF = XOR(ME,TROOT)
      IF (DIFF .GE. P) THEN
        CALL SYSLOG(MYPID(),'GOP: CALLED BY NON PARTICIPANT')
        RETURN
      ENDIF
c
c     Accumulate contributions from children, if any
c
      BIT = P/2
      BYTES = ISIZE*N
    5 IF (BIT .LE. DIFF) GO TO 20
        CALL RECVW(CI,TYPE,WORK,BYTES,CNT,IGNORE,PID)
        IF (CNT .GT. BYTES) CALL SYSLOG(PID,'GOP: LONG MESSAGE')
        IF (CNT .LT. BYTES) CALL SYSLOG(PID,'GOP: SHORT MESSAGE')
        DO 10 I = 1, N
          IF (OP .EQ. '+') X(I) = X(I) + WORK(I)
          IF (OP .EQ. '*') X(I) = X(I) * WORK(I)
          IF (OP .EQ. 'M') X(I) = DMAX1(X(I),WORK(I))
          IF (OP .EQ. 'm') X(I) = DMIN1(X(I),WORK(I))
   10   CONTINUE
        BIT = BIT/2
      GO TO 5
c
c     Pass result back to parent
c
   20 CONTINUE
      IF (BIT .NE. 0) THEN
        PARENT = XOR(ME, BIT)
        CALL SENDW(CI,TYPE,X,BYTES,PARENT,PID)
      ELSE
        IF (ROOT .LT. 0) CALL SENDW(CI,TYPE,X,BYTES,-32768,PID)
      ENDIF
      RETURN
      END
```

B.3. GRECVW

```
      SUBROUTINE GRECVW(CI, TYPE, BUF, LEN, CNT, DIM)
      INTEGER CI, TYPE, BUF(*), LEN, CNT, DIM
c
c Global send participant.  Receives message from unknown source and
c     sends it on to some neighbors.
c
c All participating processes must have the same process id (PID).
c
c Input..
c
c   CI        channel number (previously opened).
c   TYPE      message type.  Must be the same for all participating
c             processes.  There must be no other messages of this type
c             in the system.
c   LEN       the length of BUF in BYTES.
c   DIM       the dimension of the subcube participating in the send.
```

```
c
c Output..
c
c   BUF      the message (which may actually be any type).
c   CNT      the length (in BYTES) of the message received.
c
c Error Conditions
c
c   If a message longer than LEN bytes is received then only
c   LEN bytes will be stored in BUF and the rest of the
c   message will be lost. In this case an error message
c   will be sent to syslog but the remnants of the message
c   will be sent on.
c
c   NOTE: only those nodes which will participate in the
c      send can call GRECVW. Any other node which calls it
c      will never return.
c
c Calls: MYNODE, MYPID, RECVW, SENDW, SYSLOG, XOR
c
      INTEGER BIT, I, LENOUT, ME, MYNODE, MYPID, NODE, P, PID,
     *   PIDIN, XOR
c
      P = 2**DIM
      ME = MYNODE()
      PID = MYPID()
      CALL RECVW(CI, TYPE, BUF, LEN, CNT, NODE, PIDIN)
c
      LENOUT = CNT
      IF (CNT .GT. LEN) THEN
         CALL SYSLOG(PID,'GRECVW: MESSAGE TRUNCATED')
         LENOUT = LEN
      ENDIF
c
      BIT = 2*XOR(ME,NODE)
c
c Check to see if received from host.
c
      IF (IABS(NODE) .GT. 128) BIT = 1
c
      DO 10 I = 1, DIM
         IF (BIT .EQ. P) RETURN
         NODE = XOR(ME,BIT)
         CALL SENDW(CI, TYPE, BUF, LENOUT, NODE, PID)
         BIT = 2*BIT
   10 CONTINUE
      END
```

B.4. GSENDW

```
      SUBROUTINE GSENDW(CI, TYPE, BUF, LEN, DIM)
      INTEGER CI, TYPE, BUF(*), LEN, DIM
c
c Global send of data. Other participants call grecvw.
c
```

```
c All participating processes must have the same process id (PID).
c
c Input
c
c   CI        channel number (previously opened).
c   TYPE      message type.  Must be the same for all participating
c             processes.  There must be no other messages of this type
c             in the system.
c   BUF       the message buffer (which may actually be any type)
c   LEN       the length of the buffer in BYTES
c   DIM       the dimension of the subcube
c
c Calls:  MYNODE, MYPID. SENDW, XOR
c
      INTEGER BIT, I, ME, NODE, MYNODE, MYPID, PID, XOR
c
      ME = MYNODE()
      PID = MYPID()
      BIT = 1
c
      DO 10 I = 1, DIM
        NODE = XOR(ME,BIT)
        CALL SENDW(CI, TYPE, BUF, LEN, NODE, PID)
        BIT = 2*BIT
   10 CONTINUE
      RETURN
      END
```

SWAP

