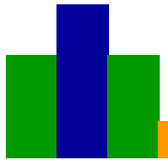
Lesson 10: Numerical algorithms

G Stefanescu — University of Bucharest

Parallel & Concurrent Programming
Fall, 2014



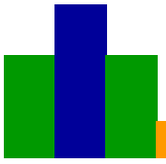
Matrices

Matrices:

- an *$m \times n$ matrix* A is a two-dimensional array with m rows and n columns (usually the entries are numbers); e.g.,

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

- we briefly denote a matrix by $A = (a_{i,j})_{0 \le i < m, 0 \le j < n}$ (when no confusion arises, the specification of the index range is omitted)



..Matrices

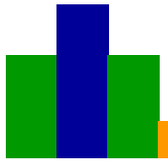
Operations on matrices:

- *addition*: given two matrices $A = (a_{i,j})$ and $B = (b_{i,j})$ of the same size $m \times n$, their sum $A + B$ is a matrix of the same size $C = (c_{i,j})$ defined by element-wise addition

$$c_{i,j} = a_{i,j} + b_{i,j}, \quad \forall 0 \leq i < m, 0 \leq j < n$$

- *multiplication*: let $A = (a_{i,j})$ be an $m \times n$ matrix and $B = (b_{i,j})$ be an $n \times p$ matrix; their product $A \cdot B$ is the $m \times p$ matrix $C = (c_{i,j})$ defined by "row-column multiplication"

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} \cdot b_{k,j}, \quad \forall 0 \leq i < m, 0 \leq j < p$$



..Matrices

(Operations on matrices, cont.)

- *matrix-vector multiplication*: given an $n \times n$ matrix $A = (a_{i,j})$ and a (column) vector $b = (b_i)_{0 \le i < n}$ (i.e., an $n \times 1$ matrix)

$$c = A \cdot b$$

(the matrix multiplication of A and b) is a column vector again;

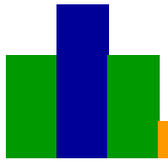
- a system of linear equations

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

may be written in a simple form

$$A \cdot x = b$$

for an appropriate matrix A and corresponding vectors x, b



Implementing matrix multiplication

Sequential code:

```
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];
  }
```

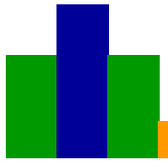
Sequential time complexity: $O(n^3)$ ($2n^3$ arithmetical operations)

Parallel matrix multiplication

Computational parallel time complexity:

- with n processors: $O(n^2)$
- with n^2 processors: $O(n)$
  (one processor for each pair (i, j))
- with n^3 processors (using a divide-and-conquer parallel procedure for the inner look): $O(log\ n)$

Notice: no communication time is included in these estimations; also, using n^3 processors to multiply $n \times n$ matrices is not a realistic assumption; usually, each processor handle blocks of sub-matrices



Partitioning into sub-matrices

Suppose a matrix is divided into s^2 sub-matrices, each having the same size $(n/s) \times (n/s)$.

Code for "block matrix multiplication": denote by $A_{p,q}$ the (p,q) sub-matrix of this partition; similarly for B and C; then

```
for (p=0; p<s; p++)
  for (q=0; q<s; q++) {
    C_{p,q} = 0;
    for (r=0; r<s; r++)
      C_{p,q} = C_{p,q} + A_{p,r} * B_{r,q};
  }
```

The line of code $C_{p,q} = C_{p,q} + A_{p,r} * B_{r,q}$ denotes operations involving multiplication and addition of sub-matrices.

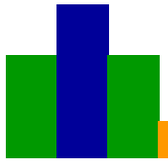
..Partitioning into sub-matrices

Example:

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

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

$= \ldots$



Direct implementation

Direct implementation, using n^2 processors:

- one process $P_{i,j}$ is used for each $C_{i,j}$
- process $P_{i,j}$ needs the i-th row of A and the j-th column of B
- data are separately sent to processes $P_{i,j}$
- alternatively, one may broadcast one row i to all processes $P_{i,j}(0 \leq j < n)$; similarly for columns

Analysis

Analysis (direct implementation, using n^2 processors):

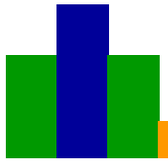
Communication

- with separate messages to each of the n^2 slave processes (including row, column, and result)
  $t_{comm} = n^2(t_s + 2nt_d) + n^2(t_s + t_d) = n^2(2t_s + (2n+1)t_d)$

- with a unique broadcast of the full matrices A and B
  $t_{comm} = (t_s + 2n^2t_d) + n^2(t_s + t_d)$
  (now, returning the results becomes dominant)

Computation

- $t_{comp} = 2n$ (n additions and n multiplications)



Performance improvement

Performance improvement:

- using a tree construction n numbers can be added in $log\ n$ steps using n processors
- computational time complexity decreases to $O(log\ n)$ using n^3 processors.

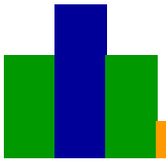
Recursive algorithm

- we have seen that block matrix multiplication is based on multiplication (and addition) of smaller matrices
- the same procedure may be used recursively to multiply these smaller matrices

Recursive algorithm

```
matMult(A, B, s) {
   if ( s == 1)
      C = A * B
   else {
      s = s/2;
      P0 = matMult (A_pp, B_pp, s);
      P1 = matMult (A_pq, B_qp, s);
      P2 = matMult (A_pp, B_pq, s);
      P3 = matMult (A_pq, B_qq, s);
      P4 = matMult (A_qp, B_pp, s);
      P5 = matMult (A_qq, B_qp, s);
      P6 = matMult (A_qp, B_pq, s);
      P7 = matMult (A_qq, B_qq, s);
      C_pp = P0 + P1;
      C_pq = P2 + P3;
      C_qp = P4 + P5;
      C_qq = P6 + P7;
   }
}
```

Notice: `s` *denotes the size of the blocks (number of rows). The number of processes is* 8^k*, so typically one takes 8 processes.*



Mash implementations / Cannon's algorithm

Cannon's algorithm: use a *torus* (mash with wraparound connections) and shift the elements/blocks of A left and those of B up.

