High Performance Computing – Programming Paradigms and Scalability

PD Dr. rer. nat. habil. Ralf-Peter Mundani
Computation in Engineering / BGU
Scientific Computing in Computer Science / INF

Summer Term 2018

Part 6: Examples of Parallel Algorithms

Everything that can be invented

has been invented.

—Charles H. Duell

commissioner U.S. Office of Patents, 1899

- overview
  - parallel matrix operations
  - iterative solvers: parallel JACOBI and GAUSS-SEIDEL
  - parallel sorting

Parallel Matrix Operations

- reminder: matrix
  - underlying basis for many scientific problems is a matrix
  - stored as 2-dimensional array of numbers
    - row-wise in memory (typical case)
    - column-wise in memory
  - typical matrix operations (K: integer, float, or double)

    1) $\mathbf{A} + \mathbf{B} = \mathbf{C}$ with $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C} \in K^{n \times m}$

    2) $\mathbf{A} \cdot \mathbf{b} = \mathbf{c}$ with $\mathbf{A} \in K^{n \times m}$, $\mathbf{b} \in K^{m}$, and $\mathbf{c} \in K^{n}$

    3) $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$ with $\mathbf{A} \in K^{n \times l}$, $\mathbf{B} \in K^{l \times m}$, and $\mathbf{C} \in K^{n \times m}$

  - 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...?

Parallel Matrix Operations

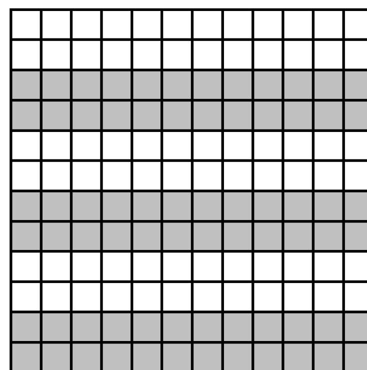
- matrix-vector multiplication
  - appearances
    - systems of linear equations $\mathbf{A}\cdot\mathbf{x} = \mathbf{b}$
    - iterative solvers (e.g. conjugate gradient)
    - neural networks (determination of output values, training, ...)
  - standard sequential algorithm for $\mathbf{A} \in K^{n\times n}$ and $\mathbf{b}, \mathbf{c} \in K^{n}$

```
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
```

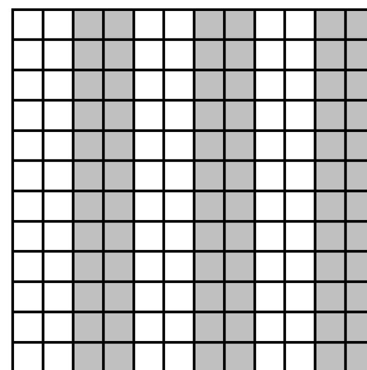
  - for full matrix **A** this algorithm has a complexity of $\mathrm{O}(n^2)$

Parallel Matrix Operations

- matrix-vector multiplication (cont'd)
  - 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

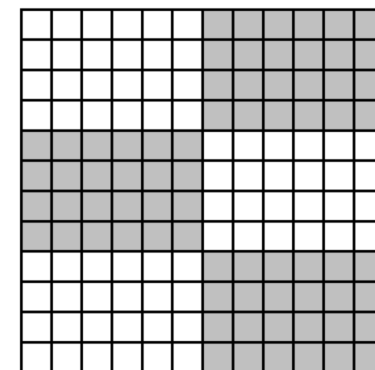


row-wise



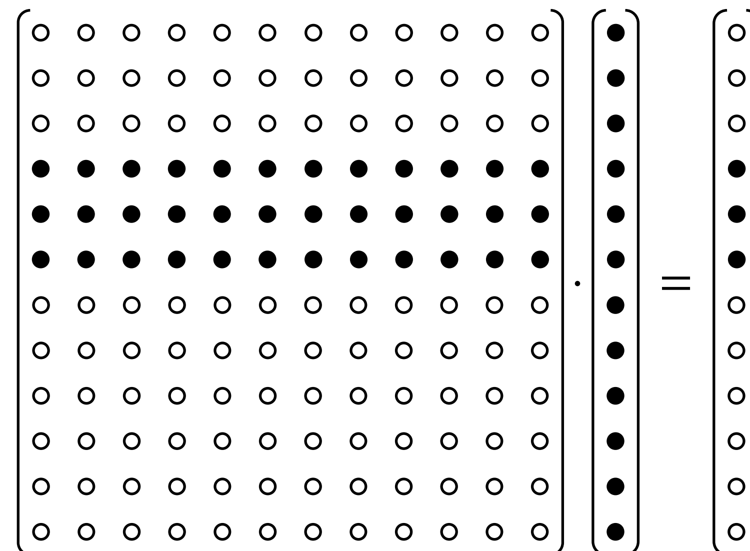
column-wise

checkerboard



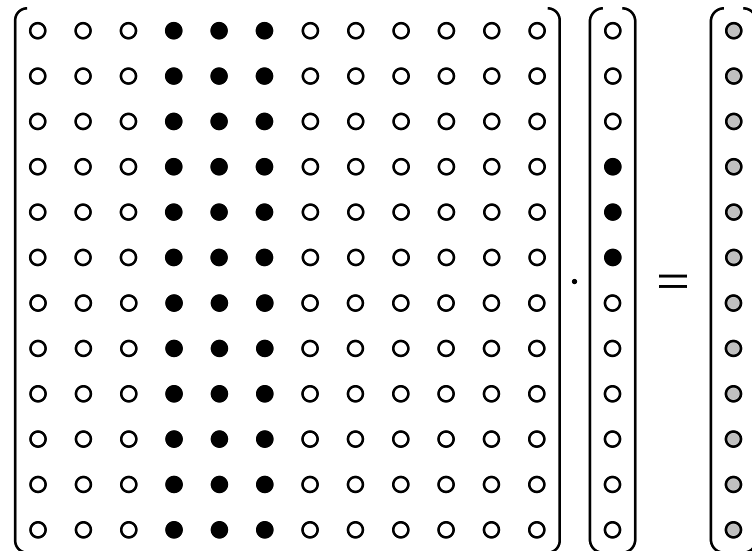
Parallel Matrix Operations

- matrix-vector multiplication (cont’d)
  - **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**
    - complexity of $O(n^2/p)$ multiplications / additions for each processes



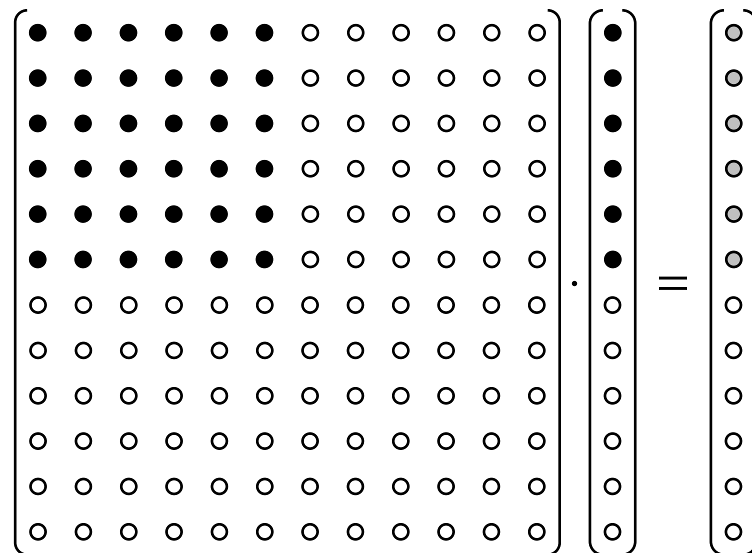
Parallel Matrix Operations

- matrix-vector multiplication (cont'd)
  - **column-wise block-striped decomposition**
    - less straightforward approach
    - each process gets some columns of **A** and respective elements of **b**
    - each process computes partial results of vector **c**
    - complexity is comparable to row-wise approach



Parallel Matrix Operations

- matrix-vector multiplication (cont'd)
  - **checkerboard block decomposition**
    - each process gets some block of **A** and respective elements of vector **b**
    - each process computes some partial results of vector **c**
    - complexity of same order as before; nevertheless, checkerboard approach has slightly better scalability properties



Parallel Matrix Operations

- matrix multiplication
  - appearances
    - computational chemistry (e.g. computing changes of state)
    - signal processing (e.g. DFT)
  - 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 $O(n^3)$

Parallel Matrix Operations

- matrix multiplication (cont'd)
  - naïve parallelisation
    - each process gets some rows of **A** and entire matrix **B**
    - each process computes some rows of **C**

$$\begin{pmatrix} \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \end{pmatrix} \cdot \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{pmatrix} = \begin{pmatrix} \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \end{pmatrix}$$

  - 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**

Parallel Matrix Operations

- matrix multiplication (cont'd)
  - recursive approach
    - algorithm follows the divide-and-conquer principle
    - subdivide both matrices **A** and **B** into four smaller submatrices

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

    - hence, the matrix multiplication can be computed as follows

$$\mathbf{C} = \begin{pmatrix} \mathbf{A}_{00}\cdot\mathbf{B}_{00} + \mathbf{A}_{01}\cdot\mathbf{B}_{10} & \mathbf{A}_{00}\cdot\mathbf{B}_{01} + \mathbf{A}_{01}\cdot\mathbf{B}_{11} \\ \mathbf{A}_{10}\cdot\mathbf{B}_{00} + \mathbf{A}_{11}\cdot\mathbf{B}_{10} & \mathbf{A}_{10}\cdot\mathbf{B}_{01} + \mathbf{A}_{11}\cdot\mathbf{B}_{11} \end{pmatrix}$$

