Scientific Linraries

ACF Spring HPC Training Workshop
Match 15-16, 2016
Kwai Wong

Joint Institute for Computational Sciences

THE UNIVERSITY OF TENNESSEE KNOXVILLE

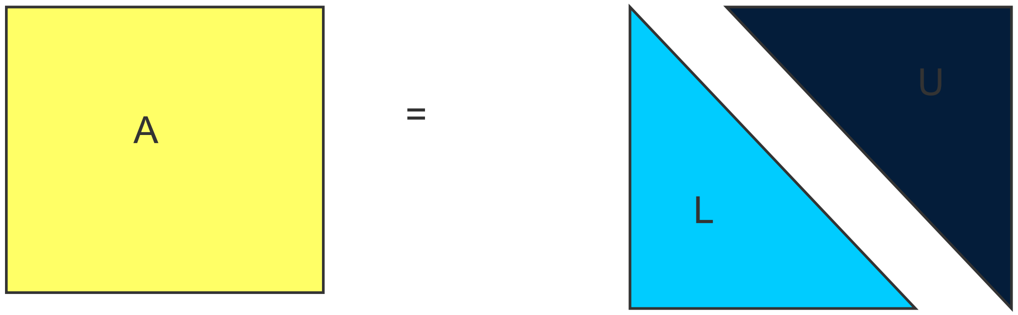
THE UNIVERSITY OF TENNESSEE HEALTH SCIENCE CENTER

Basic Computing Kennel (serial)
Basic Linear Algebra Subprograms(BLAS)

HPL (scalLAPACK) - Parallel Gausssian Elimination

$$Ax = b$$

change A into $A = L\,U$ in parallel



so $LUx = b$

first solve $Ly = b$ by direct downward solve

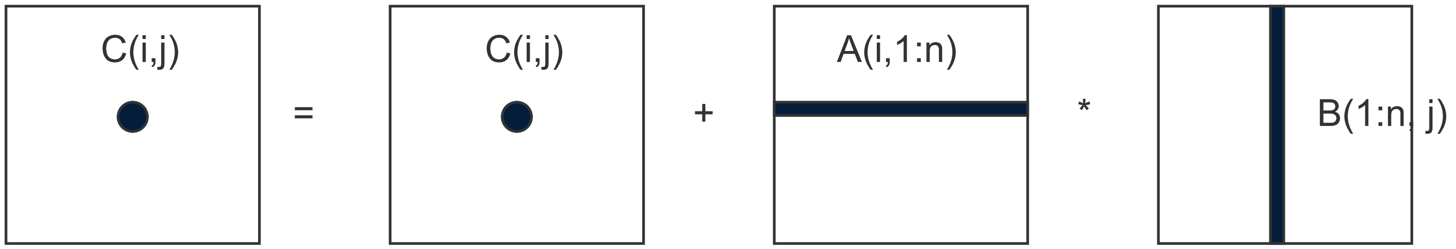
then solve $Ux = y$ by direct upward solve

Basic Linear Algebra Subprograms (BLAS)

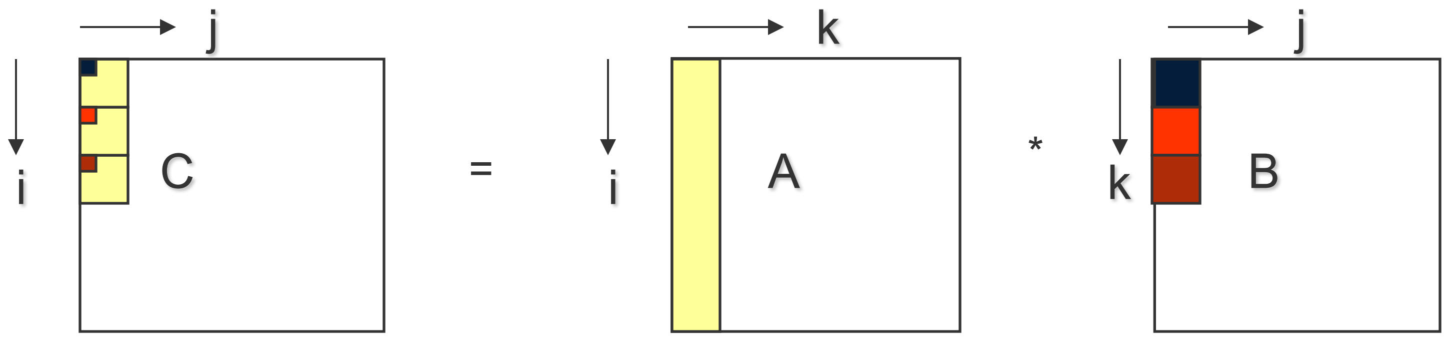
- BLAS is a library of standardized basic linear algebra computational kernels created to perform efficiently on serial computers taking into account the memory hierarchy of modern processors.
- BLAS1 does vectors-vectors operations.
    - Saxpy = y(i) = a* x(i) + y(i), ddot= 🏟 x(i) *y(i)
- BLAS2 does matrices - vectors operations.
    - MV : y = A x + b
- BLAS3 operates on pairs or triples of matrices.
  – MM : $C = \alpha AB + \beta C$, Triangular Solve : $X = \alpha T^{-1}X$
- Level3 BLAS is created to take full advantage of the fast cache memory. Matrix computations are arranged to operate in block fashion. Data residing in cache are reused by small blocks of matrices.
- Atlas, openBLAS, MKL, ESSL, libsci

MM Multiplication

- Simple MM - q = average number of flops per memory reference ~ 2



- Performance of MM can be improved by rearranging the order of multiplication indices in column fashion in Fortran or in row fashion in C.



k - j - i ordering for FORTRAN

Analysis on MM
(www.cs.berkeley.edu/ /~demmel/cs267_Spr99/)

- To quantify the analysis, a simple model of two levels of memory hierarchy, fast and slow, are used. All data initially resides in slow memory. Define

  m = number of memory references to slow memory needed just to read the input data from slow memory, and write the output data back

  f = number of floating point operations

  q = f/m = average number of flops per slow memory reference

- Hence, the higher is the value of q , the more efficient the algorithm

  m = n^3 ---> read each column of B n times +

  n^2 ---> read each row of A for each I +

  2*n^2 ---> read/write each entry of C once --------> n^3 + 3* n^2

  f = 2* n^3

  q = f/m = (2* n^3) / (n^3 + 3* n^2) ~ 2

- Ideal value of q = n/2

  ideal value of m = 4* n^2 ----> read each A(I,j), B(I,j), C(I,j) once,

  write each C(I,j) once

  hence, ideal value of q = f/m = n/2

Square blocked MM

- Consider C to be an n-by-n matrix of n/N-by-n/N subblocks Cij, with A and B similarly partitioned.

```
for j = 1 : N
    for j = 1 : N
        Read Cij into fast memory
        for k = 1 : N
            Read Aik into fast memory
            Read Bkj into fast memory
            Cij = Cij + Aik *  Bkj
        end for
        Write Cij back to slow memory
    end for
end for
```

- The inner loop is an n/N-by-n/N matrix multiply. The fast memory is large enough to hold the 3 subblocks Cij, Aik, and Bkj.

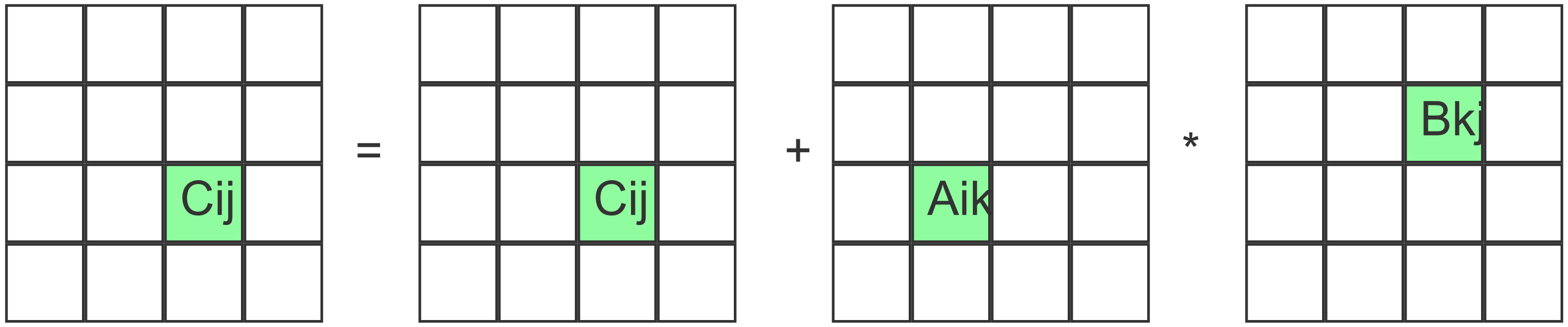
  m = # memory refs = N * n^2 -----> read each Bkj N^3 times

  + N * n^2 ------> read each Aik N^3 times

  + 2 * n^2 --------> read/write each Cij once =   (2 * N + 2) * n^2

Block MM

- q = f/m = (2*n^3) / ((2*N + 2) * n^2) ~ n / N
- If N is equal to 1, the algorithm is ideal. However, N is bounded by the amount of fast cache memory. However, N can be taken independently to the size of matrix, n.
- The optimal value of N = sqrt (size of fast memory / 3)



$$\boxed{Cij} = \boxed{Cij} + \sum_{k=1}^{n} \boxed{Aik} * \boxed{Bkj}$$

ATLAS

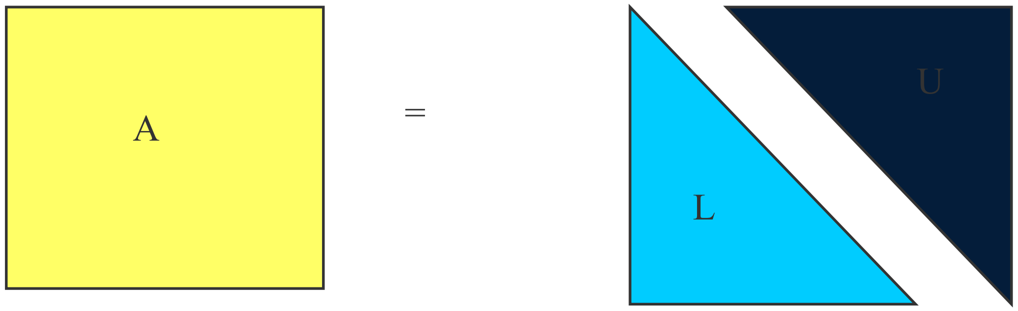
- Automatically Tuned Linear Algebra Software
- It generates a set of optimized linear algebra routines on different computer architectures taking the advantages of their specific memory hierarchies and pipelined functional units.
- In version 3.0, it supports all level of BLAS kernels as well as some LAPACK routines.
- It also provides interfaces to standard C (need cblas.h) and fortran 77.
- Prebuilt ATLAS for various computer architectures are readily available on the web.
- Good for Linux Platform
- www.netlib.org/atlas

Linear Algebra Package (LAPACK)

Gausssian Elimination

Ax = b

change A into A = L U



so LUx = b

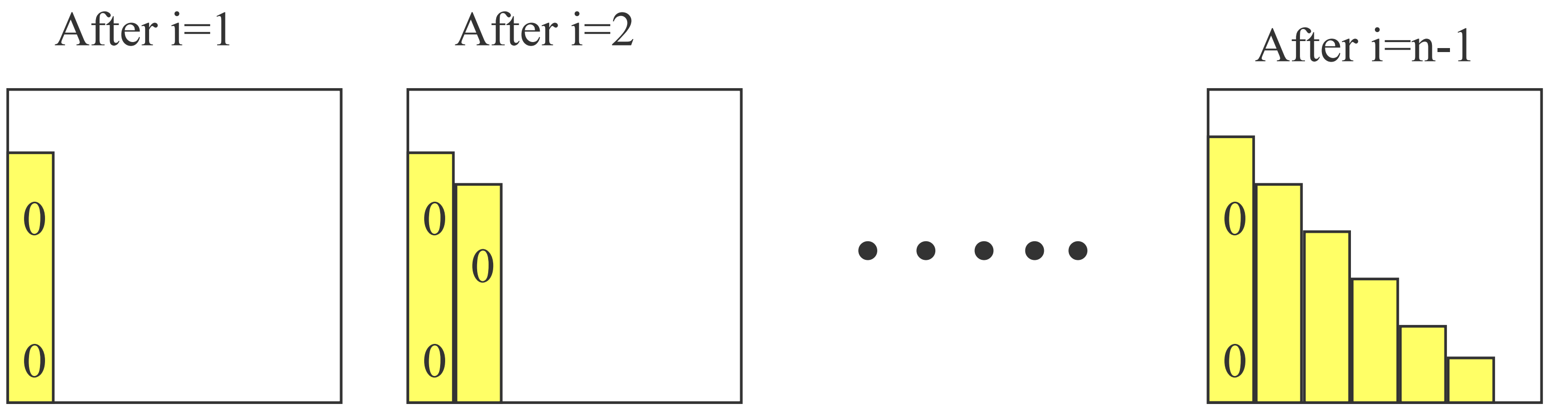
first solve Ly = b by direct downward solve

then solve Ux = y by direct upward solve

Gaussian Elimination

- For each column i, zero out the element below the diagonal by adding multiples of row i to later rows

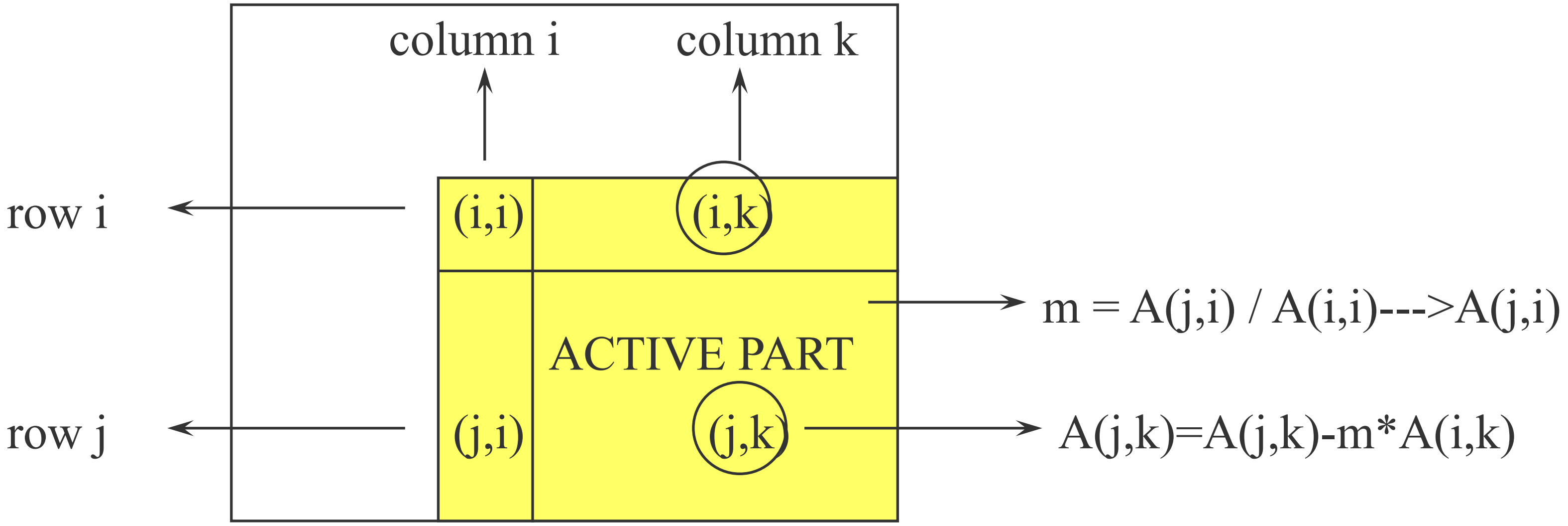
```
for i= 1 to n-1
    for j = i+1 to n
        for k = i to n
        $A(j,k) = A(j,k) - (A(j,i) / A(i,i)) * A(i,k)$
```



Gaussian Elimination (2)

- To improve the implementation, the constant A(j,i) / A(i,i) is removed from the innermost loop. Zeros below the diagonal is ignored

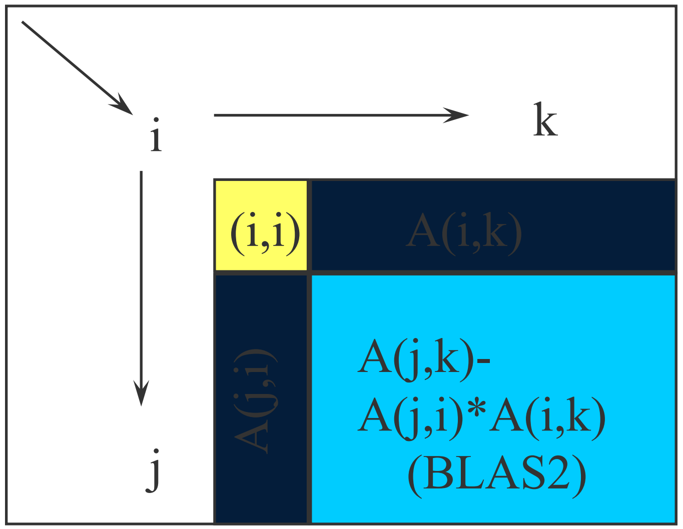
```
for i = 1 to n-1
    for j = i+1 to n
            m = A(j,i) / A(i,i) ----> m = A(j,i)
            for k = i to n
            A(j,k) = A(j,k) - m * A(i,k)
```

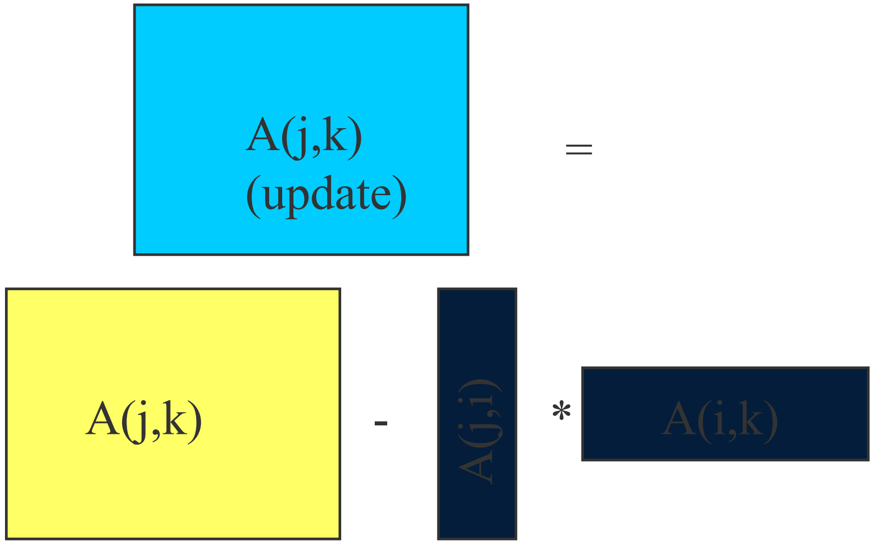


LAPACK (LU)