1. initially processor $P_{i,j}$ has elements $a_{i,j}$ and $b_{i,j}$ $(0 \leq i, j < n)$
2. shift i-th row i places left and j-th column j places upward (the effect is that $P_{i,j}$ holds $a_{i,(j+i \bmod n)}$ and $b_{(j+i \bmod n),j}$ which are used for computing $c_{i,j}$)
3. each processor $P_{i,j}$ multiplies its elements
4. the i-th row of A is shifted one place left and the j-th column of B is shifted one place upward
5. each processor $P_{i,j}$ multiplies its elements and adds the result to the accumulating sum
6. the last 2 steps are repeated till the final result is obtained.

..Cannon's algorithm

Cannon's algorithm, example: $A = \begin{array}{|c|c|c|}\hline 1&2&3\\\hline 4&5&6\\\hline 7&8&9\\\hline\end{array}$; $B = \begin{array}{|c|c|c|}\hline a&b&c\\\hline d&e&f\\\hline g&h&i\\\hline\end{array}$

After initial shift,

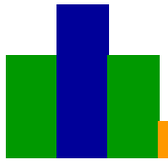
$A \rightarrow \begin{array}{|c|c|c|}\hline 1&2&3\\\hline 5&6&4\\\hline 9&7&8\\\hline\end{array}$; $B \rightarrow \begin{array}{|c|c|c|}\hline a&e&i\\\hline d&h&c\\\hline g&b&f\\\hline\end{array}$; so $C \rightarrow \begin{array}{|c|c|c|}\hline 1a&2e&3i\\\hline 5d&6h&4c\\\hline 9g&7b&8f\\\hline\end{array}$

2nd shift,

$A \rightarrow \begin{array}{|c|c|c|}\hline 2&3&1\\\hline 6&4&5\\\hline 7&8&9\\\hline\end{array}$; $B \rightarrow \begin{array}{|c|c|c|}\hline d&h&c\\\hline g&b&f\\\hline a&e&i\\\hline\end{array}$; so $C \rightarrow \begin{array}{|c|c|c|}\hline 1a+2d&2e+3h&3i+1c\\\hline 5d+6g&6h+4b&4c+5f\\\hline 9g+7a&7b+8e&8f+9i\\\hline\end{array}$

3rd shift,

$A \rightarrow \begin{array}{|c|c|c|}\hline 3&1&2\\\hline 4&5&6\\\hline 8&9&7\\\hline\end{array}$; $B \rightarrow \begin{array}{|c|c|c|}\hline g&b&f\\\hline a&e&i\\\hline d&h&c\\\hline\end{array}$; so $C \rightarrow \begin{array}{|c|c|c|}\hline 1a+2d+3g&2e+3h+1b&3i+1c+2f\\\hline 5d+6g+4a&6h+4b+5e&4c+5f+6i\\\hline 9g+7a+8d&7b+8e+9h&8f+9i+7c\\\hline\end{array}$



Analysis / Cannon's algorithm

Analysis / Cannon's algorithm: suppose s^2 blocks of size $m \times m$ are used (hence $n = ms$)

Communication

- initial alignment requires maximum $s-1$ shifts (communication operations)
- after that, there will be $s-1$ more shift operations
- each shift operation involves $m \times m$ matrices, hence

$$t_{comm} = 2(s-1)(t_{startup} + m^2 t_{data})$$

- the communication time complexity is $O(sm^2)$ or $O(mn)$

..Analysis

Computation

- each sub-matrix multiplication (done on a processor) requires m^3 multiplications and m^3 additions; the final summation is lighter: m^2 additions
- as there are $s-1$ shifts and an initial multiplication step

