Chapter 11

Numerical Algorithms

@ 2004 Pearson Education Inc. All rights reserved.

Numerical Algorithms

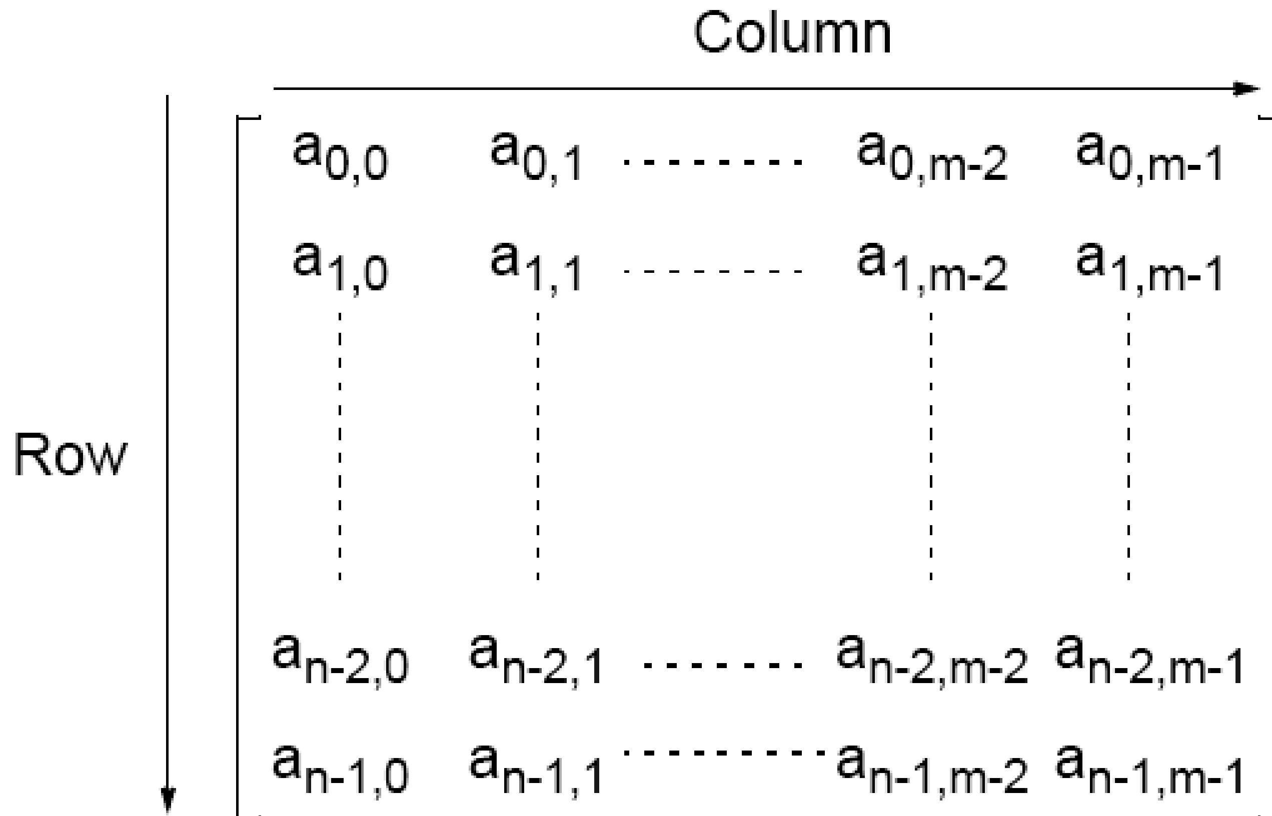
In textbook do:

- Matrix multiplication
- Solving a system of linear equations

@ 2004 Pearson Education Inc. All rights reserved.

Matrices — A Review

An *n x m* matrix



@ 2004 Pearson Education Inc. All rights reserved.

Matrix Addition

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

Given the elements of **A** as $a_{i,j}$ and the elements of **B** as $b_{i,j}$, each element of **C** is computed as

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

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

@ 2004 Pearson Education Inc. All rights reserved.

Matrix Multiplication

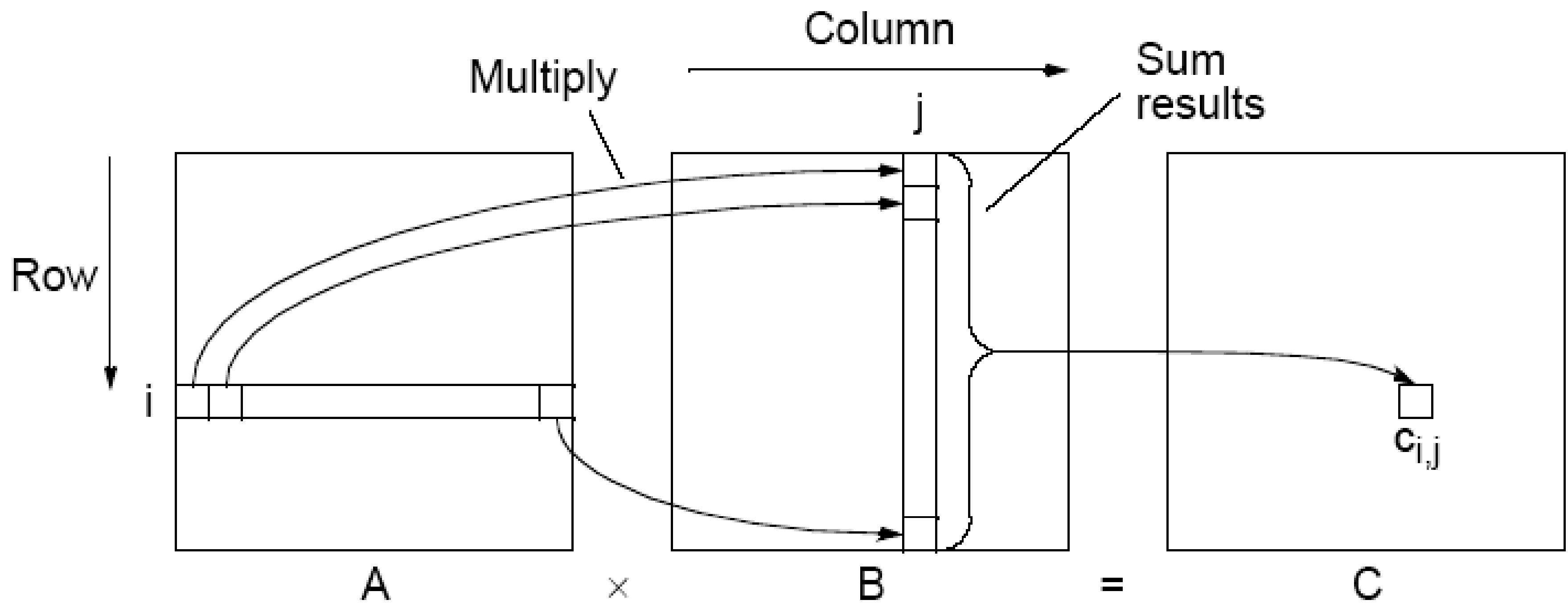
Multiplication of two matrices, **A** and **B**, produces the matrix **C** whose elements, $c_{i,j}$ ($0 <= i < n$, $0 <= j < m$), are computed as follows:

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

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

@ 2004 Pearson Education Inc. All rights reserved.

Matrix multiplication, C = A x B

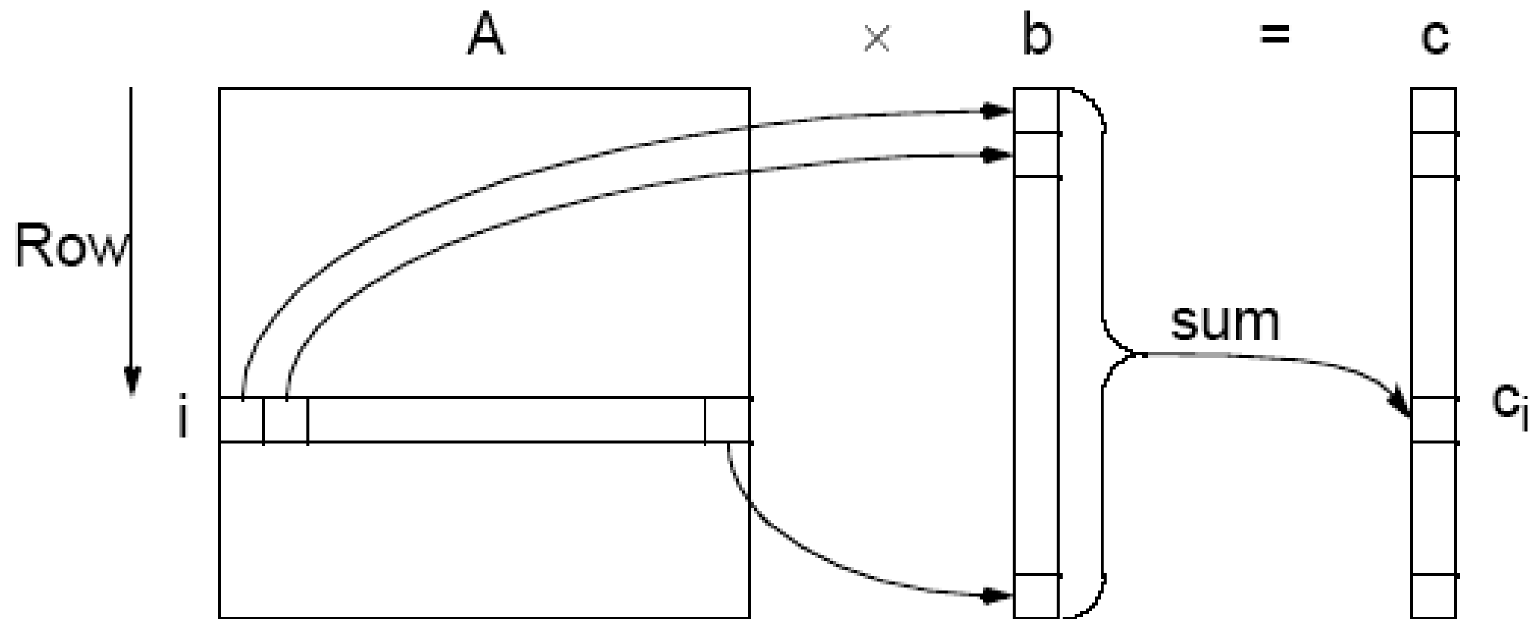


@ 2004 Pearson Education Inc. All rights reserved.

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).



@ 2004 Pearson Education Inc. All rights reserved.

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.

@ 2004 Pearson Education Inc. All rights reserved.

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 n3 multiplications and n3 additions, leading to a sequential time complexity of O(n3).

Very easy to parallelize.

@ 2004 Pearson Education Inc. All rights reserved.

Parallel Code

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

- Time complexity of O(n2) with n processors
  Each instance of inner loop independent and can be done by a separate processor
- Time complexity of O(n) with n2 processors
  One element of A and B assigned to each processor.
  Cost optimal since O(n3) = n x O(n2) = n2 x O(n)].
- Time complexity of O(log n) with n3 processors
  By parallelizing the inner loop. Not cost-optimal since
  O(n3) $\neq$ n3xO(log n)).

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

@ 2004 Pearson Education Inc. All rights reserved.

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*/
}
```

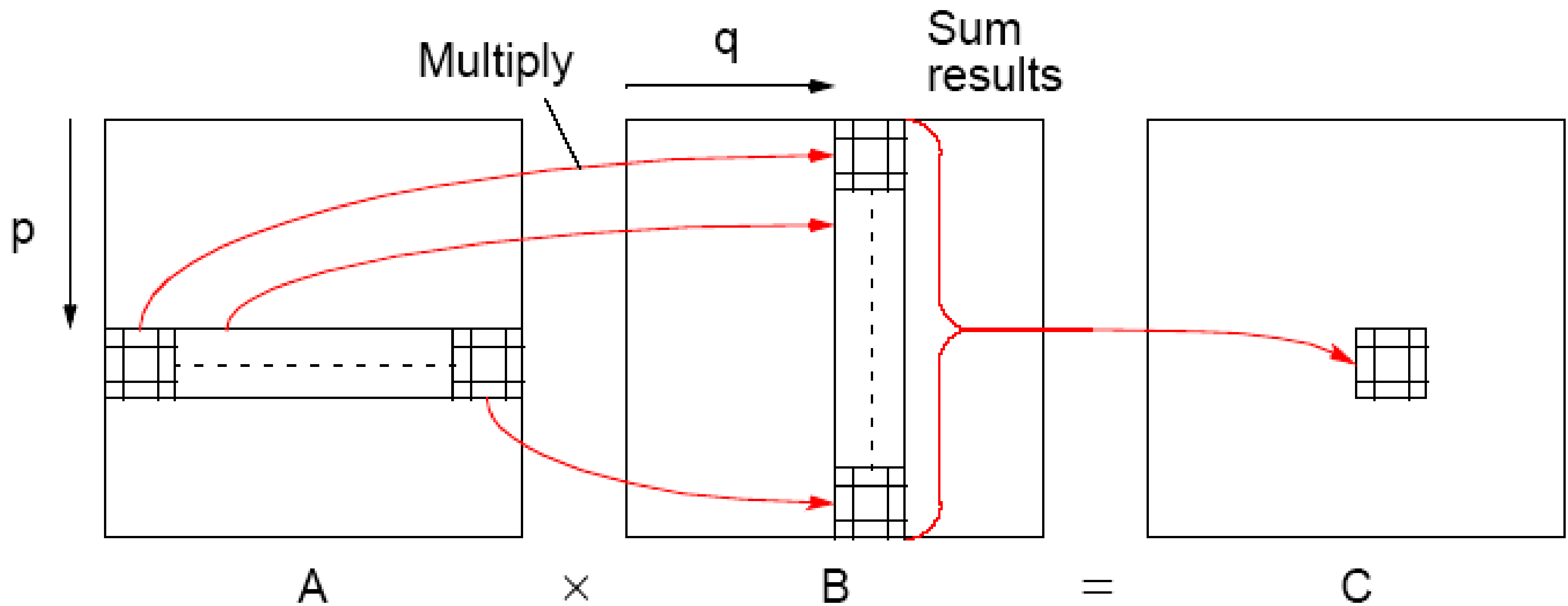
The line

```
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*.

