SIMPLE AND EFFICIENT PARALLEL DENSE EQUATION SOLVER

3. Parallel Data Storage Scheme

Assuming 3 processors are used to solve a system of 15 matrix equations. The matrix will be divided into several parts with, say, 2 columns per blocks (ncb=2). Therefore, from Fig 1, processor 1 will handle column 1, 2, 7, 8, 13 and 14. Also, processor 2 will handle column 3, 4, 9, 10 and 15, and processor 3 will handle column 5, 6, 11 and 12. The columns which belong to each processor will be stored in a one-dimensional array in a column-by-column fashion. For example, the data in row 4 and column 7 will be stored by processor 1 at the 7^{th} location of one-dimensional array A of processor 1. Likewise, each processor will store only portions of the whole matrix [A]. The advantage of this storage scheme is that the algorithm can solve much bigger problem size.

Processor	P1		P2		P3		P1		P2		P3		P1		P2	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	2	1	4	1	6	4	11	8	17	12	23	19	32	27	1
2		3	2	5	2	7	5	12	9	18	13	24	20	33	28	2
3			3	6	3	8	6	13	10	19	14	25	21	34	29	3
4				7	4	9	7	14	11	20	15	26	22	35	30	4
5					5	10	8	15	12	21	16	27	23	36	31	5
6						11	9	16	13	22	17	28	24	37	32	6
7							10	17	14	23	18	29	25	38	33	7
8								18	15	24	19	30	26	39	34	8
9									16	25	20	31	27	40	35	9
10										26	21	32	28	41	36	10
11											22	33	29	42	37	11
12												34	30	43	38	12
13													31	44	39	13
14														45	40	14
15															41	15

Fig. 1: block Columns Storage Scheme

Two more small arrays are also needed to store the starting columns (icolst) and ending columns (icolend) of each block.

Therefore, from this example,

$$icolst_1 := \begin{pmatrix} 1 \\ 7 \\ 13 \end{pmatrix} \qquad icolend_1 := \begin{pmatrix} 2 \\ 8 \\ 14 \end{pmatrix}$$

$$icolst_2 := \begin{pmatrix} 3 \\ 9 \\ 15 \end{pmatrix} \qquad icolend_2 := \begin{pmatrix} 4 \\ 10 \\ 15 \end{pmatrix}$$

$$icolst_3 := \begin{pmatrix} 5 \\ 11 \end{pmatrix} \qquad icolend_3 := \begin{pmatrix} 6 \\ 12 \end{pmatrix}$$

It should be noted that sizes of arrays {icolst} and {icolend} are the same for the same processor, but may be different from the other processors. In fact, it depends on how many blocks to be assigned to each processor (noblk). For this example, processor 1 and 2 each stores 3 blocks of the matrix, and processor 3 stores 2 blocks of the matrix.

4. Data Generating Subroutine

The stiffness matrix, stored in a one dimensional array {A} and the right-hand-side load vector {b} will be automatically generated such that the solution vector {x} will be 1 for all values of {x}. Also, from equation 4, the diagonal values of the stiffness matrix should be large to avoid the negatives value in the square root operations. The general formulas to generate the stiffness matrix is

$$a_{i,i} = 50000 \cdot (i + i)$$

$$a_{i,j} = i + j$$

, and the formula for RHS vector can be given as

$$b_i = \sum_{j=1}^{n} a_{i,j} \ .$$

5. Parallel Choleski Factorization

Assuming the first four rows of the matrix A (see Figure 1) have already been updated by multiple processors, and row 5 is currently being updated. Thus according to Figure 1, terms such as $u_{5,5}$… $u_{5,6}$ and $u_{5,11}$… $u_{5,12}$ are processed by processor 3. Similarly, terms such as $u_{5,7}$… $u_{5,8}$ and $u_{5,13}$… $u_{5,14}$ are handled by processor 1, while terms such as $u_{5,9}$, $u_{5,10}$ and $u_{5,15}$ are executed by processor 2.

As soon as processor 3 completely updated column 5 (or more precisely, updated the diagonal term $u_{5,5}$, since the terms $u_{1,5}$ $u_{2,5}$ … $u_{4,5}$ have already been factorized earlier), it will send the entire column 5 (including its diagonal term) to all other processors. Then processor 3 will continue to update its other terms of row 5. At the same time, as soon as processors 1 and 2 receive column 5 (from processor 3), these processors will immediately update its own terms of row 5.

To enhance the computational speed, by using the optimum available cache, the "scalar" product operations (such as $u_{ki}*u_{kj}$) involved in Eq. (3) can be replaced by "sub-matrix" product operations. Thus, Eq. (3) can be re-written as:

$$[u_{ij}] = [u_{ii}]^{-1} \cdot \left([a_{ij}] - \sum_{k=1}^{i-1} [u_{ki}]^T [u_{kj}] \right) \qquad (9)$$

Similar "sub-matrix" expressions can be used for Eqs. (4, 6, 8).

6.1 Loop indexes and temporary array usage

Fortran uses column-major order for array allocation. In order to get a stride of one on most of the matrices involved in the triple do loops of the matrix-matrix multiplication, one needs to interchange the indices as follows:

```
Do j=1,n
  Do i=1,m
    Do k=1,l
      C(i,j)=c(i,j)+A(i,k)*B(k,j)
    ENDDO
  ENDDO
ENDDO
```

Note that in the above equation the order of the index loop have been rearranged to get a stride of 1 on matrix C and B. However, the stride on matrix A is M. In order to have a stride of 1 during the computation on matrix A, a temporary array is used to load portion of matrix A before computations.

6.2 Blocking and strip mining

Blocking is a technique to reduce cache or misses in nested array processing by calculating in blocks or strips small enough to fit in the cache. The general idea is that an array element brought in is processed as fully as possible before it is flushed out.

To use the blocking technique in this basic matrix-matrix multiplication, the algorithm can be rewritten by adding three extra outer loops (JJ, II and KK) with an increment equal to the size of the blocks.

```
Do JJ=1,n,jb
  Do II=1,m,ib
    Do KK=1,l,kb
      Do J=JJ,min(n,JJ+jb-1)
        Do I=II,min(m,II+ib-1)
          Do K=KK,min(l,KK+kb-1)
            C(i,j)=c(i,j)+A(i,k)*B(k,j)
          ENDDO
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

Where ib, jb and kb are the block sizes for i, j, and k respectively and they are estimated function of the available cache size for different computer platforms.

6.3 Unrolling of loops

Unrolling of loops is considered at various stages. To illustrate the idea, let's consider the inner do loop (see the index k) where the actual multiplication is performed. Here a four by four submatrix is considered at the time, and the inner do loop of Eqs. (3-4) can be unrolled in the following fashion:

```
DO j=1, number of J blocks
  DO i=1, number of I blocks
    DO k=1, number of k blocks
      C{i,j} = C{i,j} + T{1,1} * B{k,j}
      C{i+1,j} = C{i+1,j} + T{1,2} * B{k,j}
      C{i,j+1} = C{i,j+1} + T{1,1} * B{k,j+1}
      C{i+1,j+1} = C{i+1,j+1} + T{1,2} * B{k,j+1}
      C{i,j+2} = C{i,j+2} + T{1,1} * B{k,j+2}
      C{i+1,j+2} = C{i+l,j+2} + T{1,2} * B{k,j+2},
      C{i,j+3} = C{i,j+3} + T{1,1} * B{k,j+3}
      C{i+l,j+3} = C{i+l,j+3} + T{1,2} * B{k,j+3}
      C{i+2,j} = C{i+2,j} + T{1,1} * B{k,j}
      C{i+3,j} = C{i+3,j} + T{1,2} * B{k,j}
      C{i+2,j+1} = C{i+2,j+1} + T{1,1} * B{k,j+1}
      C{i+3,j+1} = C{i+3,j+1} + T{1,2} * B{k,j+l}
      C{i+2,j+2} = C{i+2,j+2} + T{1,1} * B{k,j+2}
      C{i+3,j+2} = C{i+3,j+2} + T{1,2} * B{k,j+2}
      C{i+2,j+3} = C{i+2,j+3} + T{1,1} * B{k,j+3}
      C{i+3,j+3} = C{i+3,j+3} + T{1,2} * B{k,j+3}
    ENDDO
  ENDDO
ENDDO
```

Where T is the temporary array, which contains portions of the matrix [A].

6. A Blocked and Cache Based Optimized Matrix-Matrix Multiplication

Let's consider matrix C(m,n) to be the product of two dense matrices A and B of dimension (m,l) and (l,n), respectively.

$$C = A \cdot B \tag{10}$$

A basic matrix-matrix multiplication algorithm consists of a triple nested do loops as follows:

```
Do i=1,m
  Do j=1,n
    Do k=1,l
      c(i,j)=c(i,j)+a(i,k)*b(k,j)
    ENDDO
  ENDDO
ENDDO
```

Further optimization can be applied on this basic algorithm to improve the performance of the multiplication of two dense matrices. The following optimization techniques are used for the matrix-matrix multiplication subroutine.

- Reordering of loop indices
- Blocking and strip mining
- Loop unrolling on the considered sub matrices
- Stride minimization
- Use of temporary array and left over computations.

Table 2: Two Thousand (2,000) Dense Equations (Sun 10k)

n=2000	Factorization time				Forward Substitution				Backward Substitution			
	ncb=16	32	64	128	ncb=16	32	64	128	ncb=16	32	64	128
1 processor	16.243	9.896	8.552	8.337	0.176	0.173	0.173	0.175	0.110	0.097	0.098	0.097
2	7.637	5.170	4.873	5.108	0.087	0.040	0.066	0.261	0.236	0.083	0.048	0.039
4	3.736	3.028	3.010	3.548	0.028	0.034	0.044	0.194	0.109	0.046	0.028	0.026
8	2.634	2.076	2.470	3.186	0.025	0.029	0.047	0.210	0.057	0.027	0.020	0.021
16	1.766	1.569	1.723		0.034	0.049	0.095		0.038	0.020	0.016	

Table 3: Five Thousand (5,000) Dense Equations (Sun 10k)

n=5000	Factorization time				Forward Substitution				Backward Substitution			
	ncb=16	32	64	128	ncb=16	32	64	128	ncb=16	32	64	128
1 processor	277.5	182.2	143.3	130.4	1.108	1.110	1.105	1.153	0.775	0.784	0.784	0.776
2	140.0	93.8	76.8	72.2	0.582	0.554	0.545	0.620	1.553	0.826	0.623	0.582
4	73.4	51.3	42.4	44.3	0.300	0.287	0.293	0.338	0.781	0.424	0.319	0.336
8	40.8	27.6	25.5	29.4	0.170	0.162	0.208	0.435	0.445	0.209	0.164	0.183
16	42.4	23.0	25.08	22.0	2.685	0.760	0.260	0.284	0.737	0.114	0.084	0.125

Table 4: Ten Thousand (10,000) Dense Equations (Sun 10k)

n=10000	Factorization time				Forward Substitution				Backward Substitution			
	ncb=16	32	64	128	ncb=16	32	64	128	ncb=16	32	64	128
1 processor	2284.2	1548.6	1203.9	1059.2	4.565	4.881	4.541	4.548	3.317	3.560	3.323	3.302
2	1178.8	774.3	596.8	548.4	2.450	2.360	2.240	2.244	6.410	3.578	2.590	2.385
4	595.2	397.4	316.7	306.2	1.244	1.187	1.133	1.229	3.310	1.764	1.354	1.264
8	307.1	211.3	176.0	180.6	0.640	0.600	0.643	0.839	1.759	0.920	0.700	0.687
16	161.3	114.0	105.6	120.2	0.395	0.401	0.610	1.060	0.989	0.490	0.386	0.416

Table 5: Matrix-Matrix Multiplication Subroutines

	LIONS (Helios)			LIONS(Cancun)	NASA SGI/Origin-2000
	Regular version (seconds)	Cache version (seconds)	Cache Version (seconds)	Cache Version (seconds)	Cache Version (seconds)
1000x1000	177.405*	6.391*	6.420†	10.9†	9.5†
2000x2000	1588.03*	53.13*	54.02†	N/A	N/A

* using MPI_WTIME( ) † using etime( )

PARALLEL DENSE SOLVER

(a) Using CANCUN (sun 10k), and solving 5000 dense, sym. Equations, on 1 processor:

1. Using Todd+Duc's parallel dense solver + Sun DGEMM (fast matrix*matrix), /bin/time = 311 seconds
2. Using Todd+Duc's parallel dense solver + Pierrot's (fast matrix*matrix), /bin/time = 254 seconds

(b) Using CANCUN (Sun 10k), and solving 1000 dense, sym. Equations, on 1 processor:

1. Using SUN's dense, sym. Solver + Sun's fast matrix*matrix, /bin/time = 2.90 seconds
2. Using Todd+Duc's parallel dense solver + Pierrot's (fast matrix*matrix), /bin/time = 2.30 seconds

++++

August 29, 2001

Please find below the preliminary results that running interactively on Origin (SGI) computer. The reported results are runs from the 64 processors SGI/2000 running 300 MHZ IP27 R12000 Processors.

The results of solving real symmetric positive matrix A using the SGI scientific math library SCS uses the following routines:

DPOTRF – compute the Cholesky factorization of a real symmetric positive definite matrix A with calls to DPOTRF and DPOTRS

DPOTRS – solve a system of linear equations [A] x = B with a symmetric positive definite matrix A using the Cholesky factorization $A = [U]^T [U]$ or $A = [L] [L]^T$ computed by DPOTRF.

These routines are part of the SCSL Scientific Library and can be loaded using either the –lscs or the –lscs mp option. The –lscs mp option directs the linker to use the multi-processor version of the library

Table 1

N = 5000

Serial SGI Math library routines.

f77 –Ofast=IP27 dpotrf.f –lscs	f77 –O3 dpotrf.f –lscs
Factorization Time = 82.16	82.21
Solution Time = 3.38	3.33
Total Time = 85.55	85.54
/bin/time = 1:27:76	1:27:39

Table 2

N = 5000

Multiprocessors SGI Math Library routines.

OMP_NUM_THREADS	1	2	4	8
Factorization time	87.43	48.77	23.77	15.77
Solution Time	4.20	4.56	4.41	4.81
Total Time	91.63	53.33	28.18	20.58
/bin/time	1:34:06	0:56:09	0:30:00	0:22:80

Table 5

Results of the MPI Dense Solver

N = 5000

f77 –Ofast=IP27 mpi-dense2.f -lmpi

f77 –Ofast=IP27 mpi-dense3-blas.f -lscs –lmpi

Origin 2000: 64 300 MHZ IP27 Processors

CPU: MIPS R12000 Processors Chips

Interactive NCB = 64

Number of Procs	1	2	4	8
mpi-dense3.f (use BLAS)				
- Factorization	117.98	65.41	35.65	21.57
- Forward Substit.	0.34	1.43	0.26	0.23
- Backward Substit	0.66	0.55	0.25	0.19
- /bin/time	2:01:91	1:09:18	0:38:64	0:24:42

Table 6: Two Thousand (2,000) Dense Equations (NASA SGI/Origin-2000)

Matrix order = 2,000

Relative error norm = 5.2×10^{-13}

Error norm check time = 1.12 sec

RAM = 49 MB

No. Proc.	1	2	4	8	10	16
Total (sec)	8.12	4.16	2.22	1.63	1.42	1.22
Factor (sec)	7.90	4.15	2.15	1.36	1.37	1.19
Forward/ Backward (sec)	0.22	0.01	0.07	0.27	0.05	0.03

Table 7: Five Thousand (5,000) Dense Equations (NASA SGI/Origin-2000)

Matrix order =5,000

Relative error norm =3.5×10^{-12}

Error norm check time =12.26 sec

RAM = 210 MB

No. Proc.	1	2	4	8	10	16
Total (sec)	106.58	56.65	33.45	22.53	20.60	18.23
Factor (sec)	103.14	53.88	31.75	19.05	16.24	12.85
Forward/ Backward (sec)	3.44	2.77	1.70	3.48	4.36	5.38

Table 8: Ten Thousand (10,000) Dense Equations (NASA SGI/Origin-2000)

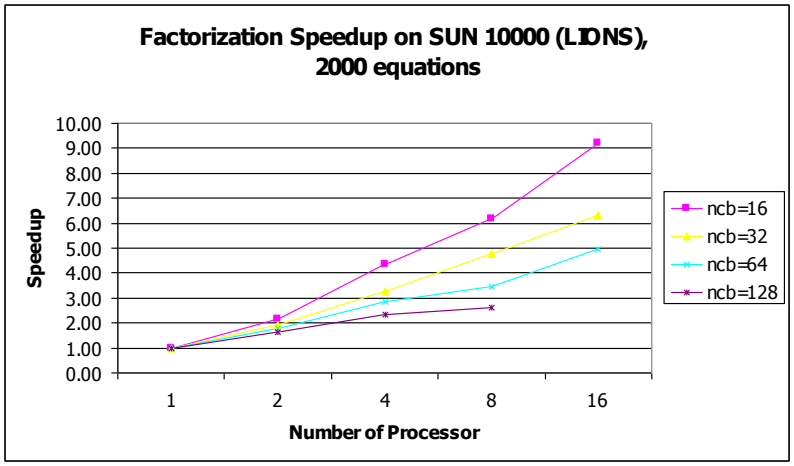
Matrix order =10,000

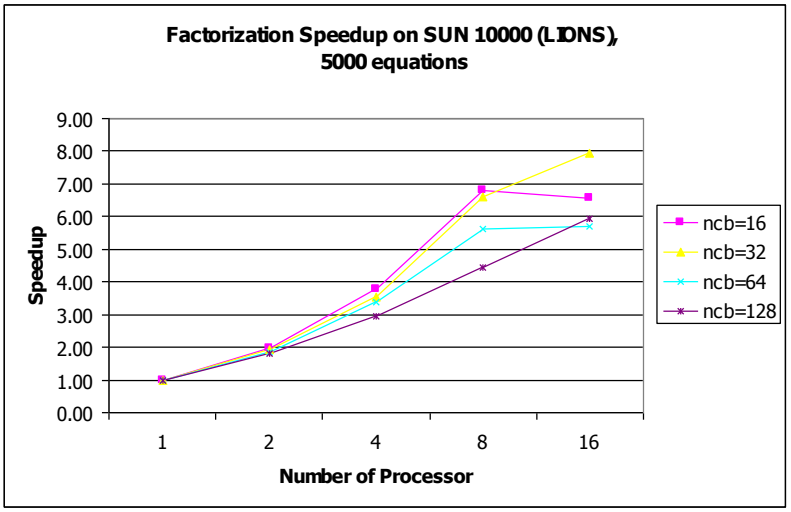
Relative error norm =1.3x10^{-11}

Error norm check time =50.48 sec

RAM =767 MB

No. Proc.	1	2	4	8	10	16
Total (sec)	815.43	432.96	246.96	151.45	130.50	147.38
Factor (sec)	798.65	404.01	212.84	117.72	111.92	80.73
Forward/ Backward (sec)	16.78	28.95	34.12	33.73	18.58	66.65





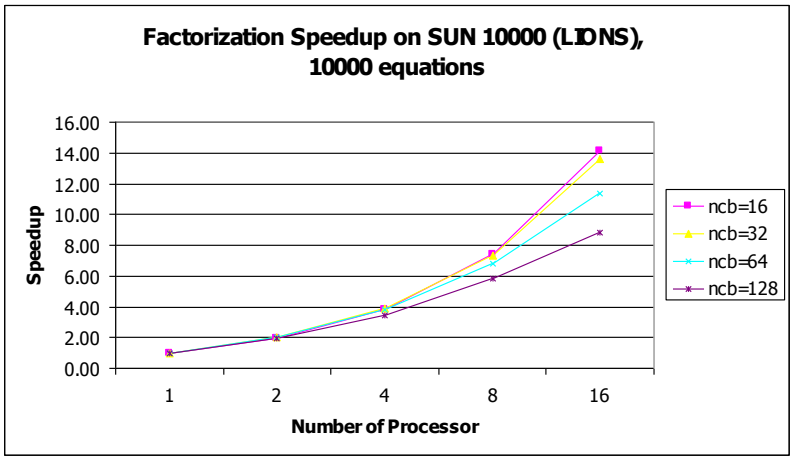
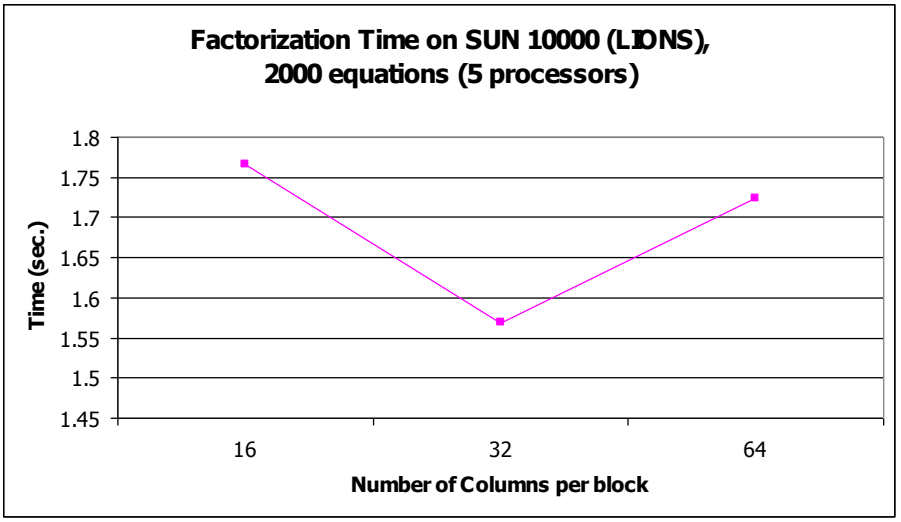
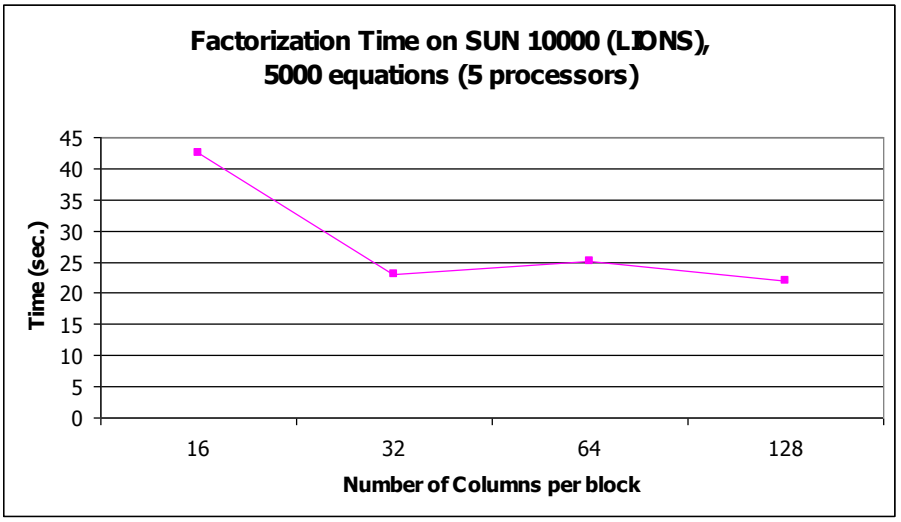


Figure 7, 8 and 9: Factorization Speedup on SUN 10000 (LIONS) for different problem size.





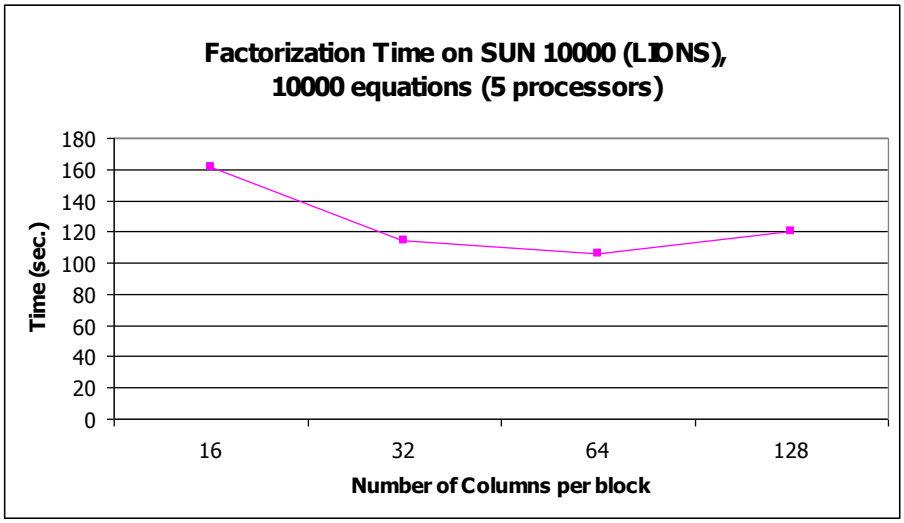


Figure 10, 11 and 12: the relationship between numbers of columns per block and factorization time (using 5 processors) for different problem size.

ELSEVIER

Advances in Engineering Software 31 (2000) 599-606

ADVANCES IN ENGINEERING SOFTWARE

www.elsevier.cornllocateladvengsott

Subspace and Lanczos sparse eigen-solvers for finite element structural and electromagnetic applications

D.T. Nguyen[b,*], C.F. Bunting[b], K.J. Moeller[a], H.Runesha[b], J. Qin[b]

[a] *NASA Langley Research Center, Hampton, VA, USA*
[b] *135 kauf. CEE Department, Old Dominion University, Norfolk, VA, USA*

Abstract

Recent developments in vectorized sparse technologies are incorporated into the Subspace and Lanczos iteration algorithms for computational enhancements. Numerical evaluations of the proposed sparse strategies for Subspace and Lanczos iteration algorithms are conducted by solving the generalized eigen-value problem associated with the Exxon-off-shore structure, HSCT aircraft and a two-dimensional reverberation chamber for electromagnetic applications. Both real and complex numbers eigen-problems can be treated.

