9th SimLab Short Course on

Parallel Numerical Simulation

Belgrade, October 3 to 9, 2010

Examples of Parallel Algorithms

October 5, 2010

Vasco Varduhn

Chair for Computation in Engineering
Technische Universität München, Germany
International Graduate School of Science and Engineering (IGSSE)

Stabilitätspakt für Südosteuropa
Gefördert durch Deutschland
Stability Pact for South Eastern Europe
Sponsored by Germany

1. Matrix-Matrix and Matrix-Vector Operations

- underlying basis for many scientific problems is a matrix
- stored as 2-dimensional array of numbers (integer, float, double)
  - row-wise in memory (typical case)
  - column-wise in memory
- typical matrix operations (K: set of numbers)
  1) $A + B = C$ with A, B, and $C \in K^{N \times M}$
  2) $A \cdot b = c$ with $A \in K^{N \times M}$, $b \in K^{M}$, and $c \in K^{N}$
  3) $A \cdot B = C$ with $A \in K^{N \times M}$, $B \in K^{M \times L}$, and $C \in K^{N \times L}$
- matrix-vector multiplication (2) and matrix multiplication (3) are main building blocks of numerical algorithms
  - both pretty easy to implement as sequential code
  - what happens in parallel?

1.1 Matrix-Vector Multiplication

- Appearances
  - systems of linear equations (SLE) $A \cdot x = b$
  - iterative methods for solving SLEs (conjugate gradient, e. g.)
  - implementation of neural networks (determination of output values, training neural networks)

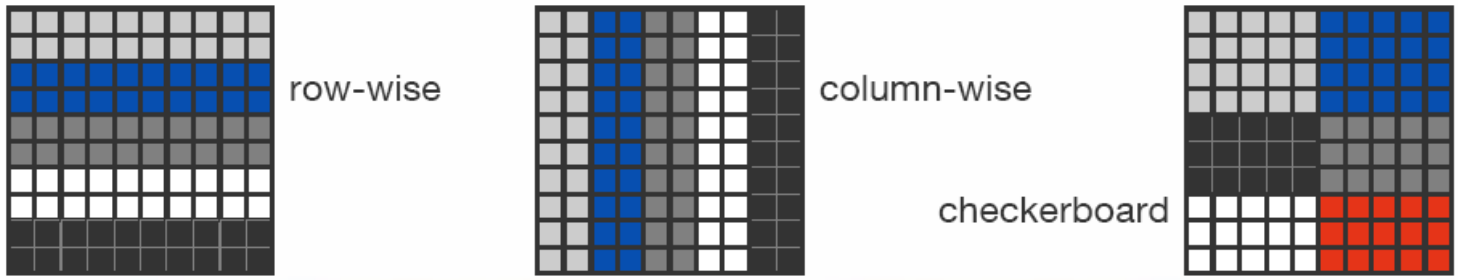
- standard sequential algorithm for $A \in K^{N \times N}$ and $b, c \in K^N$

```
for i ← 1 to N do
        c[i] ← 0;
        for j ← 1 to N do
                c[i] ← c[i] + A[i][j]*b[j];
        od
od
```

- for full matrix A this algorithm has a complexity of $O(N^2)$

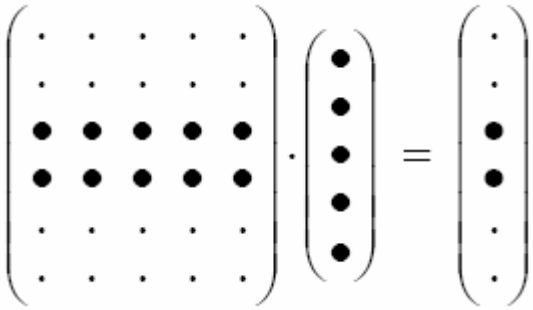
1.1 Matrix-Vector Multiplication

- in parallel, there are three main options to distribute data among P procs
  - row-wise block-striped decomposition: each process is responsible for a contiguous part of about N/P rows of A
  - column-wise block-striped decomposition: each process is responsible for a contiguous part of about N/P columns of A
  - checkerboard block decomposition: each process is responsible for a contiguous block of matrix elements
- vector b may be either replicated or block-decomposed itself