@ 2004 Pearson Education Inc. All rights reserved.

Block Matrix Multiplication



@ 2004 Pearson Education Inc. All rights reserved.

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 $C_{0,0}$

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

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

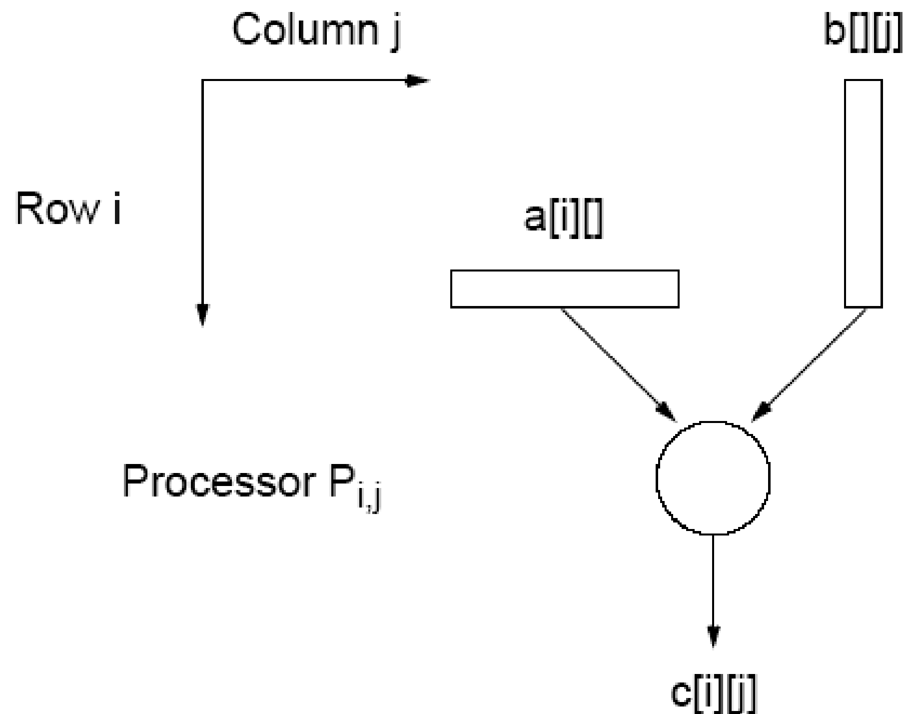
$$= \begin{bmatrix} a_{0,0}b_{0,0}+a_{0,1}b_{1,0}+a_{0,2}b_{2,0}+a_{0,3}b_{3,0} & a_{0,0}b_{0,1}+a_{0,1}b_{1,1}+a_{0,2}b_{2,1}+a_{0,3}b_{3,1} \\ a_{1,0}b_{0,0}+a_{1,1}b_{1,0}+a_{1,2}b_{2,0}+a_{1,3}b_{3,0} & a_{1,0}b_{0,1}+a_{1,1}b_{1,1}+a_{1,2}b_{2,1}+a_{1,3}b_{3,1} \end{bmatrix}$$

$$= C_{0,0}$$

@ 2004 Pearson Education Inc. All rights reserved.

Direct Implementation

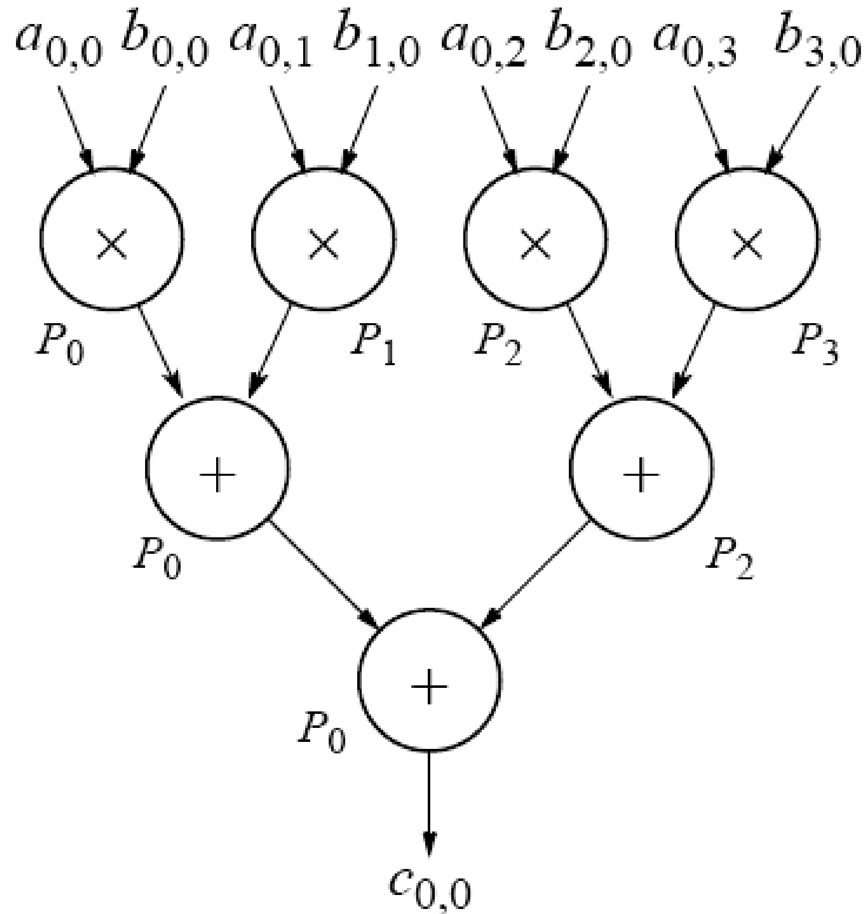
One processor to compute each element of **C** – n^2 processors would be needed. One row of elements of **A** and one column of elements of **B** needed. Some of same elements sent to more than one processor. Can use submatrices.



@ 2004 Pearson Education Inc. All rights reserved.

Performance Improvement

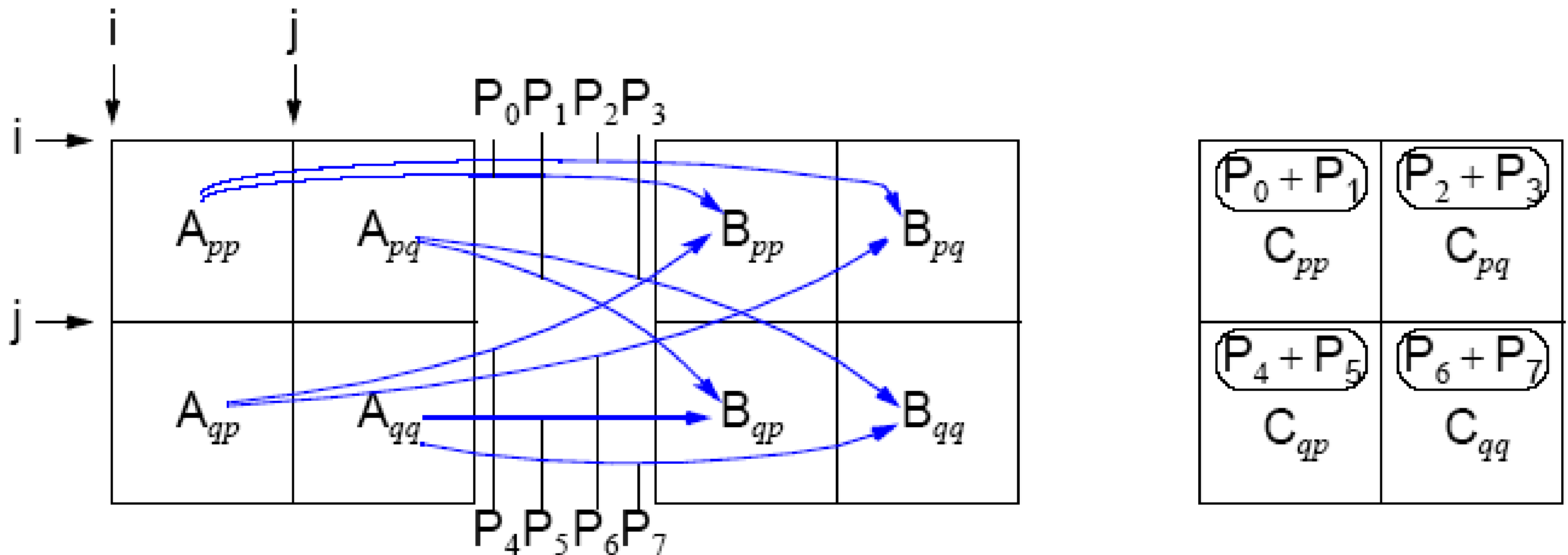
Using tree construction *n* numbers can be added in log *n* steps using *n* processors:



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

@ 2004 Pearson Education Inc. All rights reserved.

Recursive Implementation



Apply same algorithm on each submatrix recursivly.

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

@ 2004 Pearson Education Inc. All rights reserved.

Recursive Algorithm

```
mat_mult(A_pp, B_pp, 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(A_pp, B_pp, s);
   P1 = mat_mult(A_pq, B_qp, s);
   P2 = mat_mult(A_pp, B_pq, s);
   P3 = mat_mult(A_pq, B_qq, s);
   P4 = mat_mult(A_qp, B_pp, s);
   P5 = mat_mult(A_qq, B_qp, s);
   P6 = mat_mult(A_qp, B_pq, s);
   P7 = mat_mult(A_qq, B_qq, s);
   C_pp = P0 + P1;         /* add submatrix products to */
   C_pq = P2 + P3;         /* form submatrices of final matrix */
   C_qp = P4 + P5;
   C_qq = P6 + P7;
}
return (C);                /* return final matrix */
}
```


@ 2004 Pearson Education Inc. All rights reserved.

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.

@ 2004 Pearson Education Inc. All rights reserved.

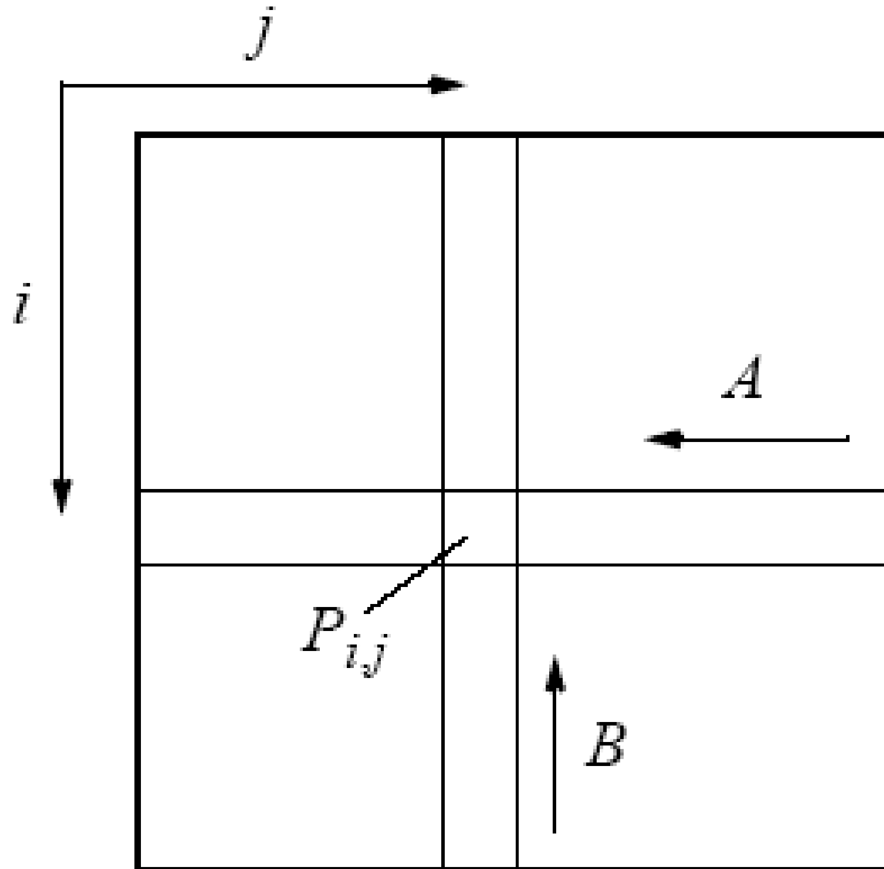
Mesh Implementations
Cannon's Algorithm

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 $P_{i,j}$ has elements $a_{i,j}$ and $b_{i,j}$ ($0 <= i < n$, $0 <= k < n$).
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 $a_{i,j+i}$ and the element $b_{i+j,j}$ in processor $P_{i,j}$,. These elements are a pair of those required in the accumulation of $c_{i,j}$.
3. Each processor, $P_{i,j}$, 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, $P_{i,j}$, 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).

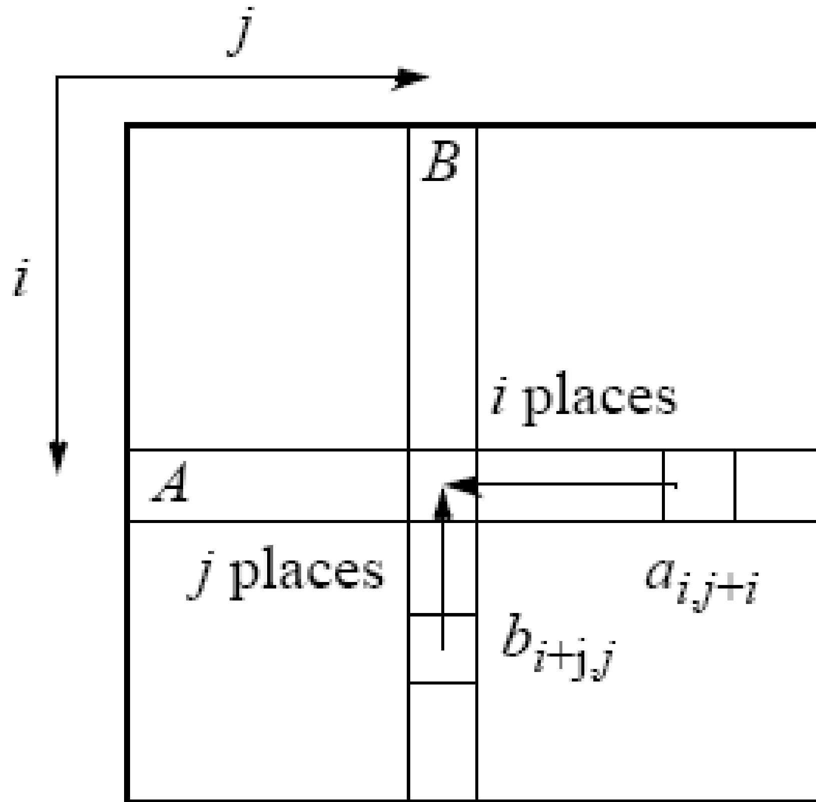
@ 2004 Pearson Education Inc. All rights reserved.

