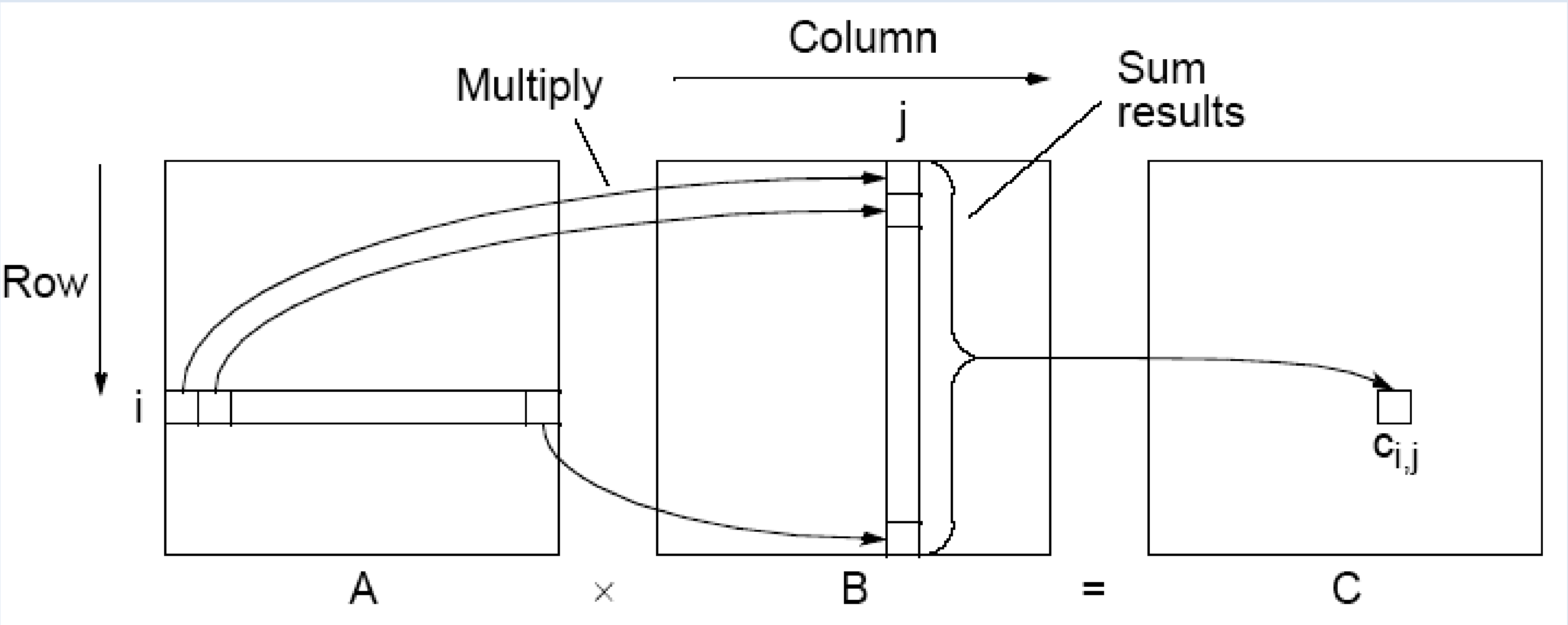
Numerical Algorithms

- Matrix multiplication
- Numerical solution of Linear System of Equations

Matrix multiplication, $C = A \times B$



Sequential Code

Assume throughout that the matrices are square (n x n matrices).
The sequential code to compute **A** x **B** :

```
for (i = 0; i < n; i++)
        for (j = 0; j < n; j++) {
                c[i][j] = 0;
                for (k = 0; k < n; k++)
                        c[i][j] = c[i][j] + a[i][k] * b[k][j];
}
```

Requires n^3 multiplications and n^3 additions

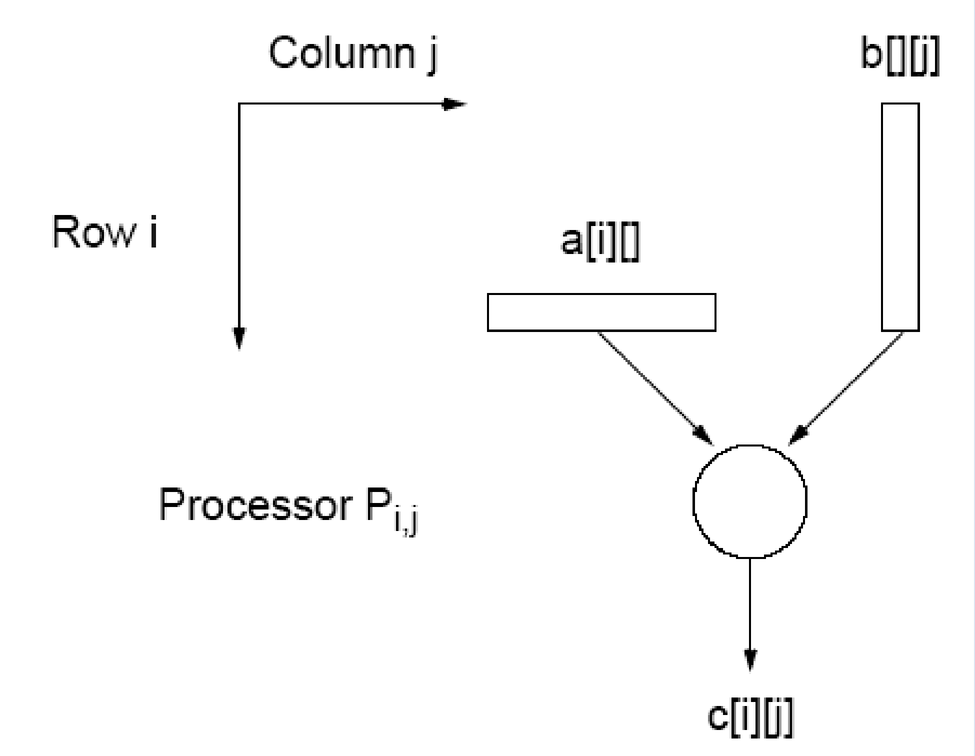
$T_{seq} = (n^3)$ (Very easy to parallelize!)

Direct Implementation ($P=n^2$)

- One PE to compute each element of **C** - n^2 processors would be needed.
- Each PE holds one row of elements of **A** and one column of elements of **B**.

$P = n^2$

$\Rightarrow T_{par} = O(n)$

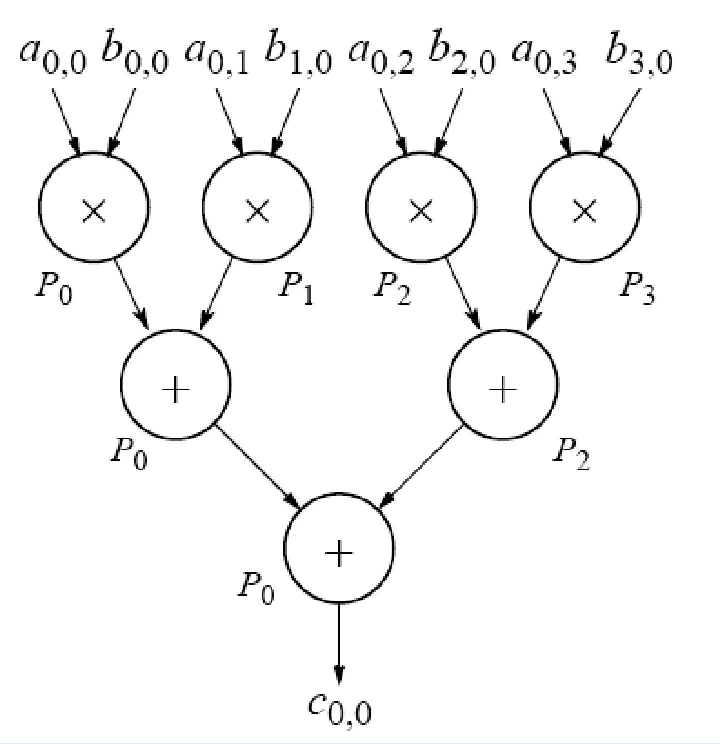


Performance Improvement ($P=n^3$)

- n processors collaborate in computing each element of **C** - n^3 processors are needed.
- Each PE holds one element of **A** and one element of **B**.

$P = n^3$

 $T_{par} = O(\log n)$



Parallel Matrix Multiplication - Summary

- $P = n$ $\qquad$ $T_{par} = O(n^2)$

  Each instance of inner loop is independent and can be done by a separate processor.

  Cost optimal since $O(n^3) = n * O(n^2)$

- $P = n^2$ $\qquad$ $T_{par} = O(n)$

  One element of C (c_{ij}) is assigned to each processor.

