Numerical Algorithms

Last update: ~~15/12/2020~~, 19/12/2021

Credit: ITCS 4/5145 Parallel Computing UNC-Charlotte, B. Wilkinson, 2009.

Numerical Algorithms

- Matrix multiplication
- Solving a system of linear equations

Matrices — A Review

An *n x m* matrix

$$\begin{array}{cc} & \xrightarrow{\text{Column}} \\ \text{Row}\Big\downarrow & \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,m-2} & a_{0,m-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,m-2} & a_{1,m-1} \\ \vdots & \vdots & & \vdots & \vdots \\ a_{n-2,0} & a_{n-2,1} & \cdots & a_{n-2,m-2} & a_{n-2,m-1} \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,m-2} & a_{n-1,m-1} \end{bmatrix} \end{array}$$

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

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

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

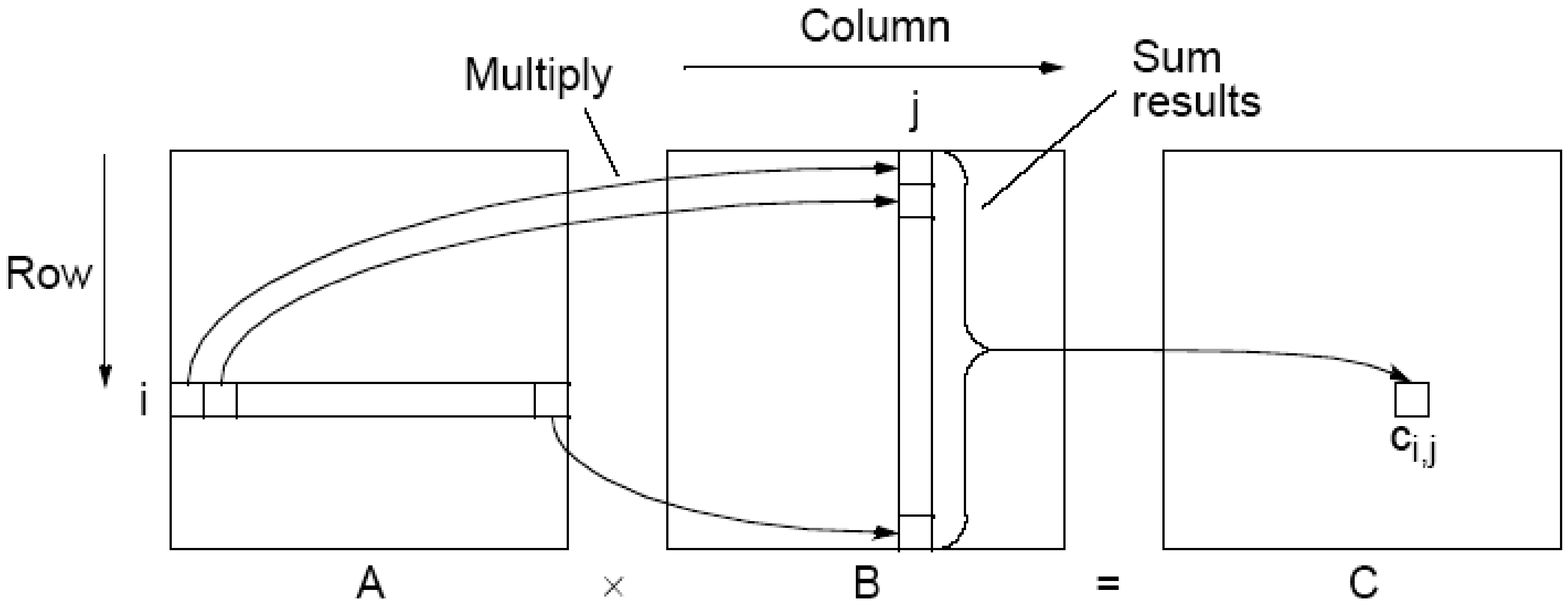
Matrix Multiplication

Multiplication of two matrices, **A** and **B**, produces 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.

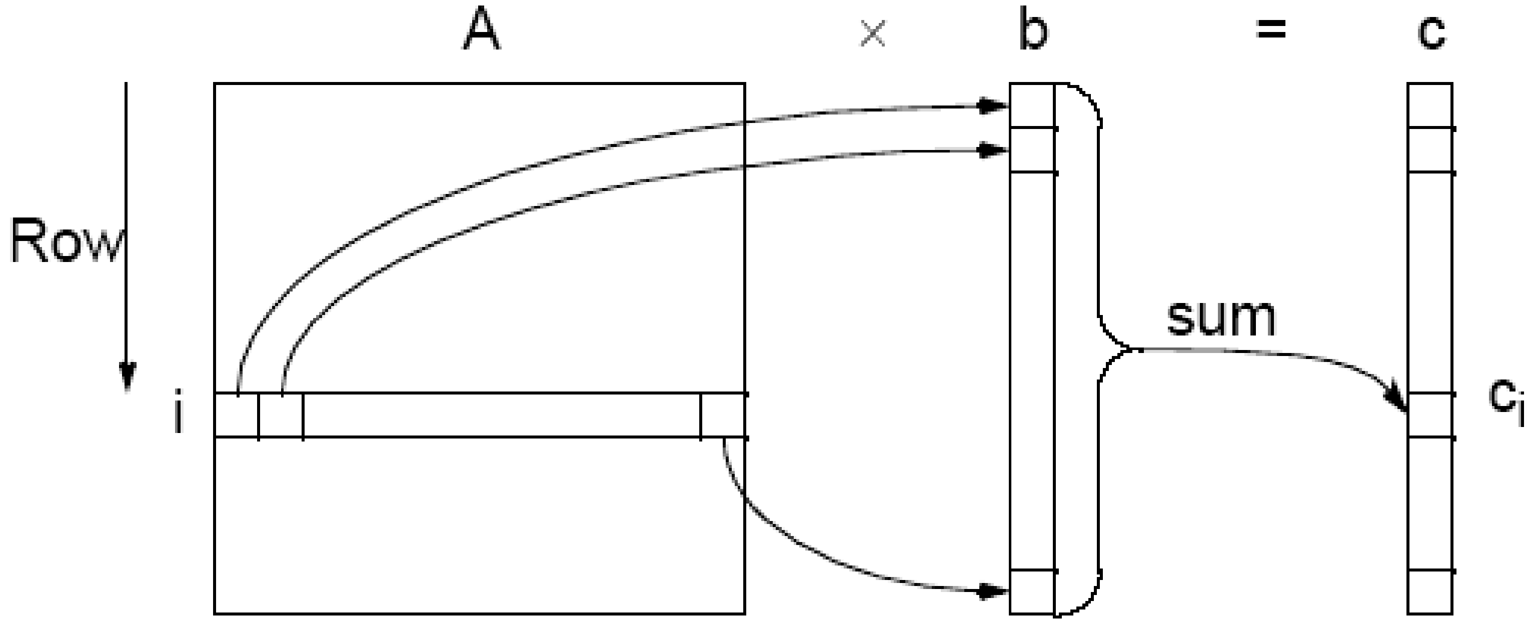
Matrix multiplication, C = A x B



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:

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

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 matrices square (*n* x *n* matrices).

Sequential code to compute **A** x **B** could simply be

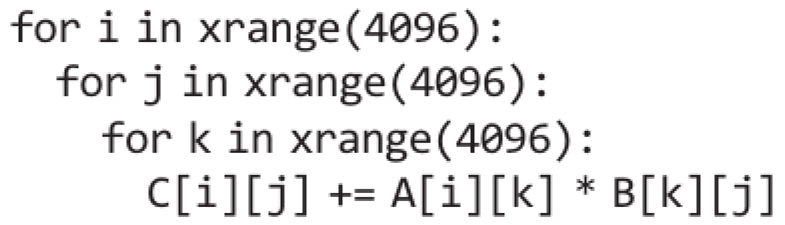
```
for (i = 0; i < n; i++)
        for (j = 0; j < n; j++) {
                c[i][j] = 0;
                for (k = 0; k < n; k++)
                        c[i][j] = c[i][j] + a[i][k] * b[k][j];
}
```

Requires n^3 multiplications and n^3 additions

Sequential time complexity of $O(n^3)$.

Very easy to parallelize.

הרחבה בעניין כפל מטריצות



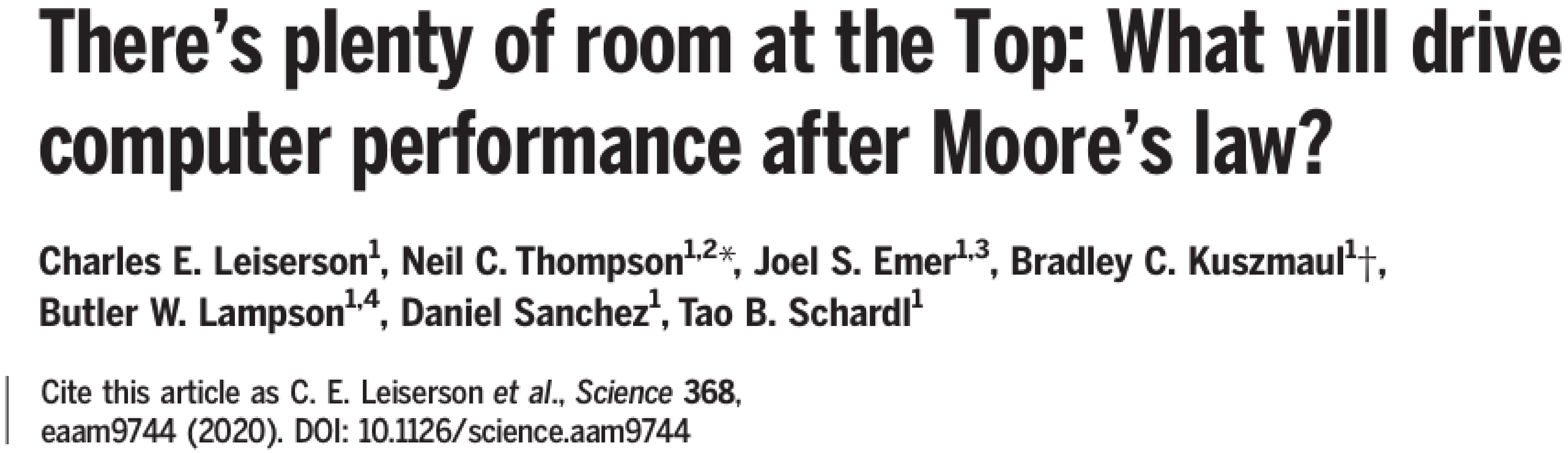
```
for i in xrange(4096):
  for j in xrange(4096):
    for k in xrange(4096):
      C[i][j] += A[i][k] * B[k][j]
```

קטע קוד אלגנטי בשפת פייטון.
הבעיה היא שהתכנית לוקחות כמה
שעות כדי לסיים אותה!!!

תכנית לדוגמה שלי

`/home/telzur/science/Teaching/PP/lectures/13/code/MatrixMatrixMult/mm.py`

סימוכין



There's plenty of room at the Top: What will drive computer performance after Moore's law?

Charles E. Leiserson[1], Neil C. Thompson[1,2]*, Joel S. Emer[1,3], Bradley C. Kuszmaul[1]†, Butler W. Lampson[1,4], Daniel Sanchez[1], Tao B. Schardl[1]

Cite this article as C. E. Leiserson *et al.*, *Science* 368, eaam9744 (2020). DOI: 10.1126/science.aam9744

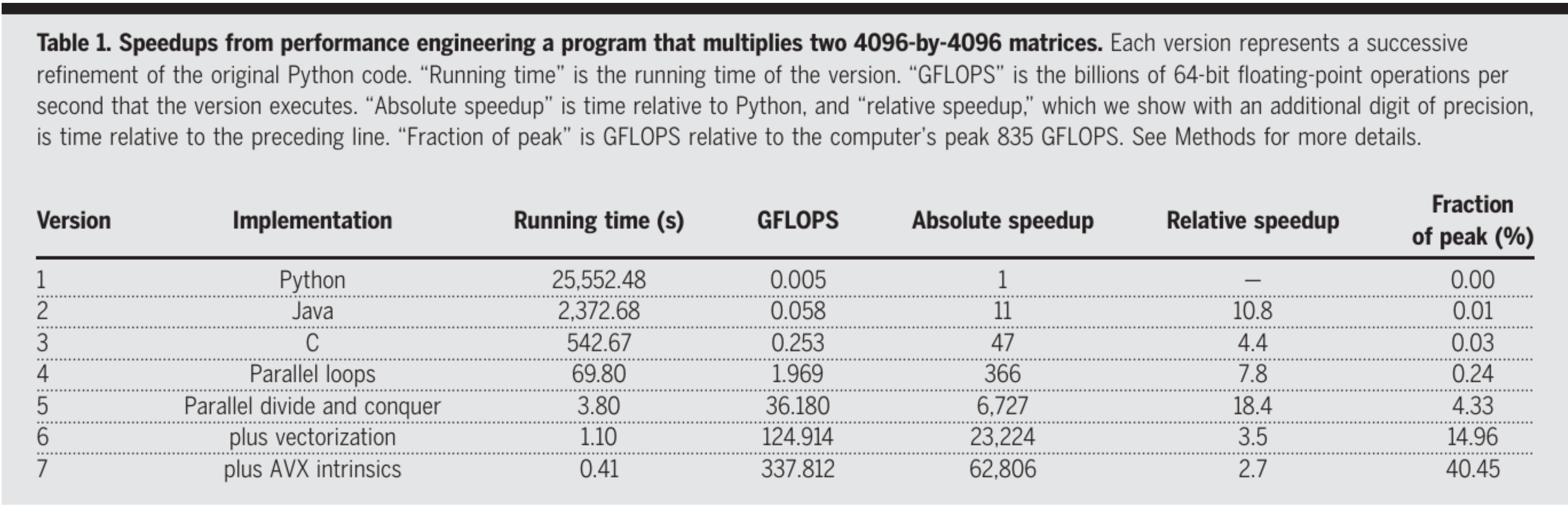


Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. “Running time” is the running time of the version. “GFLOPS” is the billions of 64-bit floating-point operations per second that the version executes. “Absolute speedup” is time relative to Python, and “relative speedup,” which we show with an additional digit of precision, is time relative to the preceding line. “Fraction of peak” is GFLOPS relative to the computer’s peak 835 GFLOPS. See Methods for more details.

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	–	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

מתוך הסימוכין מהשקף הקודם

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 $O(n^3) = n \times O(n^2) = n^2 \times O(n)$].

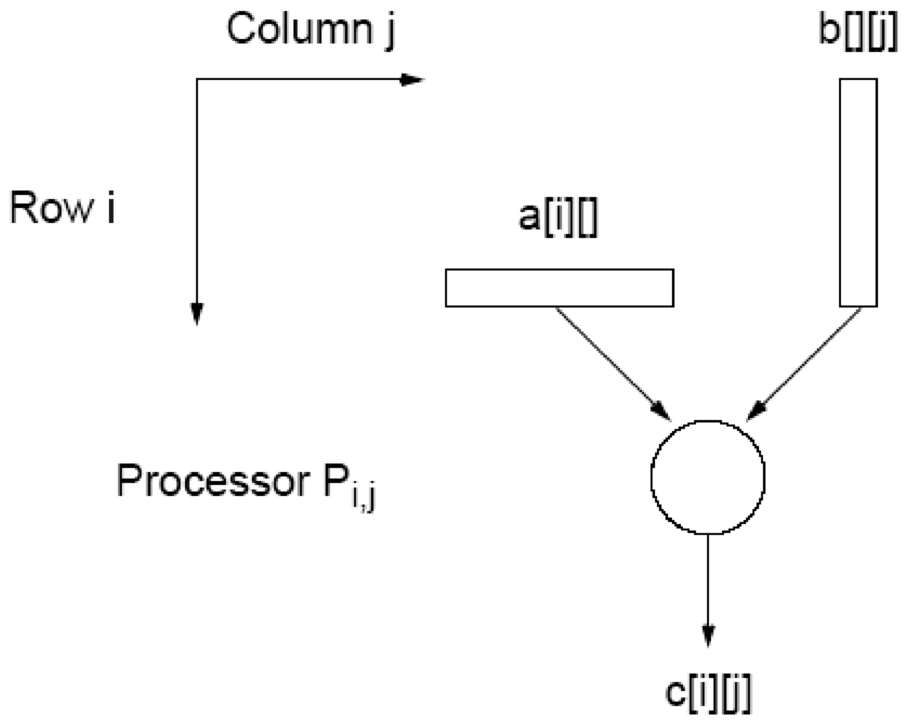
- Time complexity of $O(\log n)$ with n^3 processors
  By parallelizing the inner loop. Not cost-optimal.