```
      subroutine swap(xx,kk,mm)
c
c Exchange of the k-th and m-th components of a sparse vector
c
c on entry
c
c     xx      integer
c             pointer to the vector where the exchange takes place
c     kk      integer
c             index of one of the components to be exchanged
c     mm      integer
c             index of the other component to be exchanged
c
      integer x,xx,k,m,kk,mm
      integer kp,mp
      include 'sparse.h'
c
      k=min0(kk,mm)
      m=max0(kk,mm)
c
c find kp and mp so that row(kp) >= k and row(mp) >= m
      x = xx
      kp = x
   10 if (row(kp) .ge. k) go to 20
        kp = next(kp)
        go to 10
   20 mp = kp
   30 if (row(mp) .ge. m) go to 40
        mp = next(mp)
        go to 30
   40 continue
c
c four cases to consider
      if (row(kp) .gt. k .and. row(mp) .gt. m) then
c both components are zero.  do nothing
c
c one component is nonzero.  cut and paste
      elseif (row(kp) .eq. k .and. row(mp) .gt. m) then
        call candp(kp,mp,m)
      elseif (row(kp) .gt. k .and. row(mp) .eq. m) then
        call candp(mp,kp,k)
      else
c
c both components are nontrivial.  swap the values
        call swreal(val(kp), val(mp))
      endif
      end
```

SWREAL

```
      subroutine swreal(alfa,beta)
c
c Swap two variables
c
c on entry
c
c     alfa    double precision
c             variable to be exchanged with beta
c     beta    double precision
c             variable to be exchanged with alfa
c on return
c
c     alfa    the value entered as beta
c     beta    the value entered as alfa
c
      double precision alfa,beta,t
      t = alfa
      alfa = beta
      beta = t
      end
```

XOR function

```
      INTEGER FUNCTION XOR(M,N)
      INTEGER M,N
c
c     exclusive or
c
c     Builtin on UNIX f77.
c
c     For Intel FTN286 use:
c     XOR = M.NEQV.N
c
c     For R/M Fortran use:
      XOR = IEOR(M,N)
c
      RETURN
      END
```

APPENDIX B

```
      SUBROUTINE KJI(A,LDA,N)
c
c     FORM KJI -SAXPY
c
      REAL A(LDA,N)
      DO 40 K = 1, N-1
         DO 10 I = K+1, N
               A(I,K) = -A(I,K) / A(K,K)
10    CONTINUE
         DO 30 J = K+1, N
               DO 20 I = K+1 ,N
                     A(I,J) = A(I,J) + A(I,K) * A(K,J)
20             CONTINUE
30       CONTINUE
40    CONTINUE
      RETURN
      END
```

Form KJI

```
      SUBROUTINE JKI(A,LDA,N)
c
c     FORM JKI -GAXPY
c
      REAL A(LDA,N)
      DO 40 J = 1, N
         DO 20 K = 1 , J-1
               DO 20 I =  K+1, N
                     A(I,J) = A(I,J) + A(I,K) * A(K,J)
10             CONTINUE
20       CONTINUE
         DO 30 I = J+1, N
               A(I,J) = -A(I,J)/A(J,J)
30       CONTINUE
40    CONTINUE
      RETURN
      END
```

Form JKI

```
      SUBROUTINE IJK(A,LDA,N)
c
c     FORM IJK -DOT
c
      REAL A(LDA,N)
      DO 50 I = 1, N
           DO 20 J = 2 , I
                   A(I,J-1) = -A(I,J) / A(J-1,J-1)
                   DO 10 I =  K+1, N
                           A(I,J) = -A(I,J) + A(I,K) * A(K,J)
10                 CONTINUE
20         CONTINUE
           DO 40 J = I+1, N
                   DO 30 K = 1, I-1
                           A(I,J) = -A(I,J) + A(I,K) * A(K,J)
30                 CONTINUE
40         CONTINUE
40 CONTINUE
      RETURN
      END
```

Form IJK

```
      SUBROUTINE JKIPVT(A,LDA,N)
c
c     FORM JKI -GAXPY
c     WITH PIVOTING
c
      REAL A(LDA,N), T
      DO 60 J = 1, N
           DO 20 K = 1, J-1
                   DO 10 I = K+1, N
                           A(I,J) = A(I,J) + A(I,K) * A(K,I)
10                 CONTINUE
20         CONTINUE
c
c     PIVOT SEARCH
c
      T = ABS(A(J,J))
      L = J
```

```
      DO 30 I = J + 1, N
         IF (ABS(A(I,J)) .GT. T) THEN
                T = ABS(A(I,J))
                L = I
         ENDIF
30 CONTINUE
C
C        INTERCHANGE ROWS
C
         DO 40 I = 1, N
                T = A(J,I)
                A(J,I) = A(L,I)
                A(L,I) = T
40       CONTINUE
C
         DO 50 I = J+1, N
                A(I,J) = -A(I,J)/A(J,J)
50       CONTINUE
60 CONTINUE
      RETURN
      END
```

Form JKI (with pivoting)

Bibliography Note

The author was born on July. 13, 1960 in Tehran, Iran. He moved to Ellesmere England in 1975 graduated from Ellesmere college in 1978 then moved to Columbus Ohio in Jan 1979 and received a B.A in Computer Science , Mathematics from Otterbein College in July 1984. He began study at Oregon Graduate Center in Sept. 1984 and received M.S. in Computer Science in 1987.