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

Parallel Matrix Operations

- matrix multiplication (cont'd)
  - CANNON's algorithm
    - each process gets some rows of **A** and some columns of **B**
    - each process computes some components of matrix **C**

$$\begin{pmatrix} \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \end{pmatrix} \cdot \begin{pmatrix} \bullet & \bullet & \bullet & \circ & \circ & \circ & \circ & \circ & \circ \\ \bullet & \bullet & \bullet & \circ & \circ & \circ & \circ & \circ & \circ \\ \bullet & \bullet & \bullet & \circ & \circ & \circ & \circ & \circ & \circ \\ \bullet & \bullet & \bullet & \circ & \circ & \circ & \circ & \circ & \circ \\ \bullet & \bullet & \bullet & \circ & \circ & \circ & \circ & \circ & \circ \\ \bullet & \bullet & \bullet & \circ & \circ & \circ & \circ & \circ & \circ \\ \bullet & \bullet & \bullet & \circ & \circ & \circ & \circ & \circ & \circ \\ \bullet & \bullet & \bullet & \circ & \circ & \circ & \circ & \circ & \circ \\ \bullet & \bullet & \bullet & \circ & \circ & \circ & \circ & \circ & \circ \end{pmatrix} = \begin{pmatrix} \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\ \bullet & \bullet & \bullet & \circ & \circ & \circ & \circ & \circ & \circ \\ \bullet & \bullet & \bullet & \circ & \circ & \circ & \circ & \circ & \circ \\ \bullet & \bullet & \bullet & \circ & \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \end{pmatrix}$$

    - complexity of $\mathrm{O}(n^3/p)$ multiplications / additions for each processes
    - but additional overhead for global assembly of matrix **C** necessary

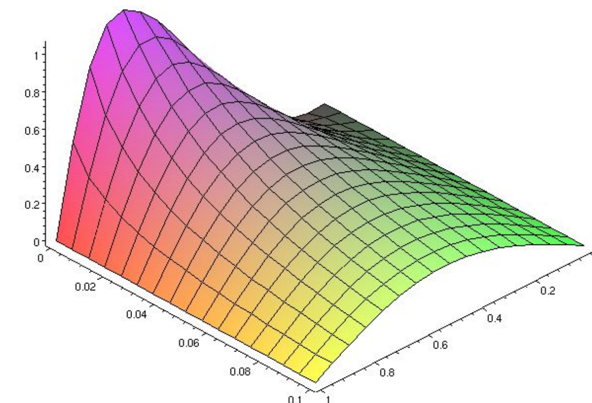
- overview
    - parallel matrix operations ✓
    - iterative solvers: parallel JACOBI and GAUSS-SEIDEL
    - parallel sorting

Iterative Solvers: Parallel JACOBI and GAUSS-SEIDEL

- scenario
  - solve an elliptic partial differential equation (PDE) with DIRICHLET boundary conditions on a given domain Ω
  - simple example: POISSON equation $-\Delta \mathbf{u} = \mathbf{f}$

$$-\Delta u(x,y) = -\frac{\partial^2 u(x,y)}{\partial x^2} - \frac{\partial^2 u(x,y)}{\partial y^2} = f(x,y)$$

    for $(x,y) \in \Omega =]0,1[\times]0,1[$ with **u** given on the boundary of Ω
  - task: $u(x,y)$ or an approximation to it has to be found
  - occurrences of such PDEs
    - fixed membrane
    - stationary heat equation (picture)
    - electrostatic fields
    - ...



Iterative Solvers: Parallel JACOBI and GAUSS-SEIDEL

- discretisation
  - for solving our sample PDE, a discretisation becomes necessary
  - typical discretisation techniques
    - finite differences
    - finite volumes
    - finite elements

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

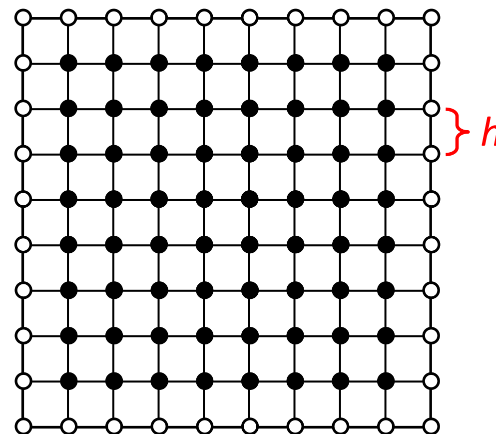
  - finite difference discretisation (using one forward and one backward difference) for the second derivatives in $-\Delta \mathbf{u} = \mathbf{f}$ leads to

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

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

Iterative Solvers: Parallel JACOBI and GAUSS-SEIDEL

- discretisation (cont'd)
  - for the computational solution of our problem a 2D grid is necessary
    - equidistant grid with $(n+1)\times(n+1)$ grid points, hence $h = 1/n$
    - instead of (x,y) we can write $(i{\cdot}h, j{\cdot}h)$ with $i, j = 0, \ldots, n$ or just (i, j)
    - further simplification: $u_{i,j} \approx u(i{\cdot}h, j{\cdot}h)$ with $i, j = 0, \ldots, n$



● inner point

○ boundary point

  - hence, our model problem simplifies to

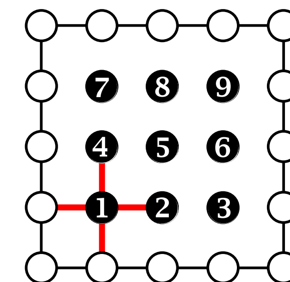
$$-u_{i-1,j} - u_{i,j-1} + 4u_{ij} - u_{i+1,j} - u_{i,j+1} = h^2 \cdot f_{ij} \qquad (1)$$

Iterative Solvers: Parallel JACOBI and GAUSS-SEIDEL

- system of linear equations
  - resulting equation on the boundary

$$u_{i,j} = g_{i,j} \qquad i, j = 0 \;\text{ or }\; i, j = n$$

  - for each inner point there is one linear equation according to (1)
  - equations in points next to the boundary (i.e. $i, j = 1$ or $i, j = n–1$) access boundary values
    - these are shifted to the right-hand side of the equation
    - hence, all unknowns are located left, all known quantities right
  - assembly of the overall vector of unknowns by lexicographic row-wise ordering results to system of linear equations $\mathbf{Au} = \mathbf{f}$
    - with $(n–1)^2$ equations in $(n–1)^2$ unknowns
    - **A** has block-tridiagonal structure and is sparse



Iterative Solvers: Parallel JACOBI and GAUSS-SEIDEL

- system of linear equations (cont'd)
  - resulting system of linear equations $\mathbf{Au} = \mathbf{f}$

$$
\underbrace{\begin{pmatrix}
4 & -1 & & & -1 & & \\
-1 & 4 & -1 & & & \ddots & \\
 & \ddots & \ddots & \ddots & & & -1 \\
 & & \ddots & \ddots & \ddots & & \\
-1 & & & \ddots & \ddots & \ddots & \\
 & \ddots & & & -1 & 4 & -1 \\
 & & -1 & & & -1 & 4
\end{pmatrix}}_{=:\,\mathbf{A}}
\cdot
\underbrace{\begin{pmatrix}
u_{1,1} \\ \vdots \\ u_{n-1,1} \\ u_{1,2} \\ \vdots \\ \vdots \\ \vdots \\ u_{n-1,n-1}
\end{pmatrix}}_{=:\,\mathbf{u}}
=
\underbrace{\begin{pmatrix}
f_{1,1} \\ \vdots \\ f_{n-1,1} \\ f_{1,2} \\ \vdots \\ \vdots \\ \vdots \\ f_{n-1,n-1}
\end{pmatrix}}_{=:\,\mathbf{f}}
\left.\vphantom{\begin{pmatrix} 1\\1\\1\\1\\1\\1\\1\\1 \end{pmatrix}}\right\} (n-1)^2
$$

Iterative Solvers: Parallel JACOBI and GAUSS-SEIDEL

- system of linear equations (cont'd)
  - schoolbook method for solving SLEs: GAUSSian elimination
    - direct solver that provides exact solution
    - has a complexity of $O(n^3)$ for n unknowns (!)
    - does not exploit scarcity of matrix **A** that is even filled-up (i.e. existing zeros are 'destroyed') during solution
  - hence, using some iterative method instead
    - approximates the exact solution
    - advantageous only if less than $O(n^3)$ steps needed
    - (some) basic methods
      - relaxation methods: JACOBI, GAUSS-SEIDEL, SOR
      - minimisation methods: steepest descent, CG
      - multigrid or multilevel methods

Iterative Solvers: Parallel JACOBI and GAUSS-SEIDEL

- JACOBI iteration
  - decompose matrix **A** in its diagonal part $\mathbf{D}_A$, its lower triangular part $\mathbf{L}_A$, and its upper triangular part $\mathbf{U}_A$

$$\mathbf{A} = \mathbf{D}_A + \mathbf{L}_A + \mathbf{U}_A \tag{2}$$

  - substituting (2) into $\mathbf{Ax} = \mathbf{b}$ leads to

$$(\mathbf{D}_A + \mathbf{L}_A + \mathbf{U}_A)\mathbf{x} = \mathbf{b},$$

    and finally to

$$\mathbf{D}_A\mathbf{x} = \mathbf{b} - (\mathbf{L}_A + \mathbf{U}_A)\mathbf{x}$$

  - denoting all **x** on the left-hand side as iteration step k+1 and all **x** on the right-hand side as iteration step k, we get our final iteration scheme as

$$\mathbf{x}^{(k+1)} = \mathbf{D}_A^{-1}(\mathbf{b} - (\mathbf{L}_A + \mathbf{U}_A)\mathbf{x}^{(k)}) \tag{3}$$

Iterative Solvers: Parallel JACOBI and GAUSS-SEIDEL

- JACOBI iteration (cont'd)
  - in component-based form (3) can be written as

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)})$$

or in case of our model problem

$$u_{ij}^{(k+1)} = \frac{1}{4}(h^2 \cdot f_{ij} + u_{i-1,j}^{(k)} + u_{i,j-1}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j+1}^{(k)})$$

which could have also been derived directly from (1)