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

Direct Implementation

הסבר מדוע
Time complexity of O(log n) with n^3 processors

יש במטריצה n**2 אלמנטים, כל אלמנט כזה ניתן לחישוב במקביל עם n מעבדים, סה"כ n כפול n**2



Performance Improvement

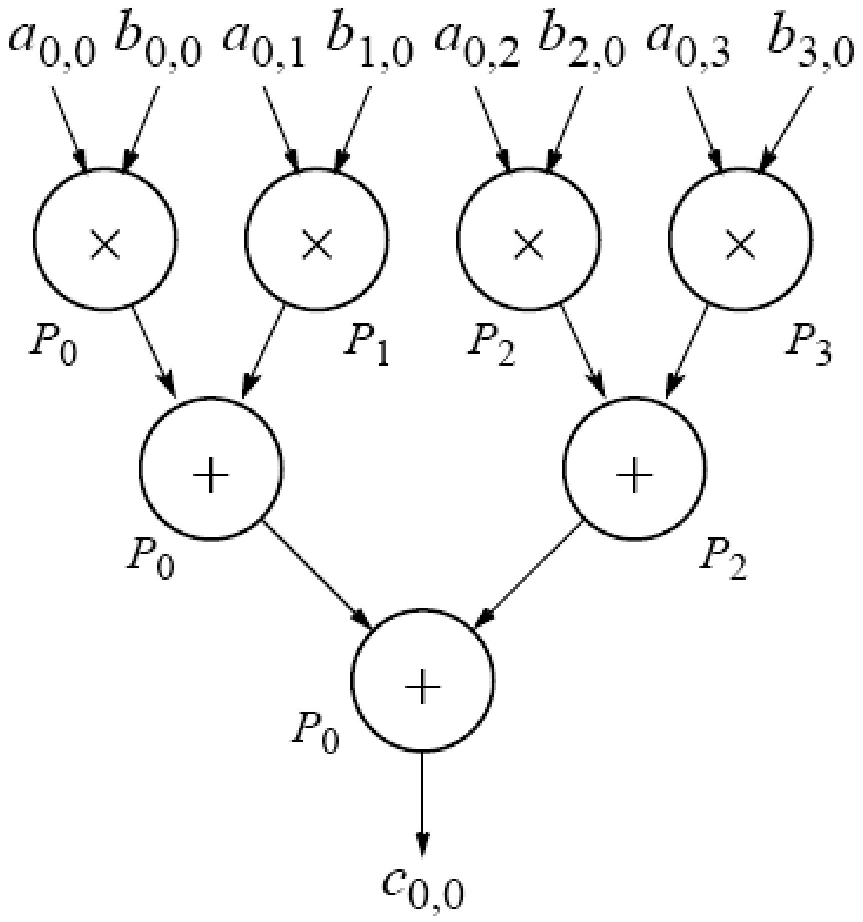
המשך הסבר מדוע
Time complexity of O(log n) with n³ processors

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

לאיבר בודד של המטריצה

לכן לכל המטריצה

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



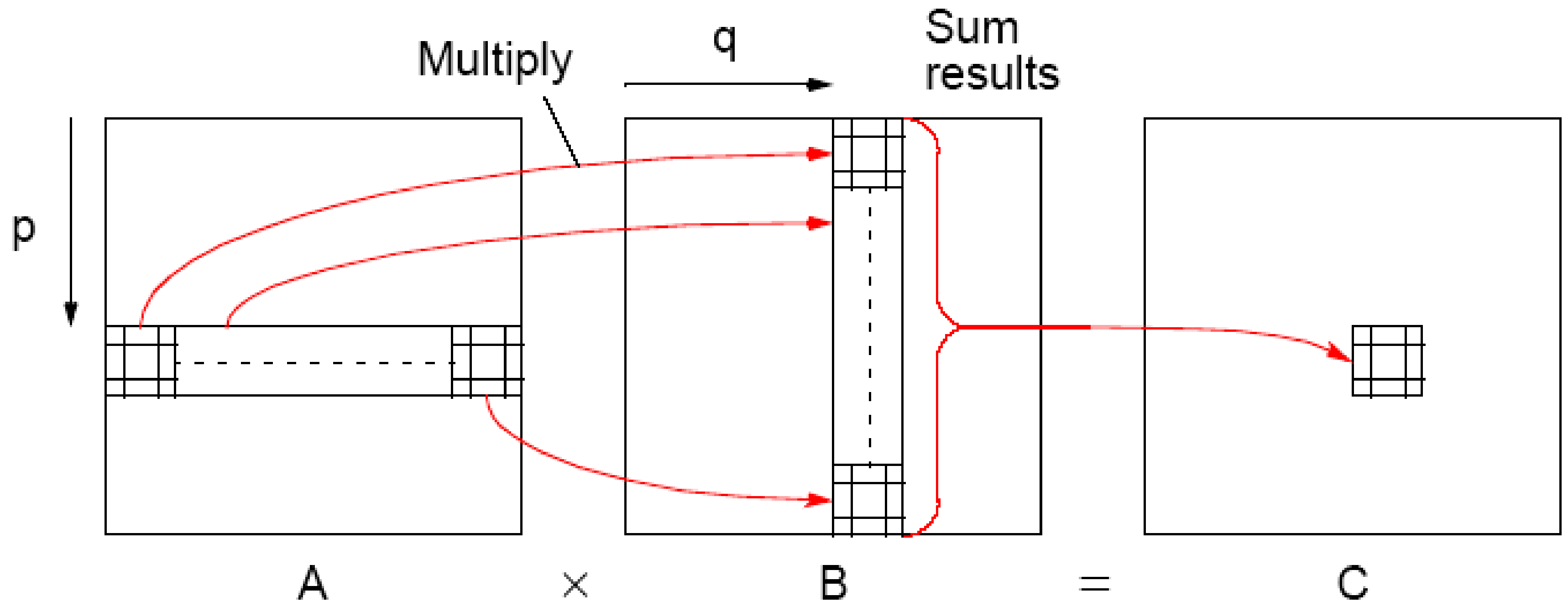
Partitioning into Submatrices

Suppose matrix divided into ***s***² submatrices.
Each submatrix has n/s x n/s elements.
$A_{p,q}$ means submatrix row p and column q of matrix A:

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

The line: **$C_{p,q} = C_{p,q} + A_{p,r} * B_{r,q}$;**
means multiply submatrix $A_{p,r}$ and $B_{r,q}$ using matrix multiplication and add to submatrix $C_{p,q}$ 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 $C_{0,0}$

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

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

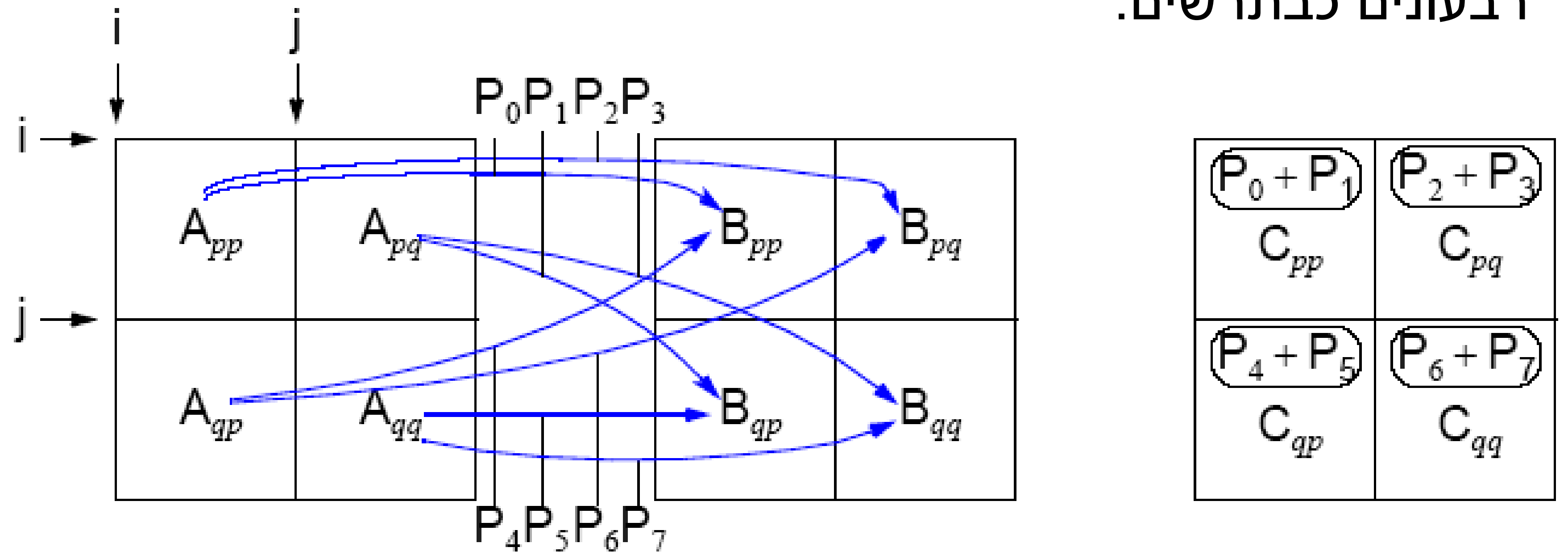
$$\begin{bmatrix} a_{0,0}b_{0,0}+a_{0,1}b_{1,0}+a_{0,2}b_{2,0}+a_{0,3}b_{3,0} & a_{0,0}b_{0,1}+a_{0,1}b_{1,1}+a_{0,2}b_{2,1}+a_{0,3}b_{3,1} \\ a_{1,0}b_{0,0}+a_{1,1}b_{1,0}+a_{1,2}b_{2,0}+a_{1,3}b_{3,0} & a_{1,0}b_{0,1}+a_{1,1}b_{1,1}+a_{1,2}b_{2,1}+a_{1,3}b_{3,1} \end{bmatrix}$$

$$= C_{0,0}$$

השקף מדגים את השקילות בין ביצוע כפל מטריצות איבר-איבר לבין כפל מטריצות באמצעות כפל תת-מטריצות

Recursive Implementation

עם 8 מעבדים ניתן לחשב <u>במקביל</u> כפל מטריצות המחולקות ל 4 רבעונים כבתרשים:



Apply same algorithm on each submatrix recursively.

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

Recursive Algorithm

s is the (sub)matrix size

```
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 */
}
```

הרחבה בעניין כפל מטריצות רקורסיבי

In the previous slide we saw that there are 8 multiplications and 4 additions.

Let’s estimate the computation time:
t~ 8M(n/2)+4A
Where: M stands for *Multiplication*, A stands for A*ddition*

MM(n/2) ~ O ((n/2)^3), 4A(n/2)~O(n^3).
(MM=Matrix Multiply)
According to Master theorem:
t ~ O(n^3) → there is no speedup in the divide and conquer MM method! למרות האלגנטיות, אין שיפור בסיבוכיות
Master theorem: https://brilliant.org/wiki/master-theorem/

The following 3 slides are taken from: CMPUT204
https://webdocs.cs.ualberta.ca/~mreza/courses/204-F07/lec-notes/week07.pdf

Strassen MM

- Want to use divide and conquer to speed things up; for simplicity assume n is a power of 2.
- Break each of X and Y into 4 submatrices of size $\frac{n}{2} \times \frac{n}{2}$ each:

$$\underbrace{\begin{bmatrix} A & B \\ C & D \end{bmatrix}}_{X} \underbrace{\begin{bmatrix} E & F \\ G & H \end{bmatrix}}_{Y} = \underbrace{\begin{bmatrix} I & J \\ K & L \end{bmatrix}}_{Z}$$

- Therefore:

$$\left.\begin{array}{l} I = AE + BG \\ J = AF + BH \\ K = CE + DG \\ L = CF + DH \end{array}\right\} \longrightarrow$$

need 8 multiplications of subproblems of size $\frac{n}{2}$ each

- We also need to spend $O(n^2)$ time to add up these results.

עד כאן סיכמנו מידע שכבר צויין בשקפים הקודמים של חישוב רקורסיבי

- If $T(n)$ is the time to multiply two matrices of size $n \times n$ each, then:
$$T(n) = 8T(\frac{n}{2}) + O(n^2)$$

- Using master theorem: $T(n) \in \Theta(n^{\log_2 8}) = \Theta(n^3)$.

- So this is as bad as the naive algorithm. No improvement yet.

- We use an idea similar to the one for multiplication of large integers: reduce the number of subproblems using a clever trick.

- compute the following 7 multiplications (each consisting of two subproblems of size $\frac{n}{2}$ each):
$$\begin{aligned} S_1 &= A(F - H) \\ S_2 &= (A + B)H \\ S_3 &= (C + D)E \\ S_4 &= D(G - E) \\ S_5 &= (A + D)(E + H) \\ S_6 &= (B - D)(G + H) \\ S_7 &= (A - C)(E + F) \end{aligned}$$

$$\begin{aligned} I &= S_5 + S_6 + S_4 - S_2 \\ &= (A+D)(E+H) + (B-D)(G+H) + D(G-E) - (A+B)H \\ &= AE + DE + AH + DH + BG - DG + BH - DH + \\ &\qquad DG - DE - AH - BH \\ &= AE + BG \end{aligned}$$

- Similarly, it can be verified easily that:

$$\begin{aligned} J &= S_1 + S_2 \\ K &= S_3 + S_4 \\ L &= S_1 - S_7 - S_3 + S_5 \end{aligned}$$

- So to compute I, J, K, and L, we only need to compute $S_1, \ldots, S_7$; this requires solving seven subproblems of size $\frac{n}{2}$, plus a constant (at most 16) number of addition each taking $O(n^2)$ time.

$$T(n) = 7T(\frac{n}{2}) + O(n^2)$$

- Using master theorem and since $\log_2 7 \approx 2.808$:

$$T(n) \in O(n^{2.808})$$

- For $n = 50,000$: $n^3 \approx 10^{17}$ and $n^{2.808} \approx 10^{13}$; $\longrightarrow$ this algorithm is about 10,000 times faster than the naive algorithm.

For further reading:
"Introduction to Algorithms" by Thomas H. Cormen
Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.
3rd Edition, 2009.
<u>Section 4.2</u>

See also this video:
https://www.youtube.com/watch?v=E-QtwPi620I

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

Mesh of processors with wraparound connections 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 moved from their initial position to an "aligned" position. *i*th row of A is shifted *i* places left and *j*th column of B is shifted *j* places upward. Has effect of placing element $a_{i,j+}i$ and element $b_{i+j,j}$ in processor $P_{i,j}$. These elements are a pair of those required in accumulation of $c_{i,j}$.

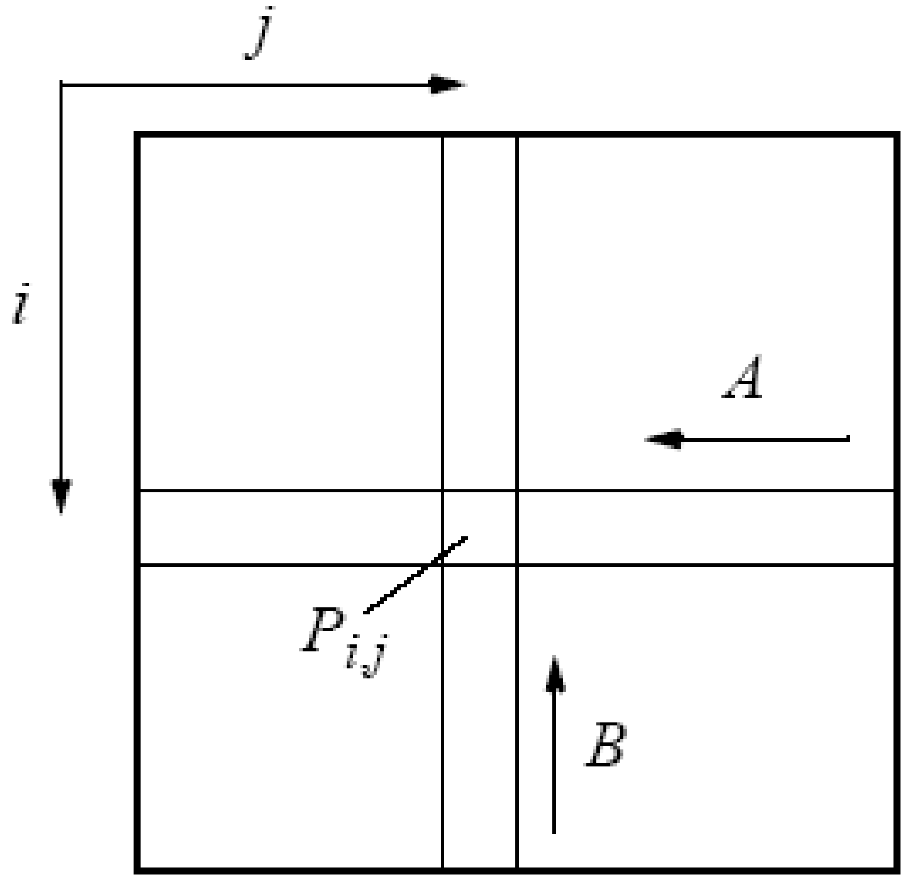
3. Each processor, $P_{i,j}$, multiplies its elements.

4. The *i*th row of A is shifted one place right, and the *j*th column of B is shifted one place upward. Has the effect of bringing together adjacent elements of A and B, which will also be required in the accumulation.

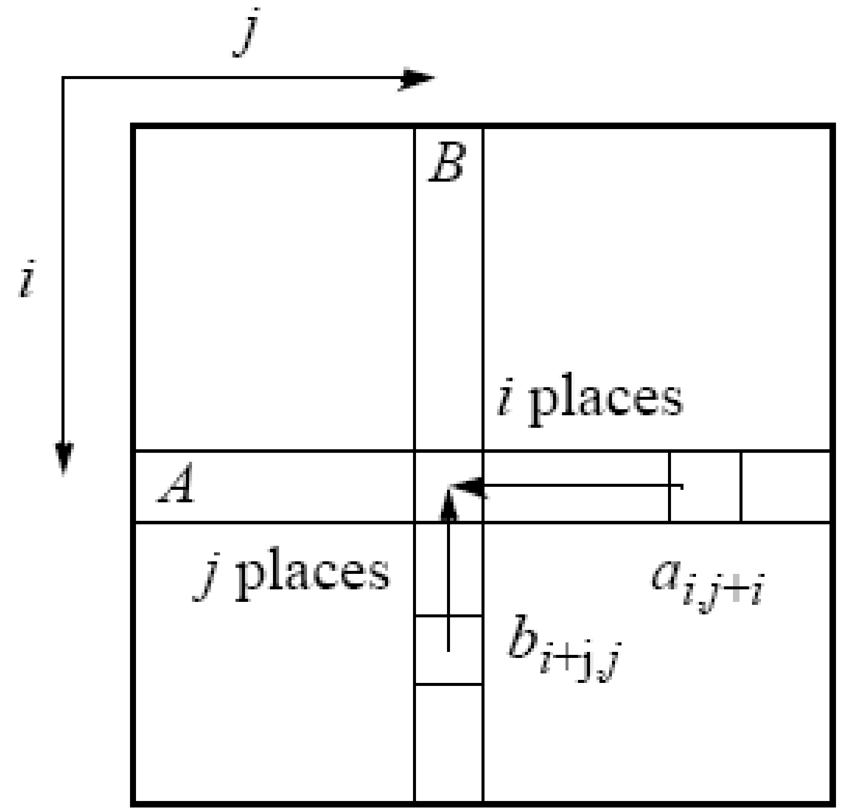
5. Each processor, $P_{i,j}$, multiplies elements brought to it and adds result to accumulating sum.

