Computing System in Engineering Vol. 2. No. 2/3 pp. 197-201, 1991
Printed in Great Britain.

0956-0521/91 $3.00 + 0.00

A PARALLEL-VECTOR EQUATION SOLVER FOR UNSYMMETRIC MATRICES ON SUPERCOMPUTERS

J.Qin, D.T. Nguyen, Civil Engineering Department, Old Dominion University, Norfolk, VA 23529, USA
C.E. Gray, Jr, Facilities Engineering Division, NASA Langley Research Center, Hampton, VA 23665, USA
And C. Mei, Mechanical Engineering and Mechanics Department, Old Dominion University, Norfolk, VA 23529, USA

(Received 18 December 1990)

PARALLEL-VECTOR UNSYMMETRIC EQUATION SOLVER

$$A \cdot x = b \tag{1}$$

$$A = L \cdot U \tag{2}$$

$$U(r,i) = a(r,i) - \sum_{k=1}^{r-1} L(r,k) \cdot U(k,i) \qquad (i = r, \ldots\ldots, n)$$

$$L(i,r) = \frac{\left(a(i,r) - \sum_{k=1}^{r-1} L(i,k) \cdot U(k,r) \right)}{U(r,r)} \qquad (i = r+1, \ldots\ldots, n) \tag{3}$$

$$L \cdot y = b \tag{4}$$

$$y(i) = b(i) - \sum_{k=1}^{i-1} L(i,k) \cdot y(k) \qquad (i = 1, \ldots, n) \tag{5}$$

Table 1. Basic algorithm for decomposition

```
      ( For I = 1,2,3,....,n )
c****ITOP is the row number of the top element of the I-th column
      Do 1 K = ITOP, I-1
c**** do-loop 2 is to update the I-th row of U due to the contribution
c**** of the K-th row.
       Do 2 J=I,K+length of the K-th row.
        a(I,J)=a(I,J)-a(I,K)*a(K,J)
2      continue
c**** do-loop 3 is to update the I-th column of L due to the
c**** contribution of the K-th column.
       Do 3 J1=I+1,K+length of the K-th column
        a(J1,I)=a(J1,I)-a(K,I)*a(J1,K)
3      continue
1     continue                    U       L

      Do 4 J1=I+1,I+length of the I-th column
       a(J1,I)=a(J1,I)/a(I,I)
4     continue
      (End for loop I)
```

Calculate [U]

Calculate [L]

Calculate [L]

Note: the first row of u_{ij} always= the first row of Aij
$U_{11}=A_{11}$
$U_{12}=A_{12}$
$U_{13}=A_{13}$

and to solve

$$U \cdot x = y \tag{6}$$

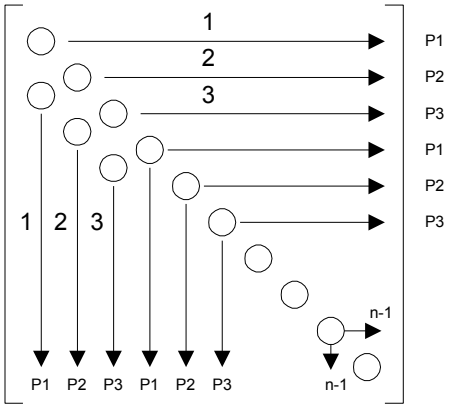
for *x*, with

$$x(i) = \frac{y(i) - \sum_{k=i+1}^{n} U(i,k) \cdot x(k)}{U(i,i)} \qquad (i = n, .., 1) \tag{7}$$

For $I = 1,2,3,\ldots,n$

Step a. Find the *I* th row of *U*.

Step b. Find the *I* th column of *L*.



A=[{row 1}, {row 2}, {row 3},…, {row n},…., {column 1}, {column 2}, {column 3},…,{column n}]

Fig. 1. Storage scheme for matrix A in a one-dimensional array.

Detailed Derivations For the [L] and [U] Matrices

In order to better understand the derived formula shown in Eq. (3), let's try to compute the factorized [L] and [U] matrices from the following 3x3 unsymmetrical matrix [A] (assuming to be a full matrix to simplify the discussion)

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{101}$$

The above unsymmetrical matrix A can be factorized as indicated in Eq. (2), or in the long form.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix} \cdot \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix} \tag{102}$$

The 9 unknowns (according to a special ordering u_{11}, u_{12}, u_{13}; then l_{21}, l_{31}; then u_{22}, u_{23}; then l_{32}; and finally u_{33}) from Eq. (102) can be found by simultaneously solving the following system of equations.

$$
\begin{aligned}
a_{11} &= u_{11} \\
a_{12} &= u_{12} \\
a_{13} &= u_{13} \\
a_{21} &= l_{21} \cdot u_{11} \\
a_{31} &= l_{31} \cdot u_{11} \\
a_{22} &= l_{21} \cdot u_{12} + u_{22} \\
a_{23} &= l_{21} \cdot u_{13} + u_{23} \\
a_{32} &= l_{31} \cdot u_{12} + l_{32} \cdot u_{22} \\
a_{33} &= l_{31} \cdot u_{13} + l_{32} \cdot u_{23} + u_{33}
\end{aligned}
\qquad (103)
$$

Thus, from Eq. (103), one obtains

$$
\begin{aligned}
u_{11} &= a_{11} \\
u_{12} &= a_{12} \\
u_{13} &= a_{13}
\end{aligned}
\quad \Big\} \text{ Factorized } 1^{st} \text{ row}
$$

$$
\begin{aligned}
l_{21} &= \frac{a_{21}}{u_{11}} \\
l_{31} &= \frac{a_{31}}{u_{11}}
\end{aligned}
\quad \Big\} \text{ Factorized } 1^{st} \text{ col. } \rightarrow \text{ need col.1 of U matrix}
$$

$$
\begin{aligned}
u_{22} &= a_{22} - l_{21} \cdot u_{12} \\
u_{23} &= a_{23} - l_{21} \cdot u_{13}
\end{aligned}
\quad \Big\} \text{ Factorized } 2^{nd} \text{ row } \rightarrow \text{ need multipliers } l_{21} \text{ (of row 2 of L matrix) \& needs earlier row(s)}
$$

$$
l_{32} = \frac{a_{32} - \left(l_{31} \cdot u_{12}\right)}{u_{22}}
\quad \Big\} \text{ Factorized } 2^{nd} \text{ col } \rightarrow \text{ need col. 2 of U matrix \& needs earlier column(s)}
$$

$$
u_{33} = a_{33} - \left(l_{31} \cdot u_{13} + l_{32} \cdot u_{23}\right)
\quad \Big\} \text{ Factorized } 3^{rd} \text{ row } \rightarrow \text{ need multipliers } l_{3x} \text{ (of row 3 of L matrix)\& needs earlier row(s)}
\qquad (104)
$$

It can be seen clearly that the 9 unknowns shown in Eq. (104) can also be obtained by directly using Eq.(3)

The ordering appeared in Eq. (104) suggests that the factorized matrix [L] and [U] can be found in the following systematic pattern:

Step 1: The 1st row of [U] can be solved (Ex: u_{11}, u_{12}, u_{13}).
Step 2: The 1st column of [L] can be solved (Ex: l_{21}, l_{31}).
Step 3: The 2nd row of [U] can be solved.
Step 4: The 2nd column of [L] can be solved.
Step 5: The 3rd row of [U] can be solved.
*
*
*
etc…..

For the case r=8, and i=9, Eq.(3) becomes

$$\left.\begin{aligned} u_{8,9} &= a_{8,9} - \left(l_{8,1}\cdot u_{1,9} + l_{8,2}\cdot u_{2,9} + \ldots.. + l_{8,7}\cdot u_{7,9}\right) \\ l_{9,8} &= \frac{a_{9,8} - \left(l_{9,1}\cdot u_{1,8} + l_{9,2}\cdot u_{2,8} + \ldots. + l_{9,7}\cdot u_{7,8}\right)}{u_{8,8}} \end{aligned}\right\} \qquad (105)$$

observing Eq.(105), one can see that to factorize the term $u_{8,9}$ of the upper triangular [U], one only needs to know the factorized row 8 of [L] and column 9 of [U].

similarly, to factorize the term $l_{9,8}$ of the lower triangular matrix [L], one only needs to know the factorized row 9 of [L] and column 8 of [U].

Also:

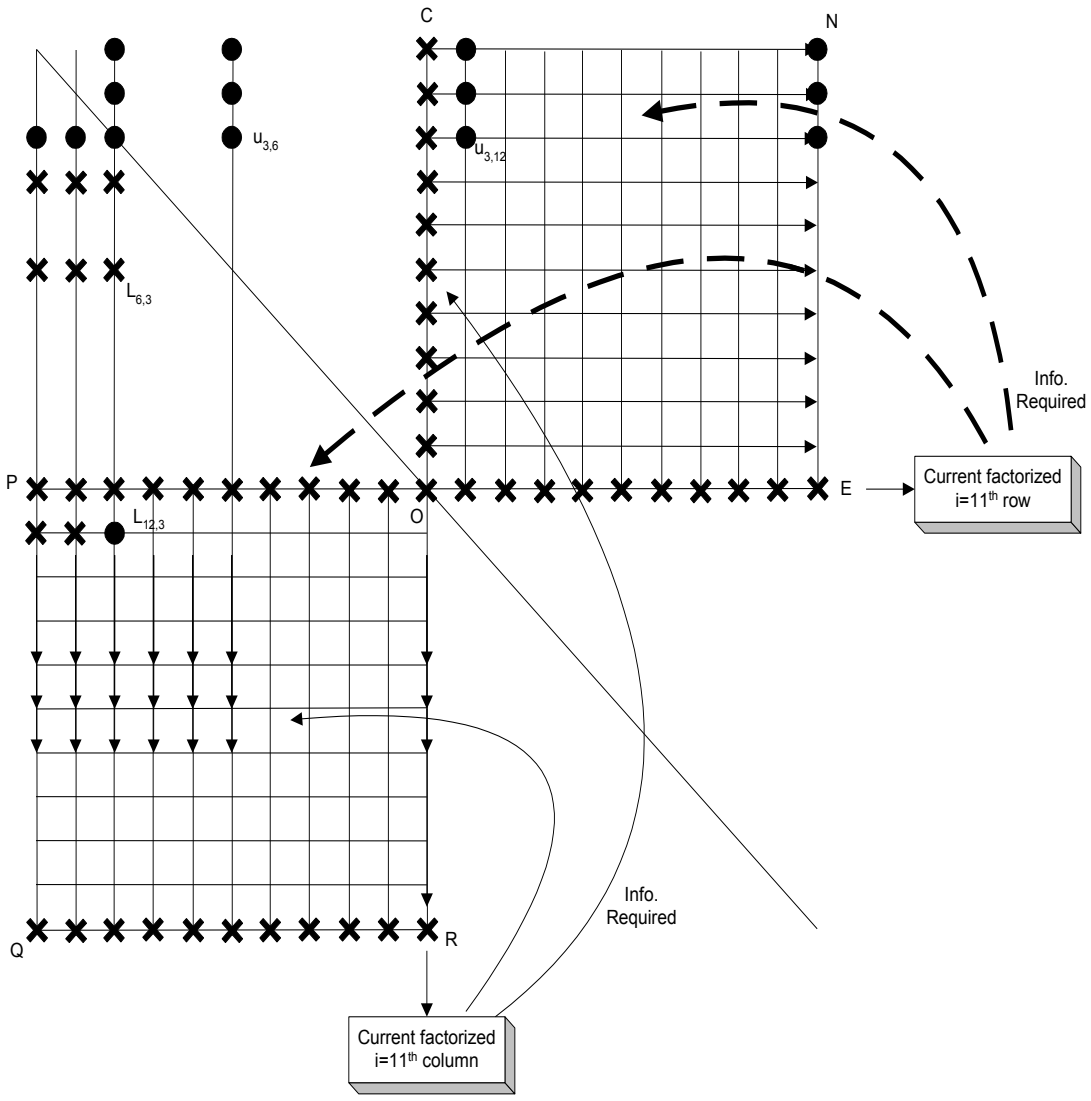
$$u_{3,6} = a_{3,6} - \left(L_{3,1}\cdot u_{1,6} + L_{3,2}\cdot u_{2,6}\right)$$

$$u_{3,12} = a_{3,12} - \left(L_{3,1}\cdot u_{1,12} + L_{3,2}\cdot u_{2,12}\right)$$

$$L_{6,3} = \frac{a_{6,3} - \left(L_{6,1}\cdot u_{1,3} + L_{6,2}\cdot u_{2,3}\right)}{u_{3,3}}$$

$$L_{12,3} = a_{12,3} - \left(L_{12,1}\cdot u_{1,3} + L_{12,2}\cdot u_{2,3}\right)$$

[A] =



If the matrix is full (& unsymmetrix)

- To factorize the entire i^{th} row (line OE),
  We need - multipliers, row ith of [L] (or line OP)
  - earlier rows of [U] (rectangular area OCNE, right above OE)
- To factorize the entire i^{th} column (line OR),
  We need - multipliers, column i^{th} of [U] (or line OC)
  - earlier columns of [L] (rectangular area OPQR, left of OR)

If the matrix is banded (& unsymmetric)

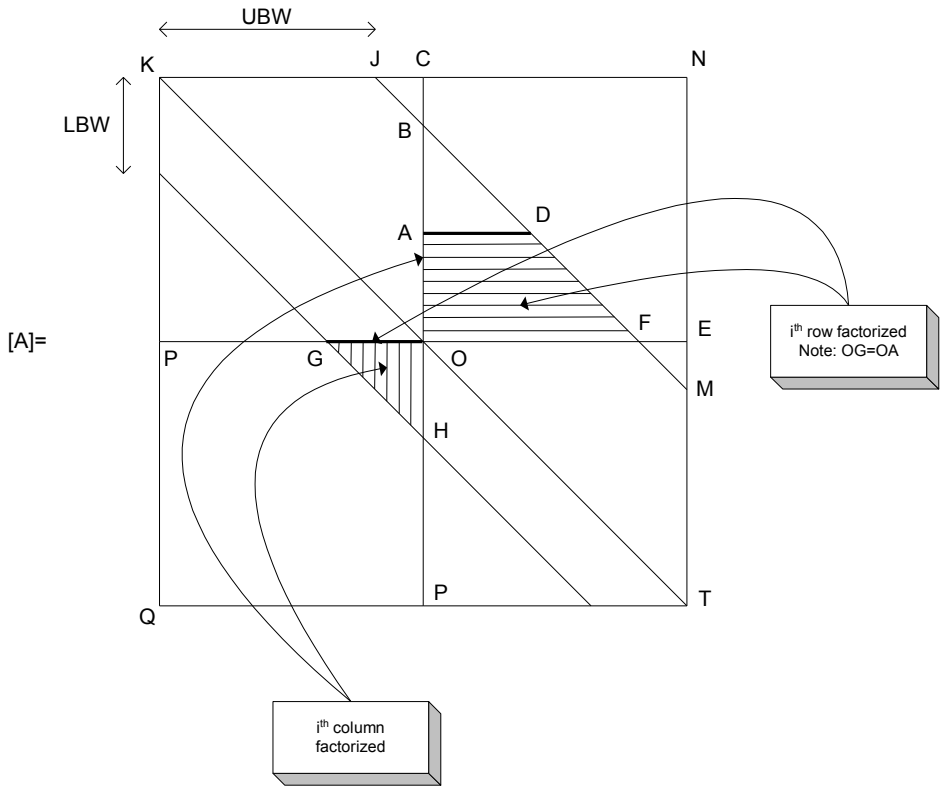


Table 2. Vector algorithm for factorization

```
      ( For I = 1,2,3,....,n )
      Do 1 K = ITOP, I-1, 6
       CDIR$ IVDEP
       Do 2 J=I,K+length of the K-th row.
        a(I,J)=a(I,J)-a(I,K)*a(K,J)-a(I,K+1)*a(K+1,J)
     +         -a(I,K+2)*a(K+2,J)-a(I,K+3)*a(K+3,J)
     +         -a(I,K+4)*a(K+4,J)-a(I,K+5)*a(K+5,J)
2     continue                          U          L          U          L
       CDIR$ IVDEP
       Do 3 J1=I+1,K+length of the K-th column
        a(J1,I)=a(J1,I)-a(K,I)*a(J1,K)-a(K+1,I)*a(J1,K+1)
     +          -a(K+2,I)*a(J1,K+2)-a(K+3,I)*a(J1,K+3)
     +          -a(K+4,I)*a(J1,K+4)-a(K+5,I)*a(J1,K+5)
3     continue
1    continue
      Do 4 J1=I+1,I+length of the I-th column
       a(J1,I)=a(J1,I)/a(I,I)
4    continue
     (End for loop I)
```

In this paper, the artificial coefficient matrix A is automatically generated as

$$a(i,j) = \frac{1.0}{j} \qquad (for j > i)$$

$$a(i,j) = \frac{1}{j} + \frac{1}{i+j} \qquad (for j < i)$$

$$a(i,i) = i \qquad (for 1 < i < n)$$

$$b(i) = 1.0 \qquad (for 1 < i < n)$$

Table 3. Parallel algorithm for decomposition

```
      Presched Do 100 I=1, n

      Do 1 K=ITOP, I-NP, 6
      CDIR$ IVDEP
       Do 2 J=I,K+length of the K-th row
         a(I,J)=a(I,J)-a(I,K)*a(K,J)-a(I,K+1)*a(K+1,J)
     +          -a(I,K+2)*a(K+2,J)-a(I,K+3)*a(K+3,J)
     +          -a(I,K+4)*a(K+4,J)-a(I,K+5)*a(K+5,J)
2     continue
      CDIR$ IVDEP
       Do 3 J1=I+1,K+length of the K-th column.
         a(J1,I)=a(J1,I)-a(K,I)*a(J1,K)-a(K+1,I)*a(J1,K+1)
     +           -a(K+2,I)*a(J1,K+2)-a(K+3,I)*a(J1,K+3)
     +           -a(K+4,I)*a(J1,K+4)-a(K+5,I)*a(J1,K+5)
3     continue
1    continue
      Do 10 K=I-NP+1,I-1
      Copy Asyn(K) into KK
      Do 20 J=I,K+length of the K-th row.
        a(I,J)=a(I,J)-a(I,K)*a(K,J)
20    continue
      Do 30 J1=I+1,K+length of the K-th column
        a(J1,I)=a(J1,I)-a(K,I)*a(J1,K)
30    continue
10   continue
      Do 4 J1=I+1,K+length of the I-th column
        a(J1,I)=a(J1,I)/a(I,I)
4     continue
      Produce Asyn(I)=1.0
100   End Presched Do
```