Movement of *A* and *B* elements



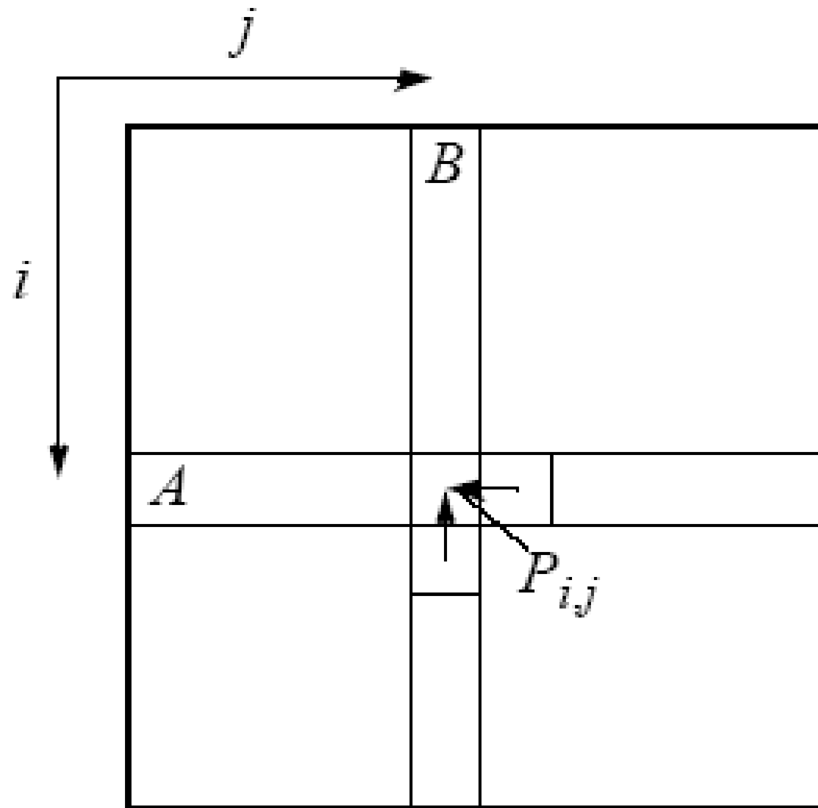
@ 2004 Pearson Education Inc. All rights reserved.

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



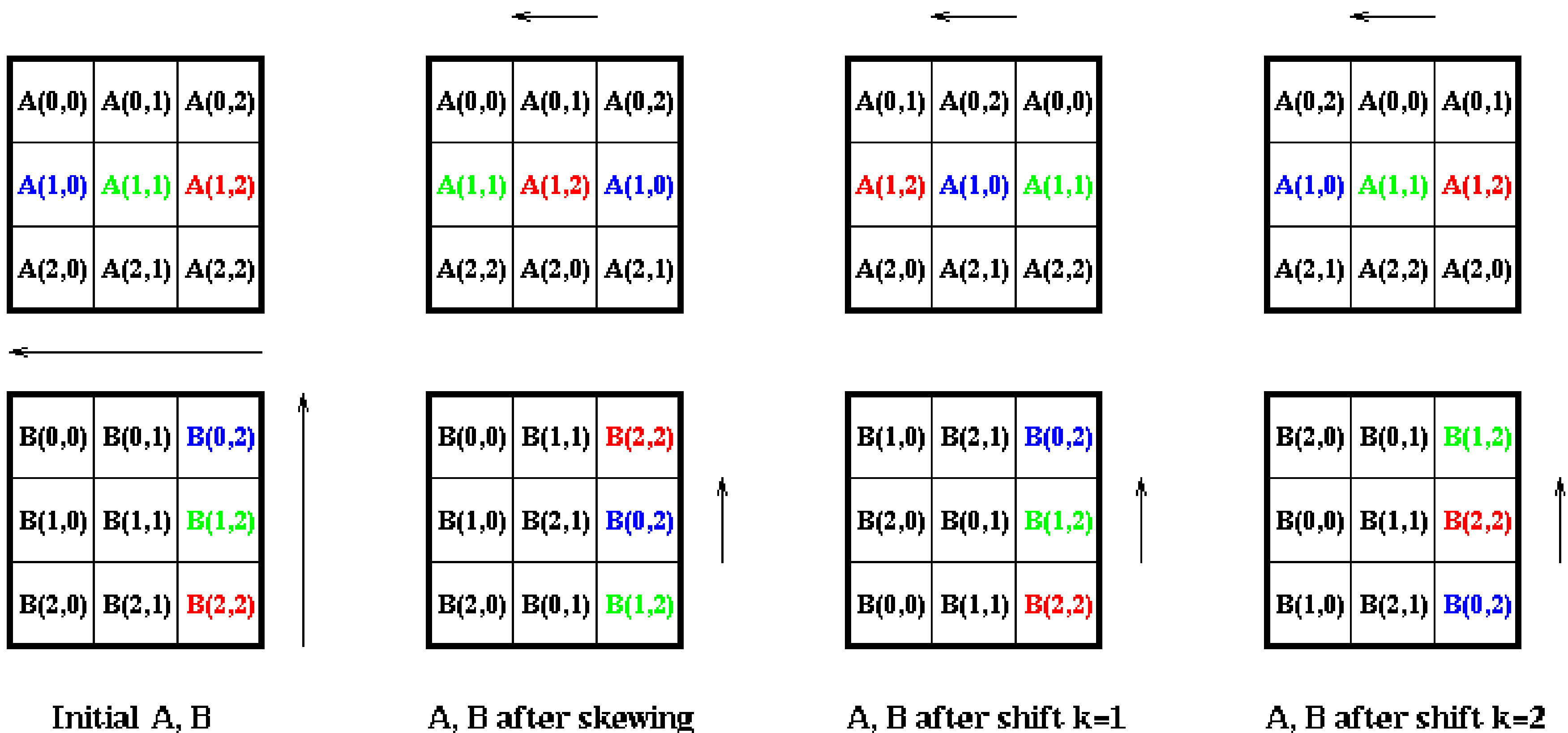
@ 2004 Pearson Education Inc. All rights reserved.

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



@ 2004 Pearson Education Inc. All rights reserved.

Cannon's Matrix Multiplication



C(1,2) = A(1,0) * B(0,2) + A(1,1) * B(1,2) + A(1,2) * B(2,2)

Initial Step to Skew Matrices in Cannon

Initial

A(0,0)	A(0,1)	A(0,2)
A(1,0)	A(1,1)	A(1,2)
A(2,0)	A(2,1)	A(2,2)

B(0,0)	B(0,1)	B(0,2)
B(1,0)	B(1,1)	B(1,2)
B(2,0)	B(2,1)	B(2,2)

After skewing

A(0,0)	A(0,1)	A(0,2)
A(1,1)	A(1,2)	A(1,0)
A(2,2)	A(2,0)	A(2,1)

B(0,0)	B(1,1)	B(2,2)
B(1,0)	B(2,1)	B(0,2)
B(2,0)	B(0,1)	B(1,2)

Shifting Steps in Cannon

- First step

A(0,0)	A(0,1)	A(0,2)
A(1,1)	A(1,2)	A(1,0)
A(2,2)	A(2,0)	A(2,1)

B(0,0)	B(1,1)	B(2,2)
B(1,0)	B(2,1)	B(0,2)
B(2,0)	B(0,1)	B(1,2)

- Second

A(0,1)	A(0,2)	A(0,0)
A(1,2)	A(1,0)	A(1,1)
A(2,0)	A(2,1)	A(2,2)

B(1,0)	B(2,1)	B(0,2)
B(2,0)	B(0,1)	B(1,2)
B(0,0)	B(1,1)	B(2,2)

- Third

A(0,2)	A(0,0)	A(0,1)
A(1,0)	A(1,1)	A(1,2)
A(2,1)	A(2,2)	A(2,0)

B(2,0)	B(0,1)	B(1,2)
B(0,0)	B(1,1)	B(2,2)
B(1,0)	B(2,1)	B(0,2)

Cost of Cannon's Algorithm

```
forall  i=0 to s-1                    … recall s = sqrt(p)
     left-circular-shift row i of A by i     … cost = s*(ts + td*n2/p)
forall  i=0 to s-1
     up-circular-shift column i of B by i … cost = s*(ts + tβ*n2/p)
for k=0 to s-1 …sequential loop
      forall  i=0 to s-1 and j=0 to s-1
           C(i,j) = C(i,j) + A(i,j)*B(i,j)   … cost = 2*(n/s)3 = 2*n3/p3/2
           left-circular-shift each row of A by 1   … cost = ts + td*n2/p
           up-circular-shift each column of B by 1     … cost = ts + td*n2/p
```

(Costs: $s \cdot (t_s + t_d \cdot n^2/p)$; $s \cdot (t_s + t_\beta \cdot n^2/p)$; $2 \cdot (n/s)^3 = 2 \cdot n^3/p^{3/2}$; $t_s + t_d \cdot n^2/p$; $t_s + t_d \cdot n^2/p$)

- Total Time = $2 \cdot n^3/p + 4 \cdot s \cdot t_s + 4 \cdot t_d \cdot n^2/s$
- Parallel Efficiency = $2 \cdot n^3 / (p \cdot \text{Total Time})$

  $= 1/(1 + t_s \cdot 2 \cdot (s/n)^3 + t_d \cdot 2 \cdot (s/n))$

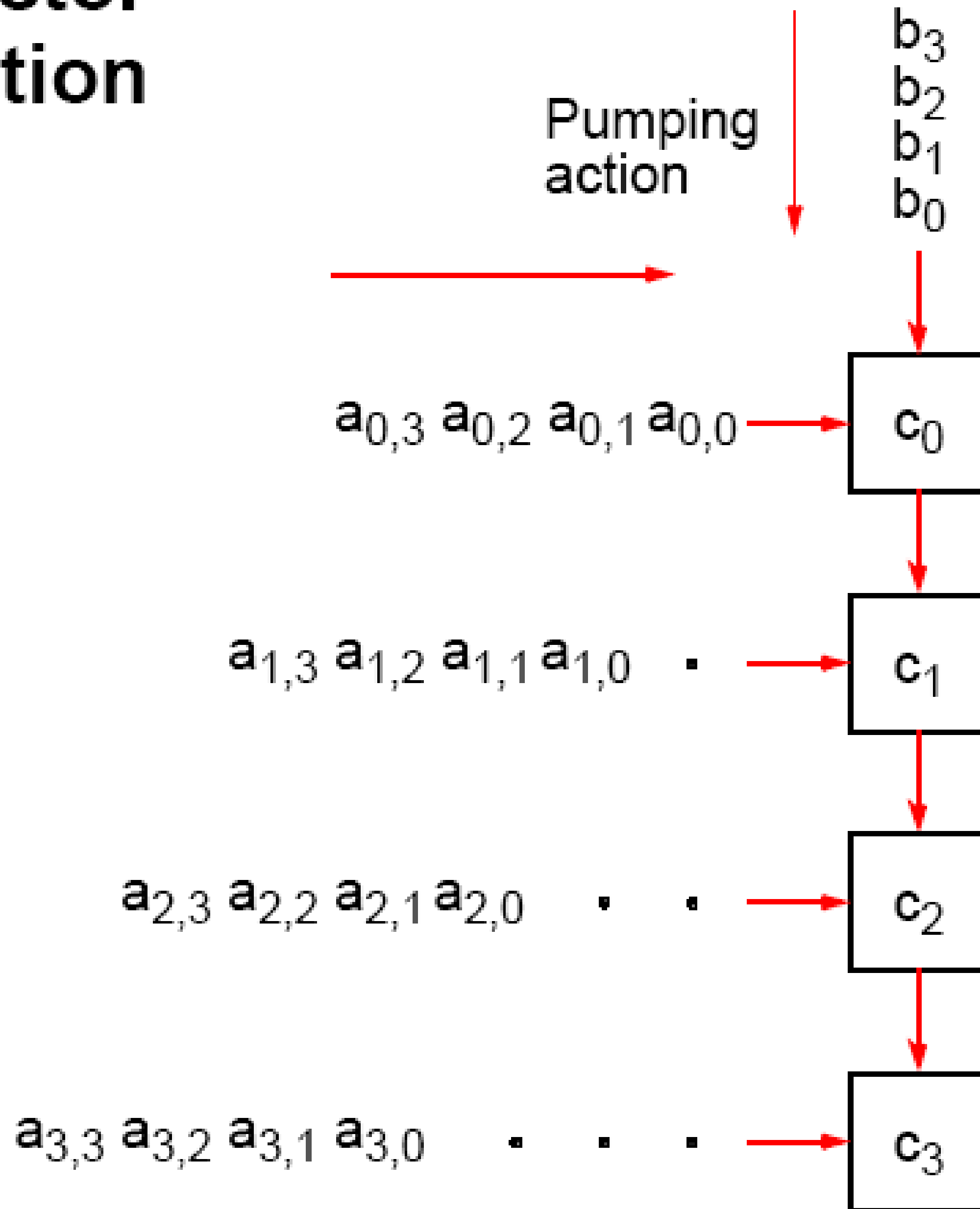
  $= 1/(1 + O(\mathrm{sqrt}(p)/n))$

- Grows to 1 as n/s = n/sqrt(p) = sqrt(data per processor) grows
- Better than 1D layout, which had Efficiency = 1/(1 + O(p/n))