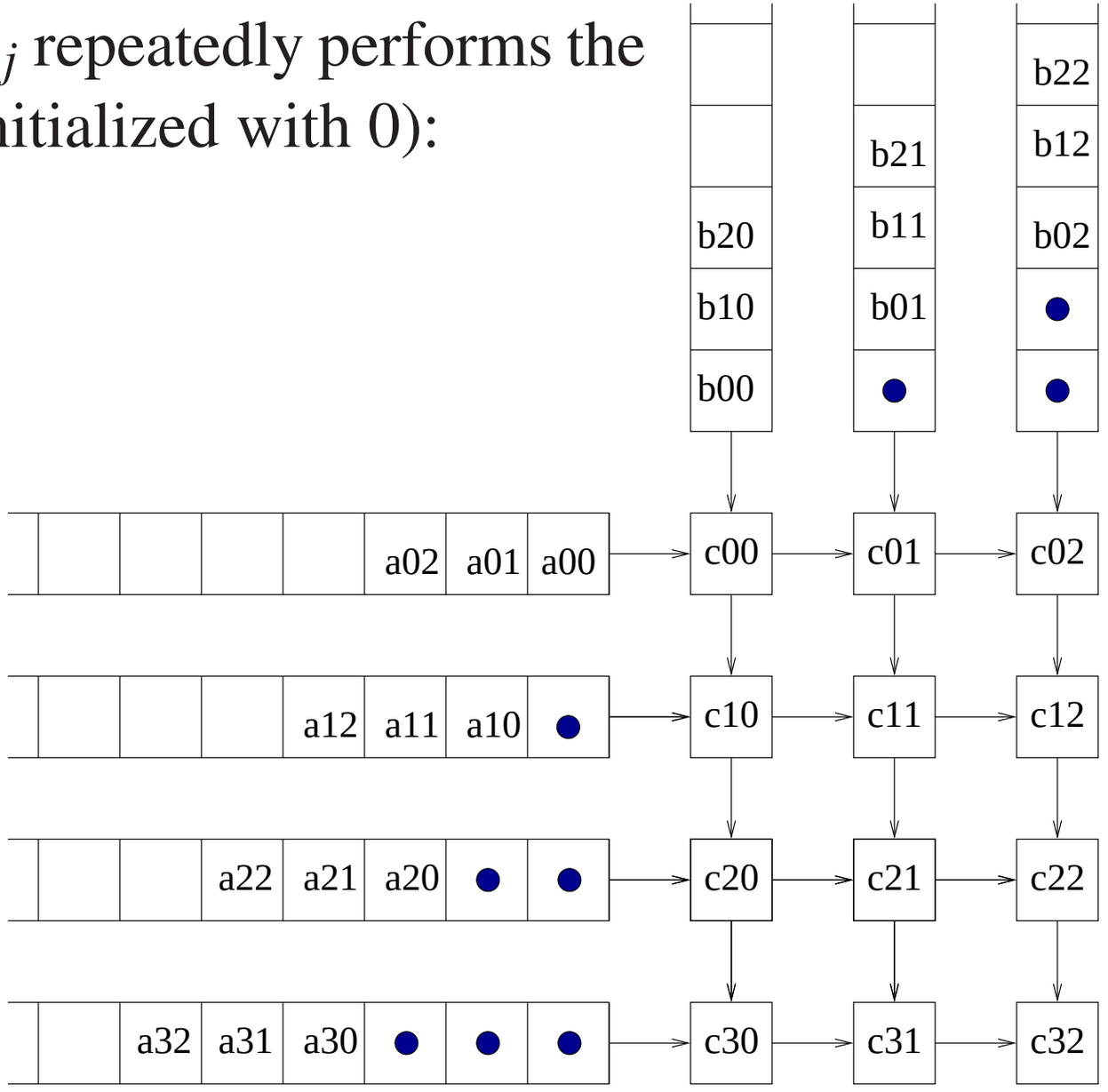
$$t_{comp} = 2s\, m^3$$

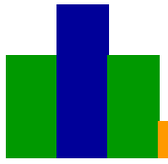
- hence the computational time complexity is $O(m^2 n)$

Systolic array

Code: each processor $P_{i,j}$ repeatedly performs the following actions (`c` is initialized with 0):

```
recv(&a, P_{i,j-1});
recv(&b, P_{i-1,j});
c = c + a * b;
send(&a, P_{i,j+1});
send(&b, P_{i+1,j});
```





Systems of linear equations

Systems of linear equations:

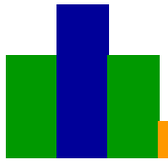
- a general system of linear equations has the following form

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

- the matrix form is

$$Ax = b$$

- the goal is to find the unknowns of vector x, given values for matrix A and vector b.



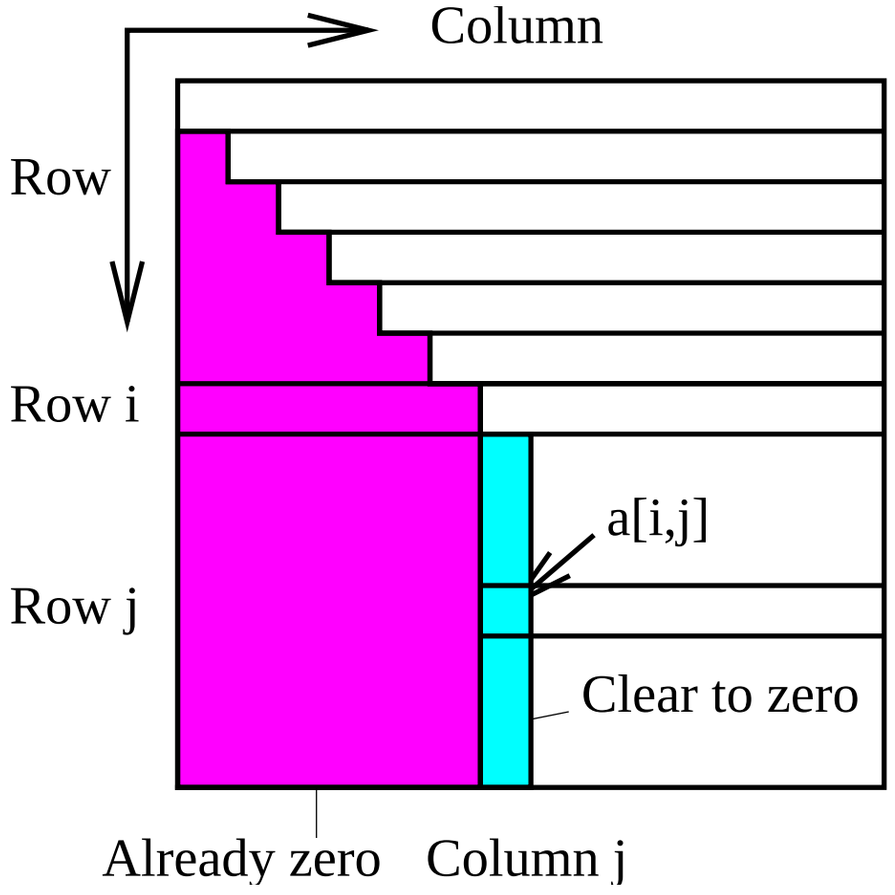
Gaussian elimination

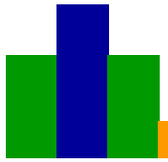
Gaussian elimination:

- convert the general system of linear equations into a *triangular system* of equations, then use *back-substitution* to solve it
- key property: the solution is not changed if any row is replaced by that row added with another row multiplied by a constant
- this leads to the following procedure:
  —start form the first row and work toward the bottom row
  —at the i-th row, each row j below it is replaced by *row* j + (*row* i)$(-a_{j,i}/a_{i,i})$; i.e., $a_{j,k} := a_{j,k} + a_{i,k}\left(\frac{-a_{j,i}}{a_{i,i}}\right)$, for all $k \geq i$
  —this way all elements of column i, below the i-th row, become zero; indeed, $a_{j,i} + a_{i,i}\left(\frac{-a_{j,i}}{a_{i,i}}\right) = 0$

..Gaussian elimination

Gaussian elimination:



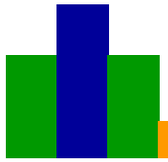


Partial pivoting

Partial pivoting

- if $a_{i,i}$ is zero or close to zero, the value $-a_{j,i}/a_{i,i}$ can not be accurately computed
- a *partial pivoting* technique may be used to avoid such a situation:
  - swap the i-th row with the row below it that has the *largest* element (absolute value) in column i from all row below the i-th row

(reordering of equations does not affect the system - notice that each row i includes the corresponding free term b_i)



Sequential code

Sequential code

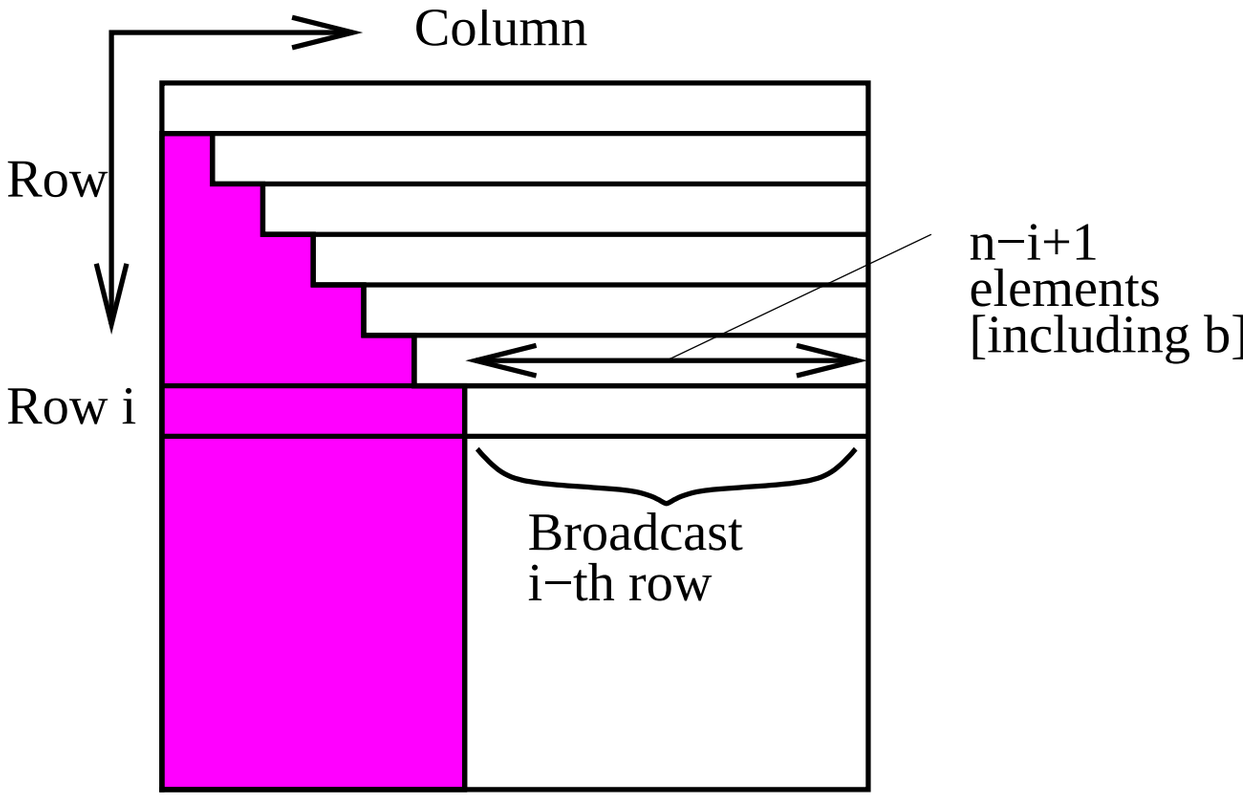
```
for (i=0; i<n-1; i++)
  for (j=i+1; j<n; j++) {
    m = a[j][i]/a[i][i];
    for (k=i; k<n; k++)
      a[j][k] = a[j][k] - a[i][k] * m;
    b[j] = b[j] - b[i] * m;
  }
```

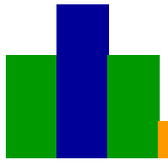
The time complexity if $O(n^3)$

(No pivoting method is considered here.)