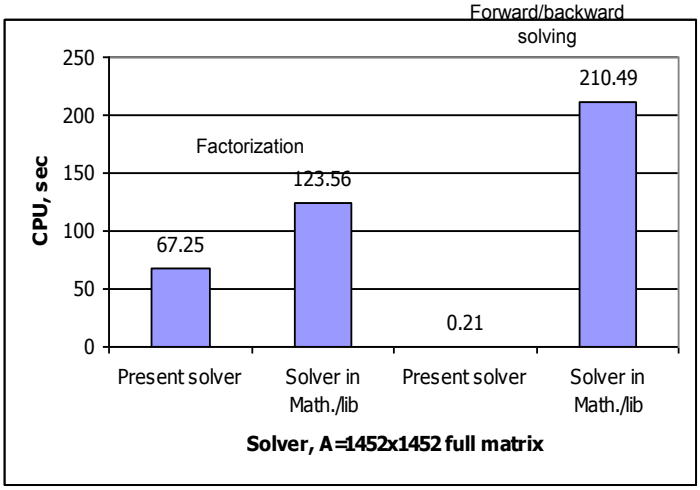


Fig. 2. Different solvers on CONVEX C220

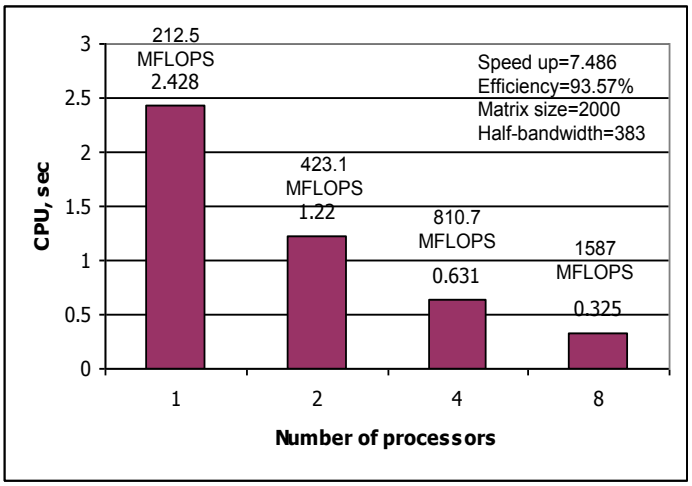


Fig. 3. Factorization of A=LU on CRAY Y-MP

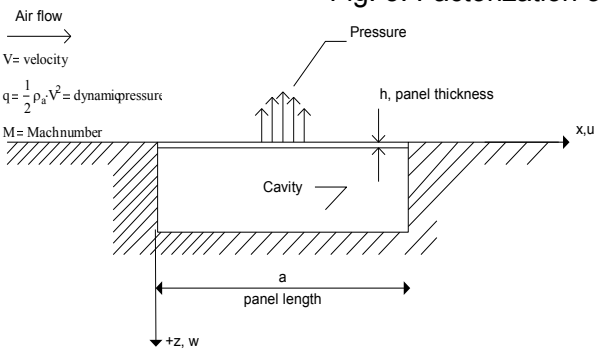


Fig. 4. Finite Element panel flutter analysis

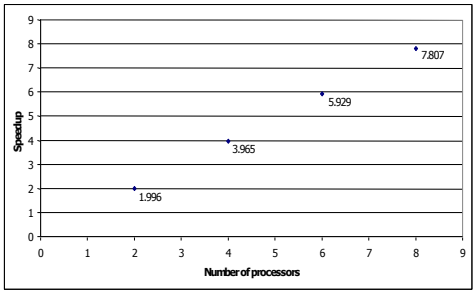


Fig.6. Speedup for panel flutter analysis on CRAY Y-MP

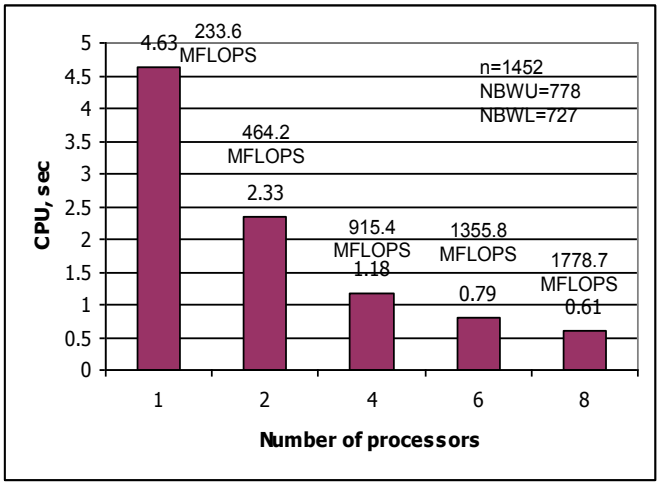


Fig. 5. Panel flutter analysis on CRAY Y-MP (CPU time for factorization and forward/backward elimination)

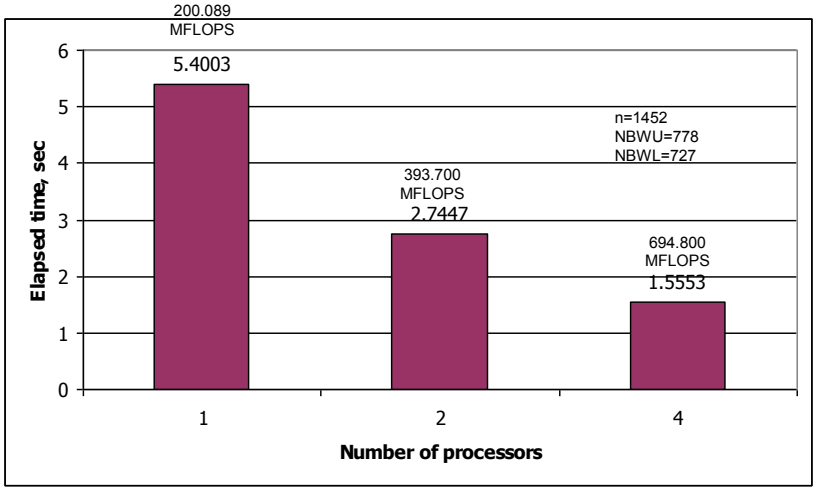
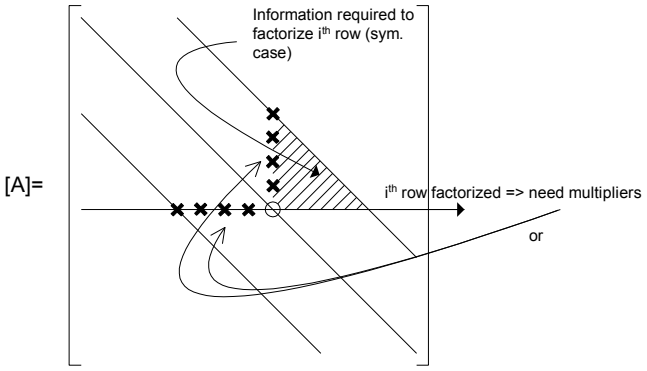


Figure 7 : Elapsed time for factorization on Cray-2 (Voyager)

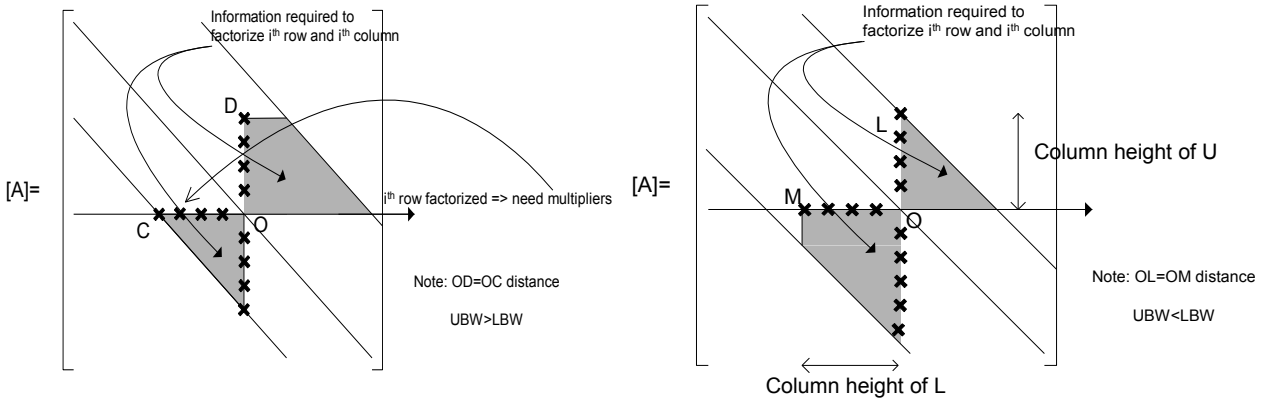
Very detailed, important notes (For efficient code implementation)

Note#1

Symmetric case ➔



Unsym. Case 2



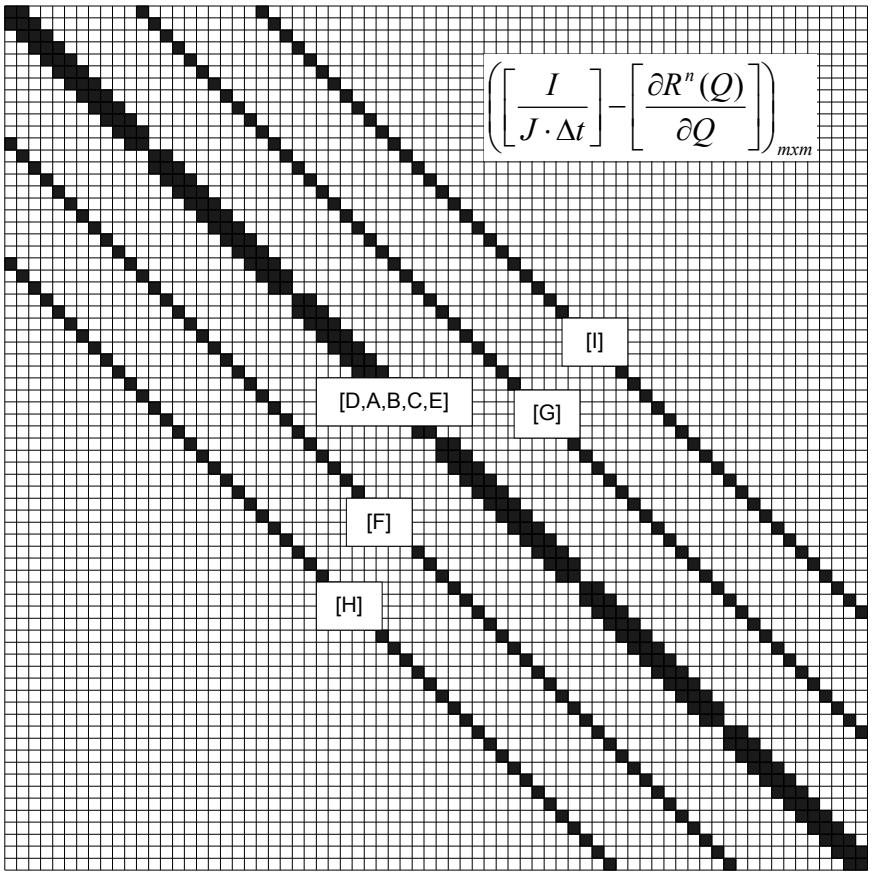
Note#2 – In practice, we also have considered variable column heights for [U] and [L] for more efficiency!!

Coefficient Matrix in CFD Application:

Equations=25,160

Upper Bandwidth=Lower Bandwidth=425

CFD Vector Solver	Present solver	Voyager (Cray-2)
≈ 65 sec	≈ 53 sec	1 processor
N/A	≈ 14 sec	4 processors



$$* \left\{{}^{n}\Delta Q_{mx1}\right\} = \left\{R^n(Q)_{mx1}\right\}$$

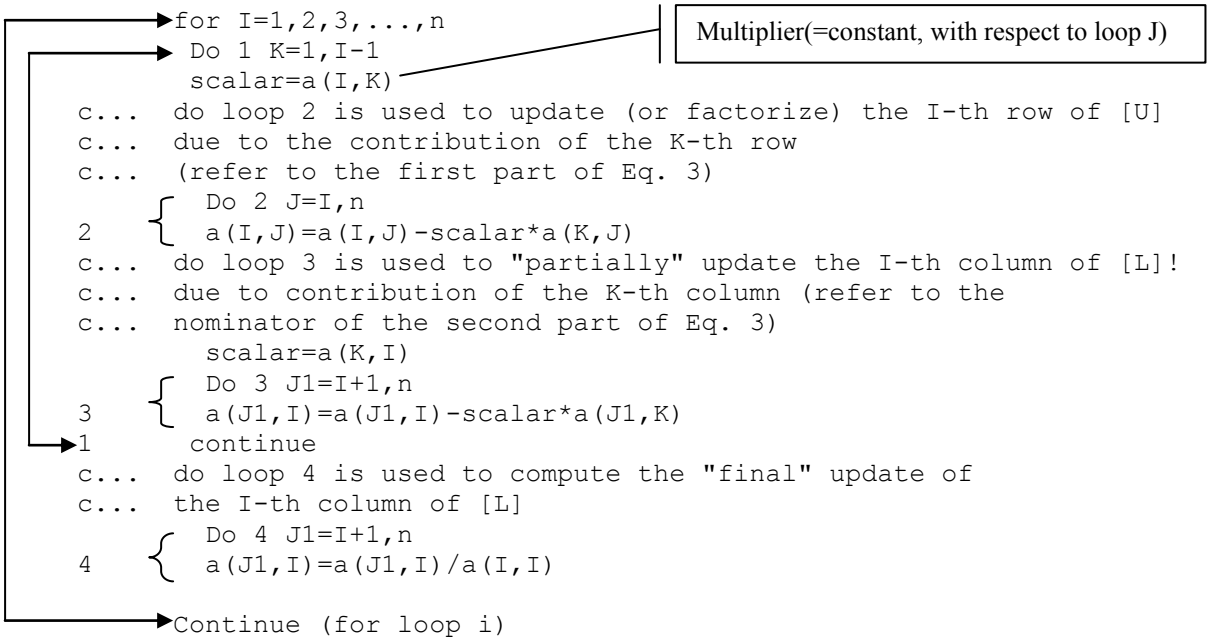


Table 11: Basic Algorithm For [L]&[U] Factorization (full-matrix is assumed)

In order to better understand the basic algorithm shown in Table 11 for factorization of a full unsymmetrical matrix, a 3x3 matrix [A] given in the previous section will be used to verify Table 11

For i=1, then (please refer to Table 11)

- loop 1 will be skipped
- from loop 4
  $a(2,1) = a(2,1)/a(1,1) \equiv l_{21}$ (refer to Eq. 104)
  $a(3,1) = a(3,1)/a(1,1) \equiv l_{31}$ (refer to Eq. 104)
- notice:
  The first row of [U] (such as u_{11}, u_{12} and u_{13}) are not required to calculate, because they are the same as the original matrix [A] ($u_{11} = a_{11}$, $u_{12} = a_{12}$ and $u_{13} = a_{13}$)

For i=2, then

- from loop 2

  a(2,2) = a(2,2)-a(2,1)(=*scalar*)*a(1,2) ≡ U_{22} (refer to Eq. 104)

  a(2,3) = a(2,3)-a(2,1)*a(1,3) ≡ U_{23} (refer to Eq. 104)

- from loop 3

  a(3,2) = a(3,2)-a(1,2)(=*scalar*)*a(3,1) ≡ Partial solution for $l_{3,2}$

- from loop 4

  a(3,2) = a(3,2)/a(2,2) ≡ l_{32} (refer to Eq. 104)

For i=3, then

- from loop 2 (with K=1)

  a(3,3) = a(3,3)-a(3,1)*a(1,3) ≡ Partial solution for $u_{3,3}$

- loop 3 will be skipped
- loop 4 will be skipped
- from loop 2 (with K=2)

  a(3,3) = a(3,3)-a(3,2)*a(2,3) ≡ $u_{3,3}$ (refer to Eq. 104)
- loop 3 will be skipped
- loop 4 will be skipped

Comments on Table 11

(a) The operations in the innermost loops 2 and 3 are "saxpy" operations (a vector + a scalar * another vector), thus these operations can be done quite fast on vector computers (such as CRAY Y-MP, CRAY-C90 computers)

(b) In loop 2, the J^{th} column of U keeps changing, thus it is important to store the upper triangular matrix U according to a row-by-row fashion (see Figure 1). This will assure to have a stride 1 in vector computation.

(c) In loop 3, the J1th row of L keeps changing, thus it is important to store the lower triangular matrix L according to a column-by-column fashion (see figure 1). This will assure to have a stride 1 in vector computation.

(d) The "scalar" defined in Table 11 is also refered to as "multiplier". In general, the average upper bandwidth (or UBW) of [U] is different from the average lower bandwidth (or LBW) of [L]. Factorizing the I^{th} row (of [U]) and the I^{th} column (of [L]) can be done much more efficiently by skipping some operations when the multiplier is zero. Figures 247 and 248 show what information are truly needed to factorize the I^{th} row and the I^{th} column.