6. Step 4 and 5 repeated until final result obtained (*n* - 1 shifts with *n* rows and *n* columns of elements).

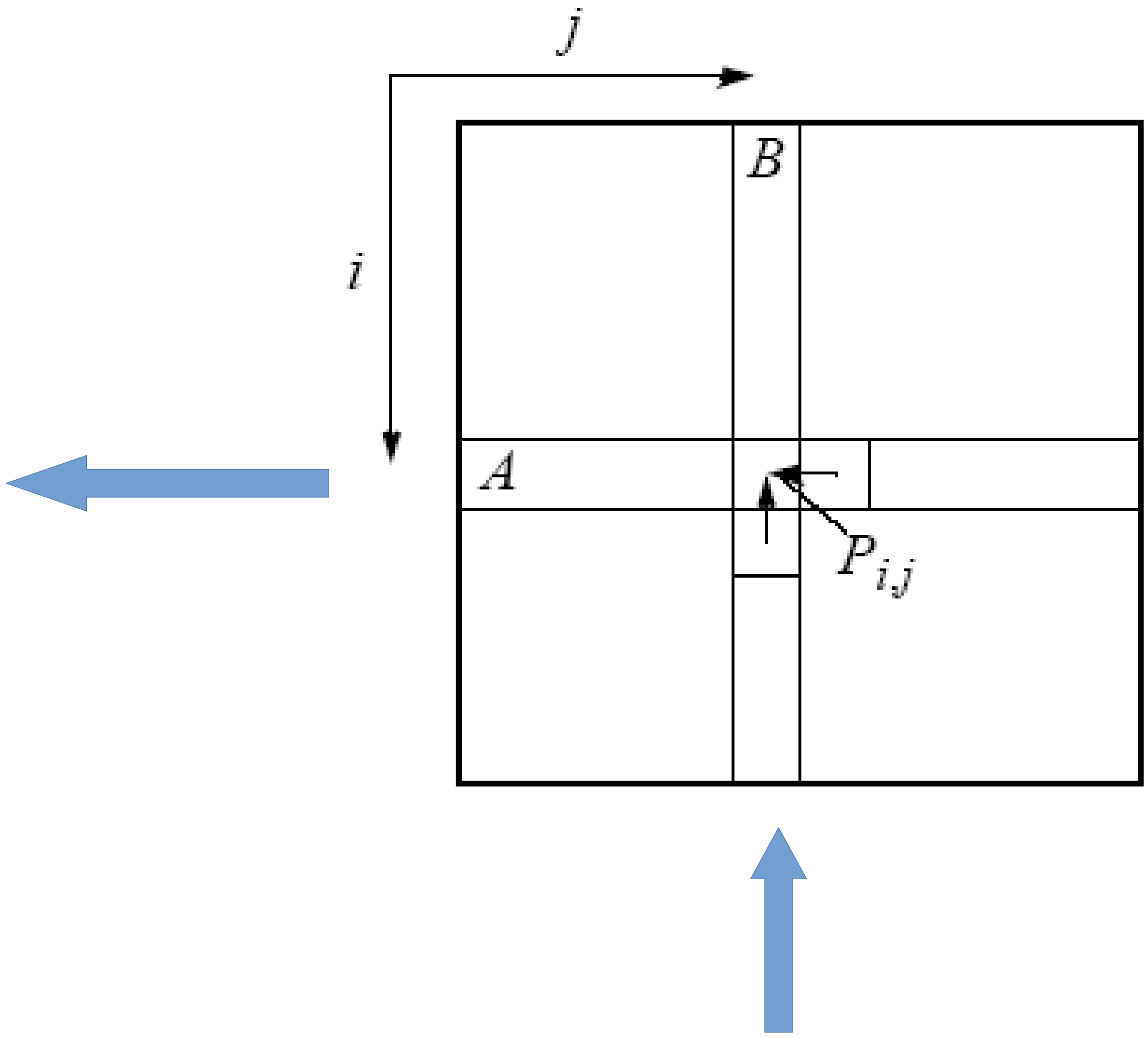
Movement of *A* and *B* elements



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

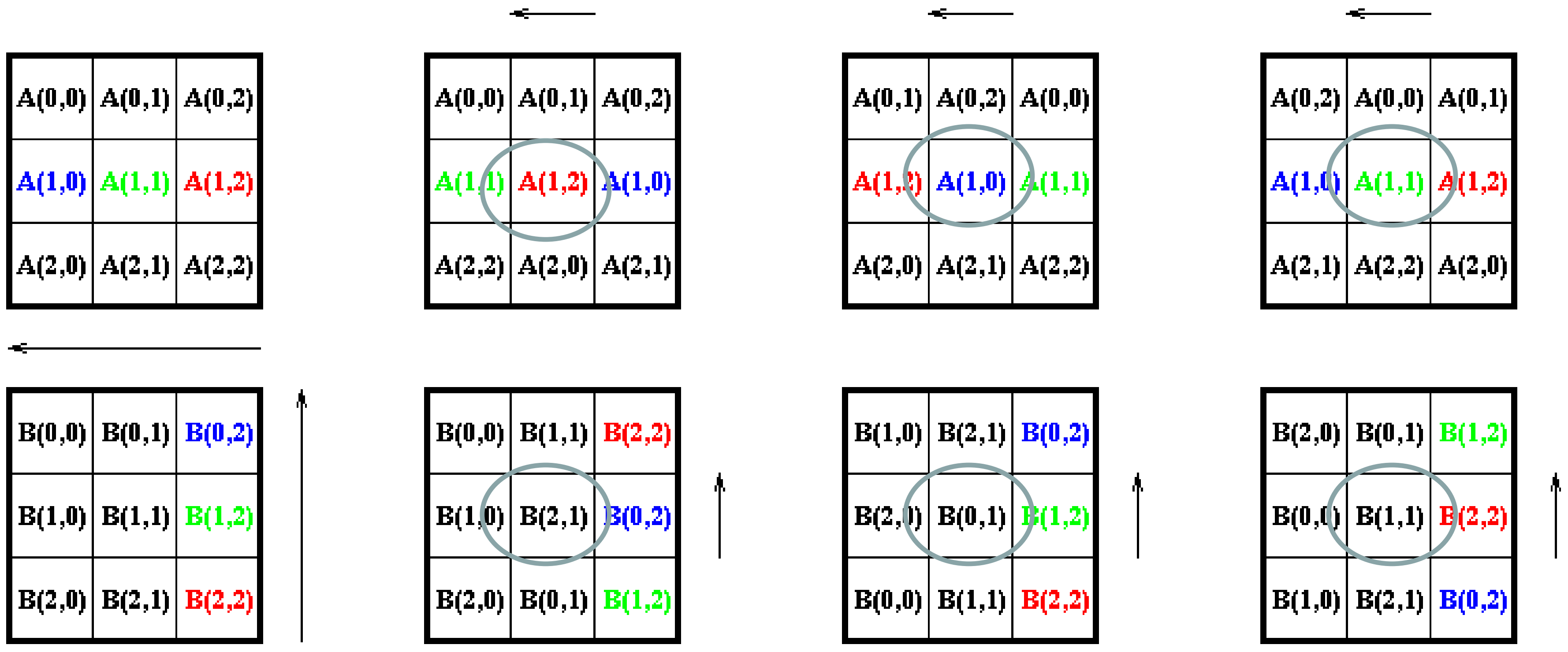


Step 4 - One-place shift of elements of *A* and *B*



הפרוססורים נייחים והמספרים מוזדים

Cannon's Matrix Multiplication Algorithm



Initial A, B   A, B after skewing   A, B after shift k=1   A, B after shift k=2

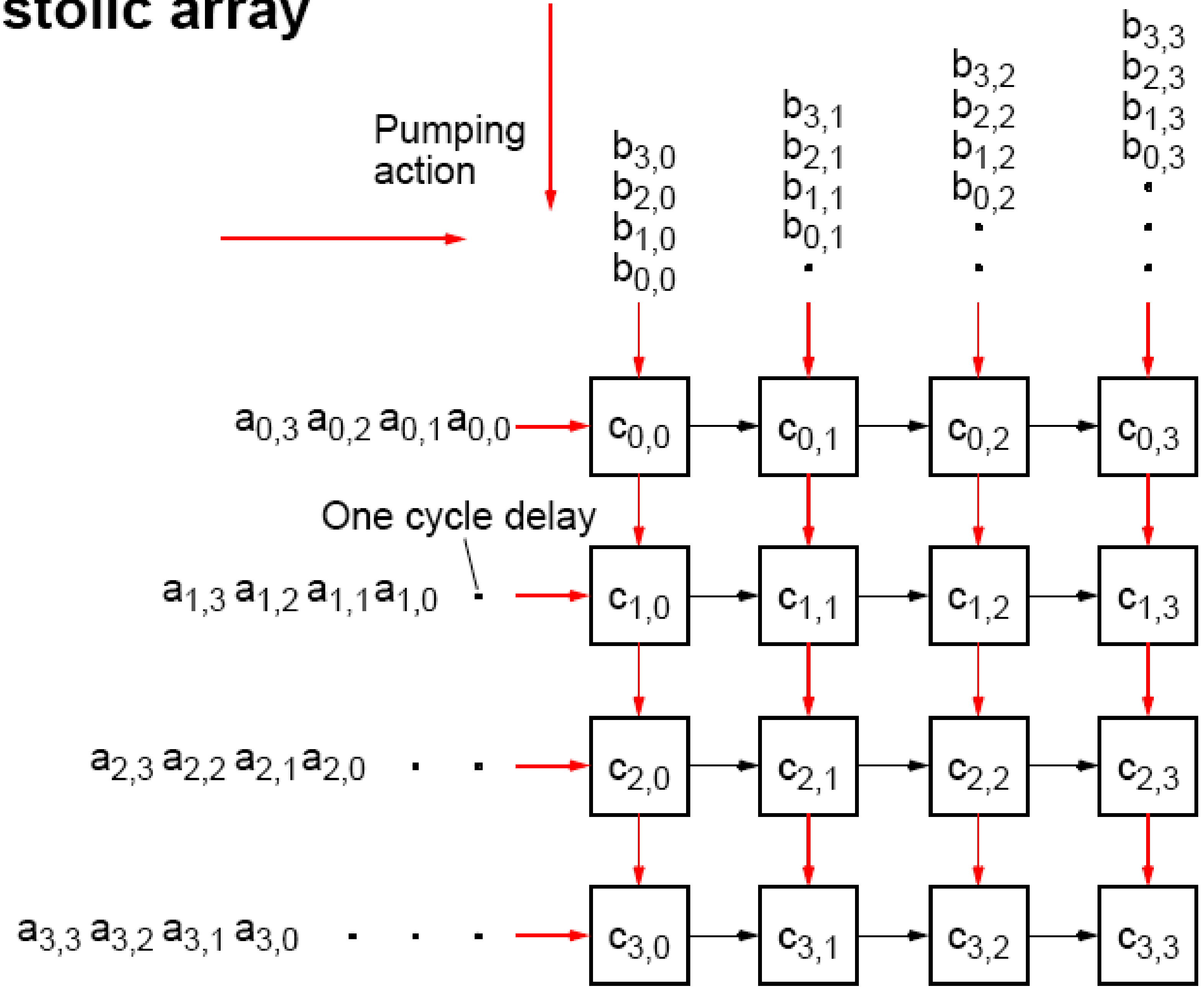
C11 = a12·b21 + a10·b01+a11·b11

http://www.eecs.berkeley.edu/~demmel/cs267/lecture11/lecture11.html

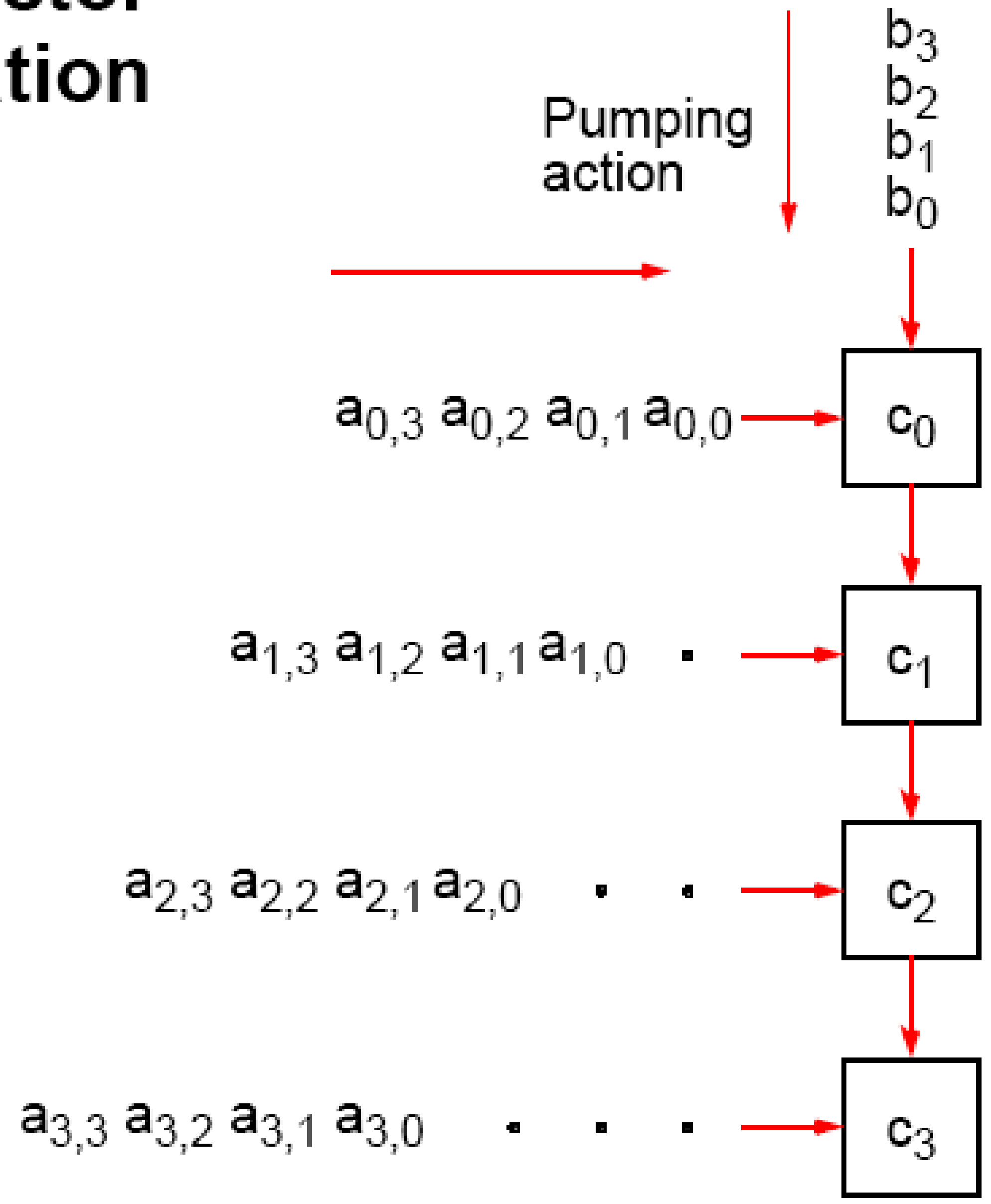
המחשות של האלגוריתם של קנון

https://www.dailymotion.com/video/x2vv5pm

Systolic array


Pumping action
$b_{3,0}$ $b_{2,0}$ $b_{1,0}$ $b_{0,0}$
$b_{3,1}$ $b_{2,1}$ $b_{1,1}$ $b_{0,1}$
$b_{3,2}$ $b_{2,2}$ $b_{1,2}$ $b_{0,2}$
$b_{3,3}$ $b_{2,3}$ $b_{1,3}$ $b_{0,3}$
$a_{0,3}$ $a_{0,2}$ $a_{0,1}$ $a_{0,0}$
$a_{1,3}$ $a_{1,2}$ $a_{1,1}$ $a_{1,0}$
$a_{2,3}$ $a_{2,2}$ $a_{2,1}$ $a_{2,0}$
$a_{3,3}$ $a_{3,2}$ $a_{3,1}$ $a_{3,0}$
One cycle delay
$c_{0,0}$ $c_{0,1}$ $c_{0,2}$ $c_{0,3}$
$c_{1,0}$ $c_{1,1}$ $c_{1,2}$ $c_{1,3}$
$c_{2,0}$ $c_{2,1}$ $c_{2,2}$ $c_{2,3}$
$c_{3,0}$ $c_{3,1}$ $c_{3,2}$ $c_{3,3}$

Matrix-Vector Multiplication


Pumping action
b_3
b_2
b_1
b_0
$a_{0,3}\ a_{0,2}\ a_{0,1}\ a_{0,0}$
$a_{1,3}\ a_{1,2}\ a_{1,1}\ a_{1,0}$.
$a_{2,3}\ a_{2,2}\ a_{2,1}\ a_{2,0}$ . .
$a_{3,3}\ a_{3,2}\ a_{3,1}\ a_{3,0}$ . . .
c_0
c_1
c_2
c_3

Solving a System of Linear Equations

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

which, in matrix form, is

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

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

Solving 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. Then solve by Back Substitution.

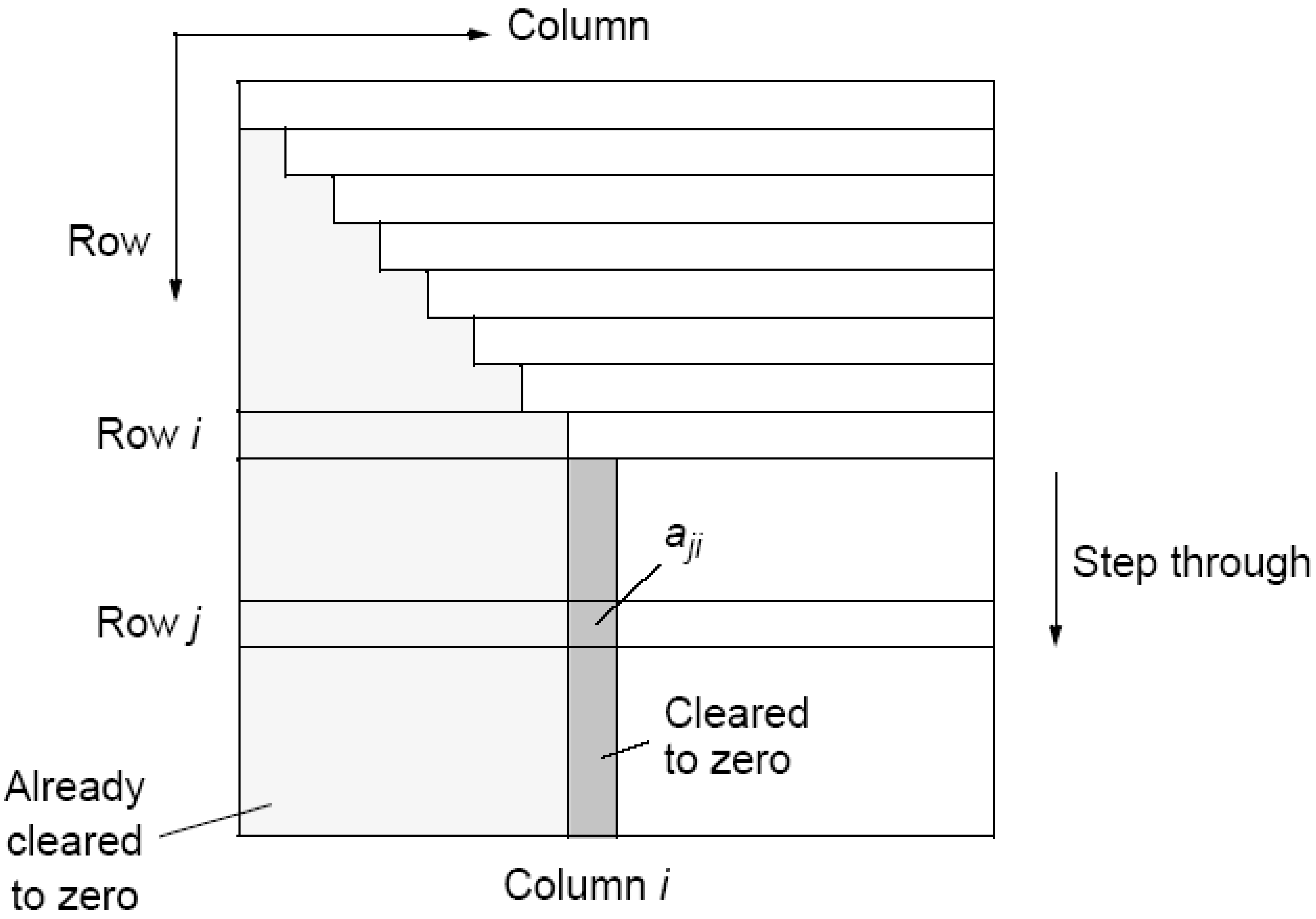
Uses characteristic of linear equations that any row can be replaced by that row added to another row multiplied by a constant.