- The inner loop consists of BLAS1 and one BLAS 2 operations.

```
for i = 1 to n-1
    for j = i+1 to n
        A(j,I) = A(j,i) / A(i,i)   <------- BLAS1 ( to BLAS2)
        for k = i+1 to n
        A(j,k) = A(j,k) - A(j,i)* A(i,k)   <----BLAS2 ( to BLAS3)
```





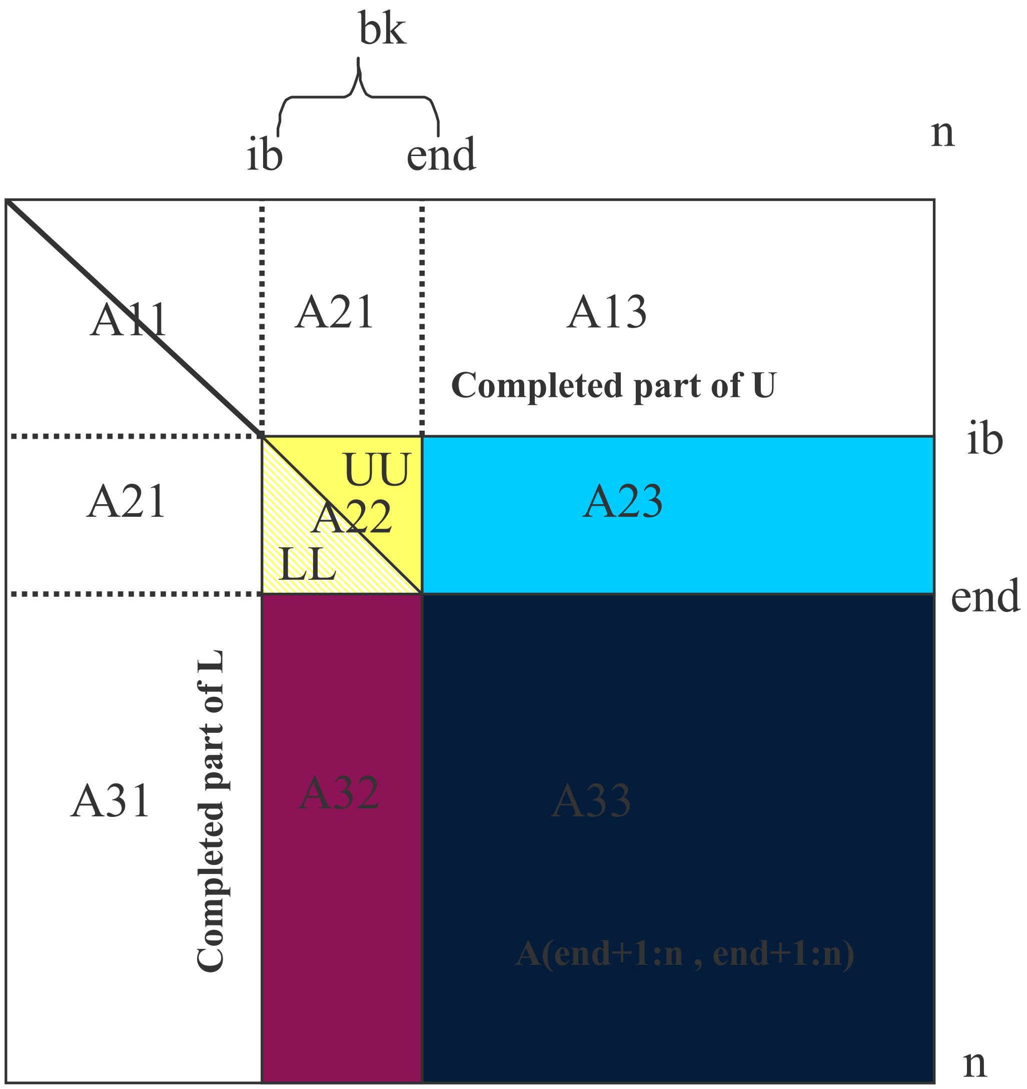
LAPACK GE Block Algorithm

- The block size of bk columns will depend on the machine architectures. It is generally small enough so that bk columns currently used for factorization fit in the fast memory of the machine, and bk is also large enough to make matrix matrix multiplication perform effectively.
- The principle is the same as in the ordinary GE algorithm above. Instead of working with one column, A(j,i) or one pivot entry, A(l,l), a block of columns and a square block of matrix are used. Hence, BLAS1 operations will become BLAS2 operations, and BLAS2 operations will become BLAS3 operation

```
choose a block size bk
for ib = 1 , n , bk
    1) L U factorize the column block of bk
    2) compute the pivoting block of rows
    3) update the remaining block of the square matrix
```

LAPACK GE Block Algorithm

```
I) Choose bk
II) for ib = 1 to n-1 step bk
      Work the colored portion of A
      1) LU factorize A22+A32
         A32 <--( UU, LL, A32)
      2) Update A23 : triangular solve
            (A23) <-- LL \ A23
      3) Update A33
      A33 <-- (A33, A23, A32)
III) Triangular solve for unknown
```



LAPACK GE Algorithm

```
Choose appropriate size for bk
for ib = 1 to n-1 step bk
        point to the end of block of bk columns
        end = min (ib+bk-1,n)
        for l = ib to end
            find and record k where
                    |A(k,i) | = max | A(j,i)|
                    if |A(k,i)| = 0, exit with a warning, A is singular
                    if l not equal to k, swap rows of i and k of A
                    A(i+1:n,i) = A(i+1:n,i) /A(i,i)
                    A(i+1:n,l+1:end) = A(i+1:n,i+1:end) - A(i+1:n,i)* A(i,i+1:end)
        Let LL be the bk-by-bk lower triangular matrix whose subdiagonal entries are
        stored in A(ib:end, ib:end), and with 1s on the diagonal. Do delayed update
        of A(ib:end , end +1 : n) by solving n-end triangular system
        A(ib:end,end+1:n) =  LL \ A(ib:end, end+1 : n)

        Do delayed update of the rest of matrix using matrix-matrix multiplication
        A(end+1:n,end+1:n) = A(end+1:n,end+1:n)-
                                  A(end+1:n,ib:end)*A(ib:end,end+1:n)
```

Example (1)

- **Solve the following system of linear equations**

$$\begin{bmatrix} 2 & 0 & -1 & 1 & 0 & 2 \\ 4 & 1 & -3 & 4 & 1 & 4 \\ 0 & -1 & 2 & -1 & -3 & -1 \\ 2 & -1 & 0 & 0 & 0 & 0 \\ -2 & 0 & 2 & -2 & -3 & 0 \\ -2 & 0 & -1 & -3 & 5 & 0 \end{bmatrix} \begin{bmatrix} x1 \\ x2 \\ x3 \\ x4 \\ x5 \\ x6 \end{bmatrix} = \begin{bmatrix} -2 \\ -3 \\ 8 \\ 3 \\ 0 \\ -18 \end{bmatrix}$$

$$2x_1 - x_3 + x_4 + 2x_6 = -2$$
$$4x_1 + x_2 - 3x_3 + 4x_4 + x_5 + 4x_6 = -3$$
$$-x_2 + 2x_3 - x_4 - 3x_5 - x_6 = 8$$
$$2x_1 - x_2 = 3$$
$$-2x_1 + 2x_3 - 2x_4 - 3x_5 = 0$$
$$-2x_1 - x_3 - 3x_4 + 5x_5 = -18$$

Example (2)

- **Choose the column block size bk = 2, so ib = 1, 3, and 5**
- **For b = 2, ib = 1, n = 6, end = 2**

1) For i = ib to end (i = 1,2)

i = 1	**a)**	**A(i+1: n,I) = A(i+1: n, I) / A(i,i) => A(2:6, 1)=A(2:6, 1) / A(1,1)**
		A(i+1:n,i+1:end) = A(i+1:n, i+1:end) - A(i+1:n, I) * A(i, i+1:end)
	b)	**only update columns i+1 (2) to end (2) , so only column 2**
		A(2:6, 2:2) = A(2:6, 2:2) - A(2:6, 1) * A(1, 2:2)
i = 2	**a)**	**A(3:6, 1) = A(3:6, 1) / A(2, 2), since A(2,2) = 1 =>** DONE

1a) $\begin{array}{|c|}\hline 4\\ \hline 0\\ \hline 2\\ \hline -2\\ \hline -2\\ \hline\end{array} \; / \; 2 = \begin{array}{|c|}\hline 2\\ \hline 0\\ \hline 1\\ \hline -1\\ \hline -1\\ \hline\end{array}$

A(2:6,1) / A(1,1) = A(2:6,1)

1b) $\begin{array}{|c|}\hline 1\\ \hline -1\\ \hline -1\\ \hline 0\\ \hline 0\\ \hline\end{array} - \begin{array}{|c|}\hline 2\\ \hline 0\\ \hline 1\\ \hline -1\\ \hline -1\\ \hline\end{array} \boxed{0} = \begin{array}{|c|}\hline 1\\ \hline -1\\ \hline -1\\ \hline 0\\ \hline 0\\ \hline\end{array}$

A(2:6, 2:2) - A(2:6, 1) * A(1, 2:2)= A(2:6,2)

Example (3)

- **For bk = 2 , n = 6, end = 2, ib=1**

  2) Do delayed update of A(ib:end, end+1:n) by solving n-end triangular system

  A(1:2, 3:6) = LL \ A(1:2, 3:6)

$$LL = \begin{array}{|c|c|}\hline 1 & 0 \\ \hline 2 & 1 \\ \hline\end{array} \qquad UU= \begin{array}{|c|c|}\hline 2 & 0 \\ \hline 0 & 1 \\ \hline\end{array}$$

$$\underset{LL}{\begin{array}{|c|c|}\hline 1 & 0 \\ \hline 2 & 1 \\ \hline\end{array}} * \underset{\text{new } A(1:2,\ 3:6)}{\begin{array}{|c|c|c|c|}\hline ? & ? & ? & ? \\ \hline ? & ? & ? & ? \\ \hline\end{array}} = \underset{A(1:2,\ 3:6)}{\begin{array}{|c|c|c|c|}\hline -1 & 1 & 0 & 2 \\ \hline -3 & 4 & 1 & 4 \\ \hline\end{array}}$$

$$\Rightarrow \begin{array}{|c|c|c|c|}\hline ? & ? & ? & ? \\ \hline ? & ? & ? & ? \\ \hline\end{array} = \begin{array}{|c|c|c|c|}\hline -1 & 1 & 0 & 2 \\ \hline -1 & 2 & 1 & 0 \\ \hline\end{array} = A(1:2,\ 3:6)$$

Example (4)

- **For bk = 2, ib = 1, n = 6, end = 2**
  2) Do delayed update of rest of matrix using matrix-matrix multiplication
  A(end+1:n, end+1:n) = A(end+1:n, end+1:n) -A(end+1:n, ib:end) *A(ib:end, end+1:n)
  A(3:6, 3:6) = A(3:6, 3:6) -A(3:6, 1:2) * A(1:2, 3 : 6)

$$\begin{bmatrix} 2 & -1 & -3 & -1 \\ 0 & 0 & 0 & 0 \\ 2 & -2 & -3 & 0 \\ -1 & -3 & -3 & 0 \end{bmatrix} - \begin{bmatrix} 0 & -1 \\ 1 & -1 \\ -1 & 0 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} -1 & 1 & 0 & 2 \\ -1 & 2 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & -2 & -1 \\ 0 & 1 & 1 & -2 \\ 1 & -1 & -3 & 2 \\ -2 & -2 & -3 & 2 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 0 & -1 & 1 & 0 & 2 \\ 4 & 1 & -3 & 4 & 1 & 4 \\ 0 & -1 & 2 & -1 & -3 & -1 \\ 2 & -1 & 0 & 0 & 0 & 0 \\ -2 & 0 & 2 & -2 & -3 & 0 \\ -2 & 0 & -1 & -3 & -3 & 0 \end{bmatrix} \Rightarrow \left[\begin{array}{cc|cccc} 2 & 0 & -1 & 1 & 0 & 2 \\ 2 & 1 & -1 & 2 & 1 & 0 \\ \hline 0 & -1 & 1 & 1 & -2 & -1 \\ 1 & -1 & 0 & 1 & 1 & -2 \\ -1 & 0 & 1 & -1 & -3 & 2 \\ -1 & 0 & -2 & -2 & -3 & 2 \end{array}\right]$$

Example (5)

- **Choose the column block size bk = 2, so ib = 1, 3, and 5**
- **For bk = 2, ib = 3, n = 6, end = 4**

1) For I = ib to end (I = 3,4)

```
i= 3  a)    A(i+1: n, i) = A( i+1: n, i) / A(i,i) => A(4:6, 3)=A(4:6, 3) / A(3,3)
      b)    only update columns i+1 (4) to end (4) , so only column 4
            A(i+1:n,i+1:end) = A(i+1:n, i+1:end) - A(i+1:n, i) * A(i, i+1:end)
                  A(4:6, 4:4) = A(4:6, 4:4) - A(4:6, 3) * A(3, 4:4)
i = 4 a)    A(5:6, 4) = A(5:6, 4) / A(4, 4),   A(4,4) = 1,  => DONE
```

1a) $\boxed{\begin{matrix}0\\1\\-2\end{matrix}} \; / \; 1 = \boxed{\begin{matrix}0\\1\\-2\end{matrix}}$

A(4:6,3) / A(3,3) = A(4:6,3)

1b) $\boxed{\begin{matrix}1\\-1\\-2\end{matrix}} - \boxed{\begin{matrix}0\\1\\-2\end{matrix}} \; \boxed{1} = \boxed{\begin{matrix}1\\-2\\0\end{matrix}}$

A(4:6,4)-A(4:6,3) * A(3,4)=A(4:6,4)

2a) $\boxed{\begin{matrix}-2\\0\end{matrix}} \; / \; 1 = \boxed{\begin{matrix}-2\\0\end{matrix}}$ A(5:6,4) = A(5:6,4)/A(4,4)

Example (6)

- **For bk = 2 , n = 6, end = 2, ib=3**

  2) Do delayed update of A(ib:end, end+1:n) by solving n-end triangular system

  A(3:4, 5:6) = LL \ A(3:4, 5:6)

$$LL = \begin{array}{|c|c|}\hline 1 & 0 \\ \hline 0 & 1 \\ \hline\end{array} \qquad UU= \begin{array}{|c|c|}\hline 1 & 1 \\ \hline 0 & 1 \\ \hline\end{array}$$

$$\underbrace{\begin{array}{|c|c|}\hline 1 & 0 \\ \hline 0 & 1 \\ \hline\end{array}}_{LL} * \underbrace{\begin{array}{|c|c|}\hline ? & ? \\ \hline ? & ? \\ \hline\end{array}}_{\text{new } A(3:4,\ 5:6)} = \underbrace{\begin{array}{|c|c|}\hline -2 & -1 \\ \hline 1 & -2 \\ \hline\end{array}}_{A(3:4,\ 5:6)}$$

$$\Rightarrow \begin{array}{|c|c|}\hline ? & ? \\ \hline ? & ? \\ \hline\end{array} = \begin{array}{|c|c|}\hline -2 & -1 \\ \hline 1 & -2 \\ \hline\end{array} = A(3:4,\ 5:6)$$

Example (7)

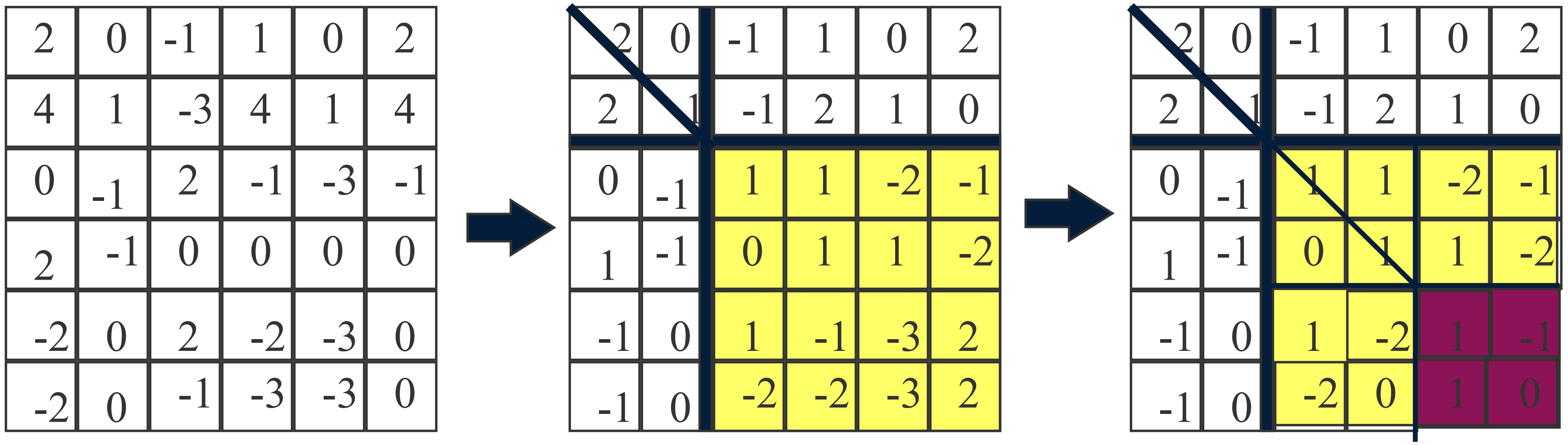
- For bk = 2, ib = 3, n = 6, end = 2

  2) Do delayed update of rest of matrix using matrix-matrix multiplication

  A(end+1:n, end+1:n) = A(end+1:n, end+1:n) -A(end+1:n, ib:end)*A(ib:end, end+1:n)

  A(5:6, 5:6) = A(5:6, 5:6) -A(5:6, 3:4) * A(3:4, 5 : 6)

$$\begin{bmatrix} -3 & 2 \\ -3 & 2 \end{bmatrix} - \begin{bmatrix} 1 & -2 \\ -2 & 0 \end{bmatrix} * \begin{bmatrix} -2 & -1 \\ 1 & -2 \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 1 & 0 \end{bmatrix}$$



$$\begin{bmatrix} 2 & 0 & -1 & 1 & 0 & 2 \\ 4 & 1 & -3 & 4 & 1 & 4 \\ 0 & -1 & 2 & -1 & -3 & -1 \\ 2 & -1 & 0 & 0 & 0 & 0 \\ -2 & 0 & 2 & -2 & -3 & 0 \\ -2 & 0 & -1 & -3 & -3 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 2 & 0 & -1 & 1 & 0 & 2 \\ 2 & 1 & -1 & 2 & 1 & 0 \\ 0 & -1 & 1 & 1 & -2 & -1 \\ 1 & -1 & 0 & 1 & 1 & -2 \\ -1 & 0 & 1 & -1 & -3 & 2 \\ -1 & 0 & -2 & -2 & -3 & 2 \end{bmatrix} \Rightarrow \begin{bmatrix} 2 & 0 & -1 & 1 & 0 & 2 \\ 2 & 1 & -1 & 2 & 1 & 0 \\ 0 & -1 & 1 & 1 & -2 & -1 \\ 1 & -1 & 0 & 1 & 1 & -2 \\ -1 & 0 & 1 & -2 & 1 & -1 \\ -1 & 0 & -2 & 0 & 1 & 0 \end{bmatrix}$$