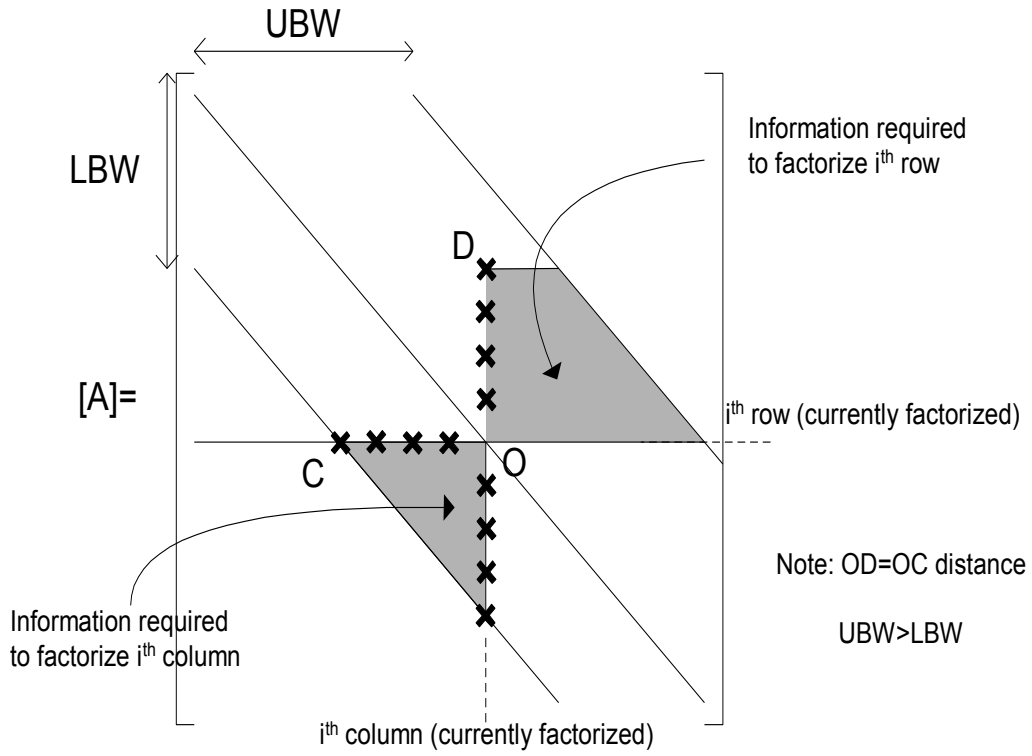


Figure 247. Unsymmetrical Factorization
(UBW>LWB; OD=OC)

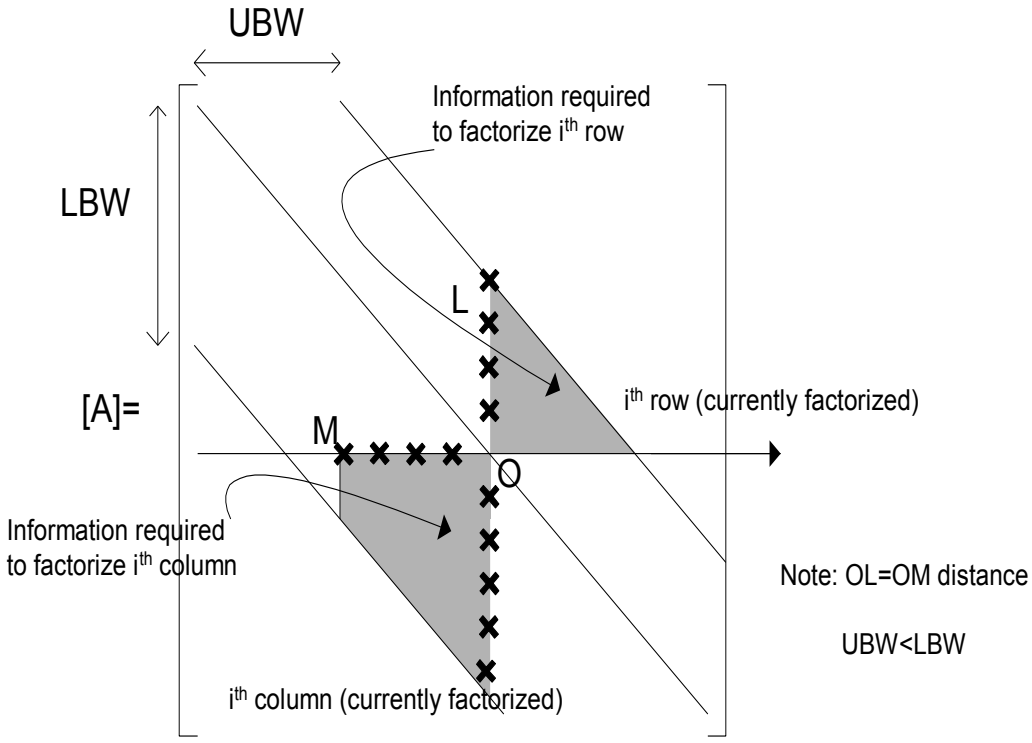


Figure 248. Unsymmetrical Factorization
(UBW<LWB; OM=OL)

Basic Algorithm for Decomposition of Variable Bandwidths / Column Heights Unsymmetrical Matrix

For many practical engineering applications, the unsymmetrical matrix is not full. Instead, the unsymmetrical matrix will have variable bandwidths and variable column heights as shown in figure 111.

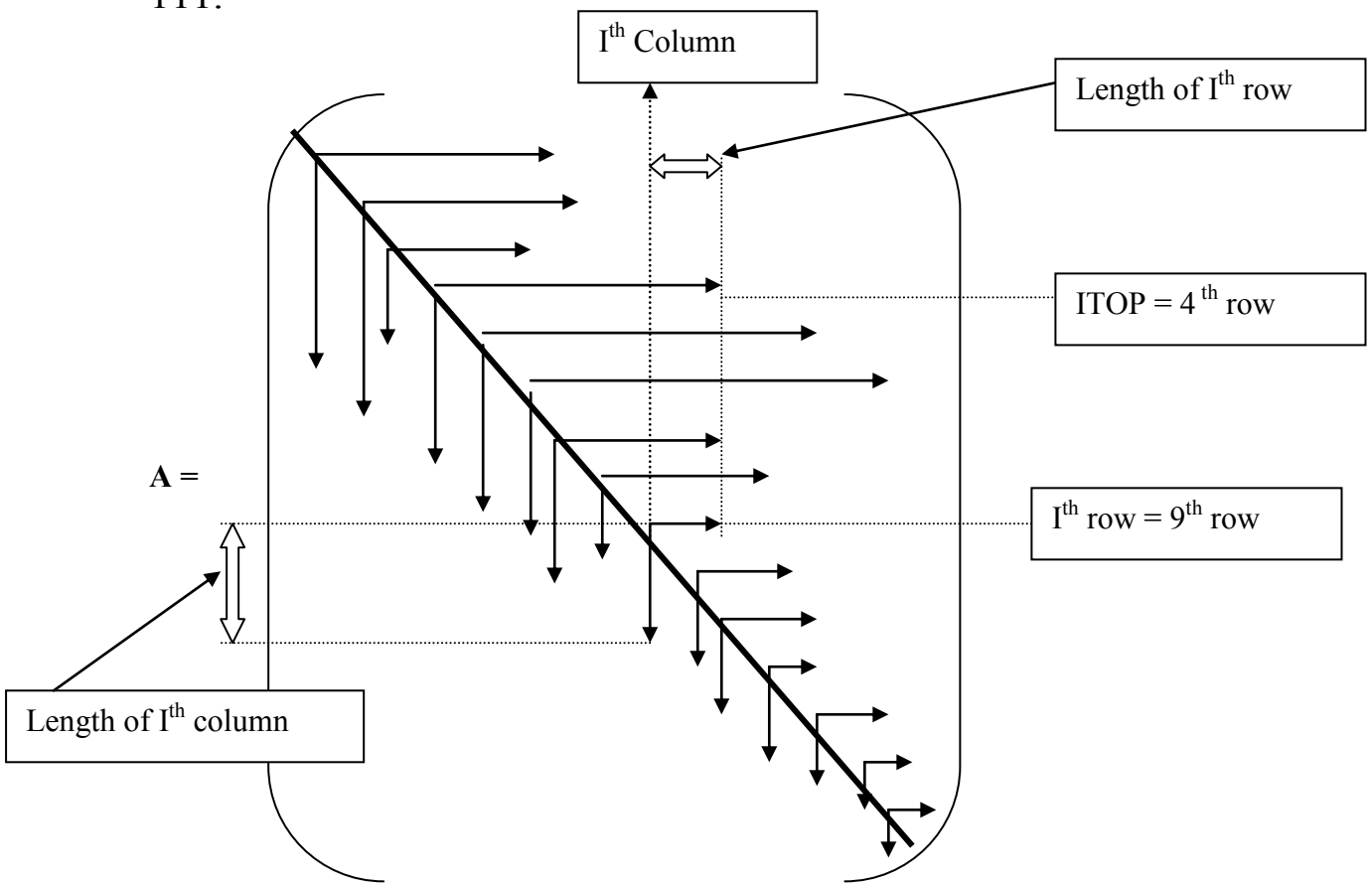


Figure 111 : Unsymmetrical Matrix with Variable Bandwidths and Column Heights.

In this case, to avoid unnecessary operations with zero values, the algorithm given in table 11 need to be modified slightly as shown in Table 1.

From Table I, one clearly sees that the previously factorized rows (please refer to loop 1) are used to partially factorized the current I^{th} row (please refer to loop 2) of the upper triangular matrix U. Thus, to improve the vector performance, one should try to increase the work loads of the innermost loop 2. This can be done by unrolling the outer loop 1. For example, a block of 6 (instead of just 1) previously factorized rows is used to partially factorize the current I^{th} row.

Similarly, the previously factorized columns (please refer to loop 1) are used to partially factorize the current I^{th} column (please refer to loop 3) of the lower triangular matrix L. A block of 6 (instead of just 1) previously factorized columns is used to partially factorize the current I^{th} column (inside loop 3).

Thus, Table 2 can be obtained by simply making the following minor modifications to Table 1.

(a) The increment of loop 1 is changed from 1 into 6 (to consider a block of 6 rows/columns at a time).

(b) Expanding the FORTRAN statement inside loop 2 to include the effects of using 6 rows at a time to partially factorize the current I^{th} row of upper matrix, [U].

(c) Expanding the FORTRAN statement inside loop 3 to include the effects of using 6 columns at a time to partially factorize the current I^{th} column of lower matrix, [L].

A careful comparison between Table 2 (vector algorithm for factorization) and Table 3 (parallel-vector algorithm for factorization) suggests that the latter can be obtained from the former with the following modifications.

(a) The outermost loop (for index I) is executed in parallel (in stead of sequential mode) by using a "Presched" parallel FORTRAN statement.

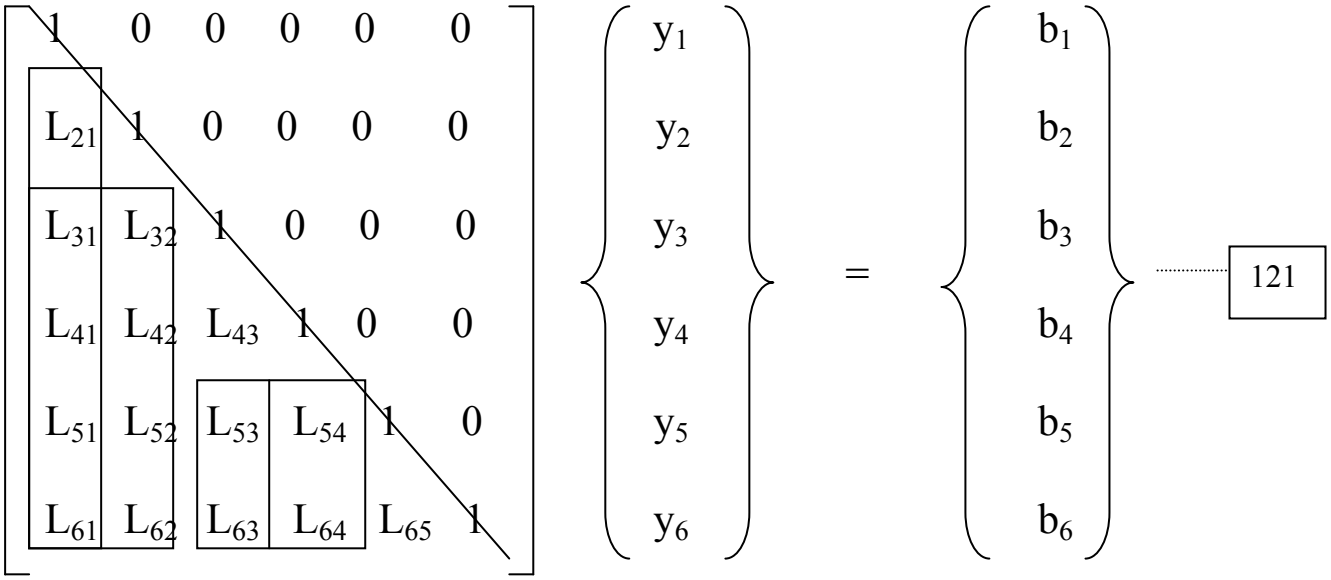
(b) Loop 1 (for index K) in Table 2 is separated into 2 loops (loops 1 and 10) in Table 3.
- In Table 2, the index K goes from "ITOP" to "I-1".
- In Table 3, the index K goes from "ITOP" to "I-NP" (see loop 1) and then, from "I-NP+1" to "I-1" (see loop 10).

(c) The "Copy" parallel FORTRAN statement inside loop 10 (of Table 3) will assure that the previous K^{th} row has been completely factorized (or else the processor will wait!) and can now be safely used to partially factorize the current I^{th} row (see loop 20) and I^{th} column (see loops 30 and 4).

(d) The "Produce" parallel FORTRAN statement (after loop 4) is used to broadcast to all other processors that the I^{th} row/column have been completely factorized now.

Forward Solution Phase [L] {y} = {b}

To simplify the discussions, let us consider a 6x6 full-system as shown in the following equations.



The forward solution for the unknown vector {y} can be proceeded as following

$$y_1 = b_1$$
$$y_2 = b_2 - L_{21} y_1$$
$$y_I = b_I - \sum_{k=1}^{I-1} L_{I,k}\, y_k$$

122

Since the lower triangular matrix has been generated and stored in a column-by-column fashion (please recall Figure 1), thus column 1 of [L] has stride 1. Furthermore, to improve the vector performance, one should try to work with long vector in the innermost do-loop. Thus, a good strategy will be outline in the following.

Step 1. Solve for the unknown y_1 (according to Eq. 122)
Step 2. Use the first column of [L] and operate on the known scalar y_1 in order to update the right-hand-side vector {b}. Thus, the unknown y_2 can be found.
Step 3. Use the second column of [L] and operate on the known scalar y_2 in order to update the right-hand-side vector {b}. Thus, the unknown y_3 can be found.
Step 4. Continue to do "similar" operating as mentioned in step 2&3, until all unknowns of vector {y} are found.

The above step-by-step procedure can be simply coded as shown in Table 123.

Table 123 : Basic Algorithm for Forward Solution

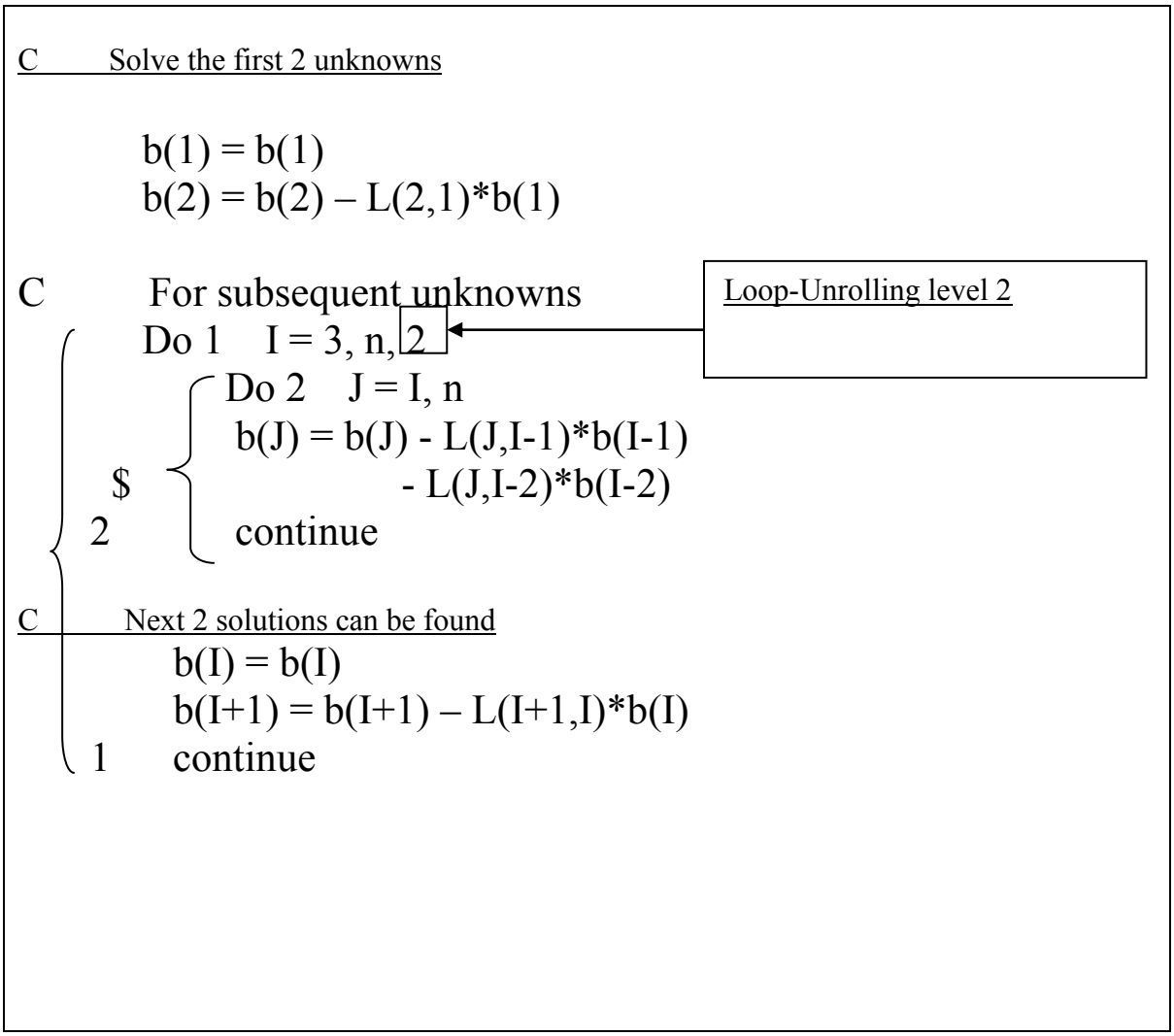
```
C       Solve for the first unknown
C       Note : solution vector {y} will over write right-hand-side
C       vector {b} to save computer memory

        b(1) = b(1)

C       Try to solve the subsequent unknowns

        Do  1  I = 2, n, 1
              Do 2          J = I, n
C       use the previously known solution to update the right-hand-side vector {b}
    2         b(J) = b(J) – L(J,I-1) * b(I-1)

C       Next solution is readily found

        b(I) = b(I)
    1   continue
```

It should be mentioned here that inside loop 2 of Table 123, one has SAXPY operations (a vector {b} ± scalar y * another vector L), thus the innermost loop 2 can be executed very efficiently on the vector computer (such as the CRAY-YMP, CRAY-C90 etc…).