  Cost optimal since $O(n^3) = n^2 \times O(n)$

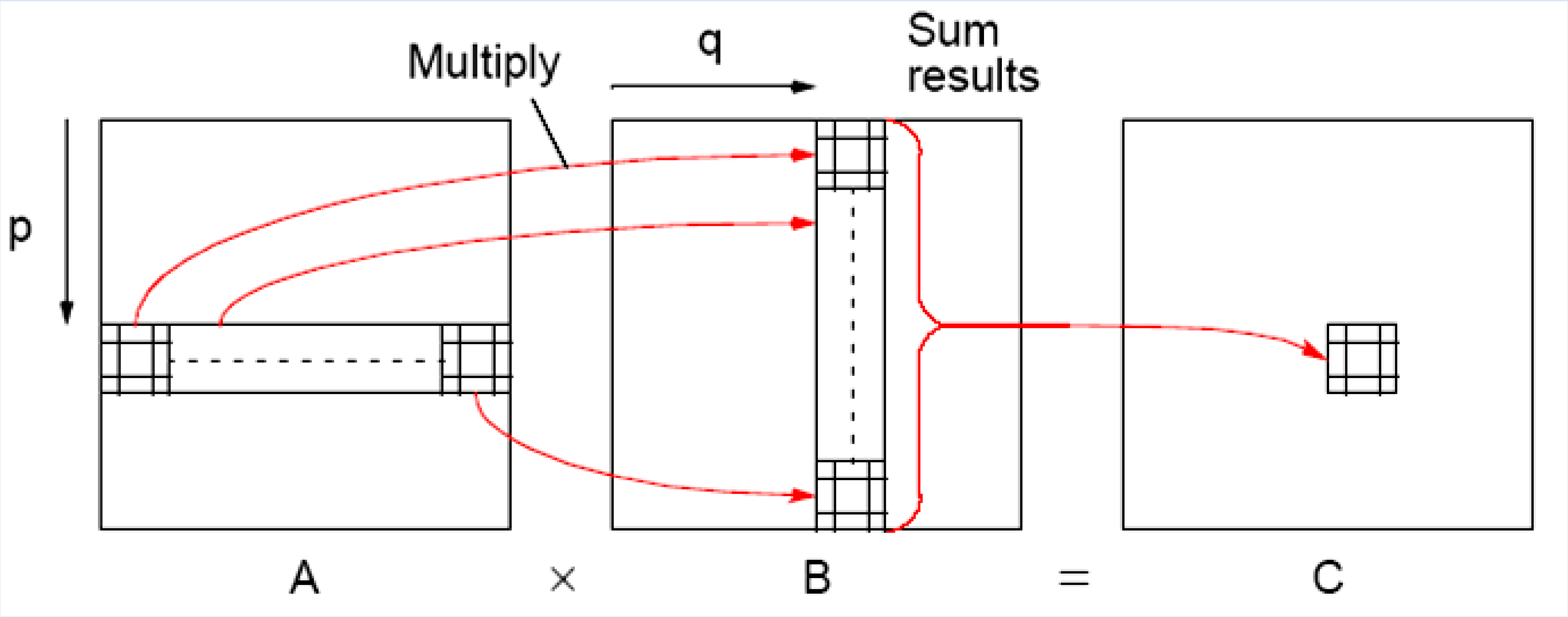
- $P = n^3$ $\qquad$ $T_{par} = O(\log n)$

  n processors compute one element of C (c_{ij}) in parallel ($O(\log n)$)

  Not cost optimal since $O(n^3) < n^3 * O(\log n)$

O(*log *n) lower bound for parallel matrix multiplication.

Block Matrix Multiplication



Submatrix multiplication

(a) Matrices

$$\left[\begin{array}{cc|cc} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ \hline a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{array}\right] \times \left[\begin{array}{cc|cc} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ \hline b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{array}\right]$$

(b) Multiplying $A_{0,0} \times B_{0,0}$ to obtain $C_{0,0}$

$$\underset{A_{0,0}}{\begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix}} \times \underset{B_{0,0}}{\begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{bmatrix}} + \underset{A_{0,1}}{\begin{bmatrix} a_{0,2} & a_{0,3} \\ a_{1,2} & a_{1,3} \end{bmatrix}} \times \underset{B_{1,0}}{\begin{bmatrix} b_{2,0} & b_{2,1} \\ b_{3,0} & b_{3,1} \end{bmatrix}}$$

$$= \begin{bmatrix} a_{0,0}b_{0,0}+a_{0,1}b_{1,0} & a_{0,0}b_{0,1}+a_{0,1}b_{1,1} \\ a_{1,0}b_{0,0}+a_{1,1}b_{1,0} & a_{1,0}b_{0,1}+a_{1,1}b_{1,1} \end{bmatrix} + \begin{bmatrix} a_{0,2}b_{2,0}+a_{0,3}b_{3,0} & a_{0,2}b_{2,1}+a_{0,3}b_{3,1} \\ a_{1,2}b_{2,0}+a_{1,3}b_{3,0} & a_{1,2}b_{2,1}+a_{1,3}b_{3,1} \end{bmatrix}$$

$$= \begin{bmatrix} a_{0,0}b_{0,0}+a_{0,1}b_{1,0}+a_{0,2}b_{2,0}+a_{0,3}b_{3,0} & a_{0,0}b_{0,1}+a_{0,1}b_{1,1}+a_{0,2}b_{2,1}+a_{0,3}b_{3,1} \\ a_{1,0}b_{0,0}+a_{1,1}b_{1,0}+a_{1,2}b_{2,0}+a_{1,3}b_{3,0} & a_{1,0}b_{0,1}+a_{1,1}b_{1,1}+a_{1,2}b_{2,1}+a_{1,3}b_{3,1} \end{bmatrix}$$

$$= C_{0,0}$$

Mesh Implementations

- Cannon's algorithm
- Systolic array

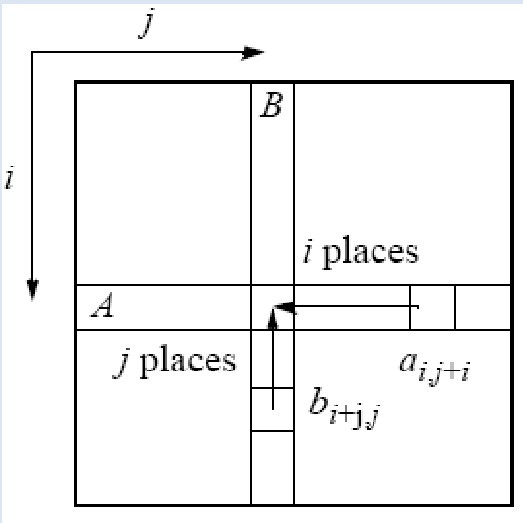
- All involve using processors arranged into a **mesh (**or **torus)** and shifting elements of the arrays through the mesh.
- Partial sums are accumulated at each processor.

Elements Need to Be Aligned

Each triangle represents a matrix element (or a block)

Only same-color triangles should be multiplied

Alignment of elements of *A* and *B*



$A_{0,0}$ $B_{0,0}$	$A_{0,1}$ $B_{0,1}$	$A_{0,2}$ $B_{0,2}$	$A_{0,3}$ $B_{0,3}$
$A_{1,0}$ $B_{1,0}$	$A_{1,1}$ $B_{1,1}$	$A_{1,2}$ $B_{1,2}$	$A_{1,3}$ $B_{1,3}$
$A_{2,0}$ $B_{2,0}$	$A_{2,1}$ $B_{2,1}$	$A_{2,2}$ $B_{2,2}$	$A_{2,3}$ $B_{2,3}$
$A_{3,0}$ $B_{3,0}$	$A_{3,1}$ $B_{3,1}$	$A_{3,2}$ $B_{3,2}$	$A_{3,3}$ $B_{3,3}$

Before

$A_{0,0}$ $B_{0,0}$	$A_{0,1}$ $B_{1,1}$	$A_{0,2}$ $B_{2,2}$	$A_{0,3}$ $B_{3,3}$
$A_{1,1}$ $B_{1,0}$	$A_{1,2}$ $B_{2,1}$	$A_{1,3}$ $B_{3,2}$	$A_{1,0}$ $B_{0,3}$
$A_{2,2}$ $B_{2,0}$	$A_{2,3}$ $B_{3,1}$	$A_{2,0}$ $B_{0,2}$	$A_{2,1}$ $B_{1,3}$
$A_{3,3}$ $B_{3,0}$	$A_{3,0}$ $B_{0,1}$	$A_{3,1}$ $B_{1,2}$	$A_{3,2}$ $B_{2,3}$