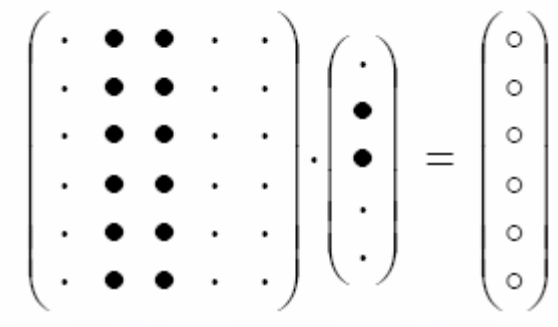
1.1 Matrix-Vector Multiplication

- row-wise block-striped decomposition
  - probably the most straightforward approach
    - each process gets some rows of A and entire vector b
    - each process computes some components of vector c
    - build and replicate entire vector c (gather-to-all, e. g.)
  - complexity of O(N2/P) multiplications / additions for P processes



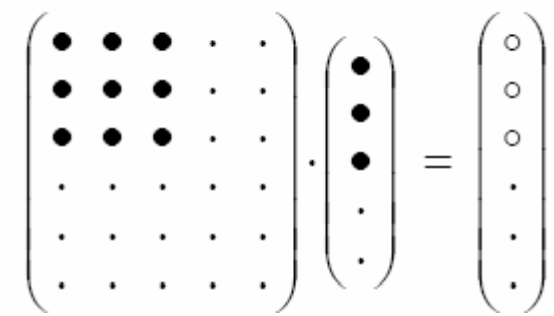
1.1 Matrix-Vector Multiplication

- column-wise block-striped decomposition
  - less straightforward approach
    - each process gets some columns of A and respective elements of vector b
    - each process computes partial results of vector c
    - build and replicate entire vector c (all-reduce or maybe a reduce-scatter if processes do not need entire vector c)
  - complexity is comparable to row-wise approach



1.1 Matrix-Vector Multiplication

- checkerboard block decomposition
  - each process gets some block of elements of A and respective elements of vector b
  - each process computes some partial results of vector c
  - build and replicate entire vector c (all-reduce, but “unused” elements of vector c have to be initialised with zero)
  - complexity of the same order as before; it can be shown that checkerboard approach has slightly better scalability properties (increasing P does not require to increase N, too)



1.2 Matrix-Matrix Multiplication

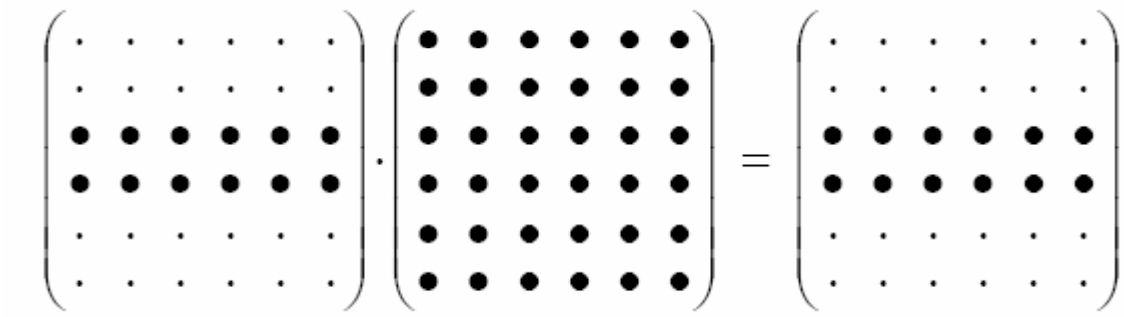
- appearances
  - computational chemistry (computing changes of state, e. g.)
  - signal processing (DFT, e. g.)
- standard sequential algorithm for A, B, C $\in K^{N\times N}$

```
for i ← 1 to N do
        for j ← 1 to N do
              C[i][j] ← 0
              for k ← 1 to N do
                    C[i][j] ← C[i][j] + A[i][k]*B[k][j];
              od
        od
od
```

