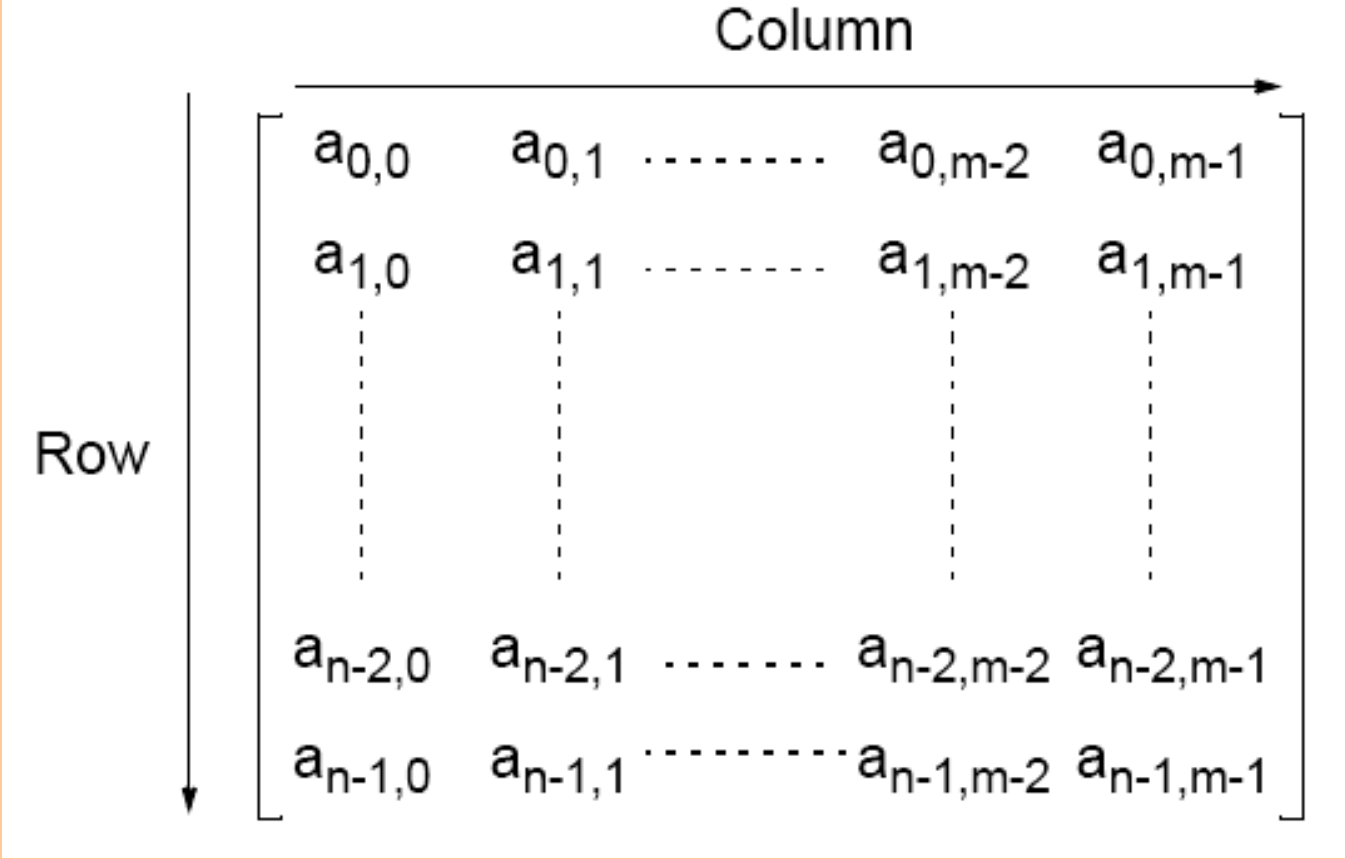
Numerical Algorithms

Matrices

An *n x m* matrix



Matrix Addition

Involves adding corresponding elements of each matrix to form the result matrix.

Given the elements of **A** as *ai*,*j* and the elements of **B** as *bi*,*j*, each element of **C** is computed as

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

$$(0 \leq i < n, 0 \leq j < m)$$

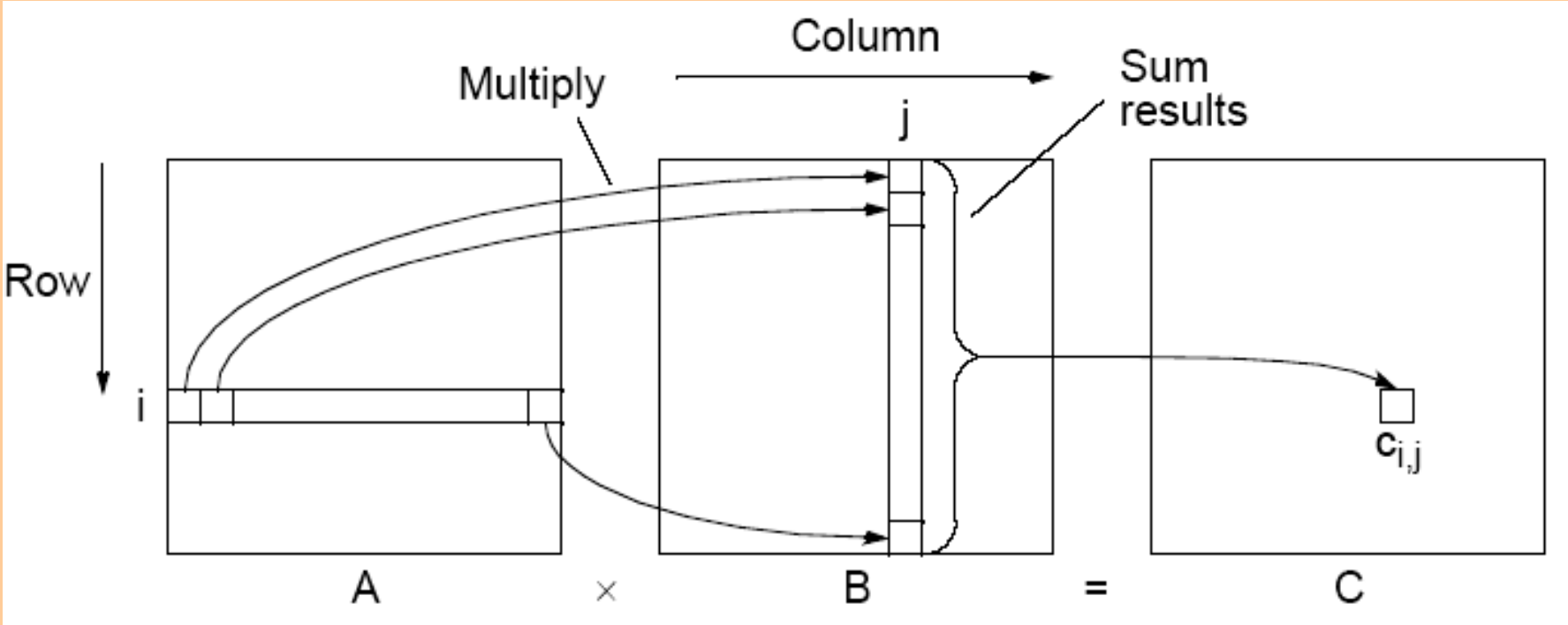
Matrix Multiplication

Multiplication of two matrices, **A** and **B**, produces the matrix **C** whose elements, *ci,j* (0 <= *i* < *n*, 0 <= *j* < *m*), are computed as follows:

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$

where **A** is an *n* x *l* matrix and **B** is an *l* x *m* matrix.

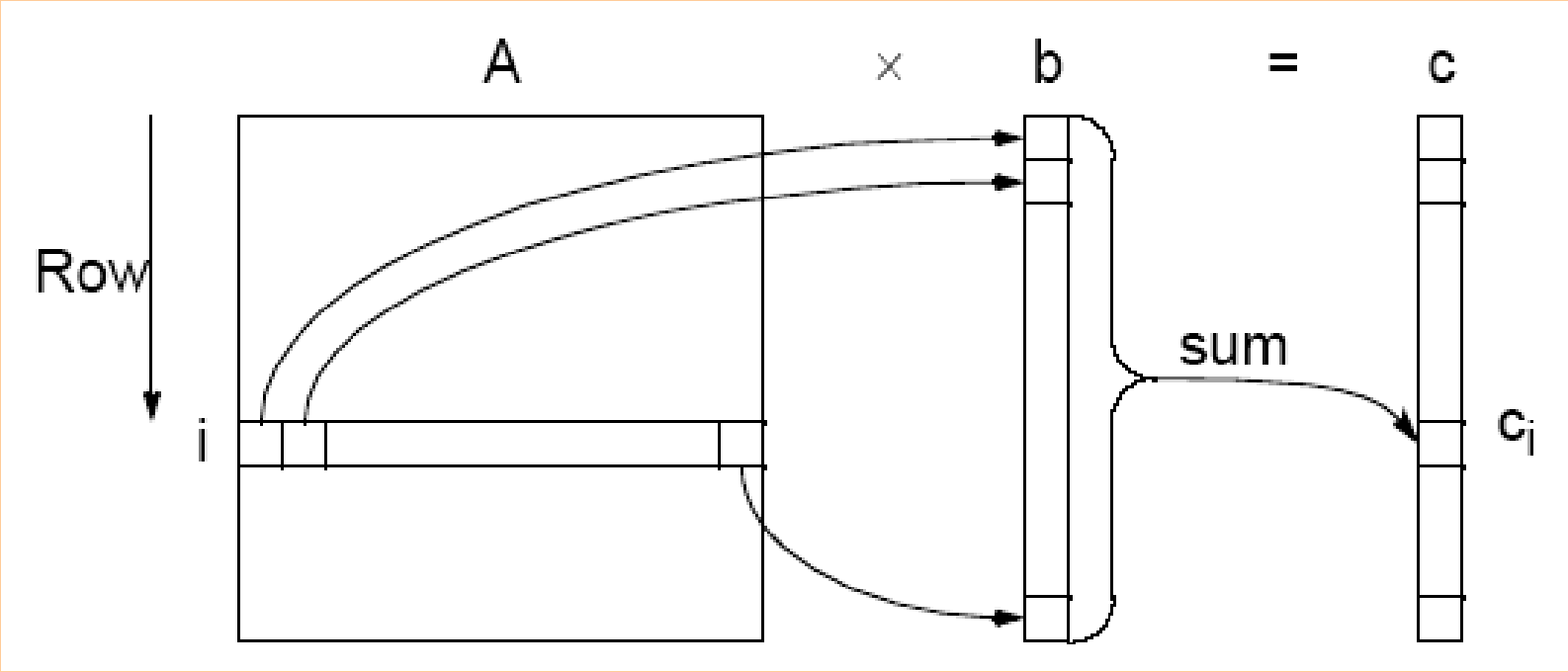
Matrix multiplication, C = A x B



Matrix-Vector Multiplication

c = A x b

Matrix-vector multiplication follows directly from the definition of matrix-matrix multiplication by making **B** an *n* x1 matrix (vector). Result an *n* x 1 matrix (vector).