However, a careful observation of the above 4-step procedure and the data structure shown in Eq. (121) suggests that even better vector-performance can be achieved (by using the "loop unrolling" technique) with a simple modification to Table 123.

The key idea in "loop unrolling" technique is to add more work loads (SAXPY operations) into the innermost do-loop (see loop 2 of Table 123). A simple way to achieve this objective is to use 2 (or more) columns (instead of just 1) of matrix [L] and operate on previously known 2 (instead of just 1) solutions. Thus, loop-unrolling algorithm for forward solution can be shown in Table 124.

Table 124 : Loop-unrolling (Level 2) for Forward Solution



Comments on Table 124 :

(a) In actual computer implementation, loop-unrolling level 6 or 8 can be used instead of using unrolling level 2 (see the increment 2 in loop 1).

(b) For a general matrix with dimension n, the use of loop-unrolling technique will require “special” treatments for the left-over columns of the matrix L

(c) To simplify the discussions, the matrix system of equations shown in Eq. (121) is assumed to be "full". However, in actual computer implementation, variable column-heights of the lower triangular matrix [L], and variable row-length (or bandwidth) of the upper triangular matrix [U] can be accommodated to avoid unnecessary operations (on the zeros).

(d) In actual computer implementation, the lower and upper factorized matrices [L] and [U] will be stored in a 1-D array and the original matrix (which is also stored in a 1-D array) will be over written by [L] and [U] in order to save computer memory.

Backward Solution Phase [U] {x} = {y}

To simplify the discussions, let us consider the following 6x6 full system of equations.

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} & U_{15} & U_{16} \\ 0 & U_{22} & U_{23} & U_{24} & U_{25} & U_{26} \\ 0 & 0 & U_{33} & U_{34} & U_{35} & U_{36} \\ 0 & 0 & 0 & U_{44} & U_{45} & U_{46} \\ 0 & 0 & 0 & 0 & U_{55} & U_{56} \\ 0 & 0 & 0 & 0 & 0 & U_{66} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{Bmatrix} = \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{Bmatrix} \quad \cdots\cdots\cdots \boxed{131}$$

The backward solution for the unknown vector {x} can be proceeded as following

$$x_6 = \frac{y_6}{u_{6,6}} \quad \Longrightarrow \quad x_5 = \frac{y_5 - u_{5,6}\, x_6}{u_{5,5}}$$

$$\vdots \qquad\qquad \vdots$$

$$x_I = \frac{\left(y_I - \sum_{K=I+1}^{N} u_{I,k}\, x_k \right)}{u_{I,I}} \qquad \text{132}$$

As an example, $x_2 = \dfrac{y_2 - \left(u_{2,3}\, x_3 + u_{2,4}\, x_4 + u_{2,5}\, x_5 + u_{2,6}\, x_6 \right)}{u_{2,2}}$

The operations involved in the above parenthesis are called "dot product" operations, since it involves $\{u_{2,3}, u_{2,4}, u_{2,5}, u_{2,6}\} \cdot \begin{Bmatrix} x_3 \\ x_4 \\ x_5 \\ x_6 \end{Bmatrix}$

Since the upper triangular matrix has been generated and stored in a row-by-row fashion (please refer to Figure 1), thus each row of [U] has stride 1. However, each column of [U] has very undesirable stride (column stride of [U] is greater than 1). Due to this reason, it is <u>not</u> efficient (in this case) to use loop-unrolling technique (for example, having found the unknown x_6, then using column 6 to operate on the scalar x_6 for the purpose of updating the right-hand vector {y}) as discussed in the previous section. The backward solution (please refer to Eq. 132) can be coded using "dot-product" operations (instead of SAXPY operations as discussed in the forward solution phase) as shown in Table 141.

Table 141 : Basic Algorithm for Backward Solution

```
C       Solve the last unknown

        X(N) = y(N) / u(N,N)
C       For subsequent unknowns
        Do  1   I = N-1, 1, -1
C       Performing the summation (or dot-product) operations in
C       Eq. (132)
          Do 2  K = I + 1, N
        2 Sum1 = Sum1 + U(I,K) * x(K)

        x(I) = (y(I) – Sum1) / u(I,I)
        1 Continue
```

It should be mentioned here that the dot-product operations inside loop 2 (of Table 141) can be vectorized quite well (on vector computers) since the row vector of [U] has stride 1 (recalled that the matrix U is stored in a row-by-row fashion).

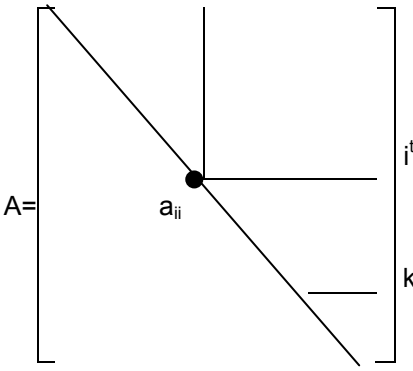
However, a careful observation of Eq. (132) and the storage scheme used for matrix [U] shown in Eq. (131) suggests that even better vector-performance can be achieved by using the "vector-unrolling" technique with a simple modification to Table 141.

The key idea in "vector-unrolling" technique is to add more work loads (dot-product operations) into the innermost do-loop (see loop 2 of Table 141). A simple way to achieve this objective is to use 2 (or more) rows (instead of just 1) of matrix [U] and operate on previously known 2 (instead of just 1) solutions. The resulted "vector-unrolling" (level 2 unrolling is assumed) algorithm for backward solution is illustrated in Table 142.

Table 142 Vector-Unrolling Algorithm for Backward Solution

```
C       Solve the last 2 (or more) unknowns

        x(N) = y(N) / u(N,N)

        x(N-1) = [y(N-1) - u(N-1,N) * x(N)] / u(N-1,N-1)

C       For subsequent unknowns
        Do  1   I = N-2, 1, -2          <-- vector-Unrolling level 2

C       Performing 2 (or more) dot-product operations in Eq. 132

        Do 2  K = I + 1, N
        Sum1 = Sum1 + U(I,K) * x(K)
2       Sum2 = Sum2 + U(I-1,K) * x(K)

        x(I) = [y(I) – Sum1] / u(I,I)

        x(I-1) = [y(I-1) – Sum2 – U(I-1,I)*x(I)] / u(I-1,I-1)

1       Continue
```

Pivoting



$a_{ii} = a_{ii} - \sum(\ )*(\ )$ => If a_{ii} near zero = ε

$a_{ij} = \left[a_{ij} - \sum(\ )*(\ )\right]/(a_{ii} = \varepsilon)$

For Sun => machine Double precision = 10^{-15}

switch

Lead to unsym. Matrix => not use cholesky

√ Partial pivoting => switch rows

X Complete Pivoting => switch rows & columns

X Diagonal Pivoting

Still sym. after switching rows & columns
Less accurate, not accurate

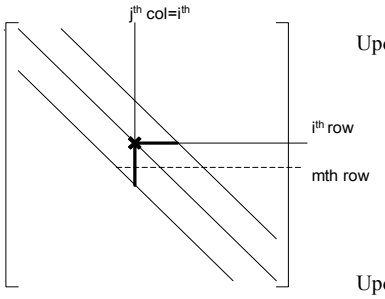
A=

10^{-10}	1	0	100	0	200	0	0	0	0
1	2								
0		3							
100			4						
0				5					
200					6				
0						7			
0							8		
0								9	
0									10

Notes (a) When switch rows, then upper b.w. will be at most double (proof: easily)
(b) When switching rows, then lower b.w. will be at most full matrix
proof:
switch row1 & row6 => lower b.w.=6

later on, we may find a_{66} too small
=> switch row 6 with ,say, row 10 => lower b.w.=10=full

[A]=



Update column

For m = i, i+1, … , i+NBW

$a_{mj} = \left[a_{mj} - (col\#j)^T (row\#m)\right]/(a_{ii} = ??)$

(a_{ii}=not known yet)
Then:
Pivoting element=Max/{a_{mj} where m= i, i+1, … , i+NBW
Example row k ⇔ row i

Update row

Update (new) row i (same as "old" row k)

$a_{il} = a_{il} - \sum(\ )*(\ )$

Note:

(a) After switching rows of the coeff matrix, we also have to switch RHS(=load vector) too.
(b) Row lengths also change when switching rows. In the very beginning of Eq. Solver, we already reserve extra memory (for worst case) for switching rows => no need to re-define diagonal pointer array MAXA (??)

A Fast Parallel Algorithm for Generation and Assembly of Finite Element Stiffness and Mass Matrices

By

Majdi Baddourah (ODU)
Olaf Storaasli (NASA-Langley)
Ed Carmona (USAF Phillips Labs)
Duc Nguyen (ODU)

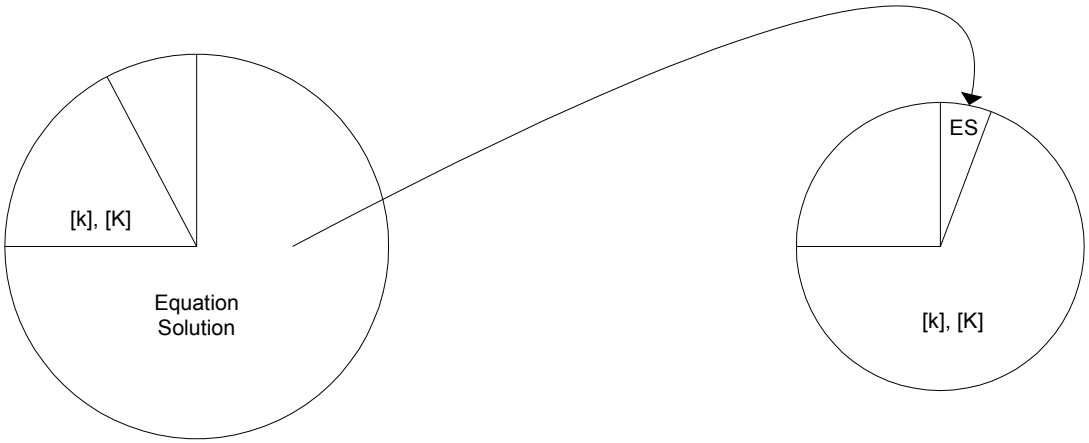
AIAA/ASME/ASCE/AHS/ASC 32nd
Structures, Structural Dynamics and Materials
Conference
Baltimore, MD
April 8-10, 1991

A NEW PARALLEL-VECTOR FINITE ELEMENT ANALYSIS SOFTWARE ON DISTRIBUTED MEMORY COMPUTERS

Jiangning Qin and Duc T. Nguyen
Center for Multidisciplinary Parallel-Vector Computation
Civil Engineering Department, Old Dominion University, Norfolk, VA 23529

Motivation

- Equation solution dominates analysis time
- Equation solution time reduced significantly (PVSOLVE by Storaasli, Nguyen and Agarwal)



- Time to generate/assembly [K] significant:
  - Complex element types
  - Nonlinear structural analysis
  - Structural optimization
  - Control-structure interaction

Traditional Element Assembly

Parallel Approach: Assign elements to different processors

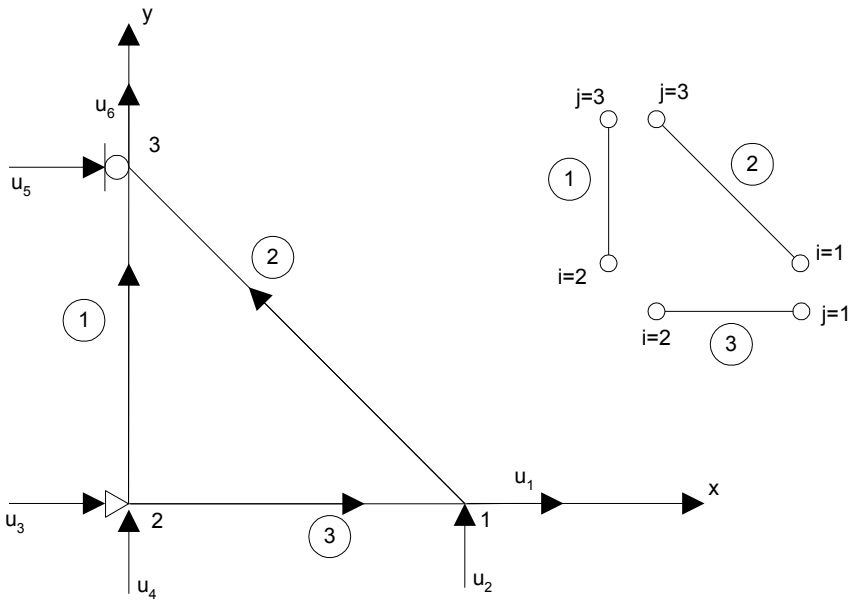
DO 1 e = 1,3 elements
Generate element stiffness matrix: $[k^{(e)}]$
Assemble global $[K]=\sum_{e} [k^{(e)}]$

1 Continue

Synchronization Bottleneck

Elements with common nodes are added simultaneously ($\Sigma[k^{(e)}]$)!

Node-by-Node Element Generation



Step 1: Element Connectivity (old)

Element No.	Node i	Node j
1	2	3
2	1	3
3	2	1

Step 2: Nodal Connectivity (new)

Node No.	DOF	Elements Attached to	
		Node i	Node j
1	1, 2	2	3
2	3, 4	1, 3	none
3	5, 6	none	1, 2

Step 3: Parallel Generation and Assembly of $k_{ii}^{(e)}$ for each Node of a Structure

Table 4 Parallel Generation-Assembly of $k_{ii}^{(e)}$

	1	2	3	4	5	6
1	2	2				
2		2				
3			1, 3	1, 3		
4				1, 3		
5						
6						

Step 4: Parallel Generation and Assembly of $k_{jj}^{(e)}$ for Each Node of a Structure

Table 5 Parallel Generation-Assembly of $k_{jj}^{(e)}$

	1	2	3	4	5	6
1	3	3				
2		3				
3						
4						
5					1, 2	1, 2
6						1,2

Step 5:

In this step, the portion $K_{ij}=\Sigma\ k_{ij}^{(e)}$ of the structural stiffness matrix, K, is generated and assembled in a parallel computer environment

In this paper, the information for nodes j is used in this step. Thus, processor 1 is assigned to node 1 to process element 3. Element 3 is connected to DOF 1, 2, 3 and 4 and its contribution to K_{ii} and K_{jj} have been done in step3 and step 4, respectively. Processor 1 will generate $k_{ij}^{(e=3)}$ and add its contribution to the appropriate locations of K_{ij}. Simultaneously, processor 3 is assigned to node 3 to process elements 1 and 2. Processor 3 will, therefore, generate $k_{ij}^{(e)}$ for elements e=1 and 2, and add its contribution to the appropriate locations of K_{ij}. In this step, Processor 2 is idle since there are no elements with node j=2. The parallel generation/assembly of $k_{ij}^{(e)}$ for each structural node is conveniently represented in Table 6.

Table 6 Parallel Generation-Assembly of $k_{ij}^{(e)}$

	1	2	3	4	5	6
1			3	3	2	2
2			3	3	2	2
3					1	1
4					1	1
5						
6						

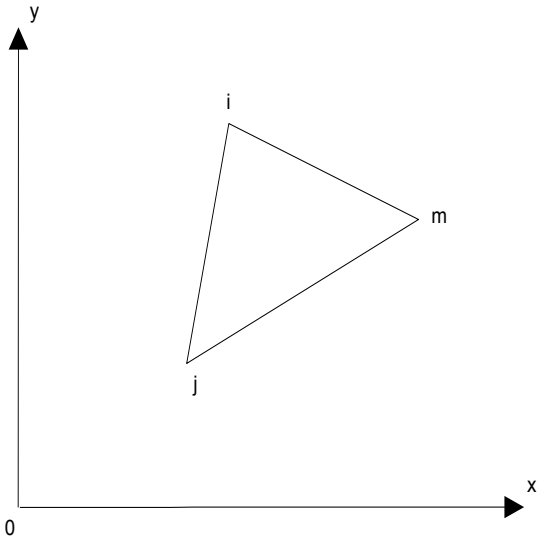


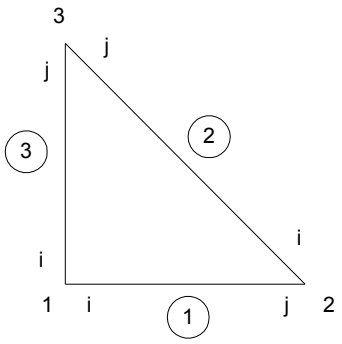
Fig 2. Three node Triangular Element

A two dimensional, 6 x 6 element stiffness matrix $[k^{(e)}]$ can be symbolically represented as:

$$[K^{(e)}]= \begin{bmatrix} K_{ii}^{(e)} & K_{ij}^{(e)} & K_{im}^{(e)} \\ & K_{ji}^{(e)} & K_{jm}^{(e)} \\ & & K_{mm}^{(e)} \end{bmatrix} \quad (2)$$

In Eq. (2), $K_{ii}^{(e)}$, $K_{jj}^{(e)}$, and $K_{mm}^{(e)}$ refer to the 2 x 2 sub-matrices which represent a portion of an element stiffness matrix attached to node i, node j, and node m, respectively. The coupling effect between nodes i, j, and m of an element stiffness matrix $[K^{(e)}]$ is represented by the sub-matrices $K_{ij}^{(e)}$, $K_{im}^{(e)}$, and $K_{jm}^{(e)}$. Thus, for a three-node triangular element, an additional step needs to be inserted before the last step (step 5) for parallel generation and assembly of $K_{mm}^{(e)}$ for each node m of the structure.

Alternative Implementation of Baddourah-Nguyen's Generation and Assembly method



Proc (or Node) Number	Els. Attached Node-i	Els. Attached Node-j
1	1,3	None
2	2	1
3	None	2,3

For Each processor p^{th}

Do 1 L =1, NEL(p^{th})

Step 1: Kii (& Kij where j>i)

	1	2	3	4	5	6th Dof
1	1, 3	1, 3	1	1	3	3
2		1, 3	1	1	3	3
3						
4						
5						
6						

Processor 1

		2	2	2	2	
			2	2	2	

Processor 2

Processor 3 (idle) !!

Step 2 : Kjj (& Kji where i>j)

	1	2	3	4	5	6th Dof
1						
2						
3						
4						
5						
6						

Processor 1 (idle!)