Example (8)

- For bk = 2, ib = 5, n = 6, end = 6

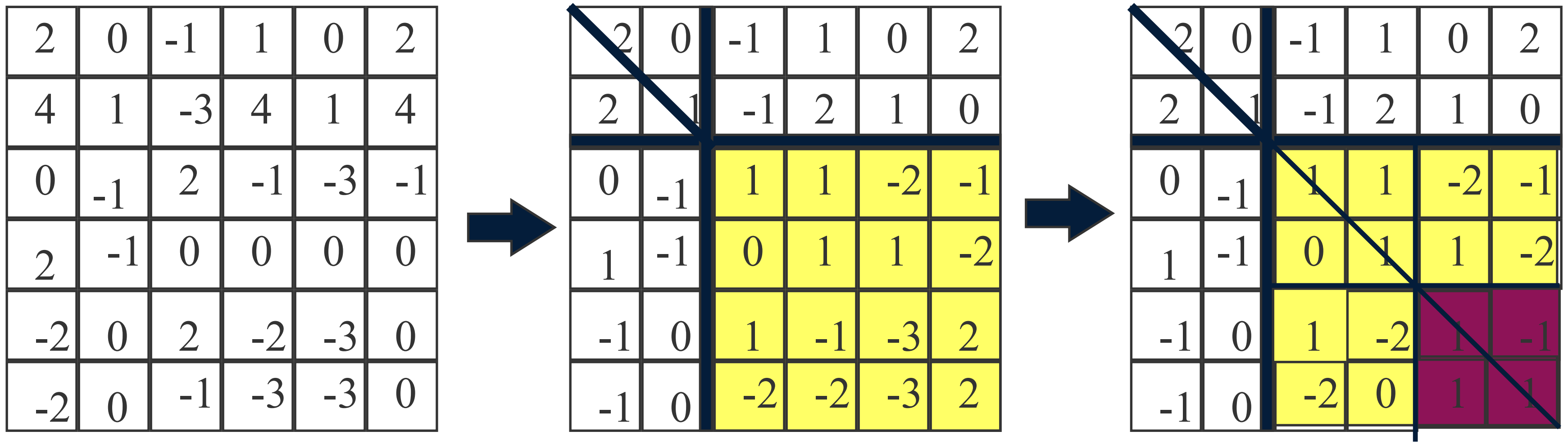
  1) For i = ib to end (i = 5,6)

     i= 1 a) A(i+1: n,i) = A(i+1: n, i) / A(i,i) => A(6, 5)=A(6, 5) / A(5,5)

     b) only update columns i+1 (2) to end (2) , so only column 2

     A(i+1:n, i+1:end) = A(i+1:n, i+1:end) - A(i+1:n, i) * A(i, i+1:end)

     A(6, 6) = A(6, 6) - A(6, 5) * A(5, 6)= 1 => DONE



Example (9)

$$\underset{A}{\begin{bmatrix} 2 & 0 & -1 & 1 & 0 & 2 \\ 4 & 1 & -3 & 4 & 1 & 4 \\ 0 & -1 & 2 & -1 & -3 & -1 \\ 2 & -1 & 0 & 0 & 0 & 0 \\ -2 & 0 & 2 & -2 & -3 & 0 \\ -2 & 0 & -1 & -3 & -3 & 0 \end{bmatrix}} = \underset{L}{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 1 & -2 & 1 & 0 \\ -1 & 0 & -2 & 0 & 1 & 1 \end{bmatrix}} * \underset{U}{\begin{bmatrix} 2 & 0 & -1 & 1 & 0 & 2 \\ 0 & 1 & -1 & 2 & 1 & 0 \\ 0 & 0 & 1 & 1 & -2 & -1 \\ 0 & 0 & 0 & 1 & 1 & -2 \\ 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}}$$

Solve $Ax = b$

$\Rightarrow L\,U\,x = b$

$\Rightarrow L\,y = b\,,\; U\,x = y$

```
For j = 1 to n
y(j) = b(j)
for j = 1 to n-1
y(j) = y(j) / L(j,j)
    for i = j+1 to n
    y(i) = y(i) - y(j)*L(i,j)
```

Triangular Solve

b	1) y	2) y	3) y	4) y	5) y	6) y	7) y	8) y	9) y	10) y	11) y
-2	-2	-2	-	-2	-2	-2	-2	-2	-2	-2	-2
-3	-3	1	2	1	1	1	1	1	1	1	1
8	8	8	8	9	9	9	9	9	9	9	9
3	3	5	5	6	6	6	6	6	6	6	6
0	0	-2	-2	-2	-2	-11	-11	1	1	1	1
-18	-18	-20	-20	-20	-20	-2	-2	-2	-2	-3	-3

x	11) x	10) x	9) x	8) x	7) x	6) x	5) x	4) x	3) x	2) x	1) x
1	2	2	2	2	2	4	4	4	4	-2	-2
-1	-1	-1	-1	-1	-1	3	3	1	1	1	1
0	0	0	0	0	0	2	2	6	6	9	9
2	2	2	2	2	2	2	2	0	0	6	6
-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	1	1
-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3

LAPACK GE Solver

- Driver subroutine to compute the solution of a real system of linear equations, Ax=b
- SUBROUTINE DGESV(N, NRHS, A, LDA, IPIV, B, LDB, INFO)
  - N : The order of the matrix A
  - NRHS : The number of right hand side, the number of columns of b
  - A : matrix A, dimension (LDA, N), on entry, the NxN coefficient matrix A, on exit, the factors L and U from factorization
  - LDA : The leading dimension of the array A
  - IPIV : The pivot indices that define the permutation matrix P
  - B : On entry, the right hand side of b, on exit, the solution x
  - LDB : The leading dimension of the array b,
  - INFO : output info, 0 = successful exit
- The DGESV subroutine calls the DGETRF subroutine which does the LU factorization and the DGETRS which solves the triangular systems.

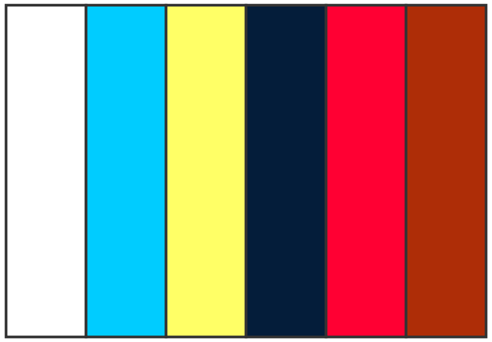
Scalable Linear Algebra Package (ScaLAPACK)
www.netlib.org/scalapack

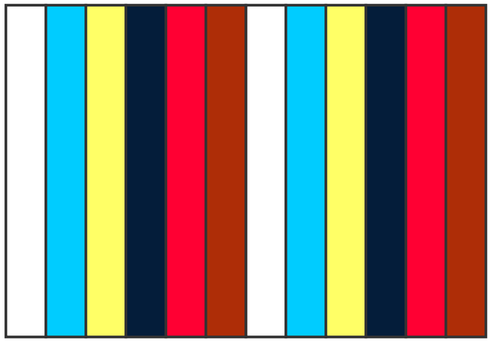
Data Layout

- ScaLAPACK is an extension of the LAPACK subroutines to perform on distributed memory parallel computers or a network of workstations running PVM or/and MPI.
- Data layout of matrices on processors will strongly affect the performance of an algorithm. There are primarily four ways to partition a matrix
- Row-wise block or column-wise block partitioning
- Row-wise block cyclic or column block cyclic partitioning
- 2D block block partitioning
- 2D block cyclic partitioning

Data Distribution

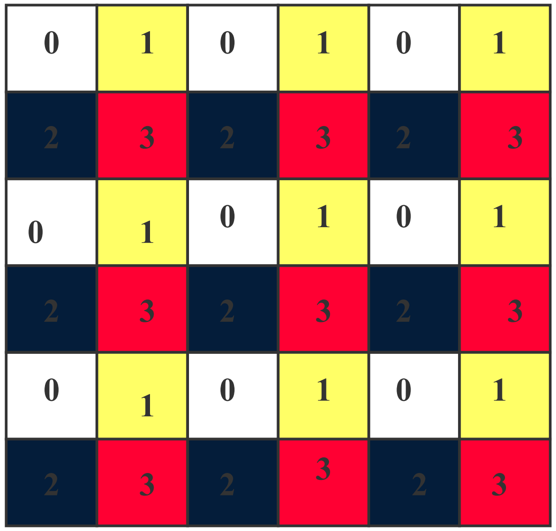
- Column Blocked Layout:
  - In this layout, a block of columns of matrix A is stored per processor as shown below.
  - This layout has the same disadvantage of the row-wise stripe partition because as soon as the first few columns have completed the elimination, the processors storing those columns remain idle for the rest of the elimination process.
- Column Block Cyclic Layout :
  - This layout tries to address the problem of load balancing by assigning blocks of columns of matrix A to processors in a cyclic fashion. However, this layout has the disadvantage that the factorization of A(ib:n, ib:end) will take place perhaps in just one processor. This would be a serial bottleneck


Column block Layout


Column Block Cyclic layout

2D Block Cyclic Layout

- The Row and Column (2D) Block Cyclic Layout will be a good compromise between the Block and Cyclic Layouts. It will alleviate the problem of load balancing and avoid the situation of a serial bottleneck. Two dimensional block structures allows efficient implementation of BLAS3 update of A(ib;end , end+1:n)



0	1	0	1	0	1
2	3	2	3	2	3
0	1	0	1	0	1
2	3	2	3	2	3
0	1	0	1	0	1
2	3	2	3	2	3

2D Block Cyclic Layout

ScaLAPACK
(www.netlib.org/scalapack)

- ScaLAPACK (version 1.7) is an extension of LAPACK using PVM or MPI on parallel computers.
- It chooses 2D block cyclic data distribution to optimize BLAS3 operations.
- It is composed of LAPACK, BLAS, PBLAS, and BLACS.
- The BLACS, Blasic Linear Algebra Communication Subprograms, are a message passing library designed for linear algebra.
- PBLAS is a set of parallel basic linear algebra subroutines similar to BLAS.
- There are four basic steps to call a ScaLAPACK routine.
  - **Initialize the process grid (BLACS)**
  - **Distribute the matrix on the process grid (DESCINIT)**
  - **Call ScaLAPACK driver routine**
  - **Release the process grid (BLACS)**

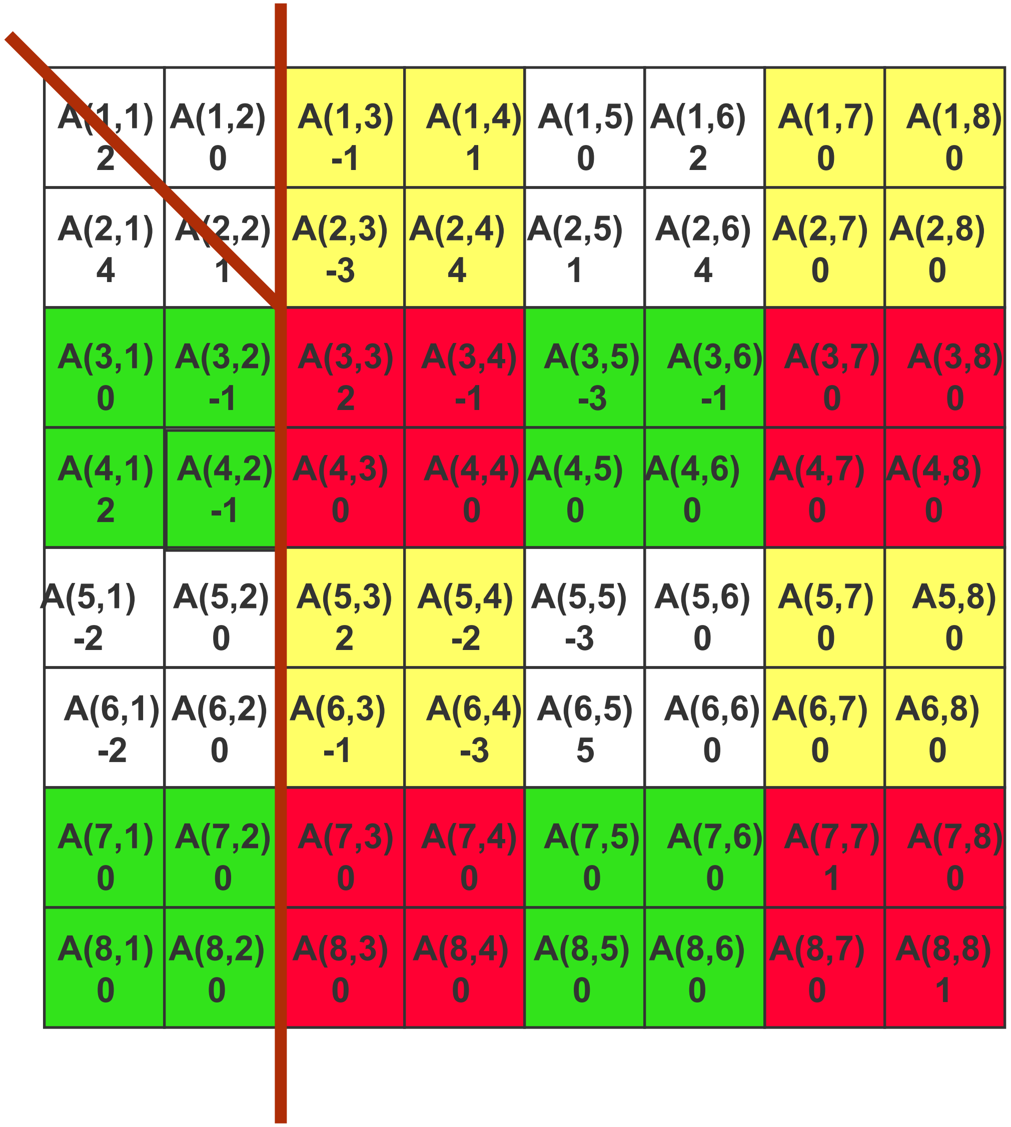
Solve a System of Equations

- General matrix factorization
  - call PDGETRF(M, N, A, IA, JA, DESC_A, IPVT, INFO)
- General matrix solve
  - call PDGETRS(TRANSA, N, NRHS, A, IA, JA, DESC_A, IPVT, B, IB, JB, ESC_B, INFO)

$$
\begin{aligned}
&2x_1 - x_3 + x_4 + 2x_6 = -2\\
&4x_1 + x_2 - 3x_3 + 4x_4 + x_5 + 4x_6 = -3\\
&-x_2 + 2x_3 - x_4 - 3x_5 - x_6 = 8\\
&2x_1 - x_2 = 3\\
&-2x_1 + 2x_3 - 2x_4 - 3x_5 = 0\\
&-2x_1 - x_3 - 3x_4 + 5x_5 = -18\\
&x_7 = 1\\
&x_8 = 1
\end{aligned}
$$

2D Block Cyclic Distribution (Scalapack)

- Consider an 8 x 8 system of linear equations using a 2D blocked cyclic data distribution
- Matrix A is first decomposed into 2x2 blocks starting at its upper left corner, bk=2.
- These blocks are then uniformly distributed across a 2x2 processor grid, nprow = npcol =2.
- There are 4 processes in the 2D process grid, nbrow = nbcol = 2.



A(1,1) 2	A(1,2) 0	A(1,3) -1	A(1,4) 1	A(1,5) 0	A(1,6) 2	A(1,7) 0	A(1,8) 0
A(2,1) 4	A(2,2) 1	A(2,3) -3	A(2,4) 4	A(2,5) 1	A(2,6) 4	A(2,7) 0	A(2,8) 0
A(3,1) 0	A(3,2) -1	A(3,3) 2	A(3,4) -1	A(3,5) -3	A(3,6) -1	A(3,7) 0	A(3,8) 0
A(4,1) 2	A(4,2) -1	A(4,3) 0	A(4,4) 0	A(4,5) 0	A(4,6) 0	A(4,7) 0	A(4,8) 0
A(5,1) -2	A(5,2) 0	A(5,3) 2	A(5,4) -2	A(5,5) -3	A(5,6) 0	A(5,7) 0	A5,8) 0
A(6,1) -2	A(6,2) 0	A(6,3) -1	A(6,4) -3	A(6,5) 5	A(6,6) 0	A(6,7) 0	A6,8) 0
A(7,1) 0	A(7,2) 0	A(7,3) 0	A(7,4) 0	A(7,5) 0	A(7,6) 0	A(7,7) 1	A(7,8) 0
A(8,1) 0	A(8,2) 0	A(8,3) 0	A(8,4) 0	A(8,5) 0	A(8,6) 0	A(8,7) 0	A(8,8) 1

Data Distribution on Local Processors

- The leading dimension of local process grid, LLD, are the same (in this case) and is equal to 4
- The number of rows of matrix A that a process own (in this case) is 4.
- The number of columns of matrix A that a process own is 4.
- Process (0,0) is chosen as the process containing the first matrix entry in its local memory, thus, the process row over which the first row of matrix A is distributed, RSRC=0, and process column over which the first column of matrix A is distributed, CSRC=0

Process grid (0,0)

A(1,1) 2	A(1,2) 0	A(1,5) 0	A(16) 2
A(2,1) 4	A(2,2) 1	A(2,5) 1	A(2,6) 4
A(5,1) -2	A(5,2) 0	A(5,5) -3	A(5,6) 0
A(6,1) -2	A(6,3) 0	A(6,5) 5	A(6,6) -3

Process grid (0,1)

A(1,3) -1	A(1,4) 1	A(1,7) 0	A(1,8) 0
A(2,3) -3	A(2,4) 4	A(2,7) 0	A(2,8) 0
A(5,3) 2	A(5,4) -2	A(5,7) 0	A(5,8) 0
A(6,3) -1	A(6,4) 3	A(6,7) 0	A(5,8) 0

Process grid (1,0)

A(3,1) 0	A(3,2) -1	A(3,5) -3	A(3,6) -1
A(4,1) 2	A(4,2) -1	A(4,5) 0	A(4,6) 0
A(7,1) 0	A(7,2) 0	A(7,5) 0	A(7,6) 0
A(8,1) 0	A(8,2) 0	A(8,5) 0	A(8,6) 0

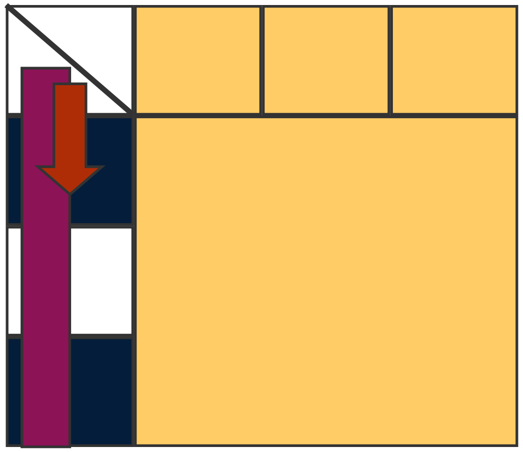
Process grid (1,1)