Keywords: Subspace and Lanczos sparse eigen-solvers: Exxon-off-shore structure: High speed civil transport aircraft

Fig. 2.1 Lanczos Method for the Solution of the Generalized Eigenproblem

$$[K][\Phi] = [M][\Phi][w^2]$$

1. Given Mass and Stiffness Matrices [M] and [K] with:
   - [M] — n x n system size
   - [K] — n x n
2. Triangularized Stiffness Matrix:
   - $[K] = [L]\,[D]\,[L]^T$ — n x n system
3. Choose an Arbitrary Starting vector {X}:
   - $b = (\{X\}^T [M]\,\{X\})^{1/2}$ — M-Normalization
   - $\{X_1\} = \{X\} * 1/b$ — Vector one
4. Solve for Additional Vectors with b_1=0 and i=2,…,r:
   a. $[K]\{\overline{X_i}\} = \{M\}\{X_{i-1}\}$ — Solve for $\{\overline{X_i}\}$
   b. $a_{i-1} = \{\overline{X_i}\}^T [M]\{X_{i-1}\}$
   c. $\{\boxed{X}_i\} = \{\overline{X}_i\} - a_{i-1}\{X_{i-1}\} - b_{i-1}\{X_{i-2}\}$ — M-Orthogonalized
   d. $b_i = (\{\boxed{X}_i\}^T [M]\{\boxed{X}_i\})^{\frac{1}{2}}$ — M-Normalization

   $$\{X_i\} = \{\boxed{X}_i\} \cdot \frac{1}{b_i}$$

5. Construct symmetric Tridiagonal Matrix [T] of order r:
   (typically r=2m where m is the number of requested eigenvalues)

$$[T_r] = \begin{bmatrix} a_1 & b_2 & 0 & . & . & 0 \\ b_2 & a_2 & b_3 & . & . & . \\ 0 & b_3 & a_3 & . & . & . \\ 0 & 0 & . & . & . & 0 \\ . & . & . & b_{r-1} & a_{r-1} & b_r \\ o & . & . & . & b_r & a_r \end{bmatrix}$$

6. Calculate Eigenvalues and Eigenvectors of $[T_r]$:

   $[T_r]\,[z] = [z]\,[\lambda]$

   $[w_2] = [1/\lambda]$

7. Expand Eigenvectors to Full System Size:

   $$[\Phi] = [X][Z]$$

Fig. ?? Lanczos Method (Qin-Nguyen's Implementation) for the Solution of the Generalized Eigenproblem $[K][\Phi]=[M][\Phi][w^2]$

I. Given Mass and Stiffness Matrices [M] and [K] with:

[M]	n x n system size
[K]	n x n

II. Triangularized Stiffness Matrix:

$[K] = [L]\,[D]\,[L]^T$	n x n system

III. Choose an Arbitrary Starting vector {X}:

$b = (\{X\}^T [M]\,\{X\})^{1/2}$	M-Normalization
$\{X_1\} = \{X\} * 1/b$	Vector one

IV. Solve for Additional Vectors with b1=0 and i=2,…,r(r=LANMAX steps):

a. $[K]\{\overline{X_i}\} = \{M\}\{X_{i-1}\}$ Solve for $\{\overline{X_i}\}$

b. $a_{i-1} = \{\overline{X_i}\}^T [M]\{X_{i-1}\}$

c. $\{\overline{\overline{X}_i}\} = \{\overline{X_i}\} - a_{i-1}\{X_{i-1}\} - b_{i-1}\{X_{i-2}\}$ M-Orthogonalized (w.r.t. previous 3 Lanczos vectors)

d. $b_i = (\{\overline{\overline{X}}_i\}^T [M]\{\overline{\overline{X}}_i\})^{\frac{1}{2}}$ M-Normalization

$\{X_i\} = \{\overline{\overline{X}}_i\} \cdot \frac{1}{b_i}$

e. Re-orthogonalization (w.r.t. much earlier Lanczos vectors, if necessary
For any new Lanczos vector X_i, calculate

$E_j = X_j^T M X_i$ (where j = 1,2,…,i-1)

Semi-Orthog. Condition

if $E_j > \sqrt{\varepsilon_{mp}}$ then (note: ε_{mp} = machine precision)
$\overrightarrow{X_i}$ should be orthogonal to $\overrightarrow{X_j}$ with respect to [M]
v.i.a. Grahmsmith Re-orthog. process
Endif

$$u_i = w_i - \sum_{k=1}^{i-1}\left(\frac{u_k^T w_i}{u_k^T u_k}\right)\cdot u_k$$
where $1 \le i \le p$

f. Construct symmetric Tridiagonal Matrix [T] of order r:
(typically r=3m ➔ 4m where m is the number of requested eigenvalues)
Assume m = 100

$$[T_r] = \begin{bmatrix} a_1 & b_2 & 0 & . & . & 0 \\ b_2 & a_2 & b_3 & . & . & . \\ 0 & b_3 & a_3 & . & . & . \\ 0 & 0 & . & . & . & 0 \\ . & . & . & b_{r-1} & a_{r-1} & b_r \\ o & . & . & . & b_r & a_r \end{bmatrix}$$

g. "Occasional" calculate Eigenvalues and Eigenvectors of $[T_r]$:

$$[T_r]\,[z] = [z]\,[\lambda]$$

$$[w_2] = [1/\lambda]$$

Notes:

1. First time to solve the above reduced tri-diag. eigen system when, say i = 2m (= say 200)
2. Convergence check:

$$Error(j) = \frac{(w_k^2)_{exact} - (\lambda_j)_{computed}}{(\lambda_j)_{computed}} < errtol \quad \text{(where j = 1,2,…,m)}$$

$$Error(j) \approx \left| \frac{b_{i+1} Z_i^{(j)}}{(\lambda_j)_{computed}} \right| < errtol$$

3. Assuming 80 (out of a 100 requested) eigen-values are converged.
Compute NEVNC ≡Number of Eigen Values Not Converged = m-80 = 100-80 = 20
4. Second time to solve the Tri-diag. eigen system will be at (2*NEVNC) iterations later (assuming i+2*NEVNC < r ≡LANMAX steps)
Go back to step IV (until all requested "m" eigenvalues converged)
5. If we want to obtain, say 100 eigenvalues, above the value w^2 = say 5, then we'll apply the shift value σ of ± 5 (depends on $\hat{K} = K \pm \sigma M$)

V. Expand Eigenvectors to Full System Size:

$$[\Phi] = [X][Z]$$

Journal of Computers and Structures, Vol. 39 (1-4), 1991, pp. 277-290

AN ALGORITHM FOR DOMAIN DECOMPOSITION IN FINITE ELEMENT ANALYSIS

Moayyad Al-Narsa[1]
Duc T. Nguyen[2]

Department of Civil Engineering
Old Dominion University
Norfolk, VA 23529

Abstract

A simple and efficient algorithm is described for automatic decomposition of an arbitrary finite element domain into a specified number of subdomains for finite element, and substructuring analysis in a multi-processor computer environment. The algorithm is designed to balance the work loads, to minimize the communication among processors and to minimize the bandwidths of the resulted system of equations. Small to large scale finite element models, which have 2-node elements (truss, beam element), 3-node elements (triangular element) and 4-node elements (quadrilateral element), are solved on the Convex computer to illustrate the effectiveness of the proposed algorithm. A FORTRAN computer program is also included.

[1] Graduate Student

[2] Assistant Professor, Member ASCE

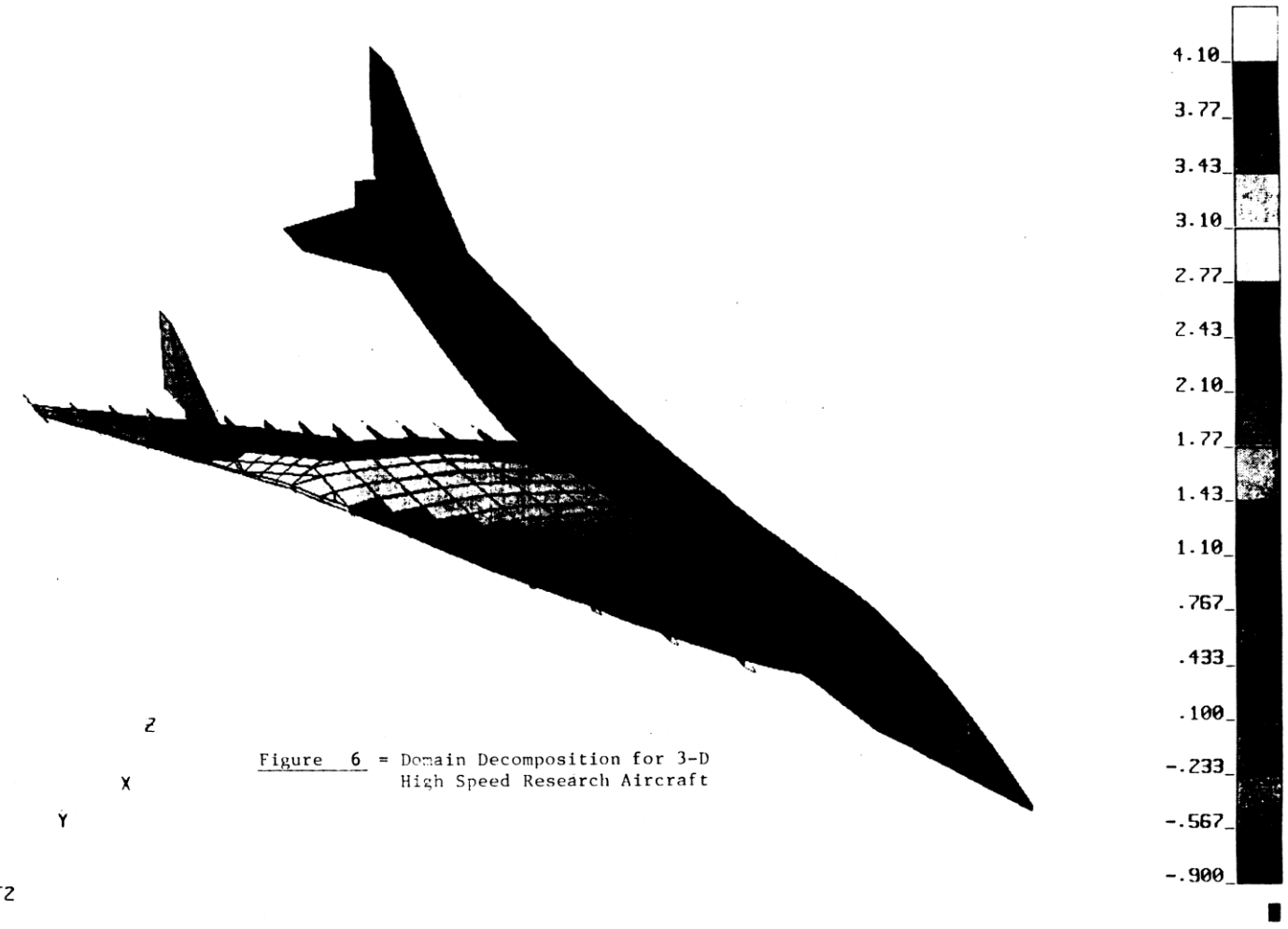
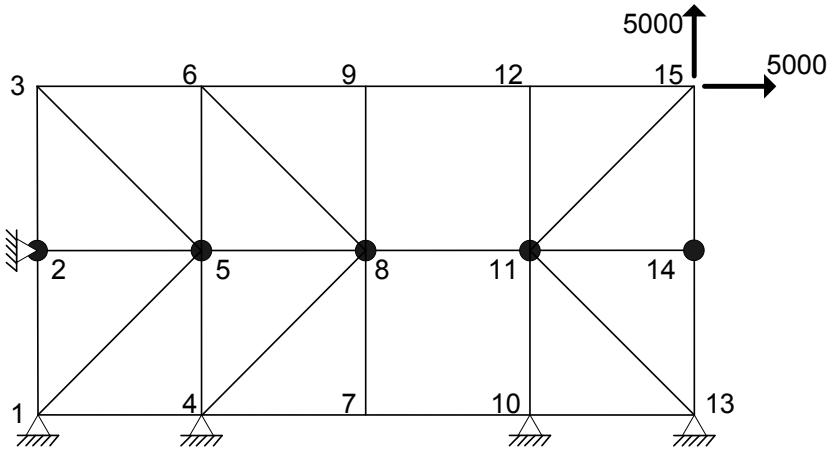
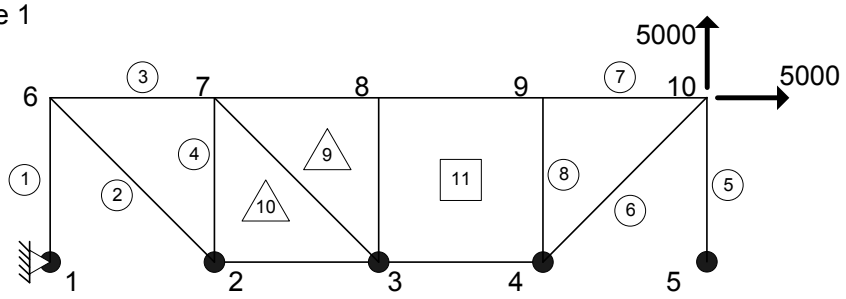


Figure 6 = Domain Decomposition for 3-D
High Speed Research Aircraft

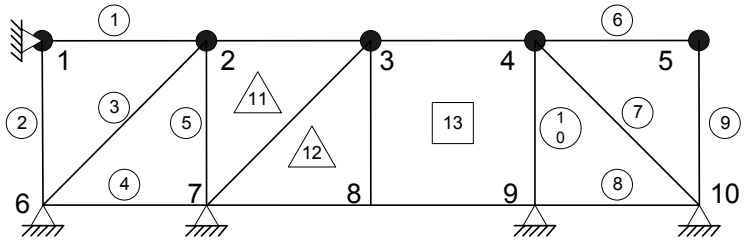


Substructure 1



nbj=5

Substructure 2



nbj=5

● indicates boundary joint. From the particular example, the support displacements at joint 1 of both substructures are set to be 1 unit in both directions

Prepared and revised by Siroj Tungkahotara, 03/31/02

$$Kb1 := Kbb1 - Kbi1 \cdot Kii1^{-1} \cdot Kbi1^{T}$$

$$Kb2 := Kbb2 - Kbi2 \cdot Kii2^{-1} \cdot Kbi2^{T}$$

$Kb1 + Kb2 =$

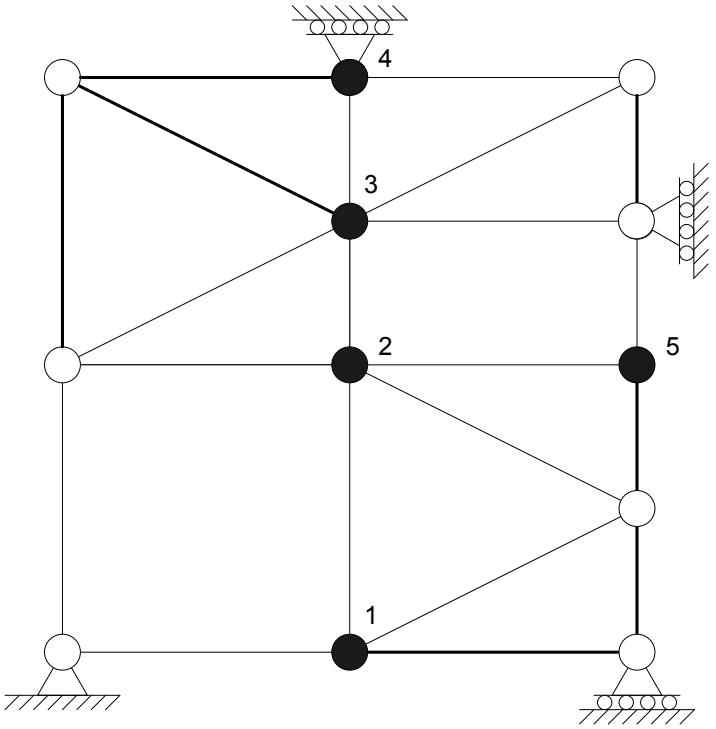
	0	1	2	3	4	5	6	7	8	9
0	2	0	0	0	0	0	0	0	0	0
1	0	2	0	0	0	0	0	0	0	0
2	0	0	67.841322	94.292882	43.934522	58.10239	-0.800132	0.497665	0.258638	0.640343
3	0	0	94.292882	120.744443	58.10239	72.270257	0.497665	1.795462	0.640343	1.022048
4	0	0	43.934522	58.10239	83.941876	99.686389	-21.48125	-20.482945	-2.267449	-1.97383
5	0	0	58.10239	72.270257	99.686389	115.430902	-20.482945	-19.48464	-1.97383	-1.68021
6	0	0	-0.800132	0.497665	-21.48125	-20.482945	79.34148	118.799296	22.404213	27.880041
7	0	0	0.497665	1.795462	-20.482945	-19.48464	118.799296	158.257112	27.880041	33.35587
8	0	0	0.258638	0.640343	-2.267449	-1.97383	22.404213	27.880041	61.428025	79.803269
9	0	0	0.640343	1.022048	-1.97383	-1.68021	27.880041	33.35587	79.803269	98.178512

$$fb := \left(fb1 - Kbi1 \cdot Kii1^{-1} \cdot fi1\right) + \left(fb2 - Kbi2 \cdot Kii2^{-1} \cdot fi2\right)$$

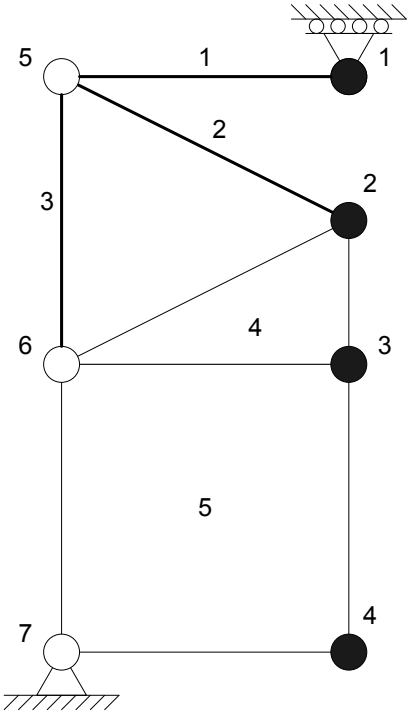
$fb =$

	0
0	2
1	2
2	390.420259
3	411.268283
4	313.536043
5	325.444779
6	-525.661936
7	-497.865029
8	-1599.462992
9	-1583.052137