Relationship of Matrices to Linear Equations

A system of linear equations can be written in matrix form:

$$\mathbf{Ax} = \mathbf{b}$$

Matrix **A** holds the *a* constants

x is a vector of the unknowns

b is a vector of the *b* constants.

Implementing Matrix Multiplication Sequential Code

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

```
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];
}
```

This algorithm requires n^3 multiplications and n^3 additions, leading to a sequential time complexity of $O(n^3)$.
Very easy to parallelize.

Parallel Code

With n processors (and n x n matrices), can obtain:

- Time complexity of $O(n^2)$ with n processors
  Each instance of inner loop independent and can be done by a separate processor

- Time complexity of O(n) with n^2 processors
  One element of A and B assigned to each processor.
  Cost optimal since $O(n^3) = n \times O(n^2) = n^2 \times O(n)$.

- Time complexity of O(log n) with n^3 processors
  By parallelizing the inner loop. Not cost-optimal since $O(n^3) \neq n^3 \times O(\log n)$.

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

Partitioning into Submatrices

Suppose matrix divided into *s*2 submatrices. Each submatrix has *n*/*s* x *n*/*s* elements. Using notation Ap,q as submatrix in submatrix row *p* and submatrix column *q*:

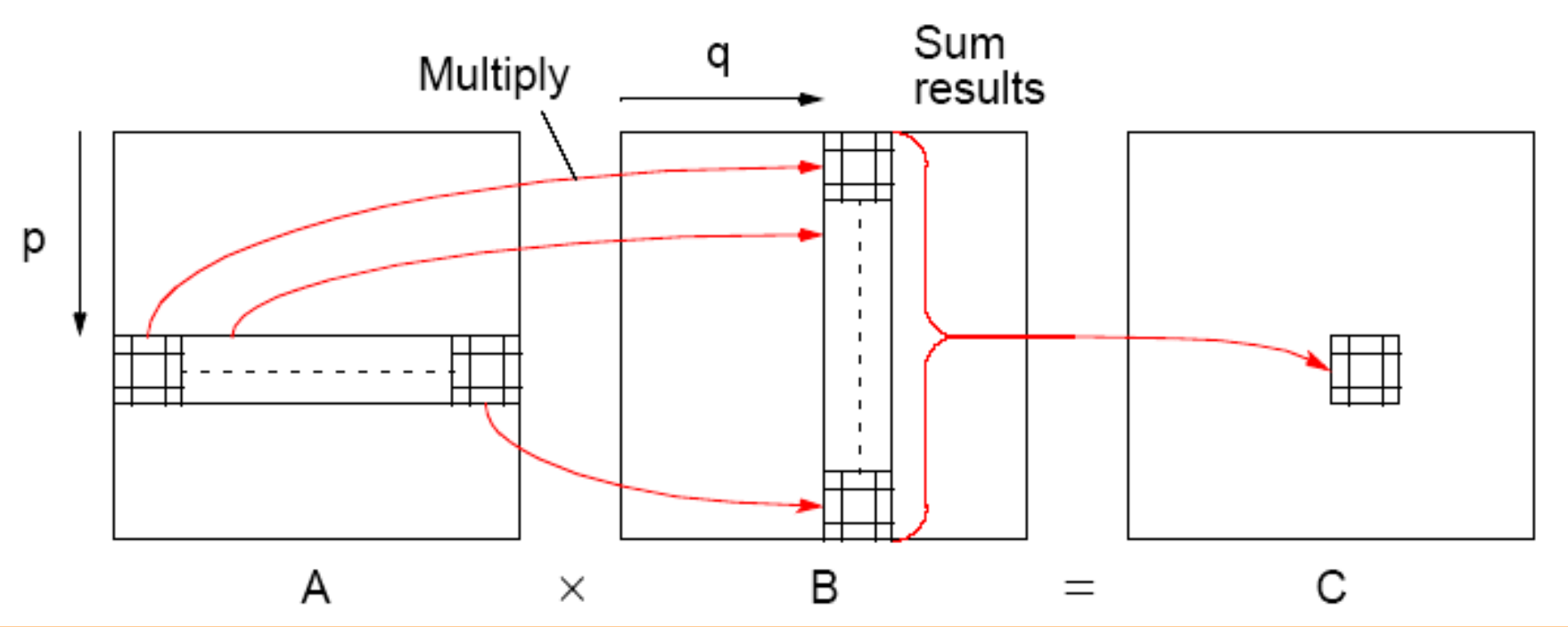
```
for (p = 0; p < s; p++)
  for (q = 0; q < s; q++) {
    Cp,q = 0;                          /* clear elements of submatrix */
    for (r = 0; r < m; r++)            /* submatrix multiplication &*/
      Cp,q = Cp,q + Ap,r * Br,q;       /*add to accum. submatrix*/
}
```

The line

```
Cp,q = Cp,q + Ap,r * Br,q;
```

means multiply submatrix Ap,r and Br,q using matrix multiplication and add to submatrix Cp,q using matrix addition. Known as *block 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 $C_{0,0}$

$$\overset{A_{0,0}}{\left[\begin{array}{cc} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{array}\right]} \times \overset{B_{0,0}}{\left[\begin{array}{cc} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{array}\right]} + \overset{A_{0,1}}{\left[\begin{array}{cc} a_{0,2} & a_{0,3} \\ a_{1,2} & a_{1,3} \end{array}\right]} \times \overset{B_{1,0}}{\left[\begin{array}{cc} b_{2,0} & b_{2,1} \\ b_{3,0} & b_{3,1} \end{array}\right]}$$

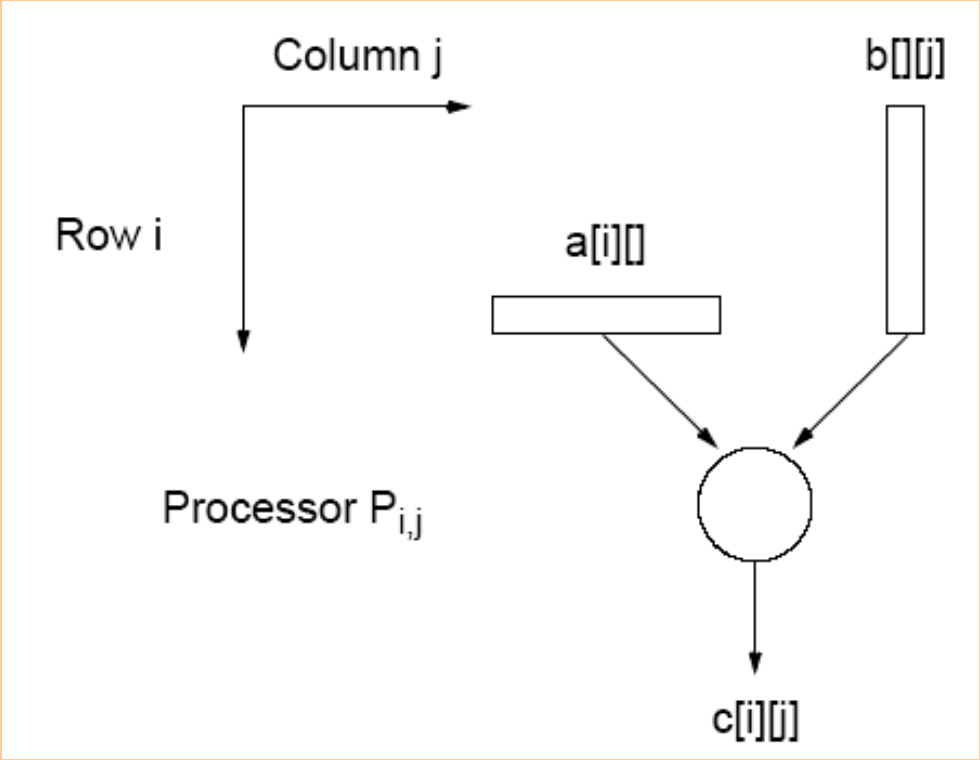
$$= \left[\begin{array}{cc} a_{0,0}b_{0,0}+a_{0,1}b_{1,0} & a_{0,0}b_{0,1}+a_{0,1}b_{1,1} \\ a_{1,0}b_{0,0}+a_{1,1}b_{1,0} & a_{1,0}b_{0,1}+a_{1,1}b_{1,1} \end{array}\right] + \left[\begin{array}{cc} a_{0,2}b_{2,0}+a_{0,3}b_{3,0} & a_{0,2}b_{2,1}+a_{0,3}b_{3,1} \\ a_{1,2}b_{2,0}+a_{1,3}b_{3,0} & a_{1,2}b_{2,1}+a_{1,3}b_{3,1}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

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

$$= C_{0,0}$$

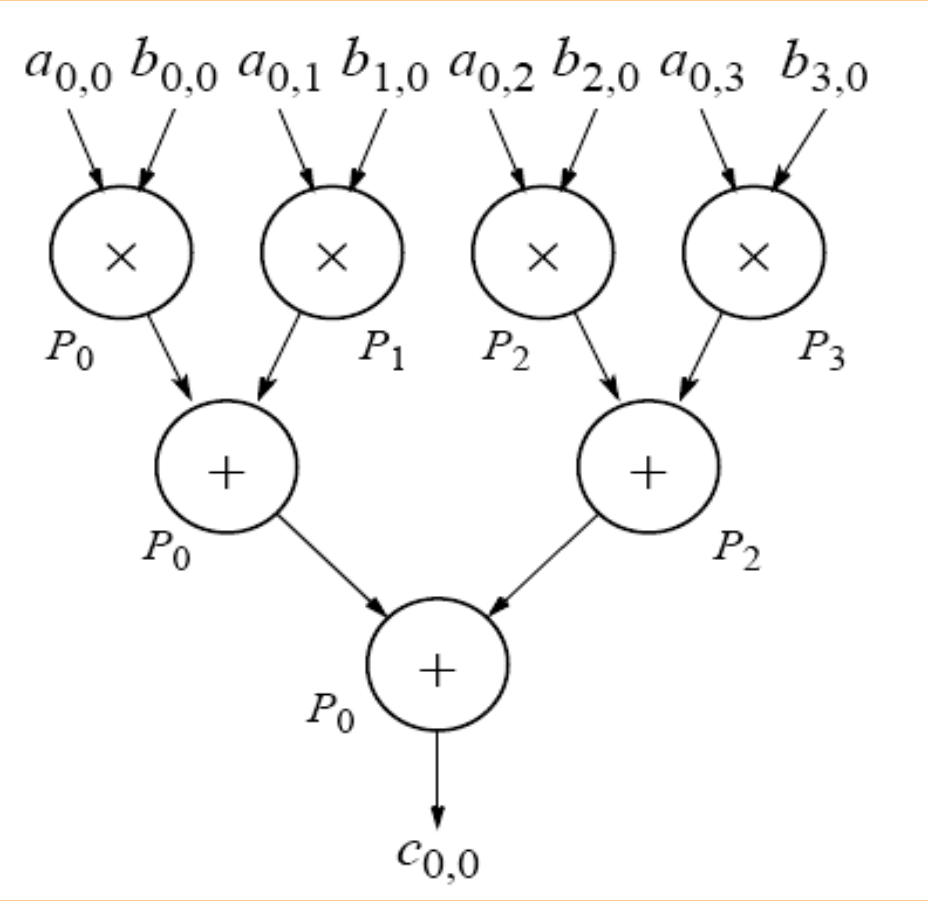
Direct Implementation

One processor to compute each element of **C**. One row of elements of **A** and one column of elements of **B** needed. Some elements sent to more than one processor. Can use submatrices.