- for full matrices A and B this algorithm has a complexity of $O(N^3)$

1.2 Matrix-Matrix Multiplication

- naïve parallelisation
  - each process gets some rows of A and entire matrix B
  - each process computes some rows of C



- problem: once N reaches a certain size, matrix B won't fit completely into cache and / or memory performance will dramatically decrease
- remedy: subdivision of matrix B instead of whole matrix B

1.2 Matrix-Matrix Multiplication

- recursive algorithm
  - algorithm follows the divide-and-conquer principle
  - subdivide both matrices A and B into four smaller submatrices

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \qquad B = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

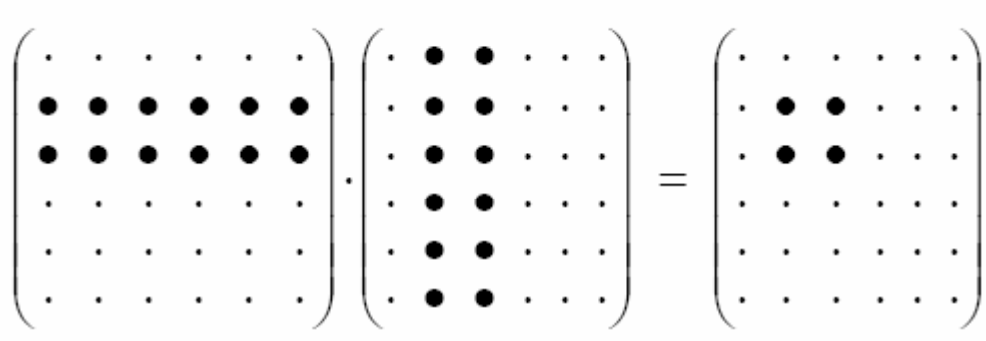
  - hence, the matrix multiplication can be computed as follows

$$C = \begin{pmatrix} A_{00} \cdot B_{00} + A_{01} \cdot B_{10} & A_{00} \cdot B_{01} + A_{01} \cdot B_{11} \\ A_{10} \cdot B_{00} + A_{11} \cdot B_{10} & A_{10} \cdot B_{01} + A_{11} \cdot B_{11} \end{pmatrix}$$

  - if blocks are still too large for the cache, repeat this step (i. e. recursively subdivide) until it fits
  - furthermore, this method has significant potential for parallelisation (especially for MemMS)

1.2 Matrix-Matrix Multiplication

- CANNON's algorithm
  - each process gets some rows of A and some columns of B
  - each process computes some components of matrix C
  - different possibilities for assembling the result
    - gather all data, build and (maybe) replicate matrix C
    - "pump" data onward to next process (-> systolic array)
  - complexity of $O(N^3/P)$ multiplications / additions for P processes



2. The Poisson equation with Dirichlet boundary conditions

- occurrences: a fitted membrane, the stationary heat equation, ...
- an elliptic partial differential equation (PDE) with Dirichlet boundary conditions on a given domain Ω
- The Poisson equation $\Delta u = f$ on the unit square $\Omega =]0, 1[^2$ with u given on Ω's boundary

$$\Delta u(x,y) = \frac{\delta^2 u(x,y)}{\delta^2 x} + \frac{\delta^2 u(x,y)}{\delta^2 y} = f(x,y) \qquad (x,y) \in \Omega$$