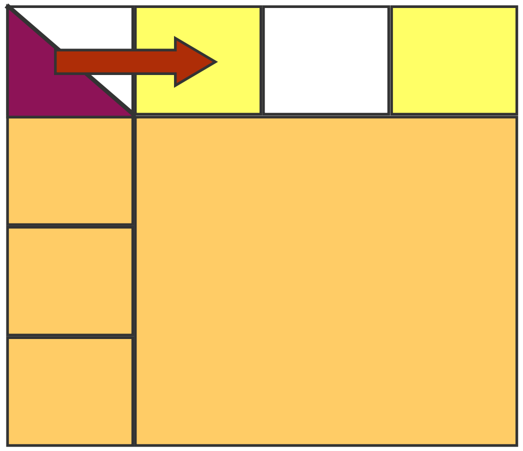
A(3,3) 2	A(3,4) -1	A(3,7) 0	A(3,8) 0
A(4,3) 0	A(4,4) 0	A(4,7) 0	A(4,8) 0
A(7,3) 0	A(7,4) 0	A(7,7) 1	A(7,8) 0
A(8,3) 0	A(8,4) 0	A(8,7) 0	A(8,8) 1

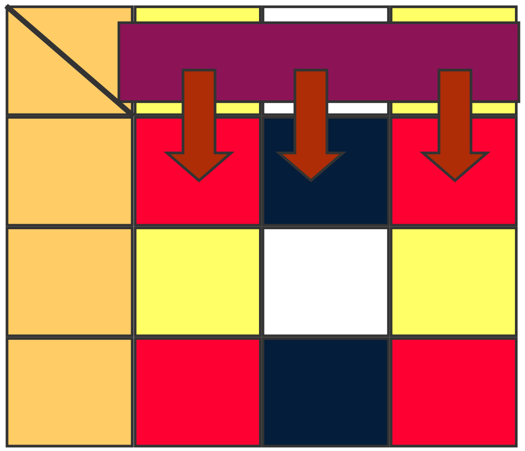
ScaLAPACK GE Subroutine

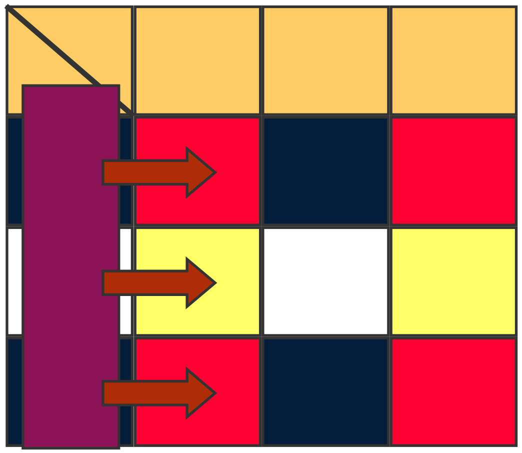
- ScaLAPACK is composed of LAPACK, BLAS, PBLAS, and BLACS.
- The BLACS, Blasic Linear Algebra Communication Subprograms, are a message passing library designed for linear algebra.
- PBLAS is a set of parallel basic linear algebra subroutines similar to BLAS.
- There are four basic steps to call a ScaLAPACK routine.
  - Initialize the process grid
  - Distribute the matrix on the process grid
  - Call ScaLAPACK driver routine
  - Release the process grid
- BLACS routines are used to initialize the process grid
- A ScaLAPACK tools routine, DESCINIT, can be used to distribute the matrix layout (or Iinitializes the Descriptor)
- A ScaLAPACK routine is called to perform a specific task
- A BLACS routine is then used to release the process grid

Distributed GE (1st sweep)

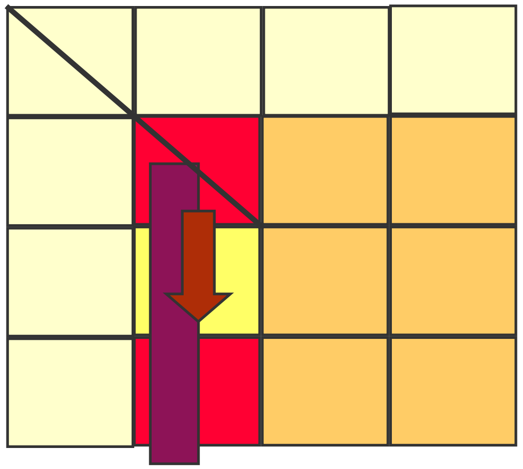

Step (1), (2), (3), (4)

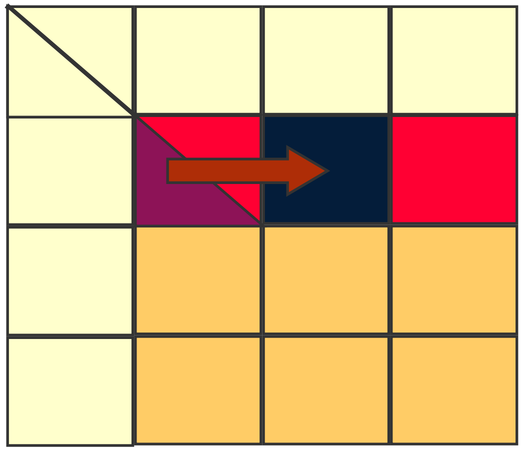

Step (7), (8)

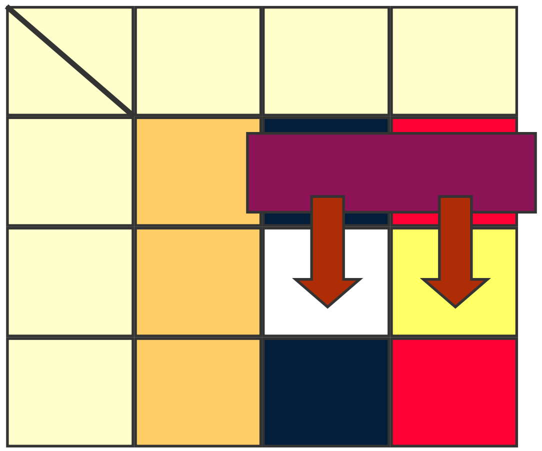

Step (9)

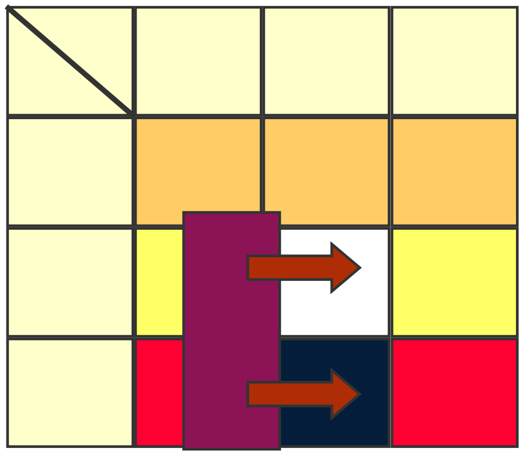

Step (10), (11)

Distributed GE (2nd sweep)

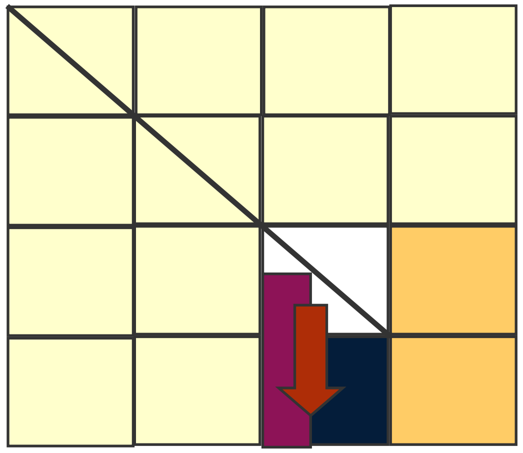

Step (1), (2), (3), (4)

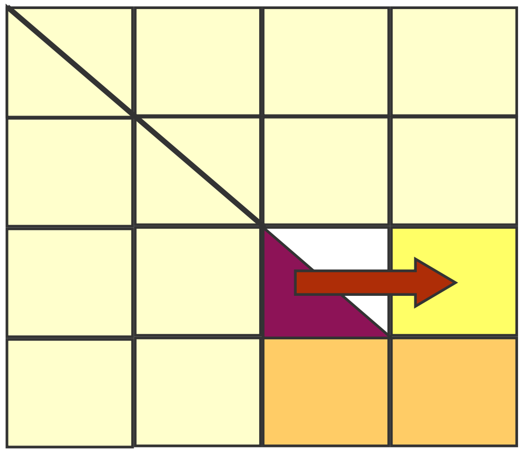

Step (7), (8)

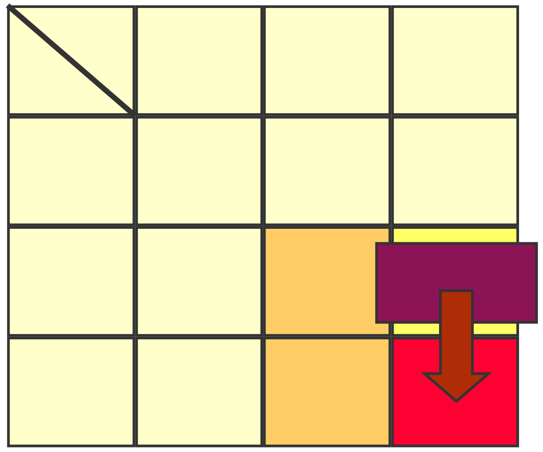

Step (9)

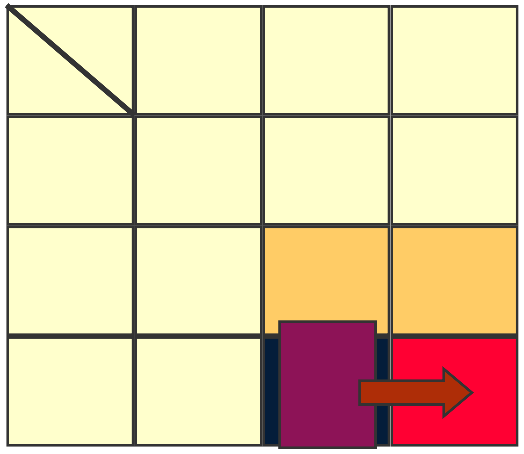

Step (10), (11)

Distributed GE (3rd sweep)


Step (1), (2), (3), (4)


Step (7), (8)


Step (9)


Step (10), (11)

ScaLAPACK Linear Solver

Process grid initialization

```
        CALL BLACS_PINFO(MYID, NPROCS)
!       Initialize the process grid, obtain system default context
        CALL BLACS_GET(-1,0,ICTXT)
!       Map the available processes to a BLACS process grid
        CALL BLACS_GRIDINIT(ICTXT,'Row-major',NPROW,NPCOL)
!       Query the process grid to identify each process's coordinate, (MYROW, MYCOL)
        CALL BLACS_GRIDINFO(ICTXT,NPROW,NPCOL,MYROW,MYCOL)
```

Data Distribution

- All global matrices must be distributed on the process grid prior
- CALL DESCINIT(DESCA, M, N, MB, NB, RSRC, CSRC, ICTXT, LLDA, INFO)
- CALL DESCINIT(DESCB, N, NRHS, NB, NBRHS, RSRC, CSRC, ICTXT, LLDB, INFO)

Call the solver routine

- CALL PDGESV(N,NRHS,A,IA,JA,DESCA,IPIV,B,IB,JB,DESCB,INFO)
- CALL PDGETRF, CALL PDGETRS

Release the process grid

- CALL BLACS_GRIDEXIT(ICONTXT)
- CALL BLACS_EXIT(0)

SCALAPACK Exercise

- TRAINING/MPI_WORKSHOP/Fortran/SCALAPACK
- Use PE-Intel MKL library. Makefile-intel
- Use PE-gnu MKL Library, Makefile-gnu
- https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor
- Use PE-gnu Scalapack, lapack, ATLAS, Makefile-scal
- mpirun –np 4 ./xdlu < LU.dat

HPL Exercise

- TRAINING/MPI_WORKSHOP/C/HPCC
- tar zxvf hpl-2.2.tar.gz
- cd hpl-2.2
- cp setup/Make.Linux_Intel64 .
- Use PE-intel MKL Library, Makefile
- Change TOPdir to your current directory in Make.Linux_Intel64, use ‘pwd’ to show your current directory
- Make arch=Linux_Intel64
- mpirun –np 4 ./xhpl

Krylov Subspace Solvers in Parallel
Matrix Vector Multiple (HPCG)
PETSc, HYPRE, TRILINOS

Iterative Methods

- Iterative methods are generally used to solve system of equations which is too large to be handled by direct methods. Iterative methods do not guarantee a solution for every system of equations. However,, when they do yield a solution, they are usually less expensive than direct methods.
- A sequence of approximations to the solution vector is usually generated by performing a matrix vector multiplication to the iterative matrix T or A matrix
- Iterative methods can be expressed in the simple form, $x^k = T x^{k-1} + c$. There are two main types of iterative methods, stationary iterative methods and non-stationary iterative methods, dependent on the nature of T and c during the iteration process.
- Traditional iterative methods such as Jacobi, Gauss Siedel, and SOR methods are stationary methods which are applicable to limited problems.
- Conjugate Gradient methods and Conjugate Gradient look-alike methods which are known as Krylov Subspace Method are the most widely used iterative methods nowadays.

$$\boxed{x_i = F(r_{0,} Ar_0, A^2 r_0, \cdots, A^{i-1} r_0) \in K^i(A; r_0)}$$

Iterative Methods

$$Ax = b \qquad r = b - Ax$$

$$\text{write} \qquad A = I - (I - A)$$

$$(I - (I - A))x = b$$

$$x = (I - A)x + b$$

$$\text{Iterative} \quad \text{Method}$$

$$\begin{aligned}
x_i &= b + (I - A)x_{i-1} \\
&= x_{i-1} + b - Ax_{i-1} \\
&= x_{i-1} + r_{i-1} \\
&= x_{i-2} + r_{i-2} + r_{i-1} \\
&= x_{i-3} + r_{i-3} + r_{i-2} + r_{i-1} \\
&= \quad \vdots \\
x_i &= x_0 + r_0 + r_1 + \cdots + r_{i-1}
\end{aligned}$$

$$x_i = x_0 + r_0 + r_1 + \cdots + r_{i-1} \quad ; \quad x_0 = 0$$

$$x_i = x_{i-1} + r_{i-1}$$

$$Ax_i = Ax_{i-1} + Ar_{i-1}$$

$$b - Ax_i = b - Ax_{i-1} - Ar_{i-1}$$

$$r_i = r_{i-1} - Ar_{i-1}$$

$$r_i = (I - A)r_{i-1}$$

$$r_i = (I - A)^i r_0$$

$$x_i = x_0 + (I - A)r_0 + \cdots + (I - A)^{i-1} r_0$$

$$x_i \in \{r_0, Ar_0, A^2 r_0, \cdots, A^{i-1} r_0\}$$

$$\boxed{x_i = F(r_{0,} Ar_0, A^2 r_0, \cdots, A^{i-1} r_0) \in K^i(A; r_0)}$$

Conjugate Gradient Method

- The value of iterates x is actually a function of initial value of x and power of matrix vector product of residual r and matrix A. We call that Krylov Subspace, and thus x belongs to the Krylov Subspace.
- Conjugate Gradient method is to find an x by systemically searching through the Krylov Subspace. Minimization of the residuals in the Krylov Subspace produce sequences of values of α and p which build the value of x.
- In every iteration of the method, the approximate solution x is updated with respect to a search direction,p, multiple by a constant α, x= x + α p. Minimization of the error lead to a specific choice of α and p which can be efficiently constructed in a three term recursion relation.

$$
\begin{aligned}
x &= x_{k-1} + \alpha_k p_k \\
r_k &= b - Ax_k \\
&= b - A(x_{k-1} + \alpha_k p_k) \\
&= (b - Ax_{k-1}) + \alpha_k p_k \\
&= r_{k-1} - \alpha_k p_k
\end{aligned}
$$

and

$$
\begin{aligned}
p_1 &= r_0 = b \\
p_{k+1} &= r_k + \frac{\| r_k \|_2^2 \; p_k}{\| r_{k-1} \|_2^2} \\
\alpha_k &= \frac{\| r_{k-1} \|_2^2}{p_k^T A p_k}
\end{aligned}
$$

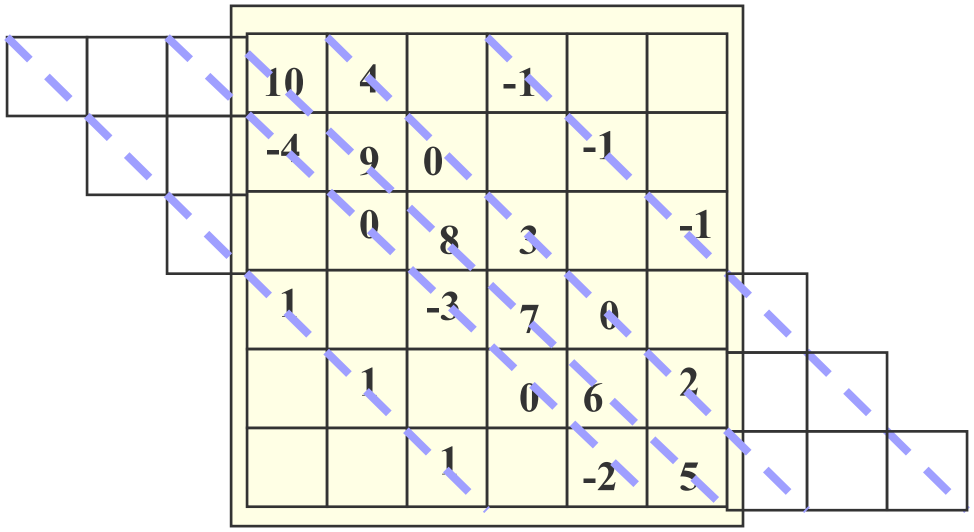
CG Algorithm – A Parallel Solver (MV)

```
i     = 0
x(0)  = 0
r(0)  = b - A*x(0)  = b
$\phi$(0)  = $r^T$(0)r(0)
while (f(i) > tolerance) and (i < maximum iteration)
do
    if (i = 0) then p(1) = r(0)
    else p(i+1) = r(i) + $\phi$(i)*p(i) / $\phi$(i-1)
    i     = i + 1
    - matrix-vector multiplication
    w(i) = A*p(i)
    - vector dot product
    $\alpha$(i) = $\phi$(i-1) / $p^T$(i)*w(i)
    x(i) = x(i-1) + $\alpha$(i)*p(i)
    r(i) = r(i-1) - $\alpha$(i)*w(i)
    - vector dot product
    $\phi$(i) = $r^T$(i)*r(i)
end while
x = x(i)
```

Libraries for Sparse Matrices

Storage Schemes of Sparse Matrix

- There are a lot of different sparse matrix storage schemes. We will introduce a few common types which can be used for general sparse matrix. Sparse storage generally consists of several vectors which stores the nonzero values of the matrix and pointers of location of the nonzero values. Obviously, the most logical and efficient storage scheme for this block tridiagonal matrix will be the Diagonal Storage scheme. The scheme stores the values of the matrix using individual vector array for each diagonal and a position pointer relative to the main-diagonal of the matrix.



```
aval(:,1)=(0,0,0,1,1,1) , apos(1)=-3
aval(:,2)=(0,-4,0,-3,0,-2), apos(2)=-1
aval(:,3)=(10,9,8,7,6,5), apos(3)=0
aval(:,4)=(4,0,3,0,2,0), apos(4)=1
aval(:,5)=(-1,-1,-1,0,0,0), apos(5)=3
```

Matrix vector product:

```
do I=1,N
    do k=1,5
    w(I) = w(I) + aval(I,k) * p(I-apos(k))
    enddo
enddo
```

Coordinate Storage Scheme

- The Coordinate Storage scheme consists of three vector arrays, one stores the nonzero values, one stores the row locations of the nonzero entries, and the last one stores the the column locations of the nonzero entries. The order of storing the nonzero entries can be arbitrary, however, rowwise or columnwise storing orders are used for computing efficiency. As can be observed later, storage of one of the location pointer can be reduced.