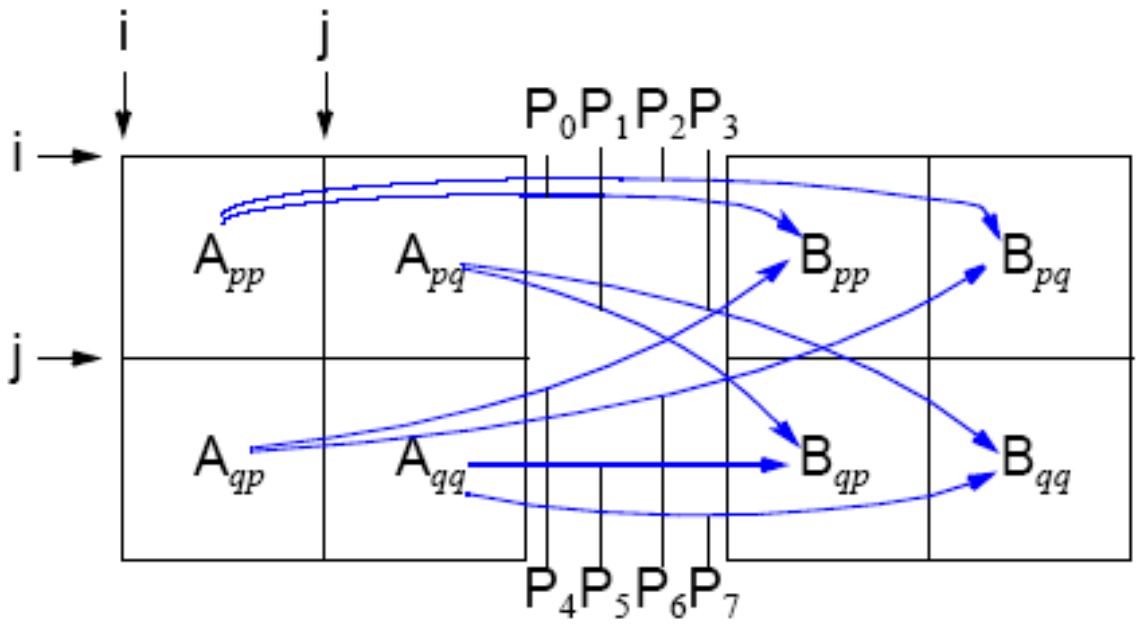
Performance Improvement

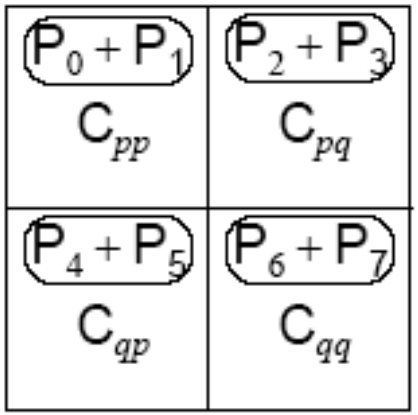
Using tree construction *n* numbers can be added in log *n* steps using *n* processors:



Computational time complexity of O(log *n*) using n^3 processors.

Recursive Implementation





Apply same algorithm on each submatrix recursivly.

Excellent algorithm for a shared memory systems because of locality of operations.

Recursive Algorithm

```
mat_mult(A , B , s)
{
if (s == 1)                 /* if submatrix has one element */
    C = A * B;              /* multiply elements */
else {                      /* continue to make recursive calls */
    s = s/2;                /* no of elements in each row/column */
    P0 = mat_mult(App, Bpp, s);
    P1 = mat_mult(Apq, Bqp, s);
    P2 = mat_mult(App, Bpq, s);
    P3 = mat_mult(Apq, Bqq, s);
    P4 = mat_mult(Aqp, Bpp, s);
    P5 = mat_mult(Aqq, Bqp, s);
    P6 = mat_mult(Aqp, Bpq, s);
    P7 = mat_mult(Aqq, Bqq, s);
    Cpp = P0 + P1;          /* add submatrix products to */
    Cpq = P2 + P3;          /* form submatrices of final matrix */
    Cqp = P4 + P5;
    Cqq = P6 + P7;
}
return (C);                 /* return final matrix */
}
```

Mesh Implementations

- Cannon's algorithm
- Fox's algorithm (not in textbook but similar complexity)
- Systolic array

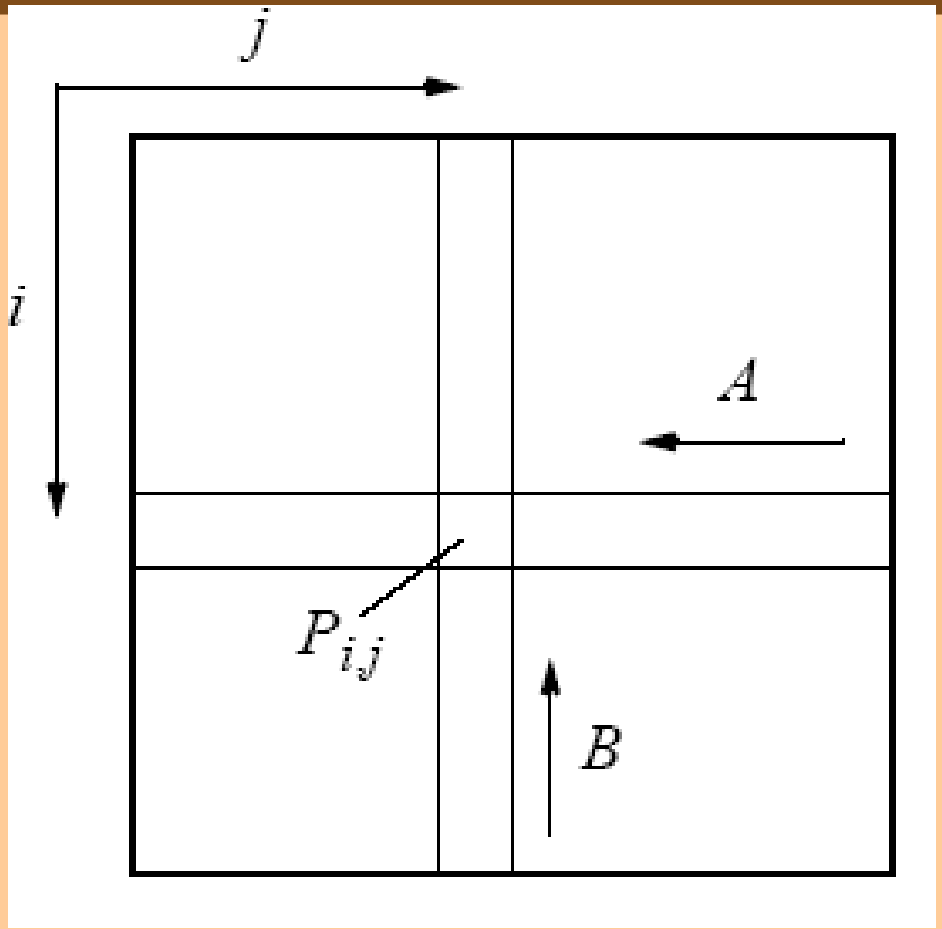
All involve using processor arranged a mesh and shifting elements of the arrays through the mesh. Accumulate the partial sums at each processor.

Mesh Implementations
Cannon's Algorithm

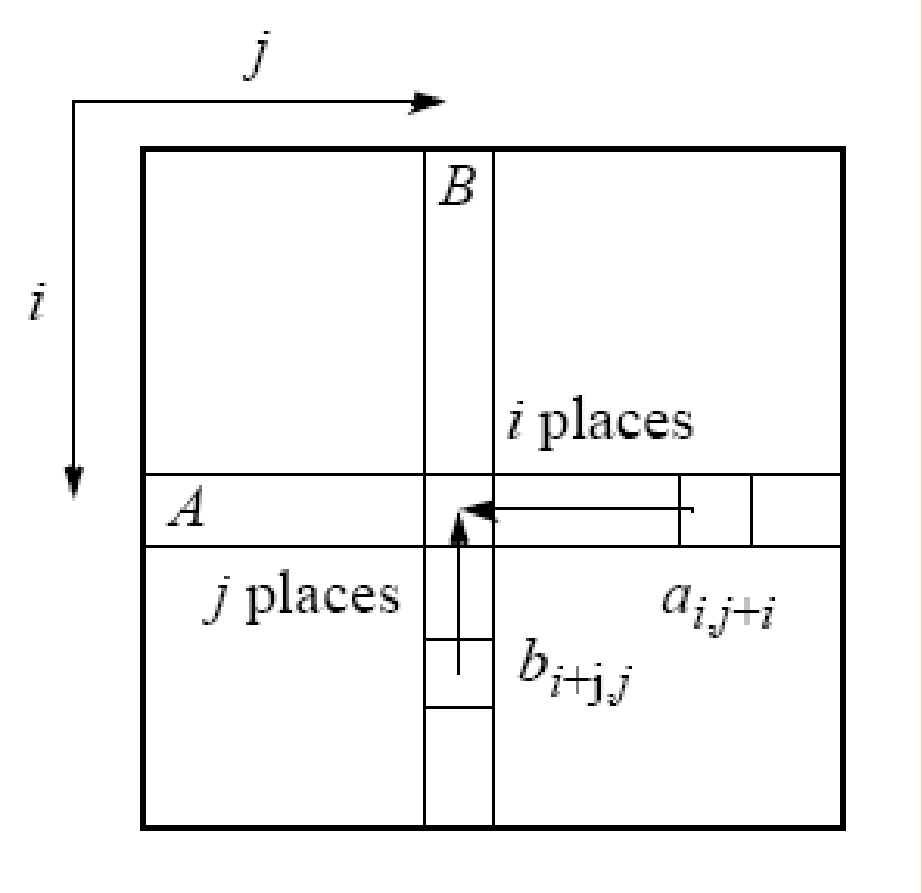
Uses a mesh of processors with wraparound connections (a torus) to shift the A elements (or submatrices) left and the B elements (or submatrices) up.

1. Initially processor Pi,j has elements ai,j and bi,j (0 <= i < n, 0 <= k < n).
2. Elements are moved from their initial position to an "aligned" position. The complete ith row of A is shifted i places left and the complete jth column of B is shifted j places upward. This has the effect of placing the element ai,j+i and the element bi+j,j in processor Pi,j,. These elements are a pair of those required in the accumulation of ci,j.
3. Each processor, Pi,j, multiplies its elements.
4. The ith row of A is shifted one place right, and the jth 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.
5. Each processor, Pi,j, multiplies the elements brought to it and adds the result to the accumulating sum.
6. Step 4 and 5 are repeated until the final result is obtained (n - 1 shifts with n rows and n columns of elements).