  - this method is know as total-step method and converges to $\mathbf{A}^{-1}\mathbf{b}$ for arbitrary start vectors $\mathbf{u}^{(0)} \in \mathbb{C}^n$ in case of diagonally dominant matrices

Iterative Solvers: Parallel JACOBI and GAUSS-SEIDEL

- GAUSS-SEIDEL iteration
  - decompose matrix $\mathbf{A}$ in its diagonal part $\mathbf{D}_A$, its lower triangular part $\mathbf{L}_A$, and its upper triangular part $\mathbf{U}_A$

$$\mathbf{A} = \mathbf{D}_A + \mathbf{L}_A + \mathbf{U}_A \qquad (4)$$

  - substituting (4) into $\mathbf{Ax} = \mathbf{b}$ leads to

$$(\mathbf{D}_A + \mathbf{L}_A + \mathbf{U}_A)\mathbf{x} = \mathbf{b},$$

    and finally to

$$(\mathbf{D}_A + \mathbf{L}_A)\mathbf{x} = \mathbf{b} - \mathbf{U}_A\mathbf{x}$$

  - denoting all $\mathbf{x}$ on the left-hand side as iteration step k+1 and all $\mathbf{x}$ on the right-hand side as iteration step k, we get our final iteration scheme as

$$\mathbf{x}^{(k+1)} = (\mathbf{D}_A + \mathbf{L}_A)^{-1}(\mathbf{b} - \mathbf{U}_A\mathbf{x}^{(k)}) \qquad (5)$$

Iterative Solvers: Parallel JACOBI and GAUSS-SEIDEL

- GAUSS-SEIDEL iteration (cont'd)
  - in component-based form (5) can be written as

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)})$$

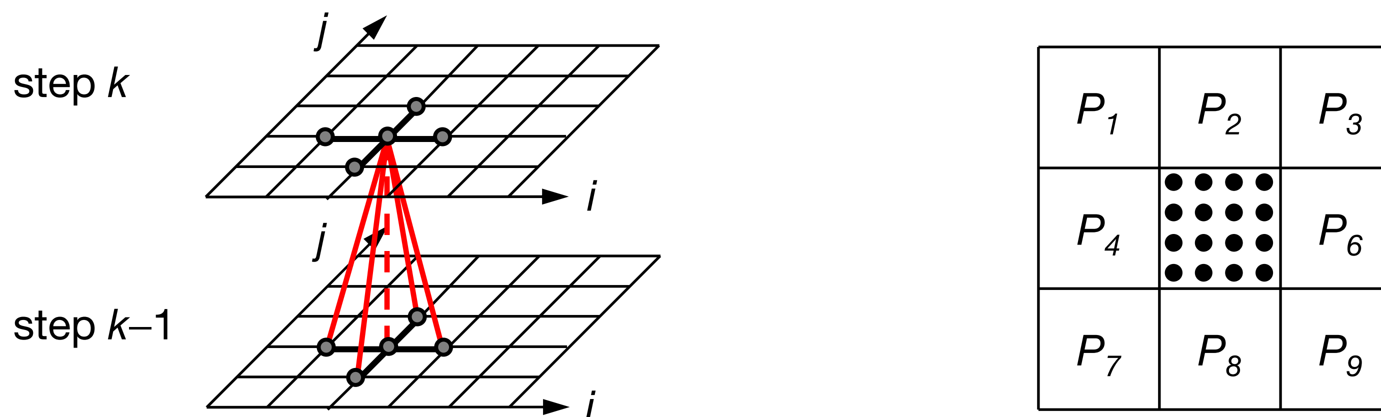
or in case of our model problem

$$u_{ij}^{(k+1)} = \frac{1}{4}(h^2 \cdot f_{ij} + u_{i-1,j}^{(k+1)} + u_{i,j-1}^{(k+1)} + u_{i+1,j}^{(k)} + u_{i,j+1}^{(k)})$$

  - this method is know as single-step method (due to usage of old and new values of **u** ➔ faster convergence to be expected) and converges to $\mathbf{A}^{-1}\mathbf{b}$ for arbitrary start vectors $\mathbf{u}^{(0)} \in \mathbb{C}^n$ in case of diagonally dominant matrices

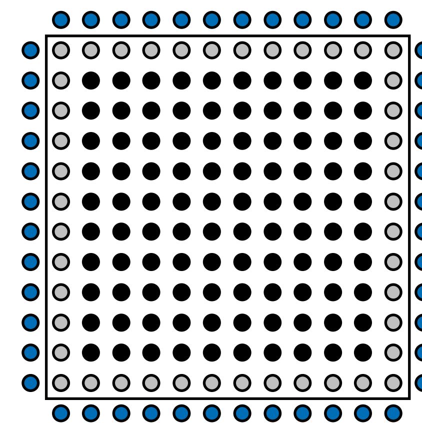
Iterative Solvers: Parallel JACOBI and GAUSS-SEIDEL

- parallelisation of JACOBI iteration
  - neither JACOBI nor GAUSS-SEIDEL are used today very frequently for solving large SLEs (they are too slow)
  - nevertheless, the algorithmic aspects are still of interest
  - parallel JACOBI is quite straightforward
    - in iteration step k only values from step k–1 are used
    - hence, all updates of one step can be made in parallel
    - but for parallelisation further domain decomposition necessary (with special processing of local boundary elements)



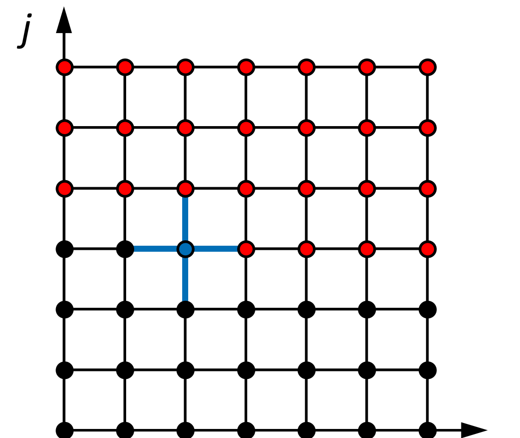
Iterative Solvers: Parallel JACOBI and GAUSS-SEIDEL

- parallelisation of JACOBI iteration (cont'd)
  - for its computations, each processor *P* needs
    - a subset of local values
    - one row/column of values ('*halo*') from its neighbouring processes
  - hence, each processor has to execute the following algorithm
    1) update all local (i.e. • and ∘) approximate values $u_{ij}^{(k)}$ to $u_{ij}^{(k+1)}$
    2) send all updates (∘) to the respective neighbouring processes
    3) receive all necessary updates (•) from neighbouring processes
    4) compute local residual and perform an all-reduce for the global residual
    5) continue if global residual is larger than some threshold value ε



Iterative Solvers: Parallel JACOBI and GAUSS-SEIDEL

- parallelisation of GAUSS-SEIDEL iteration
  - problem: since the updated values are immediately used where available, parallelisation seems to be quite complicated

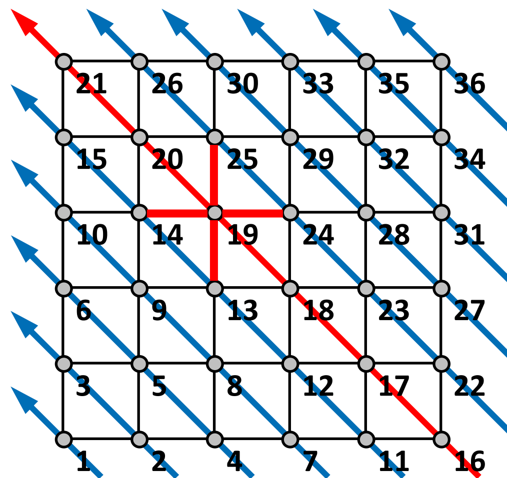


- ● updated values (step k)
- ● old values (step k–1)

  - hence, different order of visiting / updating the grid points is necessary
    - wavefront ordering
    - red-black or checkerboard ordering

Iterative Solvers: Parallel Jacobi and Gauss-Seidel

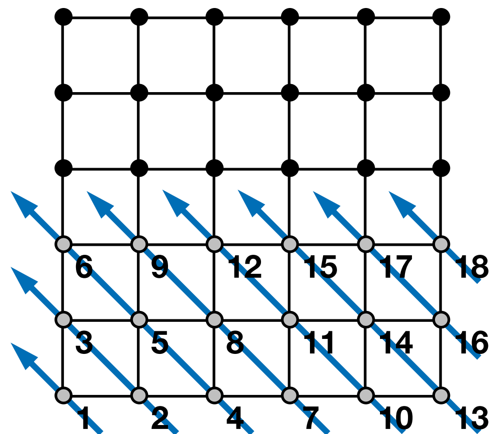
- parallelisation of Gauss-Seidel iteration (cont'd)
  - wavefront ordering
    - diagonal ordering of updates ➔ all values along a diagonal line can be updated in parallel; single diagonal lines still have to be processed sequentially
    - problem: for $p = n$ processors there are p^2 updates that need $2p–1$ sequential steps ➔ speed-up restricted to $p/2$



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					

Iterative Solvers: Parallel JACOBI and GAUSS-SEIDEL

- parallelisation of GAUSS-SEIDEL iteration (cont'd)
  - wavefront ordering
    - better
      - row-wise decomposition of matrix **A** in k blocks of n/k rows
      - for $p = n/k$ processors there are k sequential blocks of $k \cdot p^2$ updates that need $k \cdot p + p - 1$ sequential steps each
      - hence, speed-up restricted to $k \cdot p/(k+1)$