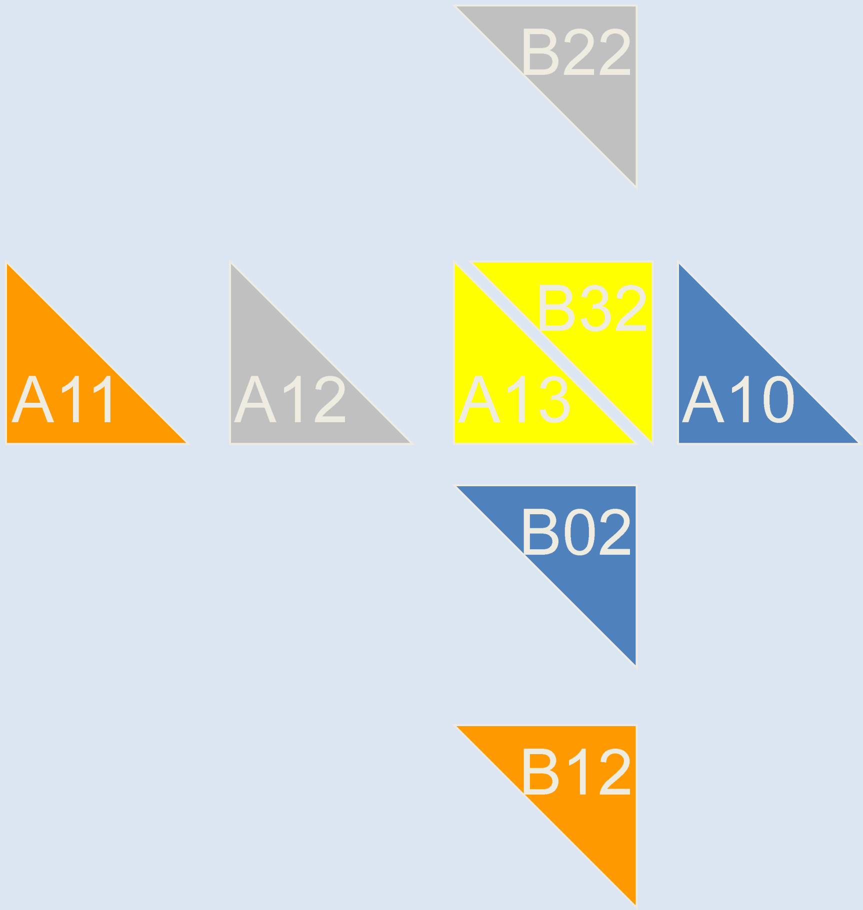
After

Alignment

A_{i*} (i^{th} row) cycles left i positions

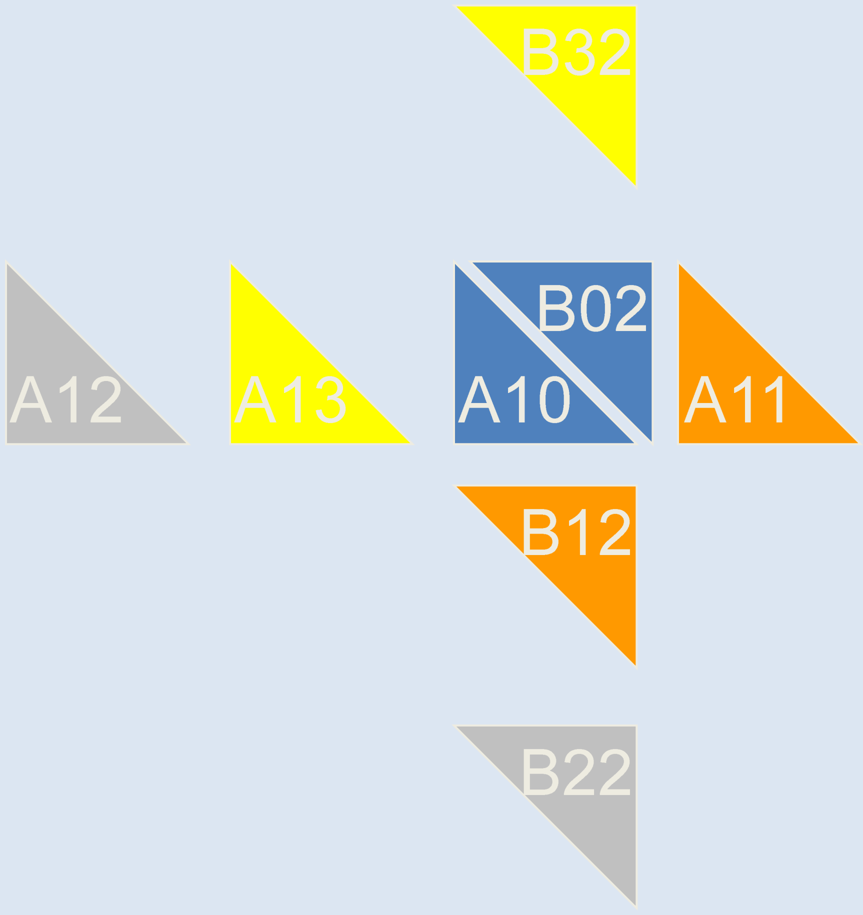
B_{*j} (j^{th} column) cycles up j positions