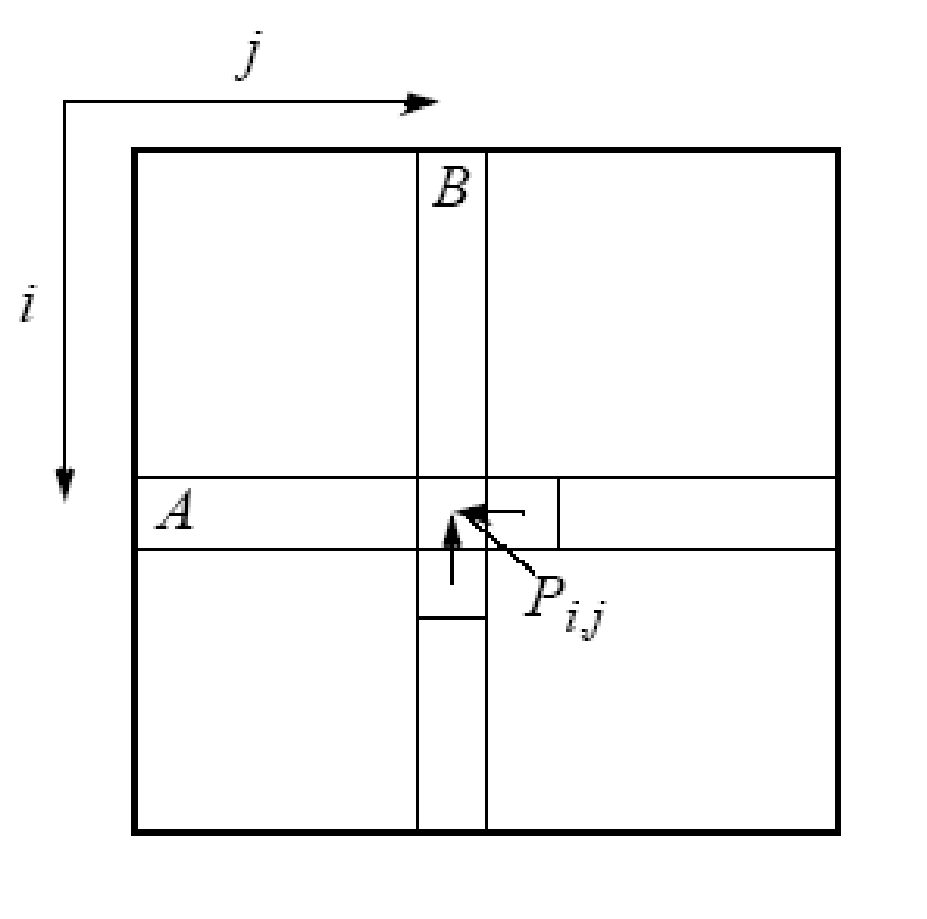
Movement of *A* and *B* elements



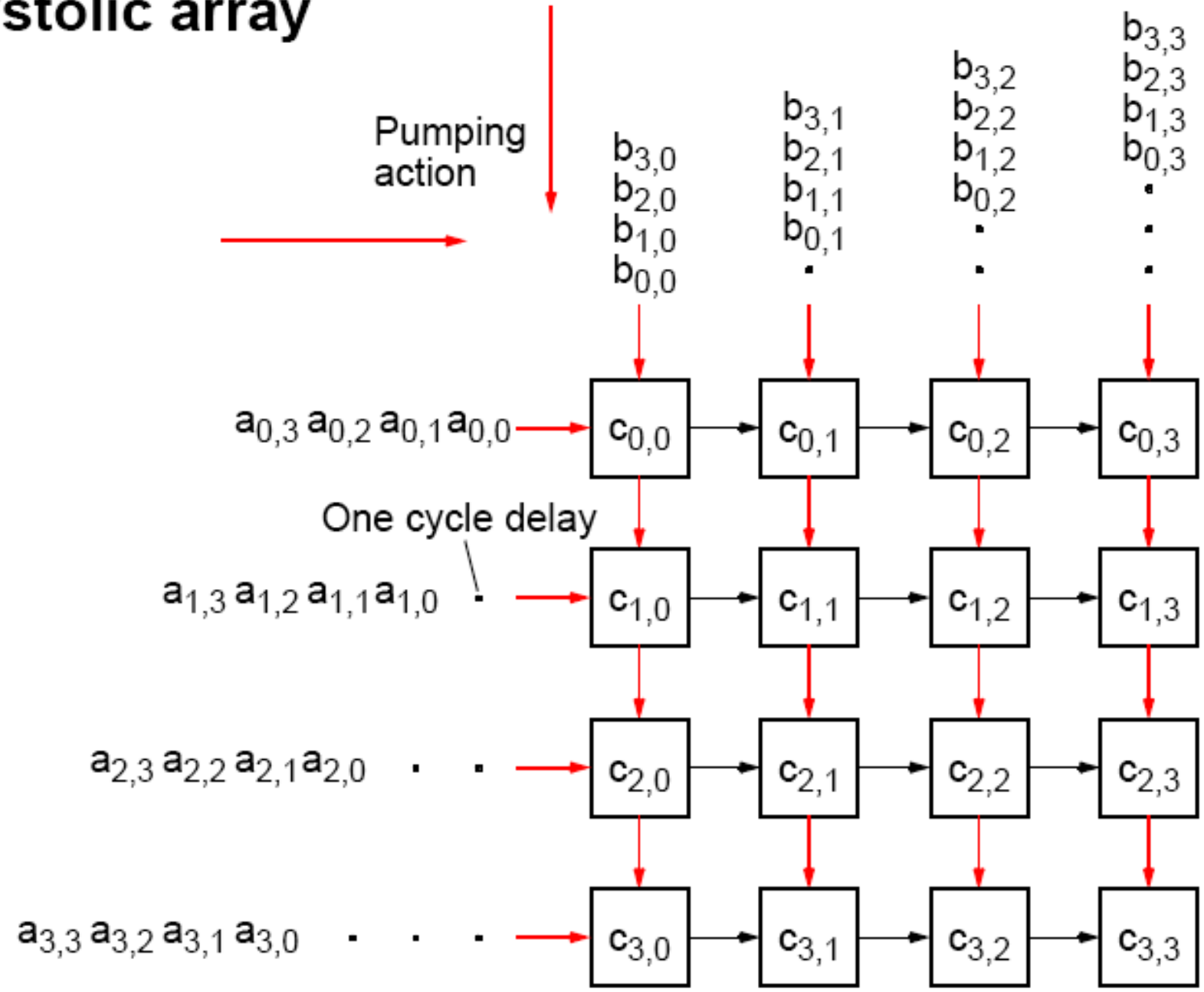
Step 2 — Alignment of elements of *A* and *B*



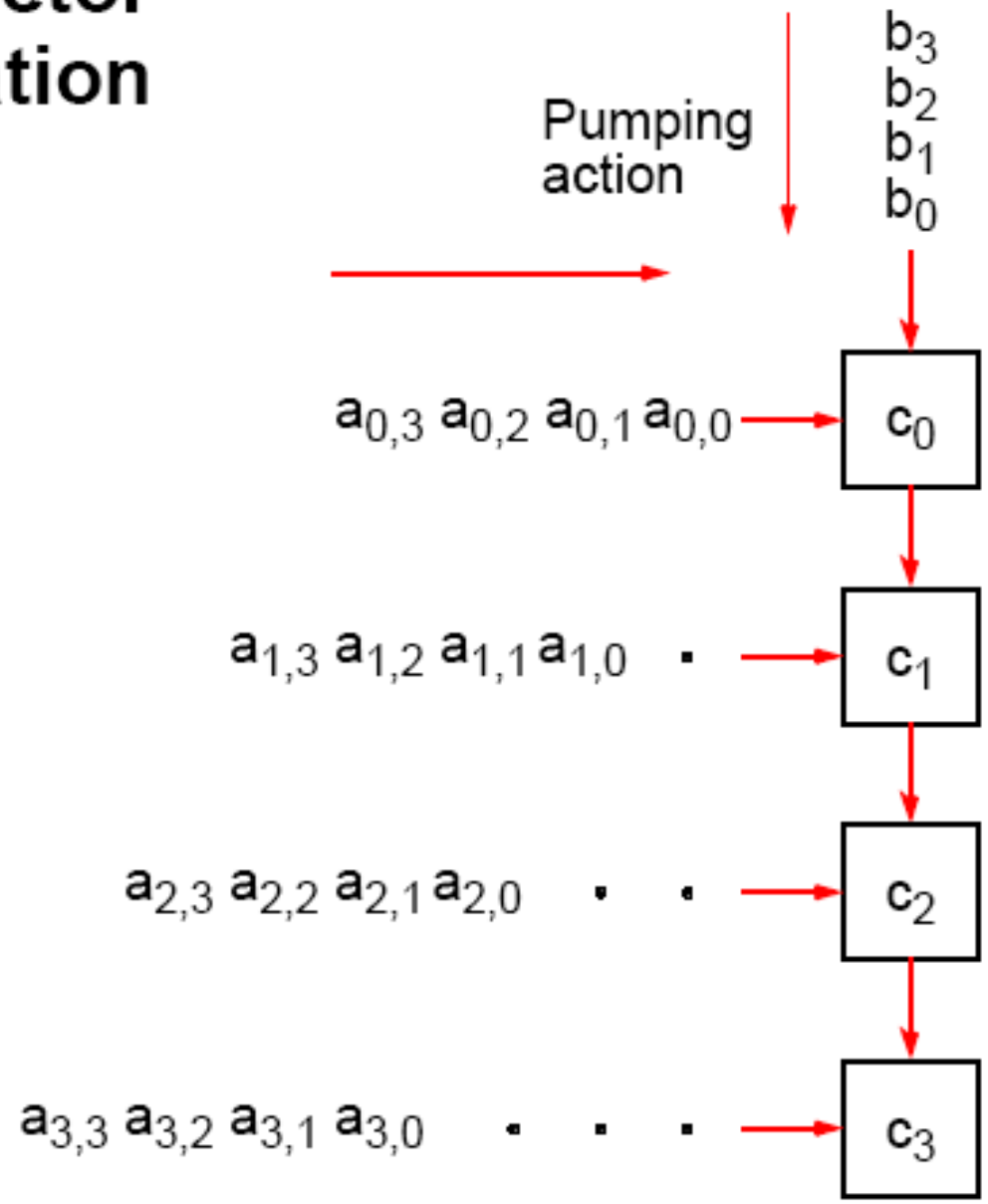
Step 4 - One-place shift of elements of *A* and *B*



Systolic array



Matrix-Vector Multiplication



Solving a System of Linear Equations

$$
\begin{aligned}
a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \quad &\ldots \quad + a_{n-1,n-1}x_{n-1} = b_{n-1} \\
&\vdots \\
a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 \quad &\ldots \quad + a_{2,n-1}x_{n-1} = b_2 \\
a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 \quad &\ldots \quad + a_{1,n-1}x_{n-1} = b_1 \\
a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 \quad &\ldots \quad + a_{0,n-1}x_{n-1} = b_0
\end{aligned}
$$

which, in matrix form, is

$$\mathbf{Ax = b}$$

Objective is to find values for the unknowns, $x0$, $x1$, …, xn-1, given values for $a0,0$, $a0,1$, …, an-1,n-1, and $b0$, …, bn .