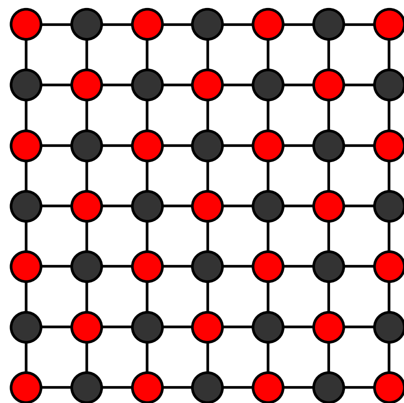


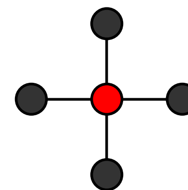
P_1	1	2	4	7	10	13	16	18
P_2		3	5	8	11	14	17	
P_3			6	9	12	15		

here, $k = 2$ ➔ speed-up $S(p) = 2p/3$

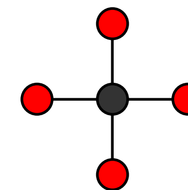
Iterative Solvers: Parallel JACOBI and GAUSS-SEIDEL

- parallelisation of GAUSS-SEIDEL iteration (cont'd)
  - red-black or checkerboard ordering
    - grid points get a checkerboard colouring of red and black
    - lexicographic order of visiting / updating the grid points
      - first the red ones, than the black ones
      - hence, no dependencies within red nor within black set
    - subdivide grid such that each processor gets some red / black points ➔ two sequential steps necessary, but perfect parallelism within both





5-point stencil
for red points



5-point stencil
for black points

- overview
  - parallel matrix operations ✓
  - iterative solvers: parallel JACOBI and GAUSS-SEIDEL ✓
  - parallel sorting

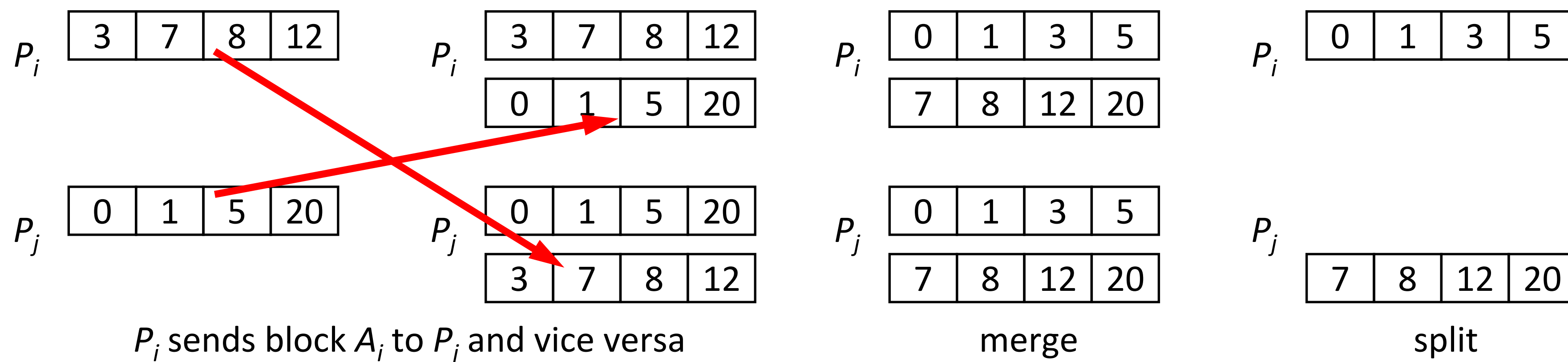
Parallel Sorting

- reminder: sorting
  - one of the most common operations performed by computers
  - let $A = \langle a_1, a_2, \ldots, a_n \rangle$ be a sequence of n elements in arbitrary order
  - sorting transforms A into a monotonically increasing or decreasing sequence $\tilde{A} = \langle \tilde{a}_1, \tilde{a}_2, \ldots, \tilde{a}_n \rangle$ such that
    - $\tilde{a}_i \leq \tilde{a}_j$ for $1 \leq i \leq j \leq n$ (increasing order)
    - $\tilde{a}_i \geq \tilde{a}_j$ for $1 \leq i \leq j \leq n$ (decreasing order)

    and $\tilde{A}$ being a permutation of A
  - in general, sorting algorithms are comparison-based, i.e. an algorithm sorts an unordered sequence of elements by repeatedly comparing / exchanging pairs of elements
  - lower bound of the sequential complexity of any comparison-based algorithm is $\mathrm{O}(n \cdot \log_2 n)$

Parallel Sorting

- basic operations
  - in sequential / parallel sorting algorithms, some basic operations are repeatedly executed
    - compare-exchange: elements a_i and a_j are compared and exchanged in case they are out of sequence
    - compare-split: already sorted blocks of elements A_i and A_j stored at different processors P_i and P_j, resp., are merged and split in the following manner



Parallel Sorting

- bubble sort
  - simple comparison-based sorting algorithm of complexity $O(n^2)$
  - standard sequential algorithm for sorting sequence A

```
for i ← n-1 to 1 by -1 do
  for j ← 0 to i-1 do
    compare-exchange (a[j], a[j+1]);
  od
od
```

  - example: iterations $i = 1, 2, 3$ for sorting $A = \langle 3, 2, 3, 8, 5, 6, 4, 1 \rangle$

inital setup	3	2	3	8	5	6	4	1
1st iteration	2	3	3	5	6	4	1	8
2nd iteration	2	3	3	5	4	1	6	8
3rd iteration	2	3	3	4	1	5	6	8

Parallel Sorting

- bubble sort (cont'd)
  - standard algorithm not very suitable for parallelisation ➔ partition of A into blocks of size n/p (for p processors) still to be processed sequentially
  - hence, different approach necessary: odd-even transposition
    - **idea**: sorting n elements (n is even) in n phases, each of which requires $n/2$ compare-exchange operations ➔ alternation between two phases
    - during odd phase, only elements with odd indices are compare-exchanged with their right neighbours, thus, the pairs

$$(a_1, a_2), (a_3, a_4), (a_5, a_6), \ldots, (a_{n-1}, a_n)$$

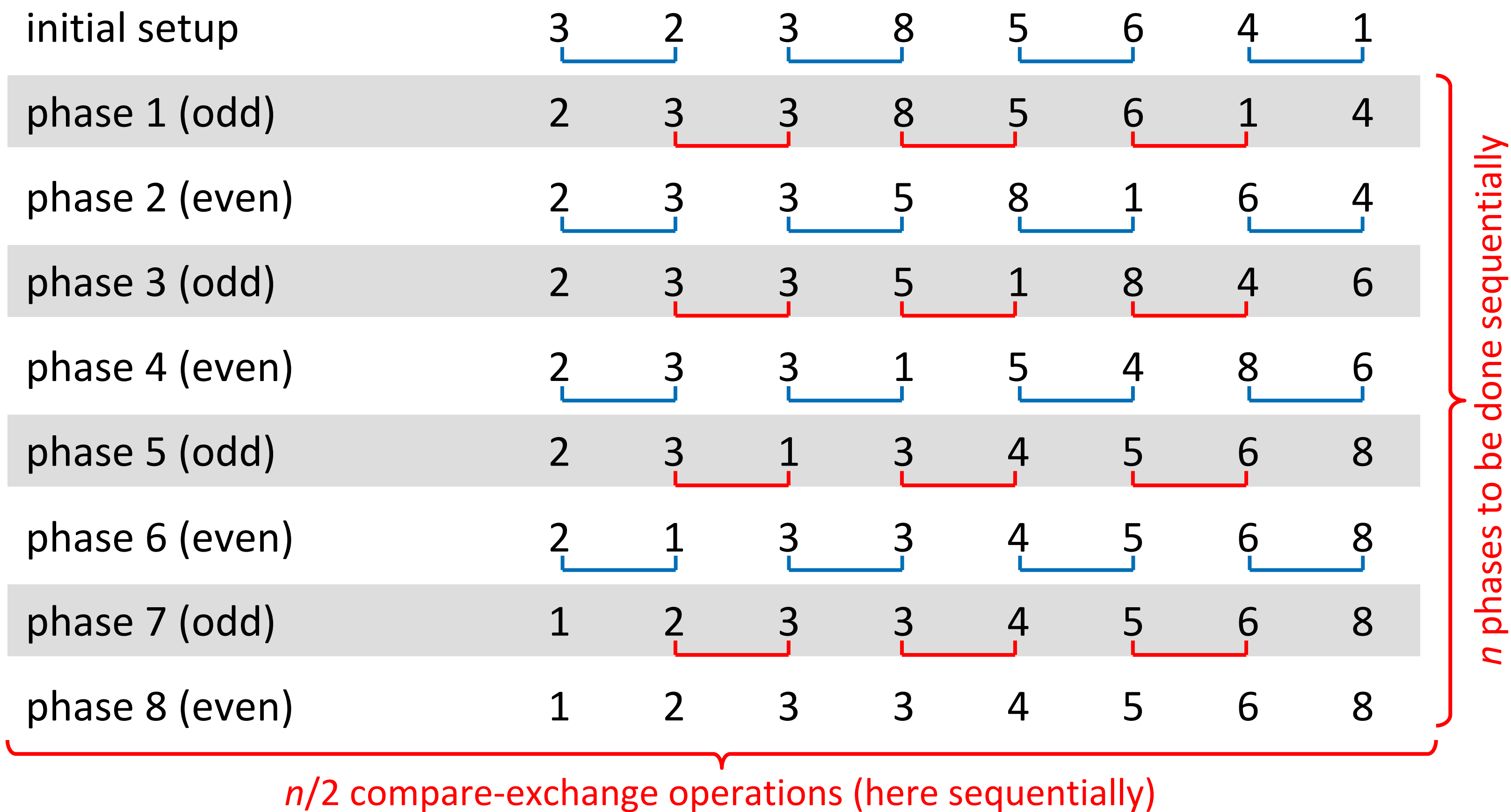
    - during even phase, only elements with even indices are compare-exchanged with their right neighbours, thus, the pairs

$$(a_2, a_3), (a_4, a_5), (a_6, a_7), \ldots, (a_{n-2}, a_{n-1})$$

    - after n phases of odd-even-exchanges, sequence A is sorted

Parallel Sorting

- bubble sort (cont'd)
  - example: odd-even-transposition for sorting *A* from before