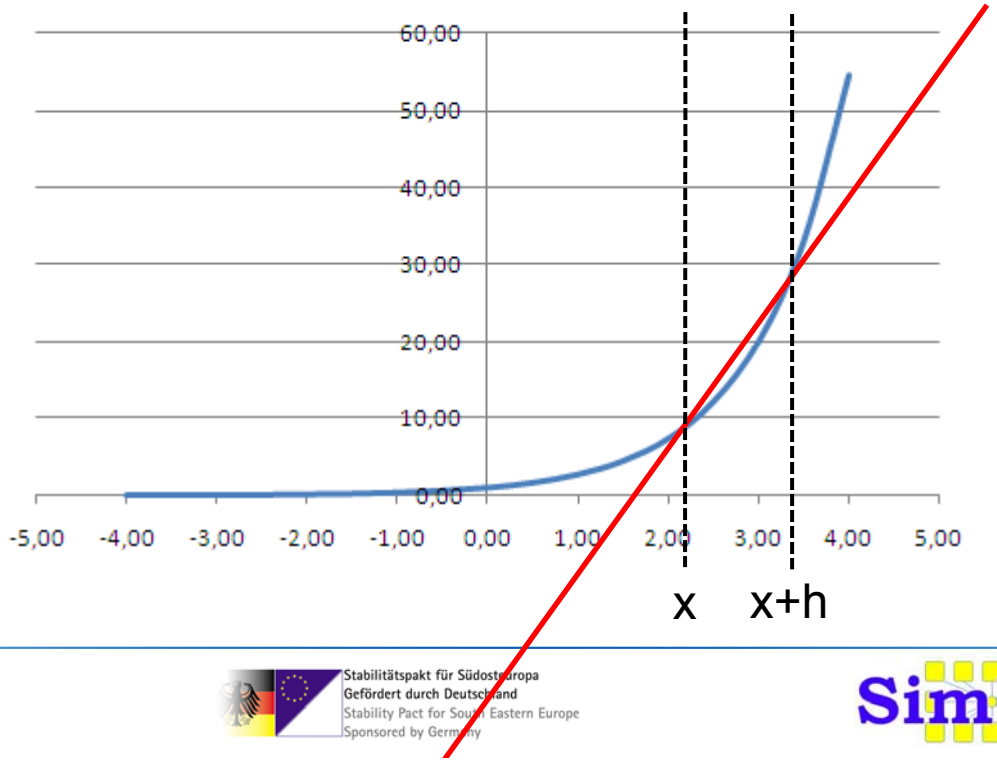
$$u(x,y) = g(x,y) \qquad (x,y) \in \partial\Omega$$

- find the function $u(x, y)$ (or an approximation to it)

2.1. Discretization using finite differences

- discretisation of the PDE to retrieve its solution
- again a simple example: the finite difference discretisation for mesh width h

$$\frac{f(x+h)-f(x)}{h} \xrightarrow{h\to 0} f'(x) \qquad \frac{f(x)-f(x-h)}{h} \xrightarrow{h\to 0} f'(x)$$



2.2. Discretization of the Poisson equation

- In our simple example the finite difference discretisation for mesh width h:

$$\frac{\delta^2 u(x,y)}{\delta^2 x} \approx \frac{u(x-h,y)-2u(x,y)+u(x+h,y)}{h^2}$$

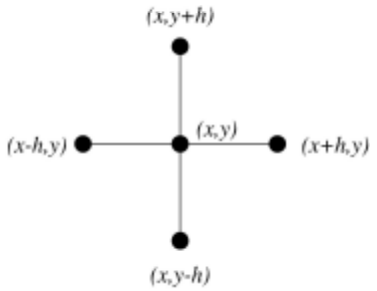
$$\frac{\delta^2 u(x,y)}{\delta^2 y} \approx \frac{u(x,y-h)-2u(x,y)+u(x,y+h)}{h^2}$$

- introduction of an equidistant grid of $(N+1)^2$ grid points

$$u_{i,j} \approx u(ih, jh) \qquad i = 0,..,N \quad j = 0,..,N \qquad N = \frac{1}{h}$$

- resulting discrete equation in the interior: **five-point difference star**

$$u_{i,j-1} + u_{i-1,j} - 4u_{i,j} + u_{i+1,j} + u_{i,j+1} = h^2 f_{i,j} \qquad 0 < i, j < N$$