Solving a System of Linear Equations

Dense matrices

Gaussian Elimination - parallel time complexity $O(n^2)$

Sparse matrices

By iteration - depends upon iteration method and number of iterations but typically O(log n)

- Jacobi iteration
- Gauss-Seidel relaxation (not good for parallelization)
- Red-Black ordering
- Multigrid

Gaussian Elimination

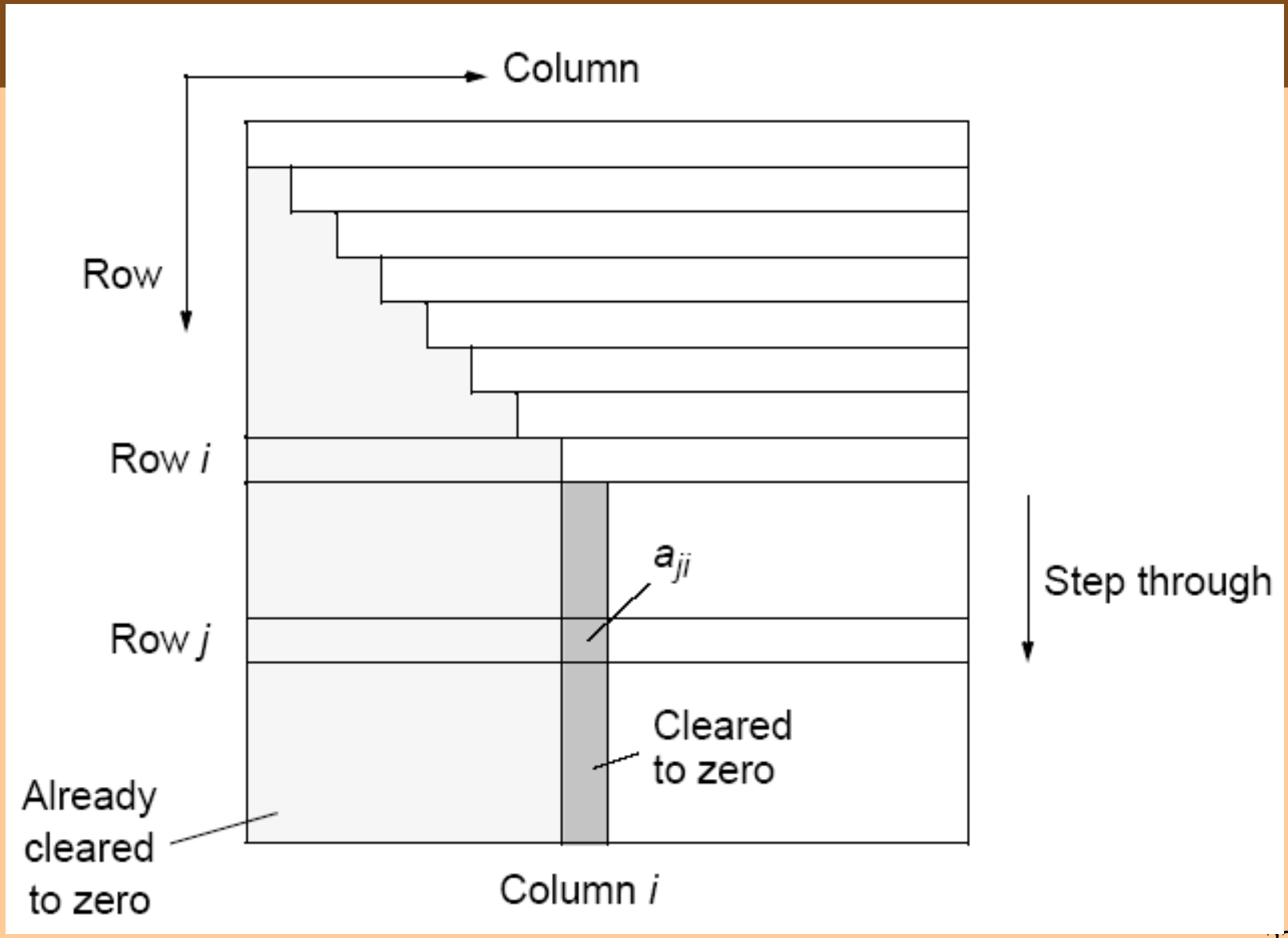
Convert general system of linear equations into triangular system of equations to be solved by Back Substitution.

Uses characteristic of linear equations that any row can be replaced by that row added to another row multiplied by a constant.

Starts at the first row and works toward the bottom row. At the *i*th row, each row *j* below the *i*th row is replaced by row *j* + (row *i*) (-*aj,i*/*ai,i*). The constant used for row *j* is -*aj,i*/*ai,i*. Has the effect of making all the elements in the *i*th column below the *i*th row zero because

$$a_{j,i} = a_{j,i} + a_{i,i}\left(\frac{-a_{j,i}}{a_{i,i}}\right) = 0$$

Gaussian elimination



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 swapping 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.)

In the following, we will not consider partial pivoting.

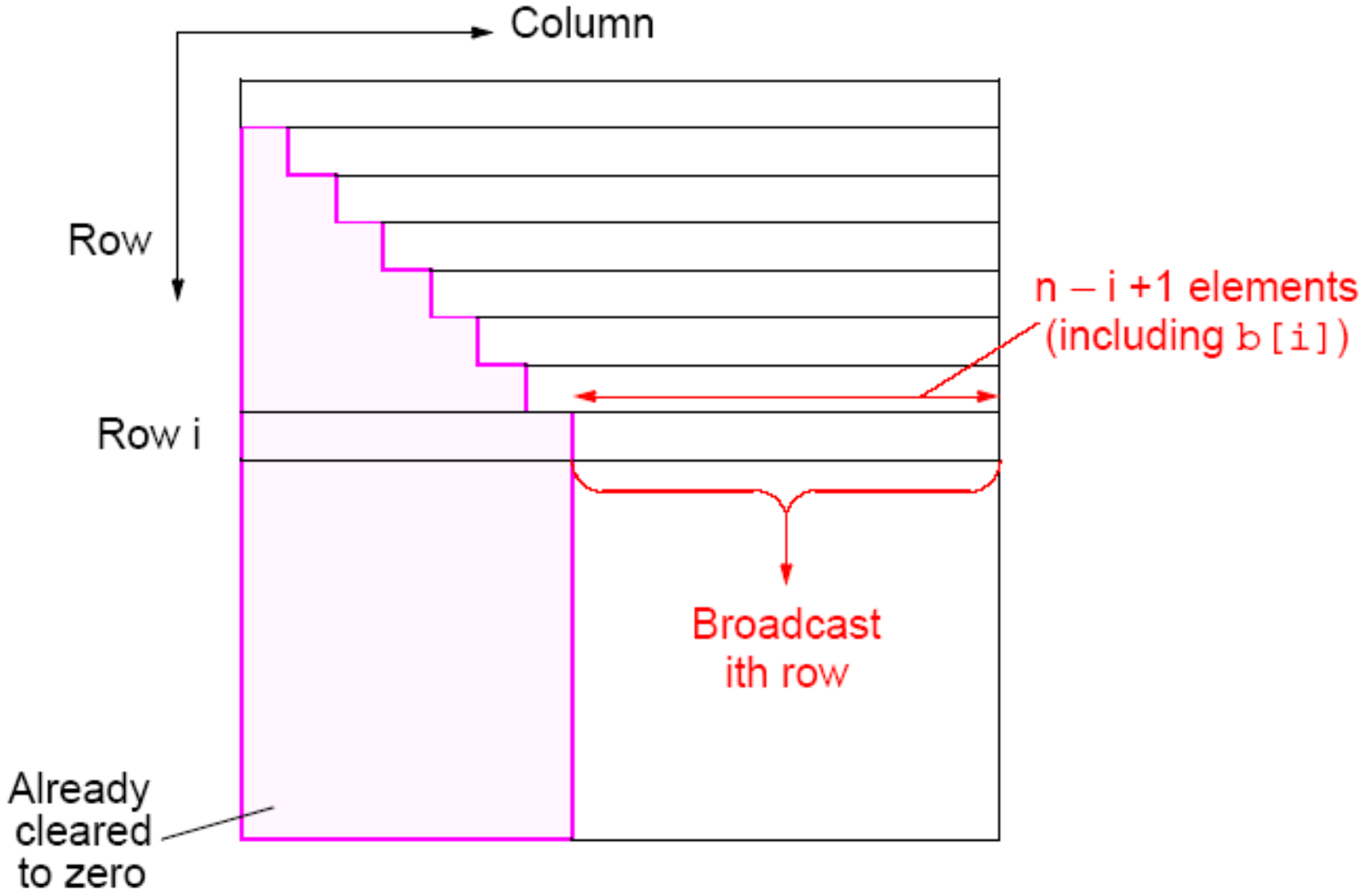
Sequential Code

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 is $O(n^3)$.

Parallel Implementation



Analysis

Communication

n - 1 broadcasts performed sequentially. *i*th broadcast contains *n* - *i* + 1 elements.