		1	1			
			1			

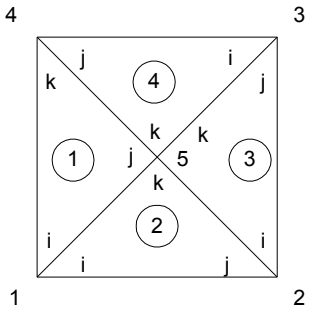
Processor 2

				2, 3	2, 3	
					2, 3	

Processor 3

1 continue

3-Node Element case (for simplicity assuming each node has only 1 dof, say x-translation)

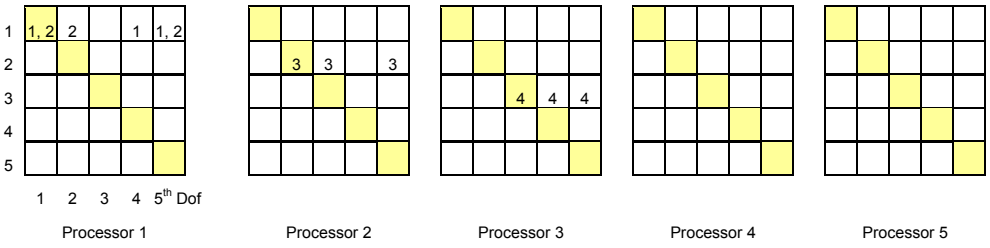


Processor (or Node) Number	Els. To Node-i	Els. To Node-j	Els. To Node-k
1	1, 2	None	None
2	3	2	None
3	4	3	None
4	None	4	1
5	None	1	2, 3, 4

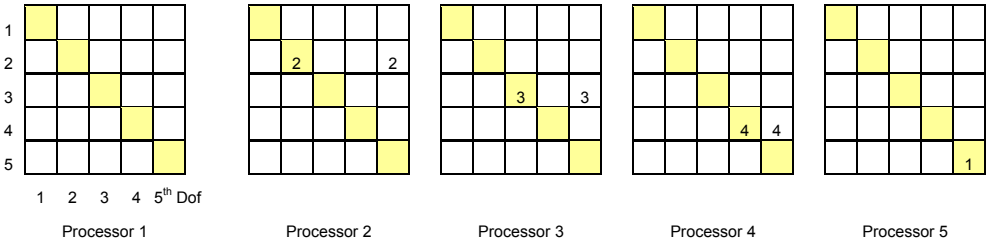
For each Processor pth

Do 1 L = 1, NEL (pth)

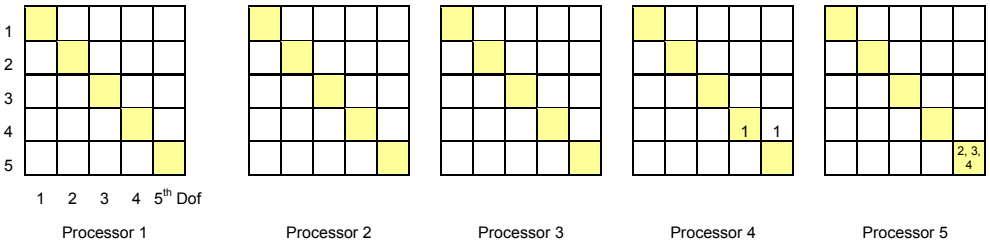
Step 1: Kii (&Kij, Kik where j, k > i)



Step 2: Kjj (&Kji, Kjk where i, k > j)



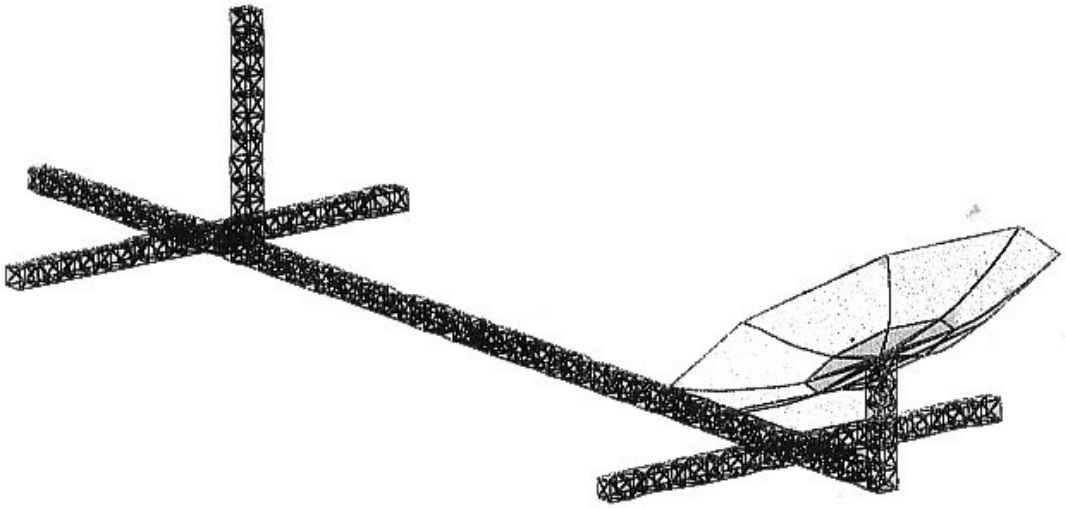
Step 3: Kkk (&Kki, Kkj where i, j > k)

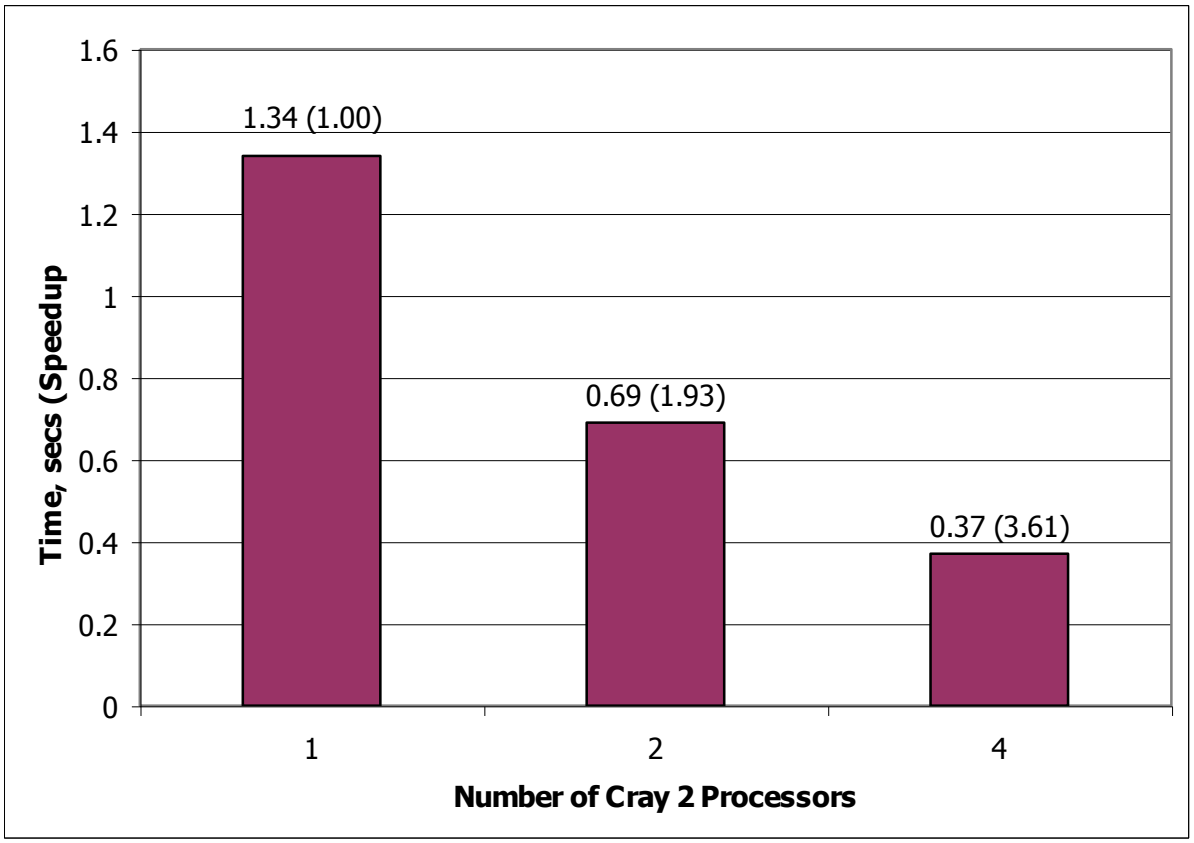


1 continue

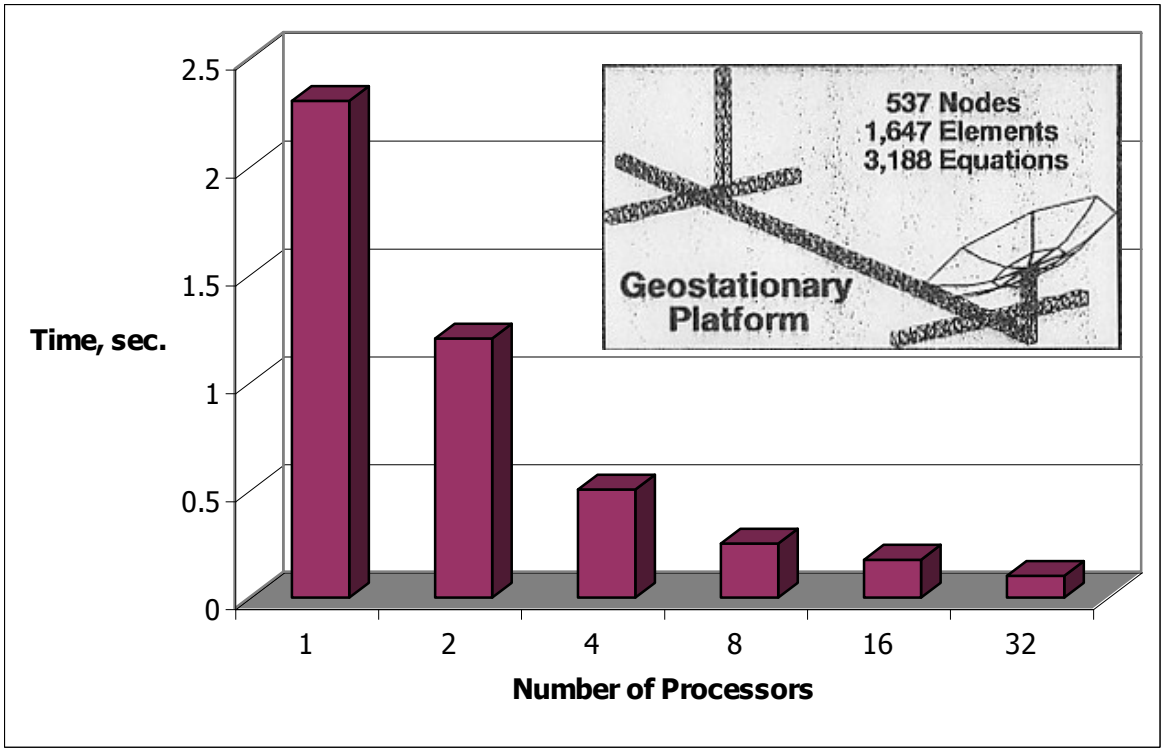
3-D Geostationary Platform

(537 Nodes, 1647 Elements, 3188 Degrees of Freedom)





Matrix Generation and Assembly on iPSC/860



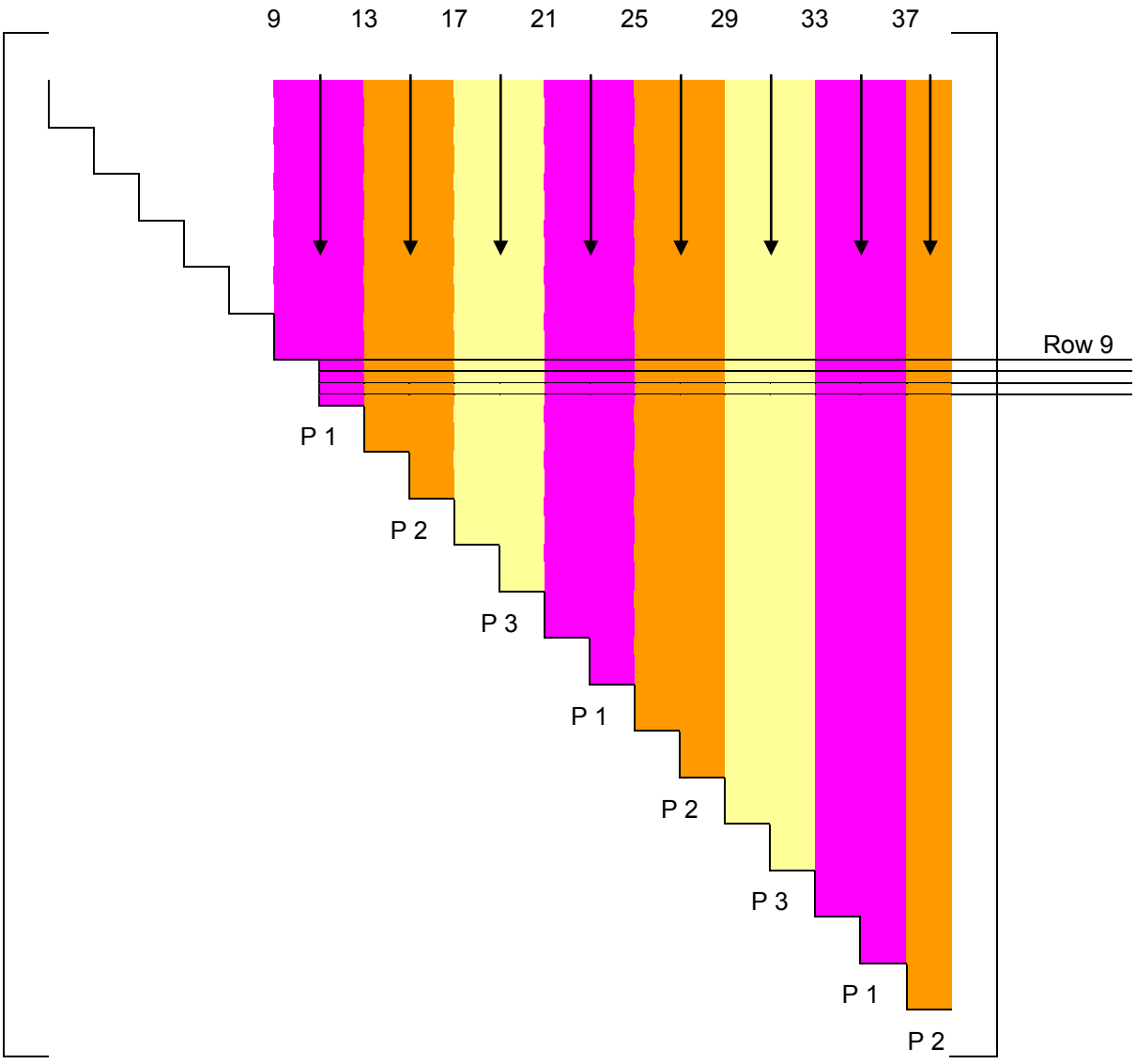
Conclusion

- Simple, efficient method developed and tested to:
  - Generate element [k(e)] in parallel
  - Assemble global [K] in parallel

Which is:

- ❖ General (2&3 node elements, mixed elements)
- ❖ Accurate (exact agreement with SAP-4)
- ❖ Efficient (good speedup on multiple processors)
- ❖ Portable (to Cray Y-MP, Cray 2, Convex, Alliant)
- ❖ Modular (to different finite-element codes)
- ❖ Consistent (with parallel-vector equation solver)

- Ingredients (element generation, assembly solver) for next generation parallel finite element code



Block Storage scheme for matrix A in a one-dimensional array
(can be skyline or variable band)
(can be shared or distributed memory computers)

New Parallel G/A Approach

Element #	Element DOF #	Processor #
1	1->6	1
2	7->12	1, 2
3	7, 8, 9, 18, 19, 20	1, 3
* * * *		
28	20, 21, 22, 80, 81, 82	3, 1, 2
* * * NEL		

Processor #	Element #
1	1, 2, 3, 28
2	2, 28
3	3, 28
*	*
*	*
*	*
*	*
*	*
NP	*

New Parallel G/A Approach

In Parallel, for each processor i:

Do 1 K=1, NEL (i)

- ELDOF (k^{th} element) = 20, 21, 22, 80, 81, 82
- Generate Stiffness of the k^{th} element
- Assemble the entire (or just a portion of) k^{th} element stiffness

1 Continue

In parallel, for each processor i:

Do 1 K = 1, ALLELS

- ELDOF (kth element) = 20, 21, 22, 80, 81, 82

Do 2 M=1, 6

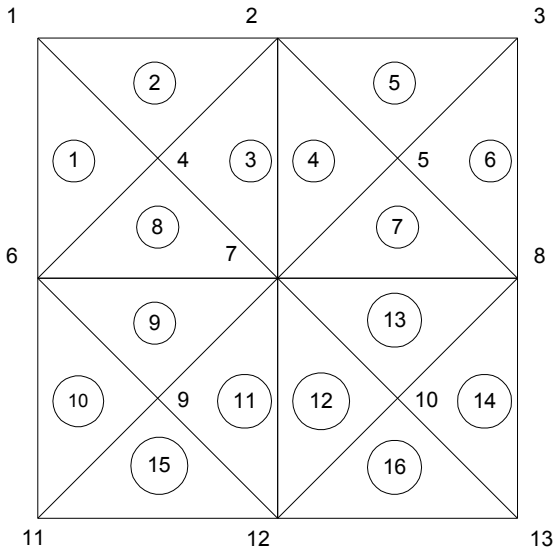
If (ELDOF(M) belongs to Processor i) then