```
Job <25028> is submitted to queue <hpc>.
<<Waiting for dispatch ... >>
<<Starting on helios.lions.odu.edu>>
connecting to execution hosts ...
pConnect(): connected to host helios.
Creating application tasks ...
started a task on node helios: tid=<1> pid=<27530> cmdline<a.out>.
started a task on node helios: tid=<2> pid=<27531> cmdline<a.out>.

me= 1

akw1   1.0000000000000     1.0000000000000      19.000000000000      37.000000000000
       32.999999999999     43.999999999999      10.5199999999996     33.520000000000
       27.000000000000     42.000000000000

a      0. 0.  67.841321545127      0. 0.  94.292882147336     120.74444274955      0.
       0.     -0.80013195939117 0.49766483749046  -21.481250040921   -20.482944812550
       79.341479823079 0. 0.      0.49766483749047      1.7954616343721
       -20.482944812550    -19.484639584180    118.79929610983       158.25711239659

fb     2.0000000000000     2.0000000000000     390.42025853215       411.26828271090
       313.53604263902     325.44477918676     -525.66193559126      -497.86502854948
       -1599.4629922168    -1583.0521372045

processor 1 finished the job

me= 0

akw1   1.000000000000      1.0000000000000      371.42025853215       374.26828271090
       280.53604263902      281.44477918677      -536.18193559126      -531.38502854948
       -1626.4629922168     -1625.0521372045

a      2.0000000000000      0.     2.0000000000000 0. 0.       43.934522425569
       58.102389555473      83.941875795671      0. 0.  58.102389555473
       72.270256685377      99.686388972520      115.43090214937 0. 0.
       0.25863816992643     0.64034311018573     -2.2674487140398     -1.9738295292249
       22.404213067491      27.880041387125      61.428025003102 0.  0.
       0.64034311018572     1.0220480504450      -1.9738295292249     -1.6802103444101
       27.880041387125      33.355869706758      79.803268626524      98.178512249945

fb     2.0000000000000     2.0000000000000     390.42025853215       411.26828271090
       313.53604263902     325.44477918676     -525.66193559126      -497.86502854948
       -1599.4629922168    -1583.0521372045

processor 0 finished the job
Note: Nonstandard floating-point mode enabled
See the numerical Computation Guide, ieee sun(3M)
Note: Nonstandard floating-point mode enabled
See the numerical Computation Guide, ieee sun(3M)
```



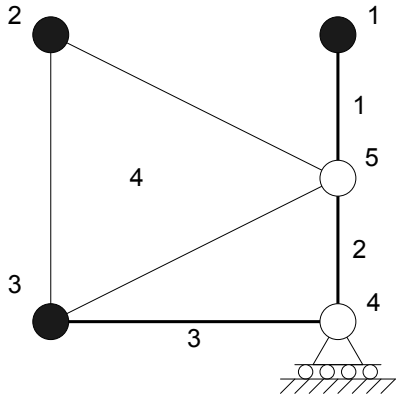
Entire Structure



Substructure 1

Element connectivity for Substructure 1

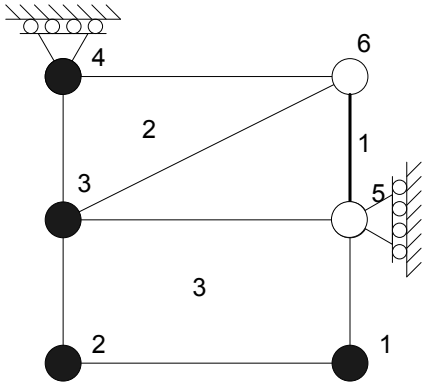
Element	Node i, j, k and m
1	1,5
2	5,2
3	5,6
4	2,6,3
5	3,4,7,6



Substructure 2

Element connectivity for Substructure 2

Element	Node i, j and k
1	1,5
2	5,4
3	3,4
4	2,5,3



Substructure 3

Element connectivity for Substructure 3

Element	Node i, j, k and m
1	6,5
2	6,4,3
3	3,2,1,5

$\text{cholesky(Kball)}^{T} =$

616.441383	0.1427164	0.0713466	0.0859419	$-4.939932 \cdot 10^{-5}$	$-6.0437491 \cdot 10^{-5}$	$3.4711043 \cdot 10^{-10}$	0	$-2.4375457 \cdot 10^{-6}$	$-2.9861135 \cdot 10^{-6}$
0	707.1067481	0.0749093	0.0876298	$-4.7153985 \cdot 10^{-5}$	$-5.7695319 \cdot 10^{-5}$	$3.3134638 \cdot 10^{-10}$	0	$-2.3754024 \cdot 10^{-6}$	$-2.9103539 \cdot 10^{-6}$
0	0	616.4413706	0.1491757	0.0583874	0.0697407	$9.5175261 \cdot 10^{-10}$	0	0.0389244	0.0437902
0	0	0	734.846875	0.0584923	0.0680137	$8.8210974 \cdot 10^{-10}$	0	0.0367263	0.040807
0	0	0	0	479.5831314	0.1188044	0.0333492	0	0.0375155	0.0437693
0	0	0	0	0	583.0951539	0.0308504	0	0.035991	0.041133
0	0	0	0	0	0	264.5751193	0	$-8.9242067 \cdot 10^{-6}$	-0.0000103
0	0	0	0	0	0	0	1.4142136	0	0
0	0	0	0	0	0	0	0	399.999986	0.089969
0	0	0	0	0	0	0	0	0	447.213571

$\text{solution} := \text{Kball}^{-1} \cdot F$

$\text{solution} =$

	0
0	1.0002629744
1	1.0002258395
2	1.0002497246
3	1.0001997744
4	1.0001786213
5	1.0001349271
6	1.0002365601
7	0
8	1.0002247989
9	1.000194819

Boundary Displacements

$$nbjglobdof1 := \begin{pmatrix} 7 \\ 8 \\ 5 \\ 6 \\ 3 \\ 4 \\ 1 \\ 2 \end{pmatrix} \quad nbjglobdof2 := \begin{pmatrix} 9 \\ 10 \\ 3 \\ 4 \\ 1 \\ 2 \end{pmatrix} \quad nbjglobdof3 := \begin{pmatrix} 9 \\ 10 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{pmatrix} \quad i := 0..7 \quad j := 0..5$$

$$zb1_i := solution_{(nbjglobdof1_i - 1)} \quad zb2_j := solution_{(nbjglobdof2_j - 1)} \quad zb3_i := solution_{(nbjglobdof3_i - 1)}$$

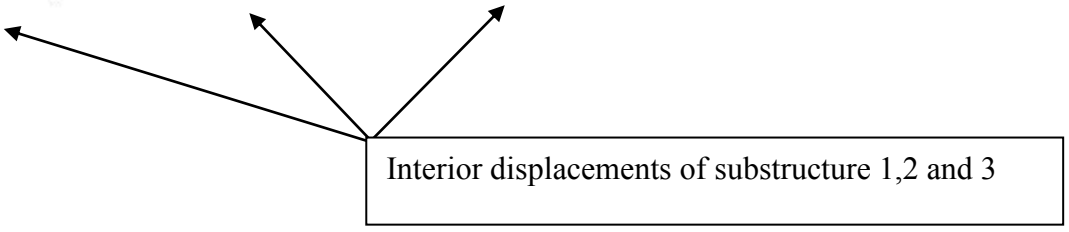
$$zb1 = \begin{pmatrix} 1.0002366 \\ 0 \\ 1.0001786 \\ 1.0001349 \\ 1.0002497 \\ 1.0001998 \\ 1.000263 \\ 1.0002258 \end{pmatrix} \quad zb2 = \begin{pmatrix} 1.0002248 \\ 1.0001948 \\ 1.0002497 \\ 1.0001998 \\ 1.000263 \\ 1.0002258 \end{pmatrix} \quad zb3 = \begin{pmatrix} 1.0002248 \\ 1.0001948 \\ 1.0002497 \\ 1.0001998 \\ 1.0001786 \\ 1.0001349 \\ 1.0002366 \\ 0 \end{pmatrix}$$

$$z1 := Kii1^{-1} \cdot (fi1 - Kbi1^{T} \cdot zb1)$$

$$z2 := Kii2^{-1} (fi2 - Kbi2^{T} \cdot zb2)$$

$$z3 := Kii3^{-1} (fi3 - Kbi3^{T} \cdot zb3)$$

$$z1 = \begin{pmatrix} 1.0625526 \\ 1.0357472 \\ 1.0002208 \\ 1.0001961 \\ 0 \\ 0 \end{pmatrix} \quad z2 = \begin{pmatrix} 1.0002333 \\ 0 \\ 1.0000586 \\ 1.0000499 \end{pmatrix} \quad z3 = \begin{pmatrix} 0 \\ 1.0001856 \\ 1.0004664 \\ 1.0002831 \end{pmatrix}$$