Time complexity of $O(n^2)$ (see textbook)

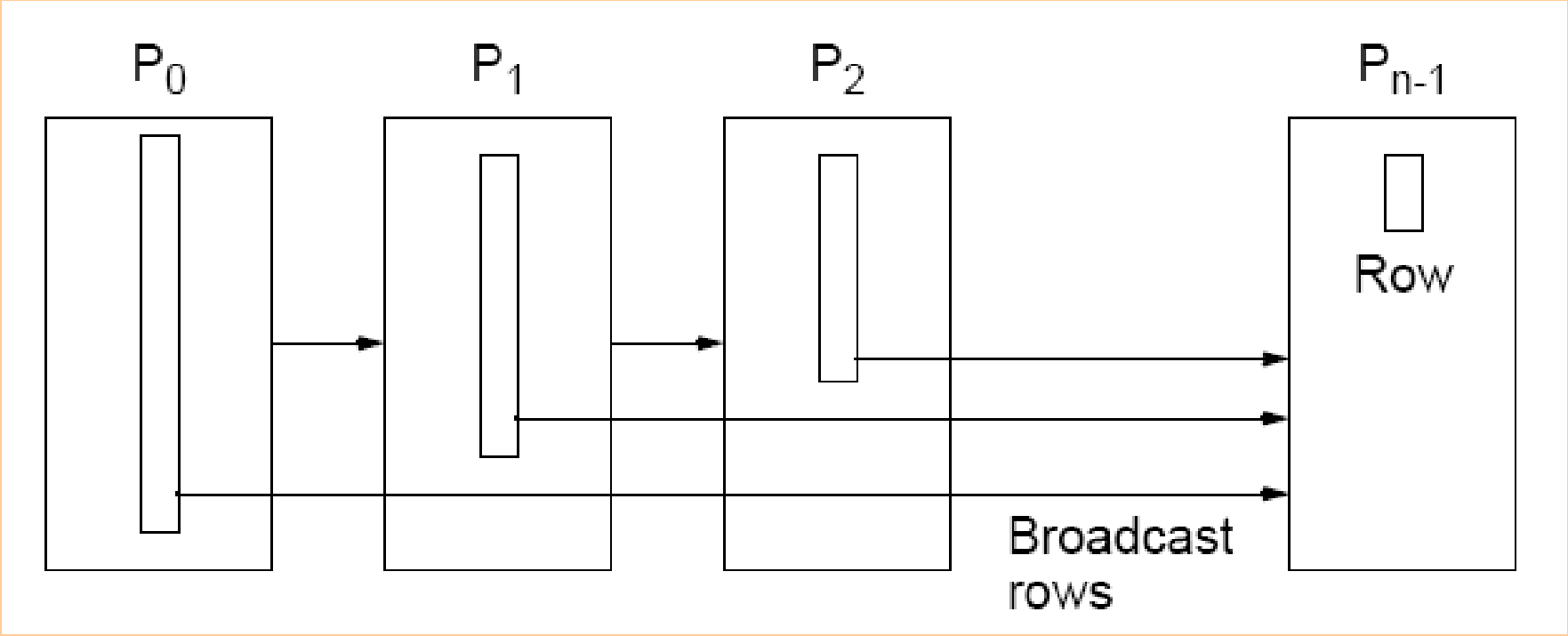
Computation

After row broadcast, each processor *Pj* beyond broadcast processor *Pi* 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.

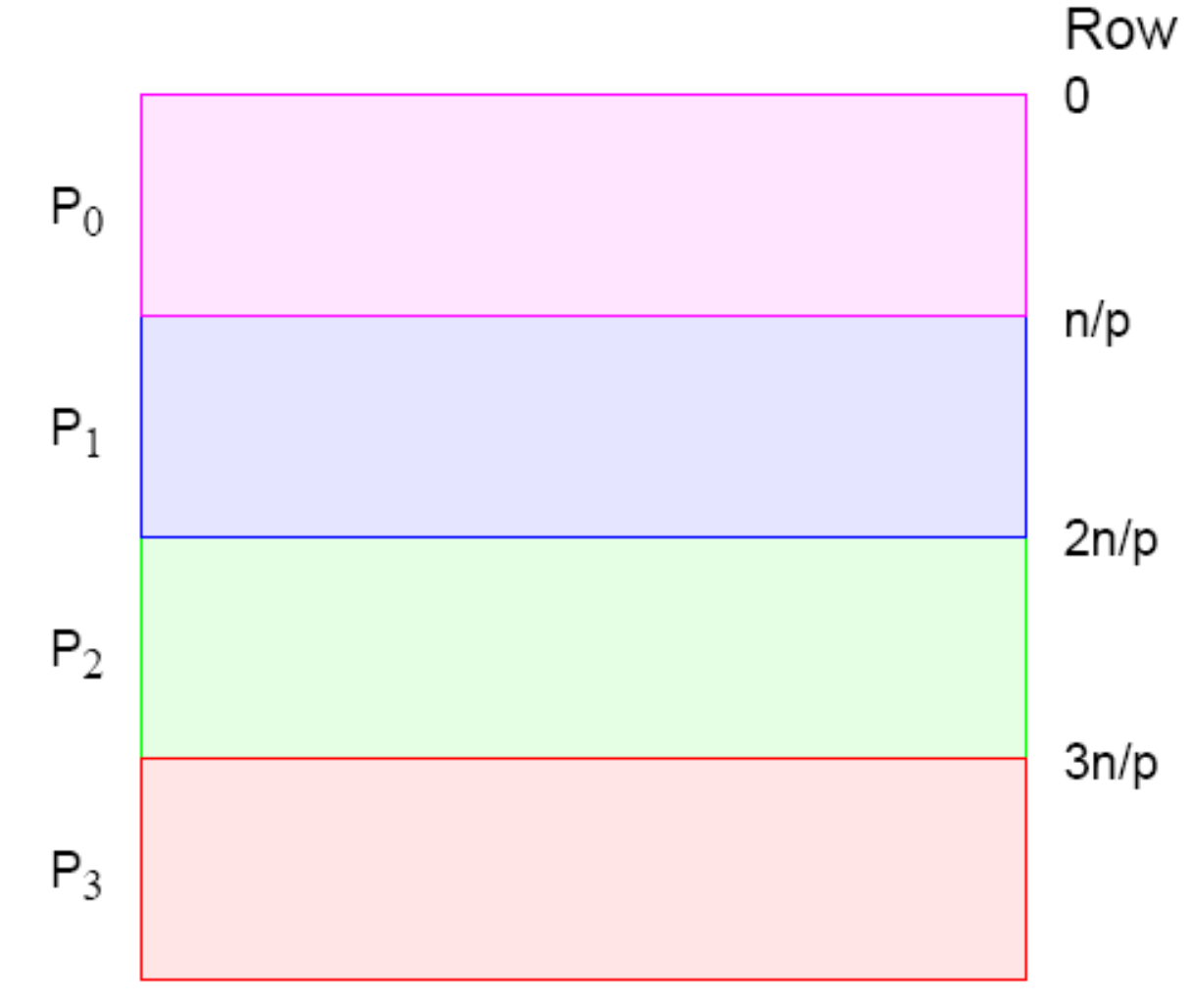
Time complexity of $O(n^2)$ (see textbook).

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

Pipeline implementation of Gaussian elimination



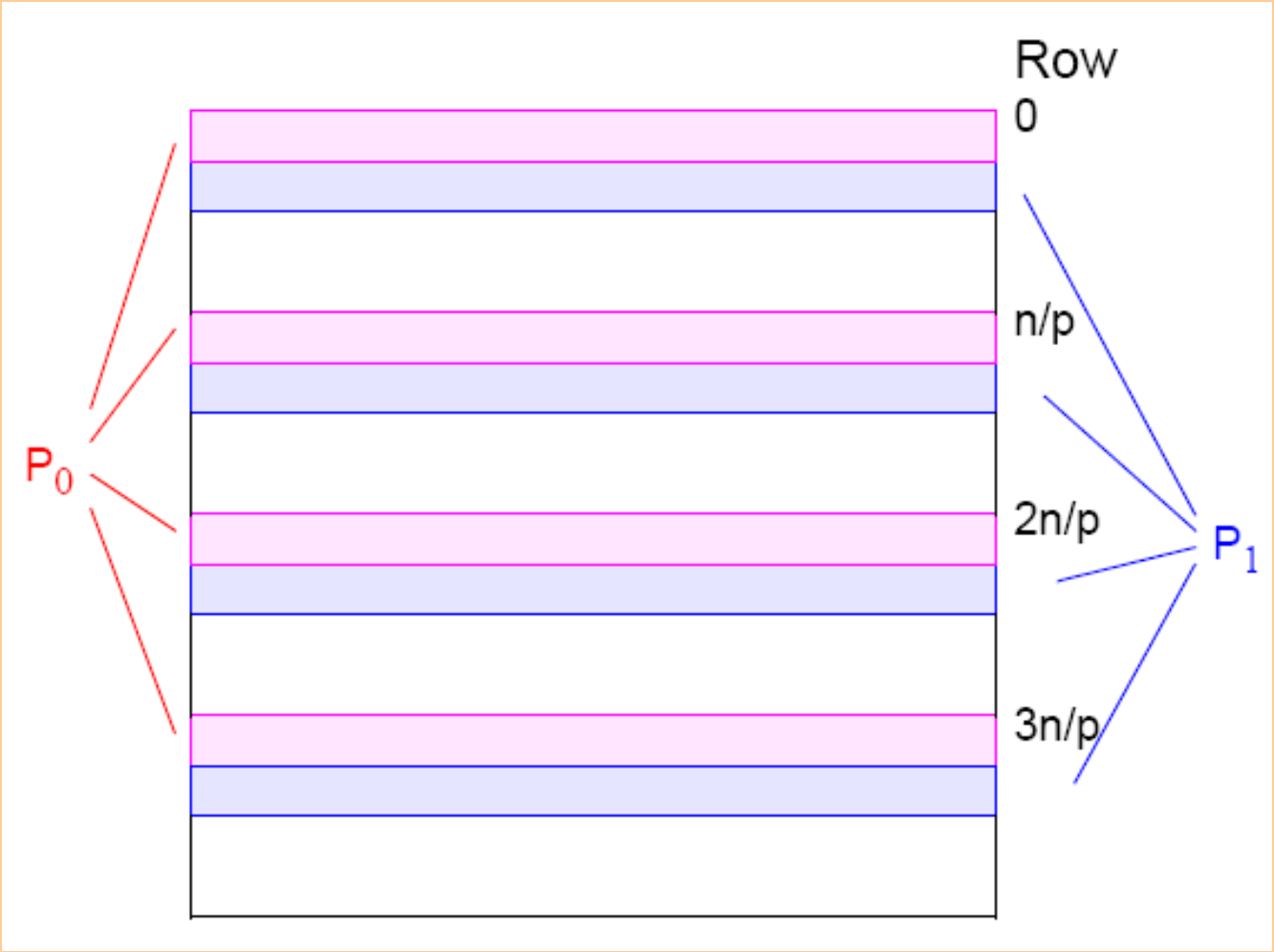
Strip Partitioning



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:



Iterative Methods

Time complexity of direct method at $O(N^2)$ with N processors, is significant.

Time complexity of iteration method depends upon:

- the type of iteration,
- number of iterations
- number of unknowns, and
- required accuracy

but can be less than the direct method especially for a few unknowns i.e a sparse system of linear equations.

Jacobi Iteration

Iteration formula - *i*th equation rearranged to have *i*th unknown on left side:

$$x_i^k = \frac{1}{a_{i,i}}\left[b_i - \sum_{j \neq i} a_{i,j} x_j^{k-1}\right]$$

Superscript indicates iteration:

x_i^k *is* kth iteration of x_i , x_j^{k-1} is $(k-1)$th iteration of x_j.