Shift, multiply, add
Consider Process $P_{1,2}$



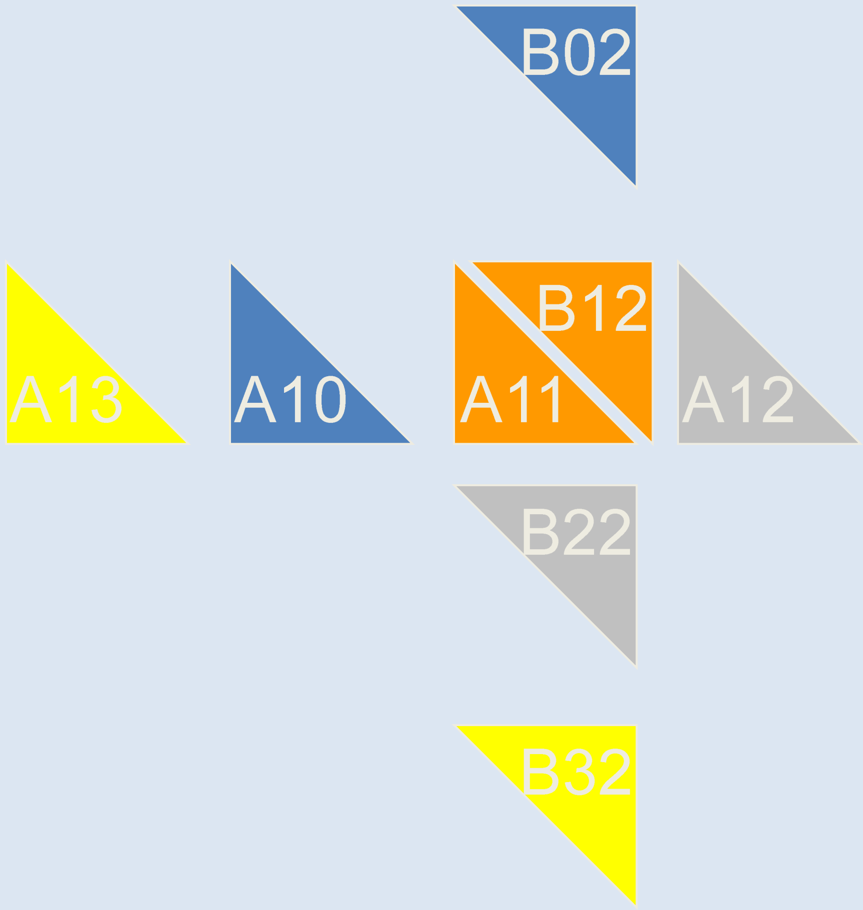
Step 1

Shift, multiply, add
Consider Process $P_{1,2}$



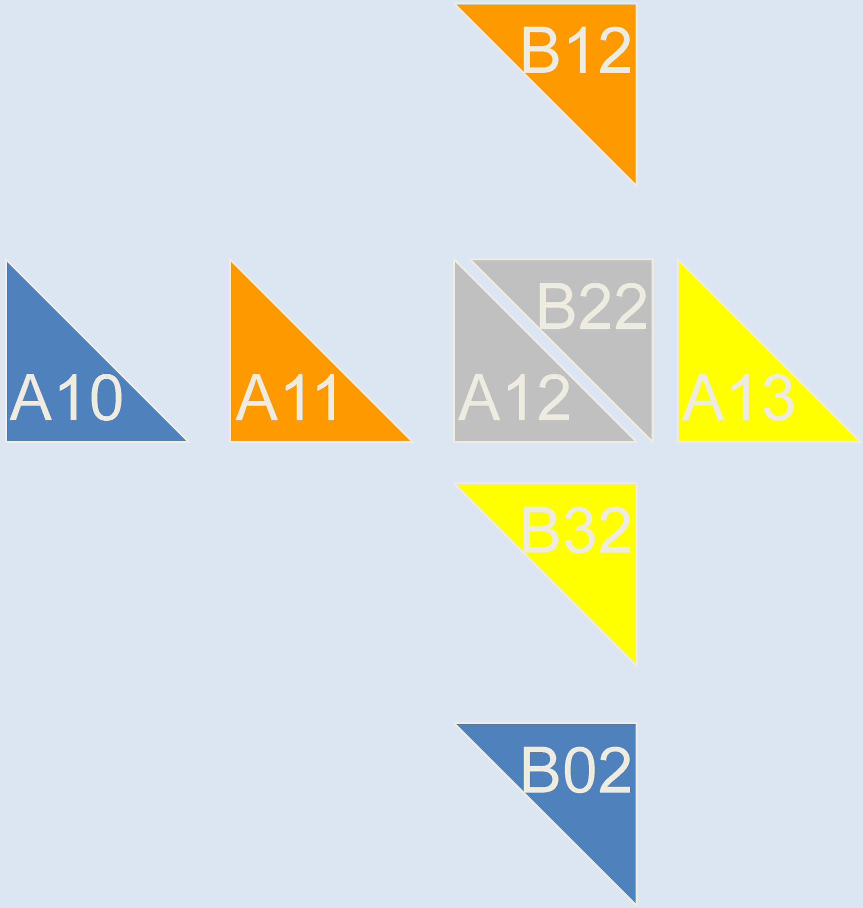
Step 2

Shift, multiply, add
Consider Process $P_{1,2}$



Step 3

Shift, multiply, add
Consider Process $P_{1,2}$



Step 4

Parallel Cannon's Algorithm

Uses a **torus** to shift the A elements (or submatrices) *left* and the B elements (or submatrices) *up* in a wraparound fashion.

- Initially processor $P_{i,j}$ has elements $a_{i,j}$ and $b_{i,j}$ ($0 \leq i < n$, $0 \leq j < n$).
- Elements are moved from their initial position to an "aligned" position. The complete i^{th} row of A is shifted i places left and the complete j^{th} column of B is shifted j places upward. This has the effect of placing the element $a_{i,j+i}$ and the element $b_{i+j,j}$ in processor $P_{i,j}$. These elements are a pair of those required in the accumulation of $c_{i,j}$.
- Each processor, $P_{i,j}$, multiplies its elements and accumulates the result in $c_{i,j}$
- The i^{th} row of A is shifted one place right, and the j^{th} column of B is shifted one place upward. This has the effect of bringing together the adjacent elements of A and B, which will also be required in the accumulation.
- Each PE ($P_{i,j}$) multiplies the elements brought to it and adds the result to the accumulating sum.
- The last two steps are repeated until the final result is obtained (n-1 shifts)

Time Complexity

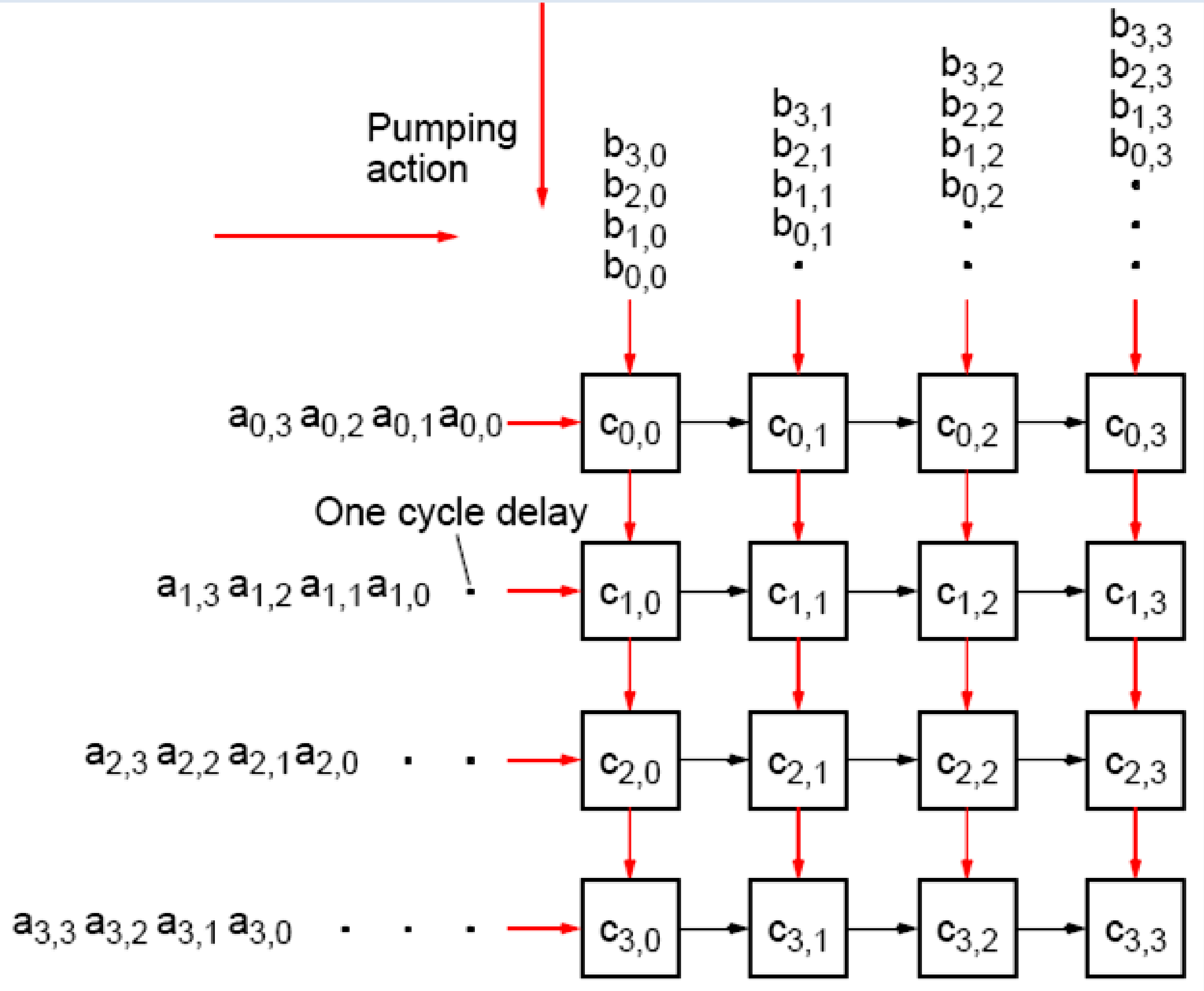
- $P = n^2$ A, B: $n \times n$ matrices with n^2 elements each

One element of C (c_{ij}) is assigned to each processor.
Alignment step takes O(n) steps.
Therefore,

$$T_{par} = O(n)$$

Cost optimal since $O(n^3) = n^2 * O(n)$
Also, **highly scalable**!

Systolic Array



Solving systems of linear equations: $Ax=b$

Dense matrices

Direct Methods:

Gaussian Elimination – seq. time complexity $O(n^3)$

LU-Decomposition – seq. time complexity $O(n^3)$

Sparse matrices (with good convergence properties)

Iterative Methods:

Jacobi iteration

Gauss-Seidel relaxation (not good for parallelization)

Red-Black ordering

Multigrid

Gauss Elimination

- Solve $Ax = b$
- Consists of two phases:
  - **Forward elimination**
  - **Back substitution**
- *Forward Elimination* reduces $Ax = b$ to an upper triangular system $Tx = b'$
- *Back substitution* can then solve $Tx = b'$ for x

$$\left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array}\right]$$

$$\Downarrow$$

$$\left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ 0 & a'_{22} & a'_{23} & b'_2 \\ 0 & 0 & a''_{33} & b''_3 \end{array}\right]$$

$$\Downarrow$$

$$x_3 = \frac{b''_3}{a''_{33}} \qquad x_2 = \frac{b'_2 - a'_{23}x_3}{a'_{22}}$$

$$x_1 = \frac{b_1 - a_{13}x_3 - a_{12}x_2}{a_{11}}$$

Forward Elimination

Back Substitution

Gauss Elimination

- Solve $Ax = b$

- Consists of two phases:
  - **–Forward elimination**
  - **–Back substitution**

- *Forward Elimination* reduces $Ax = b$ to an upper triangular system $Tx = b'$

- *Back substitution* can then solve $Tx = b'$ for x

$$\underbrace{\left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array}\right] \Downarrow \left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ 0 & a'_{22} & a'_{23} & b'_2 \\ 0 & 0 & a''_{33} & b''_3 \end{array}\right]}_{\text{Forward Elimination}}$$

$$\Downarrow$$

$$\underbrace{x_3 = \frac{b''_3}{a''_{33}} \qquad x_2 = \frac{b'_2 - a'_{23}x_3}{a'_{22}} \qquad x_1 = \frac{b_1 - a_{13}x_3 - a_{12}x_2}{a_{11}}}_{\text{Back Substitution}}$$