- Record element k belongs to processor i
- Exit loop 2, and go to loop 1

Endif

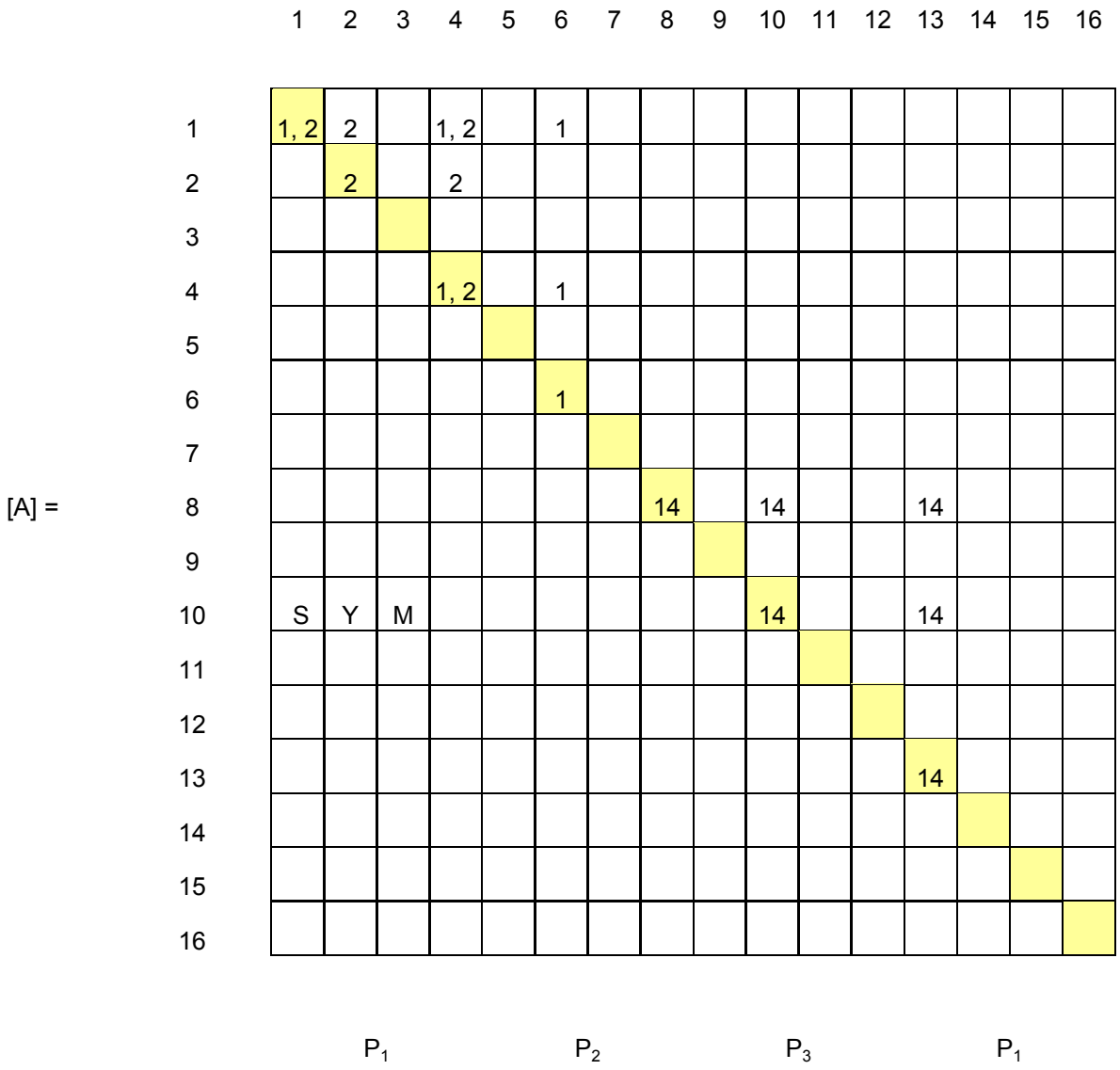
2 continue

1 continue

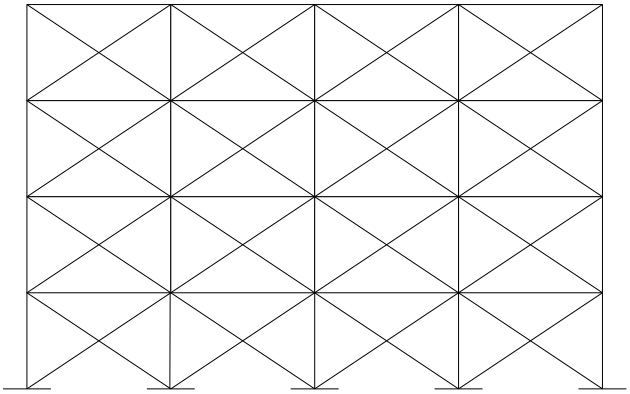


Element No.	DOF No.	Processor No.
①	1, 6, 4	1, 2
②	1, 4, 2	1
3	2, 4, 7	1, 2
4	2, 7, 5	1, 2
5	2, 5, 3	1, 2
6	3, 5, 8	1, 2
7	5, 7, 8	2
8	4, 6, 7	1, 2
9	6, 9, 7	2, 3
10	6, 11, 9	2, 3
11	7, 9, 12	2, 3
12	7, 12, 10	2, 3
13	7, 10, 8	2, 3
⑭	8, 10, 13	1, 2, 3
15	9, 11, 12	3
16	10, 12, 13	1, 3

Processor No.	Element No.
1 2 3	1, 6, 8, 14, 16 1, 3, 14 9, 16



Assuming NP=3 processors (= P_1, P_2 and P_3)
Unrolling level=4



Example. 2-D truss structures with ‘nb’ bays and ‘ns’ stories are denoted as nb x ns. Table 1 gives the CPU times for the generation and assembly of the global stiffness matrix (on the Gamma computer with up to 128 processors). For the 200 x 6 model, nel=4806, neq=2412. For the 1 x 1650 model, nel=82500 and neq=66000

Table 1. CPU times for the generation and assembly of the global matrix

nb x ns	k	1	2	4	8	16	32	64	128
200 x 6	4	0.337	0.206	0.1036	0.0593	0.029	0.01398	0.00718	0.0034
200 x 6	8	0.339	0.1885	0.1145	0.0569	0.0273	0.01412	0.00664	0.0033
1 x 16500	4	4.64	3.537	1.793	0.9083	0.4515	0.2267	0.114	0.057
1 x 16500	8	-	2.985	1.507	0.7564	0.376	0.188	0.0945	0.0481

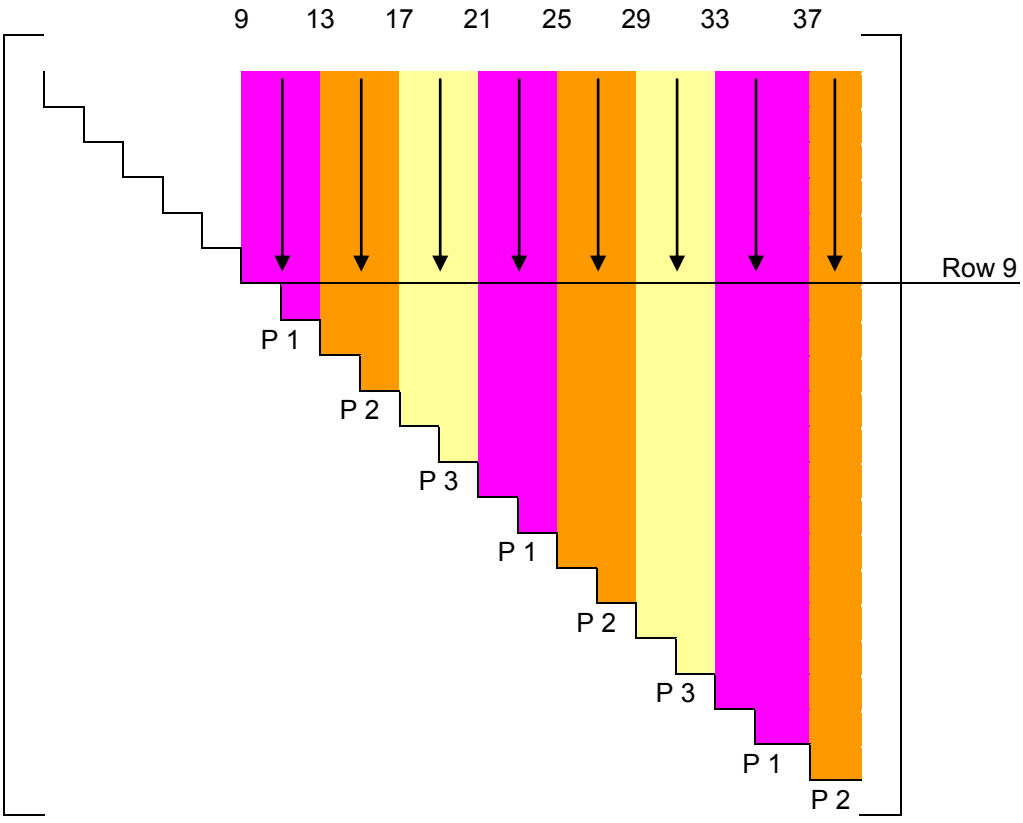
* timings for task 1 is not included here.

Parallel-Vector Equation Solver For Distributed-Memory Computers

Jiangning Qin and Duc T. Nguyen
Center for Multi-disciplinary Parallel-Vector computation
Civil Engineering Department
Old Dominion University, Norfolk, Virginia 23529

2nd Symposium
on
Parallel Computational Methods
For
Large-Scale Structural Analysis and Design
February 24-25, 1993
Marriott Hotel, Norfolk, Virginia
Sponsor
NASA Langley Research Center
Hampton, Virginia

Storage scheme for factorization



Block storage scheme for matrix A in a one-dimensional array

Vector unrolling & Blockwise updating:

$$sum_1 = sum_1 + (column9)\cdot(column9)$$

$$sum_2 = sum_2 + (column9)\cdot(column10)$$

$$sum_3 = sum_3 + (column9)\cdot(column11)$$

$$sum_4 = sum_4 + (column9)\cdot(column12)$$

are executed by processor 1, while the following dot product operation

$$sum_1 = sum_1 + (column9)\cdot(column13)$$

$$sum_2 = sum_2 + (column9)\cdot(column14)$$

$$sum_3 = sum_3 + (column9)\cdot(column15)$$

$$sum_4 = sum_4 + (column9)\cdot(column16)$$

are processed by processor 2 etc…

A skeleton pseudo-code for parallel-vector Choleski factorization

```
      Parallel Do 100 ith row=1, n, 4          (say i=9)
C     …me = processor number
      If ("me" have the ith column) then
       Do 200 j = 0, 3                         (This loop is required due to skipping occurs in the first
                                                loop)
      II=(Global row #) = i+j
       Do 400 column # JJ = II, n, 4           (The values of JJ maybe skipped, depending on the value
                                                of "me")

        Sum1=sum2=sum3=sum4=0

      Do 500 row k=1, 2, …., II

        Sum1=sum1+u(k,II)*u(k,jj)
        Sum2=sum2+u(k,II)*u(k,jj+1)
        Sum3=sum3+u(k,II)*u(k,jj+2)
        Sum4=sum4+u(k,II)*u(k,jj+3)

500   Continue

      if (II.Eq.JJ) then

        u(ii,jj) = SQRT(A(ii,jj) – sum1)

        send column jj to all other processors
      else

        u(ii,jj) = (A(ii,jj) – Sum1) / u(ii,ii),  u(ii,jj+1) = (A(ii,jj+1) – Sum2) / u(ii,ii),
        u(ii,jj+2) = (A(ii,jj+2) – Sum3) / u(ii,ii),  u(ii,jj+3) = (A(ii,jj+3) – Sum4) / u(ii,ii)

      endif
400   continue
200   continue

      Else
c…    all other processors do the following
   • receive column #jj
   • update row ii
      Endif
100   continue
```

Table 2. Vector performance with different options.
Decomposition of a 1000x1000 matrix on one processor (Intel iPSC/860)

Options	Time (seconds)
No vector-unrolling	44.8
Vector-unrolling level 4	35.8
Vector-unrolling+block-wise updating	22.2
DDOT+vector-unrolling+block-wise updating	20.2
DDOT+vector-unrolling without block-wise updating	14.5

Table 3 : Communication schemes

Single-send scheme	Double-send scheme

Do 100 i = 1, n, 4
If (I have the i^{th} column) then
Do 200 j = 0,3
Update the $(i+j)^{th}$ column
Send the $(i+j)^{th}$ column to all other processors (fan-out)
Update the $(i+j)^{th}$ row
200 continue
else
do 300 j=0, 3
receive the $(i+j)^{th}$ column
update the $(i+j)^{th}$ row
300 continue
endif
100 continue
	Do 100 i = 1, n, 4
	If (I have the i^{th} column) then
	Do 200 j = 0,3
	Update the $(i+j)^{th}$ column
	Send the $(i+j)^{th}$ column to the next processor
	Send the $(i+j)^{th}$ column to all other processors
	Update the $(i+j)^{th}$ row
	200 continue
	else
	do 300 j=0, 3
	receive the $(i+j)^{th}$ column
	update the $(i+j)^{th}$ row
	300 continue
	endif
	100 continue

Sequential send (or Ring) scheme

Do 100 i = 1, n, 4
If (ME have the i^{th} column) then

Loop 200
- Update the $(i+j)^{th}$ column
- Send the $(i+j)^{th}$ column to next processor
- Update the $(i+j)^{th}$ row

P_0 send info to P_1 (=col 9)

Else
- Every processor receives info from previous processor (ME-1)
- If (ME ≠Processor which possess [i-1]) then
  - ❖ Send info to next processor which possess (i+4)
  - ❖ With exception: Last processor NP send info to processor 0

  endif
- Update $(i+j)^{th}$ row

Endif

P_1 sends info to P_2
P_2 sends info to P_3
P_3 sends info to P_4

We don't want P_4 send info to P_1 (col.9) because P_1 wants to send col.9 to all others!

Table 4. Communication performance
Decomposition of a 4000x4000 matrix on 16 processors (Intel iPSC/860)

Options	Time (sec)
Single-send (with CSEND, CRECV)	231
Single-send (with ISEND, IRECV)	192
Single-send (DDOT+ISEND, IRECV)	120
Double-send (DDOT+ISEND, IRECV)	108
Sequential-send (DDOT+CSEND, CRECV)	104

Forward Elimination:

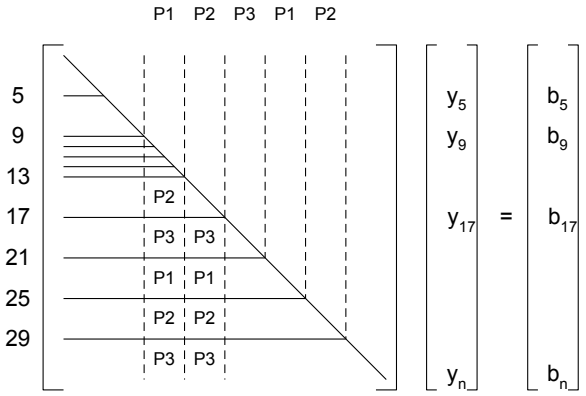


Figure 2: Parallel Forward Elimination

Assuming the first 8 unknowns have been solved completely

$$y_9 = \frac{b_9 - \sum_{i=1}^{8} u_{i,9} \cdot y_i}{u_{9,9}}$$

or

$$y_9 = \frac{b_9 - (u_{1,9} \cdot y_1 + u_{2,9} \cdot y_2 + \ldots + u_{8,9} \cdot y_8)}{u_{9,9}}$$

(10, 11, 12)

To be solved "completely" by Processor P_1

For example, processor 1 will compute: (partially)

$$y_{21} \cdot (\text{incomplete}) = \frac{b_{21} - (u_{1,21} \cdot y_1 + \ldots + u_{12,21} \cdot y_{12}) - (\ldots)}{u_{21,21}}$$

(22, 23, 24)

At the same time, processor 3 will compute: (partially)

$$y_{17} \cdot (\text{incomplete}) = \frac{b_{17} - (u_{1,17} \cdot y_1 + \ldots + u_{12,17} \cdot y_{12}) - (\ldots)}{u_{17,17}}$$

(18, 19, 20)

and will also compute:
y_{18}(incomplete) through y_{20} (incomplete), and y_{29} (incomplete) through y_{32} (incomplete)

Forward elimination

Do 100 i = 1, n, 4
 	
If ("me" have the i^{th} row) then ➔ Ex: row i=9
 	
c.. For 1 processor
 	
update y(i), y(i+1), y(i+2), y(i+3)
fan-out (or send to all)
y(i), y(i+1), y(i+2), y(i+3)
partially update y(j) (for i+3 < j < n)
for processor "me"'s portion only
 	
else
c… For all other processors
 	
receive y(i), y(i+1), y(i+2), y(i+3)
partially update y(j) (for i+3 < j < n)
endif
 	
100 Continue

Backward Elimination:

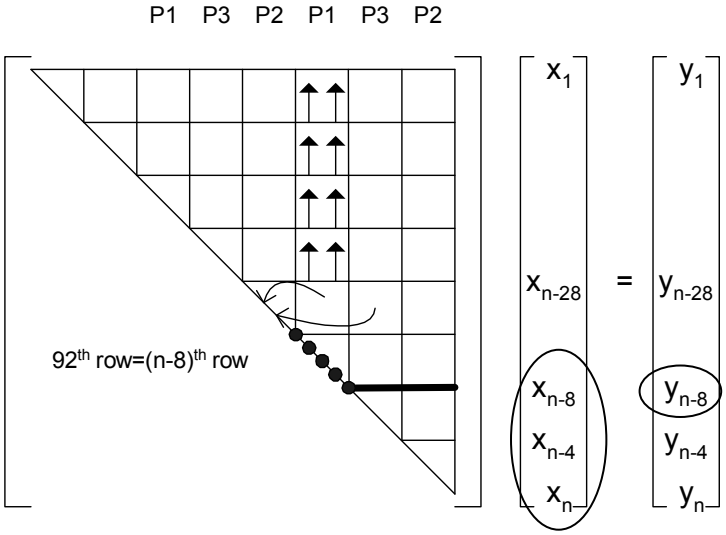


Figure 3: Backward Elimination

$$x_{92} = \frac{y_{92} - \sum_{i=93}^{100} u_{92,i}\, x_i}{u_{92,92}}$$

Or

$$x_{92} = \frac{y_{92} - \left(u_{92,93} \cdot x_{93} + u_{92,94} \cdot x_{94} + \ldots + u_{92,100} \cdot x_{100}\right)}{u_{92,92}}$$

Thus, one can clearly see that processor 1 (see Figure 3) can easily calculate unknowns x_{92} to x_{89}, since x_{100} to x_{93} have already been "completely" calculated.

Having completed the final solution for x_{92} through x_{89}, processor 1 continues to compute the "partial" (or incomplete) solution for x_{88} through x_{85}. Processor 1 then send these partial solutions to the next processor (on its left, say processor 2) and processor 1 continues to find the partial solution for x_{84} through x_1.