Starts at first row and works toward bottom row. At *i*th row, each row *j* below *i*th row replaced by row *j* + (row *i*) (-*aj,i*/ *ai,i*).

Constant used for row *j* is -*aj,i*/*ai,i*. Has effect of making all elements in *i*th column below *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, will not be able to compute quantity $-a_{j,i}/a_{i,i}$.

Procedure must be modified into so-called *partial pivoting* by swapping *i*th row with row below it that has largest absolute element in *i*th column of any of rows below *i*th row if there is one. (Reordering equations will not affect the system.)

In the following, we will not consider partial pivoting.

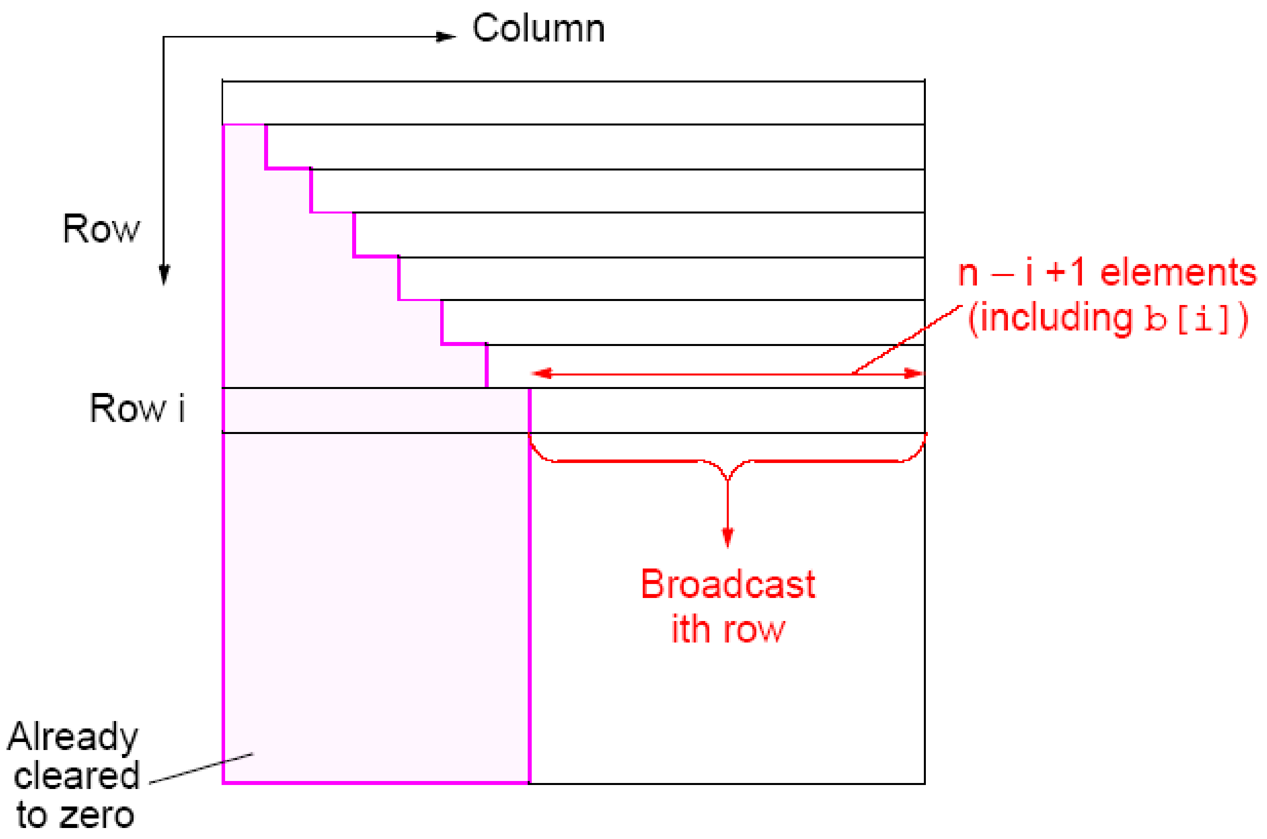
Sequential Code

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



Bcast
כדי לספק את
a[i][k]

לקריאה נוספת:

Designing and Building Parallel Programs:
https://www.mcs.anl.gov/~itf/dbpp/text/node90.html#SECTION03480000000000000000

UCB CS267:
https://people.eecs.berkeley.edu/~demmel/cs267_Spr16/Lectures/lecture13_densela_2_jwd16_4pp.pdf

Analysis

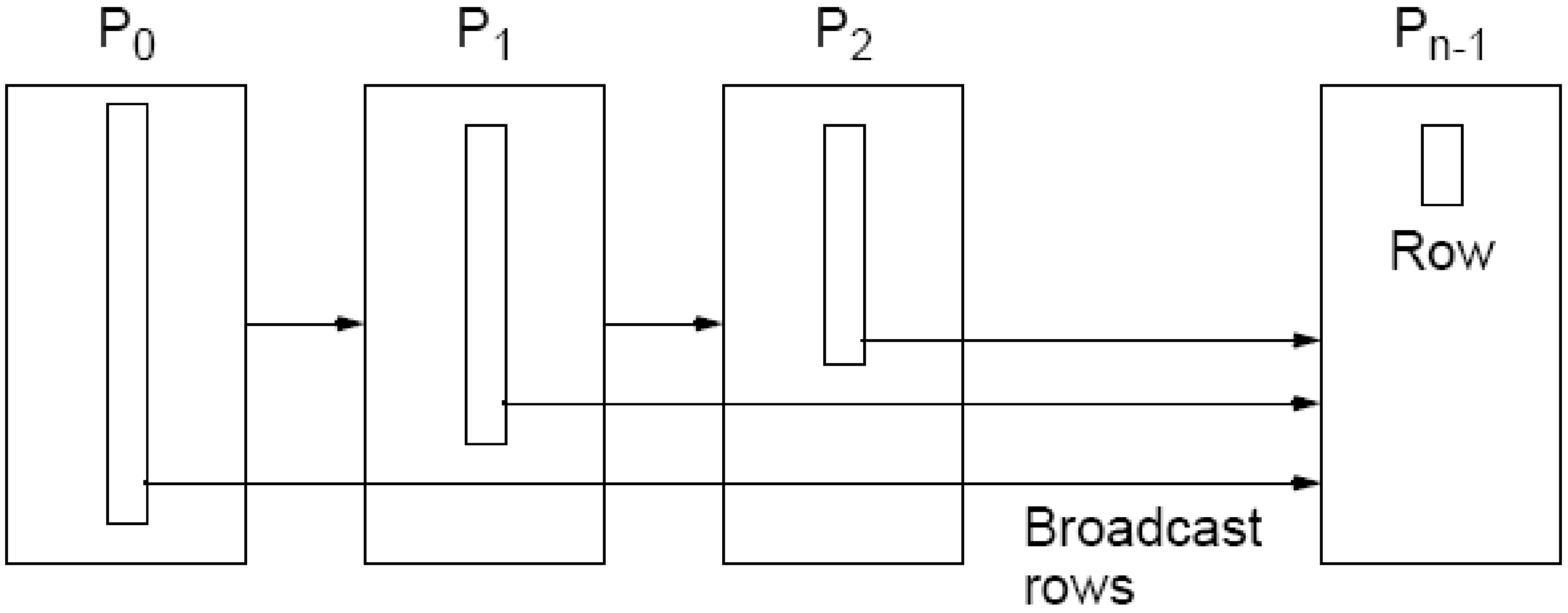
Communication

n - 1 broadcasts performed sequentially.
*i*th broadcast contains *n* - *i* + 1 elements.
Time complexity of O(n^2)   (see textbook)

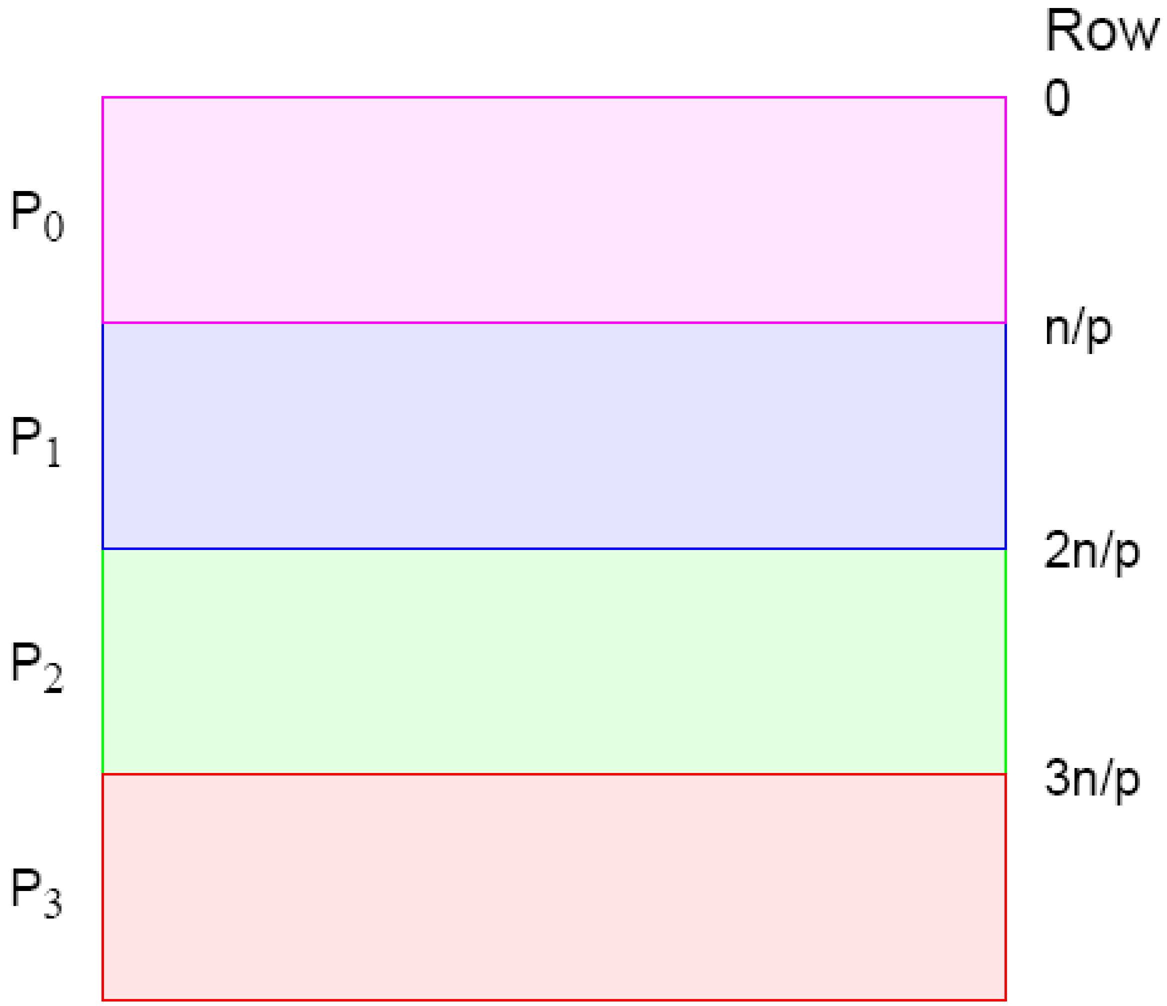
Computation

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 computation of multiplier, *n* - *j* + 2 multiplications and *n* - *j* + 2 subtractions.
Time complexity of O(n^2)   (see textbook).
Efficiency relatively low because all processors before processor holding row *i* do not participate in computation again.

Pipeline implementation of Gaussian elimination



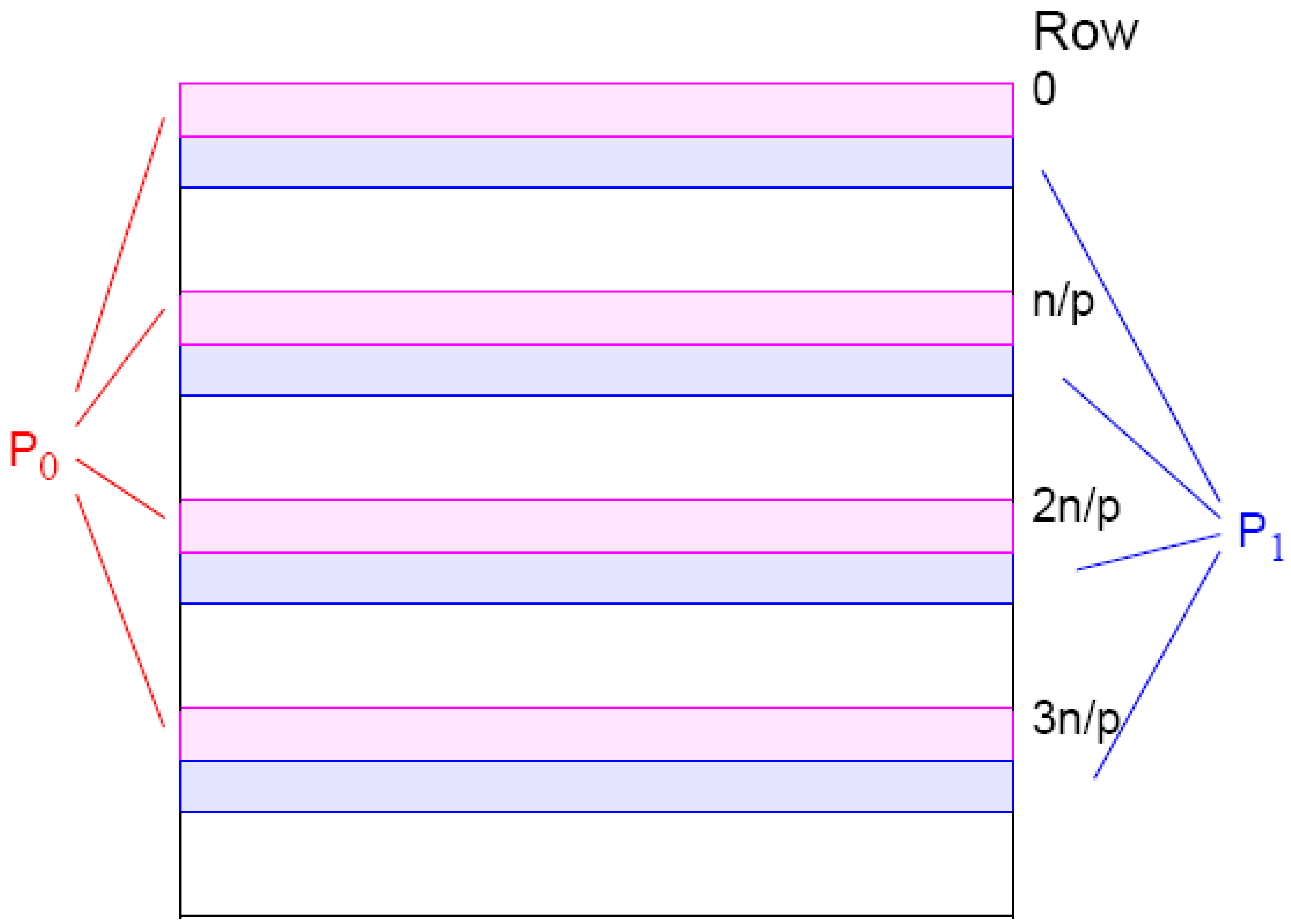
Strip Partitioning



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:

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

but can be less than 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 k*th 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

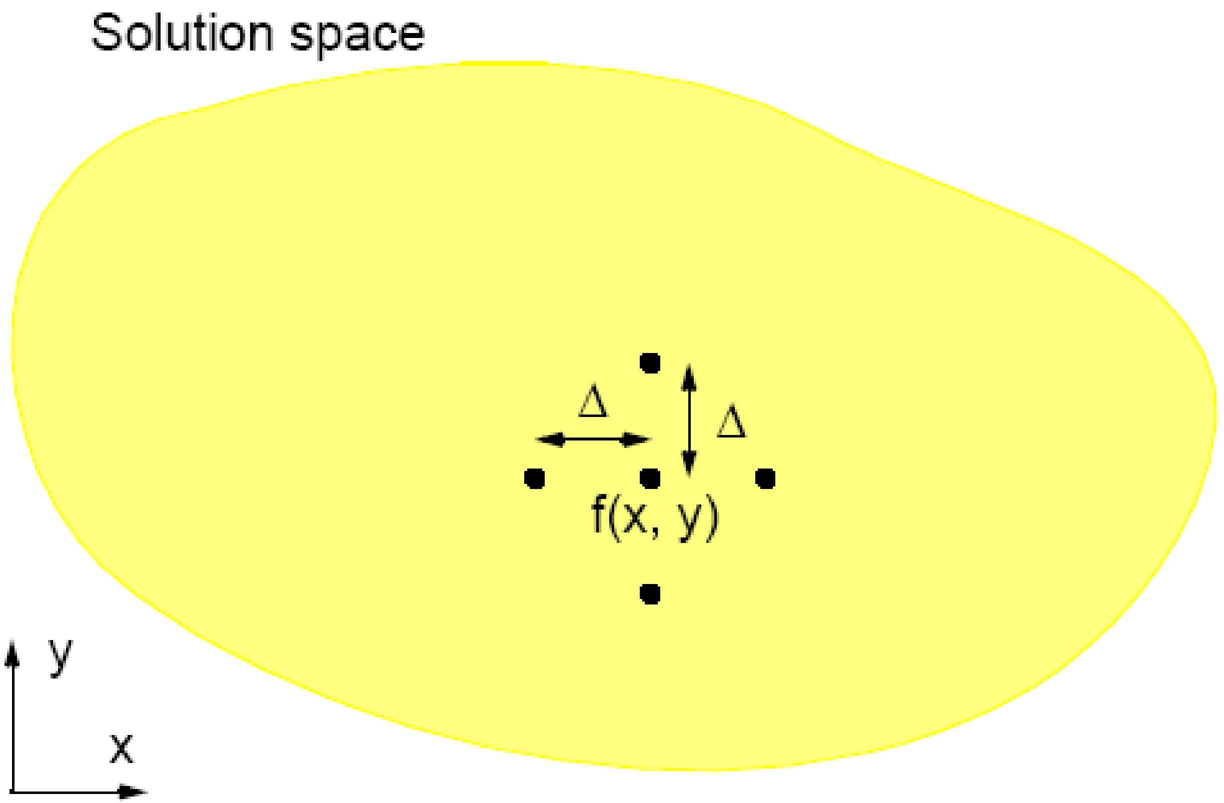
$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0$$