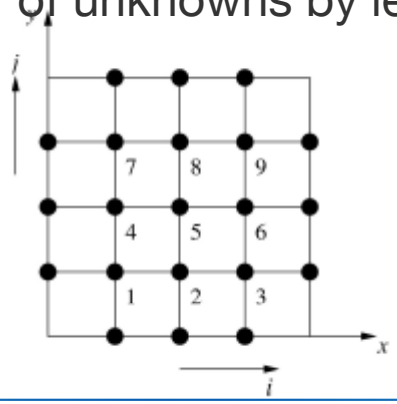


- resulting equation on the boundary

$$u_{i,j} = g(ih, jh) \qquad i = 0 \vee i = N \vee j = 0 \vee j = N$$

2.3. Resulting system of linear equations

- for each inner point one linear equation in the unknowns $u_{i,j}$
- equations in points next to the boundary $i \in \{1, N-1\}$, $j \in \{1, N-1\}$ access the boundary values
  - shift these to the right-hand side of the equation
  - hence, all unknowns are located to the left of the '=' sign, all known quantities to its right
- assemble the overall vector of unknowns by lexicographic row-wise ordering



2.4. Resulting matrix structure

- this results in a system $Ax = b$ of $(N-1)^2$ linear equations in $(N-1)^2$ unknowns

$$A = \begin{pmatrix} T & I & & & \\ I & T & I & & \\ & I & \ddots & \ddots & \\ & & \ddots & \ddots & I \\ & & & I & T \end{pmatrix}$$

- matrix A is block-tridiagonal with identity or tridiagonal blocks I or T, resp.

$$T = \begin{pmatrix} -4 & 1 & & & \\ 1 & -4 & 1 & & \\ & 1 & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & -4 \end{pmatrix} \in \mathbf{R}^{N-1,N-1}$$

2.5. Direct solving large sparse systems of linear equations

- the standard textbook method is Gaussian elimination

- this is a so-called direct solver which provides the exact solution of the system (apart from round-off errors)

- drawbacks of Gaussian elimination:
  - for M unknowns $O(M^3)$ arithmetic operations are needed(not acceptable for really large M as they are standard in modern simulation problems)
  - no exploitation of the sparsity of the matrix by the algorithm:
    existing zeroes are “destroyed” (turned into non-zeroes),
    =>more computational work and more storage requirements

2.6. Iterative solving large sparse systems of linear equations

- use iterative methods instead
  - Approaching exact solution and approximate it, but typically don't reach it
  - costs of O(M) operations for one step of iteration
  - typically much less than $O(M^2)$ steps needed (the gain)
  - ideal case(multigrid or multilevel methods): only O(1) steps needed
  - basic (and not that sophisticated) methods (number of steps still depending on M):
    - relaxation methods: Jacobi, Gauß-Seidel, SOR (Successive Over-Relaxation)
    - minimization methods: steepest descent, conjugate gradients

3.1. The Jacobi iteration

- decompose $A = L_A + D_A + U_A$ in its diagonal part D_A, its upper triangular part U_A and its lower triangular part L_A

- starting point: $b = Ax = D_A x + (L_A + U_A)x$

- writing $b = D_A x^{it+1} + (L_A + U_A)x^{it}$ with x^{it} denoting the approximation to *x* after *it* steps of the iteration leads to the following iterative scheme:

$$x^{it+1} := -D_A{}^{-1}(L_A + U_A)x^{it} + D_A{}^{-1}b = x^{it} + D_A{}^{-1}r^{it}$$

residual is defined as $r^{it} = b - Ax^{it}$

- a more explicit algorithmic form:

```
for it=0,1,2,...:
    for k=1,...,M:
```

$$x_k{}^{it+1} = \frac{1}{a_{k,k}}\left(b_k - \sum_{j \neq k} a_{k,j} x_j^{it}\right)$$