10	4		-1		
-4	9	0		-1	
	0	8	3		-1
1		-3	7	0	
	1		0	6	2
		1		-2	5

Coordinate Storage Scheme:

aval(I) = (10,4,-1,-4,9,-1,8,3,-1,1,-3,7,1,6,2,1,-2,5)

irow(I) = (1,1,1,2,2,2,3,3,3,4,4,4,5,5,5,6,6,6)

jcol(I) = (1,2,4,1,2,5,3,4,6,1,3,4,2,5,6,3,5,6)

Matrix vector multiplication

```
do i=1,N
    w(irow(i))=w(irow(i)+aval(i)*p(icol(I))
end do
```

Compressed Row and Column Storage Schemes

- The Compressed Row Storage (CRS) scheme put the subsequent non-zeros of the matrix row in contiguous memory locations. Three vectors are used. One contains the values of the nonzero entries (aval), one stores the column number of each nonzero entries (icol), and the last one stores the pointers to the first entry of the ith row in aval and icol (jprow)
- The Compressed column Row Storage (CCS) scheme is identical to CRS scheme except the matrix nonzero entries are stored in columnwise fashion.
- Due the structural symmetry of the following example, the position indicators of the CRS and CCS are the same!

10	4		-1		
-4	9	0		-1	
	0	8	3		-1
1		-3	7	0	
	1		0	6	2
		1		-2	5

Compressed Row Storage:
aval(I) = (10,4,-1,-4,9,-1,8,3,-1,1,-3,7,1,6,2,1,-2,5)
icol(I) = (1,2,4,1,2,5,3,4,6,1,3,4,2,5,6,3,5,6)
jprow(I) = (1,4,7,10,13,16,19)

Compressed Column Storage:
aval(I) = (10,-4,1,4,9,1,8,-3,1,-1,3,7,-1,6,-2,-1,2,5)
jrow(I) = (1,2,4,1,2,5,3,4,6,1,3,4,2,5,6,3,5,6)
ipcol(I) = (1,4,7,10,13,16,19))

Matrix Vector Product for CRS and CCS

- Compressed Row Storage : w=A*p

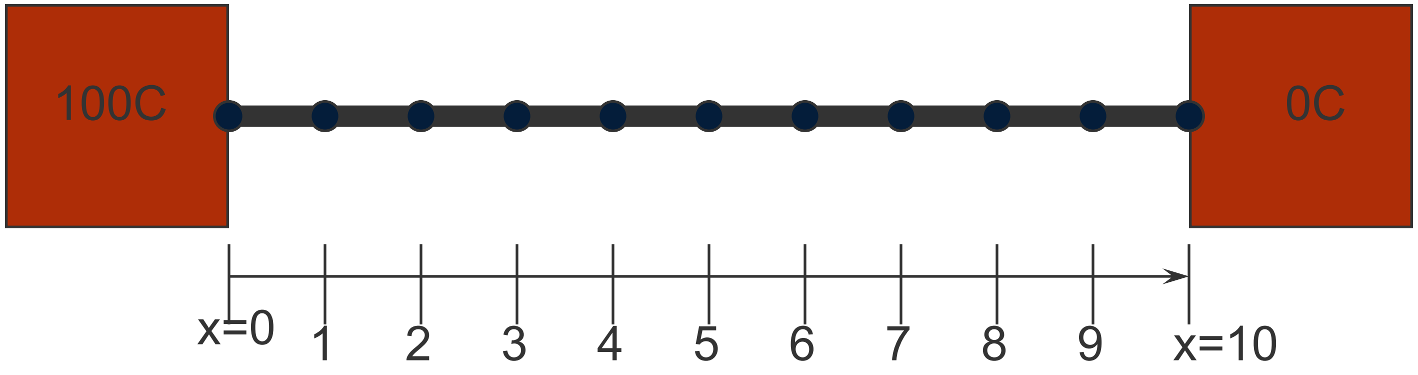
```
do I=1,NROW
    w(I)=0
    do j = jprow(I), jprow(I+1) -1
                w(I) = w(I) + aval(j) * p(icol(j))
    end do
end do
```

- Compressed Row Storage

```
Do I=1,NROW
    w(I)=0
end do
do I=1,NCOLUMN
    do j = ipcol(I), ipcol(I+1) -1
                w(jrow(j)) = w(jrow(j)) + aval(j) * p(I)
    end do
end do
```

Example – 1D Diffusion Problem

- A ten meters long iron rod is supported at both ends by two water tanks as shown. The temperatures of the water in the banks are maintained at 100C and 0C. The temperature, T(I), at any point on the iron rod is approximated by the average value of the temperatures of its neighboring points, i.e. T(1) = (T(0) + T(2))/2. As a result, the temperature of the iron rod at any location of x can be represented by the following system of equations.



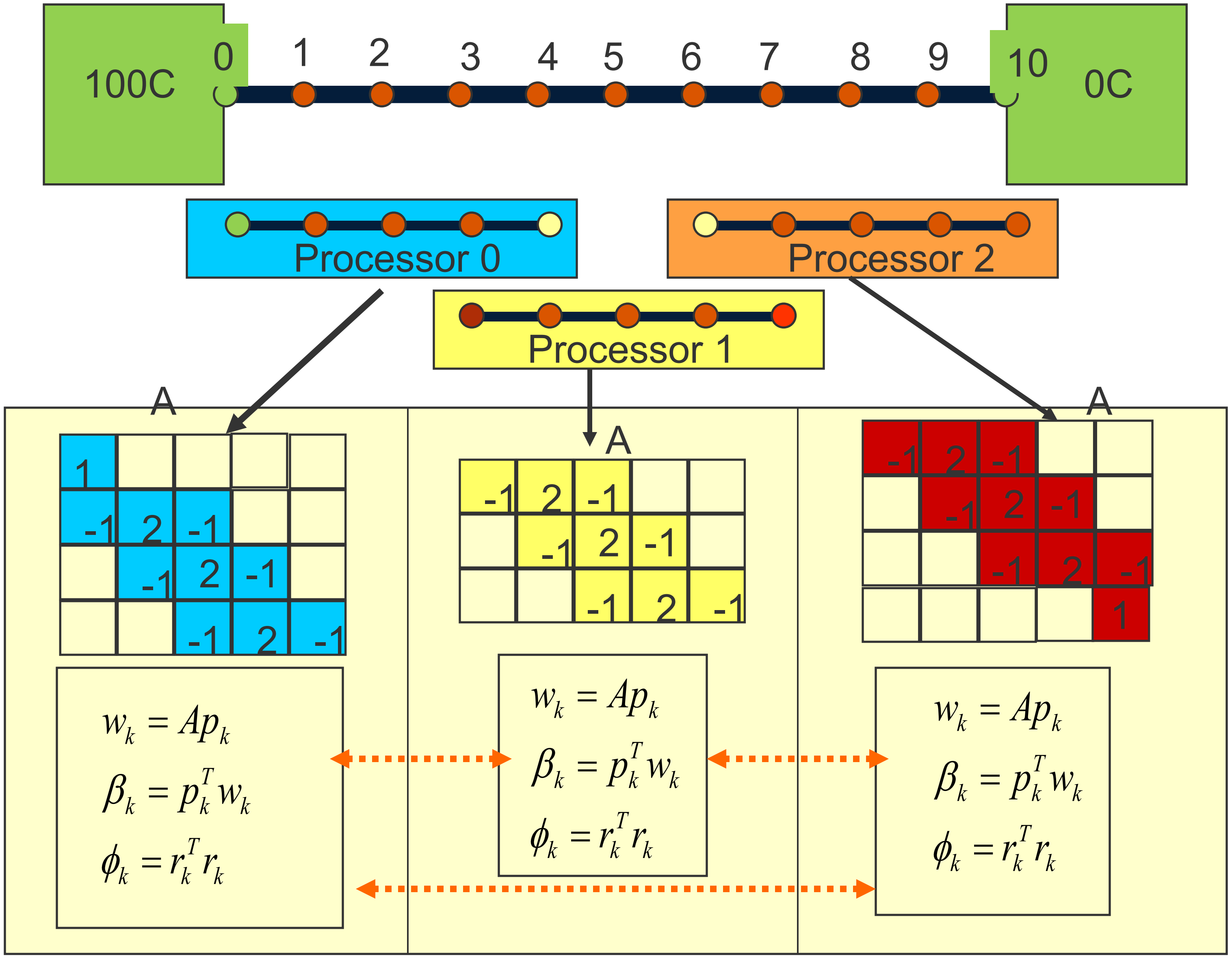
T(0) = 100, T(1)={T(2)+T(0)}/2, T(2)={T(3)+T(1)}/2,
T(3)={T(2)+T(4)}/2 T(4)={T(3)+T(5)}/2, T(5)={T(4)+T(6)}/2,
T(6)={T(5)+T(7)}/2, T(7)={T(6)+T(8)}/2, T(7)={T(6)+T(8)}/2,
T(8)={T(7)+T(9)}/2, T(9)={T(8)+T(10)}/2, T(10)=0

Resultant Matrix

$$\boxed{\frac{d^2t}{dx^2} = 0} \xrightarrow[\text{FEM}]{} \boxed{\text{Solve } A\,x = b}$$

$$\begin{array}{c|ccccccccccc|c|c|c}
 & \multicolumn{11}{c|}{A} & x & & b \\
 & 1 & & & & & & & & & & & T0 & & 100 \\
P0 & -1 & 2 & -1 & & & & & & & & & T1 & & 0 \\
 & & -1 & 2 & -1 & & & & & & & & T2 & & 0 \\
 & & & -1 & 2 & -1 & & & & & & & T3 & & 0 \\
\hline
 & & & & -1 & 2 & -1 & & & & & & T4 & & 0 \\
P1 & & & & & -1 & 2 & -1 & & & & & T5 & = & 0 \\
 & & & & & & -1 & 2 & -1 & & & & T6 & & 0 \\
\hline
 & & & & & & & -1 & 2 & -1 & & & T7 & & 0 \\
P2 & & & & & & & & -1 & 2 & -1 & & T8 & & 0 \\
 & & & & & & & & & -1 & 2 & -1 & T9 & & 0 \\
 & & & & & & & & & & & 1 & T10 & & 0
\end{array}$$

CG in Parallel

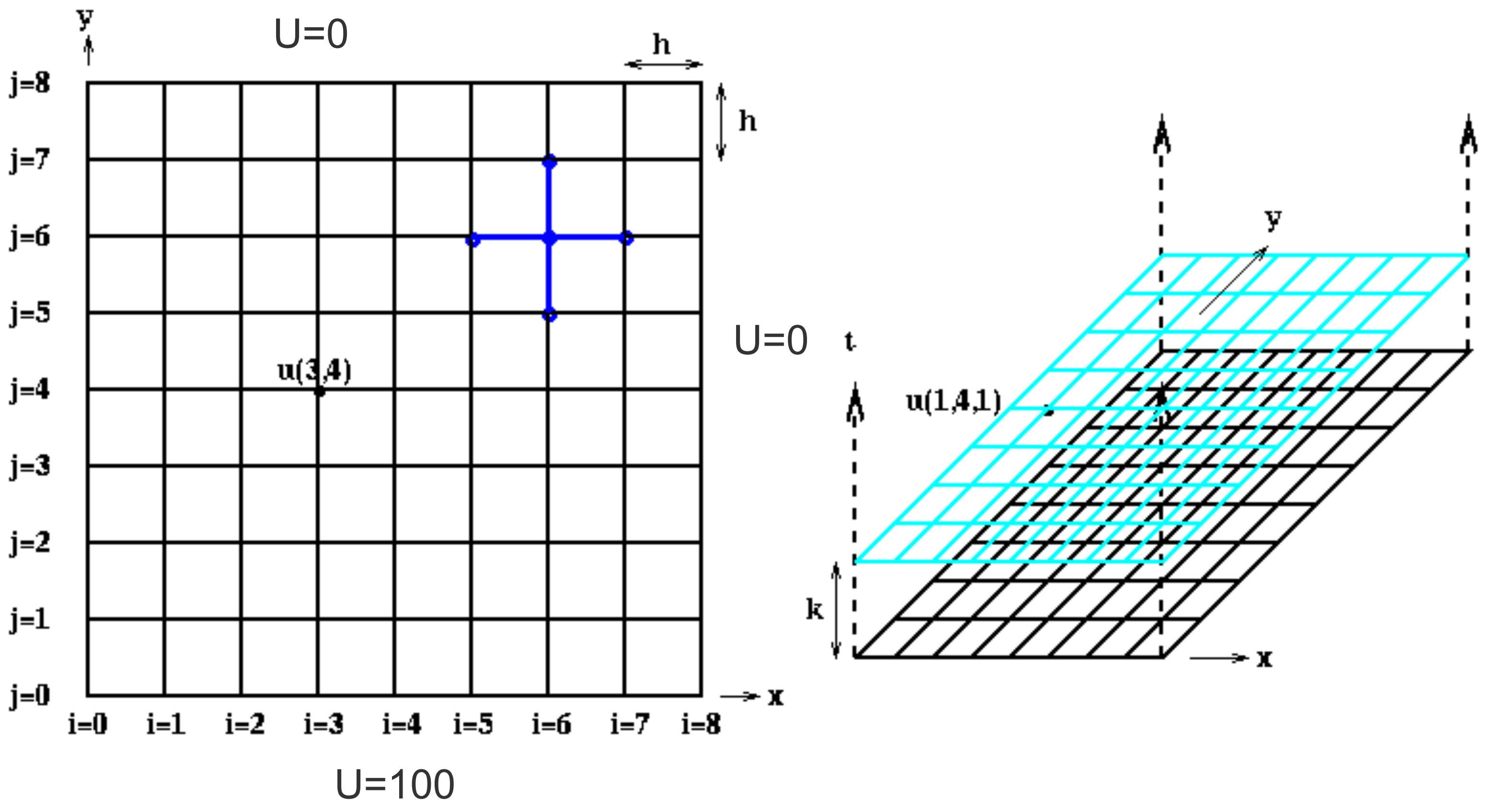


2D Heat Equation

- The one dimensional equation can be generalized to a two dimensional case. Approximation of the derivatives by Taylor's series is carried out with respect to x and y similarly as before.

$$\frac{\partial u}{\partial t} = c\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

Discretization of the 2D Heat Equation



2D Heat Equation FD Formulation

- **A 2D square plate is discretized uniformly in x and y directions with Δx= Δy=h, and the time step Δt=k, the second derivation of the u at any grid points with respect to x and y can be approximated by the average value of the neighbors in the north, south, west, and east directions.**

$$(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}) = \nabla^2 u \approx (U_{i-1,j}+U_{i+1,j}+U_{i,j-1}+U_{i,j+1}-4U_{i,j})/h^2$$

- **Hence, the 2D Forward Euler's formula is**

$$\frac{U_{i,j}^{m+1}-U_{i,j}^{m}}{k} = \frac{(U_{i-1,j}^{m}+U_{i+1,j}^{m}+U_{i,j-1}^{m}+U_{i,j+1}^{m}-4U_{i,j}^{m})}{h^2}$$

$$\Rightarrow \quad U_{i,j}^{m+1} = zU_{i-1,j}^{m}+zU_{i+1,j}^{m}+zU_{i,j-1}^{m}+zU_{i,j+1}^{m}+(1-4z)U_{i,j}^{m}$$

- **The 2D Backward Euler's formula is**

$$\frac{U_{i,j}^{m+1}-U_{i,j}^{m}}{k} = \frac{(U_{i-1,j}^{m+1}+U_{i+1,j}^{m+1}+U_{i,j-1}^{m+1}+U_{i,j+1}^{m+1}-4U_{i,j}^{m+1})}{h^2}$$

$$\Rightarrow \quad zU_{i-1,j}^{m+1}+zU_{i+1,j}^{m+1}+zU_{i,j-1}^{m+1}+zU_{i,j+1}^{m+1}-(1+4z)U_{i,j}^{m+1} = -U_{i,j}^{m}$$

Resultant matrix

- **The 2D Crank-Nicholson scheme will be the average of the explicit and implicit Euler's schemes.**

$$\frac{z}{2}(U_{i-1,j}^{m+1}+U_{i+1,j}^{m+1}+U_{i,j-1}^{m+1}+U_{i,j+1}^{m+1})-(1+2z)U_{i,j}^{m+1}=$$

$$-\frac{z}{2}(U_{i-1,j}^{m+1}+U_{i+1,j}^{m+1}+U_{i,j-1}^{m+1}+U_{i,j+1}^{m+1})-(1-2z)U_{i,j}^{m+1}=b_{i,j}^{m}$$

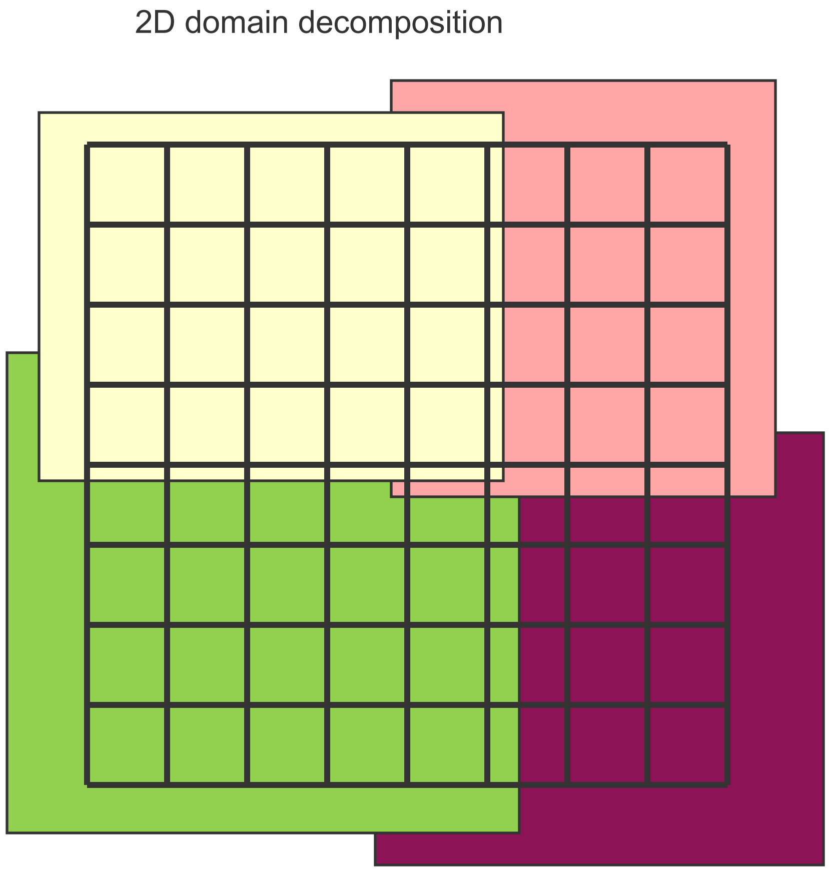
- **The result is a block tridiagonal matrix as shown below. Each block is a 9x9 matrix.**