Forward Elimination

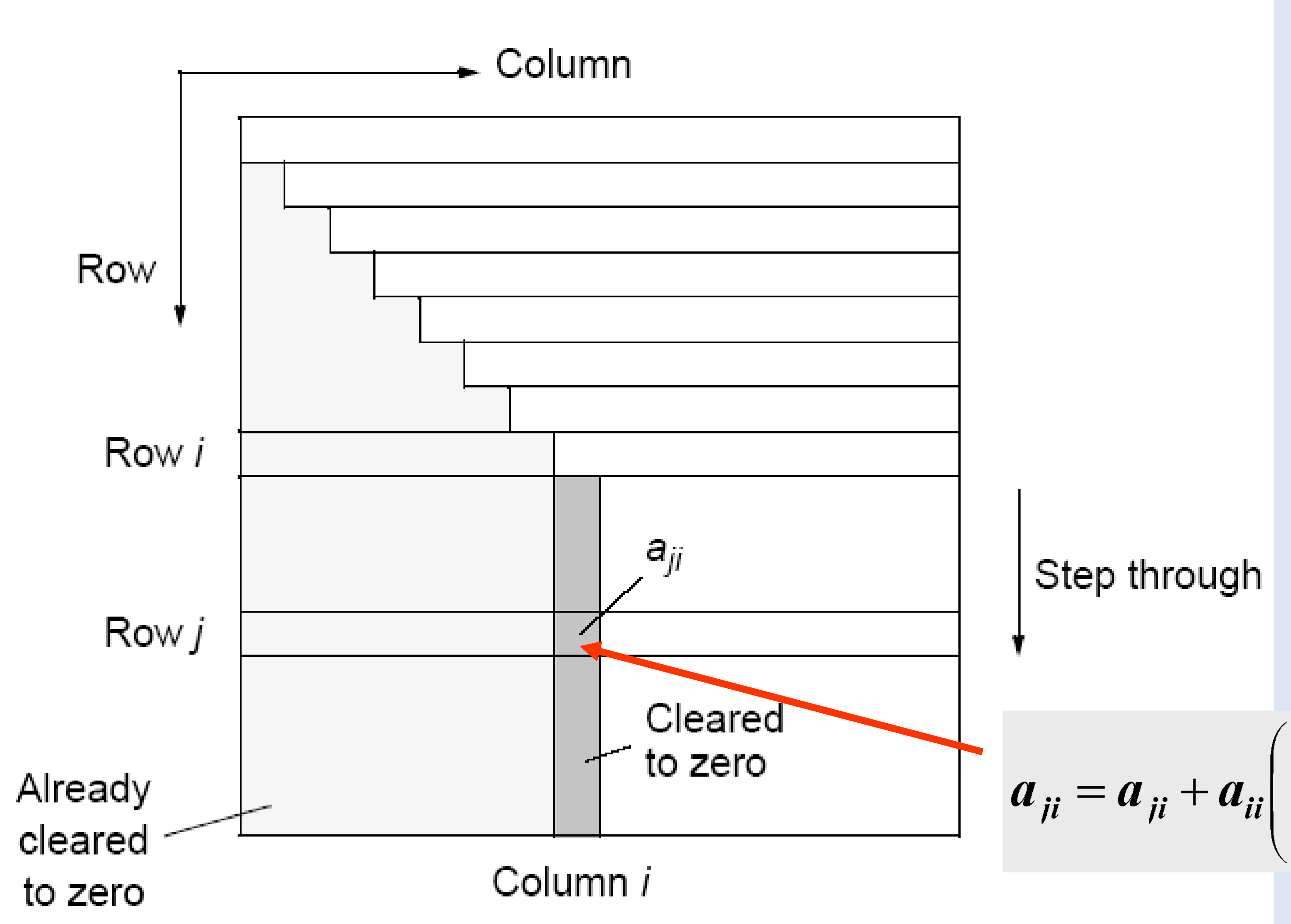
$$
\begin{array}{l}
 \\
-(3/1) \\
-(2/1)
\end{array}
\quad
\begin{array}{l}
x_1 - x_2 + x_3 = 6 \\
3x_1 + 4x_2 + 2x_3 = 9 \\
2x_1 + x_2 + x_3 = 7
\end{array}
\Rightarrow
\begin{array}{l}
 \\
 \\
-(3/7)
\end{array}
\quad
\begin{array}{l}
x_1 - x_2 + x_3 = 6 \\
0 \; +7x_2 - x_3 = -9 \\
0 \; + 3x_2 - x_3 = -5
\end{array}
$$

$$
\Rightarrow
\begin{array}{l}
x_1 - x_2 + x_3 = 6 \\
0 \quad 7x_2 - x_3 = -9 \\
0 \quad 0 \; -(4/7)x_3 = -(8/7)
\end{array}
$$

Solve using BACK SUBSTITUTION: $x_3 = 2$ $x_2 = -1$ $x_1 = 3$

0								
0	0							
0	0	0						
0	0	0	0					
0	0	0	0	0				
0	0	0	0	0	0			
0	0	0	0	0	0	0		
0	0	0	0	0	0	0	0	

Forward Elimination



$$a_{ji} = a_{ji} + a_{ii}\left(\frac{-a_{ji}}{a_{ii}}\right) = 0$$

Gaussian Elimination

M
U
L
T
I
P
L
I
E
R
S

-(2/4)

-(-4/4)

-(8/4)

$$
\begin{array}{llllll}
4x_0 & +6x_1 & +2x_2 & -2x_3 & = & 8 \\
2x_0 & & +5x_2 & -2x_3 & = & 4 \\
-4x_0 & -3x_1 & -5x_2 & +4x_3 & = & 1 \\
8x_0 & +18x_1 & -2x_2 & +3x_3 & = & 40
\end{array}
$$

Gaussian Elimination

M
U
L
T
I
P
L
I
E
R
S

	$4x_0$	$+6x_1$	$+2x_2$	$-2x_3$	$=$	8
		$-3x_1$	$+4x_2$	$-1x_3$	$=$	0
-(3/-3)		$+3x_1$	$-3x_2$	$+2x_3$	$=$	9
-(6/-3)		$+6x_1$	$-6x_2$	$+7x_3$	$=$	24

Gaussian Elimination

$$
\begin{array}{llllll}
4x_0 & +6x_1 & +2x_2 & -2x_3 & = & 8 \\
 & -3x_1 & +4x_2 & -1x_3 & = & 0 \\
 & & 1x_2 & +1x_3 & = & 9 \\
 & & 2x_2 & +5x_3 & = & 24
\end{array}
$$

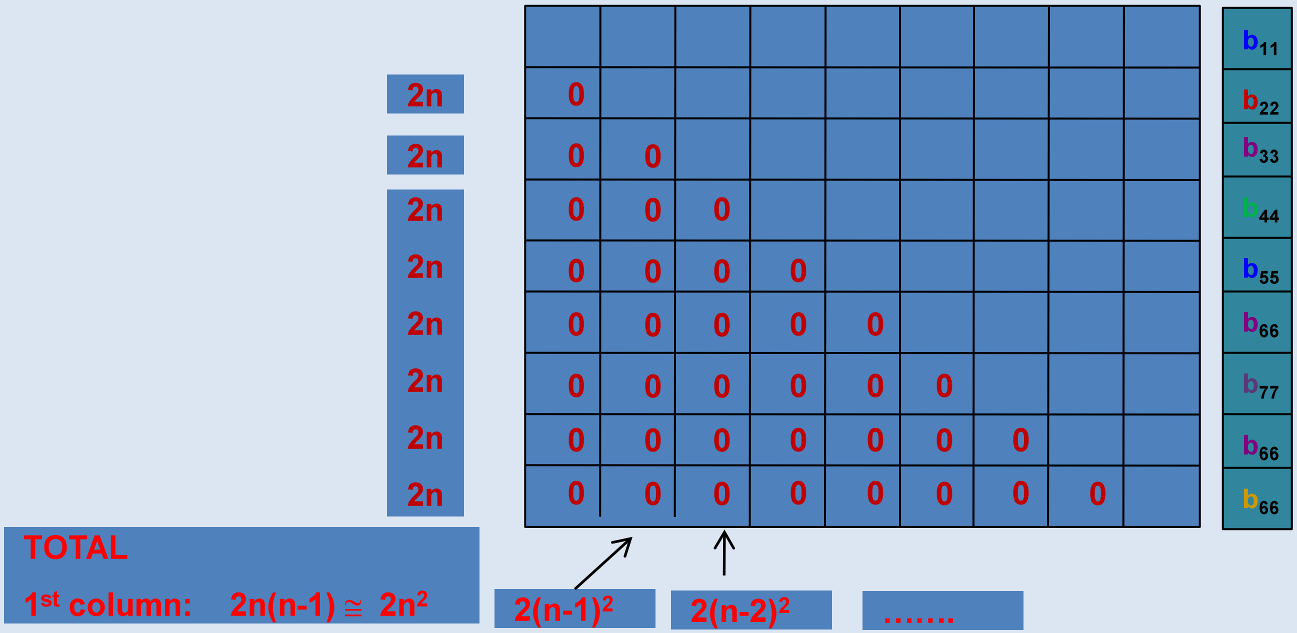
M
U
L
T
I
P
L
I
E
R

??