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

For computer solution, *finite difference* methods appropriate

Two-dimensional solution space "discretized" into 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

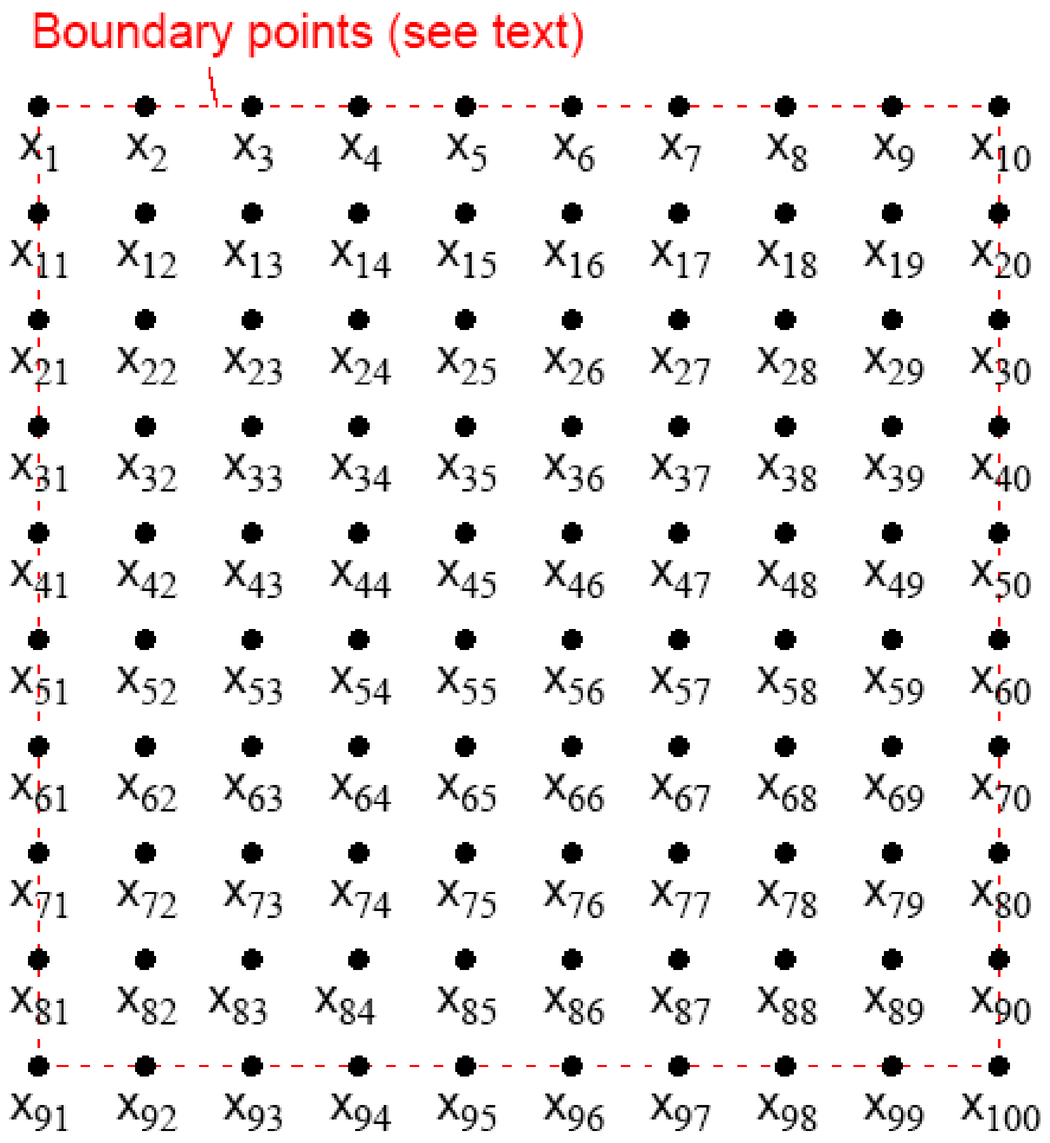
$$f(x, y) = \frac{[f(x-\Delta, y) + f(x, y-\Delta) + f(x+\Delta, y) + f(x, y+\Delta)]}{4}$$

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



Boundary points (see text)
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10}
x_{11} x_{12} x_{13} x_{14} x_{15} x_{16} x_{17} x_{18} x_{19} x_{20}
x_{21} x_{22} x_{23} x_{24} x_{25} x_{26} x_{27} x_{28} x_{29} x_{30}
x_{31} x_{32} x_{33} x_{34} x_{35} x_{36} x_{37} x_{38} x_{39} x_{40}
x_{41} x_{42} x_{43} x_{44} x_{45} x_{46} x_{47} x_{48} x_{49} x_{50}
x_{51} x_{52} x_{53} x_{54} x_{55} x_{56} x_{57} x_{58} x_{59} x_{60}
x_{61} x_{62} x_{63} x_{64} x_{65} x_{66} x_{67} x_{68} x_{69} x_{70}
x_{71} x_{72} x_{73} x_{74} x_{75} x_{76} x_{77} x_{78} x_{79} x_{80}
x_{81} x_{82} x_{83} x_{84} x_{85} x_{86} x_{87} x_{88} x_{89} x_{90}
x_{91} x_{92} x_{93} x_{94} x_{95} x_{96} x_{97} x_{98} x_{99} x_{100}

Relationship with a General System of Linear Equations

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

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

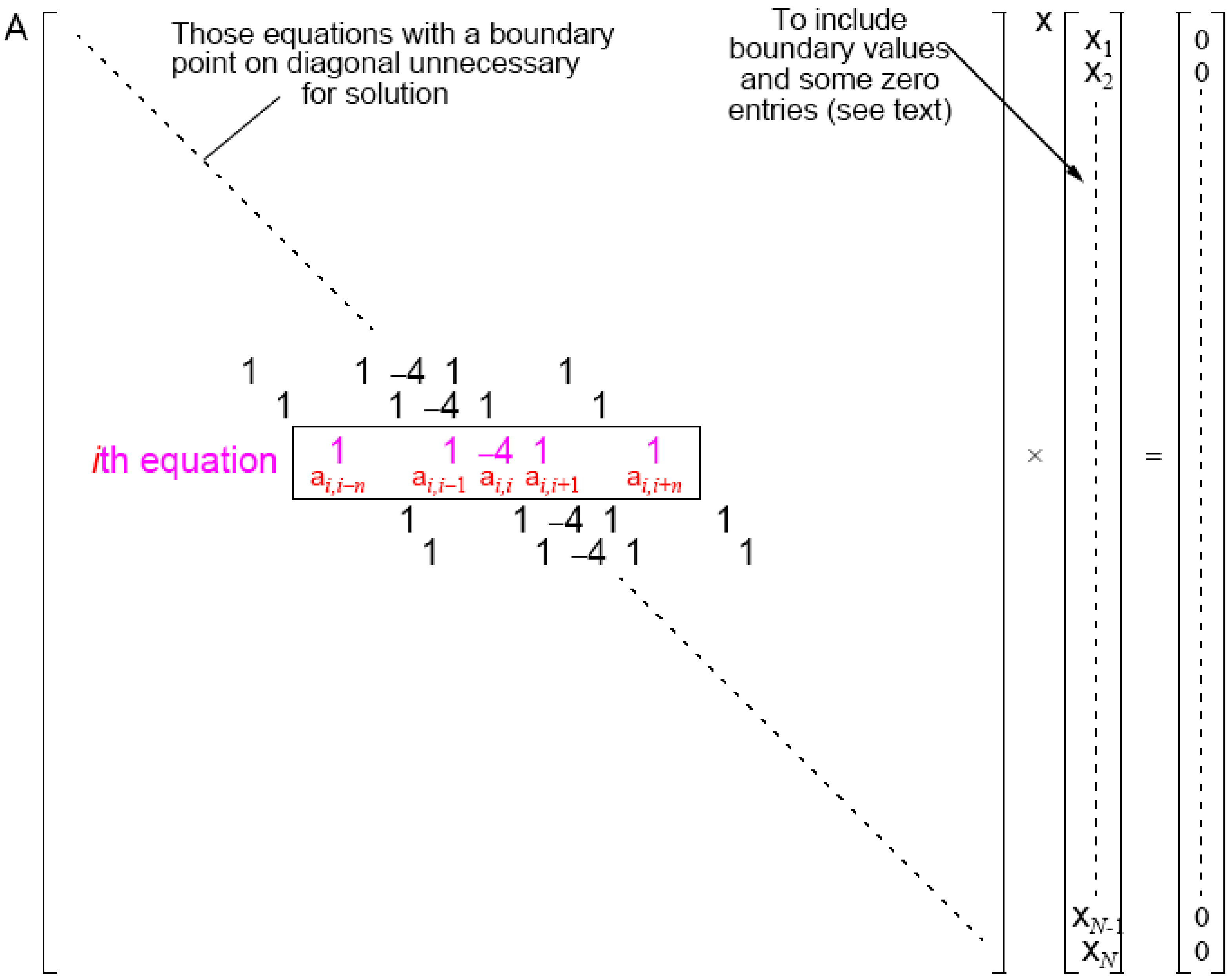
or

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

which is a linear equation with five unknowns (except those with boundary points).
In general form, the *i*th equation becomes:

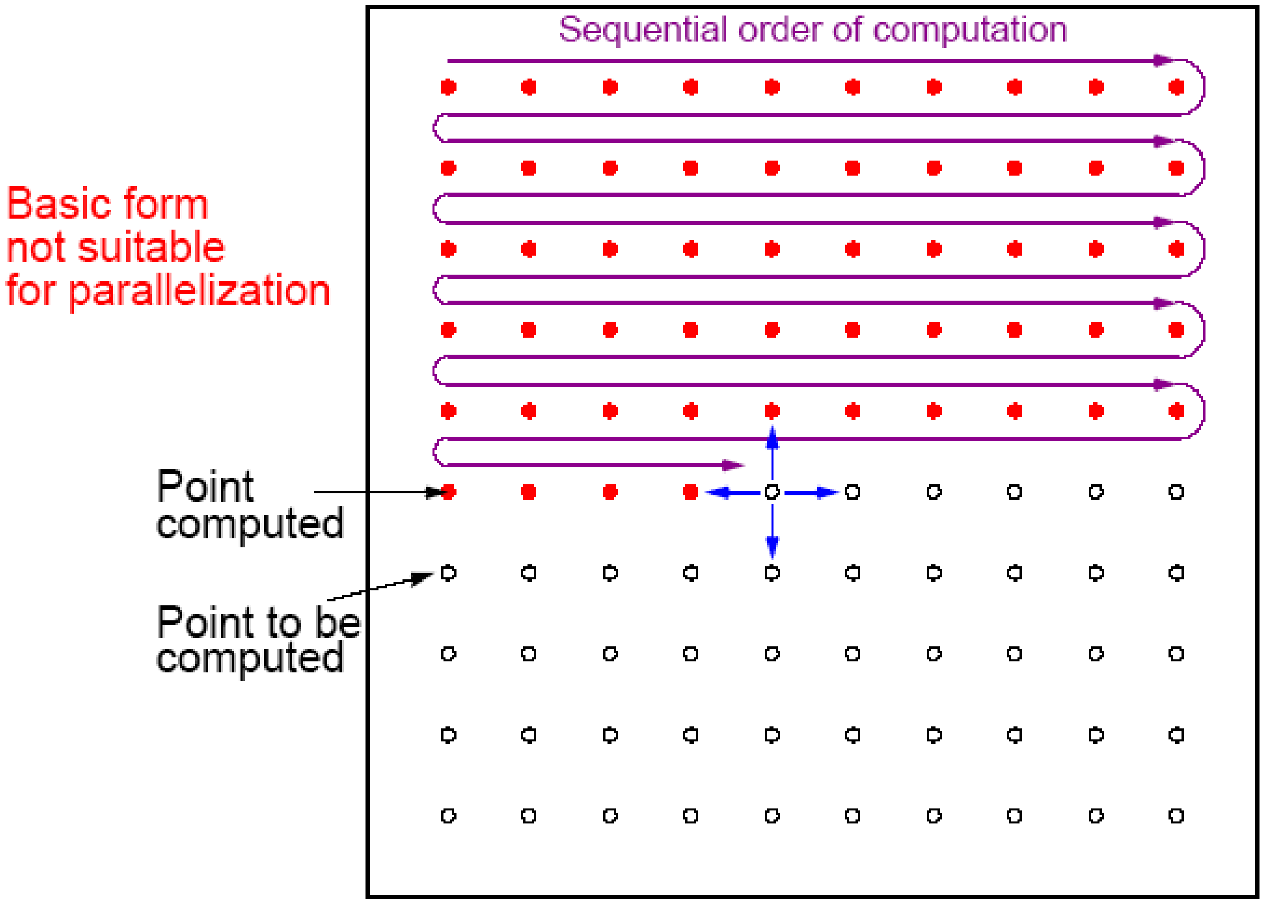
$$a_{i,i-n}x_{i-n} + a_{i,i-1}x_{i-1} + a_{i,i}x_i + a_{i,i+1}x_{i+1} + a_{i,i+n}x_{i+n} = 0$$

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
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_1
x_2
x_{N-1}
x_N
=
0
0
0
0

Gauss-Seidel Relaxation

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


Sequential order of computation
Basic form not suitable for parallelization
Point computed
Point to be computed

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 superscript indicates iteration.

With natural ordering of unknowns, formula reduces to

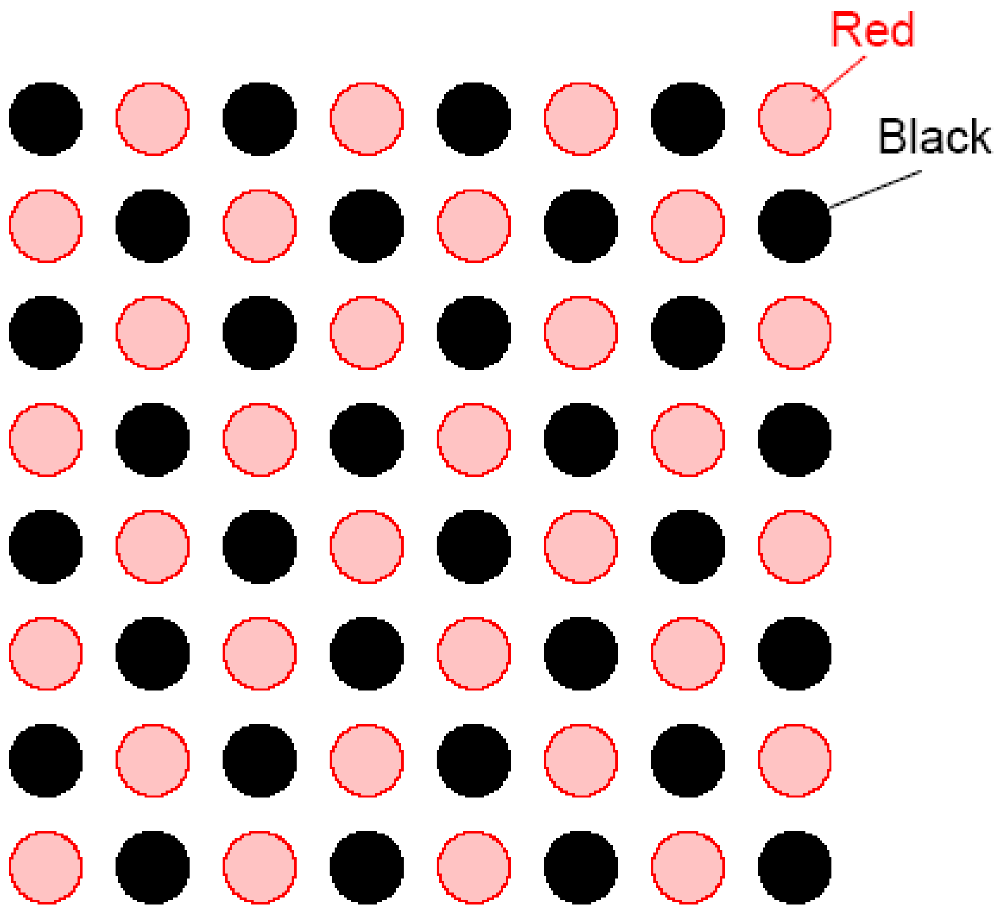
$$x_i^k = (-1/a_{i,i})[a_{i,i-n}x_{i-n}^k + a_{i,i-1}x_{i-1}^k + a_{i,i+1}x_{i+1}^{k-1} + a_{i,i+n}x_{i+n}^{k-1}]$$