3.2. The Jacobi iteration for the Poisson equation

- for our special *A* resulting from the finite difference discretization of the Poisson equation follows (pay attention to the indices!):

```
for it=0,1,2,...:
    for j=1,...,N-1:
        for i=1,...,N-1:
```

$$u_{i,j}^{it+1} = \frac{1}{4}\left(u_{i,j-1}^{it} + u_{i-1,j}^{it} + u_{i,j+1}^{it} + u_{i+1,j}^{it} - h^2 f_{i,j}\right)$$

- remember: boundary values are fixed

4.1. The Gauß-Seidel iteration

- take the same decomposition $A = L_A + D_A + U_A$

- new starting point: $b = Ax = (D_A + L_A)x + U_A x$ Writing $b = (D_A + L_A)x^{it+1} + U_A x^{it}$ leads to the following iterative scheme:

$$x^{it+1} := -(D_A + L_A)^{-1} U_A \; x^{it} + (D_A + L_A)^{-1} b = x^{it} + (D_A + L_A)^{-1} r^{it}$$

- in a more explicit algorithmic form:

```
for it=0,1,2,...:
    for k=1,...,M:
```

$$x_k^{it+1} = \frac{1}{a_{k,k}} \left(b_k - \sum_{j=1}^{k-1} a_{k,j} x_j^{it+1} - \sum_{j=k+1}^{M} a_{k,j} x_j^{it} \right)$$

4.2. The Gauß-Seidel iteration for the Poisson equation

- for our special *A* resulting from the finite difference discretization of the Poisson equation follows (pay attention to the indices!):

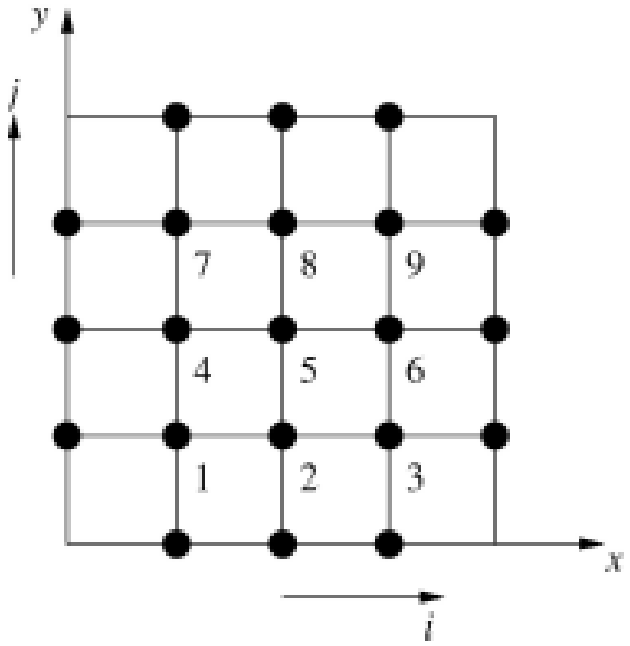
```
for it=0,1,2,...:
    for j=1,...,N-1:
        for i=1,...,N-1:
```

$$u_{i,j}^{it+1} = \frac{1}{4}\left(u_{i,j-1}^{it+1} + u_{i-1,j}^{it+1} + u_{i,j+1}^{it} + u_{i+1,j}^{it} - h^2 f_{i,j}\right)$$

- remember: again boundary values are fixed

- no general superiority of Gauß-Seidel to Jacobi; in our case discussed here, however, Gauß-Seidel is twice as fast as Jacobi