Example of a Sparse System of Linear Equations

Laplace's Equation

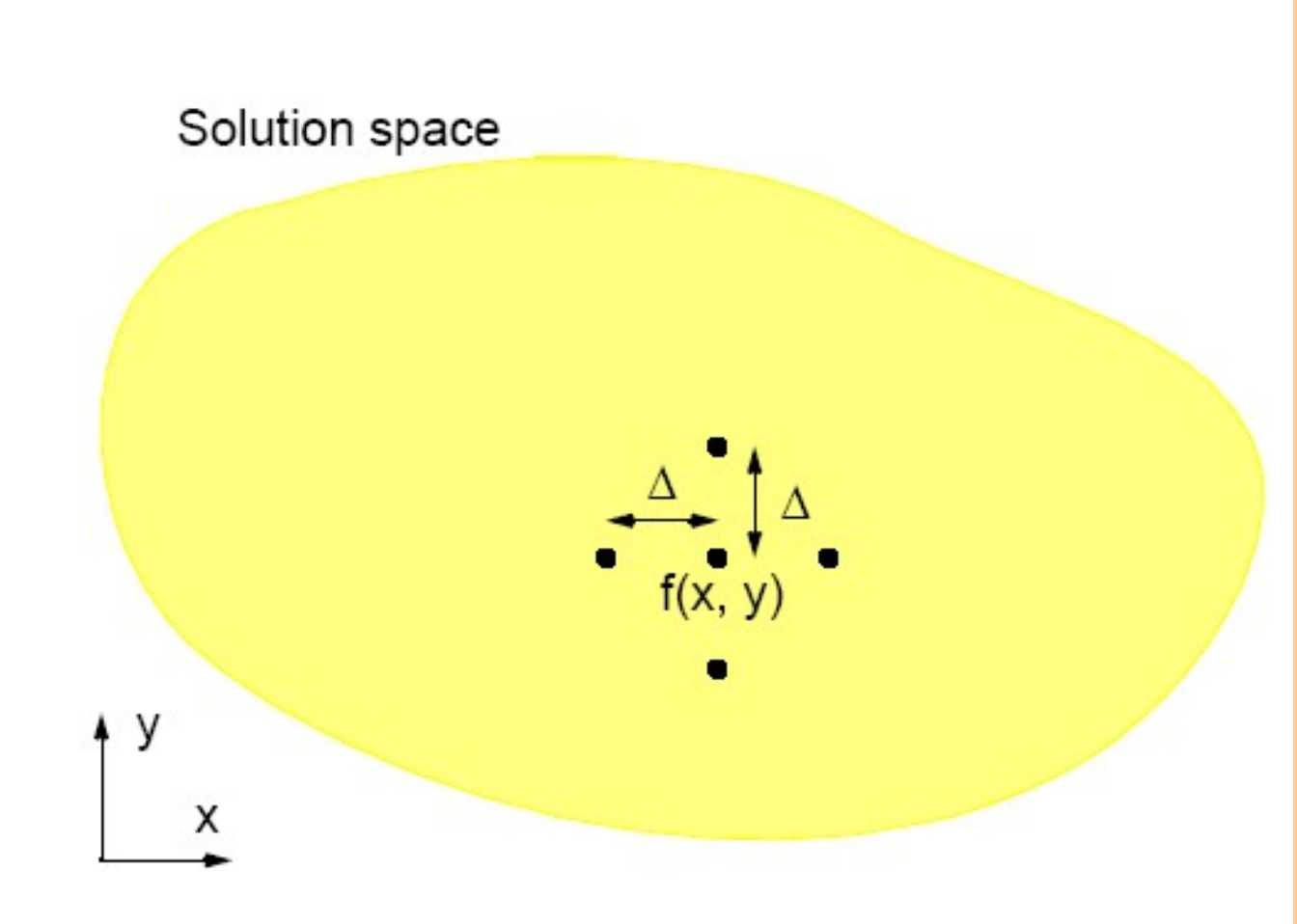
$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0$$

Solve for *f* over the two-dimensional x-y space.

For a computer solution, *finite difference* methods are appropriate

Two-dimensional solution space is "discretized" into a large number of solution points.

Finite Difference Method



If distance between points, Δ, made small enough:

$$\frac{\partial^2 f}{\partial x^2} \approx \frac{1}{\Delta^2}[f(x+\Delta, y) - 2f(x, y) + f(x-\Delta, y)]$$

$$\frac{\partial^2 f}{\partial y^2} \approx \frac{1}{\Delta^2}[f(x, y+\Delta) - 2f(x, y) + f(x, y-\Delta)]$$

Substituting into Laplace's equation, we get

$$\frac{1}{\Delta^2}[f(x+\Delta, y) + f(x-\Delta, y) + f(x, y+\Delta) + f(x, y-\Delta) - 4f(x, y)] = 0$$

Rearranging, we get

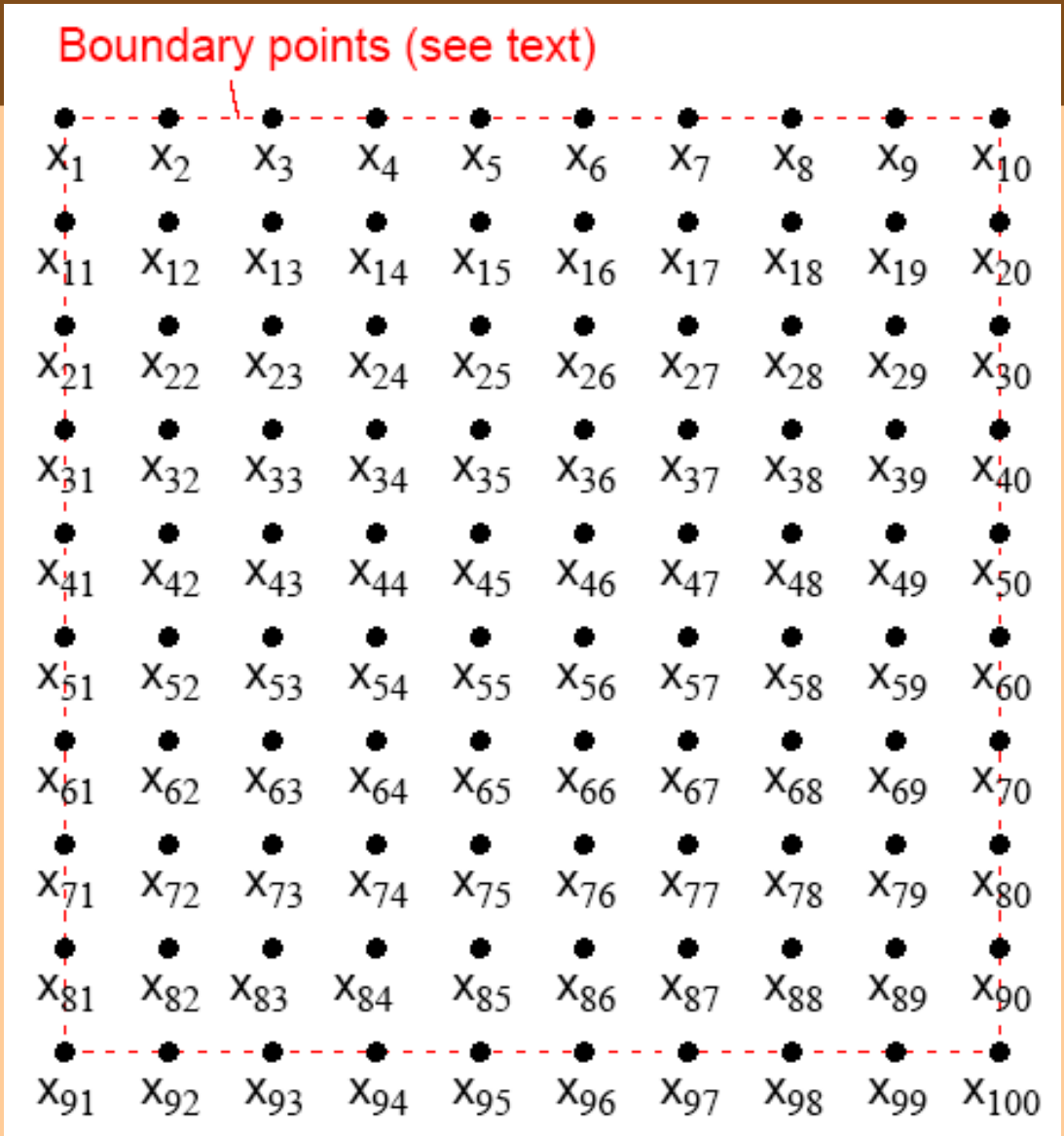
$$f(x, y) = \frac{[f(x-\Delta, y) + f(x, y-\Delta) + f(x+\Delta, y) + f(x, y+\Delta)]}{4}$$

Rewritten as an iterative formula:

$$f^k(x, y) = \frac{[f^{k-1}(x-\Delta, y) + f^{k-1}(x, y-\Delta) + f^{k-1}(x+\Delta, y) + f^{k-1}(x, y+\Delta)]}{4}$$

$f^k(x, y)$ - kth iteration, $f^{k-1}(x, y)$ - $(k-1)$th iteration.

Natural Order



Relationship with a General System of Linear Equations

Using natural ordering, *i*th point computed from *i*th equation:

$$x_i = \frac{x_{i-n} + x_{i-1} + x_{i+1} + x_{i+n}}{4}$$

or

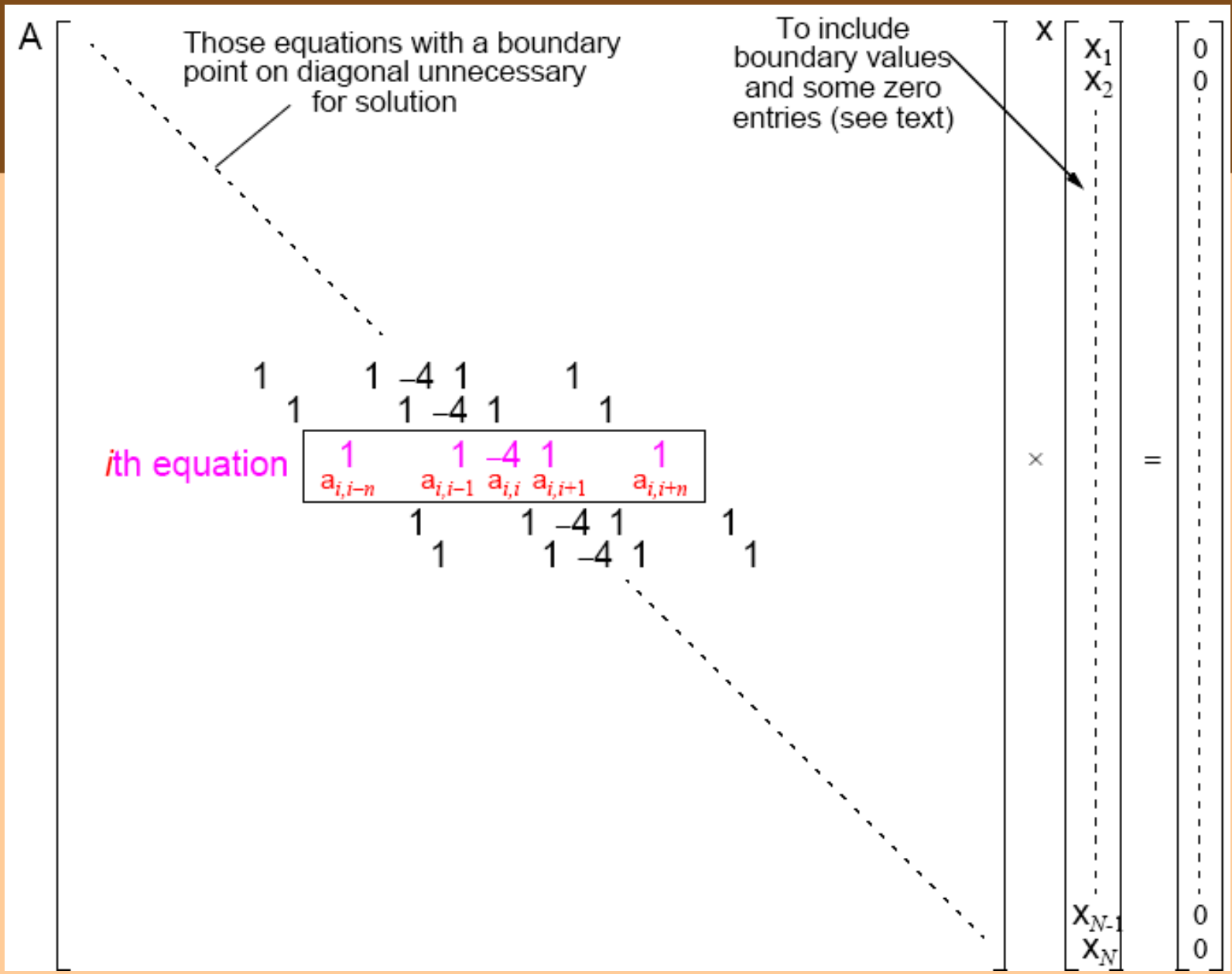
$$x_{i-n} + x_{i-1} - 4x_i + x_{i+1} + x_{i+n} = 0$$

which is a linear equation with five unknowns (except those with boundary points).