At *k*th iteration, two of the four values (before *i*th element) taken from *k*th iteration and two values (after *i*th element) taken from (*k*-1)th iteration. 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]);
```

Higher-Order Difference Methods

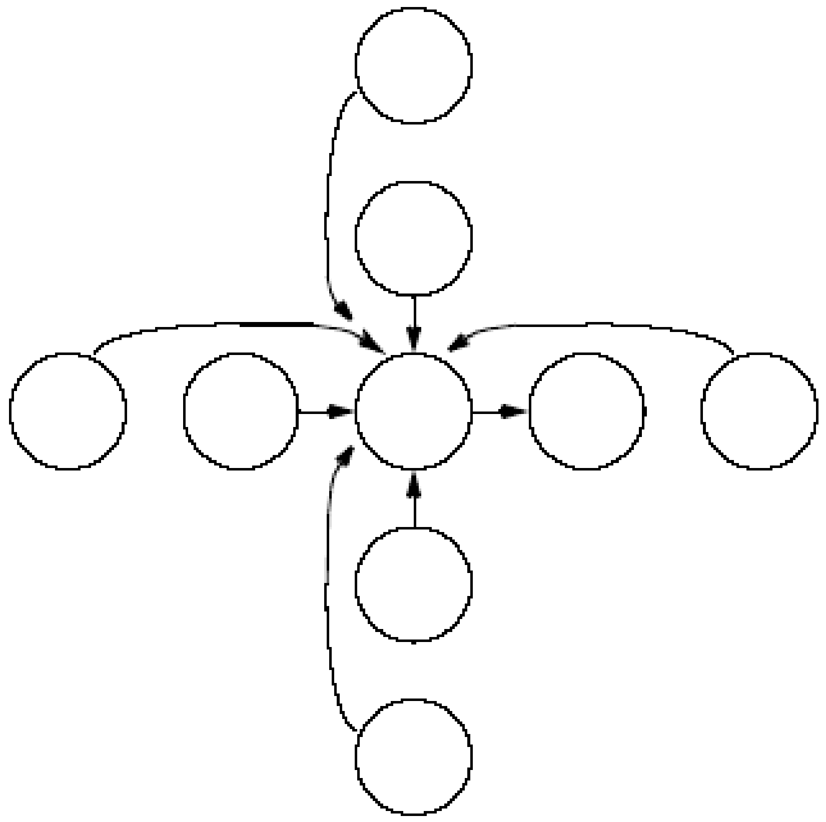
More distant points could be used in the computation. The following update formula:

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

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

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

Nine-point stencil



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.

Multigrid Method

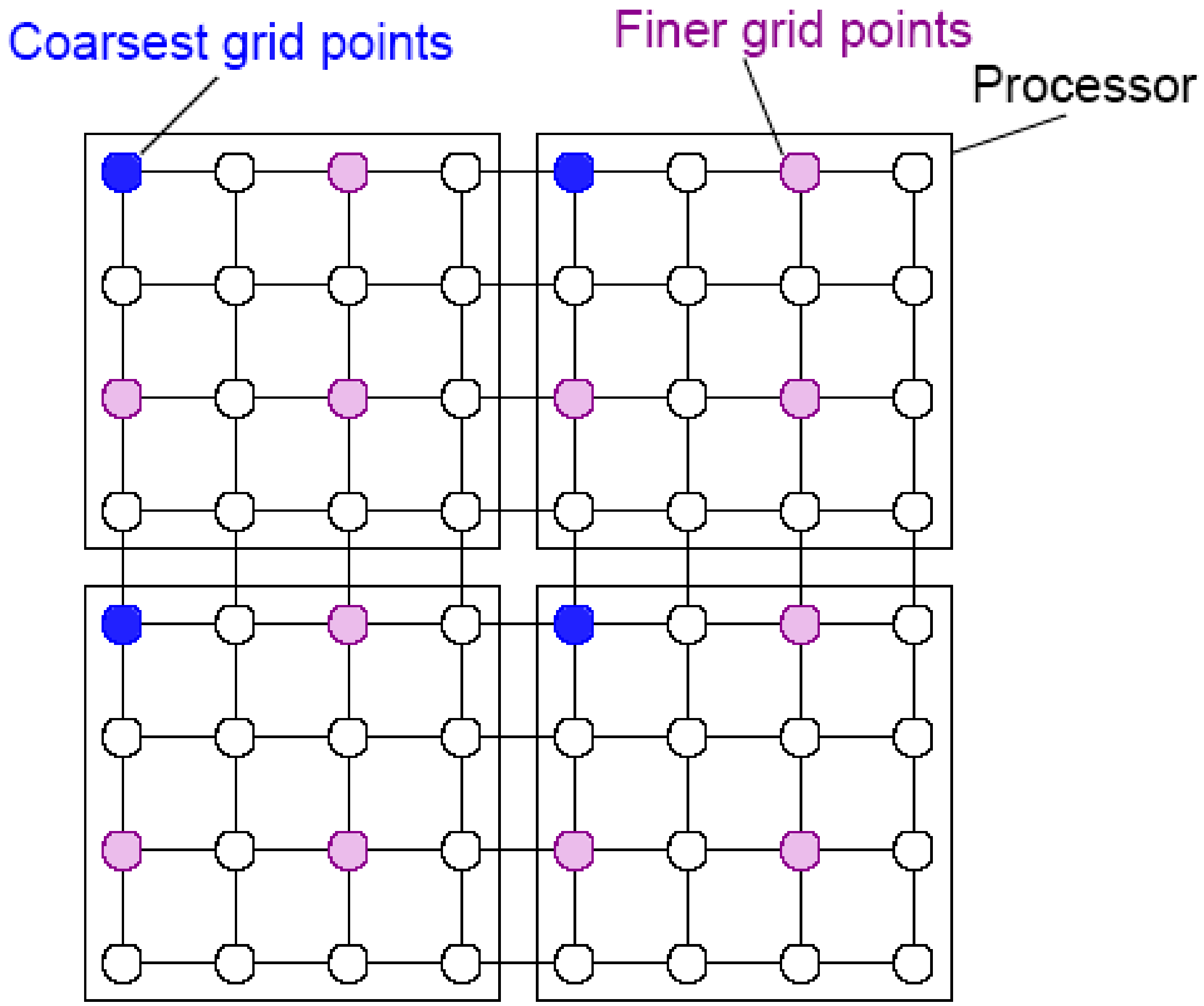
First, a coarse grid of points used. With these points, iteration process will start to converge quickly.

At some stage, number of points increased to include points of coarse grid and extra points between points of 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



(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.

Numerical Algorithms

Last update: ~~15/12/2020~~, 19/12/2021

Credit: ITCS 4/5145 Parallel Computing UNC-Charlotte, B. Wilkinson, 2009.

Numerical Algorithms

- Matrix multiplication
- Solving a system of linear equations

Matrices — A Review

An *n x m* matrix

$$\begin{array}{cc}
 & \xrightarrow{\text{Column}} \\
\text{Row}\Big\downarrow & \begin{bmatrix}
a_{0,0} & a_{0,1} & \cdots & a_{0,m-2} & a_{0,m-1} \\
a_{1,0} & a_{1,1} & \cdots & a_{1,m-2} & a_{1,m-1} \\
\vdots & \vdots & & \vdots & \vdots \\
a_{n-2,0} & a_{n-2,1} & \cdots & a_{n-2,m-2} & a_{n-2,m-1} \\
a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,m-2} & a_{n-1,m-1}
\end{bmatrix}
\end{array}$$

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

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

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

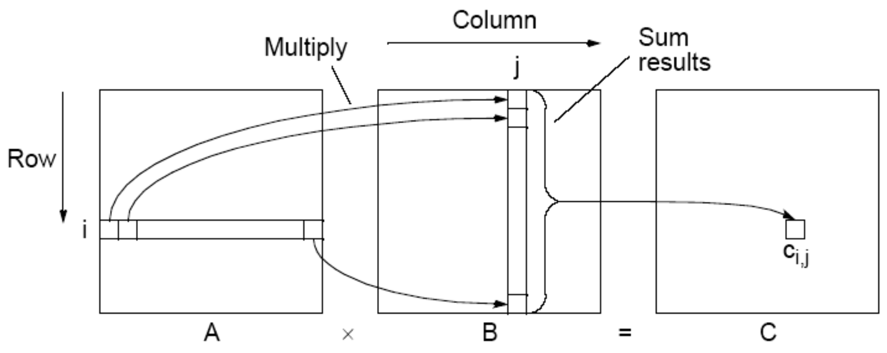
Matrix Multiplication

Multiplication of two matrices, **A** and **B**, produces 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.

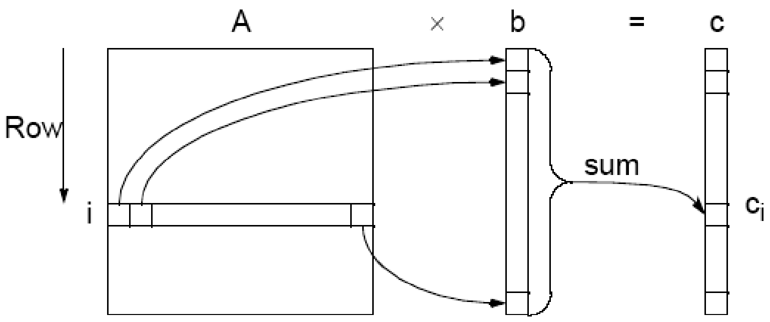
Matrix multiplication, C = A x B



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:

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

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

x is a vector of the unknowns

b is a vector of the *b* constants.

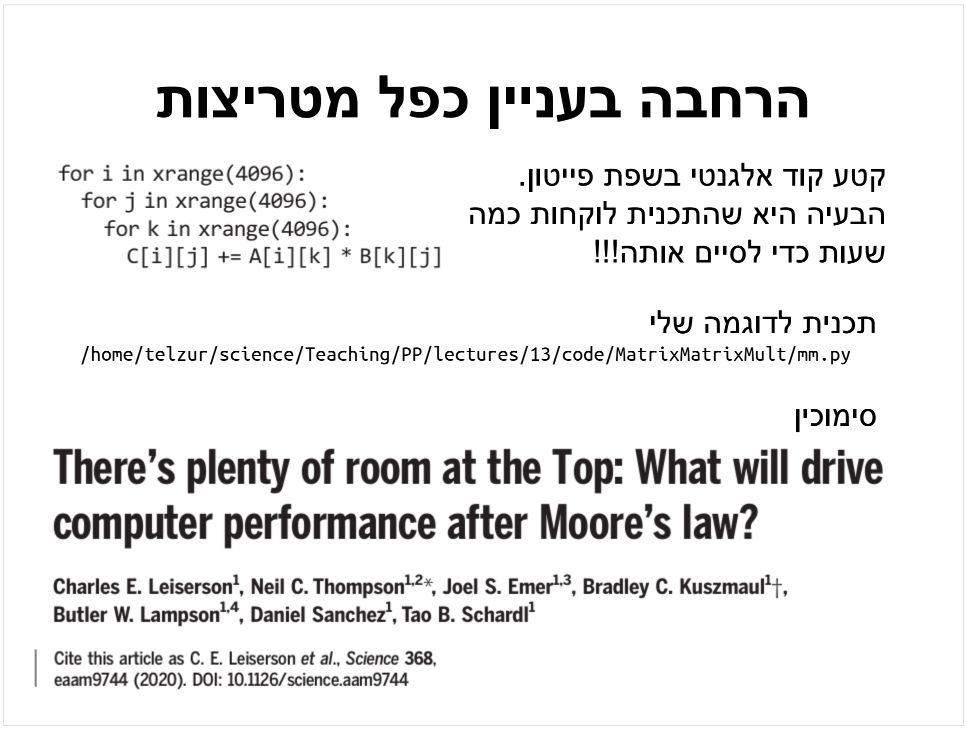
Implementing Matrix Multiplication Sequential Code

Assume throughout that matrices square (*n* x *n* matrices).

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];
}
```

Requires n^3 multiplications and n^3 additions
Sequential time complexity of $O(n^3)$.
Very easy to parallelize.



הרחבה בעניין כפל מטריצות

קטע קוד אלגנטי בשפת פייטון.
הבעיה היא שהתכנית לוקחות כמה שעות כדי לסיים אותה!!!

```
for i in xrange(4096):
  for j in xrange(4096):
    for k in xrange(4096):
      C[i][j] += A[i][k] * B[k][j]
```

תכנית לדוגמה שלי

```
/home/telzur/science/Teaching/PP/lectures/13/code/MatrixMatrixMult/mm.py
```

סימוכין

There's plenty of room at the Top: What will drive computer performance after Moore's law?

Charles E. Leiserson[1], Neil C. Thompson[1,2]*, Joel S. Emer[1,3], Bradley C. Kuszmaul[1]†, Butler W. Lampson[1,4], Daniel Sanchez[1], Tao B. Schardl[1]

Cite this article as C. E. Leiserson *et al.*, *Science* **368**, eaam9744 (2020). DOI: 10.1126/science.aam9744

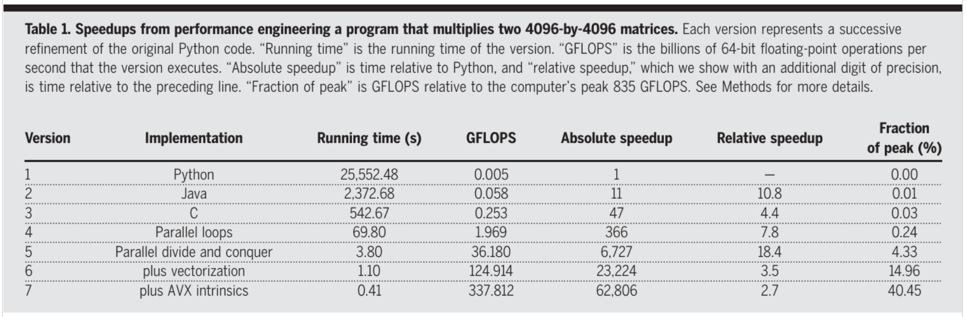
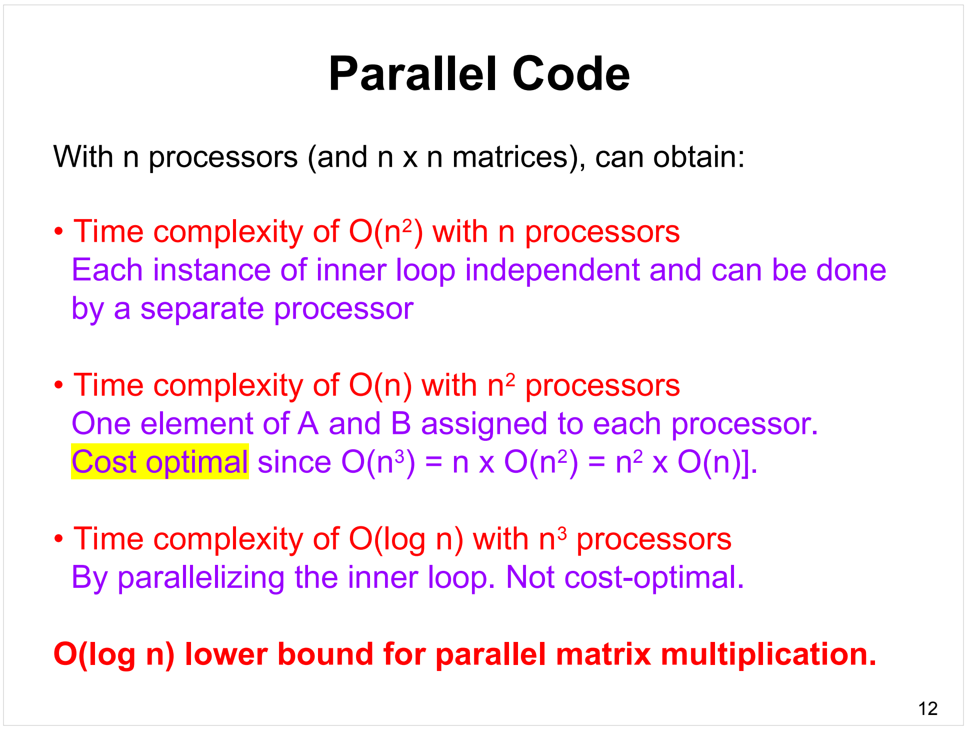


Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. “Running time” is the running time of the version. “GFLOPS” is the billions of 64-bit floating-point operations per second that the version executes. “Absolute speedup” is time relative to Python, and “relative speedup,” which we show with an additional digit of precision, is time relative to the preceding line. “Fraction of peak” is GFLOPS relative to the computer’s peak 835 GFLOPS. See Methods for more details.

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

מתוך הסימוכין מהשקף הקודם



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.

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

12

With n^2: paralllelize the two outer loops

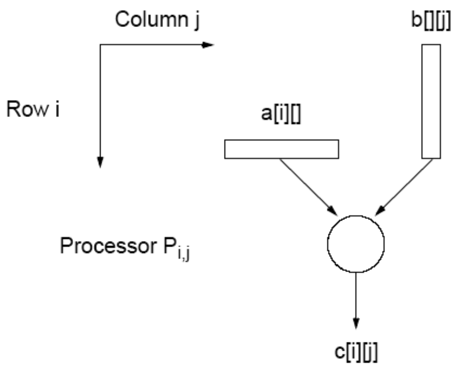
With n^3: there are n^2 elements. Each element is computed by the summation of the rowA X column B which can be parallelized with complexity log(n) because with n processors this work is like a computation over a binary tree

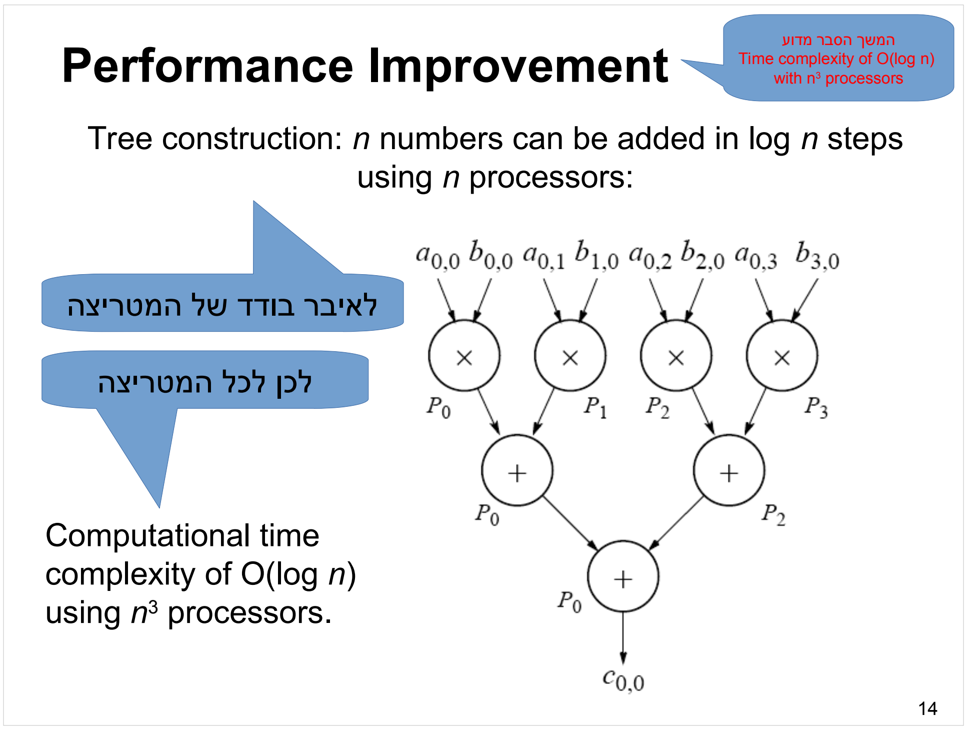
Guy

Direct Implementation

הסבר מדוע
Time complexity of O(log n)
with n^3 processors

יש במטריצה 2**n אלמנטים, כל אלמנט כזה ניתן לחישוב במקביל
עם n מעבדים, סה"כ n כפול 2**n





Performance Improvement

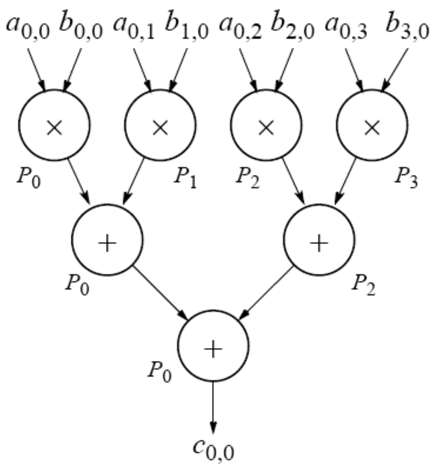
המשך הסבר מדוע
Time complexity of O(log n) with n³ processors

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

לאיבר בודד של המטריצה

לכן לכל המטריצה

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


$a_{0,0}$ $b_{0,0}$ $a_{0,1}$ $b_{1,0}$ $a_{0,2}$ $b_{2,0}$ $a_{0,3}$ $b_{3,0}$
$\times$
P_0 P_1 P_2 P_3
$+$
P_0 P_2
P_0
$c_{0,0}$

14

לאיבר בודד של C אני נזקק ל- n מעבדים.