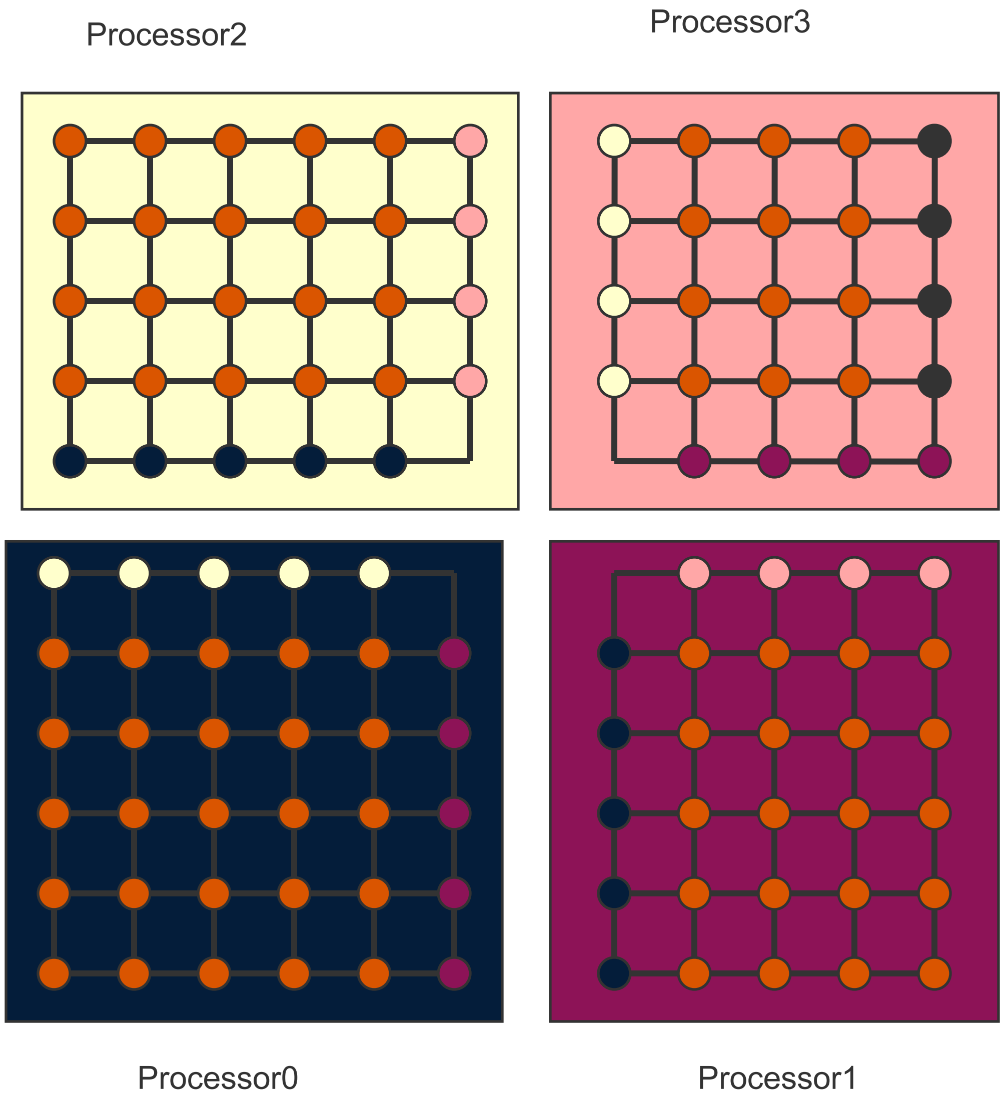
$$\begin{bmatrix} I & 0 & & & \\ -C & A & -C & & \\ & \ddots & \ddots & \ddots & \\ & & -C & A & -C \\ & & & 0 & I \end{bmatrix} \begin{bmatrix} U_{i,0}^{m+1} \\ U_{i,1}^{m+1} \\ \vdots \\ U_{i,7}^{m+1} \\ U_{i,8}^{m+1} \end{bmatrix} = \begin{bmatrix} b_{i,0}^{m} \\ b_{i,1}^{m} \\ \vdots \\ b_{i,7}^{m} \\ b_{i,8}^{m} \end{bmatrix} \quad ; \quad i=0:8$$

$$A=\begin{bmatrix} 1 & 0 & & & \\ -z/2 & 1+2z & -z/2 & & \\ & \ddots & \ddots & \ddots & \\ & & -z/2 & 1+2z & -z/2 \\ & & & 0 & 1 \end{bmatrix} \qquad C=\begin{bmatrix} 1 & & & \\ & z/2 & & & \\ & & \ddots & & \\ & & & z/2 & \\ & & & & 1 \end{bmatrix} \qquad I=identity\ matrix$$

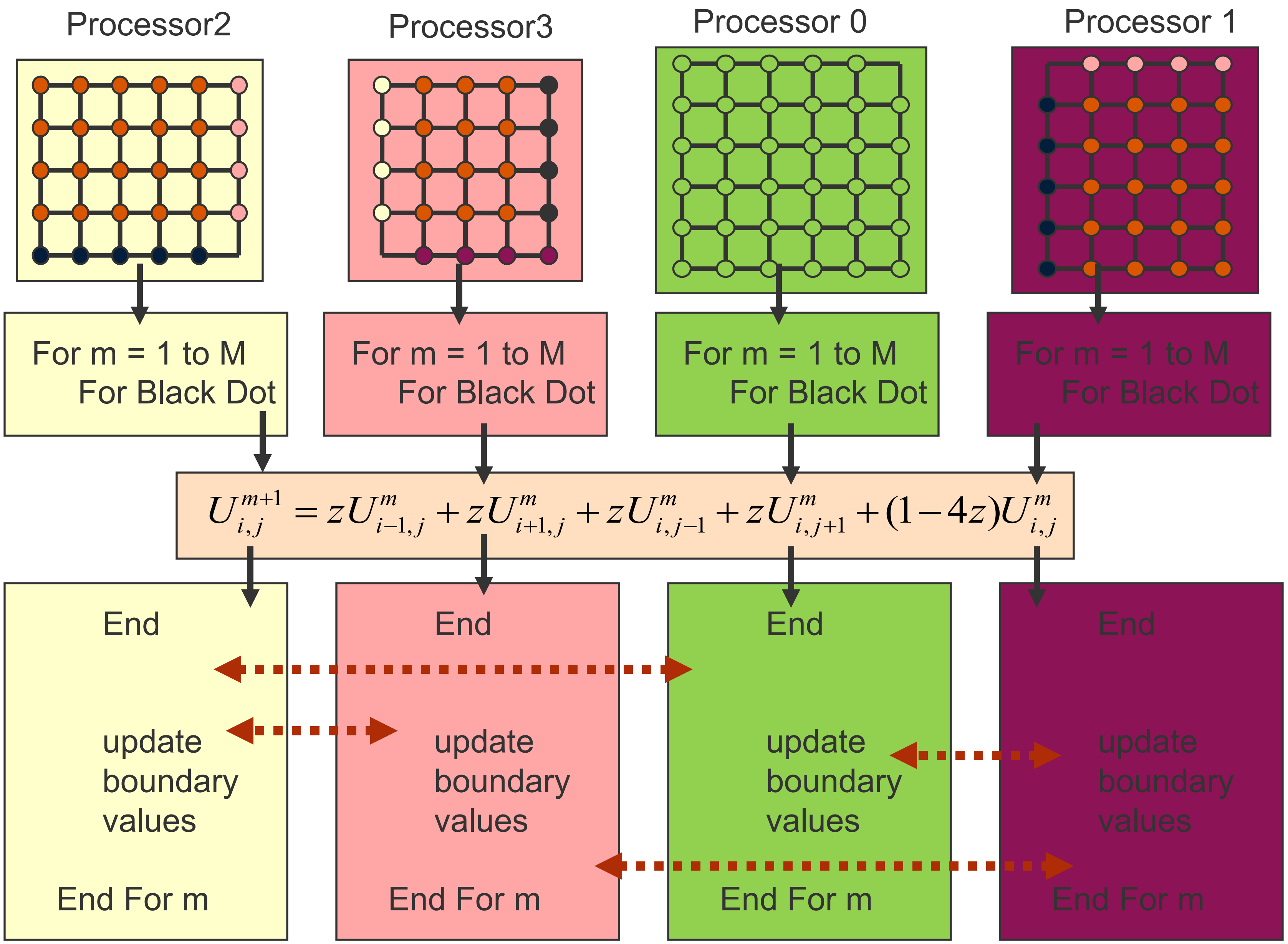
2D Domain Decomposition

- **The 2D grid is decomposed into four sections with overlapped grid points, each of them assigned to a processor.**





Parallel Implementation of Explicit Scheme



$$U_{i,j}^{m+1} = zU_{i-1,j}^{m} + zU_{i+1,j}^{m} + zU_{i,j-1}^{m} + zU_{i,j+1}^{m} + (1-4z)U_{i,j}^{m}$$

Parallel Implementation of Implicit Scheme

- **Gaussian elimination can be used to solve the linear system of equation resulting from the Backward Euler's or Crank Nicholson schemes. LAPACK and ScaLAPACK provide a band solver subroutine for solving such system of equations. Iterative solvers such as the Jacobi ,SOR, and Conjugate Gradient methods can also be used.**
- **If only the steady solution of the problem is sought. The time dependency, the partial derivative with respect to time, can be eliminated. The result is a 2D Laplace equation.**
- **The one dimensional Laplace equation was examined earlier. The resulting matrix of the 2D equation is block tridiagonal as shown below.**

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \qquad -U_{i-1,j} - U_{i+1,j} - U_{i,j-1} - U_{i,j+1} + 4U_{i,j} = b_{i,j}$$

$$\begin{bmatrix} I & 0 & & & \\ -I & A & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & A & -I \\ & & & 0 & I \end{bmatrix} \begin{bmatrix} U_{i,0} \\ U_{i,1} \\ \vdots \\ U_{i,7} \\ U_{i,8} \end{bmatrix} = \begin{bmatrix} b_{i,0} \\ b_{i,1} \\ \vdots \\ b_{i,7} \\ b_{i,8} \end{bmatrix} \quad A = \begin{bmatrix} 1 & 0 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 4 & -1 \\ & & & 0 & 1 \end{bmatrix} \quad I = identity\ matrix$$

Conjugate Gradient Method

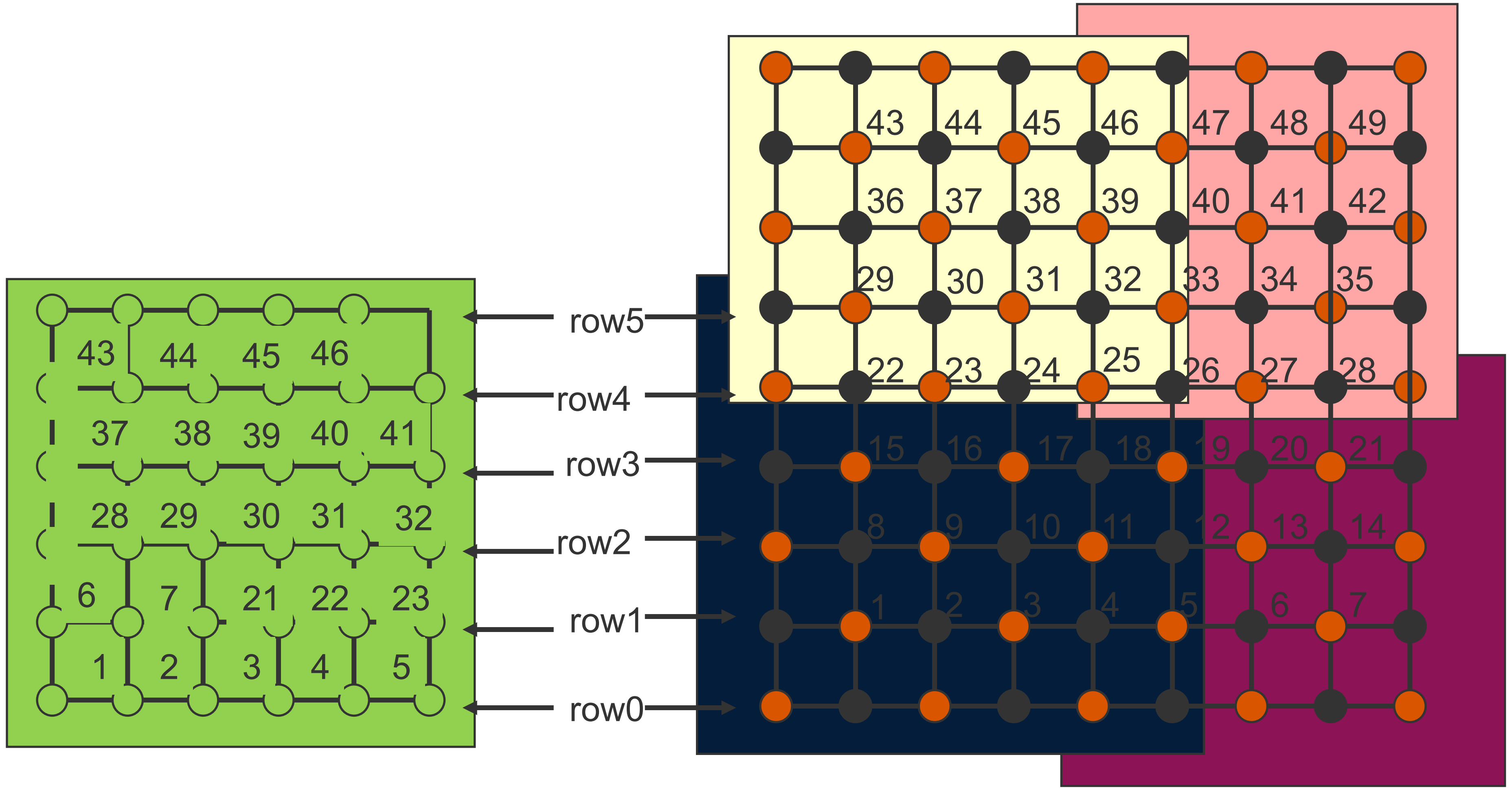
- **Conjugate gradient method can be used to solve the system of equations. To make the matrix symmetric, the boundary values are incorporated to the right hand side, only the interior nodes will be used. As a result, the system matrix will be as follows.**

$$\begin{bmatrix} T & -I & & & & & \\ -I & T & -I & & & & \\ & -I & T & -I & & & \\ & & -I & T & -I & & \\ & & & -I & T & -I & \\ & & & & -I & T & -I \\ & & & & & -I & T \end{bmatrix} \begin{bmatrix} U_{i,1} \\ U_{i,2} \\ U_{i,3} \\ U_{i,4} \\ U_{i,5} \\ U_{i,6} \\ U_{i,7} \end{bmatrix} = \begin{bmatrix} b_{i,1} \\ b_{i,2} \\ b_{i,3} \\ b_{i,4} \\ b_{i,5} \\ b_{i,6} \\ b_{i,7} \end{bmatrix}$$

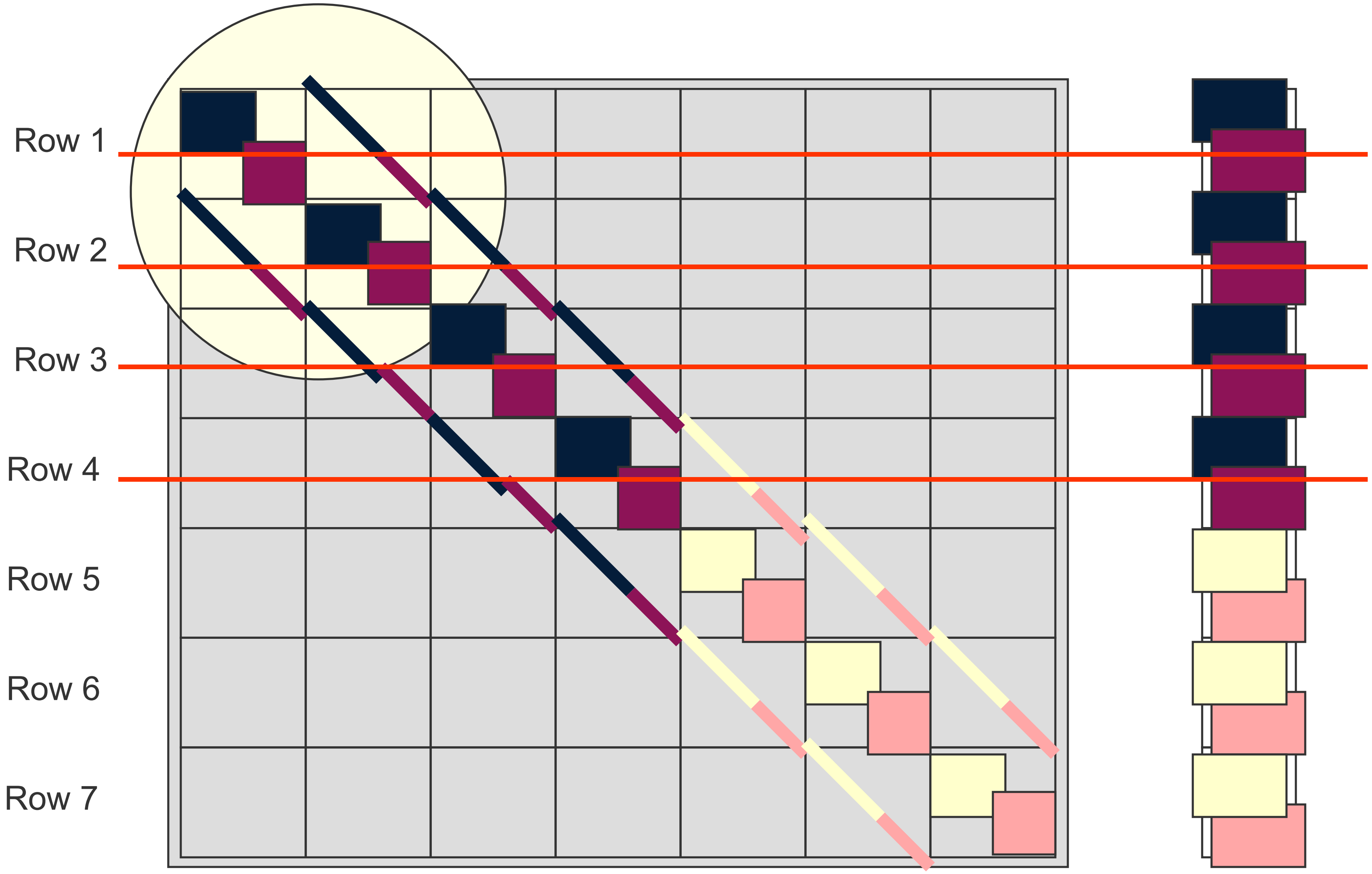
$$T = \begin{bmatrix} 4 & -1 & & & & & \\ -1 & 4 & -1 & & & & \\ & -1 & 4 & -1 & & & \\ & & -1 & 4 & -1 & & \\ & & & -1 & 4 & -1 & \\ & & & & -1 & 4 & -1 \\ & & & & & -1 & 4 \end{bmatrix}$$