Gaussian Elimination

$$
\begin{array}{ccccccc}
4x_0 & +6x_1 & +2x_2 & -2x_3 & = & 8 \\
 & -3x_1 & +4x_2 & -1x_3 & = & 0 \\
 & & 1x_2 & +1x_3 & = & 9 \\
 & & & 3x_3 & = & 6
\end{array}
$$

Operation count in Forward Elimination



TOTAL # of Operations for FORWARD ELIMINATION :

$$2n^2 + 2(n-1)^2 + \ldots + 2*(2)^2 + 2*(1)^2 = 2\sum_{i=1}^{n} i^2$$

$$= 2\frac{n(n+1)(2n+1)}{6}$$

$$= O(n^3)$$

Back Substitution
(* Pseudocode *)

for $i \leftarrow n - 1$ down to 1 do

/* calculate x_i */

$x[i] \leftarrow b[i] / a[i, i]$

/* substitute in the equations above */

for $j \leftarrow 0$ to $i - 1$ **do**

$b[j] \leftarrow b[j] - x[i] \times a[j, i]$

endfor

endfor

Time Complexity? ***$O(n^2)$***

PARTIAL PIVOTING

If $a_{i,i}$ is zero or close to zero, we will not be able to compute the quantity $-a_{j,i} / a_{i,i}$

Procedure must be modified into so-called ***partial pivoting*** by **swap**ping the i^{th} row with the row below it that has the **largest** absolute element in the i^{th} column of any of the rows below the i^{th} row (if there is one).

(Reordering equations will not affect the system.)

SEQUENTIAL CODE

Without partial pivoting:

```
for (i = 0; i < n-1; i++)                  /* for each row, except last */
    for (j = i+1; j < n; j++) {            /* step thro subsequent rows */

        m = a[j][i]/a[i][i];               /* Compute multiplier  */

        for (k = i; k < n; k++)            /* last n-i-1 elements of row j */
          a[j][k] = a[j][k] - a[i][k] * m;

        b[j] = b[j] - b[i] * m;            /* modify right side */
    }
```

The time complexity: $T_{seq} = O(n^3)$

Computing the Determinant

Given an upper triangular system of equations

$$D = t_{00}\, t_{11} \ldots t_{n-1,n-1}$$

$$D = \begin{vmatrix} t_{00} & t_{01} & \cdots & t_{0,n-1} \\ 0 & t_{11} & \cdots & t_{1,n-1} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & t_{n-1,n-1} \end{vmatrix}$$

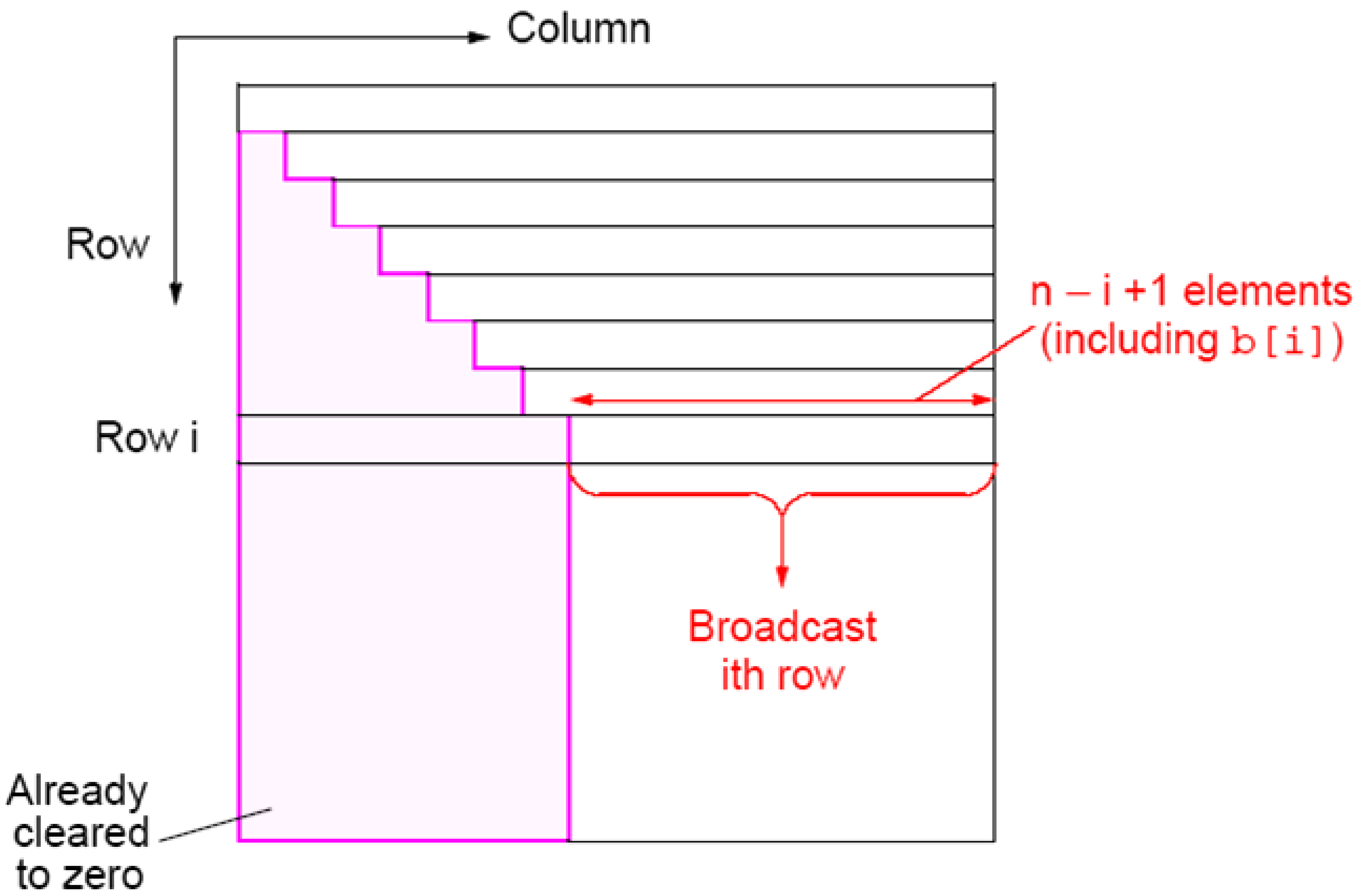
If ***pivoting*** is used then

$D = t_{00}\, t_{11} \ldots t_{n-1,n-1} (-1)^p$ *where p is the number of times the rows are pivoted*

Singular systems

- When two equations are identical, we would loose one degree of freedom
  n-1 equations for *n* unknowns ⇒ infinitely many solutions
- This is difficult to find out for large sets of equations.
  The fact that the **determinant** of a singular system is **zero** can be used and tested after the elimination stage.

PARALLEL IMPLEMENTATION



Time Complexity Analysis (P = n)

Communication

(n-1) broadcasts performed sequentially - i^{th} broadcast contains (n-i) elements.

Total Time: $T_{par} = O(n^2)$ (How ?)

Computation

After row i is broadcast, each processor P_j will compute its multiplier, and operate upon n-j+2 elements of its row. Ignoring the computation of the multiplier, there are (n-j+2) multiplications and (n-j+2) subtractions.