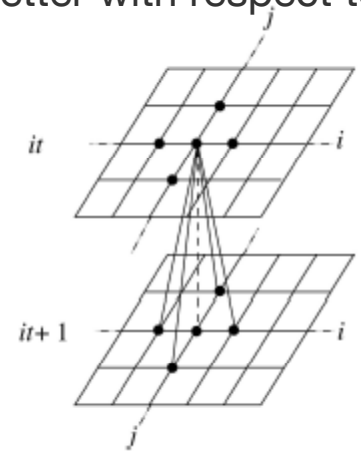
5. Parallelizing Jacobi

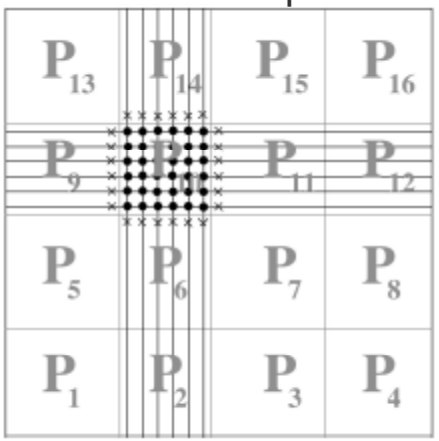


5.1. Parallelizing Jacobi

- neither Jacobi nor Gauß-Seidel are used today any more – they are too slow; nevertheless, the algorithmic aspects are still of interest

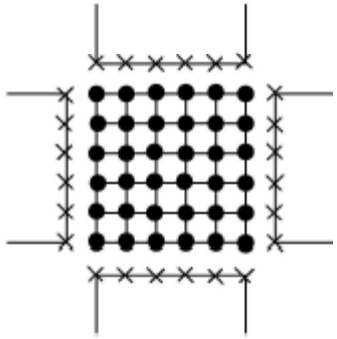
- a parallel Jacobi algorithm is quite straightforward:
  - in the current iteration step, only values from the previous step are used
  - hence, all updates of one iteration step can be made in parallel (if that many processors are available)
  - more realistic scenario: subdivide the domain into strips or squares for example (what is better with respect to a good communication-computation ratio?)





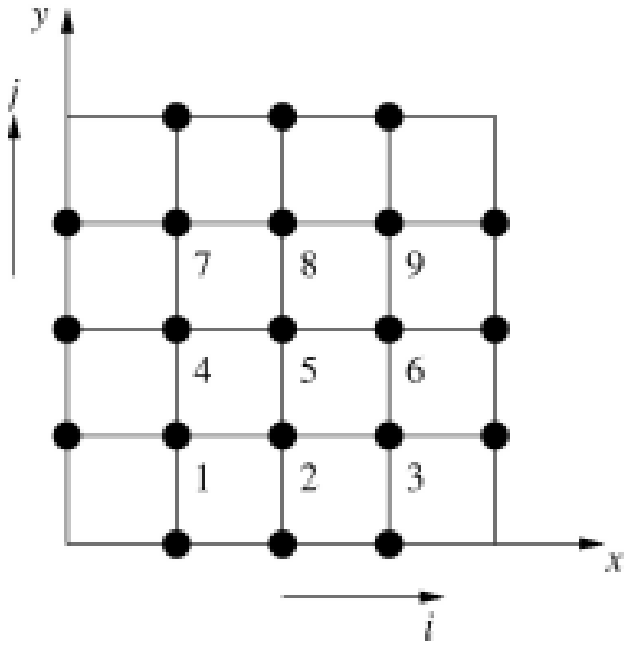
5.2. Parallelizing Jacobi (cont´d)

- each processor needs for its calculations:
  - if adjacent to the boundary: a subset of the boundary values
  - one row or one column of values from the processors dealing with the neighboring subdomains
  - some hint when to stop



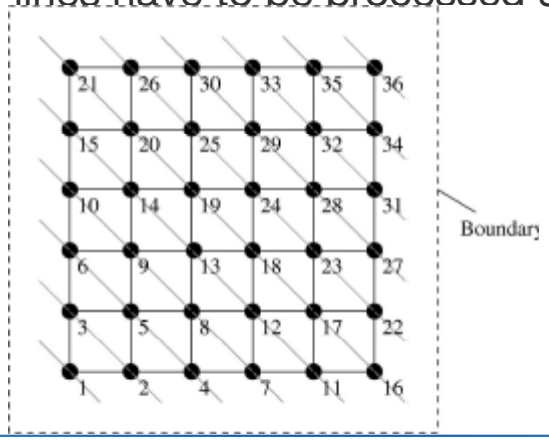
- the above considerations lead to the following algorithm each processor has to execute:
  1. update all local approximate values $u_{i,j}^{it}$ to $u_{i,j}^{it+1}$
  2. send all updates in points next to interior boundaries to the respective processors
  3. receive all necessary updates from the “neighboring” processors
  4. compute the local residual values and provide them via a reduce operation
  5. receive the overall residual as the reduce operation’s result and go back to 1. if this value is larger than some given threshold