Parallel Sorting

- bubble sort (cont'd)
  - parallelisation of odd-even-transposition
    - each process is assigned a block of n/p elements, which are sorted internally (e.g. using merge sort or quicksort) with a complexity of $O((n/p)\cdot\log_2 (n/p))$
    - afterwards, each processor executes p phases ($p/2$ odd and $p/2$ even ones), performing compare-split operations
    - at the end of these phases, sequence A is sorted (and distributed stored over p processes where process p_i holds block A_i with $a_i \le a_j$ for $a_i \in A_i$, $a_j \in A_j$ and $i < j$)
    - during each phase $O(n/p)$ comparisons are performed, thus, the total complexity of the parallel sort can be computed as

$$\underbrace{O((n/p)\cdot\log_2 (n/p))}_{\text{local sort}} + \underbrace{O(n)}_{\text{comparisons}} + \text{communication}$$

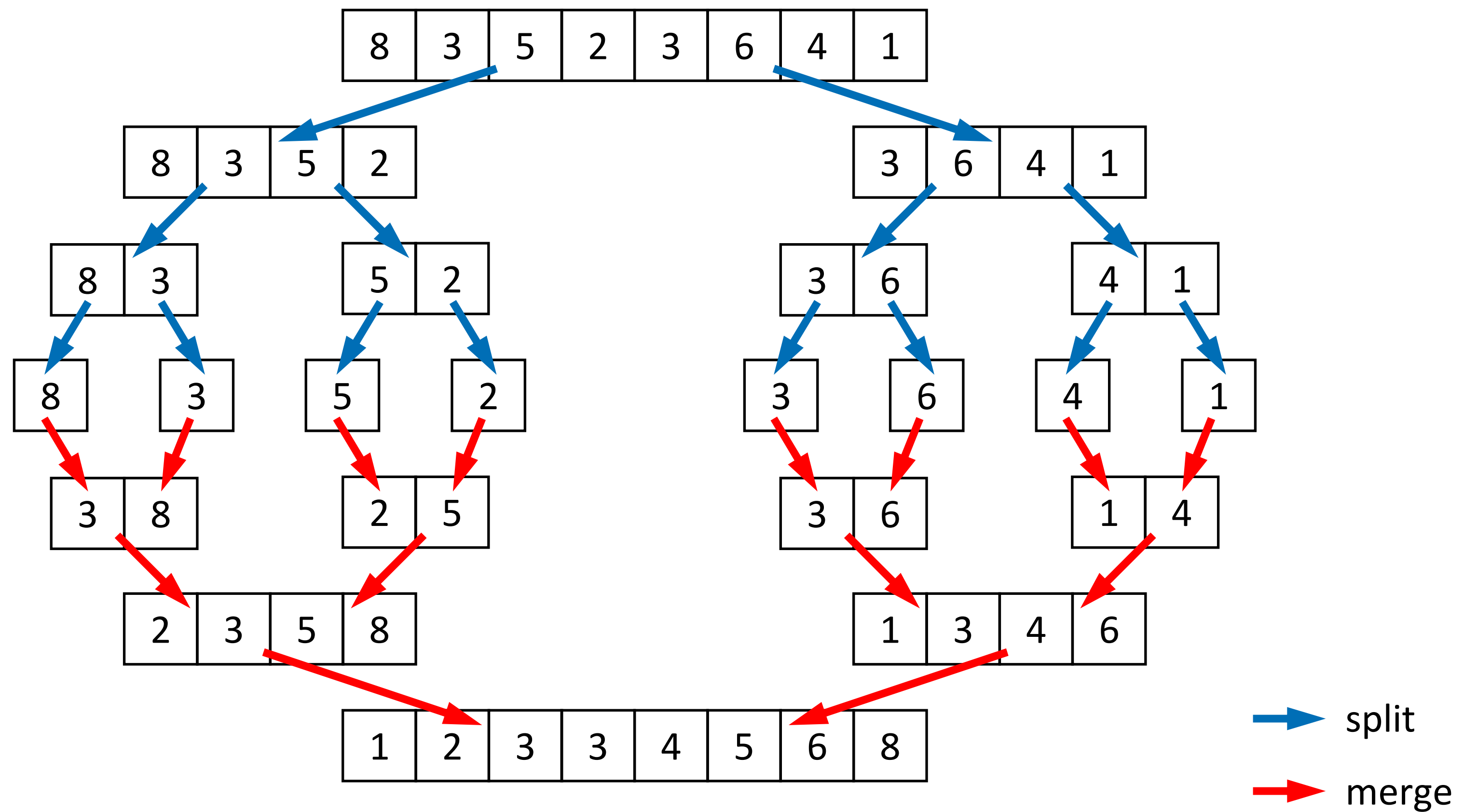
Parallel Sorting

- merge sort
  - comparison-based sorting algorithm of complexity $O(n \cdot \log_2 n)$
  - based on the divide-and-conquer (DAC) strategy
  - basic idea: construct a sorted list by merging two sorted lists
    1) recursively divide unsorted list into two sublists of about half the size
    2) merge the two sorted sublists back into one sorted list

```
begin procedure mergesort (list)
  if |list| = 1 then
    return list
  else
    divide list into sublists left and right
    left ← mergesort (left)
    right ← mergesort (right)
    list ← merge (left, right)
    return list
  fi
end
```

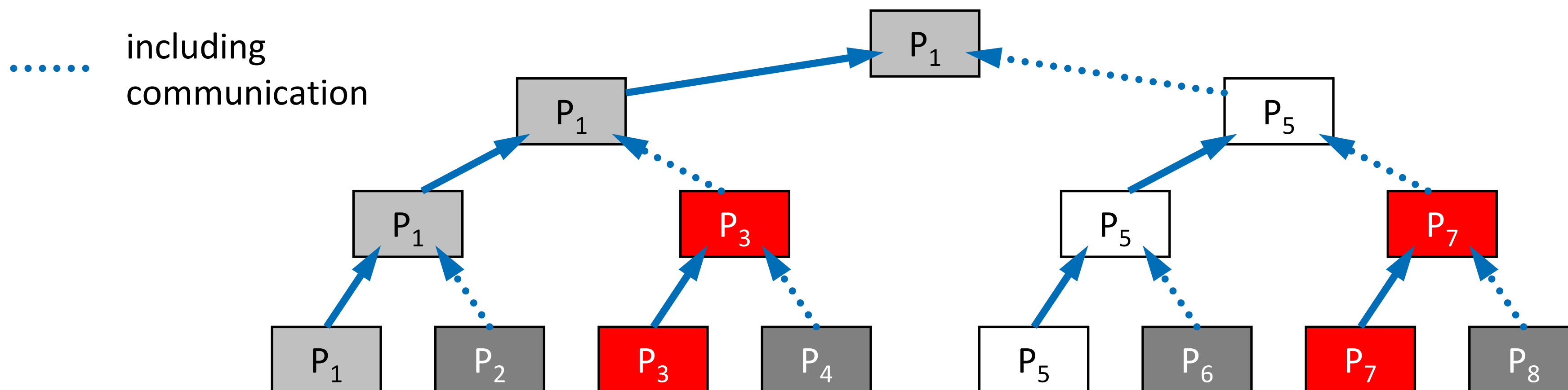
Parallel Sorting

- merge sort (cont'd)
  - example: merge sort for sequence $A = \langle 8, 3, 5, 2, 3, 6, 4, 1 \rangle$



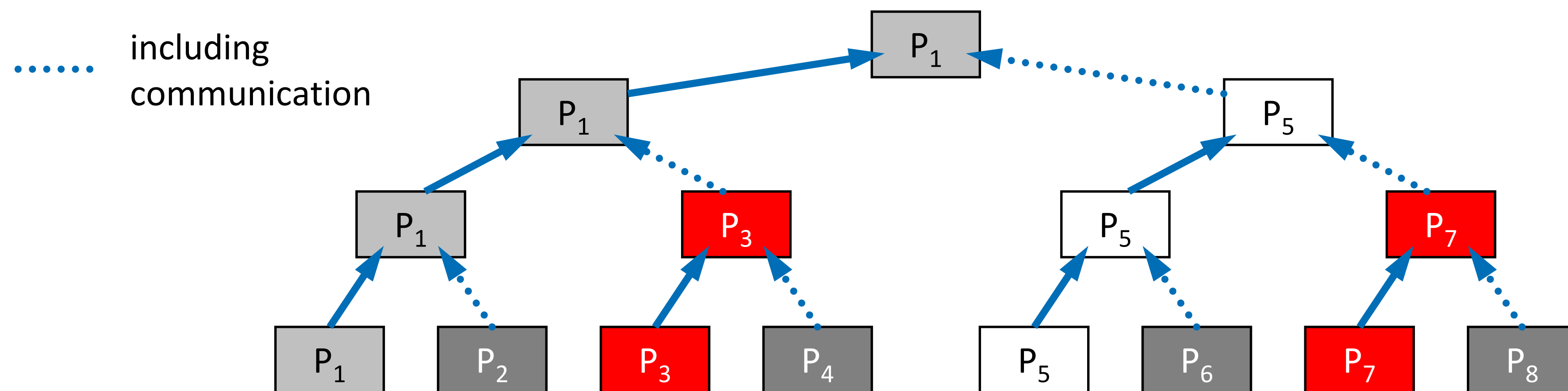
Parallel Sorting

- merge sort (cont'd)
  - parallelisation of merge sort: *naïve approach*
    - construct a binary processing tree of k leaf nodes and assign p processors, $p \geq k$, to tree nodes (i.e. inner and leaf nodes)
    - divide sequence A into blocks A_i of size n/k to be stored at leaf nodes
    - local sort of blocks A_i (sequentially) with complexity $\mathrm{O}((n/k) \cdot \log_2 (n/k))$
    - bottom-up parallel merge of sorted (sub)lists with complexity of $\mathrm{O}(n)$ (#compare operation at different levels ➔ $n + n/2 + n/4 + n/8 + \ldots$)