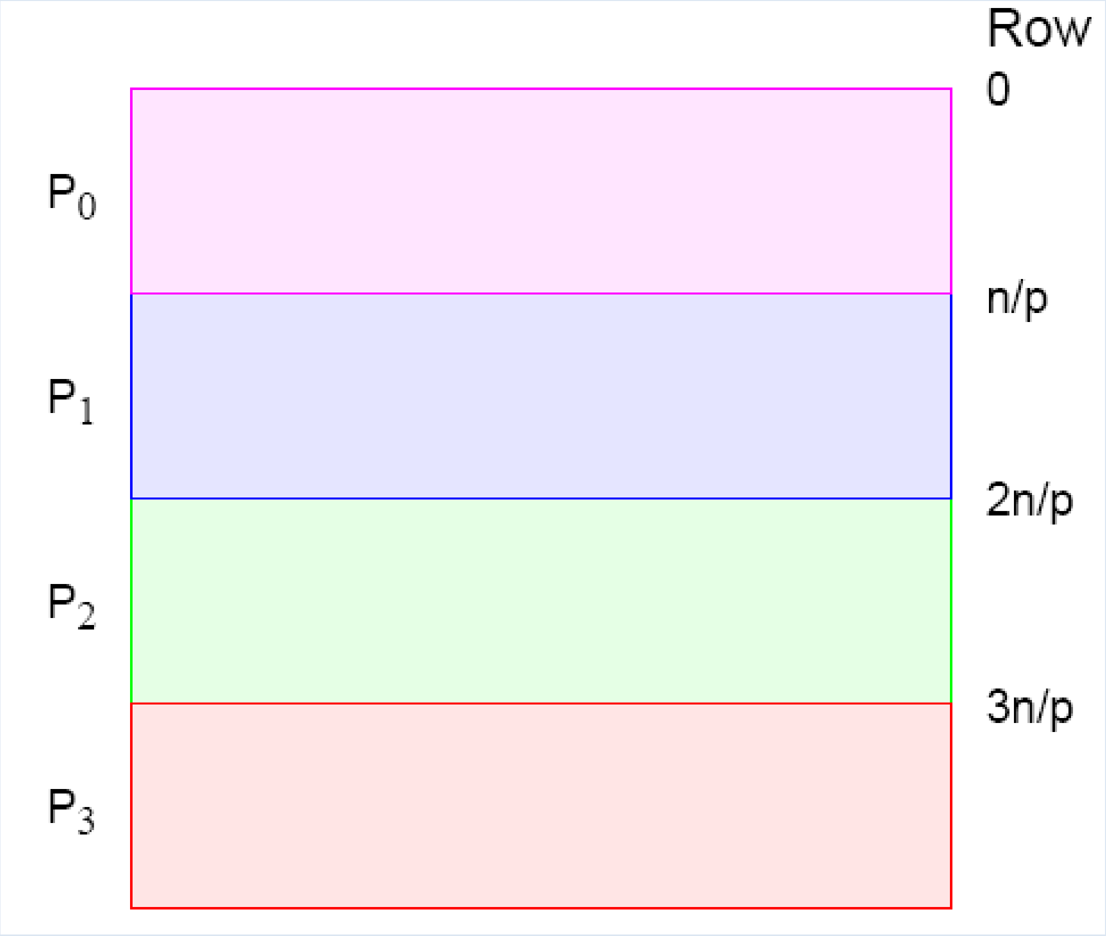
Total Time: $T_{par} = O(n^2)$

Therefore,

$$\boldsymbol{T_{par} = O(n^2)}$$

Efficiency will be relatively low because all the processors before the processor holding row i do not participate in the computation again.

Strip Partitioning

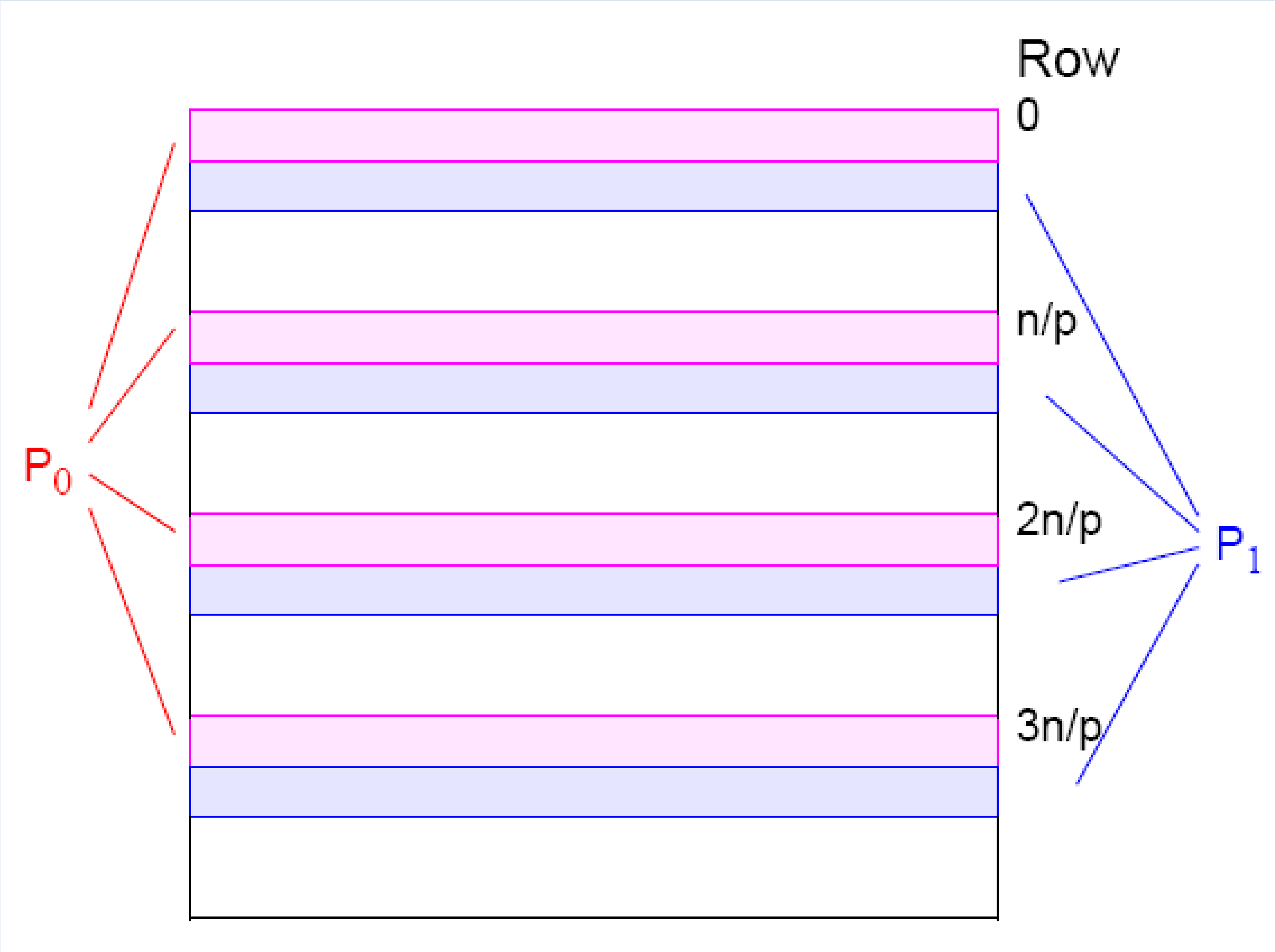


Poor processor allocation!

Processors do not participate in computation after their last row is processed.

Cyclic-Striped Partitioning

An alternative which equalizes the processor workload



Jacobi Iterative Method (Sequential)

Iterative methods provide an **alternative** to the *elimination methods*.

$$Ax = b \qquad A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \qquad D = \begin{bmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{bmatrix}$$

$$[D + (A - D)]x = b \quad \Rightarrow \quad Dx = b - (A - D)x \quad \Rightarrow \quad x = D^{-1}[b - (A - D)x]$$

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_{new} = \begin{bmatrix} 1/a_{11} & 0 & 0 \\ 0 & 1/a_{22} & 0 \\ 0 & 0 & 1/a_{33} \end{bmatrix} * \left(\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} - \begin{bmatrix} 0 & a_{12} & a_{13} \\ a_{21} & 0 & a_{23} \\ a_{31} & a_{32} & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_{old} \right)$$

$$x_1^k = \frac{b_1 - a_{12}x_2^{k-1} - a_{13}x_3^{k-1}}{a_{11}} \qquad x_2^k = \frac{b_2 - a_{21}x_1^{k-1} - a_{23}x_3^{k-1}}{a_{22}} \qquad x_3^k = \frac{b_3 - a_{31}x_1^{k-1} - a_{32}x_2^{k-1}}{a_{33}}$$

Choose an initial guess (i.e. all zeros) and Iterate until the equality is satisfied.
No guarantee for convergence! Each iteration takes O(n^2) time!

Gauss-Seidel Method (Sequential)

- The *Gauss-Seidel* method is a commonly used *iterative method*.

- It is same as **Jacobi technique** except with one important difference:
  A newly computed x value (say x_k) is substituted in the subsequent equations (equations $k+1$, $k+2$, …, n) **in the same iteration**.

Example: Consider the *3x3* system below:

$$x_1^{new} = \frac{b_1 - a_{12}x_2^{old} - a_{13}x_3^{old}}{a_{11}}$$

$$x_2^{new} = \frac{b_2 - a_{21}x_1^{new} - a_{23}x_3^{old}}{a_{22}}$$

$$x_3^{new} = \frac{b_3 - a_{31}x_1^{new} - a_{32}x_2^{new}}{a_{33}}$$

$$\{X\}_{old} \leftarrow \{X\}_{new}$$

- First, choose initial guesses for the x's.
- A simple way to obtain initial guesses is to assume that they are all **zero**.
- Compute **new** $\mathbf{x_1}$ using the previous iteration values.
- **New** $\boldsymbol{x_1}$ is substituted in the equations to calculate x_2 and x_3
- The process is repeated for $\mathbf{x_2}$, $\mathbf{x_3}$, …

Convergence Criterion for Gauss-Seidel Method

- Iterations are repeated until the convergence criterion is satisfied:

$$\left|\varepsilon_{a,i}\right| = \left|\frac{x_i^j - x_i^{j-1}}{x_i^j}\right| 100\% \ \surd \varepsilon_s$$

For all *i*, where *j* and *j-1* are the *current* and *previous* iterations.