לכן לכל המטריצה שהיא nXn אזדקק ל- n^3 מעבדים.

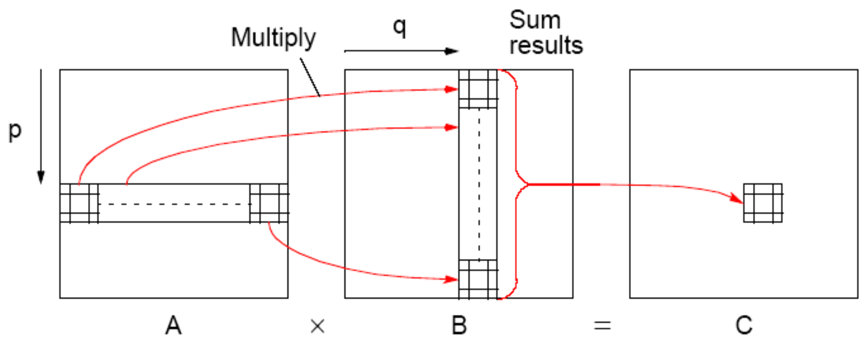
Partitioning into Submatrices

Suppose matrix divided into ***s*²** submatrices.
Each submatrix has *n*/*s* x *n*/*s* elements.
$A_{p,q}$ means submatrix row *p* and column *q* of matrix A:

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

The line: $\mathbf{C_{p,q} = C_{p,q} + A_{p,r} * B_{r,q};}$
means multiply submatrix $A_{p,r}$ and $B_{r,q}$ using matrix multiplication and add to submatrix $C_{p,q}$ 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 $C_{0,0}$

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

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

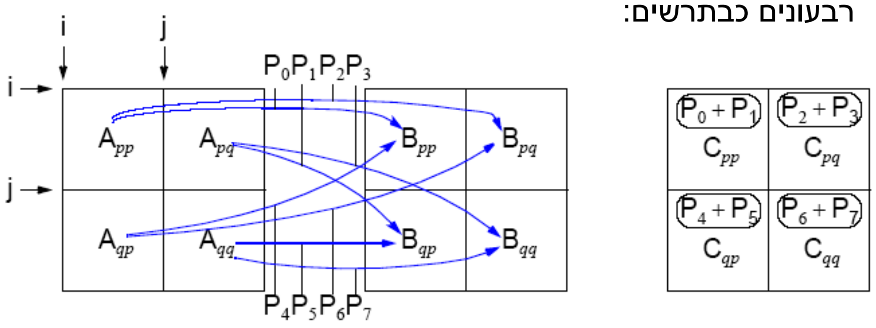
$$\begin{bmatrix} a_{0,0}b_{0,0}+a_{0,1}b_{1,0}+a_{0,2}b_{2,0}+a_{0,3}b_{3,0} & a_{0,0}b_{0,1}+a_{0,1}b_{1,1}+a_{0,2}b_{2,1}+a_{0,3}b_{3,1} \\ a_{1,0}b_{0,0}+a_{1,1}b_{1,0}+a_{1,2}b_{2,0}+a_{1,3}b_{3,0} & a_{1,0}b_{0,1}+a_{1,1}b_{1,1}+a_{1,2}b_{2,1}+a_{1,3}b_{3,1} \end{bmatrix}$$

$$= C_{0,0}$$

השקף מדגים את השקילות בין ביצוע כפל מטריצות איבר-איבר לבין כפל מטריצות באמצעות כפל תת-מטריצות

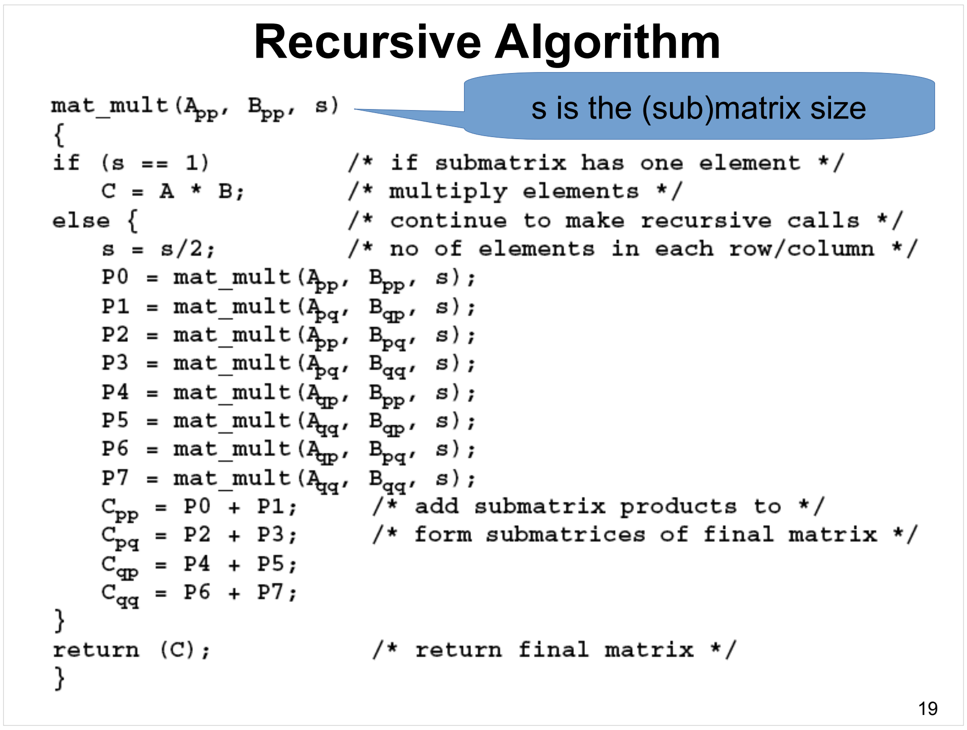
Recursive Implementation

עם 8 מעבדים ניתן לחשב <u>במקביל</u> כפל מטריצות המחולקות ל 4 רבעונים כבתרשים:



Apply same algorithm on each submatrix recursively.

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



Recursive Algorithm

s is the (sub)matrix size

```
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 */
}
```

הרחבה בעניין כפל מטריצות רקורסיבי

In the previous slide we saw that there are 8 multiplications and 4 additions.

Let’s estimate the computation time:
t~ 8M(n/2)+4A
Where: M stands for *Multiplication*, A stands for A*ddition*

MM(n/2) ~ O ((n/2)^3), 4A(n/2)~O(n^3).
(MM=Matrix Multiply)
According to Master theorem:
t ~ O(n^3) → there is no speedup in the divide and conquer MM method! למרות האלגנטיות, אין שיפור בסיבוכיות
Master theorem: https://brilliant.org/wiki/master-theorem/

The following 3 slides are taken from: CMPUT204
https://webdocs.cs.ualberta.ca/~mreza/courses/204-F07/lec-notes/week07.pdf

Strassen MM

- Want to use divide and conquer to speed things up; for simplicity assume n is a power of 2.

- Break each of X and Y into 4 submatrices of size $\frac{n}{2} \times \frac{n}{2}$ each:

$$\underbrace{\begin{bmatrix} A & B \\ C & D \end{bmatrix}}_{X} \underbrace{\begin{bmatrix} E & F \\ G & H \end{bmatrix}}_{Y} = \underbrace{\begin{bmatrix} I & J \\ K & L \end{bmatrix}}_{Z}$$

- Therefore:

$$\left.\begin{array}{l} I = AE + BG \\ J = AF + BH \\ K = CE + DG \\ L = CF + DH \end{array}\right\} \longrightarrow$$

need 8 multiplications of subproblems of size $\frac{n}{2}$ each

- We also need to spend $O(n^2)$ time to add up these results.

עד כאן סיכמנו מידע שכבר צויין בשקפים הקודמים של חישוב רקורסיבי

- If $T(n)$ is the time to multiply two matrices of size $n \times n$ each, then:
$$T(n) = 8T(\frac{n}{2}) + O(n^2)$$
- Using master theorem: $T(n) \in \Theta(n^{\log_2 8}) = \Theta(n^3)$.
- So this is as bad as the naive algorithm. No improvement yet.
- We use an idea similar to the one for multiplication of large integers: reduce the number of subproblems using a clever trick.
- compute the following 7 multiplications (each consisting of two subproblems of size $\frac{n}{2}$ each):
$$\begin{aligned} S_1 &= A(F-H) \\ S_2 &= (A+B)H \\ S_3 &= (C+D)E \\ S_4 &= D(G-E) \\ S_5 &= (A+D)(E+H) \\ S_6 &= (B-D)(G+H) \\ S_7 &= (A-C)(E+F) \end{aligned}$$

$$\begin{aligned}
I &= S_5 + S_6 + S_4 - S_2 \\
&= (A+D)(E+H) + (B-D)(G+H) + D(G-E) - (A+B)H \\
&= AE + DE + AH + DH + BG - DG + BH - DH + \\
&\qquad DG - DE - AH - BH \\
&= AE + BG
\end{aligned}$$

- Similarly, it can be verified easily that:

$$\begin{aligned}
J &= S_1 + S_2 \\
K &= S_3 + S_4 \\
L &= S_1 - S_7 - S_3 + S_5
\end{aligned}$$

- So to compute I, J, K, and L, we only need to compute $S_1, \ldots, S_7$; this requires solving seven subproblems of size $\frac{n}{2}$, plus a constant (at most 16) number of addition each taking $O(n^2)$ time.

$$T(n) = 7T(\frac{n}{2}) + O(n^2)$$

- Using master theorem and since $\log_2 7 \approx 2.808$:

$$T(n) \in O(n^{2.808})$$

- For $n = 50,000$: $n^3 \approx 10^{17}$ and $n^{2.808} \approx 10^{13}$; $\longrightarrow$ this algorithm is about 10,000 times faster than the naive algorithm.

For further reading:
"Introduction to Algorithms" by Thomas H. Cormen
Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.
3rd Edition, 2009.
Section 4.2

See also this video:
https://www.youtube.com/watch?v=E-QtwPi620I

Mesh Implementations

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

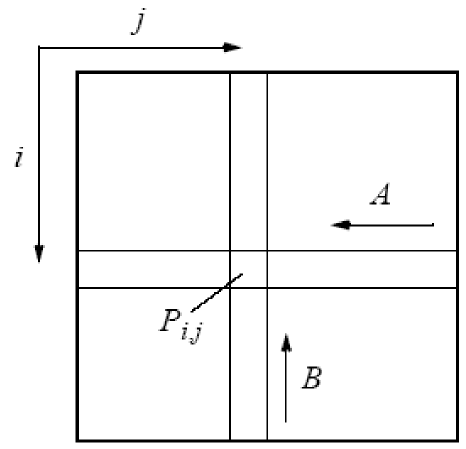
All involve using processor arranged a mesh and shifting elements of the arrays through the mesh. Accumulate the partial sums at each processor.

Mesh Implementations
Cannon's Algorithm

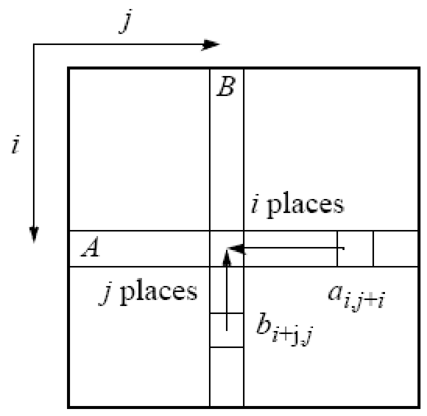
Mesh of processors with wraparound connections to shift the A elements (or submatrices) left and the B elements (or submatrices) up.

1. Initially processor $P_{i,j}$ has elements $a_{i,j}$ and $b_{i,j}$ (0 <= i < n, 0 <= k < n).
2. Elements moved from their initial position to an "aligned" position. *i*th row of A is shifted *i* places left and *j*th column of B is shifted *j* places upward. Has effect of placing element $a_{i,j+}$i and element $b_{i+j,j}$ in processor $P_{i,j}$. These elements are a pair of those required in accumulation of $c_{i,j}$.
3. Each processor, $P_{i,j}$, multiplies its elements.
4. The *i*th row of A is shifted one place right, and the *j*th column of B is shifted one place upward. Has the effect of bringing together adjacent elements of A and B, which will also be required in the accumulation.
5. Each processor, $P_{i,j}$, multiplies elements brought to it and adds result to accumulating sum.
6. Step 4 and 5 repeated until final result obtained (*n* - 1 shifts with *n* rows and *n* columns of elements).

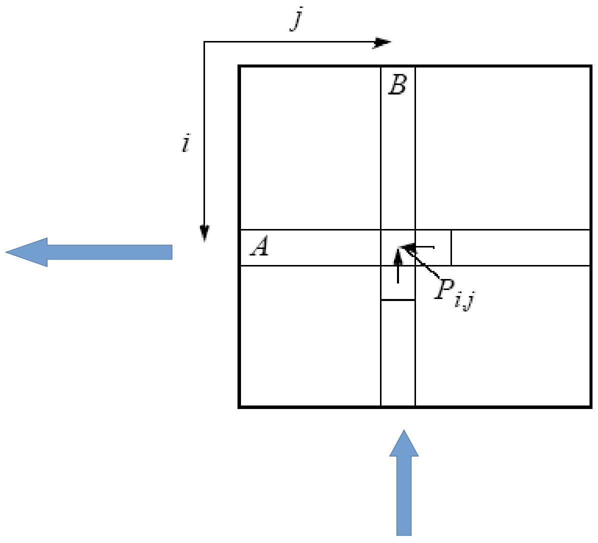
Movement of *A* and *B* elements



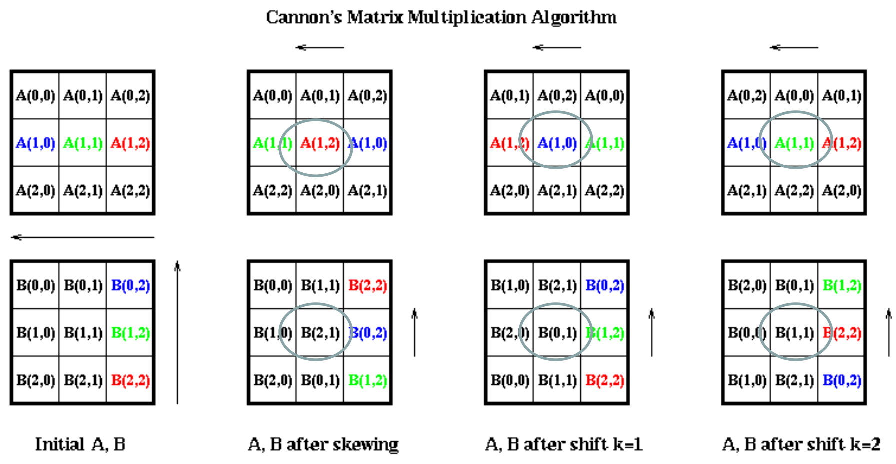
Step 2 — Alignment of elements of *A* and *B*



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



הפרוססורים נייחים והמספרים מוזזים



$C11 = a12 \cdot b21 + a10 \cdot b01 + a11 \cdot b11$

http://www.eecs.berkeley.edu/~demmel/cs267/lecture11/lecture11.html

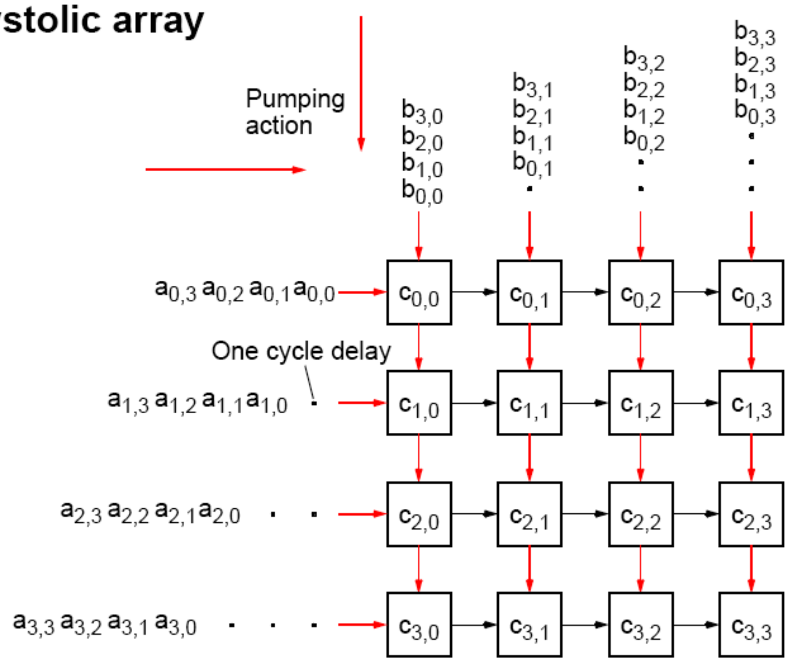
30

משמאל לימין. עמודה הכי שמאלית = מצב ראשוני. עמודה 2 משמאל: הבאה למצב התחלתי. 0 הזזה בשביל שורה ראשונה ב A ועמודה ראשונה של B, 1 הזזה לשורה 2 של A ועמודה 2 של B וכך הלאה. בעמודות שלוש וארבע יש בכל שלב הזזה אחד של כל השורות/עמודות. התוצאה הסופית היא סכום המכפלות שמוקפות בעיגולים

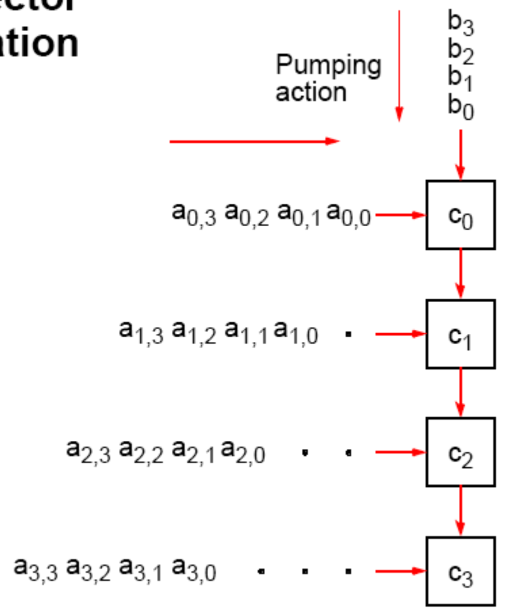
המחשות של האלגוריתם של קנון

https://www.dailymotion.com/video/x2vv5pm

Systolic array



Matrix-Vector Multiplication


Pumping action
b_3
b_2
b_1
b_0
$a_{0,3}\ a_{0,2}\ a_{0,1}\ a_{0,0}$
$a_{1,3}\ a_{1,2}\ a_{1,1}\ a_{1,0}$
$a_{2,3}\ a_{2,2}\ a_{2,1}\ a_{2,0}$
$a_{3,3}\ a_{3,2}\ a_{3,1}\ a_{3,0}$
c_0
c_1
c_2
c_3

Solving a System of Linear Equations

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

which, in matrix form, is

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

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

Solving 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. Then solve by Back Substitution.