Matrix-Vector Multiplication

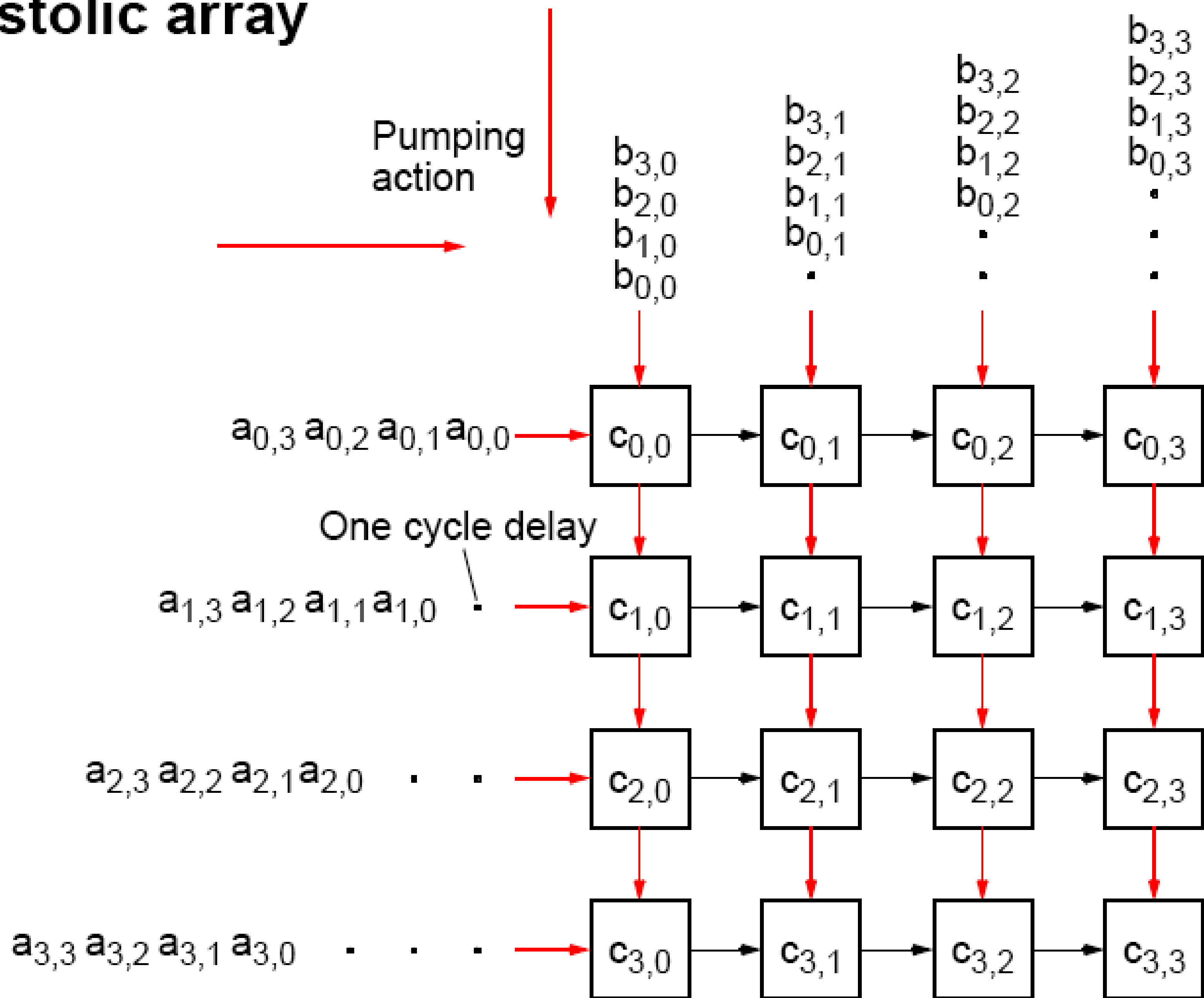
Systolic Algorithms Kung et al.

$c = A \times b$



© 2004 Pearson Education Inc. All rights reserved.

Systolic array



@ 2004 Pearson Education Inc. All rights reserved.

Solving a System of Linear Equations

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

which, in matrix form, is

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

Objective is to find values for the unknowns, x_0, x_1, …, x_{n-1}, given values for $a_{0,0}$, $a_{0,1}$, …, $a_{n-1,n-1}$, and b_0, …, b_n .

@ 2004 Pearson Education Inc. All rights reserved.

Solving a System of Linear Equations

Dense matrices

Gaussian Elimination - parallel time complexity O(n2)

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

@ 2004 Pearson Education Inc. All rights reserved.

Gaussian Elimination

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

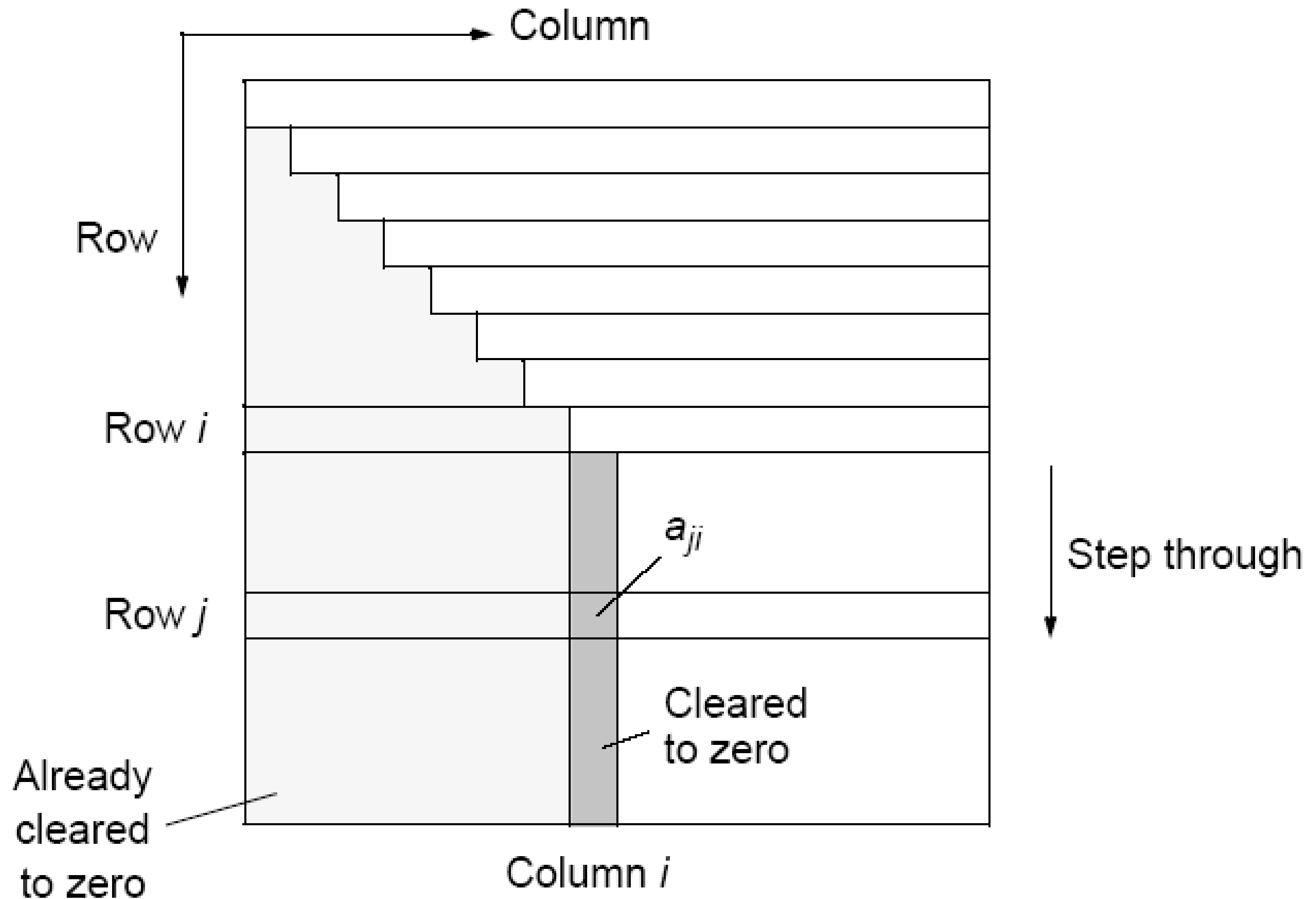
Uses characteristic of linear equations that any row can be replaced by that row added to another row multiplied by a constant.

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*) ($-a_{j,i}/a_{i,i}$). The constant used for row *j* is $-a_{j,i}/a_{i,i}$. Has the effect of making all the elements in the *i*th column below the *i*th row zero because

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

@ 2004 Pearson Education Inc. All rights reserved.

Gaussian elimination



@ 2004 Pearson Education Inc. All rights reserved.

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 ith row with the row below it that has the largest absolute element in the ith column of any of the rows below the ith row if there is one. (Reordering equations will not affect the system.)

In the following, we will not consider partial pivoting.

@ 2004 Pearson Education Inc. All rights reserved.