$$I = identity \\ matrix$$

2D CG Calculations in Parallel

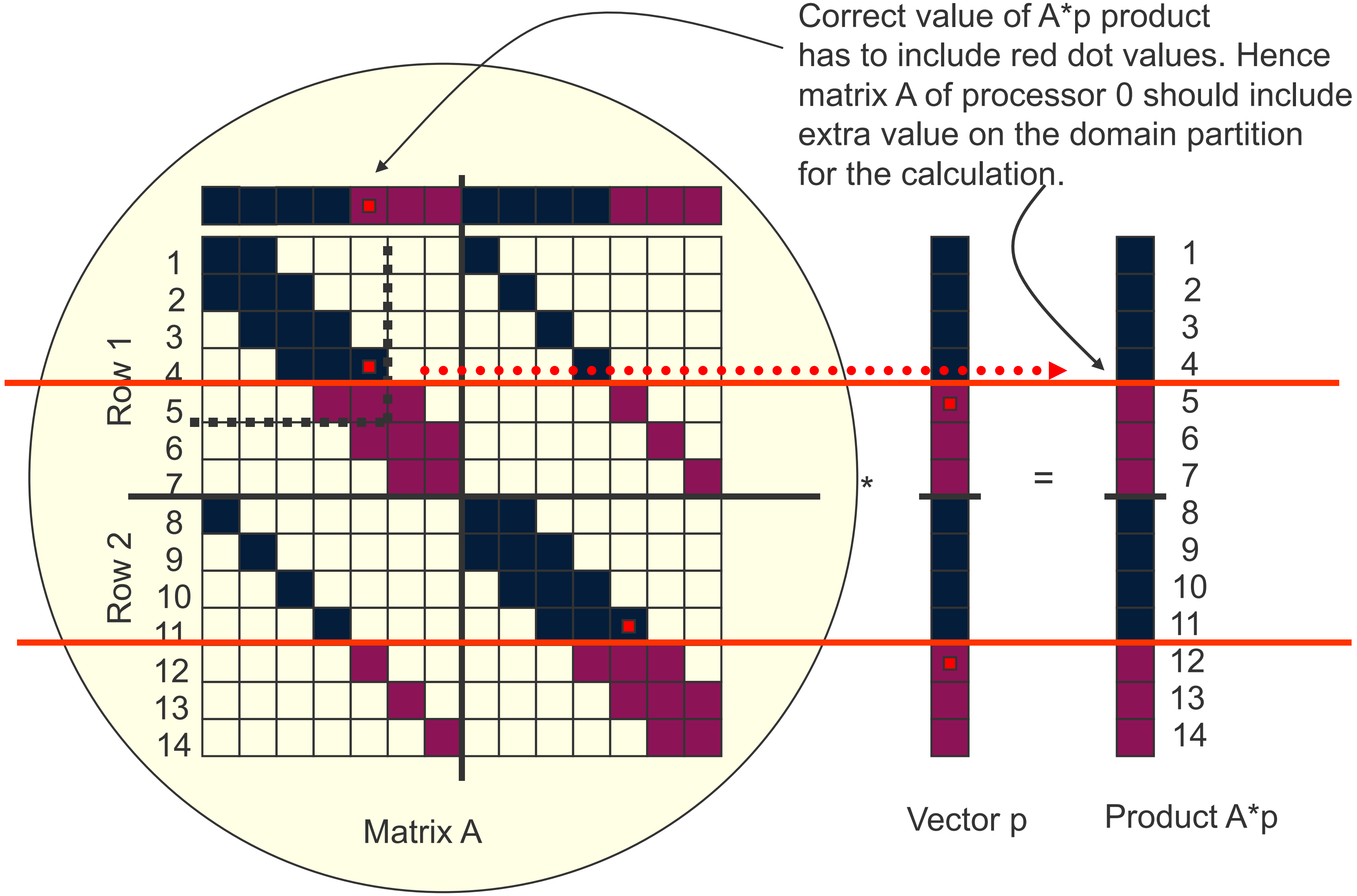


Global Matrix Data Distribution

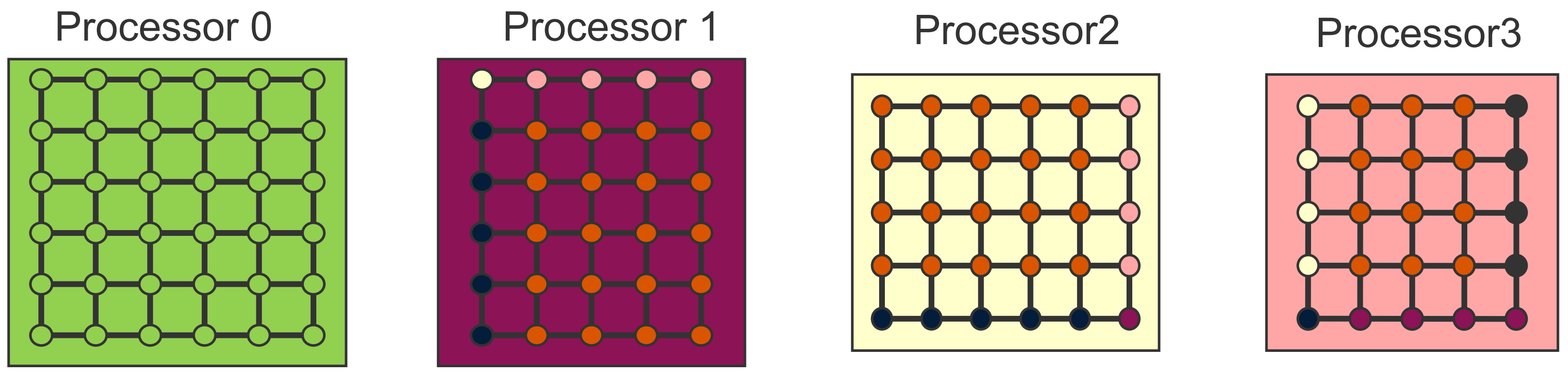


Matrix Vector Multiplication

Correct value of A*p product has to include red dot values. Hence matrix A of processor 0 should include extra value on the domain partition for the calculation.



CG Parallel Implementation for 2D Laplace Equation

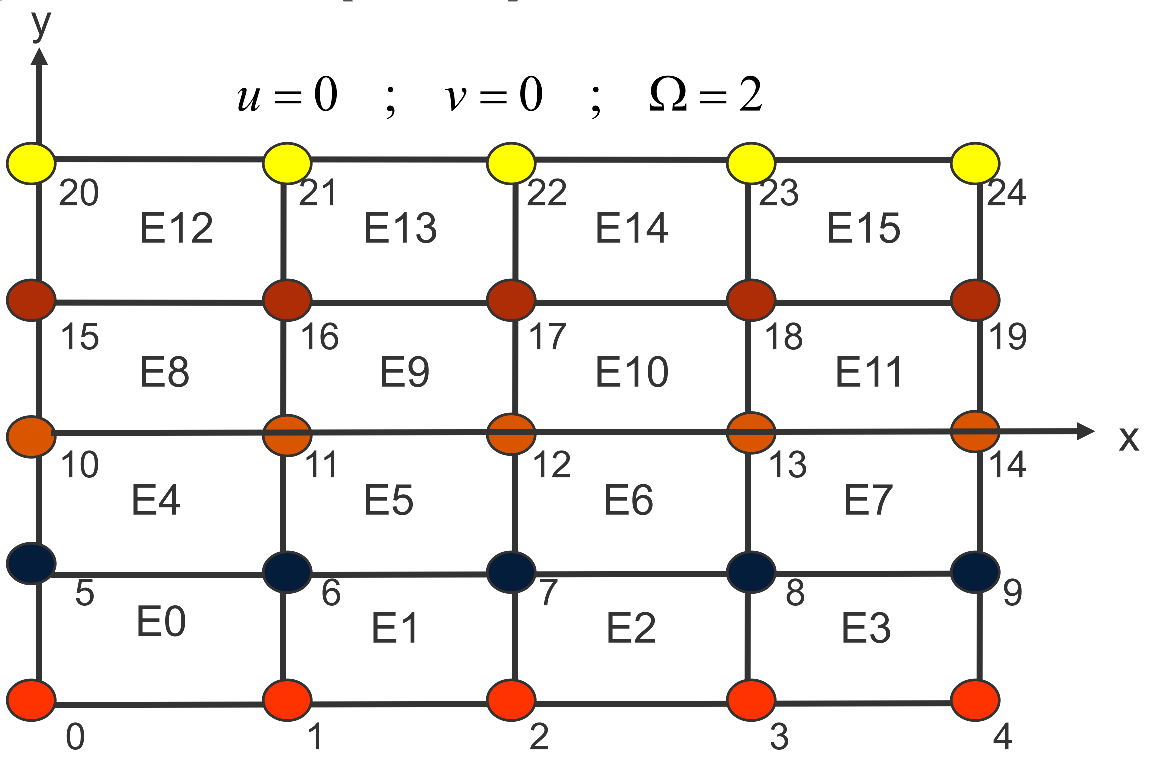


```
$k = 0 ; x_0 = 0 ; r_0 = b ; \phi_0 = r_0^T r_0 ; \varepsilon = big$
  $while(\phi > \varepsilon)$ $do$ ; $k = k + 1$
      $p_k = r_{k-1} + \phi_{k-1} p_{k-1} / \phi_{k-2}$ $or$ $p_1 = r_0$
      $w_k = Ap_k$    $w_k = Ap_k$    $w_k = Ap_k$    $w_k = Ap_k$
      $\beta_k = p_k^T w_k$    $\beta_k = p_k^T w_k$    $\beta_k = p_k^T w_k$    $\beta_k = p_k^T w_k$
      $\alpha_k = \phi_{k-1} / \beta_k ; x_k = x_{k-1} + \alpha_k p_k ; r_k = r_{k-1} - \alpha_k w_k$
      $\phi_k = r_k^T r_k$    $\phi_k = r_k^T r_k$    $\phi_k = r_k^T r_k$    $\phi_k = r_k^T r_k$
$end$ $while$ ; $x = x_k$
```

Example Grid (5x5)

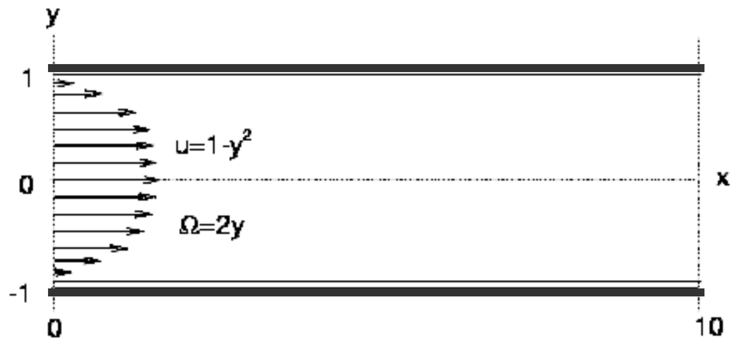
$$u = 0 \quad ; \quad v = 0 \quad ; \quad \Omega = 2$$

$$u = (1 - y^2)$$
$$v = 0$$
$$\Omega = 2y$$



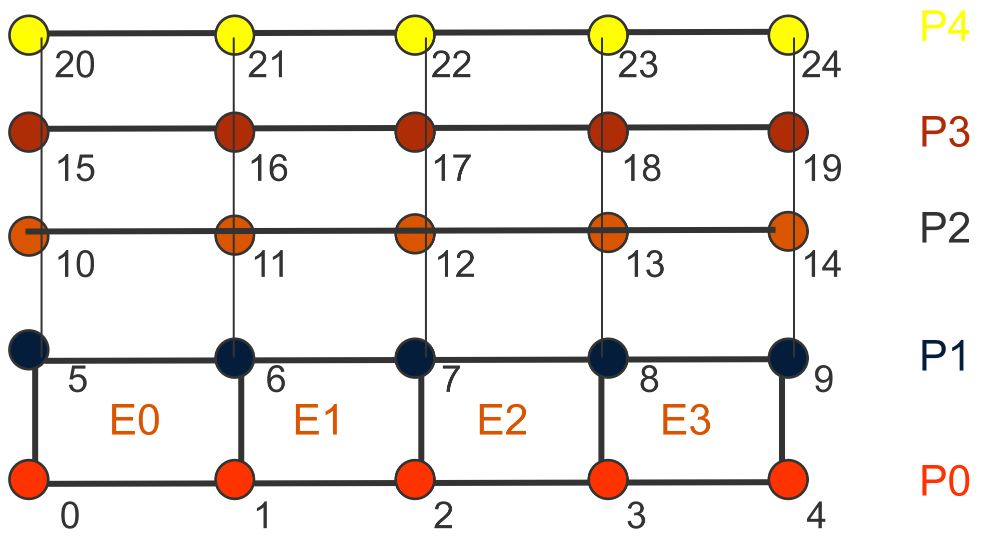
$$\frac{\partial u}{\partial x} = 0$$
$$\frac{\partial v}{\partial x} = 0$$
$$\frac{\partial \Omega}{\partial x} = 0$$

$$u = 0 \quad : \quad v = 0 \quad : \quad \Omega = -2$$



Parallel FE Assembly, Example, 5 x 5 Mesh : Process 0

Process 0
Process Grid : 0, 1, 2, 3, 4
Element : E0, E1, E2, E3
Compute Grid:
0,1,2,3,4,5,6,7,8,9

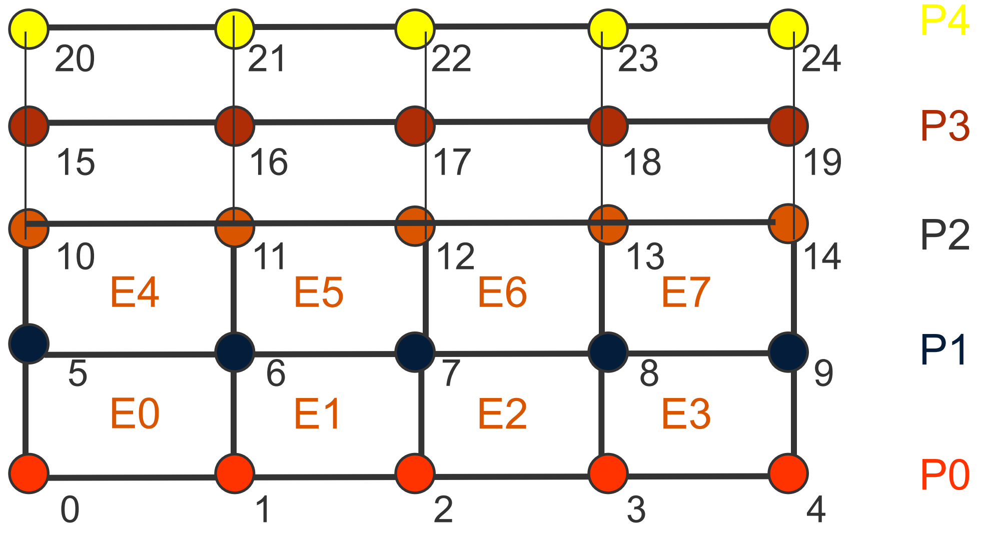


Process 0
Process Grid : 0, 1, 2, 3, 4
represents global matrix
row 0, 1, 2, 3, 4

	0	1	2	3	4	5	6	7	8	9	10
0	X	X				X	X				
1	X	X	X			X	X	X			
2		X	X	X			X	X	X		
3			X	X	X			X	X	X	
4				X	X				X	X	
5											
6											

Parallel FE Assembly, Example, 5 x 5 Mesh : Process 1

Process 1
Process Grid : 5, 6, 7, 8, 9
Element : E0, E1, E2, E3,
E4, E5, E6, E7
Compute Grid:
0,1,2,3,4,5,6,7,8,9,10
11,12,13,14



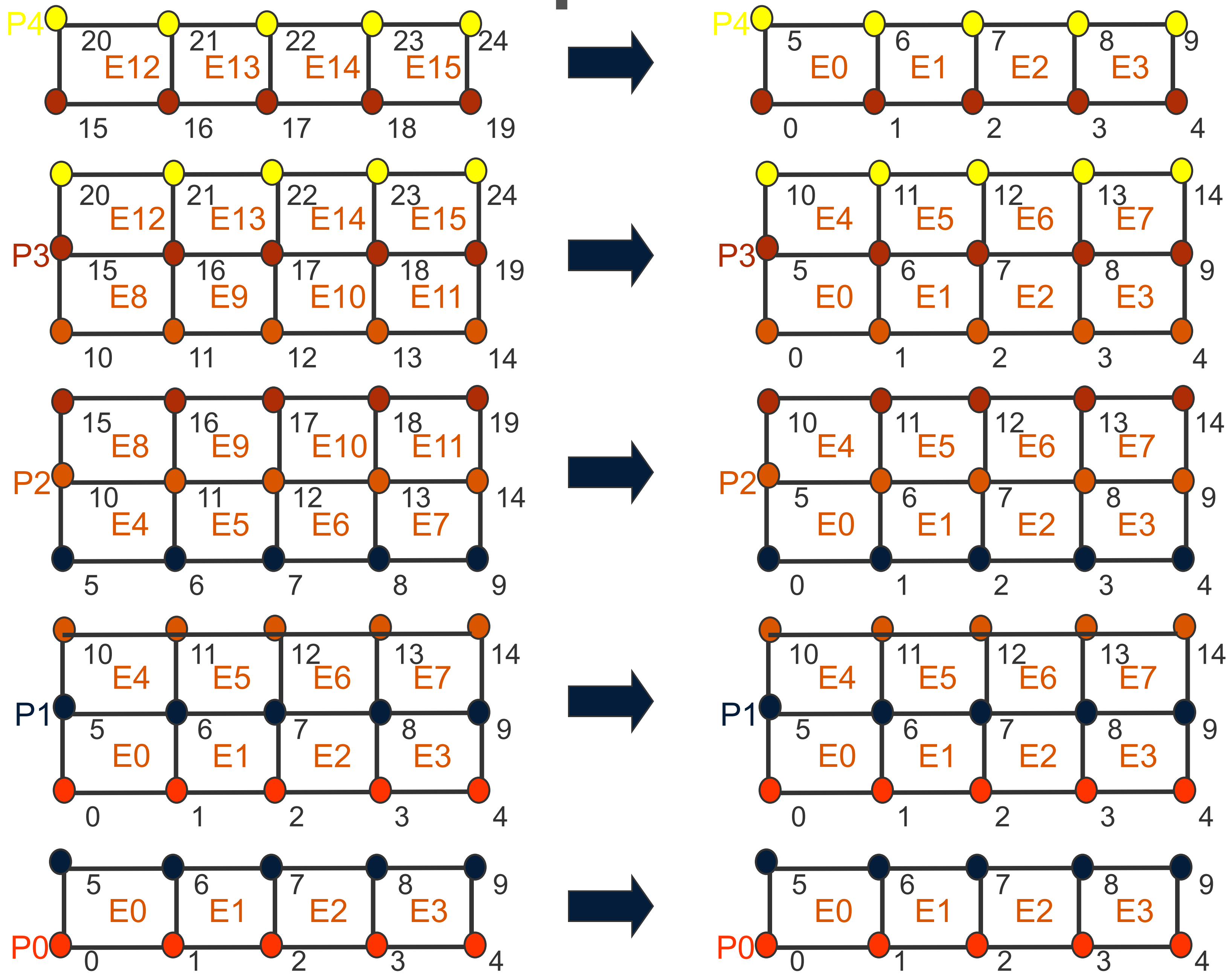
Process 1 , Process Grid : 5, 6, 7, 8, 9 represents global matrix row 5, 6, 7, 8, 9