Parallel Sorting

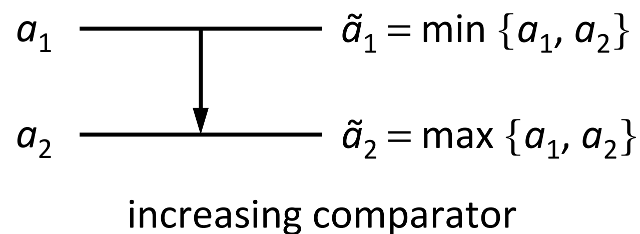
- merge sort (cont'd)
  - parallelisation of merge sort: *naïve approach*



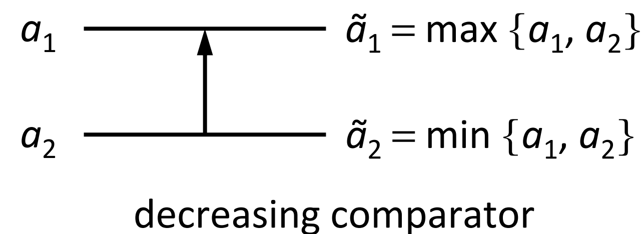
  - problem
    - number of processes halves in every step (cf. estimate of MINSKY)
    - number of elements (per processor) doubles in every step
    - hence, bad scalability / parallel performance to be expected
  - solution ➔ different strategy for parallel merge sort necessary

Parallel Sorting

- sorting networks
  - sorting networks are based on a comparison network model that sort n elements in significantly smaller than $O(n \cdot \log_2 n)$ operations
  - key component of a sorting network: comparator
    - device with two inputs a_1, a_2 and two outputs $\tilde{a}_1$, $\tilde{a}_2$
    - increasing comparator: $\tilde{a}_1 := \min \{a_1, a_2\}$ and $\tilde{a}_2 := \max \{a_1, a_2\}$
    - decreasing comparator: $\tilde{a}_1 := \max \{a_1, a_2\}$ and $\tilde{a}_2 := \min \{a_1, a_2\}$
  - sorting networks consist of several columns of such comparators, each column performing a permutation, thus the final column is sorted in increasing / decreasing order (➔ permutation networks)



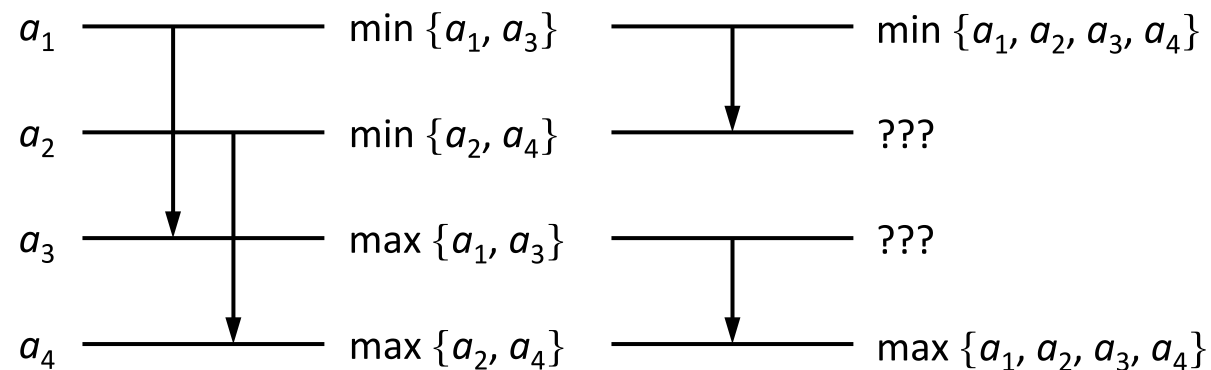
increasing comparator



decreasing comparator

Parallel Sorting

- sorting networks (cont'd)
  - first (naïve) idea of a parallel sorter

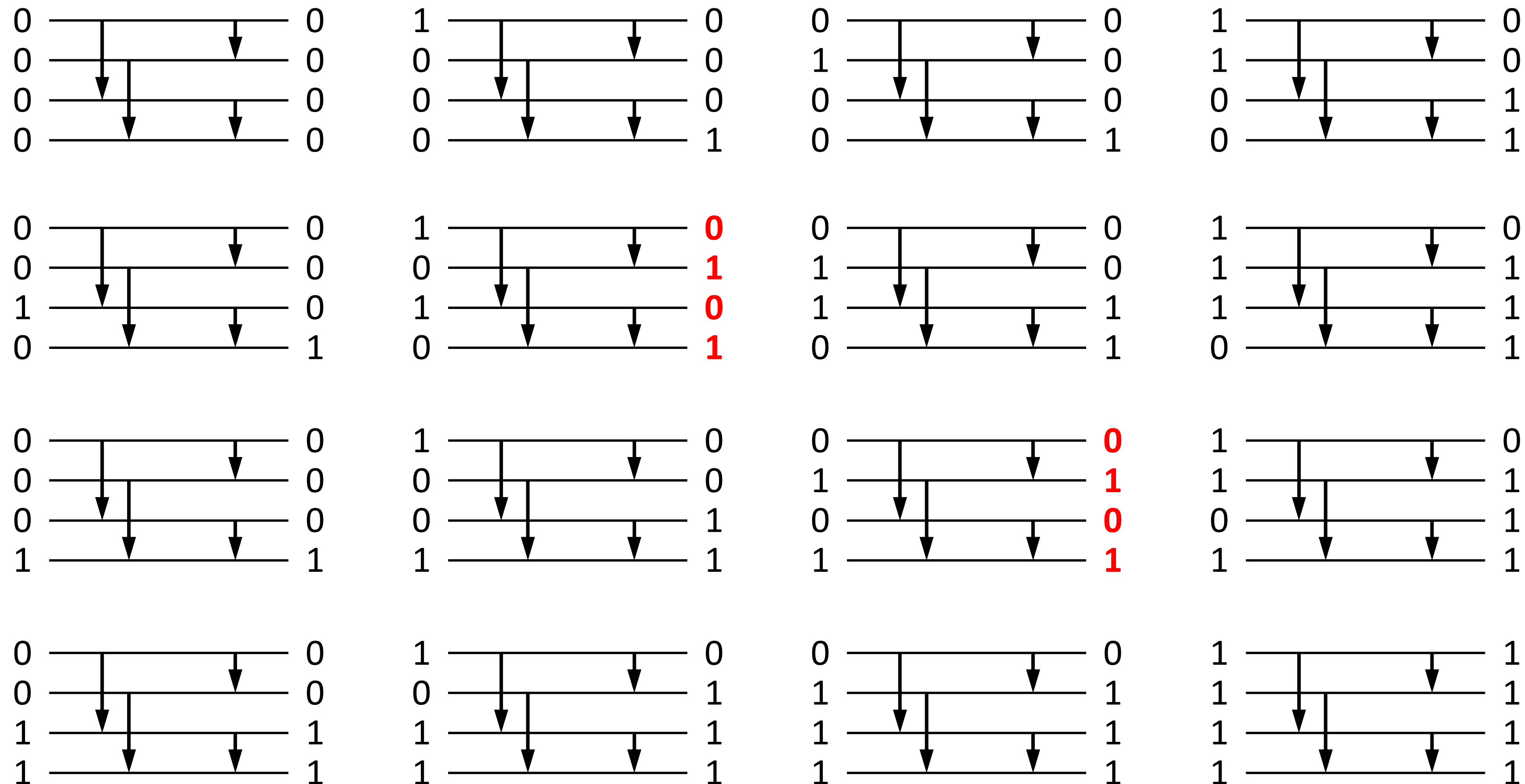


  - weak aspects
    - does not work for all permutations of input sequences
    - question: which permutations lead to incorrect results (n! tests)

Any network sorts correct any input sequence iff it sorts all possible 0-1-bit vectors ➔ only 2^n tests necessary (but for large n still impossible)

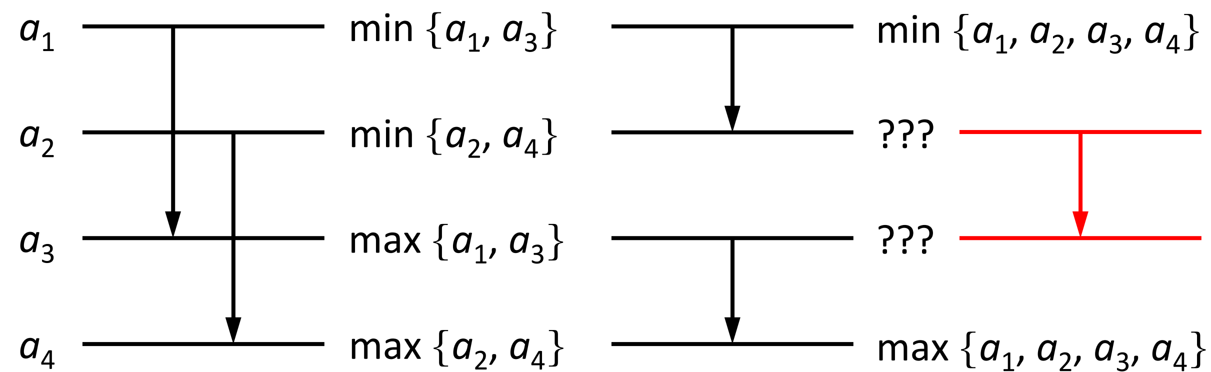
Parallel Sorting

- sorting networks (cont'd)
  - all possible 0-1-bit vectors for naïve parallel sorter

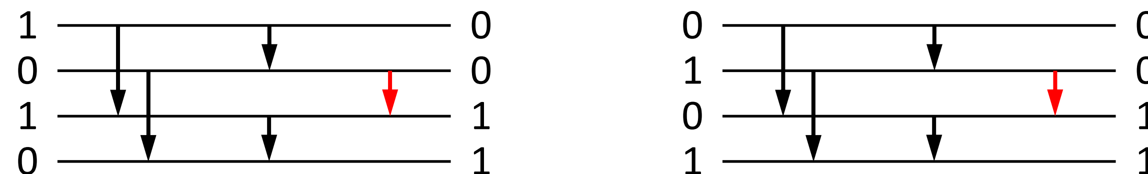


Parallel Sorting

- sorting networks (cont'd)
  - modified (naïve) parallel sorter



  - corresponding 0-1-bit vectors



  - for n inputs around $\mathrm{O}(n)$ steps necessary ➔ could this be done better and how to construct an efficient parallel sorter?