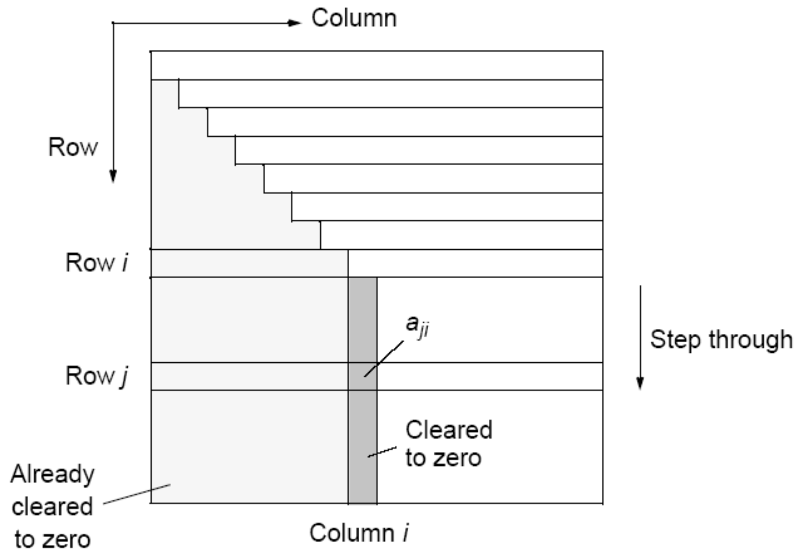
Uses characteristic of linear equations that any row can be replaced by that row added to another row multiplied by a constant.

Starts at first row and works toward bottom row. At *i*th row, each row *j* below *i*th row replaced by row *j* + (row *i*) (-*aj,i*/ *ai,i*).

Constant used for row *j* is -*aj,i*/*ai,i*. Has effect of making all elements in *i*th column below *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 *ai,i* is zero or close to zero, will not be able to compute quantity -*aj,i*/*ai,i*.

Procedure must be modified into so-called *partial pivoting* by swapping *i*th row with row below it that has largest absolute element in *i*th column of any of rows below *i*th row if there is one. (Reordering equations will not affect the system.)

In the following, we will not consider partial pivoting.

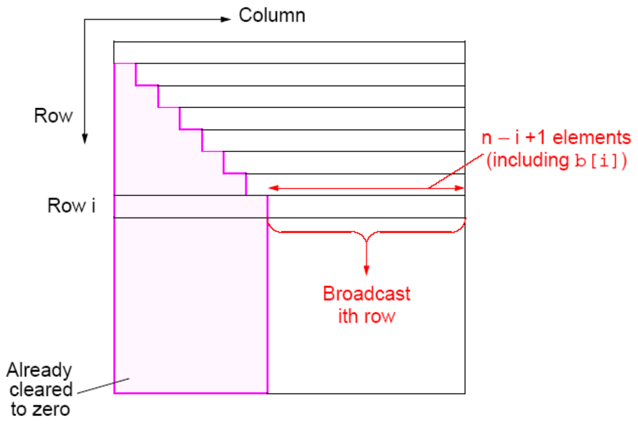
Sequential Code

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



Bcast
כדי לספק את
a[i][k]

לקריאה נוספת:

Designing and Building Parallel Programs:
https://www.mcs.anl.gov/~itf/dbpp/text/node90.html#SECTION03480000000000000000
UCB CS267:
https://people.eecs.berkeley.edu/~demmel/cs267_Spr16/Lectures/lecture13_densela_2_jwd16_4pp.pdf

Analysis

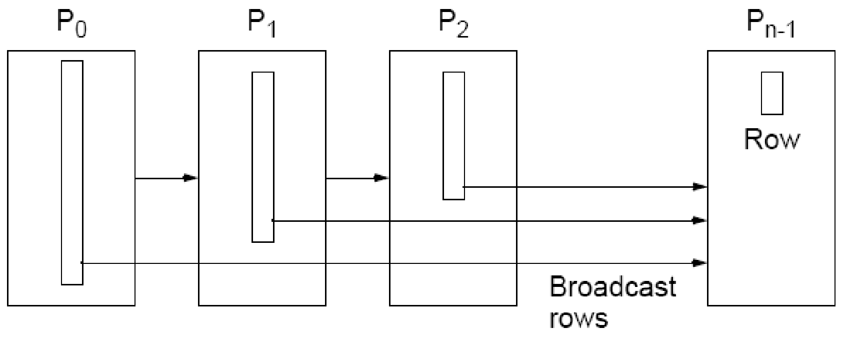
Communication

n - 1 broadcasts performed sequentially.
ith broadcast contains n - i + 1 elements.
Time complexity of O(n^2)   (see textbook)

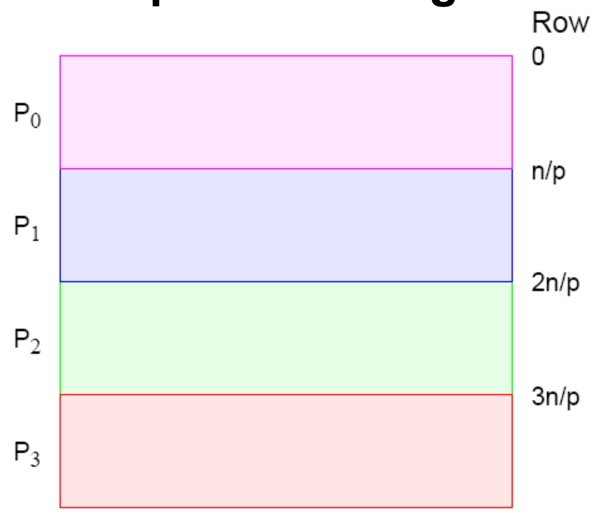
Computation

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 computation of multiplier, n - j + 2 multiplications and n - j + 2 subtractions.
Time complexity of O(n^2)   (see textbook).
Efficiency relatively low because all processors before processor holding row i do not participate in computation again.

Pipeline implementation of Gaussian elimination



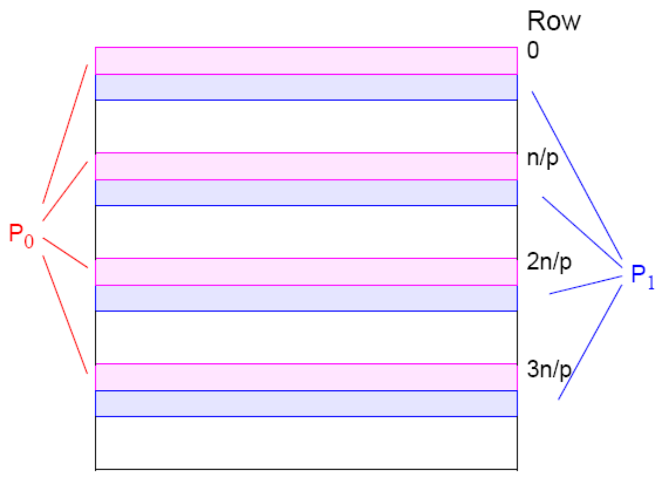
Strip Partitioning



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:

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

but can be less than 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 k*th 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

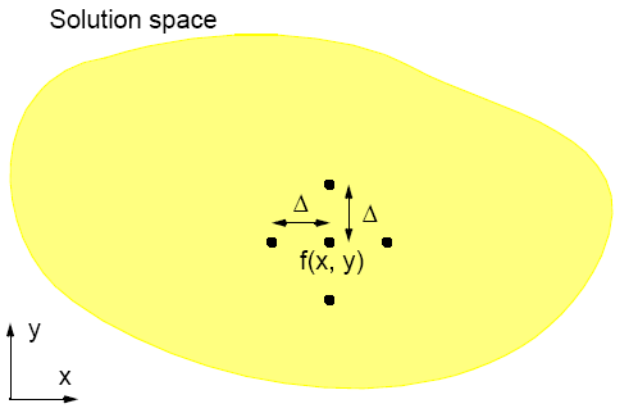
$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial \mathrm{y}^2} = 0$$

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

For computer solution, *finite difference* methods appropriate

Two-dimensional solution space "discretized" into 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

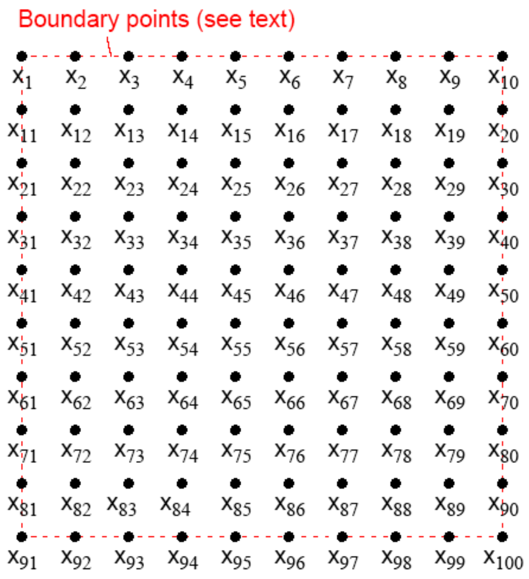
$$f(x, y) = \frac{[f(x-\Delta, y) + f(x, y-\Delta) + f(x+\Delta, y) + f(x, y+\Delta)]}{4}$$

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


Boundary points (see text)
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10}
x_{11} x_{12} x_{13} x_{14} x_{15} x_{16} x_{17} x_{18} x_{19} x_{20}
x_{21} x_{22} x_{23} x_{24} x_{25} x_{26} x_{27} x_{28} x_{29} x_{30}
x_{31} x_{32} x_{33} x_{34} x_{35} x_{36} x_{37} x_{38} x_{39} x_{40}
x_{41} x_{42} x_{43} x_{44} x_{45} x_{46} x_{47} x_{48} x_{49} x_{50}
x_{51} x_{52} x_{53} x_{54} x_{55} x_{56} x_{57} x_{58} x_{59} x_{60}
x_{61} x_{62} x_{63} x_{64} x_{65} x_{66} x_{67} x_{68} x_{69} x_{70}
x_{71} x_{72} x_{73} x_{74} x_{75} x_{76} x_{77} x_{78} x_{79} x_{80}
x_{81} x_{82} x_{83} x_{84} x_{85} x_{86} x_{87} x_{88} x_{89} x_{90}
x_{91} x_{92} x_{93} x_{94} x_{95} x_{96} x_{97} x_{98} x_{99} x_{100}

Relationship with a General System of Linear Equations

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

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

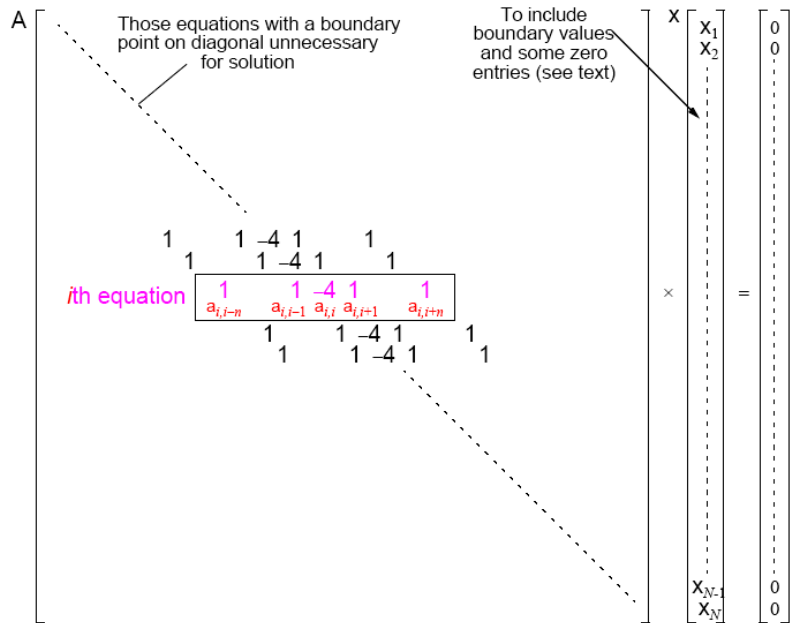
or

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

which is a linear equation with five unknowns (except those with boundary points).
In general form, the *i*th equation becomes:

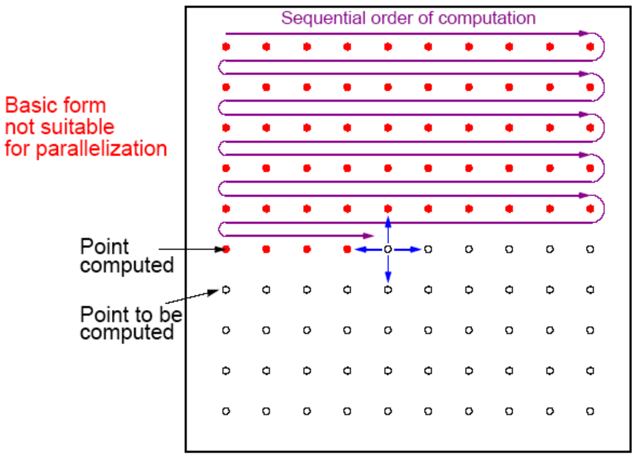
$$a_{i,i-n}x_{i-n} + a_{i,i-1}x_{i-1} + a_{i,i}x_i + a_{i,i+1}x_{i+1} + a_{i,i+n}x_{i+n} = 0$$

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
ith equation
1 1 –4 1 1
1 1 –4 1 1
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_1
x_2
x_{N-1}
x_N
=
0
0
0
0

Gauss-Seidel Relaxation

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


Sequential order of computation
Basic form not suitable for parallelization
Point computed
Point to be computed

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 superscript indicates iteration.

With natural ordering of unknowns, formula reduces to

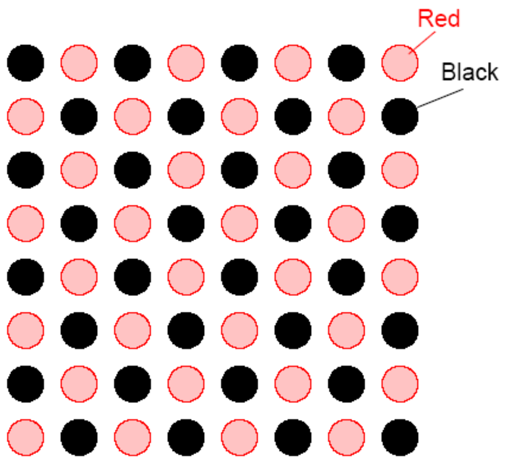
$$x_i^k = (-1/a_{i,i})[a_{i,i-n}x_{i-n}^k + a_{i,i-1}x_{i-1}^k + a_{i,i+1}x_{i+1}^{k-1} + a_{i,i+n}x_{i+n}^{k-1}]$$

At *k*th iteration, two of the four values (before *i*th element) taken from *k*th iteration and two values (after *i*th element) taken from (*k*-1)th iteration. 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]);
```

Higher-Order Difference Methods

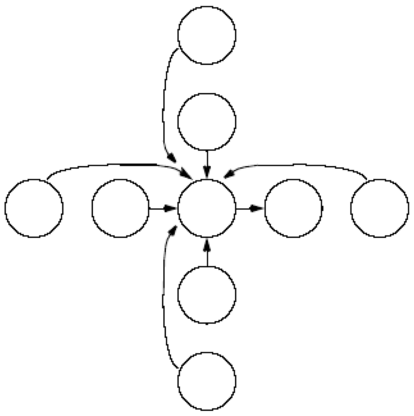
More distant points could be used in the computation. The following update formula:

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

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

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

Nine-point stencil



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.

Multigrid Method

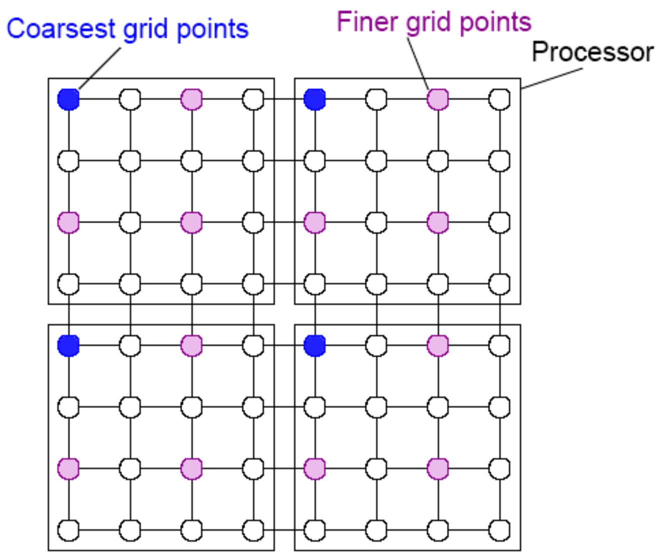
First, a coarse grid of points used. With these points, iteration process will start to converge quickly.

At some stage, number of points increased to include points of coarse grid and extra points between points of 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



(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.

62