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
4				X	X				X	X					
5	X	X				X	X				X	X			
6	X	X	X			X	X	X			X	X	X		
7		X	X	X			X	X	X			X	X	X	
8			X	X	X			X	X	X			X	X	X
9				X	X				X	X				X	X
10															

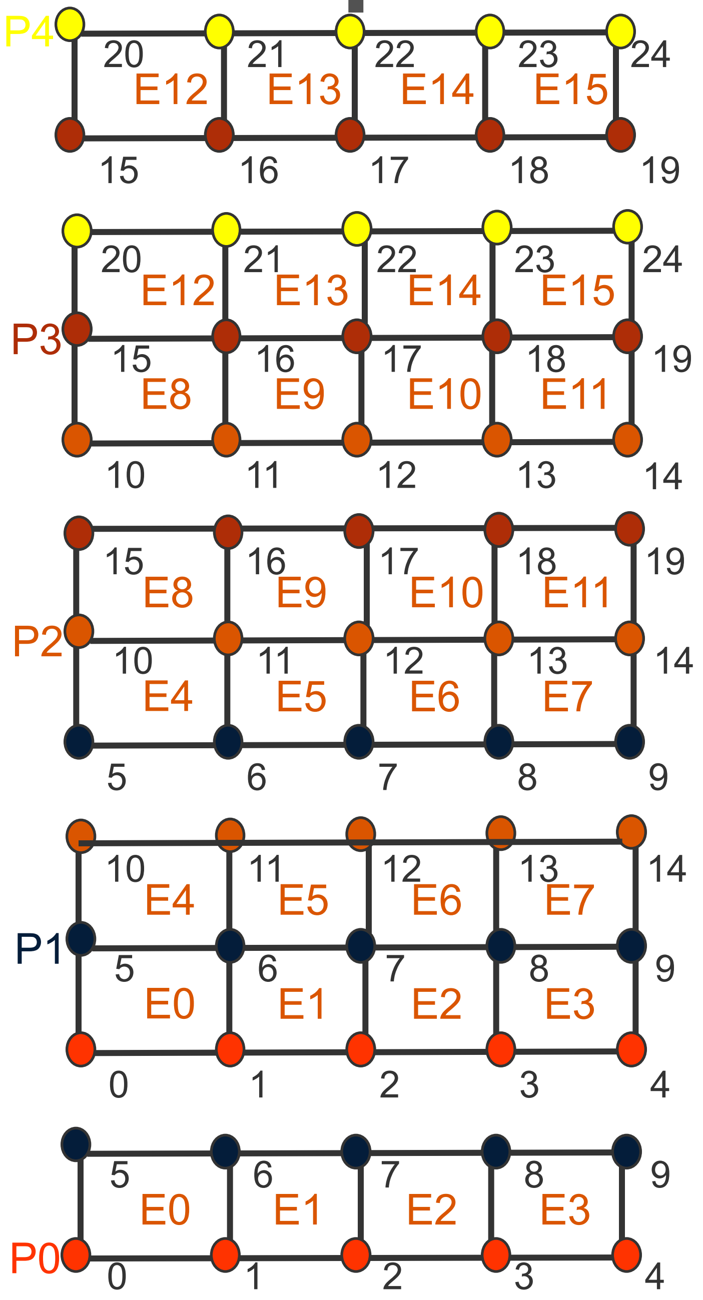
Distributed Global GWS FE Matrix Structure

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
	0	x	x				x	x																		
	1	x	x	x			x	x	x																	
Store x only in P0	2		x	x	x			x	x	x																
	3			x	x	x			x	x	x															
	4				x	x				x	x															
	5	x	x				x	x			x	x														
	6	x	x	x			x	x	x			x	x	x												
Store x only in P1	7		x	x	x			x	x	x			x	x	x											
	8			x	x	x			x	x	x			x	x	x										
	9				x	x				x	x				x	x										
	10						x	x				x	x				x	x								
	11						x	x	x			x	x	x			x	x	x							
Store x only in P2	12							x	x	x			x	x	x			x	x	x						
	13								x	x	x			x	x	x			x	x	x					
	14									x	x				x	x				x	x					
	15											x	x				x	x				x	x			
	16											x	x	x			x	x	x			x	x	x		
Store x only in P3	17												x	x	x			x	x	x			x	x	x	
	18													x	x	x			x	x	x			x	x	x
	19														x	x				x	x				x	x
	20																x	x				x	x			
	21																x	x	x			x	x	x		
Store x only in P4	22																	x	x	x			x	x	x	
	23																		x	x	x			x	x	x
	24																			x	x				x	x

Global to Local Representations



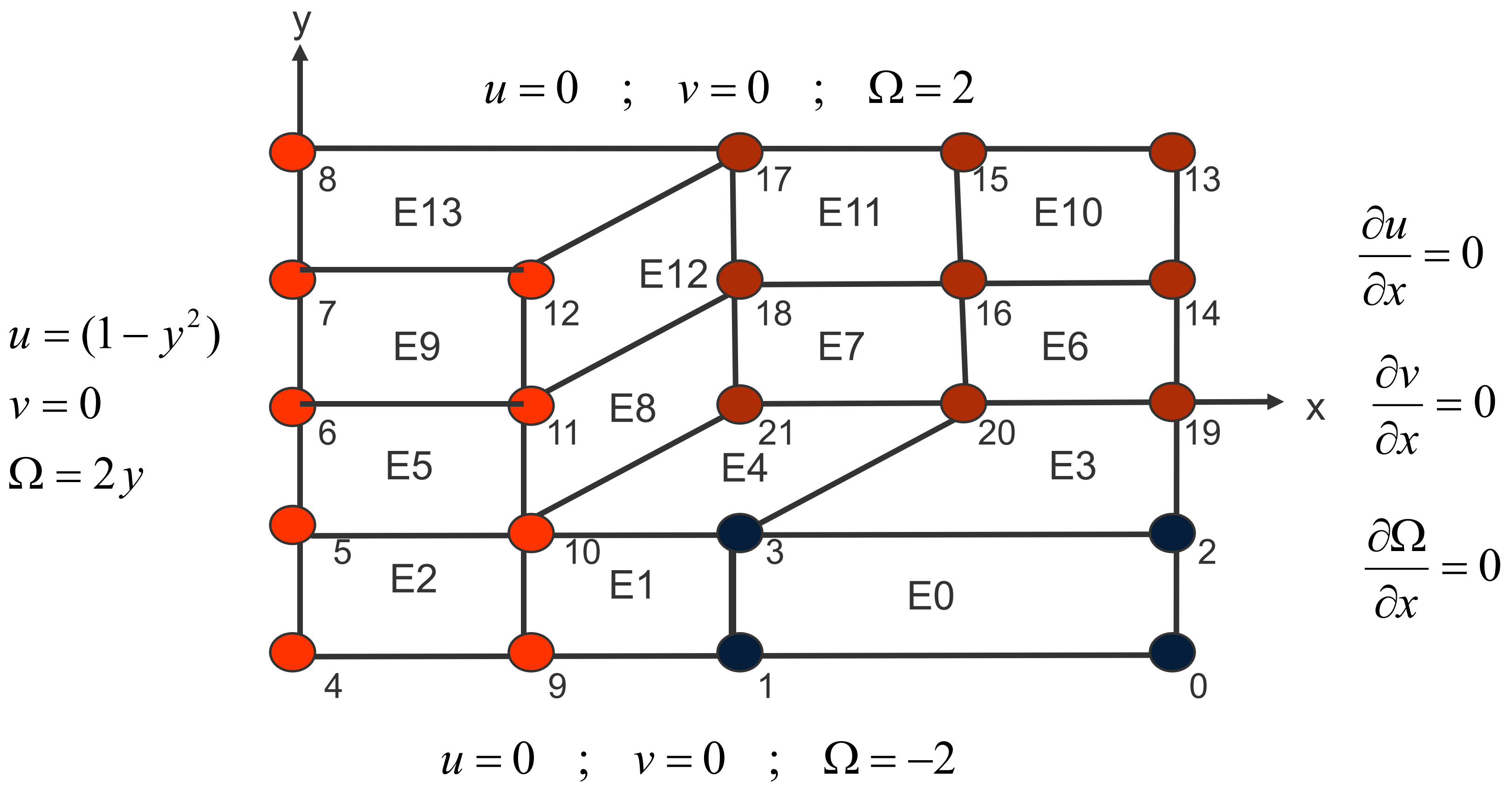
Aztec Representation (DMSR)



Process 2
nupdate = 5 ; iupdate = {10, 11, 12, 13, 14}
ibindx = {6, 11, 19, 27, 32,
5, 6, 11, 15, 16
5, 6, 7, 10, 12, 15, 16, 17
6, 7, 8, 11, 13, 16, 17, 18
7, 8, 9, 12, 14, 17, 18, 19
8, 9, 13, 18 ,19}

Process 0
nupdate = 5 ; iupdate = {0, 1, 2, 3, 4}
ibindx = {6, 9, 14, 19, 24, 27,
1, 5, 6,
0, 2, 5, 6, 7,
1, 3, 6, 7, 8
2, 4, 7, 8, 9
3, 8 9 }

Unstructured Grid



$u = 0 \quad ; \quad v = 0 \quad ; \quad \Omega = 2$

$u = (1 - y^2)$

$v = 0$

$\Omega = 2y$

$\frac{\partial u}{\partial x} = 0$

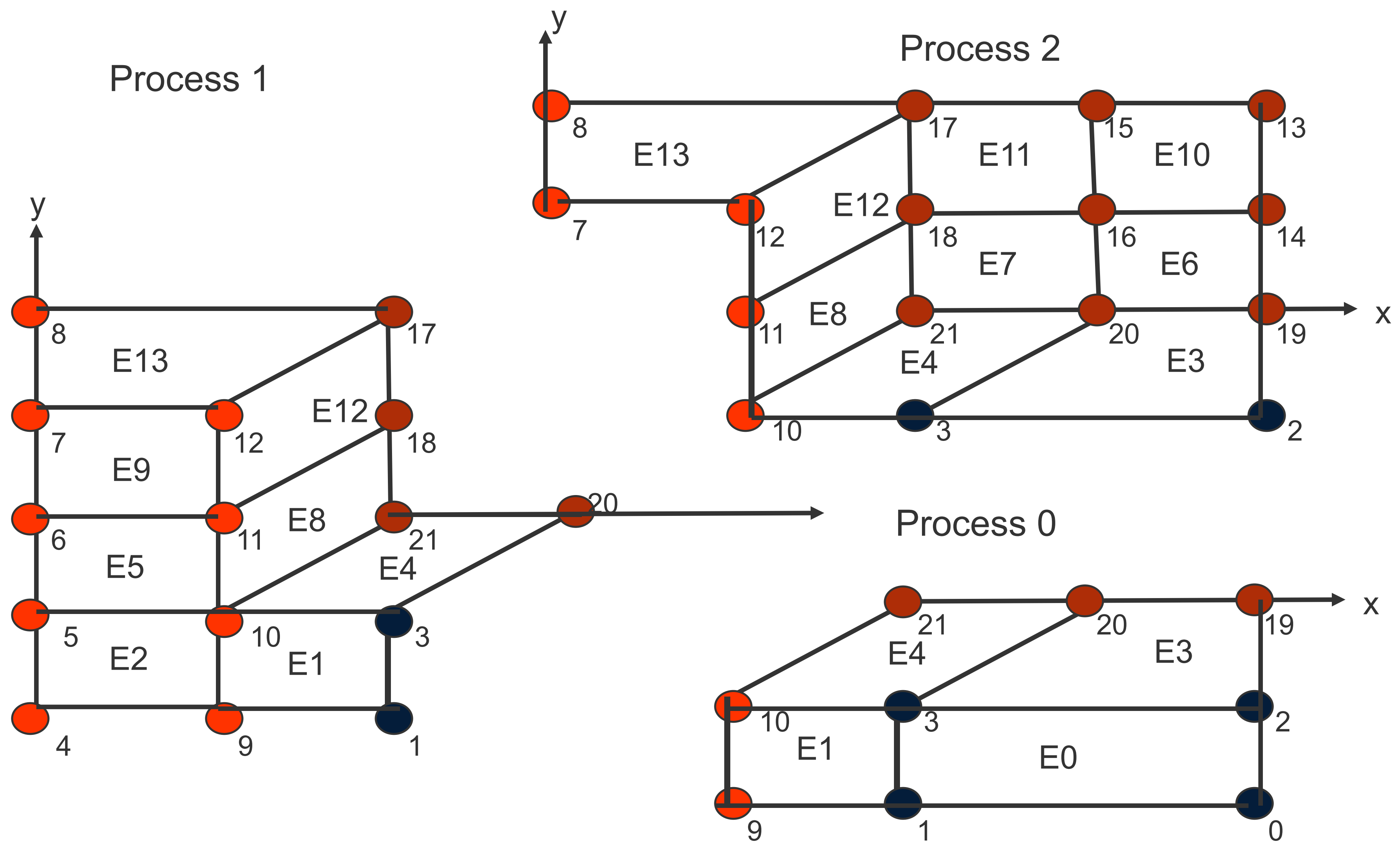
$\frac{\partial v}{\partial x} = 0$

$\frac{\partial \Omega}{\partial x} = 0$

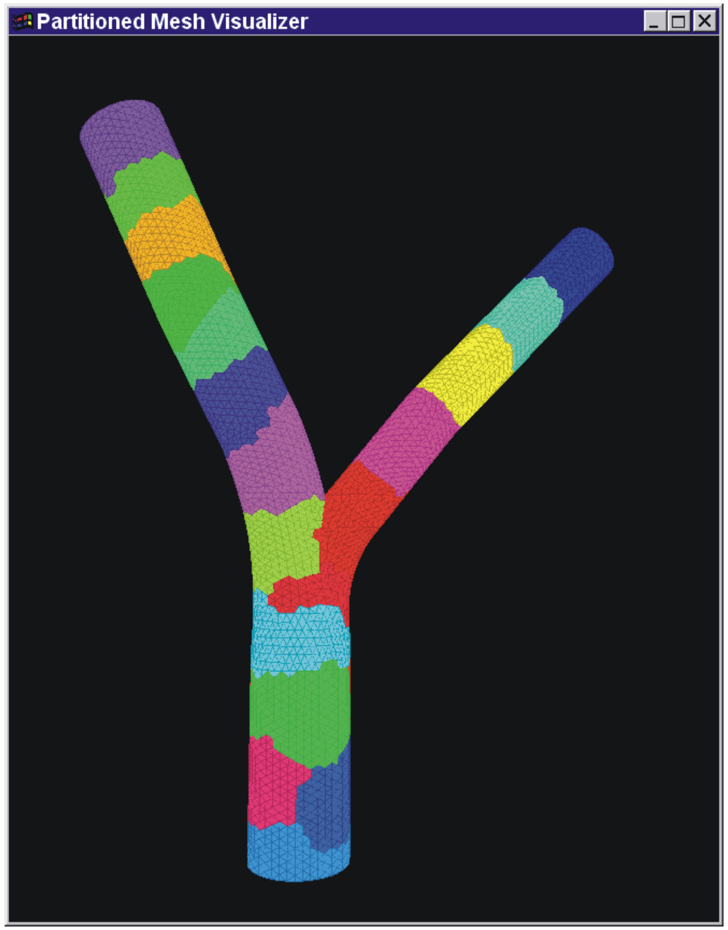
$u = 0 \quad ; \quad v = 0 \quad ; \quad \Omega = -2$

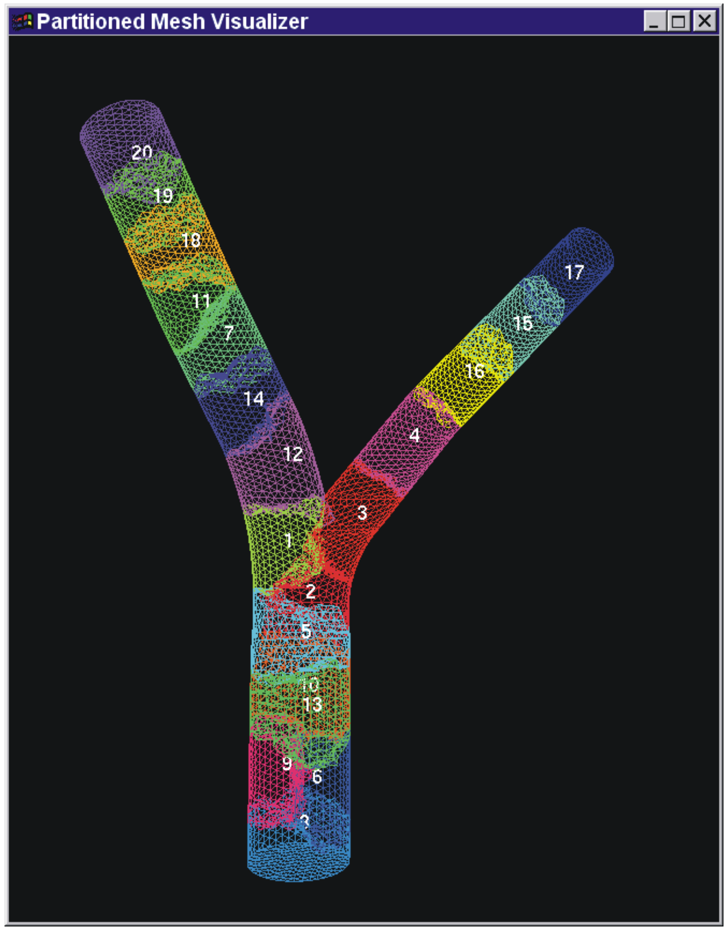
Parallel Finite Element Mesh

Non-Lexicographic Grid Partitioned for 3 Processors



Domain Decomposition with METIS*





*METIS: A Software Package for Partitionng Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version 4.0, G. Karypis and V. Kumar, University of Minnesota, Dept. of Computer Science, Minneapolis, MN, September 20, 1998.

Mesh: 22,797 nodes and 119,210 elements
Partitioned into 20 domains

ptw 3/5/01

The End

Quote: "I think there is a world market for maybe five computers"
Thomas Watson, chairman of IBM, 1943