Parallel Sorting

- bitonic sort
  - a bitonic sorting network sorts n elements in $O(\log_2^2 n)$ operations
  - key task: rearrangement of a bitonic sequence into a sorted one
  - definition: bitonic sequence

    sequence $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ is bitonic <u>iff</u>

    1) there exists an index i, $0 \leq i \leq n-1$, such that $\langle a_0, \ldots, a_i \rangle$ is monotonically increasing and $\langle a_{i+1}, \ldots, a_{n-1} \rangle$ is monotonically decreasing, or

    2) there exists a cyclic shift of indices so that (1) is satisfied

  - example
    - $\langle 1, 2, 4, 7, 6, 0 \rangle$ first increases and then decreases
    - $\langle 8, 9, 2, 1, 0, 4 \rangle$ can be cyclic shifted to $\langle 0, 4, 8, 9, 2, 1 \rangle$

Parallel Sorting

- bitonic sort (cont'd)
  - let $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ be a bitonic sequence (w.l.o.g.) such that
    - $a_0 \le a_1 \le \ldots \le a_{n/2-1}$ and
    - $a_{n/2} \ge a_{n/2+1} \ge \ldots \ge a_{n-1}$
  - consider the following subsequences of A
    - $A_1 = \langle \min\{a_0, a_{n/2}\}, \min\{a_1, a_{n/2+1}\}, \ldots, \min\{a_{n/2-1}, a_{n-1}\} \rangle$
    - $A_2 = \langle \max\{a_0, a_{n/2}\}, \max\{a_1, a_{n/2+1}\}, \ldots, \max\{a_{n/2-1}, a_{n-1}\} \rangle$
  - in sequence A_1, there is an element $a_i = \min\{a_i, a_{n/2+i}\}$ such that all elements before a_i are from the increasing part of A and all elements after a_i are from the decreasing part of A
  - also, in sequence A_2, there is an element $\hat{a}_i = \max\{a_i, a_{n/2+i}\}$ such that all elements before $\hat{a}_i$ are from the decreasing part of A and all elements after $\hat{a}_i$ are from the increasing part of A
  - hence, sequences A_1 and A_2 are bitonic

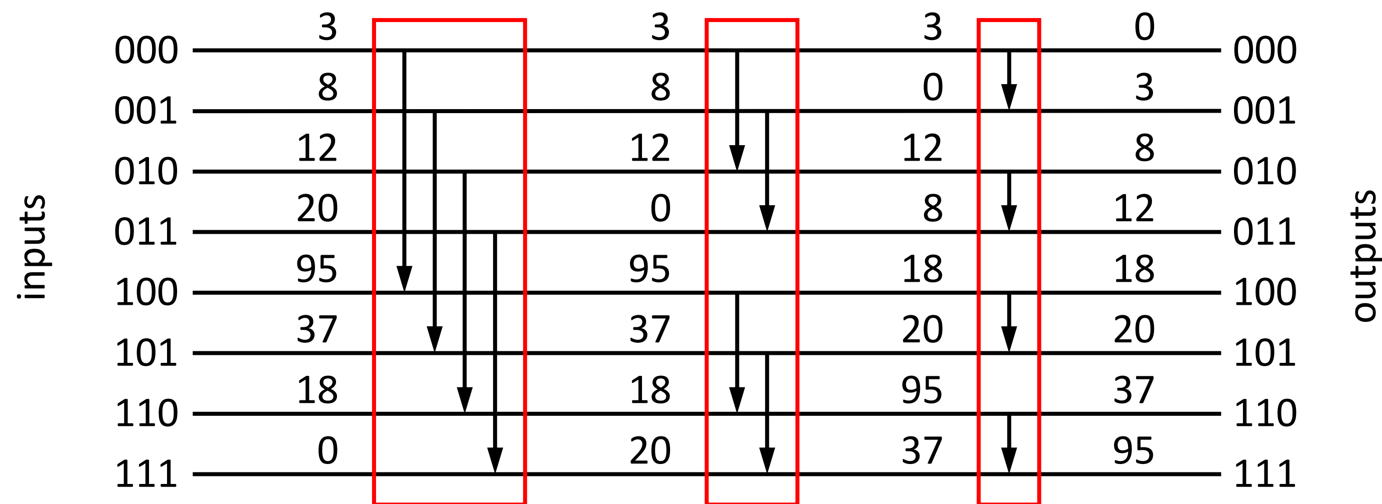
Parallel Sorting

- bitonic sort (cont'd)
  - furthermore, $A_1 \leq A_2$ because a_i is greater equal to all elements of A_1, $\hat{a}_i$ is less equal to all elements of A_2, and $\hat{a}_i$ is greater equal a_i
  - hence, the initial problem of rearranging a bitonic sequence of size n was reduced to that of rearranging two smaller bitonic sequences of size $n/2$ and concatenating the results
  - this operation is further referred to as bitonic split (although assuming A_1 and A_2 had increasing / decreasing sequences of the same length, the bitonic split operation holds for any bitonic sequence)

  - the recursive usage of the bitonic split operation until all obtained subsequences are of size one leads to a sorted output in increasing order
  - sorting a bitonic sequence using bitonic splits is called bitonic merge

Parallel Sorting

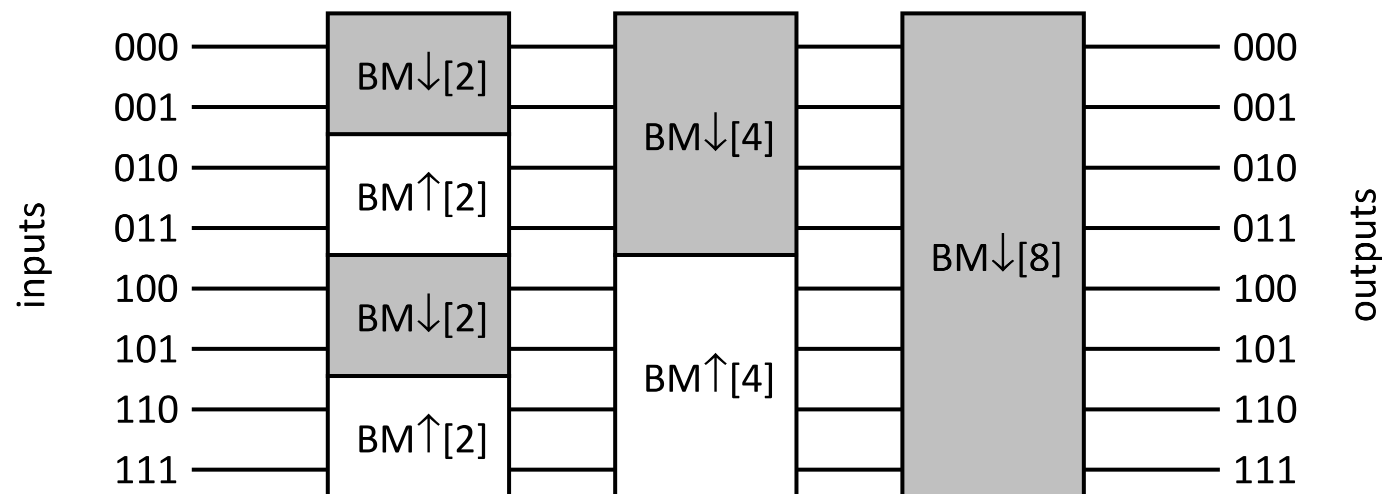
- bitonic sort (cont’d)
  - example: bitonic merging network with 8 inputs (BM↓[8])



initial sequence	3	8	12	20	95	37	18	0
1st bitonic split	3	8	12	0	95	37	18	20
2nd bitonic split	3	0	12	8	18	20	95	37
3rd bitonic split	0	3	8	12	18	20	37	95