Sequential Code

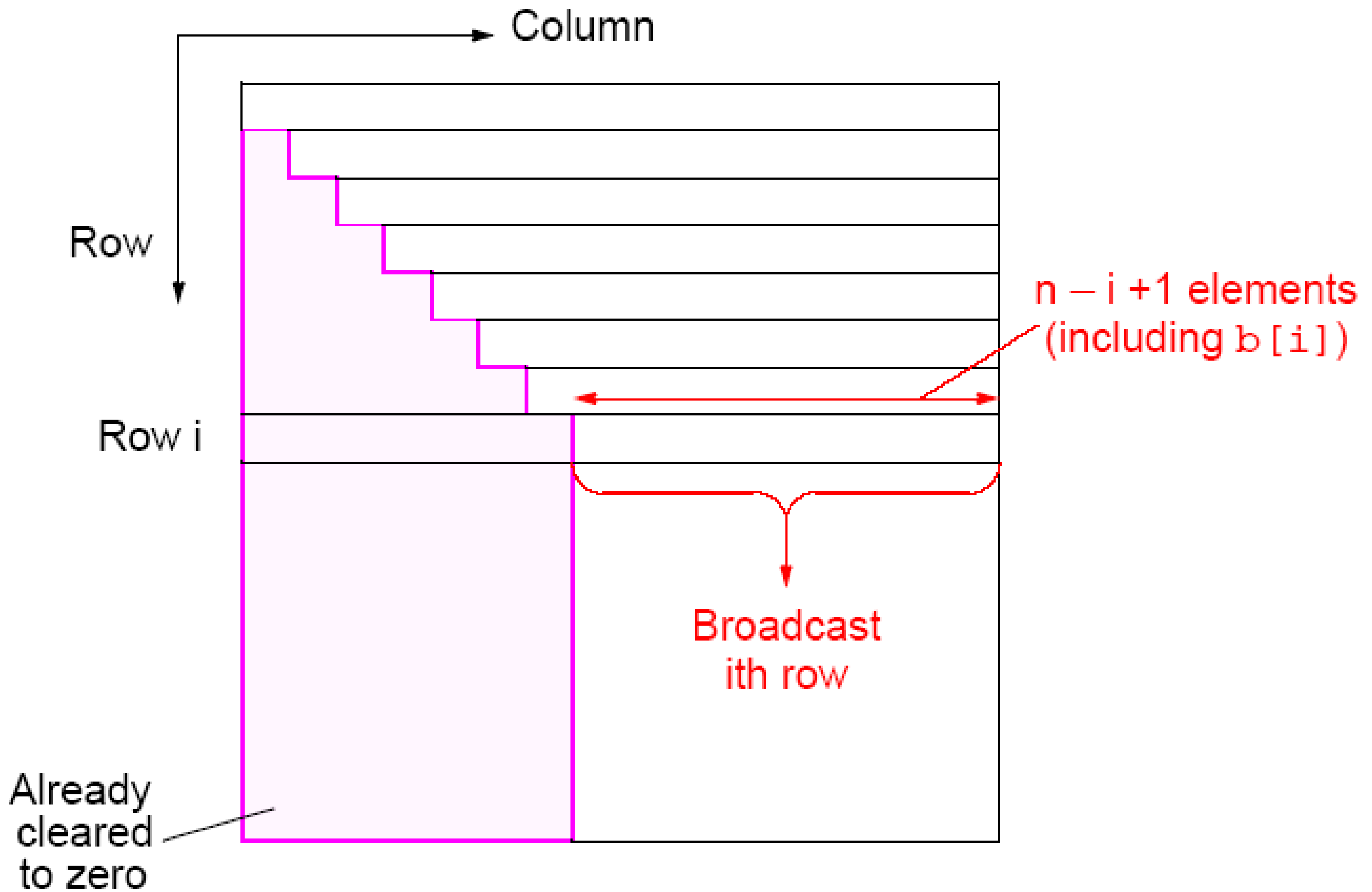
Without partial pivoting:

```
   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).

@ 2004 Pearson Education Inc. All rights reserved.

Parallel Implementation



@ 2004 Pearson Education Inc. All rights reserved.

Analysis

Communication

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

Time complexity of $O(n^2)$

Suppose Broadcast can be done in one step. There are (n-1) broadcasts. ith broadcast message contains n-i+1 elements

Hence total communication given by

$$t_{comm} = \sum_{i=0}^{n-2} (t_s + (n - i + 1)t_{d)} = ((n-1)t_s + (\frac{(n+2)(n+1)}{3} - 3)t_d)$$

More realistically assuming broadcast is log(p)

$$t_{comm} = \log(\mathrm{p})((n-1)t_s + (\frac{(n+2)(n+1)}{3} - 3)t_d)$$

@ 2004 Pearson Education Inc. All rights reserved.

Analysis

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.

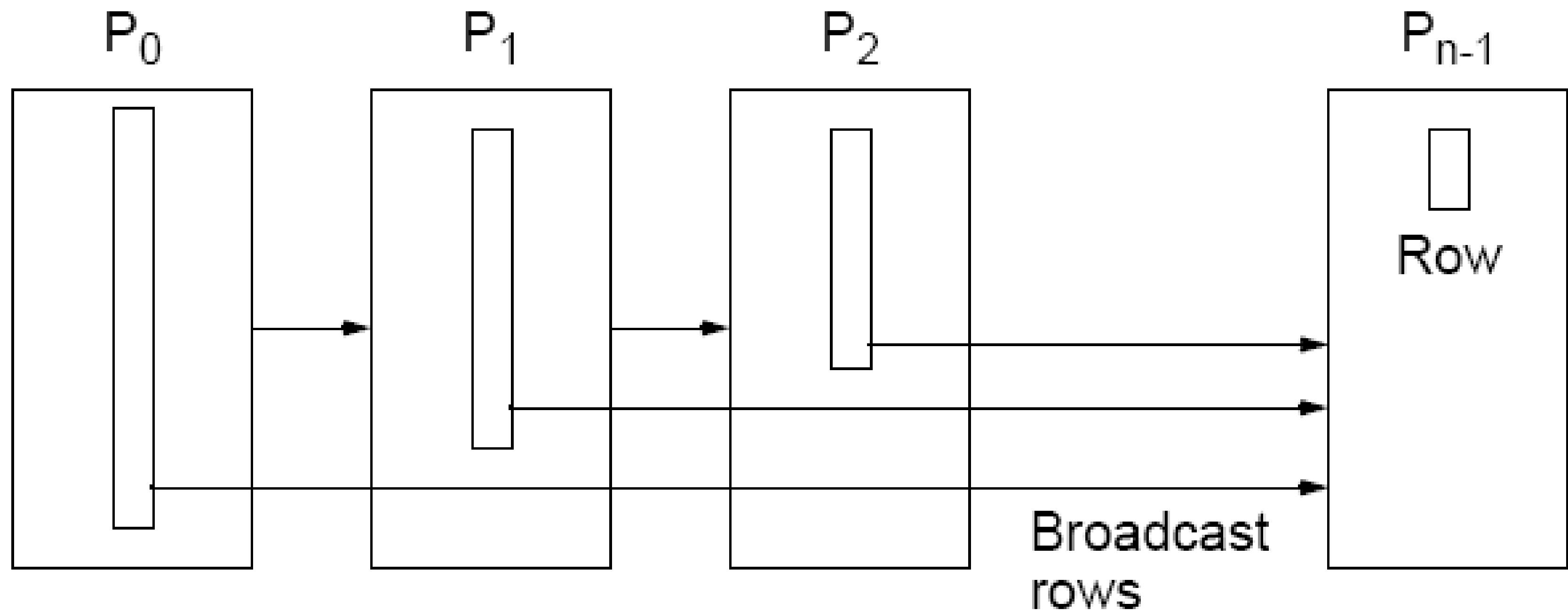
Time complexity of $O(n^2)$

$$t_{comp} = 2\sum_{j=1}^{n-1}(n - j + 2) = \frac{(n+2)(n+3)}{2}3 = O(n^2)$$

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

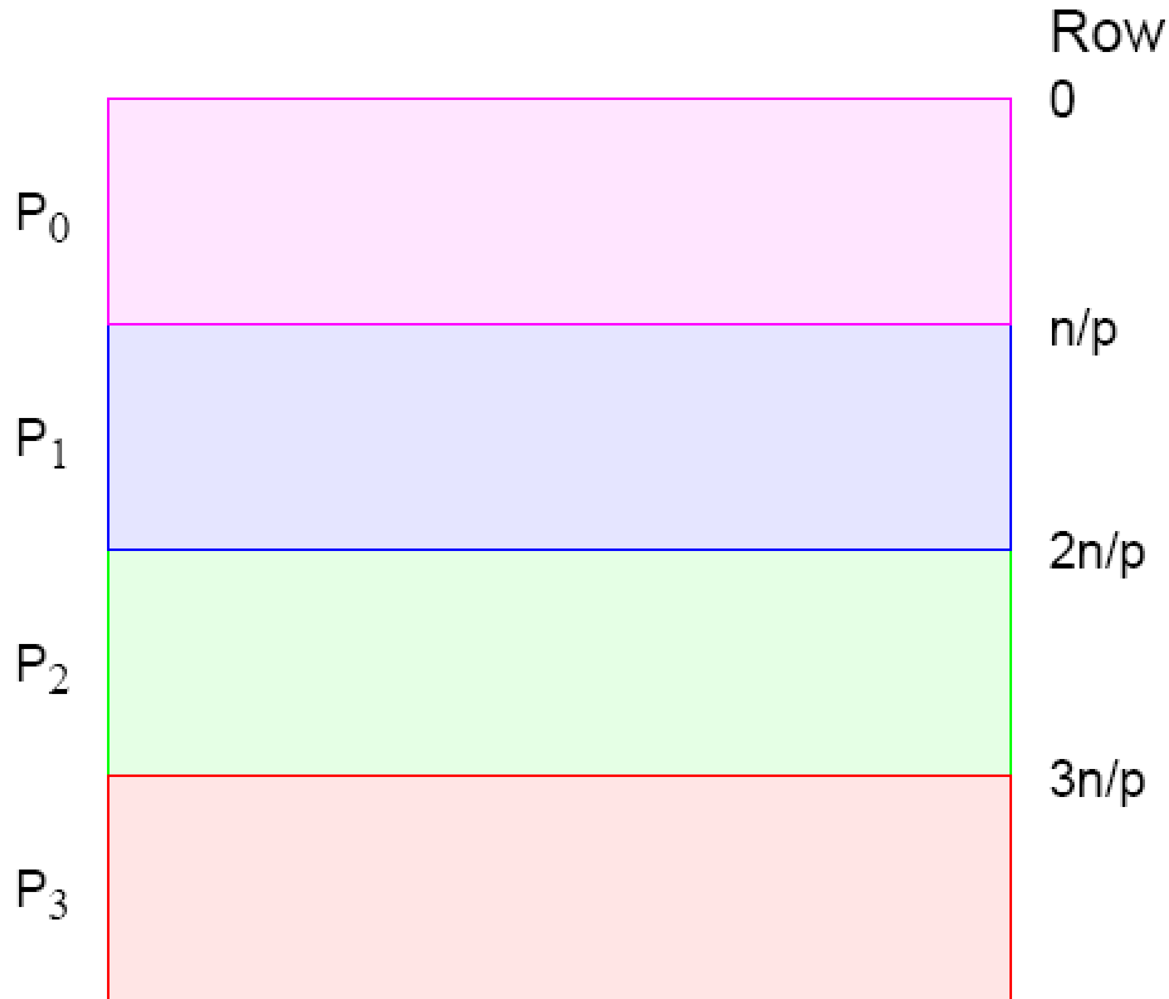
@ 2004 Pearson Education Inc. All rights reserved.

Pipeline implementation of Gaussian elimination



@ 2004 Pearson Education Inc. All rights reserved.

Strip Partitioning

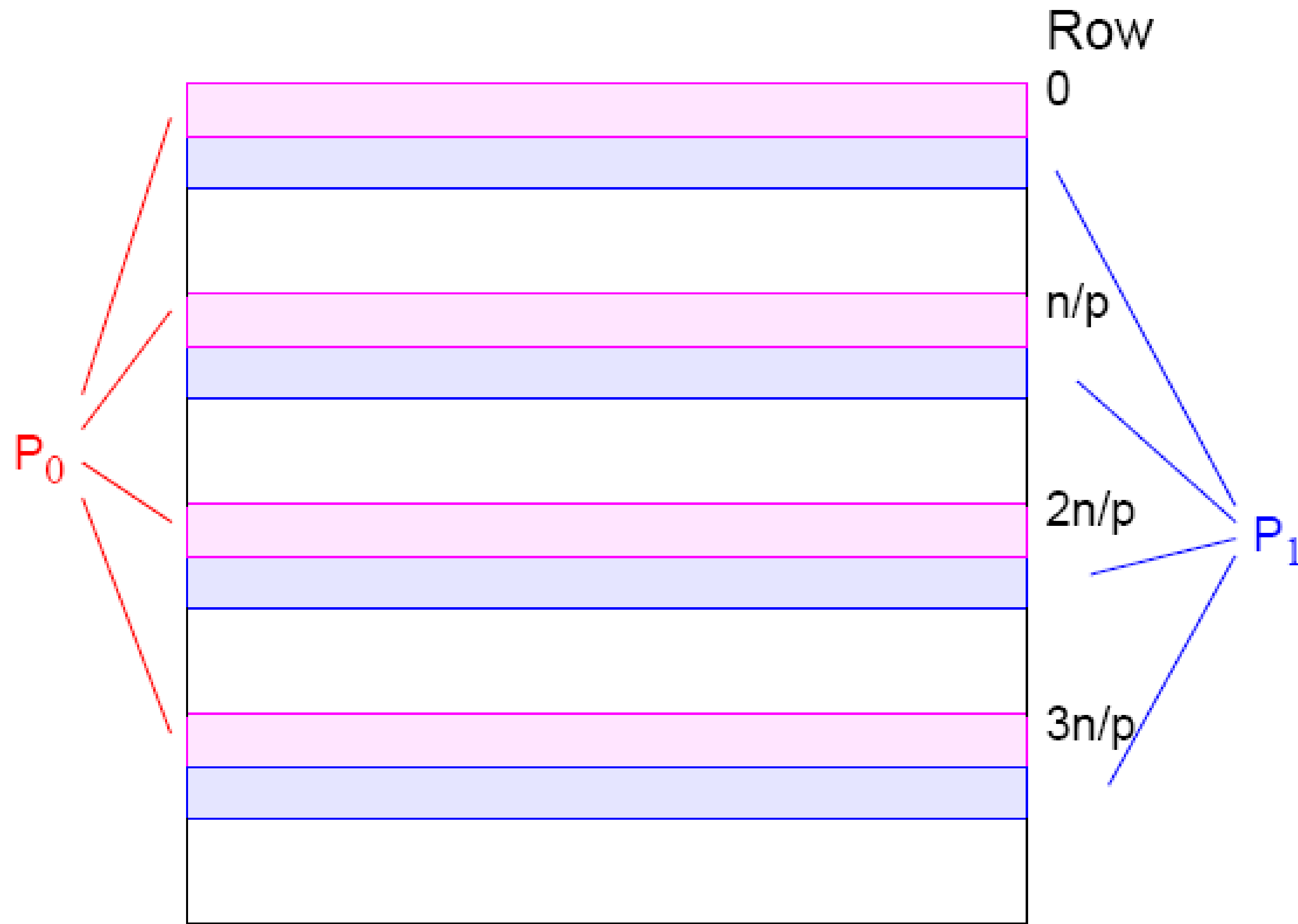


Poor processor allocation! Processors do not participate in computation after their last row is processed.

@ 2004 Pearson Education Inc. All rights reserved.

Cyclic-Striped Partitioning

An alternative which equalizes the processor workload:



@ 2004 Pearson Education Inc. All rights reserved.

Analysis of Strip Case

Serial Case LU decomposition costs $\frac{2}{3}n^3 + \frac{n^2}{2} + \frac{n}{6}$

Assume that strip approach parallelizes this on p cores to get a cost of $\frac{1}{p}(\frac{2}{3}n^3 + \frac{n^2}{2} + \frac{n}{6})$

Efficiency is then
$$\frac{1}{1 + \dfrac{\log(\text{p})((n-1)t_s + (\frac{(n+2)(n+1)}{3} - 3)t_d)}{2n^3/3 + n^2/2 + n/6}}$$

Or
$$\frac{1}{1 + \dfrac{3\log(\text{p})(\text{t}_s + n\ t_d)}{2n^2}}$$

Hence need $\dfrac{3\log(\text{p})(\text{t}_s + n\ t_d)}{2n^2}$ constant

@ 2004 Pearson Education Inc. All rights reserved.

Relationship Between LU Decomposition and Gaussian Elimination

There many different ways to decompose the matrix A.
A common one is U=Gaussian eliminated matrix
L=Multipliers used for elimination

$$A = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ m_{2,1} & 1 & 0 & \cdots & 0 & 0 \\ m_{3,1} & m_{3,2} & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots & 0 \\ m_{n-1,1} & m_{n-1,2} & m_{n-1,3} & \cdots & 1 & \vdots \\ m_{n,1} & m_{n,2} & m_{n,3} & m_{n,4} & \cdots & 1 \end{bmatrix} \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2n}^{(2)} \\ 0 & 0 & a_{33}^{(3)} & \cdots & a_{3n}^{(3)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & a_{n-1n-1}^{(n)} & a_{n-1n}^{(n)} \\ 0 & 0 & 0 & 0 & a_{nn}^{(n)} \end{bmatrix}$$