Meanwhile, a "if check" is performed in order to determine the workloads for the remaining processor (not including processor 1). If a processor is adjacent to the left of processor 1 (say processor 2), it will receive the "partial" solutions (for example, x_{88} to x_{85}) from all other processors (fan in), say processors 1 and 3. All the other processors (not including processor 1 and the adjacent processor 2) will send the required information to processor 2. Therefore, processor 2 is now ready to compute the "final" solution for x_{88} through x_{85}.

Backward elimination

```
Do 200 i = n, 4, -4

  If ("me" have the i^th row) then    ➔ Ex: row i=92

c.. For 1 processor

   update x(i), x(i-1), x(i-2), x(i-3)
   send partially updated x(i-4), x(i-5), x(i-6), x(i-7) to the next processor
   partially update x(j) (for 1 < j < i-7)

  else
c… For 1 processor (the adjacent processor to "me")
   if ("me" have the (i-4)th column) then
    fan-in (or receive) x(i-4), x(i-5), x(i-6), x(i-7)
   else
c… For all other processors
    send information correspond to row (i-4), (i-5), (i-6) and (i-7) to processor which contains the (i-4)th column
   endif
 endif

200 Continue
```

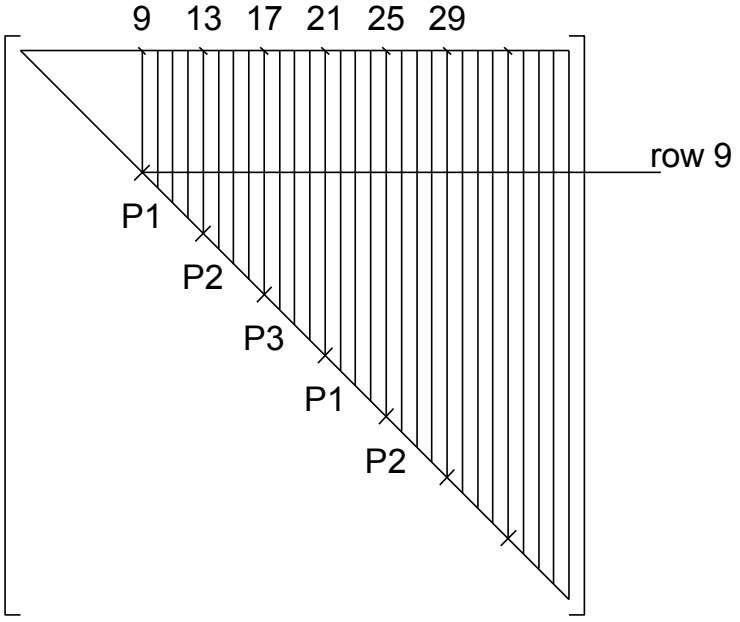


Figure 1. Block storage scheme for matrix A in a one-dimensional array

Table 7A. Comparison of equation solvers
N=16152, nbw=328, using 32 nodes on Intel iPSC/860 Gamma.

	Intel Pro-Solver (SES)	The Present Solver*
Decomposition	51.18	25.85
Forward	9	0.8156
Backward	62 (-10)	0.9401
Total	122.18 (-10)	27.607

* Sequential-send scheme with vector-unrolling level 8.

Table 5. Solving High-Speed Research Aircraft problem on 8, 16 and 32 nodes (Gamma)

Task(s)	Time(seconds)			
	8 nodes	16 nodes	32 nodes	*32 nodes
Factorization	35.9	30.7	28.7	26
Forward Elimination	1.6	1.4	1.4	0.8
Backward Elimination	1.4	1.5	1.7	1
Total	38.9	33.6	31.8	27.8

Notes: Sequential send scheme with vector unrolling level 4
Max. displ. = +0.45; sum displ.=189
* vector unrolling level 8 is used

Table 6. Comparisons of MPFEA on the Gamma and Delta (shaded area) Computers

(750 bays, 6 stories, 18006 els., NEQ=9016, Ave BW=1512)

	8	16	32	64	128	256	512
g/a	-	0.1585	0.0806	0.0403	0.0188	-	-
fac	-	80.2	61.383	57.018	48.081	-	-
forw	-	0.8228	0.6241	0.7751	1.2786	-	-
back	-	0.5411	0.5405	0.6015	0.709	-	-
g/a	0.3203	0.1506	0.0785	0.0377	0.0176	0.0095	0.00463
fac	136.05	75.686	54.854	46.893	40.848	39.044	39.401
forw	1.265	0.795	0.603	0.735	0.596	0.552	0.575
back	0.616	0.428	0.356	0.468	0.399	0.478	0.589

Table 7B. Parallel Performance of MPFEA on 256 processors

Task(s)	CPU time (second)	Mflops
generation/assembly	0.12251	38.46
factorization	662.417	631.56
forward elimination	4.9654	77.94
backward elimination	3.1579	120.16
others (overhead, etc.)	3.4624	-
Total	674.13	621.73

1096 bays, 41 stories, 179785 els., NEQ=89960, Ave BW=2208

To further improve the computational efficiencies, block-wise updating strategies are also employed. A block-wise updating (see Figure 7.2) means there are four rows being concurrently up-dated by multiprocessors. A block-wise updating also means that having completed all 4 columns (say 9, 10, 11 and 12 in Figure 7.1), processors 1 will send all these 4 columns to all other processors.

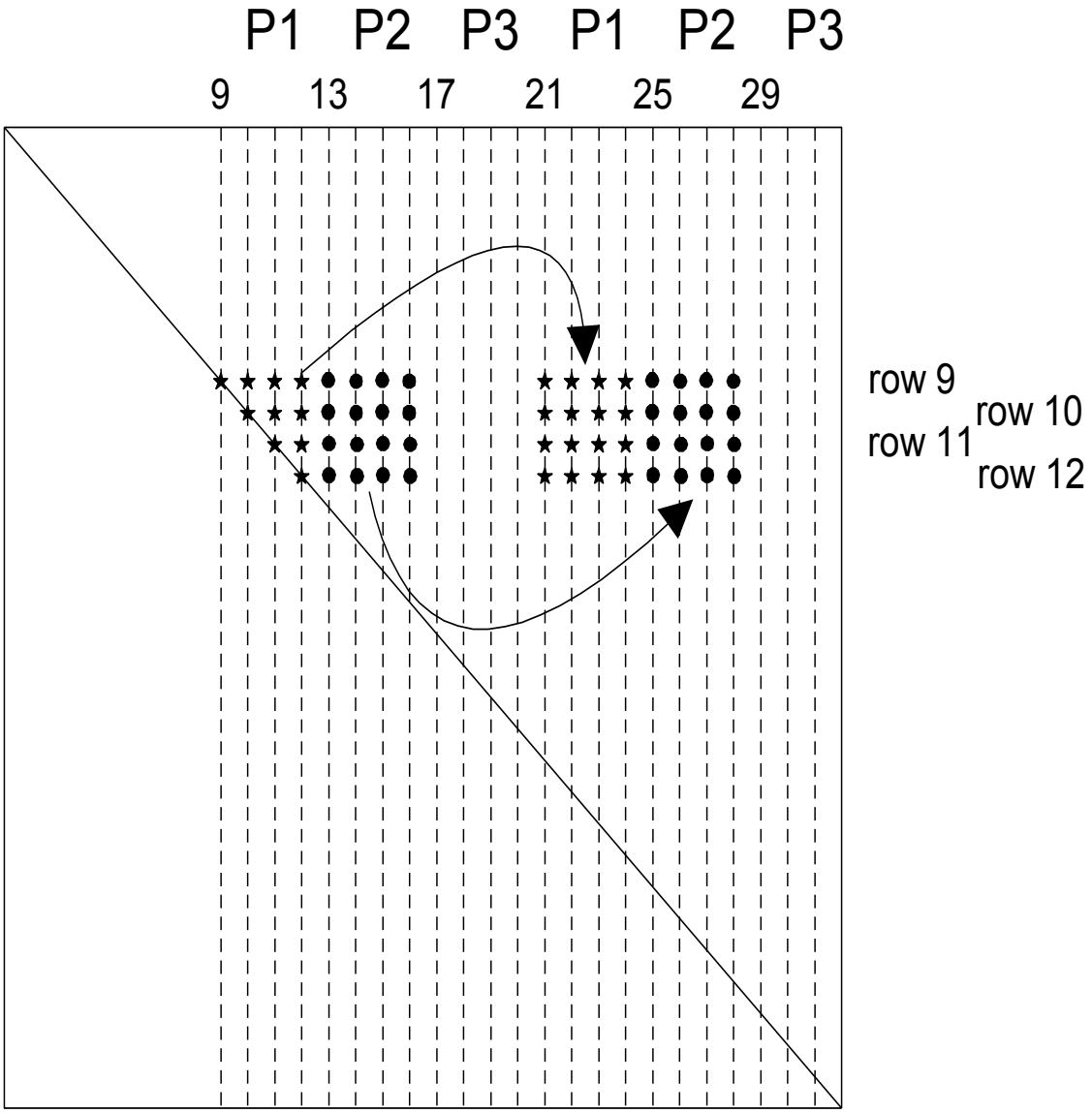


Figure 7.2: Block-wise (rows) factorization for matrix A (in a 1-D array)

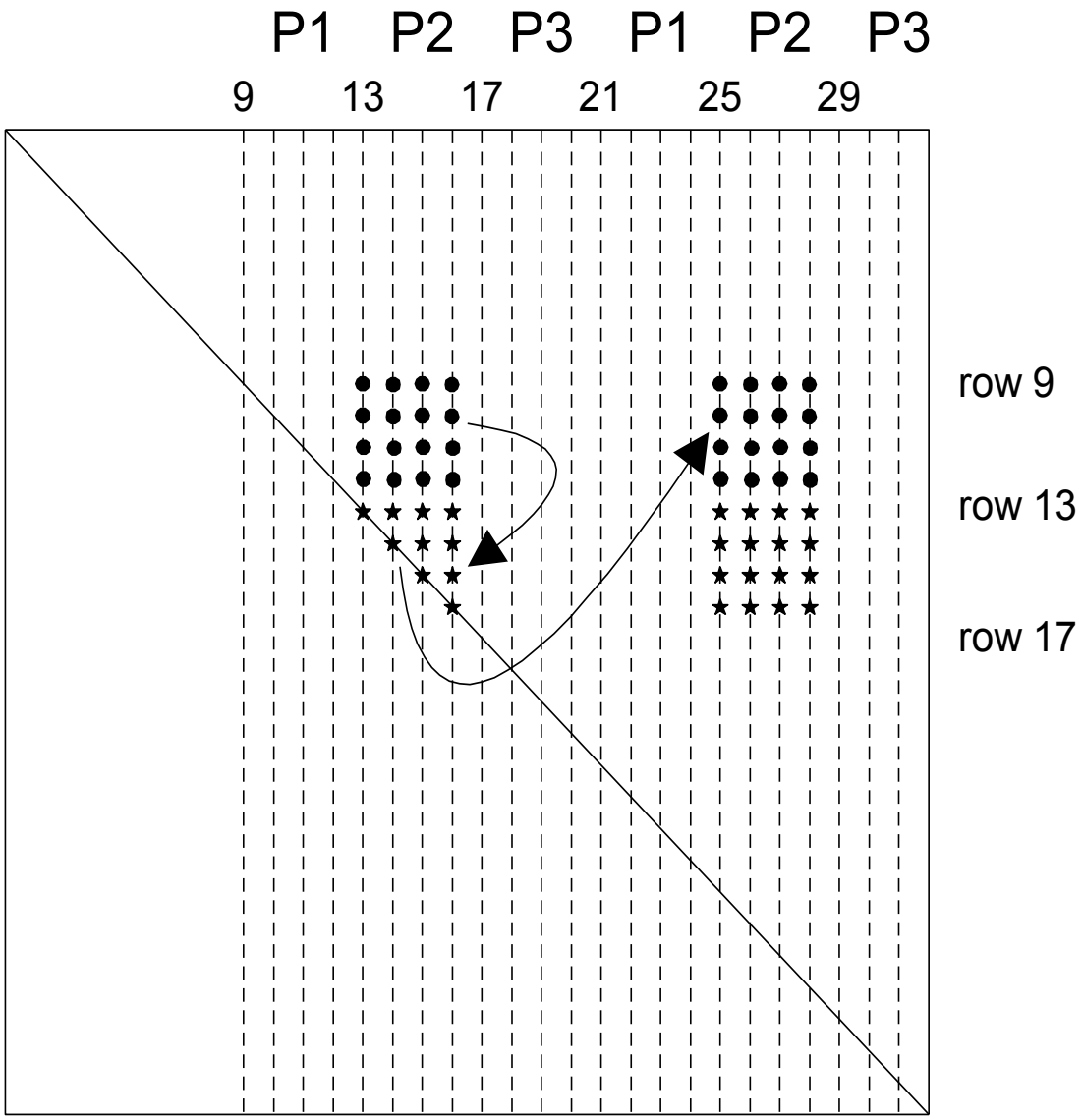


Figure 7.3: “Twice” block-wise (rows) factorization for matrix A (in a 1-D array)

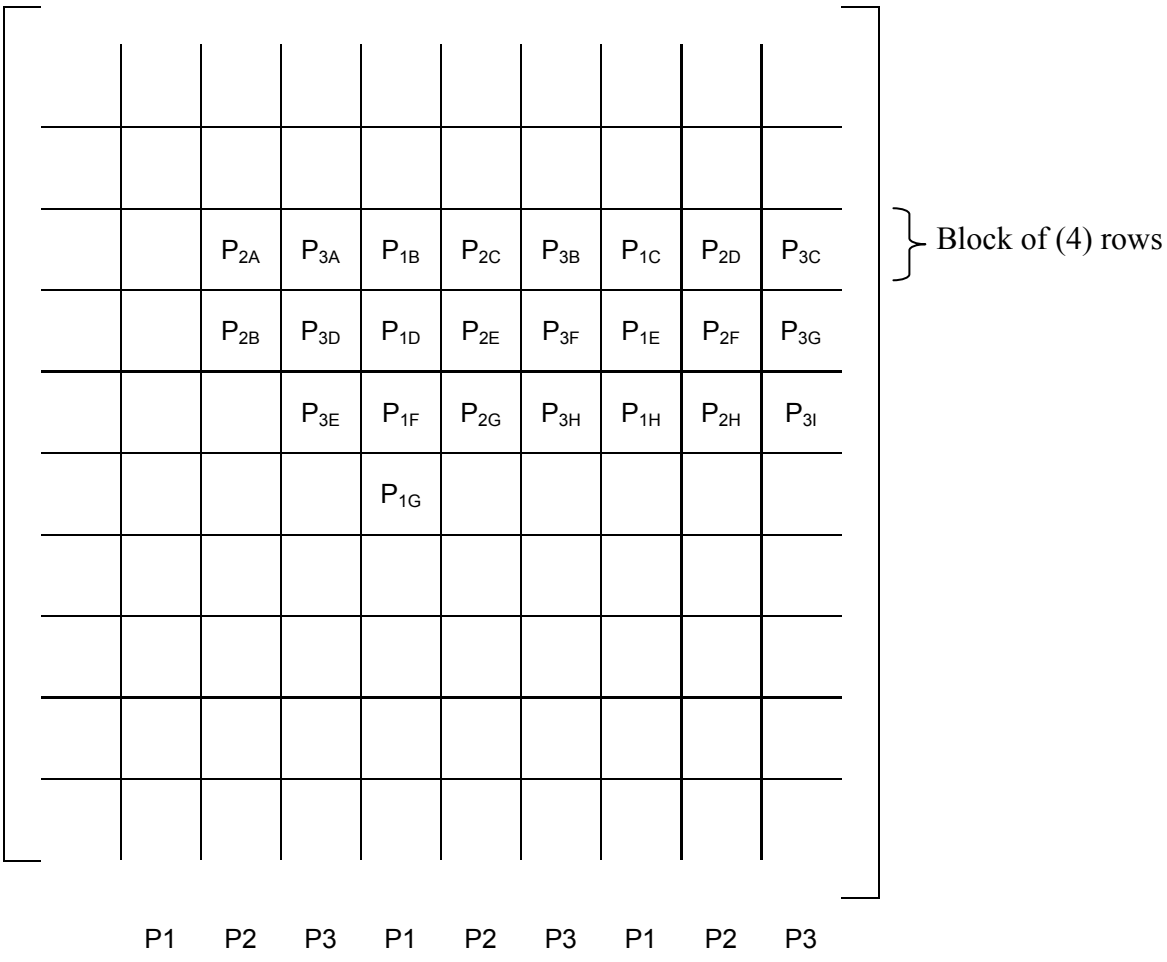


Figure 7.4 Flows of “Twice” Block-wise (Rows) Factorization Algorighm

NEQ=9016

```
 Start ...        2 nodes&nbw=      1512
FORTRAN STOP
 ** Start Time (dec)  =  10.14999961853027
 ** Start Time (for)  =  294.1999816894531
 ** Start Time (bac)  =  313.6099853515628
 ** Finishing Time    =  313.6899902343750
 ** Processor #   =         2
 ** Time (dec.)   =  284.0499820709229
 ** Time (for.)   =  19.41000366210938
 ** Time (bac.)   = 0.3800048828125000
 ** Time(total)   =  303.8399906158447
 ** Mflops(dec)   = -2.049262645025570
 ** Mflops(for)   = 0.5851564182809850
 ** Mflops(bac)   =  29.92435201172502
 ** Mflops(tot)   = -1.840981193299589
 ** # of Max.  =          1
 ** Max. x(i)  =  0.2644967118825578
 ** SUM of x   =   3.930352212734294
 ** Elastic Modulus =   29001.00000000000
 ** Time in boundc  =  1.9999504089355469E-002
 ** Time in jointc  =  1.0000228881835938E-002
 ** Time in apload  =  0.0000000000000000E+000
 ** Time in elconn  =  1.0000228881835938E-002
 ** Time in materp  =  0.0000000000000000E+000
 ** Time in colht   =  0.5399999618530273
 ** Time in gen+file=   1.029999732971191
 TOTAL TIME:(nel,neq,ielm)  1.609999656677246   18006 9016  10969
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

750 bay x 6 story

Gamma&Delta Intel needs
≥ 16&8 nodes, respectively