- As any other iterative method, the **Gauss-Seidel** method has problems:
  - It may not converge or it converges very slowly.

- If the coefficient matrix A is **Diagonally Dominant** Gauss-Seidel is guaranteed to converge.

Diagonally Dominant □

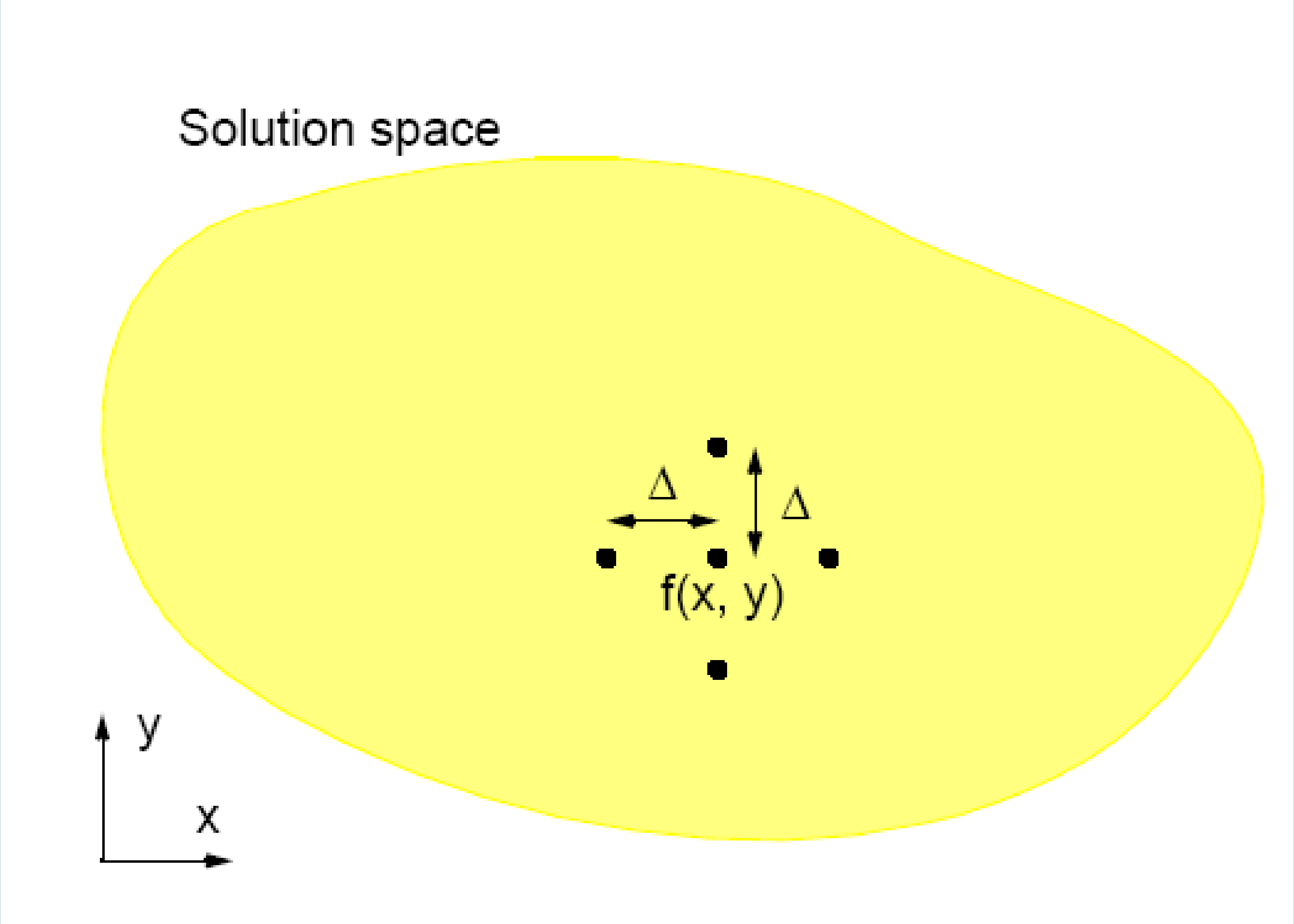
For each equation i :

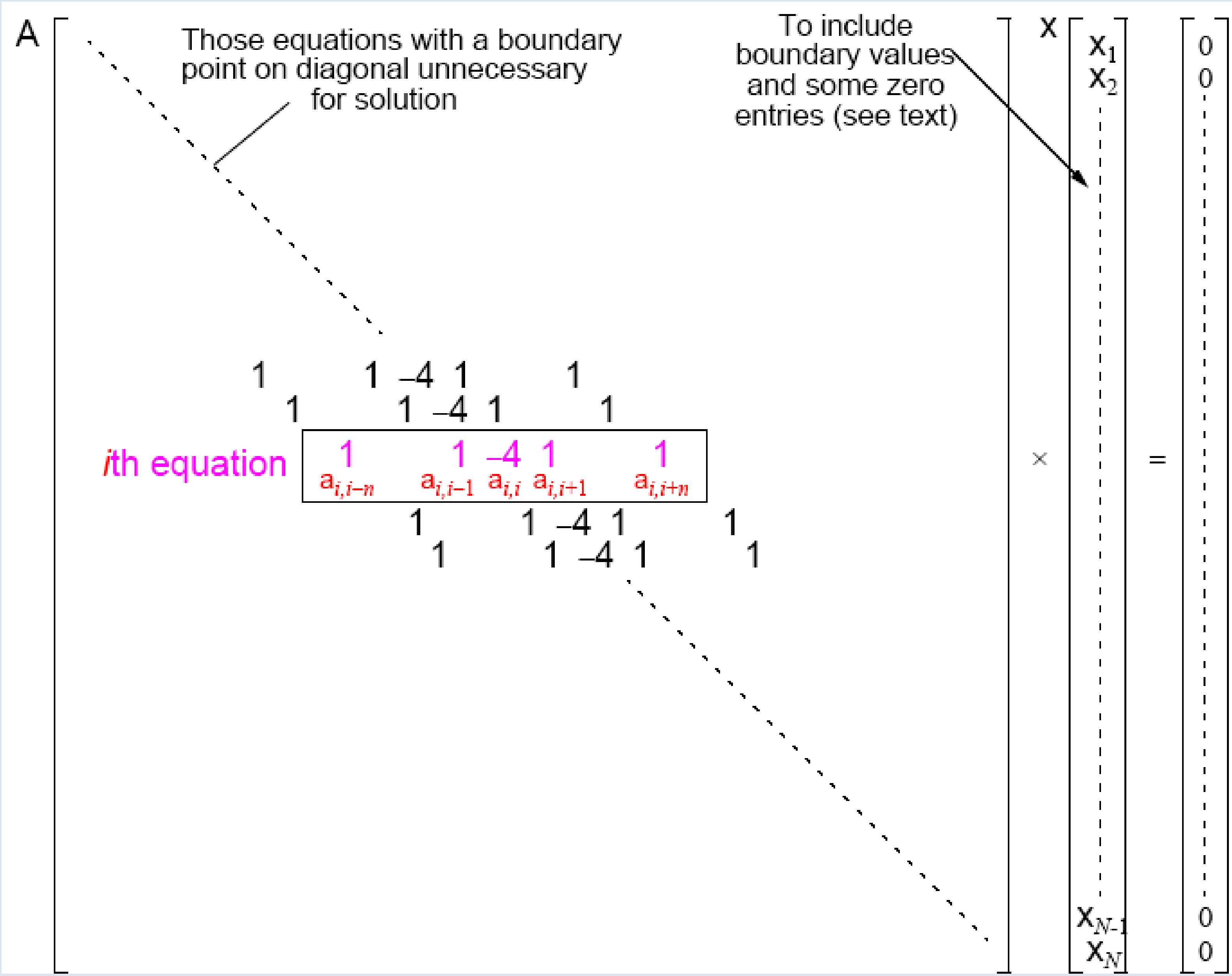
$$\left|a_{ii}\right| \ \natural \sum_{\substack{j=1 \\ j \neq i}}^{n} \left|a_{i,j}\right|$$

- Note that this is not a necessary condition, i.e. the system *may* still have a chance to converge even if A is not diagonally dominant.

Time Complexity: Each iteration takes O(n^2)

Finite Difference Method




A
Those equations with a boundary point on diagonal unnecessary for solution
To include boundary values and some zero entries (see text)
1 1 –4 1 1
1 1 –4 1 1
ith equation
1 1 –4 1 1
$a_{i,i-n}$ $a_{i,i-1}$ $a_{i,i}$ $a_{i,i+1}$ $a_{i,i+n}$
1 1 –4 1 1
1 1 –4 1 1
×
x
x_1
x_2
x_{N-1}
x_N
=
0
0
0
0

Red-Black Ordering

First, **black** points computed simultaneously.
Next, red points computed simultaneously.