..Gaussian elimination

Gaussian elimination (parallel):



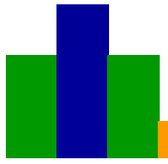


Analysis, parallel algorithm

Analysis, parallel algorithm:

Communication

- using n processes (one for each row)
- using broadcast to pass row i (namely, `a[i][i..n-1]`) and b_i to processes $i+1,..,n-1$; we have to pass $n-i+1$ data
- hence,
  $t_{comm} = \sum_{i=0}^{n-2}(t_{s-bcast} + (n-i+1)t_{d-bcast})$
  $= (n-1)t_{s-bcast} + (3+4+..+(n+1))t_{d-bcast}$
  $= (n-1)t_{s-bcast} + ((n+1)(n+2)/2-3)t_{d-bcast}$
- the communication time complexity is $O(n^2)$



..Analysis, parallel algorithm

Computation:

- after broadcast, each process (below i) computes the multiplier and then update $n-i+1$ elements - each updating requires 2 operations
- hence,
  $t_{comp} = \sum_{i=1}^{n-1}(2(n-i+1)+1)$
  $= n-1+2(n(n+1)/2-1) = O(n^2)$

Notice: For this algorithm processes may be mapped onto a pipeline configuration where different stages of computation may overlap.

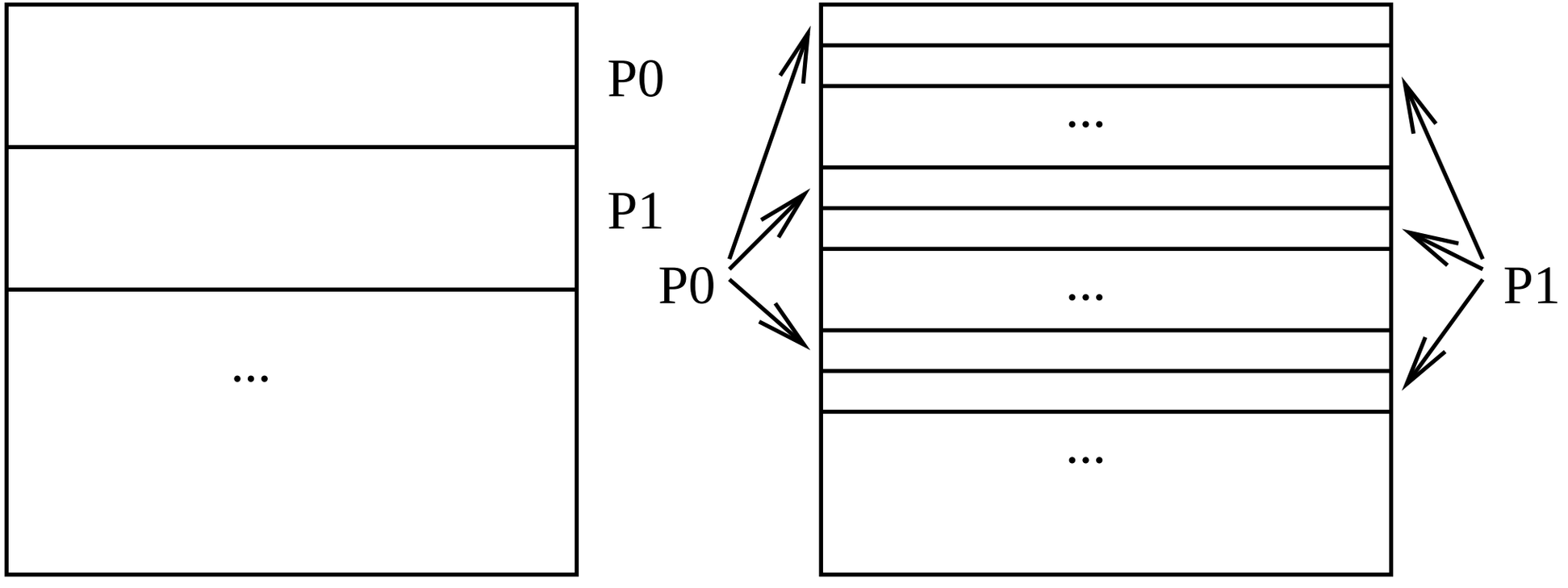
Partitioning

Partitioning:

- *strip partitions* may be used when less processors are available
- ... but the work is not well balanced (the processors handling the top rows do less work - they do not participate in computation after their last row is processed)
- *cyclic-strip partitions* provide more balanced work:
  if there are p processes, in this strategy
  —$P0$ handles rows $0, n/p, 2n/p, ..,$
  —$P1$ rows $1, n/p+1, 2n/p+1, ..,$
  —etc.

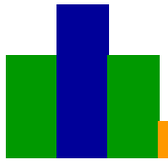
..higher order difference methods

Cyclic vs. strip partitions:



Strip partitions

Cyclic partitions



Iterative methods

Jacobi iteration:

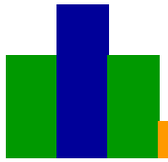
- by rearranging the equations of a system of linear equations we get

$$x_i = \frac{1}{a_{i,i}} \left(b_i - \sum_{j \neq i} a_{i,j} x_j \right)$$

- this relation may be used to get a approximating iterative method

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

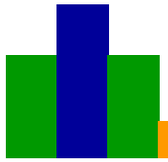
where the superscript indicates iteration:
—x_i^k is x_i at the k-th iteration;
—it is computed using the x_j's $(j \neq i)$ from the previous iteration



..Iterative methods

Rough comparison (direct vs. iterative method):

- the direct method mainly requires $O(n^2)$ computation steps, when n processors are to be used
- in the iterative algorithm with n processors, one iteration is relatively light: $n+1$ arithmetical operations are used;
- ... but the overall computation time complexity depends on the *number of iterations* needed to get the result with the required accuracy
- iterative methods are particularly useful for *sparse linear equations* (i.e., when there are many variables, but each equation constrains only a few of them)



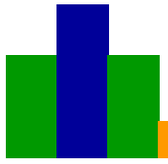
Laplace's equation

Laplace's equation:

- if f is a function with two arguments x and y, the aim is to solve

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

- usually, this is to be solved within a finite area of a two-dimensional vector space, knowing the values at the border
- *finite difference* method is generally used for a computer solution: the space is *discretized* and Laplace's equation is transformed into a system of linear equations



..Laplace's equation

(Laplace's equation, cont.)