As the diagonal entries of L matrix are all 1′s they are not stored,
Also LU is stored in a single matrix.

Other Direct Linear Solver Algorithms

Use pipelined approach and overlap communication and computatIon, see book by Quinn.

Use block partition of large matrix so each core has only a block.

Many other approache in use.

However – large dense linear systems not often solved apart from Top500 list . Or if they are advantage is taken of special from e.g. fast multipole method.

@ 2004 Pearson Education Inc. All rights reserved.

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.

@ 2004 Pearson Education Inc. All rights reserved.

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

@ 2004 Pearson Education Inc. All rights reserved.

Example of a Sparse System of Linear Equations

Laplace's Equation

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

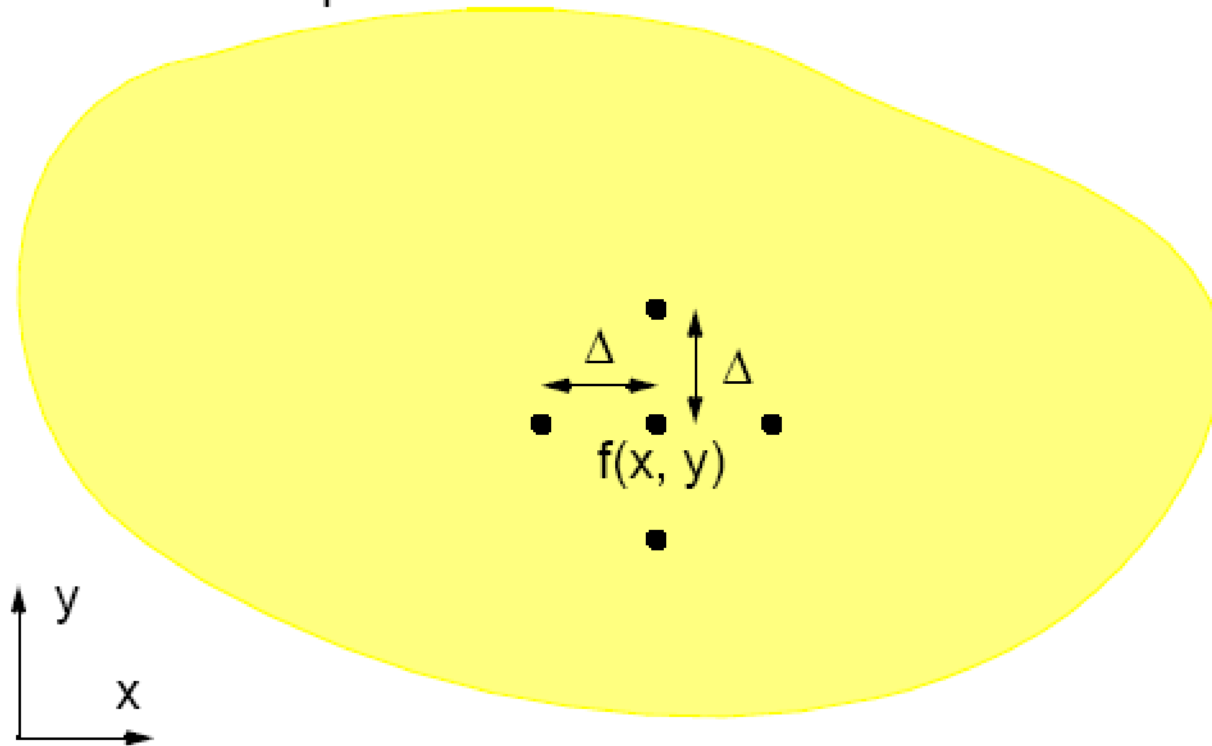
Solve for *f* over the two-dimensional x-y space.

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

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

@ 2004 Pearson Education Inc. All rights reserved.

Finite Difference Method



@ 2004 Pearson Education Inc. All rights reserved.

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

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

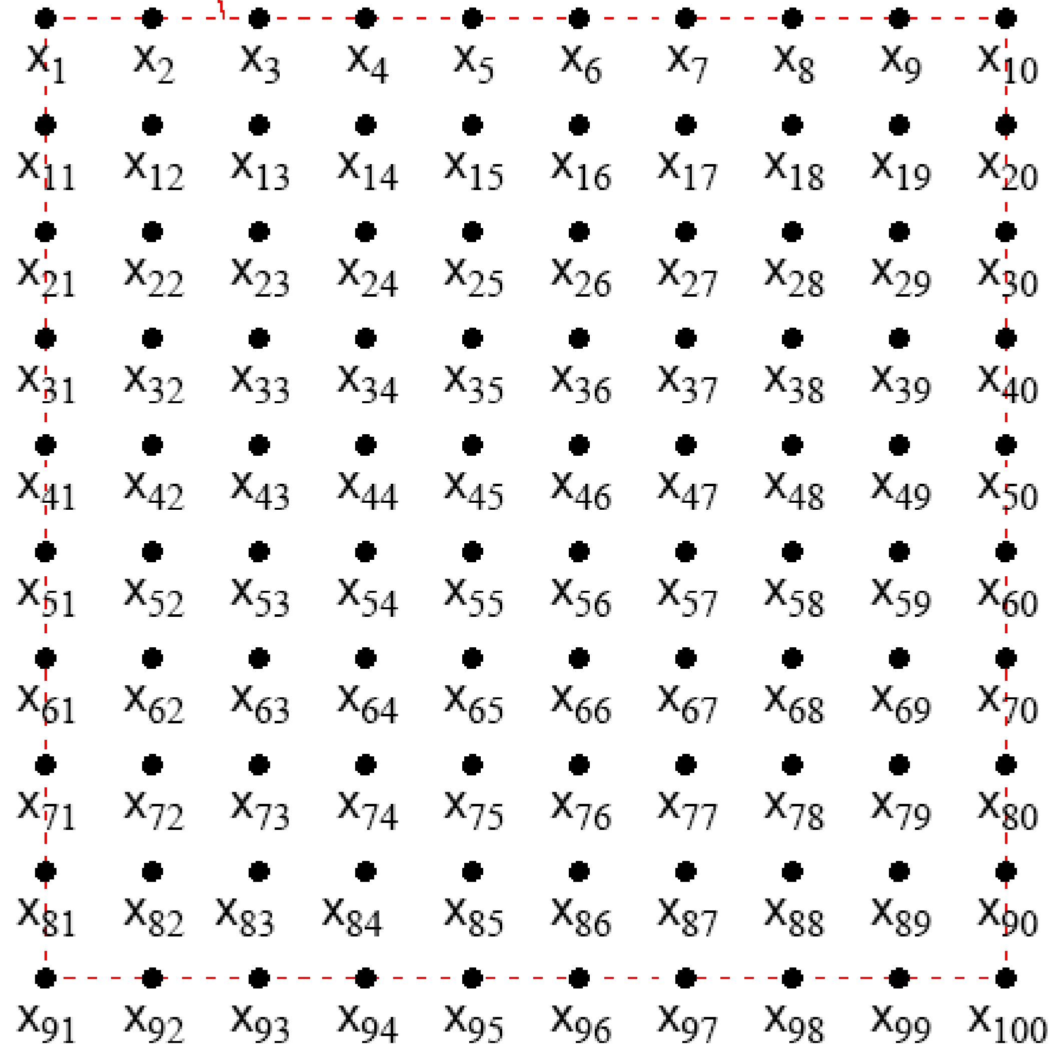
Rewritten as an iterative formula:

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

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

@ 2004 Pearson Education Inc. All rights reserved.

Natural Order



@ 2004 Pearson Education Inc. All rights reserved.

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

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

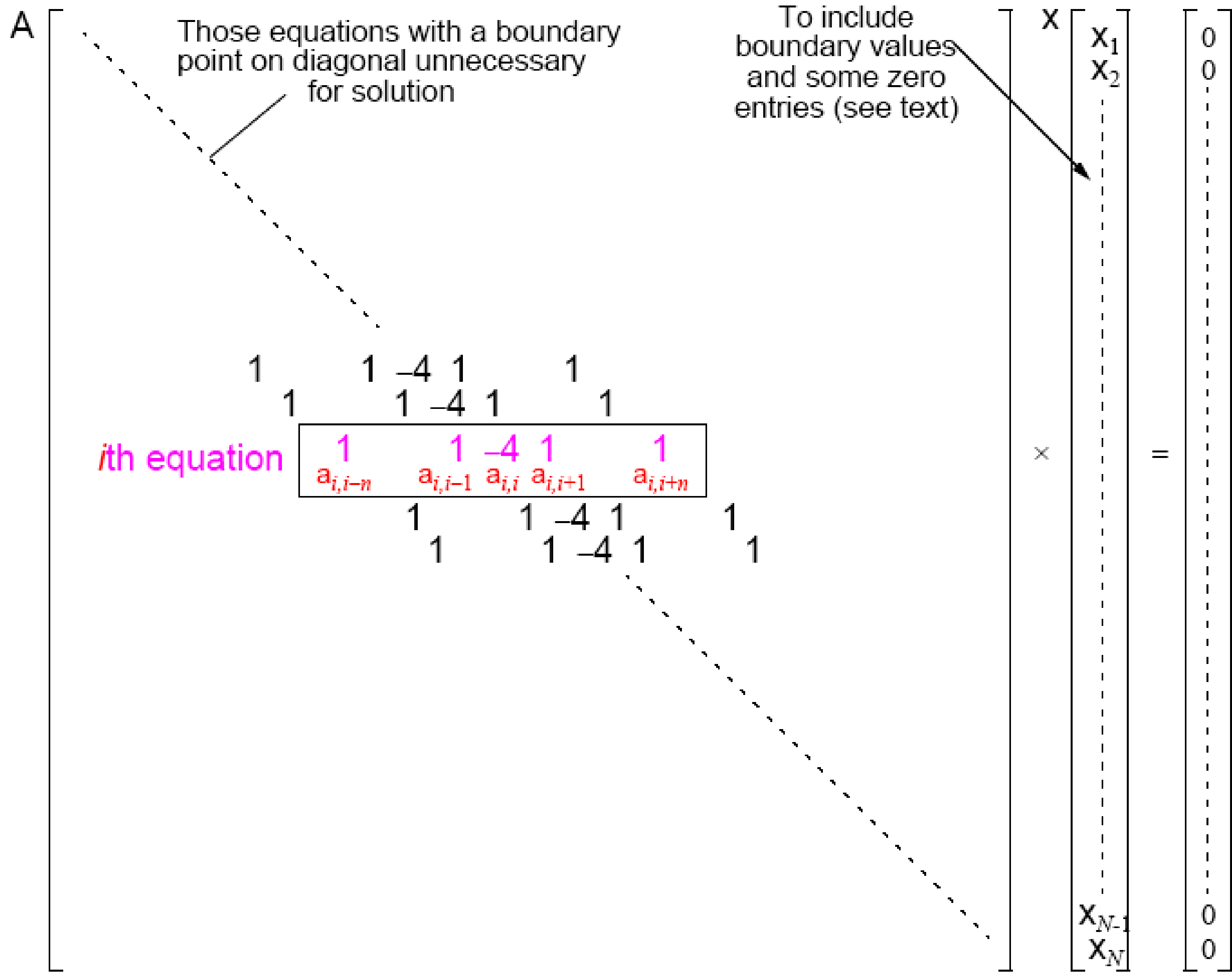
which is a linear equation with five unknowns (except those with boundary points).

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

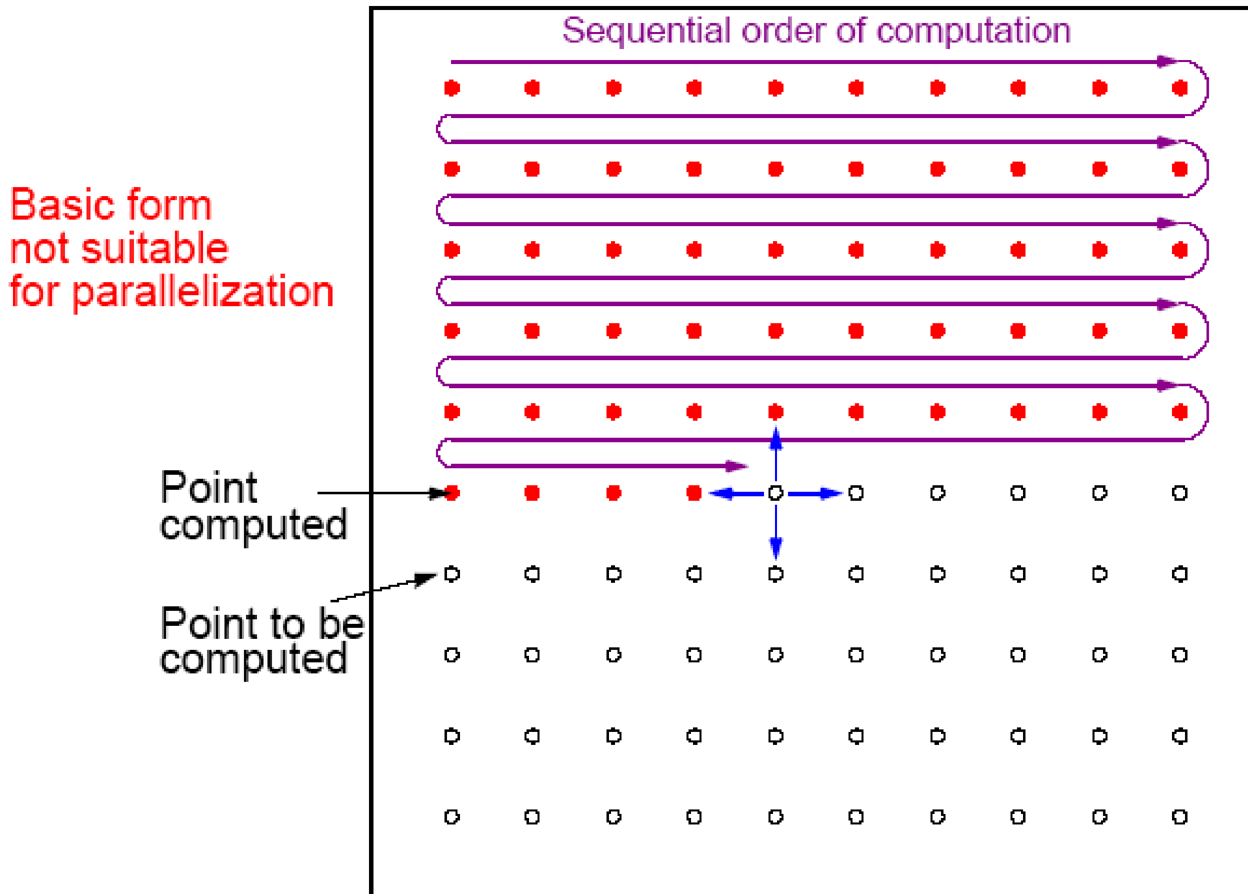
@ 2004 Pearson Education Inc. All rights reserved.



@ 2004 Pearson Education Inc. All rights reserved.

Gauss-Seidel Relaxation

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



@ 2004 Pearson Education Inc. All rights reserved.

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

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

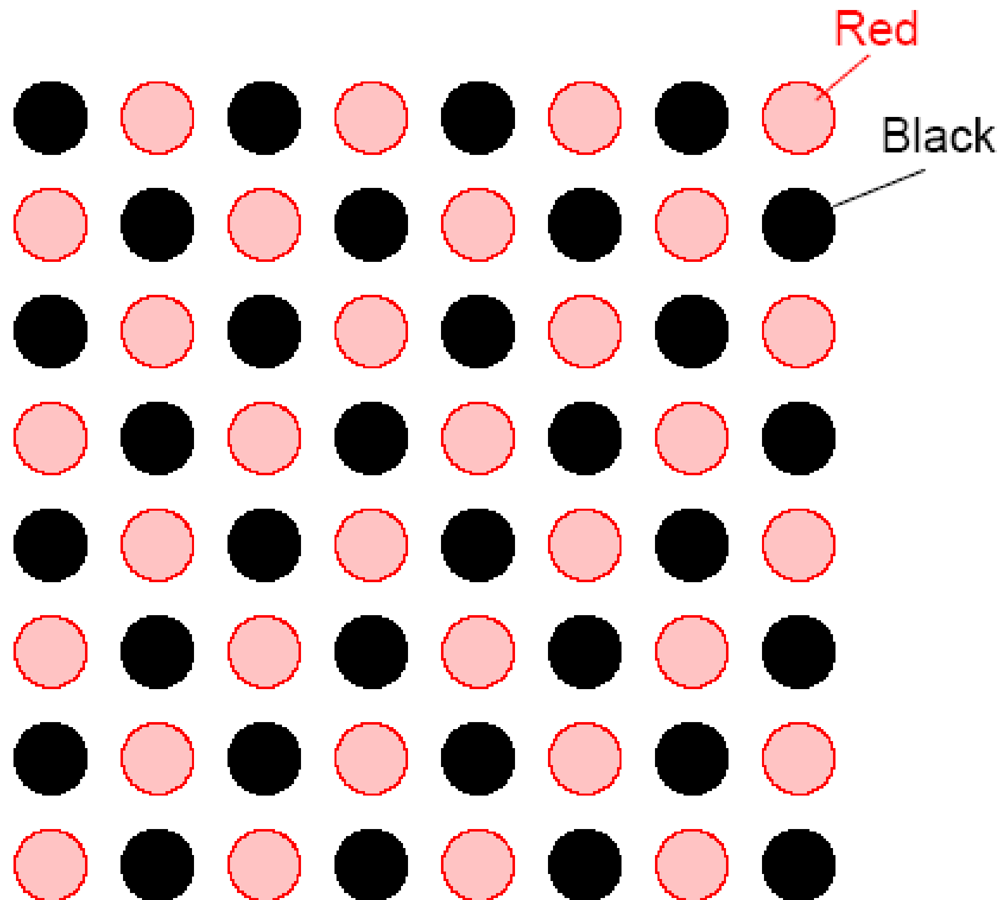
At the *k*th iteration, two of the four values (before the *i*th element) taken from the *k*th iteration and two values (after the *i*th element) taken from the (*k*-1)th iteration. We have:

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

@ 2004 Pearson Education Inc. All rights reserved.

Red-Black Ordering

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



@ 2004 Pearson Education Inc. All rights reserved.

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]);
```