6. Parallelizing Gauß-Seidel



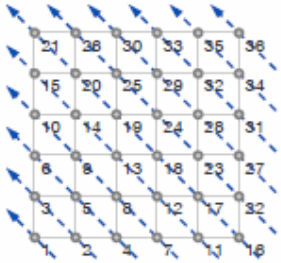
6.1. Parallelizing Gauß-Seidel by wavefront ordering

- at first glance, there seems to be an enforced sequential order, since the updated values are immediately used where available
- remedy: change the order of visiting and updating the grid points
- first possibility: wavefront ordering
  - diagonal order of updating
  - all values along a diagonal line can be updated in parallel
  - the single diagonal lines have to be processed sequentially, however



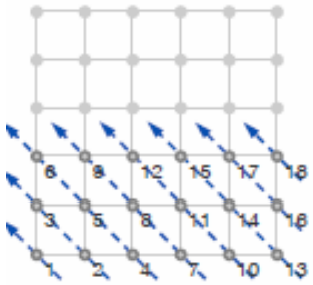
6.2. Parallelizing Gauß-Seidel by wavefront ordering (cont´d)

- problem: suppose we have $P = N - 1$ processors; then there are P^2 overall updates that can be organized in $2P - 1$ sequential steps (diagonals), which restricts the speed-up to roughly $P/2$



P_1:	1	2	4	7	11	16	22	27	31	34	36
P_2:		3	5	8	12	17	23	28	32	35	
P_3:			6	9	13	18	24	29	33		
P_4:				10	14	19	25	30			
P_5:					15	20	26				
P_6:						21					

- better: use $P = (N - 1)/k$ processors only; then we get k sequential strips of kP^2 updates and $kP + P - 1$ sequential internal steps; now, the speed-up is given by $k \cdot kP^2/(k(kP + P - 1))$, which is roughly $kP/(k + 1)$

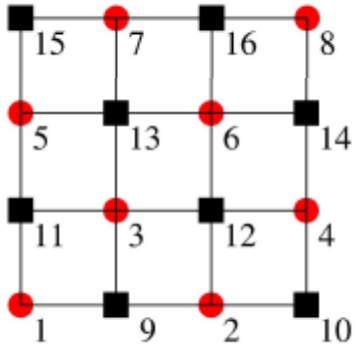


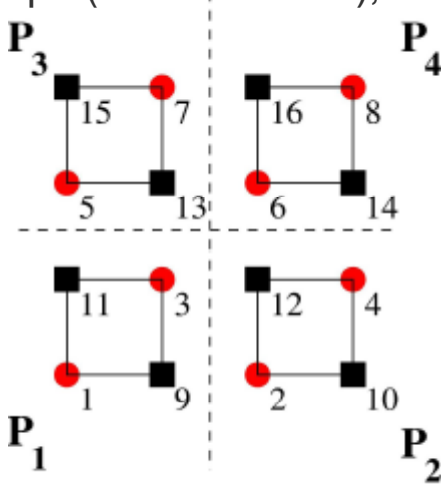
P_1:	1	2	4	7	10	13	16	18
P_2:		3	5	8	11	14	17	
P_3:			6	9	12	15		

here, K = 2 → speed-up S(p) = 2P/3

6.3. Parallelizing Gauß-Seidel by checkerboard ordering

- second possibility: red-black or checkerboard ordering
  - give the grid points a checkerboard coloring of red and black
  - order of visiting and updating: first lexicographically the red ones, then lexicographically the black ones
  - no dependences within the red set nor within the black set
  - subdivide the grid such that each processor has some red and some black points (roughly the same number)
  - the result: two necessarily sequential steps (red and black),but perfect parallelism within each of them





Thank you for your attention!