- if the distance between neighboring points in both x and y directions is small, say Δ, then

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

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

- use these approximations into Laplace's equation; one gets

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

- this leads to the following iterative formula

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

for computing the values $f^k(x,y)$ at the k-th iteration using the values $f^{k-1}(..,..)$ of the previous iteration

..Laplace's equation

The result is just a system of linear equations. Indeed,

- for a square area with $n \times n$ points and the natural order (left-to-right, top to bottom), the equations are

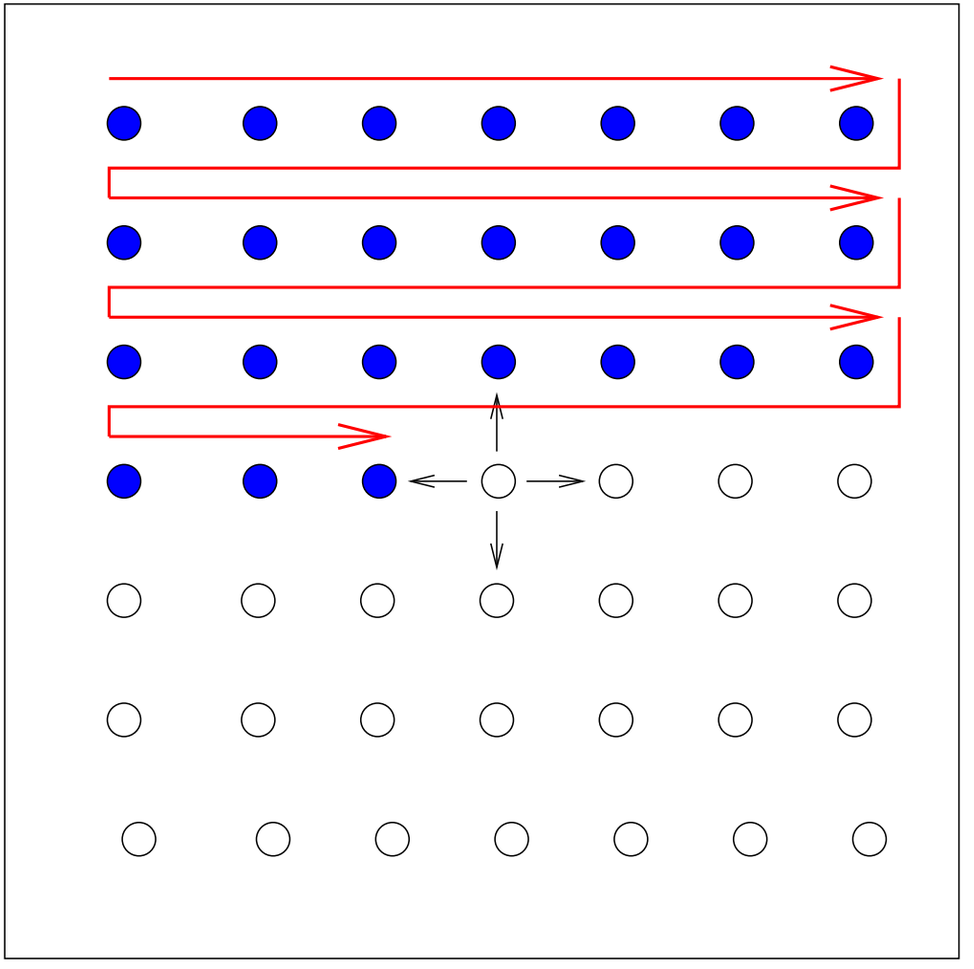
$$x_{i-n} + x_{i-1} - 4x_i + x_{i+1} + x_{i+n} = 0$$

- with usual pair-notation for indexes, the equations may be written as

$$x_{i-1,j} + x_{i,j-1} - 4x_{i,j} + x_{i,j+1} + x_{i+1,j} = 0$$

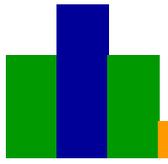
- near the border some variables are replaced by the corresponding known values (constants); these are collected in the constant b_i's terms of the system
- a system of sparse equations is obtained

Faster convergence methods



Gauss-Seidel relaxation:

Notice: This order is for a sequential approach; for parallel methods other orders may be used.