```
 Start ...       16 nodes&nbw=     1512
 ** Start Time (bac)  =  59.84999847412109
 ** Finishing Time    =  60.31999969482422
 ** Processor #   =         16
 ** Time (dec.)   =  48.63999938964844
 ** Time (for.)   =  3.069999694824219
 ** Time (bac.)   = 0.4700012207031250
 ** Time(total)   =  52.18000030517578
 ** Mflops(dec)   =  20.60519921911640
 ** Mflops(for)   = 0.4681433527961609
 ** Mflops(bac)   =  3.036894164745796
 ** Mflops(tot)   =  19.26219690592816
 ** # of Max.  =          1
 ** Max. x(i)  =  0.2644967831671393
 ** SUM of x   =   3.930355197070884
 ** Elastic Modulus =   29001.00000000000
 ** Time in boundc  =  1.9999980926513672E-002
 ** Time in jointc  =  0.0000000000000000E+000
 ** Time in apload  =  0.0000000000000000E+000
 ** Time in elconn  =  1.9999980926513672E-002
 ** Time in materp  =  9.9997520446777344E-003
 ** Time in colht   =   2.380000114440918
 ** Time in gen+file=  0.2899994850158691
 TOTAL TIME:(nel,neq,ielm)  2.719999313354492   18006 9016  1895
FORTRAN STOP
```

Gamma Intel=80.20 sec => Intel Delta=75.7 sec
Gamma=0.82 sec => Delta=0.79 sec
Gamma=0.54 sec => Delta=0.43 sec

Meiko Parallel Computer (@LLNL) & ODU-Solver

NEQ=9016

750 bay x 6 story

```
Start ...          32 nodes&nbw=     1512
 ** Start Time (dec)  =  8.699999809265137
FORTRAN STOP
 ** Start Time (for)  =  40.20999908447266
 ** Start Time (bac)  =  42.29000091552734
 ** Finishing Time    =  42.95999908447266
 ** Processor #   =         32
 ** Time (dec.)   =  31.50999927520752
 ** Time (for.)   =  2.080001831054688
 ** Time (bac.)   = 0.6699981689453125
 ** Time(total)   =  34.25999927520752
 ** Mflops(dec)   =  16.04534005693415
 ** Mflops(for)   = 0.3510535398250376
 ** Mflops(bac)   =  1.083047734376779
 ** Mflops(tot)   =  14.79989710560952
 ** # of Max.  =          1
 ** Max. x(i)  =  0.2644967831671393
 ** SUM of x   =   3.930355197071425
 ** Elastic Modulus =   29001.00000000000
 ** Time in boundc  =  9.9997520446777344E-003
 ** Time in jointc  =  1.0000228881835938E-002
 ** Time in apload  =  0.0000000000000000E+000
 ** Time in elconn  =  1.9999980926513672E-002
 ** Time in materp  =  9.9997520446777344E-003
 ** Time in colht   =   3.069999694824219
 ** Time in gen+file=  0.1000003814697266
 TOTAL TIME:(nel,neq,ielm)  3.219999790191650   18006 9016  955
FORTRAN STOP
```

Gamma Intel=61.40 sec => Intel Delta=54.9 sec
Gamma=0.78 sec => Delta=0.60 sec
Gamma=0.54 sec => Delta=0.36 sec

Meiko Parallel Computer (@LLNL) & ODU-Solver

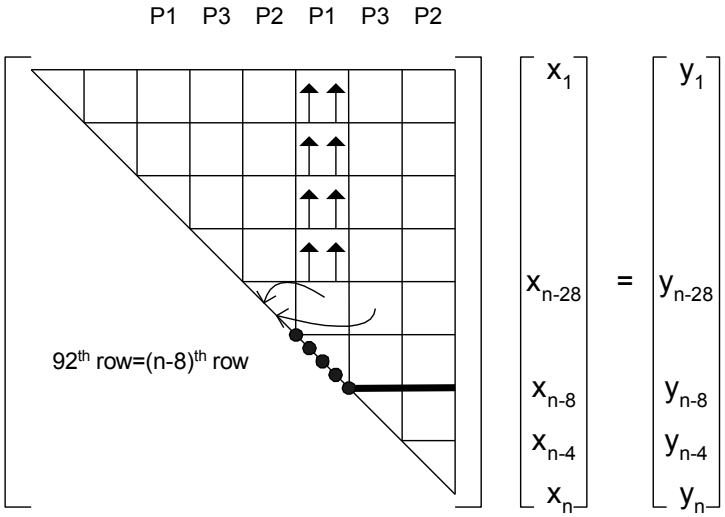


Figure 3: Backward Elimination (“some” Parallel)

Conclusions

- A general, fast, minimum memory requirement equation solver on distributed computers (i.e. Intel iPSC/860) has been developed

- Forward & Backward solution time is quite efficient. Thus, open doors to solutions of eigenvalue, nonlinear, D.S.A., structural dynamics and structural optimization problems.

- The present solver takes advantage of both parallel & vector capabilities, hence, seems to offer very fast solution time on the Intel iPSC/860 (including the practical, NASA focused HSCT finite element model.)

- Ideal parallel speed up obtained for g/a

- Present parallel g/a can be applied to both shared and distributed computers

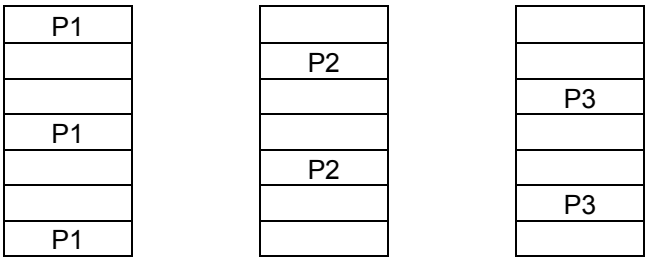


Figure 4: GDSUM is used to merge partial (processor) solution for final solution

NASA Technical Memorandum 102614

A Parallel-Vector Algorithm for Rapid Structural Analysis on High-Performance Computers

Olaf O. Storaasli, Duc T. Nguyen and Tarun K. Agarwal

April 1990

National Aeronautics and Space Administraion

Langley Research Center
Hampton, Virginia 23665-5225

Forcecall PVS(a,b,maxa,irowl,icolh,neq,nterms,iif,opf)

Where:

a = a real vector, dimensioned nterms, containing the coefficients of the stiffness matrix, [K]

b = a real vector dimensioned neq, containing the load vector, {f}. Upon return from subroutine PVS, b contains the displacement solution, {u}.

maxa = an integer vector, dimensioned neq, containing the location of the diagonal terms of [K] in vector{a}, equal to the sum of the number coefficients.

irowl = an integer vector, dimensioned neq, containing the row lengths (i.e., half-bandwidth of each row excluding the diagonal term) of [K].

icolh = an integer vector, dimensioned neq, containing the column heights (excluding the diagonal term) of each column of the stiffness matrix, [K].

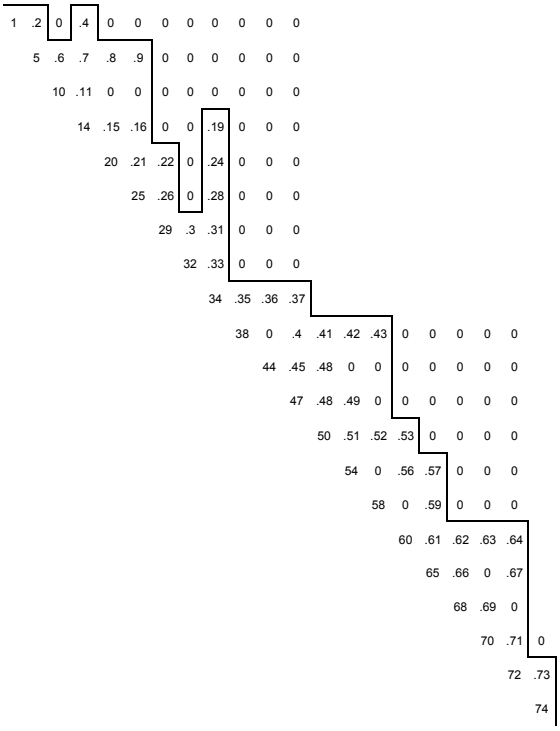
neq = number of equations to solve (= degrees of freedom)

nterms = the dimension of the vector, {a} , [=maxa(neq)].

iif = 1 factor system of equations without internal zero check
= 2 factor system of equations with internal zero check
= 4 perform forward/backward substitution
= 5 perform forward/backward substitution and error check

opf,ops = an integer vector, dimensioned to the number of processors (8 for Cray Y-MP), containing the number of operations performed by each processor during factor and solve, respectively.

For example, the values of these input variables to solve a system of 21 equations, whose right hand side is the vector of real numbers from 1. to 21., and [K] is the symmetric, positive-definite matrix in Fig. B1 are given in Table B1.



i	a(i)	b(i)	maxa(i)	icolh(i)	irowl(i)
1	1.	1.	1	0	11
2	.2	2.	13	1	10
3	0	3.	24	1	9
4	.4	4.	34	3	8
5	0	5.	43	3	7
6	0	6.	51	4	6
7	0	7.	58	2	5
8	0	8.	64	1	4
9	0	9.	69	5	3
10	0	10.	73	1	10
11	0	11.	84	2	9
12	0	12.	94	3	8
13	5.	13.	103	3	7
14	.6	14.	111	4	6
15	.7	15.	118	5	5
16	.8	16.	124	3	4
17	.9	17.	129	3	3
18	0	18.	133	2	2
19	0	19.	136	3	2
20	0	20.	139	4	1
21	0	21.	141	1	0
22	0				
23	0				
24	10.				
25	.11				
26-33	0				
34	14.				
35	.15				
36	.16				
37-38	0				
39	.19				
*	*				
*	*				
*	*				
135	0				
136	70.				
137	.71				
138	0				
139	72.				
140	.73				
141	74.				

```
            Force PVSOLVE of np ident me
            Shared real a(5208900),b(16150),at(499600),opf(8)
      csrb  Shared real a(21090500),b(54890),at(1350761)
            Shared real t0(8),t1(8),t2(8),t3(8),t4(8),t5(8),ops(8)
            Shared real et0(8),et1(8),et2(8),et3(8),et4(8),et5(8)
            Shared integer maxa(16150),irow(16150),irowl(16150)
            shared integer icoln(499600),icolh(16150),nc,neq
            End declarations
              et0(me)=timef()/1000
              t0(me)=second()/np
            if (me.eq.1) then call CSMIN(a,b,maxa,irowl,icolh,neq,
     +        nterms,irow,icoln,nc,maxbw,8,locrow,iavebe)
            write(*,*)'* PVSOLVE - pvsolve -PVSOLVE Mar. 1990'
            write(*,*)'* Parallel-Vector equation SOLVEr by Olaf'
            write(*,*)'* Storaasli, Tarun Agarwal and Duc Nguyen'
            write(*,*)'* ',np,' proc. solve',neq,' equations; nc= ',nc
            write(*,*)'* bandwidth: max= ',maxbw,',avg.= ',iavcbw
            write(*,*)'* [K] matrix size, nterms= ',nterms,' words'
            endif
              et1(me)=timef()/1000.
              t1(me)=second()/np
            Barrier
            End barrier
              et2(me)=timef()/1000.
              t2(me)=second()/np
      call PVS to factor [k] with internal zero check (iif = 2).......
              iif=2
            Forcecall PVS(a,b,maxa,irowl,icolh,neq,nterms,iif,opf(me))
              et3(me)=timef()/1000.
              t3(me)=second()/np
      call PVS to backsolve for {u} (iif=4, 5 error check eqs. 11-13)
              iif = 5
            Forcecall PVS(a,b,maxa,irowl,icolh,neq,nterms,iif,ops(me))
              et4(me)=timef()/1000.
              t4(me)=second()/np
            Barrier
              nat=499600
              umax=abs(b(1))
              do 1 i=1,neq
1              umax=amax1(umax,abs(b(i)))
            write(*,*)'* Maximum displacement = ',umax
            if(iif.eq.5) call NORM(irowl,icoln,b,neq,nc)
c......reorder displacements and write to CSM Testbed........
             call TOCSM(b,irowl,icoln,at,at,icoln,8,nat)
             *
             *
             *
             Join
             end


             Forcesub PVS(a,b,maxa,irowl,icolh,neq,nterms,iif,ops)
     +         of np ident me
             dimension a(*),b(*),icolh(*),maxa(*),irowl(*)
             Async real x(16150)
             End declarations
              if(iif.le.2) then
             Presched do 9 i=1,neq
             Void x(i)
9            End presched do
                ops=0
             Barrier
                a(1) = sqrt(a(1))
                xinv=1.0/a(1)
cdir$  ivdep
              do 20 k=1,irowl(1)
20             a(k+1)=xinv*a(k+1)
               ops=ops+irowl(1)+2
             Produce x(1)=a(1)
             End barrier
c.............factor stiffness matrix in parallel from row 2 to neq
             Presched do 100 i = 2, neq
```

INTEL SOLVER (Skyline storage scheme)

Subroutine Node(nodes, iam, n, nbw, imod, a, z, y, x, maxa, irow, icolg, tem, kflag)

Node = number of processors
Iam = my node id#
n = degree-of-freedom
nbw = maximum bandwidth (include diagonal)
imod = 1 (for real problem)
a = stiffness matrix (dimension=nterms)
z = working array, z(nbw, 8)
y = load vector, y(n)
x = displacement vector, x(n)

maxa = diagonal locations, dimension $\geq \left\{ \left[\frac{\left(\frac{n-1}{8} \right)+1}{nodes} \right] + 2 \right\} * 8$

irow = ith row length (include diagonal) , dimension $\geq \left(\frac{n-1}{8} \right)+1$

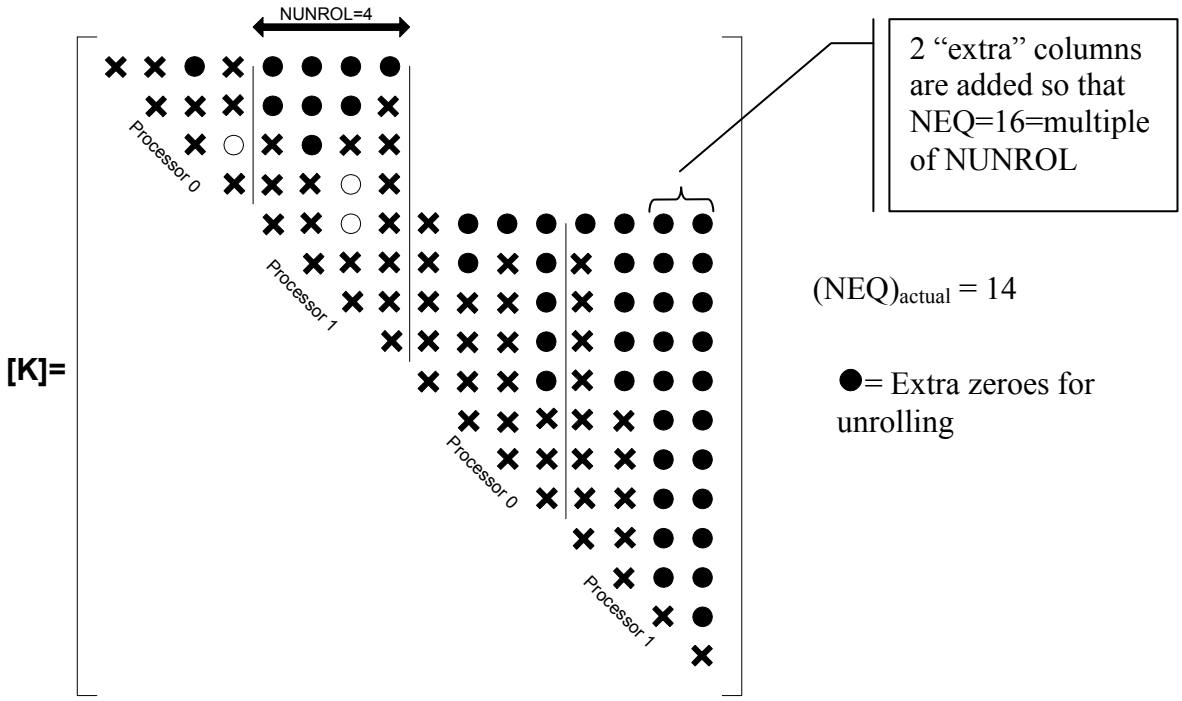
icolg = column height, dimension $\geq \left(\frac{n-1}{8} \right)+1$

tem = real array, dimension ≥ nbw
kflag = 1, for factorization
= else, for forward/backward

(a) All processors need have this info.
(b) Only the info. Of row length, column height of the last row and the last column of each block is needed

Note:
All real arrays are declared as double precisions

Massively Distributed Storage Scheme
For Equation solver



Assuming - $(NEQ)_{actual}$ = 14
- NUNROL = 4
- NP (=No. of Processors) =2=(processor)
- Thus: NEQ=16=(=Multiple of NUNROL)

Notes:

(a) column height = from diagonal upward (include diag. Term)

(b) row-length = include diag. Term

(c) since NUNROL = 4 is used in Equation (Intel) solver, each block (of 4) columns must have same level high => Extra zeroes • need be added.

(d) The last column height in each block (of 4) must be a <u>multiple</u> of <u>NUNROL</u> and must be ≥ NUNROL.

(e) For the above example of [K], max NP=4 (Processor 0, 1, 2, 3), If NP > 4, then we'll have idle processors

(f) We need GLOBAL column height ➔

$$colh\begin{pmatrix} 1 \\ 2 \\ * \\ * \\ \left(\frac{NEQ-1}{NUNROL}\right)+1 \end{pmatrix} = colh\begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \begin{Bmatrix} 4 \\ 8 \\ 8 \\ 12 \end{Bmatrix}$$

Where only Global col. Height of the LAST column in each block (of 4 columns) need be calculated.

(g) We also need Global row-length information

$$IROWL\begin{pmatrix}1\\2\\.\\.\\.\\\frac{NEQ-1}{NUNROL}+1\end{pmatrix} = IROWL\begin{pmatrix}1\\2\\3\\4\end{pmatrix} = \begin{pmatrix}5\\9\\5\\1\end{pmatrix}$$

Where only Global row-length of LAST row in each block (of 4 rows) need be calculated.

(h) We also need Local (for each processor) MAXA information

$$MAXA\begin{pmatrix}1\\2\\3\\.\\.\\\frac{NEQ}{NP}\end{pmatrix} = MAXA\begin{pmatrix}1\\2\\3\\4\\5\\6\\7\\8\end{pmatrix} = \begin{pmatrix}1\\2\\4\\7\\11\\16\\22\\29\end{pmatrix}$$ For Processor 0

And

$$MAXA\begin{pmatrix}1\\2\\3\\4\\5\\6\\7\\8\end{pmatrix} = \begin{pmatrix}1\\6\\12\\19\\27\\36\\46\\57\end{pmatrix}$$ For Processor 1