In general form, the *i*th equation becomes:

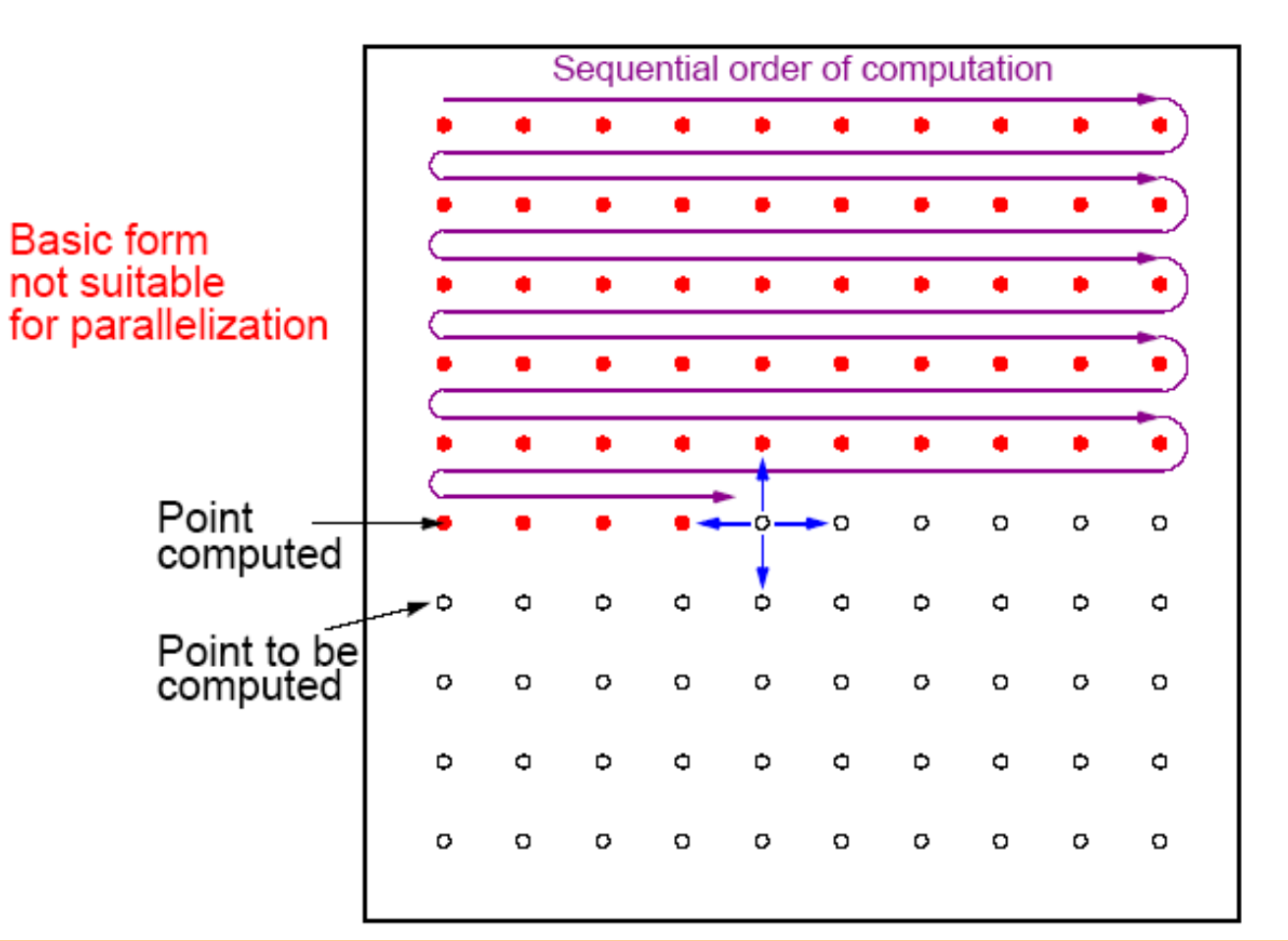
$$a_{i,i-n}x_{i-n} + a_{i,i-1}x_{i-1} + a_{i,i}x_i + a_{i,i+1}x_{i+1} + a_{i,i+n}x_{i+n} = 0$$

where $a_{i,i} = -4$, and $a_{i,i-n} = a_{i,i-1} = a_{i,i+1} = a_{i,i+n} = 1$.


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

Gauss-Seidel Relaxation

Uses some newly computed values to compute other values in that iteration.



Gauss-Seidel Iteration Formula

$$x_i^k = \frac{1}{a_{i,i}}\left[b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^k - \sum_{j=i+1}^{N} a_{i,j} x_j^{k-1} \right]$$

where the superscript indicates the iteration.

With natural ordering of unknowns, formula reduces to

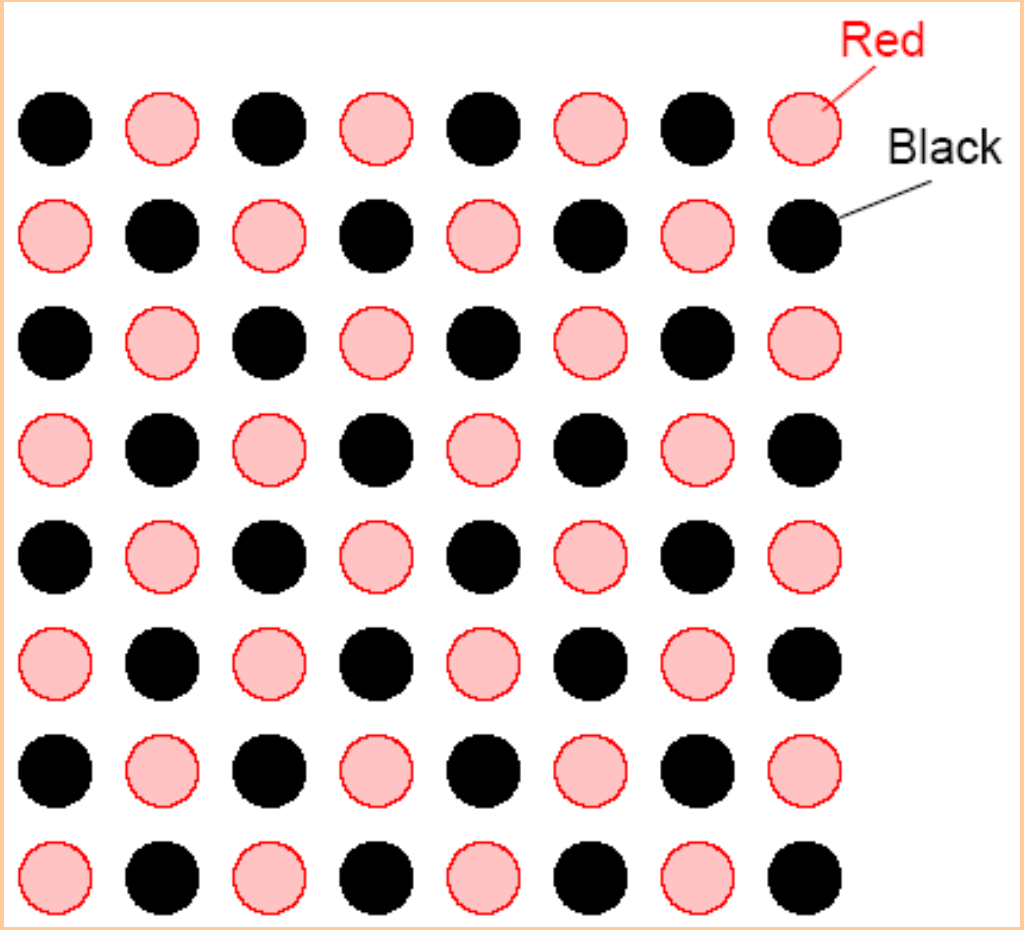
$$x_i^k = (-1/a_{i,i})[a_{i,i-n} x_{i-n}^k + a_{i,i-1} x_{i-1}^k + a_{i,i+1} x_{i+1}^{k-1} + a_{i,i+n} x_{i+n}^{k-1}]$$

At the *k*th iteration, two of the four values (before the *i*th element) taken from the *k*th iteration and two values (after the *i*th element) taken from the (*k*-1)th iteration. We have:

$$f^k(x,y) = \frac{[f^k(x-\Delta,y) + f^k(x,y-\Delta) + f^{k-1}(x+\Delta,y) + f^{k-1}(x,y+\Delta)]}{4}$$

Red-Black Ordering

First, black points computed. Next, red points computed. Black points computed simultaneously, and red points computed simultaneously.



Red-Black Parallel Code

```
forall (i = 1; i < n; i++)
      forall (j = 1; j < n; j++)
        if ((i + j) % 2 == 0)              /* compute red points */
            f[i][j] = 0.25*(f[i-1][j] + f[i][j-1] + f[i+1][j] + f[i][j+1]);
forall (i = 1; i < n; i++)
     forall (j = 1; j < n; j++)
       if ((i + j) % 2 != 0)                /* compute black points */
           f[i][j] = 0.25*(f[i-1][j] + f[i][j-1] + f[i+1][j] + f[i][j+1]);
```

Multigrid Method

First, a coarse grid of points used. With these points, iteration process will start to converge quickly.

At some stage, number of points increased to include points of the coarse grid and extra points between the points of the coarse grid. Initial values of extra points found by interpolation. Computation continues with this finer grid.

Grid can be made finer and finer as computation proceeds, or computation can alternate between fine and coarse grids.

Coarser grids take into account distant effects more quickly and provide a good starting point for the next finer grid.

Multigrid processor allocation