```
Job <28431> is submitted to queue <hpc>.
************************************
 me=   0
 ************************************
 Factorized Matrix
    616.44138302700   0.14271635132720    707.10674805977    3.4711043269346D-1
0
    3.3134637513534D-10    9.5175260723522D-10    8.8210973644533D-10
    3.3349194521292D-02    3.0850387191883D-02    264.57511931093  0.  0.  0.
  0.  0.  0.  0.    1.4142135623731
 Boundary Displ.
    1.0002629743647    1.0002258394546    1.0002497246297    1.0001997743869
    1.0001786212623    1.0001349270555    1.0002365600700  0.    1.000224798938
6
    1.0001948189830
 interior displ. of r-th substructure =
    1.0625525530814    1.0357472315382    1.0002207563602    1.0001960556702
  0.  0.
 processor  0 finished the job
 ************************************
 me=   1
 ************************************
 Factorized Matrix
    7.1346624935313D-02    7.4909314811260D-02    616.44137055781
    8.5941875032465D-02    8.7629778800567D-02   0.14917566112798
    734.84687498689   -2.4375456911889D-06   -2.3754023652774D-06
    3.8924400750449D-02    3.6726303242222D-02    3.7515537313216D-02
    3.5990956735679D-02   -8.9242066523707D-06  0.    399.99998600365
   -2.9861134694282D-06   -2.9103539242623D-06    4.3790182374182D-02
    4.0806977764713D-02    4.3769276792737D-02    4.1133031295348D-02
   -1.0311968207420D-05  0.    8.9968957550343D-02    447.21357103527
 Boundary Displ.
    1.0002629743647    1.0002258394546    1.0002497246297    1.0001997743869
    1.0001786212623    1.0001349270555    1.0002365600700  0.    1.000224798938
6
    1.0001948189830
 interior displ. of r-th substructure =
    1.0002332832602  0.    1.0000586451764    1.0000498528336
 processor  1 finished the job
 ************************************
 me=   2
 ************************************
 Factorized Matrix
   -4.9399320103486D-06   -4.7153985477711D-06    5.8387354744480D-02
    5.8492267825622D-02    479.58313142022   -6.0437490990576D-06
   -5.7695318799826D-06    6.9740674894121D-02    6.8013725340238D-02
   0.11880441378090    583.09515393801
 Boundary Displ.
    1.0002629743647    1.0002258394546    1.0002497246297    1.0001997743869
    1.0001786212623    1.0001349270555    1.0002365600700  0.    1.000224798938
6
    1.0001948189830
 interior displ. of r-th substructure =
  0.    1.0001855525591    1.0004663533058    1.0002831277396
 processor  2 finished the job
```

2.7. Developing/Debuging Parallel MPI Application Code on Your Own LapTop

A Brief Description of MPI/Pro:

MPI/Pro is a commercial MPI middleware product. MPI/Pro optimizes time to solution for parallel processing applications.

MPI/Pro supports the full Interoperable Message Passing Interface (IMPI). IMPI allows the user to create heterogeneous clusters, which gives the user added flexibility while creating their cluster.

Verari Systems Software offers MPI/Pro on a wide variety of operating systems and interconnects, including Windows , Linux, and Mac OS X, as well as Gigabit ethernet, Myrinet , and InfiniBand.

Web site:

http://www.mpi-softtech.com/products/

Contact Information:

Telephone:

Voice: +1-(205) 397-3141
Toll Free: +1-(866) 851-5244
Fax: +1-(205) 397-3142

Sales support:

sales@mpi-softtech.com

Technical support:

mpipro-sup@mpi-softtech.com

Cost:

- $100 per processor (8 processor minimum), one time payment
- Support and Maintenance is required for the first year at a rate of 20% per annum of the purchase price.
- Support ONLY includes:
  - NT4SP5 and higher, 2000 Pro, XP Pro, and Windows 2003 Server.

Steps To Run Mpi/Pro on Windows OS

MPI/Pro requires Visual Studio 98 to run parallel FORTRAN code on Windows OS. Visual Studio 98 supports both FORTRAN and C programming languages. MPI is usually written in C programming language, therefore Visual Studio suffices user's needs.

A. Open Fortran Compiler Visual Studio

B. Open Project

Open a new "project" as a *Win 32 Console Application* . Give a name to your project, i.e. ***project1*.**
This will create a folder named "project1".

C. Create a Fortran File

Create a new fortran file under the project folder you have just created, i.e. *project1.for*, and when you create a file, click on the option "*Add to Project*".

Then, type your program in this file.If you have other files or subroutines to link to this file, do as follows:

- Create a file or files under the folder "*project1*",
- Then, from the pulldown menu, go to Project/ Add to Project and select 'Files'
- Here, select the files to be linked one by one and add them to your project.

D. Parallelizing your own program with MPI/Pro

There are total four call statements to type in a program running sequentially. But, first of all one has to type the following line at the very beginning of the code

include 'mpif.h'

Then, right after defining the data types and chatacter strings, the following three call statements need to be typed

```
call MPI_INIT (ierr)
call MPI_COMM_RANK (MPI_COMM_WORLD, me,ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, np,ierr)
```

where,

me = the ID of each processor *(for example, me 0, 1, 2, 3, 4, 5 if six processors are used)*
np = total number of processors
ierr = error message

In order to finalize the procedure of parallel programming, another call statement has to be placed at the end of the program;

call MPI_FINALIZE (ierr)

E. A Simple Example of Parallel MPI/FORTRAN Code

The entire list of the "trivial" MPI code is shown in the following table:

```
implicit real*8(a-h,o-z)
include 'mpif.h'
real*8 a(16)

call MPI_INIT (ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, me, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)

if (me .eq. 0) then
open(5,file='rectdata.txt')
read(5,*) b,h
write(6,*) me,' b,h',b,h
endif

do i = 1,np
a(i) = (me+i)*2
enddo

write(6,*) 'me=',me,' a(-)',(a(i),i=1,np)

call sub1(me)
call sub2(me)

call MPI_FINALIZE(ierr)

stop
end
```

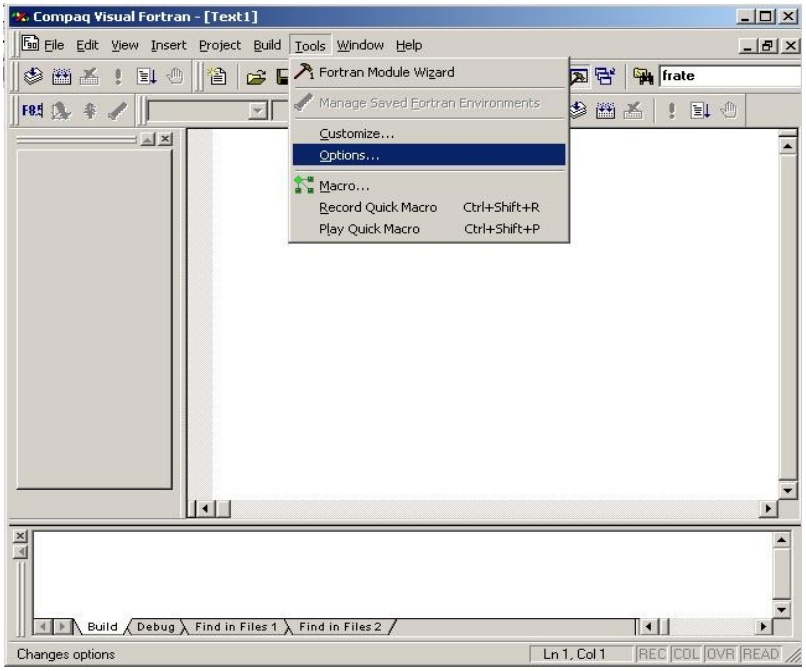
As you can see, the are two subroutine calls in the trivial MPI code. This subroutines are added to the project as decribed in Section C. The subroutines created for this example is below:

```
subroutine sub1(me)
write(6,*) me,' is in sub1 subroutine'
return
end

subroutine sub2(me)
write(6,*) me,' is in sub2 subroutine'
return
end
```

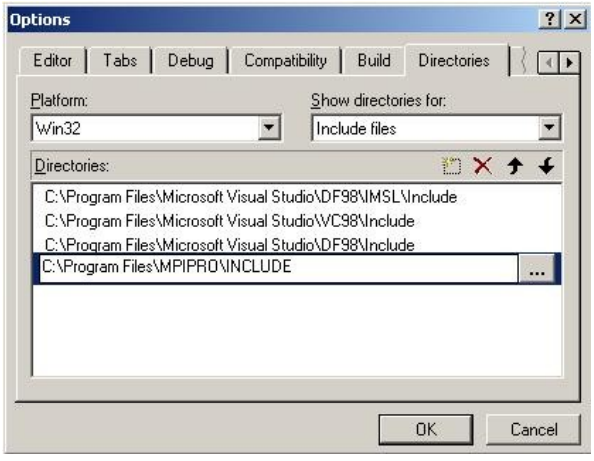
F. Before Compilation

User makes some modifications in the FORTRAN compiler, before he/she compiles his/her code. These modifications include defining the paths for the directory in which the file *'mpif.h'* is stored and for the library files required by MPI/Pro.

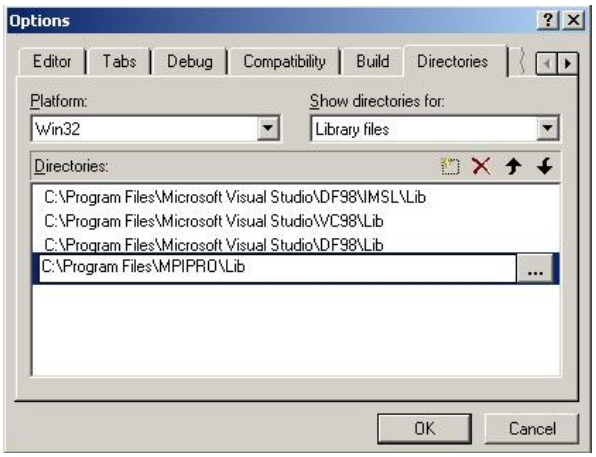
1.From the pulldown menu, go to Tools , then click on Options



It will open up Options window, click on Directories. Under 'Show directories for' submenu, the path for MPI/Pro include file is defined

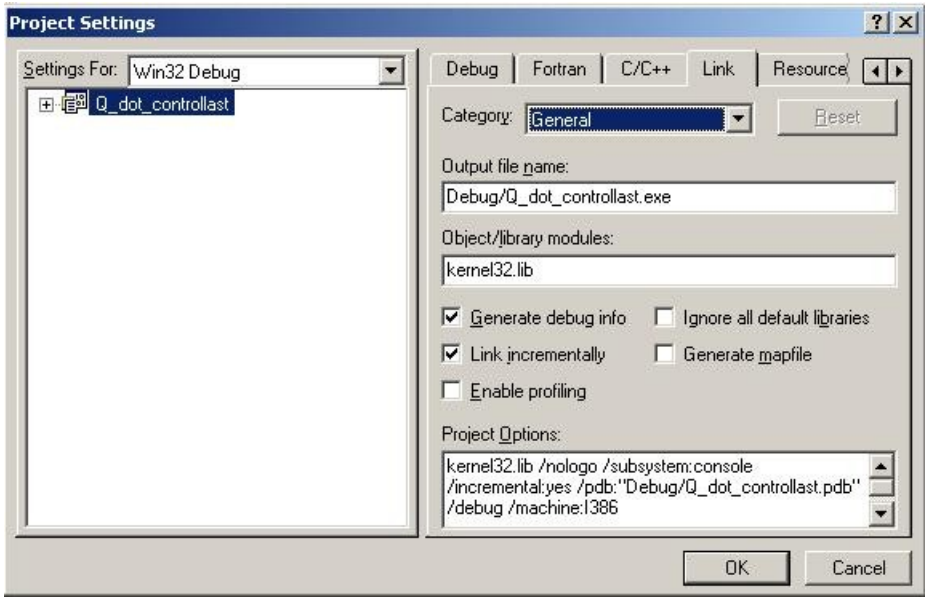


The same procedure is repeated for MPI/Pro library files:



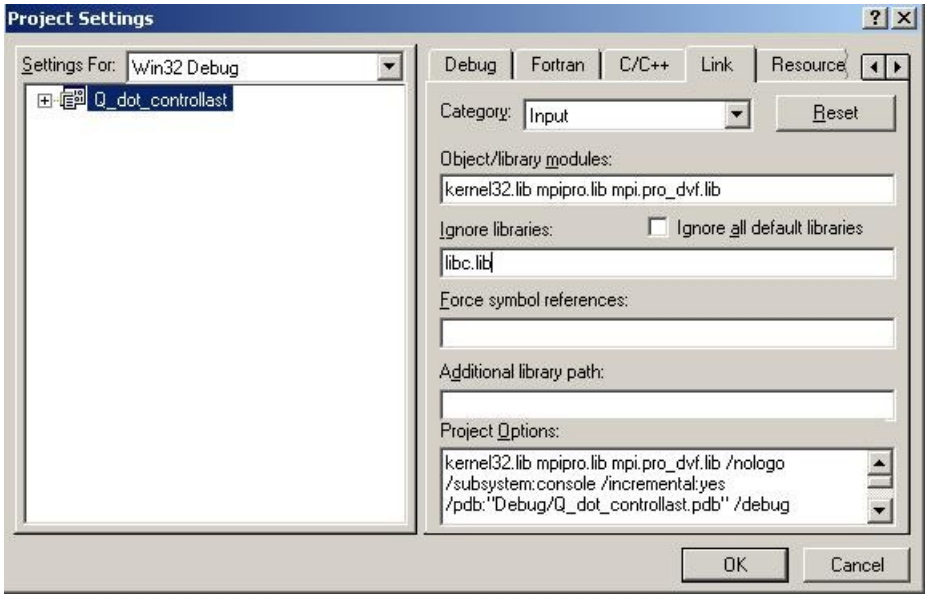
2. From the pulldown menu, go to Project , then click on Settings;

Here, choose Link and then General for Category



Add MPI/Pro libraries *mpipro.lib* and *mpipro_dvf.lib* to 'Object/libraries' modules. Then, switch the Category to Input:

Type *libc.lib* in the Ignore libraries section as in figure below:



G. Compilation

As the next step, the user is supposed to compile his/her program and create the executable file of it. To do this,

- From the pulldown menu, go to Build, then select 'Compile' to compile the MPI code. When you do this, a folder named "Debug" will be created.
- Go to Build again, and select Build to create the executable of the MPI code. Executable file will be located under the folder "Debug"

H. Specifying the computer name(s)

In order to specify the machine name, create a file named '*machines*' under the same directory where the executable file exists. Type in the computer name as your processor for parallel processing.

I. Running parallel

1. Open up the window for command prompt. To do this, goto Windows Start/Run and type cmd or cmd.exe. This will run command prompt.

2. In command prompt, go to the folder "Debug", where you have created the executable file.

3. Then type ;

mpirun –np number of processor executable file name

J. Outputs of the Simple Demonstrated MPI/FORTRAN Code

Assuming six processors are used to "simulate" the parallel MPI environment on the "LapTop" personal computer, using the MPI code discussed in Section E. The results obtained from MPI/FORTRAN application code are shown in the following paragraphs:

```
me=          1  a(-)   4.00000000000000       6.00000000000000
  8.00000000000000       10.0000000000000       12.0000000000000
  14.0000000000000
        1  is in sub1 subroutine
        1  is in sub2 subroutine

 me=          4  a(-)   10.0000000000000       12.0000000000000
  14.0000000000000       16.0000000000000       18.0000000000000
  20.0000000000000
        4  is in sub1 subroutine
        4     is in sub2 subroutine

 me=          2  a(-)   6.00000000000000       8.00000000000000
  10.0000000000000       12.0000000000000       14.0000000000000
  16.0000000000000
        2  is in sub1 subroutine
        2  is in sub2 subroutine

 me=          3  a(-)   8.00000000000000       10.0000000000000
  12.0000000000000       14.0000000000000       16.0000000000000
  18.0000000000000
        3  is in sub1 subroutine
        3  is in sub2 subroutine

 me=          5  a(-)   12.0000000000000       14.0000000000000
  16.0000000000000       18.0000000000000       20.0000000000000
  22.0000000000000
        5  is in sub1 subroutine
        5  is in sub2 subroutine
        0  b,h   6.00000000000000       8.00000000000000