..Gauss-Seidel relaxation

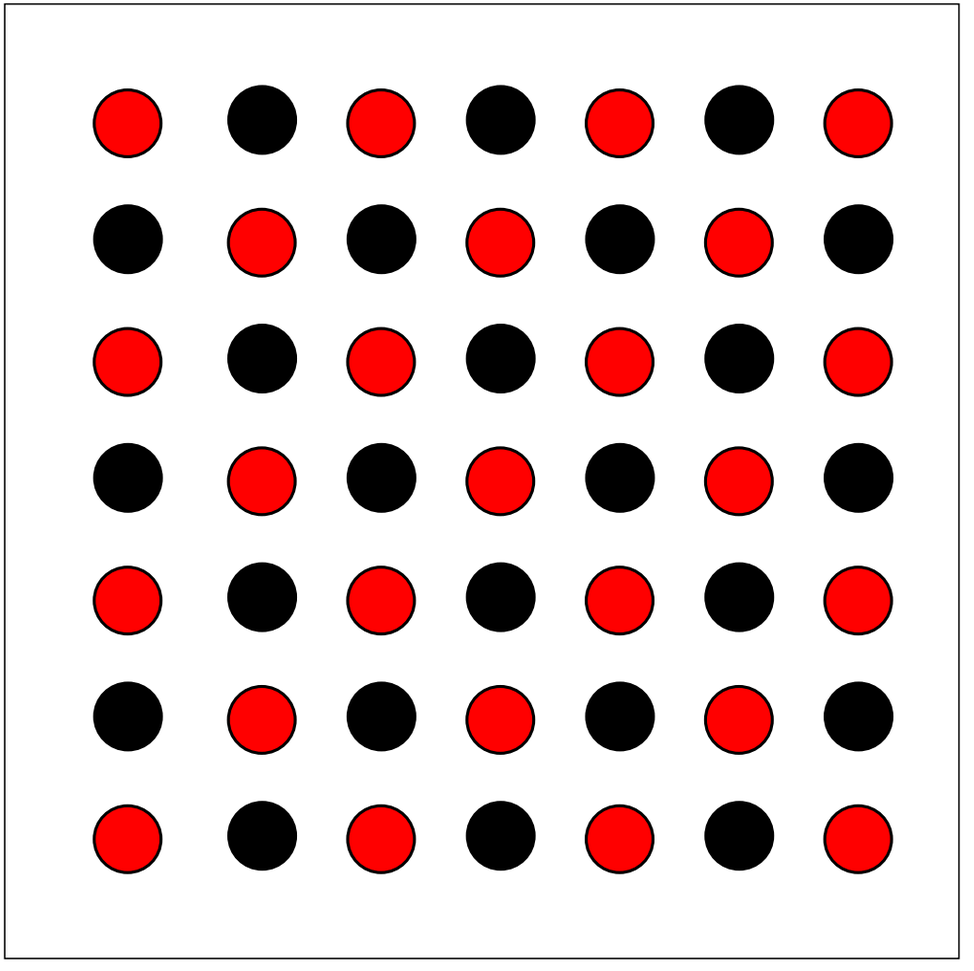
- this is a particular iterative method for system of linear equations
- mix old and newly computed values, according to the following scheme

$$x_i^k = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=0}^{i-1} a_{i,j} x_j^k - \sum_{j=i+1}^{n-1} a_{i,j} x_j^{k-1} \right)$$

- for Laplace's equation, this scheme produces the following result

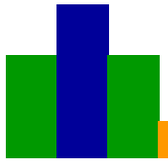
$$f^k(x,y) = (1/4)(f^k(x-\Delta,y) + f^k(x,y-\Delta) + f^{k-1}(x+\Delta,y) + f^{k-1}(x,y+\Delta))$$

Relaxation / Red-black ordering



Relaxation / Red-black ordering:

Notice: This order is for a parallel approach.



Red-black parallel code

Red-black parallel code:

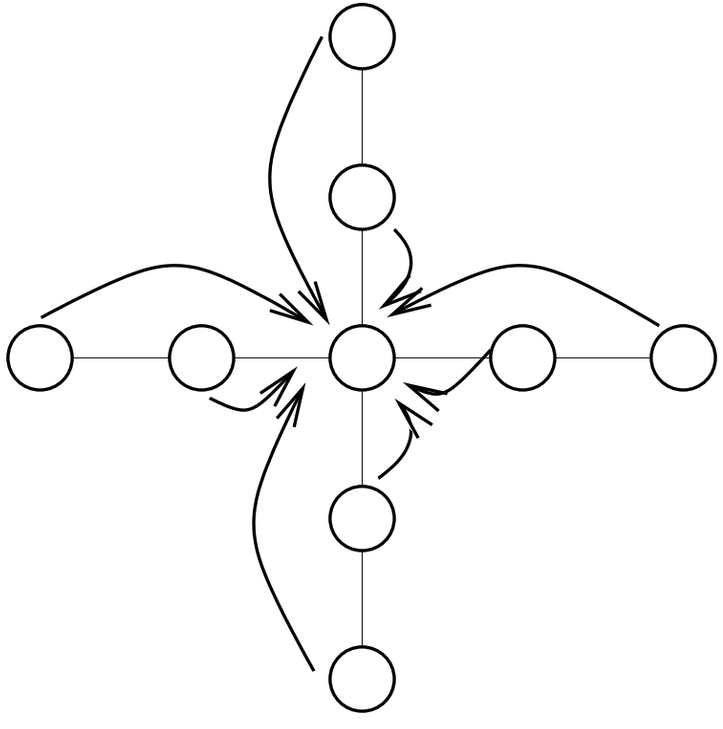
```
forall (i=1; i<n; i++)
  forall (j=1; j<n; j++)
    if ((i+j)%2 == 0)
      f[i][j] = 0.25 * (f[i-1][j] + f[i][j-1]
                        + f[i+1][j] + f[i][j+1]);
forall (i=1; i<n; i++)
  forall (j=1; j<n; j++)
    if ((i+j)%2 != 0)
      f[i][j] = 0.25 * (f[i-1][j] + f[i][j-1]
              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

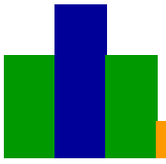
Higher order difference methods

In this case more distant points have to be used in the computation; an example is:

$$f^k(x,y) = (1/60)[16f^{k-1}(x-\Delta,y) + 16f^{k-1}(x,y-\Delta) + 16f^{k-1}(x+\Delta,y) + 16f^{k-1}(x,y+\Delta) - f^{k-1}(x-2\Delta,y) - f^{k-1}(x,y-2\Delta) - f^{k-1}(x+2\Delta,y) - f^{k-1}(x,y+2\Delta)]$$

—in this scheme eight nearby points (two points in each directions) are used to update a point





Over-relaxation

Over-relaxation:

- an improved convergence is obtained by including the values of the current point, i.e., adding a factor $(1-\omega)x_i$ to Jacobi or Gauss-Seidel formulas; ω is called *over-relaxation parameter*
- *Jacobi over-relaxation formula* is

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

  where $0 < \omega < 1$.

..Over-relaxation

(Over-relaxation, cont.)

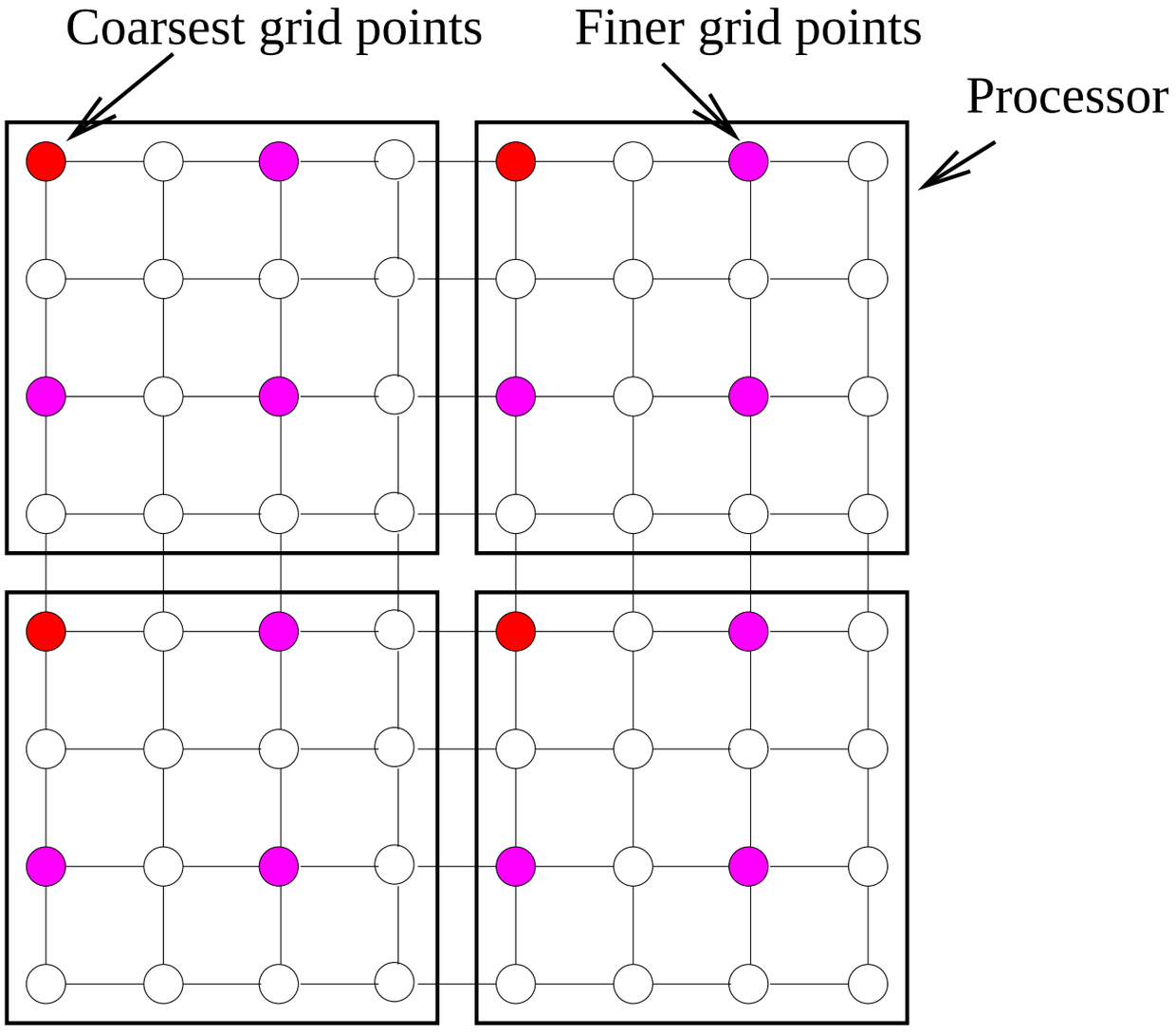
- *Gauss-Seidel over-relaxation formula* is

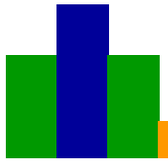
$$x_i^k = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=0}^{i-1} a_{i,j} x_j^k - \sum_{j=i+1}^{n-1} a_{i,j} x_j^{k-1} \right) + (1 - \omega) x_i^{k-1}$$

where $0 < \omega \leq 2$ (when $\omega = 1$, Gauss-Seidel method is obtained)

Multi-grid method

Multi-grid processor allocation:





..Multi-grid method

Multi-grid method:

- first, a *coarse grid* of points is used
  - with these points, the iteration process will start to *converge quickly*
  - .. but only a rough approximation for the starting problem is obtained
- at some stage
  - the *number of points* is *increasing*, including extra point between the points of the coarse grid;
  - the initial values of these extra points are obtained by *interpolation*;
  - the computation continues in this finer grid

..Multi-grid method

(Multi-grid method, cont.)

- the grid may be made finer and finer, or computation may alternate between fine and coarse grids
- the coarse grid takes into account distant effect more quickly and provide a good starting point for the next finer grid