@ 2004 Pearson Education Inc. All rights reserved.

Higher-Order Difference Methods

More distant points could be used in the computation. The following update formula:

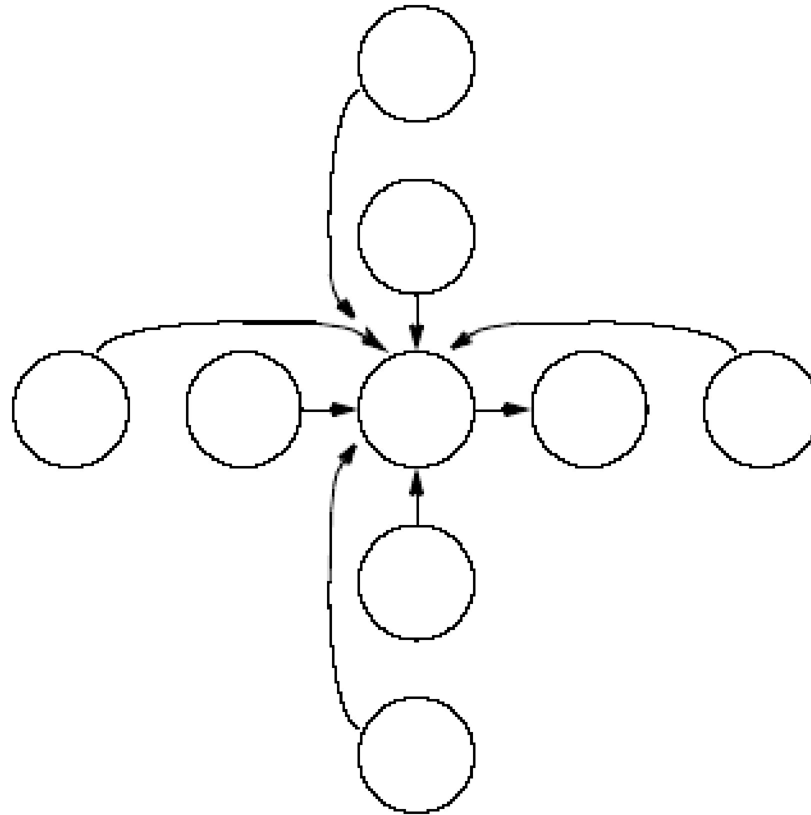
$$f^{k}(x, y) =$$

$$\frac{1}{60}\Big[16f^{k-1}(x-\Delta, y) + 16f^{k-1}(x, y-\Delta) + 16f^{k-1}(x+\Delta, y) + 16f^{k-1}(x, y+\Delta)$$

$$- f^{k-1}(x-2\Delta, y) - f^{k-1}(x, y-2\Delta) - f^{k-1}(x+2\Delta, y) - f^{k-1}(x, y+2\Delta)\Big]$$

@ 2004 Pearson Education Inc. All rights reserved.

Nine-point stencil



@ 2004 Pearson Education Inc. All rights reserved.

Overrelaxation

Improved convergence obtained by adding factor $(1 - \omega)x_i$ to Jacobi or Gauss-Seidel formulae. Factor ω is the *overrelaxation parameter*.

Jacobi overrelaxation formula

$$x_i^k = \frac{\omega}{a_{ii}}\left[b_i - \sum_{j \neq i} a_{ij} x_i^{k-1}\right] + (1 - \omega)x_i^{k-1}$$

where $0 < \omega < 1$.

Gauss-Seidel successive overrelaxation

$$x_i^k = \frac{\omega}{a_{ii}}\left[b_i - \sum_{j=1}^{i-1} a_{ij} x_i^k - \sum_{j=i+1}^{N} a_{ij} x_i^{k-1}\right] + (1 - \omega)x_i^{k-1}$$

where $0 < \omega \leq 2$. If $\omega = 1$, we obtain the Gauss-Seidel method.

@ 2004 Pearson Education Inc. All rights reserved.

Multigrid Method

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

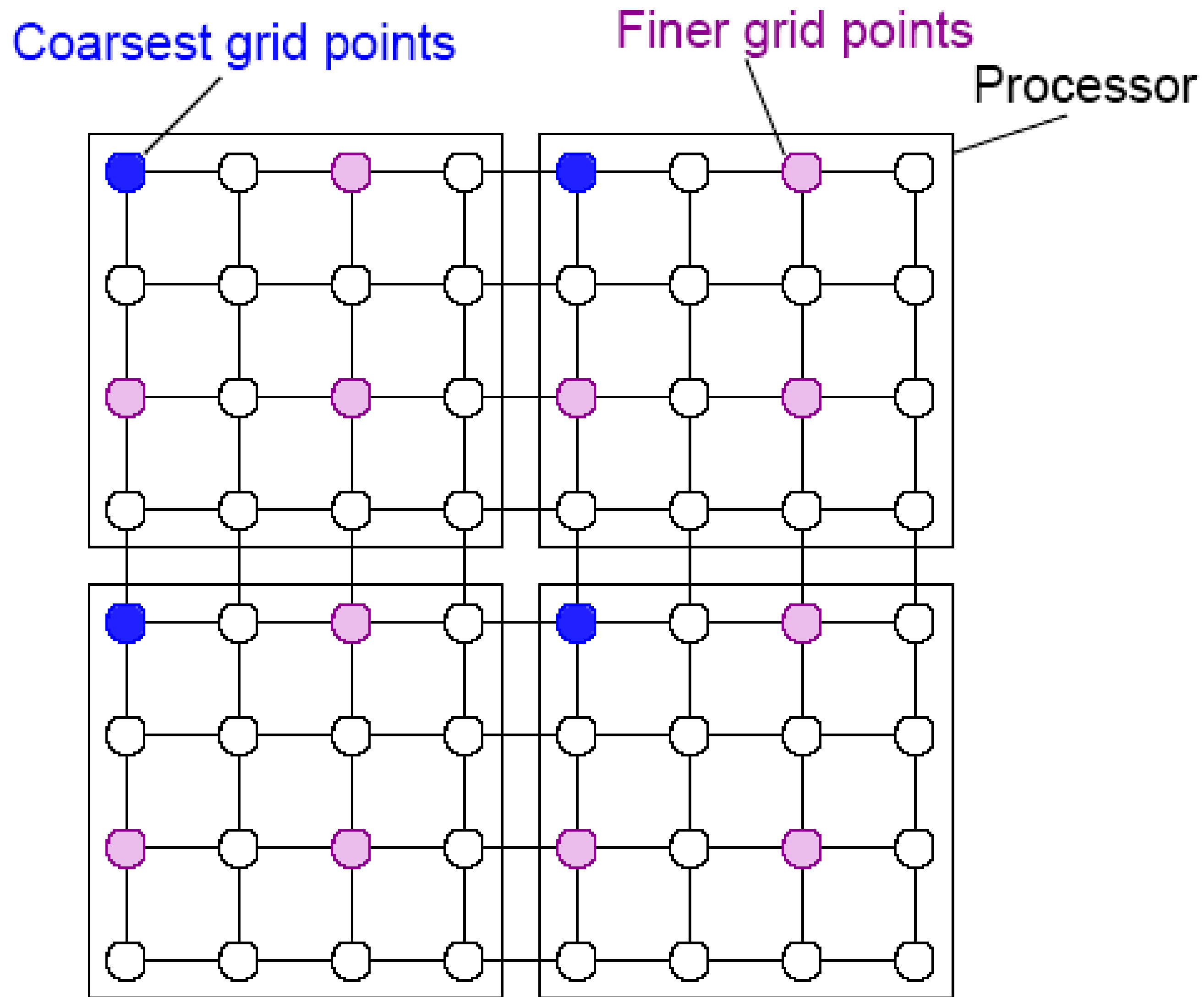
At some stage, number of points increased to include points of the coarse grid and extra points between the points of the coarse grid. Initial values of extra points found by interpolation. Computation continues with this finer grid.

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.

@ 2004 Pearson Education Inc. All rights reserved.

Multigrid processor allocation



@ 2004 Pearson Education Inc. All rights reserved.

(Semi) Asynchronous Iteration

As noted early, synchronizing on every iteration will cause significant overhead - best if one can is to synchronize after a number of iterations.

@ 2004 Pearson Education Inc. All rights reserved.

Conjugate Gradient Method

- *A* is positive definite if for every nonzero vector x and its transpose x^T, the product $x^T A x > 0$
- If *A* is symmetric and positive definite, then the function

$$q(x) = \frac{1}{2} x^T A \quad - x x^T b + c$$

has a unique minimizer that is solution to $Ax = b$

- Conjugate gradient is an iterative method that solves $Ax = b$ by minimizing *q(x)*

Slides for Parallel Programming Techniques & Applications Using Networked Workstations & Parallel Computers 2nd ed., by B. Wilkinson & M. Allen,
@ 2004 Pearson Education Inc. All rights reserved.

Conjugate Gradient Method

Let $r_k = b - A x_k, \quad p_0 = r_0, k = 0$

where x_k is the *k th approximation to the true solution x*

Then do

let $x_{k+1} = x_k + a_k p_k$

and $r_{k+1} = r_{k+1} - a_k p_k$

Where $a_k = ((r_k)^T r_k) / ((p_k)^T A p_k)$

let $b_k = ((r_k)^T r_k) / ((r_{k+1})^T r_{k+1})$ *and*

$$p_{k+1} = r_{k+1} + b_k p_k$$

$$k = k + 1$$

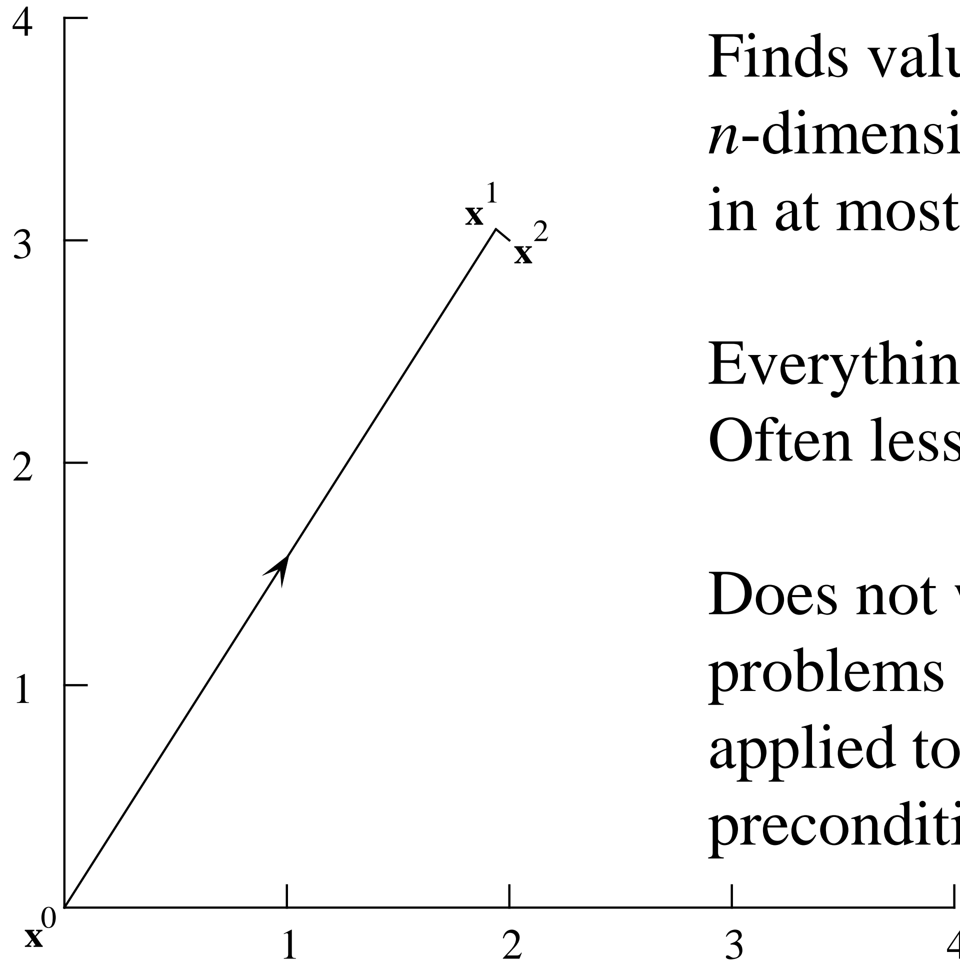
Until r_{k+1} is small enough

Two inner products and one matrix vector product

Slides for Parallel Programming Techniques & Applications Using Networked Workstations & Parallel Computers 2nd ed., by B. Wilkinson & M. Allen,

@ 2004 Pearson Education Inc. All rights reserved.