 me=          0  a(-)   2.00000000000000       4.00000000000000
  6.00000000000000       8.00000000000000       10.0000000000000
  12.0000000000000
        0  is in sub1 subroutine
        0  is in sub2 subroutine
```

In Table 6.1, Preconditioned Conjugate Gradient (PCG) algorithm for solving system of symmetrical linear equations $[A]\vec{x} = \vec{b}$, with the preconditioned matrix [B] is summarized.

Table 6.1: Preconditioned Conjugate Gradient Algorithm For Solving $[A]\vec{x} = \vec{b}$

Step 1: Initialized $\vec{x_o} = \vec{0}$

Step 2: Residual vector $\vec{r_o} = \vec{b}$ (or $\vec{r_o} = \vec{b} - A\vec{x_0}$, for "any" initial guess $\vec{x_o}$)

Step 3: "Inexpensive" preconditioned $\vec{z_0} = [B]^{-1} \cdot \vec{r_0}$

Step 4: Search direction $\vec{d_0} = \vec{z_0}$

For i = 0, 1, 2, …, maxiter

Step 5: $\alpha_i = \dfrac{r_i^T z_i}{d_i^T \{A \cdot d_i\}}$

Step 6: $x_{i+1} = x_i + \alpha_i d_i$

Step 7: $r_{i+1} = r_i - \alpha_i [A d_i]$

Step 8: Convergence check: if $\|r_{i+1}\| < \|r_0\| \cdot \varepsilon$ ➔ stop

Step 9: $z_{i+1} = B^{-1} r_{i+1}$

Step 10: $\beta_i = \dfrac{r_{i+1}^T z_{i+1}}{r_i^T z_i}$

Step 11: $d_{i+1} = z_{i+1} + \beta_i d_i$

End for

6.15 Iterative Solution with Succesive Right-Hand-Sides

To keep the discussion more general, our objective here is to solve the system of "unsymmetrical", linear equations, which can be expressed in the matrix notations as:

$$[A]\,\vec{x} = \vec{b} \tag{6.381}$$

where [A] is a given N×N unsymmetrical matrix, and $\vec{b}$ is a given, single right-hand-side vector. System of "symmetrical" linear equations can be treated as a special case of "unsymmetrical" equations. However, more computational efficieny can be realized if specific algorithms (such as the Conjugate Gradient algotihms) which exploit the symmetrical property are used. Succesive right-hand-side vectors will be discussed near the end of this section.

Solving Eq.(6.381) for the unknown vector $\vec{x}$ is equivalent to either of the following optimization problem :

$$\text{Minimize} \quad \underset{x \in R^N}{\psi_1} \equiv \frac{1}{2} x^T A x - x^T b \tag{6.382}$$

or

$$\text{Minimize} \quad \underset{x \in R^N}{\psi_2} \equiv (A\,\vec{x} - \vec{b})^T * (A\,\vec{x} - \vec{b}) \tag{6.383}$$

Equation (6.382) is preferred for the case matrix [A] is "symmetrical", while Eq. (6.383) is suggested if matrix [A] is "unsymmetrical".

The minimization problem, described by Eq.(6.382) or Eq.(6.383), can be iteratively solved by the following step-by-step procedures as indicated in Table 6.9.

Table 6.9 Step-by-Step Iterative Optimization Procedures

Step 0 Initial guessed vector:

$$\vec{x} = \overrightarrow{x^{(0)}} \quad \text{; and set iterarion count } i = 0 \tag{6.390}$$

Step 1 Set i=i+1 at the current design point, find the "search direction" to travel, $s^{(i-1)}$?

Step 2 Find the step-size, α(how far should we travel along a given direction $s^{(i-1)}$)?

Step 3 Find the updated, improved design, $x^{(i)}$?

$$x^{(i)} = x^{(i-1)} + \alpha\, s^{(i-1)} \tag{6.391}$$

Step 4 Convergence test ??

Convergence is achieved if :

$$\left\| \nabla \psi_1^T \right\| \le \text{Tolerance} \tag{6.392}$$

or

$$\left\| \nabla \psi_2^T \right\| \le \text{Tolerance} \tag{6.393}$$

and / or

$$\left\| x^{(i)} - x^{(i-1)} \right\| \le \text{Tolerance} \tag{6.393}$$

If convergence is achieved (or, iteration # i=max. # of iterations allowed) then stop the process.

Else

Return to step1

End if

In Table 6.9 , the 2 most important steps are to find "the search direction " $s^{(i-1)}$ to travel from the current design point (see Step1), and to find "the step size" α (see Step 2).

[A] How to Find the "Step Size" , α , along a Given Direction $\vec{s}$??

Assuming the search direction $\overrightarrow{s^{(i)}}$ has already been found, then the new, improved design point can be computed as (see Step 3 of Table 6.9):

$$x^{(i+1)} = x^{(i)} + \alpha\, s^{(i)} \tag{6.395}$$

Thus, Eqs.(6.382 – 6.383) becomes minimization problems with only 1 variable (= α), as following:

$$\text{Min.} \quad \psi_1 = \frac{1}{2}\left(x^i + \alpha_1 s^i\right)^T A \left(x^i + \alpha_1 s^i\right) - \left(x^i + \alpha_1 s^i\right)^T b \tag{6.396}$$

and

$$\text{Min.} \quad \psi_2 = \left[A\left(x^i + \alpha_2 s^i\right) - b\right]^T * \left[A\left(x^i + \alpha_2 s^i\right) - b\right] \tag{6.397}$$

In order to minimize the function values ψ_1 (or ψ_2), one needs to satisfy the following requirements :

$$\frac{d\psi_1}{d\alpha_1} = 0 \tag{6.398}$$

and

$$\frac{d\psi_2}{d\alpha_2} = 0 \tag{6.399}$$

and

$$\alpha_2 = \frac{-\left(s^i\right)^T A^T \left(r^i\right)}{\left(s^i\right)^T A^T A\, s^i} \tag{6.412}$$

[B] How to Find the "Search Direction" , s^i ??

The initial direction , s^0 , is usually selected as the initial residual :

$$s^0 \equiv -r^0 = -(A\, x^0 - b) \tag{6.413}$$

The reason for the above selection of the initial search direction was because Eq.(6.413) represents the gradient $\nabla\psi_1(x^0)$, or "steepest discent direction" , of the objective ψ_1 ,defined in Eq.(6.382).

The step size is selected such that

$$\frac{d\psi_1(x^{i+1})}{d\alpha_1} = 0 = \frac{d\psi_1(x^{i+1} + \alpha_1 s^i)}{d\alpha_1} = \frac{d\psi_1}{dx} * \frac{dx}{d\alpha_1} \tag{6.414}$$

$$0 = \nabla\psi_1(x^{i+1}) * s^i \tag{6.415}$$

and

$$\frac{d\psi_2(x^{i+1})}{d\alpha_2} = 0 = \frac{d\psi_2(x^{i+1} + \alpha_2 s^i)}{d\alpha_2} = \frac{d\psi_2}{dx} * \left(\frac{dx}{d\alpha_2} = s^i\right) \tag{6.416}$$

Utilizing Eq.(6.409) , the above equation becomes :

$$\left(s^i\right)^T A^T * \left[A\left(x^i + \alpha_2 s^i\right) - b\right] = 0 \tag{6.417}$$

or

$$\left(s^i\right)^T A^T * \left[A\left(x^{i+1}\right) - b\right] = 0 \tag{6.418}$$

$$\left(s^i\right)^T A^T * \left[r^{i+1}\right] = 0 \tag{6.419}$$

Since Eq.(6.419) represents the "scalar quantity" , hence it can also be presented in its "transposed" form, as following :

$$\left[r^{i+1}\right]^T * A\, s^i = 0 \tag{6.420}$$

Comparing Eq.(6.420) with Eq.(6.416) , one concludes :

$$\nabla\psi_2 \equiv \frac{\partial\psi_2}{\partial x} \equiv \left[r^{i+1}\right]^T * A \tag{6.421}$$

Thus, Eq.(6.420) can also be presented as :

$$\nabla\psi_2(x^{i+1}) * s^i = 0 \tag{6.422}$$

One can built a set of "A_conjugate" vectors $(= s^0, s^1, \cdots, s^i, s^{i+1})$ by applying the

Gram-Schmidth procedures to the (new) residual vector :

$$\hat{r}^{\,i+1} = A\, x^{i+1} - b = 0 \tag{6.423}$$

for obtaining the search direction :

$$\hat{s}^{\,i+1} = \hat{r}^{\,i+1} + \sum_{k=0}^{i} \beta_k\, \hat{s}^{\,k} \tag{6.424}$$

where :

$$\beta_k = \frac{-\hat{r}^{\,i+1} A\, \hat{s}^{\,k}}{\left(\hat{s}^{\,k}\right)^T A\, \hat{s}^{\,k}} \tag{6.425}$$

and the following property of "A_conjugate" vectors will be satisfied :

$$\left(\hat{s}^{\,i+1}\right)^T A\, \hat{s}^{\,k} = 0 \quad ; k = 0, 1, 2, \cdots, i \tag{6.426}$$

Utilizing Eq.(6.461), Polak-Rebiere Algorithm for β_i in Eq.(6.454) becomes :

$$\beta_i = \frac{\left(\hat{r}^{\,i+1}\right)^T * (\hat{r}^{\,i+1})}{\left(\hat{r}^{\,i}\right)^T * (\hat{r}^{\,i})} = \text{Fletcher} - \text{Reeves Algorithm} \tag{6.462}$$

#[4] If [A] is an "unsymmetrical" matrix, then Reference [6.23] suggests to use " $[A^T A]$_conjugate " vectors as :

$$\hat{s}^{i+1} = \hat{r}^{i+1} + \sum_{k=0}^{i} \beta_k \hat{s}^k \tag{6.463}$$

where:

$$\beta_k = \frac{\left(-A\hat{r}^{i+1}\right)^T \left(A\hat{s}^k\right)}{\left(\hat{s}^k\right)^T A^T A \left(\hat{s}^k\right)} \tag{6.464}$$

with the following "Conjugate_like" property:

$$\left(A\hat{s}^{i+1}\right)^T \left(A\hat{s}^k\right) = 0 \tag{6.465}$$

[C] Based upon the discussions in previous sections, the Generalized Conjugate Residual (GCR) algorithms can be described in the following step-by-step procedures, as indicated in Table 6.10.

Table 6.10 GCR Step-by-step Algorithms

Step 1	Choose an initial guess for the solution , x^0	
	Compute $r^0 = b - A x^0$	(see Eq.6.406)
Step 2	Start optimization iteration , for j=1,2,...	
	Choose an initial search direstion , $\hat{s}^j$	
	where $\hat{s}^j = r^{j-1}$	(see Eq.6.413)
	Compute $\hat{v}^j = A\hat{s}^j$; (portion of Eq.6.412)	(6.493)
Step 3	Generate a set of conjugate vectors $\hat{s}^j$, The Gram-Schmidth Process	
	for $k = 1, 2, \cdots j-1$	
	$\beta = \left(\hat{v}^j\right)^T \left(v^k\right)$; see Eq.(6.464)	(6.494)
	$\hat{v}^j = \hat{v}^j - \beta v^k$	(6.495)
	$\hat{s}^j = \hat{s}^j - \beta s^k$; see Eq.(6.463)	(6.496)
	End for	

Step 4 Normalize the vectors

$$\gamma = \left(\hat{v}^{\,j}\right)^T \left(\hat{v}^{\,j}\right) \quad \text{; complete the denominator of Eq.(6.412)} \tag{6.497}$$

$$v^{\,j} = \frac{\hat{v}^{\,j}}{\gamma} \quad \text{; "almost" completing Eq.(6.412)} \tag{6.498}$$

$$s^{\,j} = \frac{\hat{s}^{\,j}}{\gamma} \tag{6.499}$$

Step 5 Compute the step size

$$\alpha = \left(r^{j-1}\right)^T \left(v^{j}\right) \quad \text{; completely done with Eq.(6.437)} \tag{6.500}$$

Step 6 Compute the updated solution

$$x^{j} = x^{j-1} + \alpha\, s^{j} \qquad \text{(see Eq.6.395)}$$

Step 7 Update the residual vector

$$r^{j} = r^{j-1} - \alpha\, v^{j} \tag{6.501}$$

Step 8 Convergence check ??

$$\text{If} \quad \left[\frac{\|r^{j}\|}{\|b\|} \right] \le \varepsilon_{Tol} \quad ; \quad \text{Then} \tag{6.502}$$

Stop

Else

$j = j + 1$

Go To Step 2

[D] How to Efficiently Handle Succesive Right-Hand-Side Vectors ??

Assuming we have to solve for the following problems :

$$[A]\vec{x}_i = \vec{b_i} \tag{6.511}$$

In Eq.(6.511), the right-hand-side (RHS) vectors are NOT all available at the same time, but that $\vec{b_i}$ depens on $\vec{x}_{i-1}$. There are 2 objectives in this section, which will be discussed in subsequent paragraphs :

#(1) Assuming the 1st solution $\vec{x}_1$, which corresponds to the 1st RHS vector $\vec{b_1}$ has already been found in "n_1" iterations. Here, we would like to utilize the first "n_1" generated (and orthogonal) vectors $s_{i=1}^{j=1,2,\cdots,n_1}$ to find "better initial guess" for the 2nd RHS solution vector $\vec{x}_{i=2}$. The new, improved algorithms will select the initial solution $\vec{x}_{i=2}^{0}$, to minimize the errors defined by ψ_1 (see Eq.6.382), or ψ_2 (see Eq.6.383) in the vector space spanned by the already existing (and expanding) "n_1" conjugate vectors. In other words, one has :

$$\vec{x}_{i=2}^{0} \equiv [s_1^1, s_1^2, \cdots, s_1^{n_1}]_{n\times n_1} * \{P\}_{n_1\times 1} = [S_1] * \{P\} \tag{6.512}$$

Eq.(6.512) express that the initial guess vector $\vec{x}_2^0$ is a linear combinations of columns $s_1^1, s_1^2, \cdots, s_1^{n_1}$.

By minimizing (with respect to $\{P\}$) ψ_2 (defined in Eq.6.383), one obtains :

$$\nabla_p(\psi_2) \equiv \frac{\partial \psi_2}{\partial \vec{P}} = \vec{0} \tag{6.513}$$

which will lead to :

$$\{P\}_{n_1\times 1} = \frac{[s_1^T]_{n_1\times 1} * [A^T]_{n\times n} * \{\vec{b}_2\}_{n\times 1}}{[s_1^T][A^T][A][s_1]} \tag{6.514}$$

For the "symmetrical" matrix case, one gets :

$$\nabla_p(\psi_1) \equiv \frac{\partial \psi_1}{\partial \vec{P}} = \vec{0} \tag{6.515}$$

which will lead to :

$$\{P\} = \frac{[s_1^T] * \{\vec{b}_2\}}{[s_1^T][A][s_1]} \tag{6.516}$$

(1.4) The step-by-step algorithms to generate "good" initial guess $\overrightarrow{x_2^0}$ for the 2nd RHS vector can be given as shown in Table 6.11.

Table 6.11 Step-by-step Algorithms To Generate "Good" Initial Guess for RHS Vectors

$$\overrightarrow{x_2^0} = \vec{0} \tag{6.522}$$

$$\overrightarrow{r_2^0} = \overrightarrow{b_2} \tag{6.523}$$

for $k = 1, 2, \cdots, n_1$ (6.524)

$$P = \left(r_2^0\right)^T \left(v_1^k\right) \tag{6.525}$$

$$x_2^0 = x_2^0 + P s_1^k \qquad \text{(6.526 also see Eq.6.512)}$$

c Recalled : $v_1^k = A s_1^k$ (see Eq.6.493 , where s_1^k has already been normalized

c according to Eq.6.499)

c Furthermore, kept in mind that v_1^k and s_1^k vectors had already been generated

c and stored when the 1st-RHS had been processed.

c Thus, Eq.(6.525) represents the implementation of Eq.(6.514), but corresponding

c to "ONLY 1 column" of matrix [S_1]. That's why we used to have a "do loop

c index k" (see Eq.6.524) to completely execute Eq.(6.514) & Eq.(6.512).

$$r_2^0 = r_2^0 - P v_1^k \tag{6.527}$$

c Eq.(6.527) can be derived as following :

c First, pre-multiplying both sides of Eq.(6.526) by (-A), one obtains

c (note :P=scalar, in Eq.6.525) :

c $(-A) x_2^0 = (-A) x_2^0 + P(-A) s_1^k$

c Then, adding (b_2) to both sides of the above equation, one gets :

c $-A x_2^0 + b_2 = -A x_2^0 + b_2 - P\left(A s_1^k\right)$

c or

c $r_2^0 = r_2^0 - P\left(A s_1^k\right)$

c or, referring to Eq.(6.493), then the above Eq. becomes :

c $r_2^0 = r_2^0 - P\left(v_1^k\right)$, which is the same as indicated in Eq.(6.527).

End for

#(2) For successive RHS vectors, the "search vectors" s^j (see Eq.6.496 , in Table6.10) need to be modified, so that these vectors s^j will not only orthogonal amongst themselves (corresponding to the current 2nd RHS vector) , but they also will orthogonal with the axisting n_1 vectors [corresponding to ALL "previous" RHS vector(s)]. Thus, the total number of (cumulative) conjugate vectors will be increased, and hence "faster convergence" in the GCR algorithm can be expected.

Obviously, there will be a "trade-off" between "faster convergence rate" versus the "undesired" increase in computer memory requirement. In practical computer implementation, the user will specify "how many "cumulative" vectors s^j and v^j that can be stored in RAM. Then, if these vectors s^j and v^j have already filled up the user's specified incore memory available and convergence is still NOT YET achieved, one may have to "restart" the process (by starting with the new initial guess etc ...).

The amount of work and the amount of memory that is required are proportional to the total number of search vectors that have been generated. To make the extended GCR algorithm competitive with direct solution algorithms, it is essential to keep the total number of generated search vectors as small as possible. This can be achieved by preconditioning the system of equations (6.511). Another way to reduce the amount of work and the memory is to use the extended GCR algorithm in combination with the Domain Decomposition (DD) approach.

6.16 Summary

In this chapter, various domain decomposition (DD) formulations have been presented. Sparse and dense matrix technologies, mixed direct-iterative equation solvesrs, preconditioned algorithms, indefinite sparse solver,generalized sparse inverse (or factorization) , parallel computation etc ... have all been integrated into a DD formulation. Several numerical examples with explicit,detailed calculation have been used to verify the step-by-step algorithms.

```
c================================================
c......Purposes: Reviewing some basic things about
c                FORTRAN90
c......Author(s):  D.T. Nguyen
c......Latest Date:  April 04, 2005
c---------------------------------------------------------------
c      generating "real" random numbers
c                (uniform distribution) between
c                [0.00, 1.00]
c---------------------------------------------------------------
      implicit real*8 (a-h,o-z)
c     include 'mpif.h'
      character*80 title
      parameter(num=10)
      dimension a(num)
       allocatable:: ia(:), a11(:,:), a22(:,:)
c---------------------------------------------------------------
c      call MPI_INIT(ierr)
c      call MPI_COMM_RANK(MPI_COMM_WORLD, me, ierr)
c      call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)
c....... call MPI_BARRIER(MPI_COMM_WORLD, ierr)
c....... call MPI_SEND(num,1,MPI_INTEGER,i_destination,1,
c.......               MPI_COMM_WORLD,ierr)
c....... call MPI_RECV(num,1,MPI_INTEGER,master,mtype,
c.......               MPI_COMM_WORLD,status,ierr)
c---------------------------------------------------------------
      idum=0           ! ******************
      sum=0.
c     master processor me = 0
       me=0
c
       if (me .eq. 0) then
      write(6,*) 'i, a(i) = random real numbers'
       endif
c
      do 1 i=1,num
c**********************
       a(i)=drand(idum)
c**********************
       sum=sum+a(i)
       if (i .le. 10) then
       write(6,*) i,a(i)
       elseif (i .ge. 100) then
       write(6,*) 'skip printing too many random # '
       endif
 1    continue
c---------------------------------------------------------------
       open(unit=7,file='K.INFO',status='old',form='formatted')
c      open(unit=6,file='out1',status='old',form='formatted')
       read(7,115) title
 115   format(a60)
       write(6,115) title
c---------------------------------------------------------------
       memory_need=2*num
       allocate ( ia(memory_need), a11(memory_need,memory_need),
     $            a22(num,num) )
       do 2 i=1,memory_need
```

```
       ia(i)=i
       write(6,*) 'ia(memory_need) = ', ia(i)
 2     continue
        deallocate (a11,a22)
c-------------------------------------------------------------
       call dummy1(num,memory_need,ia,a,sum_int,sum_real)
       write(6,*) 'sum_int, sum_real = ',sum_int, sum_real
c-------------------------------------------------------------
c      call MPI_FINALIZE(ierr)
c
       stop
       end
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
       subroutine dummy1(num,memory_need,ia,a,sum_int,sum_real)
       implicit real*8(a-h,o-z)
       dimension ia(*), a(*)
c
       sum_int=0
       sum_real=0.0
        iloop=num
        if (memory_need .le. num) iloop=memory_need
       do 1 i=1,iloop
       sum_int=sum_int+ia(i)
       sum_real=sum_real+a(i)
 1     continue
c
       return
       end
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 i, a(i) = random real numbers
 1 0.9680711915567849
 2 0.06673061943926412
 3 0.4782810935183806
 4 0.9095344240355931
 5 0.3516923782190738
 6 0.932533659940834
 7 0.6544358239762652
 8 0.021070247060186345
 9 0.512204862903899
 10 0.20201893067081408
  this is just a test for reading NASA K* data files
 ia(memory_need) =  1
 ia(memory_need) =  2
 ia(memory_need) =  3
 ia(memory_need) =  4
 ia(memory_need) =  5
 ia(memory_need) =  6
 ia(memory_need) =  7
 ia(memory_need) =  8
 ia(memory_need) =  9
 ia(memory_need) =  10
 ia(memory_need) =  11
 ia(memory_need) =  12
 ia(memory_need) =  13
 ia(memory_need) =  14
 ia(memory_need) =  15
 ia(memory_need) =  16
```

```
 ia(memory_need) =  17
 ia(memory_need) =  18
 ia(memory_need) =  19
 ia(memory_need) =  20
 sum_int, sum_real =  55.0 5.096573231321095
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Appendix A: Singular Value Decomposition (SVD)

Singular Value Decomposition (SVD) has many important, practical applications: search engines[2], computational information retrieval[3], least square problems[1], image compressing[4], etc...