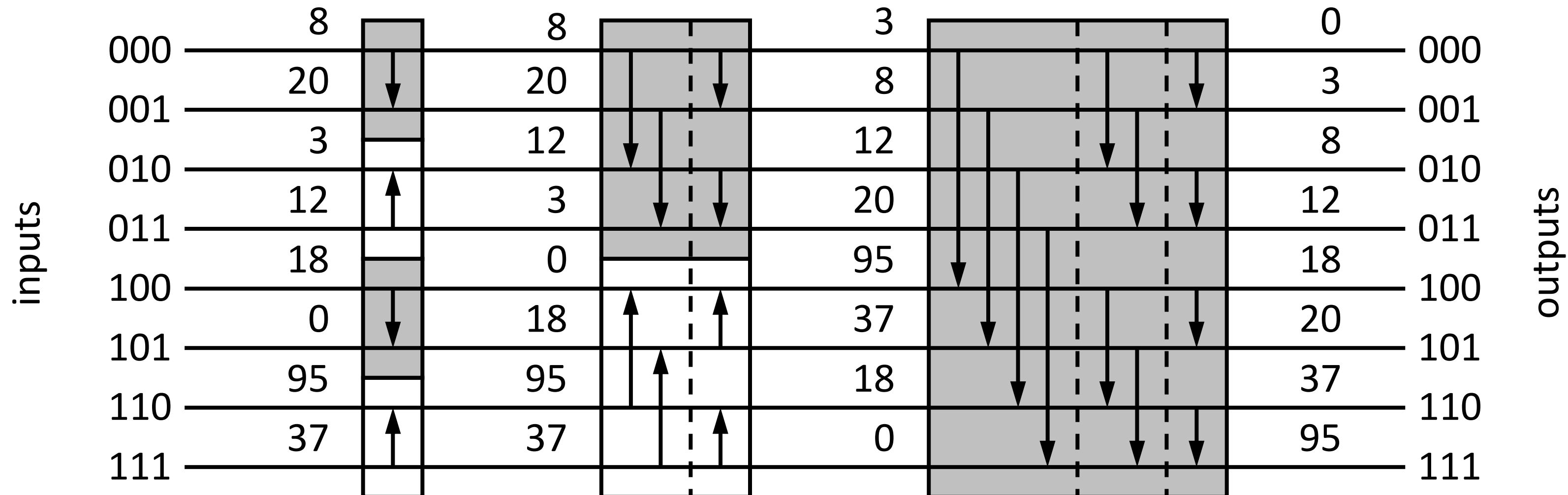
Parallel Sorting

- bitonic sort (cont'd)
  - for sorting bitonic sequence in decreasing order, ↓ comparators have to be replaced by ↑ comparators ➔ BM↑[8]
  - problem: how to get a bitonic sequence $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ out of n unordered elements
  - solution: construct A by repeatedly merging bitonic sequences of increasing length (here, the last bitonic merge (BM↓[8]) sorts the input)



Parallel Sorting

- bitonic sort (cont'd)
  - example: bitonic sorting network with 8 inputs



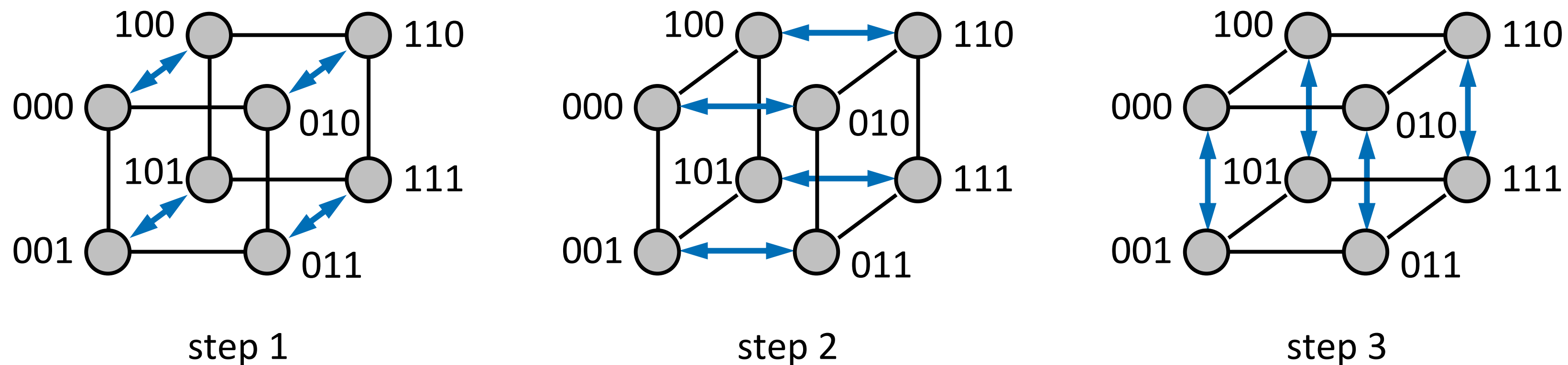
  - depth $d(n)$ of a bitonic sorting network with n inputs can be computed by the following recursion

$$d(n) = d(n/2) + \log_2 n$$

  - hence, $d(n) = \sum_{i=1}^{\log_2 n} i = (\log_2^2 n + \log_2 n) / 2 = \mathrm{O}(\log_2^2 n)$

Parallel Sorting

- bitonic sort (cont'd)
  - the bitonic algorithm is communication intensive ➔ a proper mapping must take into account the underlying network topology
    - hypercube: compare-exchange operations take only place between nodes whose labels differ in the *k*-th bit for step *k*



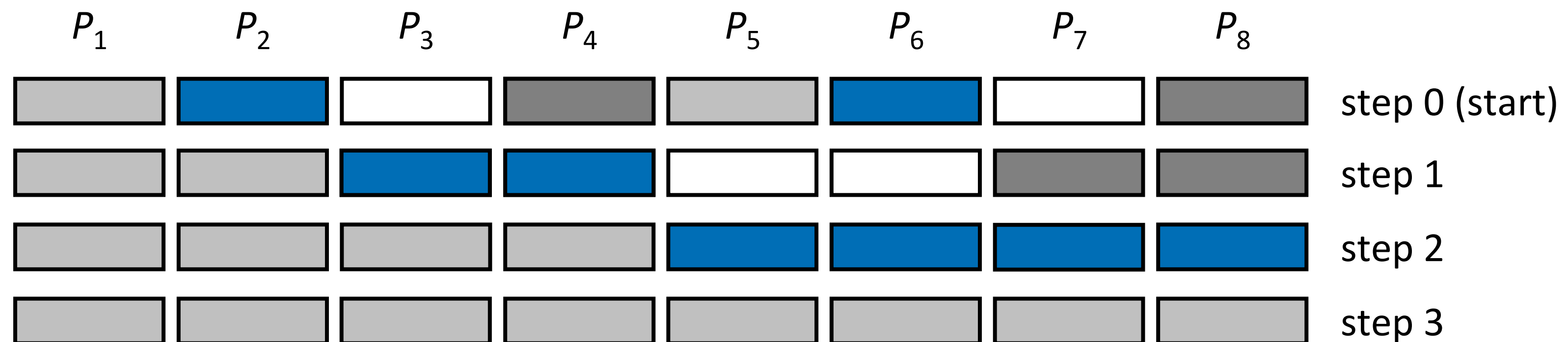
    - mesh: there exist several possibilities for a proper mapping

Parallel Sorting

- again: merge sort
  - here, different approach for parallelisation of merge sort using a sorting network instead of a binary tree
  - idea
    - divide sequence A into blocks A_i of size n/p
    - local sort of blocks A_i (sequentially) with complexity $O((n/p)\cdot\log_2(n/p))$
    - parallel merge
      1) starting point: p sorted lists each distributed over one processor
      2) merging two lists using compare-split operations leads to $p/2$ sorted lists each distributed over two processors
      3) repeatedly executing step 2 finally leads to one sorted list distributed over p processors

Parallel Sorting

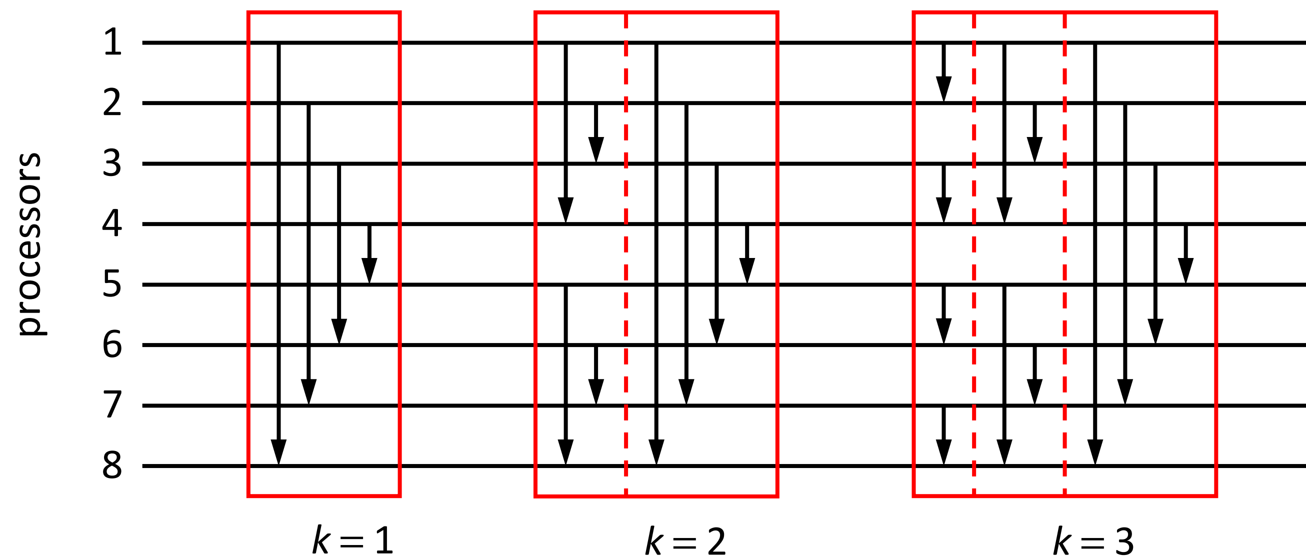
- again: merge sort (cont'd)
  - example: repeated merge of sorted lists for $p = 8$



  - there is a total of $(\log_2 p)$ steps, where in the k-th step each processor performs k compare-split operations with its neighbours to obtain (parts of) a sorted list distributed over 2^k nodes
  - hence, the parallel merge can be implemented by a sorting network with a complexity of $\mathrm{O}(\log_2^2 p)$ compare-split operations

Parallel Sorting

- again: merge sort (cont'd)
  - example: parallel merge for $p = 8$ processors with $k = 3$ steps



  - total complexity of the parallel merge sort can be computed as

$$\underbrace{O((n/p)\cdot\log_2 (n/p))}_{\text{local sort}} + \underbrace{O((n/p)\cdot\log_2^2 p)}_{\text{comparisons}} + \text{communication}$$

- overview
  - parallel matrix operations ✓
  - iterative solvers: parallel JACOBI and GAUSS-SEIDEL ✓
  - parallel sorting ✓