Conjugate Gradient Convergence



Finds value of n-dimensional solution in at most n iterations

Everything needs to be exact! Often less iterations will do.

Does not work well for all problems and often needs to be applied to a transformed or preconditioned problems

@ 2004 Pearson Education Inc. All rights reserved.

Conjugate Gradient Method

- Matrix-vector multiplication
- Inner product (dot product)
- Matrix-vector multiplication has higher time complexity
- Replicate vectors
  - Need all-gather step after matrix-vector multiply
  - Inner product has time complexity $\Theta(n)$
- Block decomposition of vectors
  - Need all-gather step before matrix-vector multiply
  - Inner product has time complexity $\Theta(n/p + \log p)$
- Care needed with sparse matrices
- Method is and example of a Krylov methods

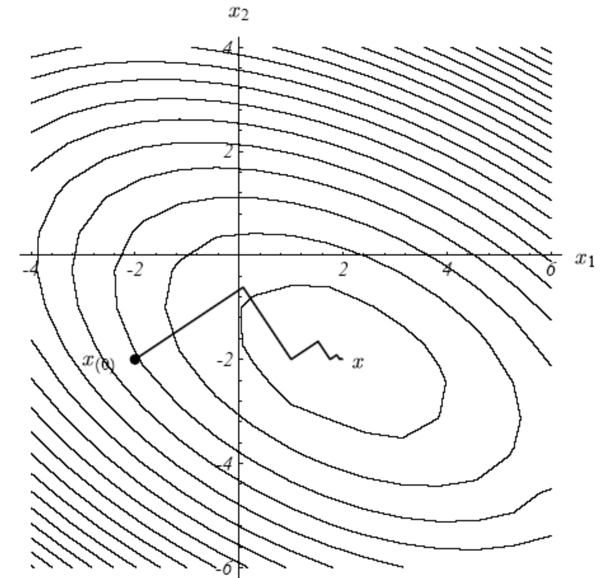
@ 2004 Pearson Education Inc. All rights reserved.

Summary

$$r_{(i)} = b - Ax_{(i)},$$

$$\alpha_{(i)} = \frac{r_{(i)}^T r_{(i)}}{r_{(i)}^T A r_{(i)}},$$

$$x_{(i+1)} = x_{(i)} + \alpha_{(i)} r_{(i)}.$$



b – A$x_{(i+1)}$ = b – A($x_{(i)} + \alpha_{(i)} r_{(i)}$)

Simplification [Ax, Ar]→Ar

$$r_{(i+1)} = r_{(i)} - \alpha_{(i)} A r_{(i)}$$

$$r_{(0)} = b - Ax_{(0)} \qquad r_{(1)} = r_{(0)} - \alpha_{(0)} A r_{(0)} \qquad r_{(2)} = r_{(1)} - \alpha_{(1)} A r_{(1)}$$

$$\alpha_{(0)} = \frac{r_{(0)}^T r_{(0)}}{r_{(0)}^T A r_{(0)}} \qquad \alpha_{(1)} = \frac{r_{(1)}^T r_{(1)}}{r_{(1)}^T A r_{(1)}} \qquad \alpha_{(2)} = \frac{r_{(2)}^T r_{(2)}}{r_{(2)}^T A r_{(2)}} \quad \bullet\bullet\bullet$$

$$x_{(1)} = x_{(0)} + \alpha_{(0)} r_{(0)} \qquad x_{(2)} = x_{(1)} + \alpha_{(1)} r_{(1)} \qquad x_{(3)} = x_{(2)} + \alpha_{(2)} r_{(2)}$$

Preconditioned Conjugate Gradients

Transform the original system into one that Conjugate Gradients will work on

$$E^{-1}A(E^{-1})^T\tilde{x} = E^{-1}b \qquad *$$

Where $EE^T = M$ = positive definite matrix

and $\tilde{x} = E^T x$

Solve for $\tilde{x}$ from eqn * and then for x from $E^T x = \tilde{x}$

@ 2004 Pearson Education Inc. All rights reserved.

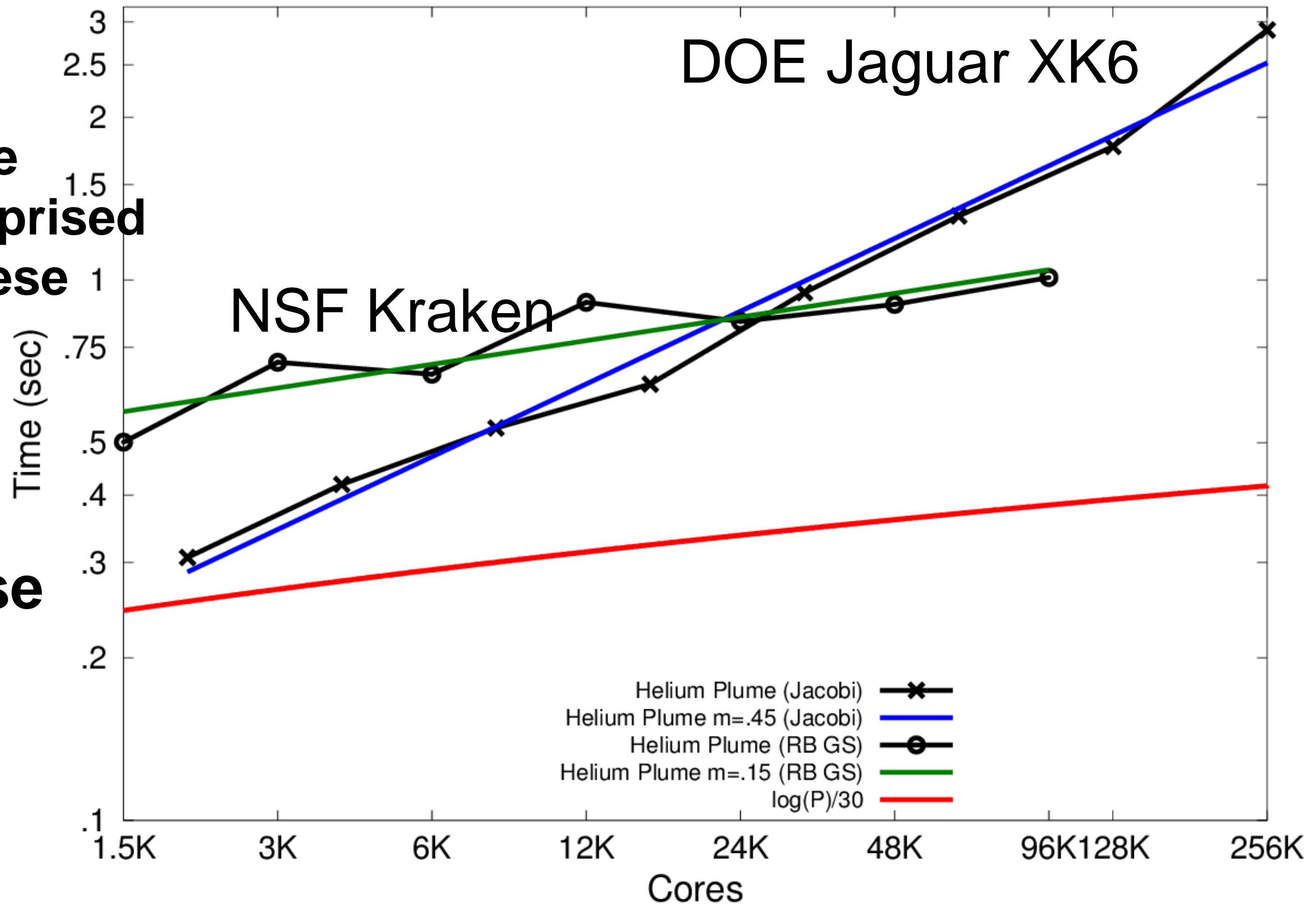
Congugate Gradients Continued

- In general preconditioning may involve techniques like
- Matrix multiplication partial LU factorisation or even multigrid
- On the next slides is an example of a pre-conditioned Krylov method.
- State of the art parallel iterative solver is hypre from Lawrence Livermore National Laboratory
- A multigrid method is used as a preconditioner to a conjugate gradient solver for Laplaces equation with a RHS (pressure poisson equation).
- The relaxation scheme in the multigrid solver is a red-black Gauss Seidel method.
- See J. Schmidt, M. Berzins, J. Thornock, T. Saad, J. Sutherland. "**Large Scale Parallel Solution of Incompressible Flow Problems using Uintah and hypre,**" SCI Technical Report, No. UUSCI-2012-002, *SCI Institute, University of Utah*, 2012.

@ 2004 Pearson Education Inc. All rights reserved.

Helium Plume using Wasatch with Uintah – successful scaling with hypre

Even the hypre team were surprised that we got these results!

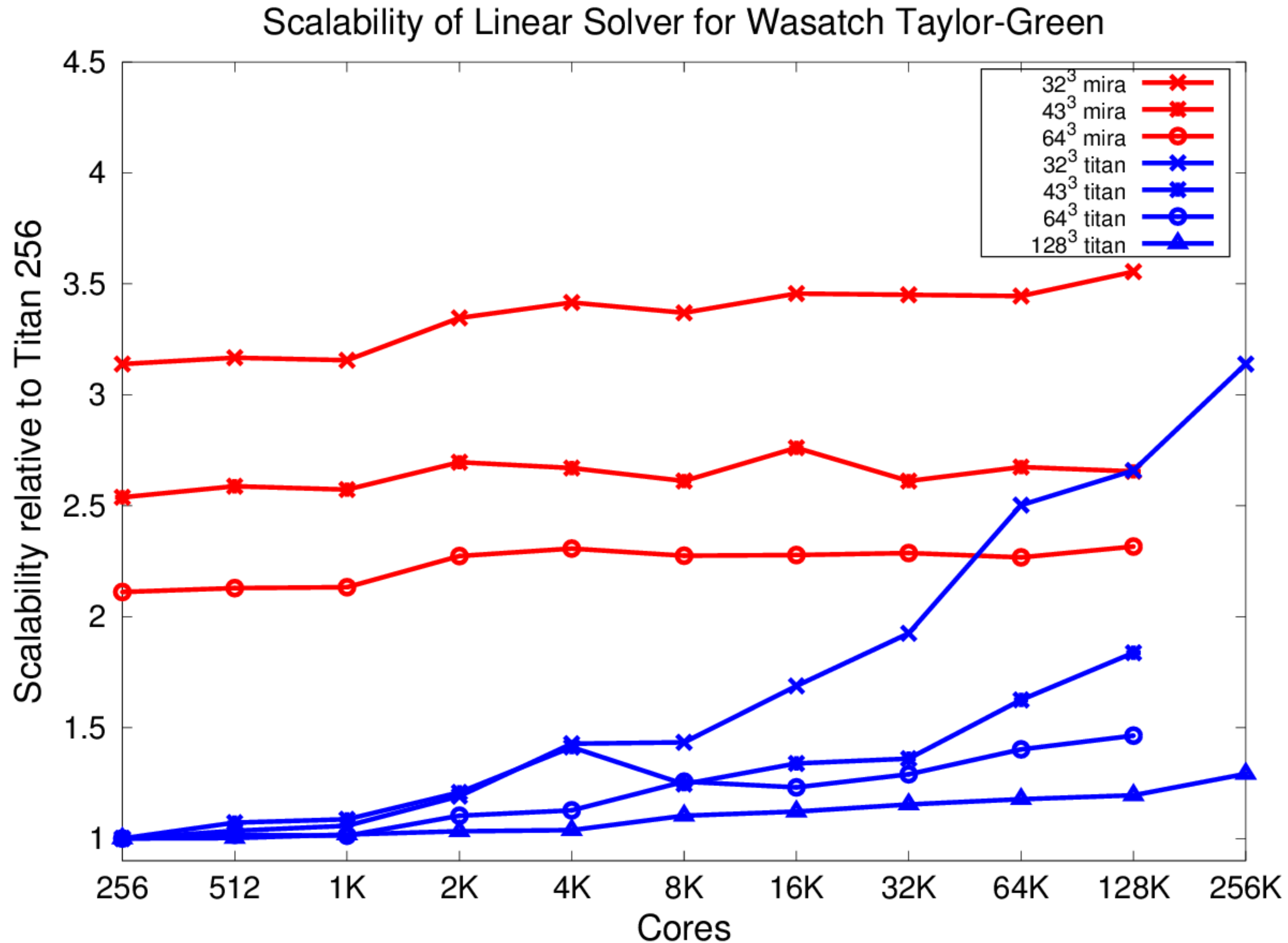


Largest case 30B unknowns

These results involved moving the hypre data structures into Uintah

[John Schmidt 2012]

Weak Scaling of Linear Solvers for Alstom-type application



Each **Mira Run** is scaled wrt the **Titan Run at 256 cores**
Note these times are not the same for different patch sizes.

2.2 Trillion DOF