A general (square, or rectangular) matrix A can be decomposed as:

$$A = U\Sigma V^H \tag{A.1}$$

where:

$[\Sigma]$ = diagonal matrix (does not have to be a square matrix)

$$= \begin{cases} \Sigma_{ij} = 0, & \text{for } i \neq j \\ \Sigma_{ij} \geq 0, & \text{for } i = j \end{cases}$$

$[U]$ and $[V]$ = unitary matrices $\begin{cases} U^H = U^T \text{ (for real matrix)} \\ = U^{-1} \end{cases}$

The SVD procedures to obtain matrices $[U],[\Sigma]$ and $[V]$ from a given matrix $[A]$ can be illustrated by the following examples:

Example 1: Given $[A] = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ (A.2)

Step 1: Compute $A A^H = A A^T = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} = \begin{bmatrix} 5 & 11 \\ 11 & 25 \end{bmatrix}$ (A.3)

Also: $A A^H = (U\Sigma \underbrace{V^H)(V}_{\text{Identity Matrix}}\Sigma^H U^H) = U\Sigma^2 U^H$ (A.4)

$$A^H A = V\Sigma^2 V^H \tag{A.5}$$

---

[1] B. Noble et.al (2nd Edition), Applied Linear Algebra, Prentice Hall

[2] M.W. Berry, and M. Bworne, "Understanding Search Engines", SIAM, ISBN# 0-89871-437-0

[3] M.W. Berry, "Computational Information Retrieval", SIAM, ISBN# 0-89871-500-8

[4] J.W. Demmel, "Applied Numerical Linear Algebra", SIAM, ISBN# 0-89871-389-7

Step 2: Compute the standard eigen-solution of $A A^T$

$$\left[A A^T - \lambda I\right]\vec{u} = \vec{0} \tag{A.6}$$

Hence: $$\det\begin{bmatrix} 5-\lambda & 11 \\ 11 & 25-\lambda \end{bmatrix} = \vec{0} \tag{A.7}$$

or: $$\lambda^2 - 30\lambda + 4 = 0 \tag{A.8}$$

Thus: $$\lambda = 15 \pm \sqrt{221} = \begin{pmatrix} 0.1339 & 29.87 \end{pmatrix} \tag{A.9}$$

Now: $$\sigma = \sqrt{\lambda} = \begin{pmatrix} 0.3660 & 5.465 \end{pmatrix} \tag{A.10}$$

Hence: $$\left[\Sigma\right] = \begin{bmatrix} 0.3660 & 0 \\ 0 & 5.465 \end{bmatrix} \tag{A.11}$$

- For $\lambda = 15 - \sqrt{221}$, then Eq.(A.6) becomes :

$$11\, u_1^{(1)} + (25-\lambda)\, u_2^{(1)} = 0 \tag{A.12}$$

or $$u_1^{(1)} = \frac{(\lambda - 25)\, u_2^{(1)}}{11} \tag{A.13}$$

Let $u_2^{(1)} = 1$, then : (A.14)

$$u_1^{(1)} = \frac{(15 - \sqrt{221} - 25) * 1}{11} = -2.261 \tag{A.15}$$

So: $$u_1^{(1)} = \begin{Bmatrix} -2.261 \\ 1 \end{Bmatrix} \tag{A.16}$$

Eq.(A.16) can be normalized, so that $\left\|u^{(1)}\right\| = 1$, to obtain

$$u_{\text{Normalized}}^{(1)} = \frac{1}{\sqrt{(-2.261)^2 + (1)^2}} \begin{Bmatrix} -2.261 \\ 1 \end{Bmatrix} = \begin{Bmatrix} -0.9145 \\ 0.4046 \end{Bmatrix} \tag{A.17}$$

- Similarly, for $\lambda = 15 + \sqrt{221}$, and let $u_2^{(2)} = 1,$ then:

$$u^{(2)} = \begin{Bmatrix} 0.4424 \\ 1 \end{Bmatrix} \tag{A.18}$$

Hence:

$$u_{\text{Normalized}}^{(2)} = \frac{1}{\sqrt{(0.4424)^2 + (1)^2}} \begin{Bmatrix} 0.4424 \\ 1 \end{Bmatrix} = \begin{Bmatrix} 0.4046 \\ 0.9145 \end{Bmatrix} \tag{A.19}$$

Thus:

$$\left[U\right] = \begin{bmatrix} u_N^{(1)} & u_N^{(2)} \end{bmatrix} = \begin{bmatrix} -0.9145 & 0.4046 \\ 0.4046 & 0.9145 \end{bmatrix} \tag{A.20}$$

Step 3: Compute $A^H A = A^T A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 10 & 14 \\ 14 & 20 \end{bmatrix}$ (A.21)

The 2 eigen-values associated with ($A^T A$) can be computed as :

$$\lambda = 15 \pm \sqrt{221} = \begin{pmatrix} 0.1339 & 29.87 \end{pmatrix} \quad \text{(A.22)}$$

Hence: $$\sigma = \sqrt{\lambda} = \sqrt{\begin{pmatrix} 0.1339 & 29.87 \end{pmatrix}} = \begin{pmatrix} 0.3660 & 5.465 \end{pmatrix} \quad \text{(A.23)}$$

- For $\lambda = 15 - \sqrt{221}$, and let $v_2^{(1)} = 1$, then :

$$v^{(1)} = \begin{Bmatrix} -1.419 \\ 1 \end{Bmatrix} \text{ ; hence } v_N^{(1)} = \begin{Bmatrix} -0.8174 \\ 0.5760 \end{Bmatrix} \quad \text{(A.24)}$$

- For $\lambda = 15 + \sqrt{221}$, and let $v_2^{(2)} = 1$, then :

$$v^{(2)} = \begin{Bmatrix} 0.7047 \\ 1 \end{Bmatrix} \text{ ; hence } v_N^{(2)} = \begin{Bmatrix} 0.5760 \\ 0.8174 \end{Bmatrix} \quad \text{(A.25)}$$

Thus:

$$[V] = \begin{bmatrix} v_N^{(1)} & v_N^{(2)} \end{bmatrix} = \begin{bmatrix} -0.8174 & 0.5760 \\ 0.5760 & 0.8174 \end{bmatrix} \quad \text{(A.26)}$$

Therefore, the SVD of [A] can be obtained from Eq.(A.1) as:

$$A = U\Sigma V = \begin{bmatrix} -0.9145 & 0.4046 \\ 0.4046 & 0.9145 \end{bmatrix}\begin{bmatrix} 0.3660 & 0 \\ 0 & 0.3660 \end{bmatrix}\begin{bmatrix} -0.8174 & 0.5760 \\ 0.5760 & 0.8174 \end{bmatrix}$$

Example 2: Given $[A] = \begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 2 & 2 \end{bmatrix}$ (A.27)

- Compute $\underset{3\times2}{A} \underset{2\times3}{A^H} = \begin{bmatrix} 2 & 4 & 4 \\ 4 & 8 & 8 \\ 4 & 8 & 8 \end{bmatrix}$ (A.28)

The corresponding eigen-values, and eigen-vectors of Eq.(A.28) can be given as:

$$(\lambda_1 \ \lambda_2 \ \lambda_3) = (18 \ 0 \ 0) \ \Rightarrow \ \sigma = \sqrt{(\lambda_1 \ \lambda_2 \ \lambda_3)} = \left(3\sqrt{2} \ 0 \ 0\right) \quad \text{(A.29)}$$

$$\left(u^{(1)} \ u^{(2)} \ u^{(3)}\right) = \begin{bmatrix} 1/3 & -2/\sqrt{5} & 2\sqrt{5}/15 \\ 2/3 & 1/\sqrt{5} & 4\sqrt{5}/15 \\ 2/3 & 0 & -5\sqrt{5}/15 \end{bmatrix} \quad \text{(A.30)}$$

- Compute $\underset{2\times3}{A^H} \underset{3\times2}{A} = \begin{bmatrix} 9 & 9 \\ 9 & 9 \end{bmatrix}$ (A.31)

The corresponding eigen-values, and eigen-vectors of Eq.(A.31) can be given as:

$$(\lambda_1 \ \lambda_2) = (18 \ 0) \quad \text{(A.32)}$$

$$\sigma = \sigma_1 = \sqrt{\lambda} = \sqrt{18} = 3\sqrt{2} \quad \text{(A.33)}$$

$$\left(v^{(1)}\ \ v^{(2)}\right)=\begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix} \tag{A.34}$$

Hence $$\underset{3\times2}{A}=\underset{3\times3}{U}\ \underset{3\times2}{\Sigma}\ \underset{2\times2}{V} \tag{A.35}$$

$$A=\begin{bmatrix} 1/3 & -2/\sqrt{5} & 2\sqrt{5}/15 \\ 2/3 & 1/\sqrt{5} & 4\sqrt{5}/15 \\ 2/3 & 0 & -5\sqrt{5}/15 \end{bmatrix}\begin{bmatrix} 3\sqrt{2} & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix} \tag{A.36}$$

Example 3: Given $$[A]=\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} \tag{A.37}$$

The SVD of A is:

$$A=U\Sigma V^{H}=U\Sigma V^{T}=\begin{bmatrix} 0.141 & 0.825 & -0.420 & -0.351 \\ 0.344 & 0.426 & 0.298 & 0.782 \\ 0.547 & 0.028 & 0.664 & -0.509 \\ 0.750 & -0.371 & -0.542 & 0.079 \end{bmatrix}*\begin{bmatrix} 25.5 & 0 & 0 \\ 0 & 1.29 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}*\begin{bmatrix} 0.504 & 0.574 & 0.644 \\ -0.761 & -0.057 & 0.646 \\ 0.408 & -0.816 & 0.408 \end{bmatrix} \tag{A.38}$$

Example 4: (Relationship between SVD and generalized inverse)

"Let the $m\times n$ matrix A of rank k have the SVD $A=U\Sigma V^{H}$; with $\sigma_1\geq\sigma_2\geq\cdots\geq\sigma_k\rangle 0$. Then the generalized inverse A^{+} of A is the n×m matrix.

$A^{+}=V\Sigma^{+}U^{H}$; where $\Sigma^{+}=\begin{bmatrix}[E] & [0] \\ [0] & [0]\end{bmatrix}$ and E is the k×k diagonal matrix, with $E_{ii}=\sigma_i^{-1}$ for $1\leq i\leq k$"

$$\text{Given SVD of } A=\begin{bmatrix} 1/3 & -2\sqrt{5}/5 & 2\sqrt{5}/15 \\ 2/3 & \sqrt{5}/5 & 4\sqrt{5}/15 \\ 2/3 & 0 & -\sqrt{5}/15 \end{bmatrix}\begin{bmatrix} 3\sqrt{2} & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} \sqrt{2}/2 & \sqrt{2}/2 \\ \sqrt{2}/2 & -\sqrt{2}/2 \end{bmatrix}$$

$$\text{Hence : } A^{+}=\begin{bmatrix} \sqrt{2}/2 & \sqrt{2}/2 \\ \sqrt{2}/2 & -\sqrt{2}/2 \end{bmatrix}\begin{bmatrix} 1/3\sqrt{2} & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} 1/3 & 2/3 & 2/3 \\ -2\sqrt{5}/5 & \sqrt{5}/5 & 0 \\ 2\sqrt{5}/15 & 4\sqrt{5}/15 & -\sqrt{5}/15 \end{bmatrix}=\begin{bmatrix} 1/18 & 1/9 & 1/9 \\ 1/18 & 1/9 & 1/9 \end{bmatrix}$$