Parallel Programming and High-Performance Computing

Part 7: Examples of Parallel Algorithms

Dr. Ralf-Peter Mundani
CeSIM / IGSSE

Overview

- matrix operations
- JACOBI and GAUSS-SEIDEL iterations
- sorting

Everything that can be invented
has been invented.
—Charles H. Duell
commissioner U.S. Office of Patents, 1899

Matrix Operations

- reminder: matrix
  - 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 $A,\ B,\ \text{and}\ C \in K^{N \times M}$
    2) $A \cdot b = c$ with $A \in K^{N \times M},\ b \in K^{M},\ c \in K^{N}$
    3) $A \cdot B = C$ with $A \in K^{N \times M},\ B \in K^{M \times L},\ \text{and}\ C \in K^{N \times L}$
  - 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?

Matrix Operations

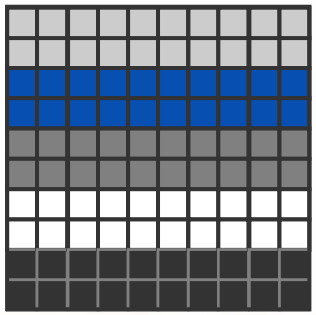
- matrix-vector multiplication
  - appearances
    - systems of linear equations (SLE) $A \cdot x = b$
    - iterative methods for solving SLEs (conjugate gradient, e. g.)
    - implementation of neural networks (determination of output values, training neural networks)
  - standard sequential algorithm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

```
for (i = 0; i < N; ++i) {
    c[i] = 0;
    for (j = 0; j < N; ++j) {
        c[i] = c[i] + A[i][j]*b[j];
    }
}
```

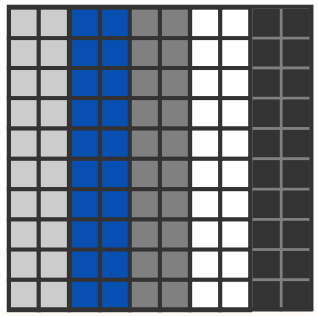
  - for full matrix A this algorithm has a complexity of $O(N^2)$

Matrix Operations

- matrix-vector multiplication (cont'd)
  - for a parallel implementation, there exist three main options to distribute the data among P processors
    - *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

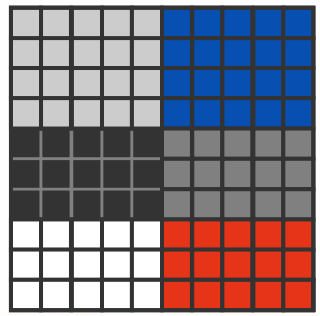


row-wise



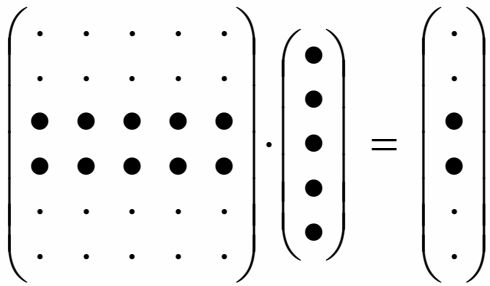
column-wise

checkerboard



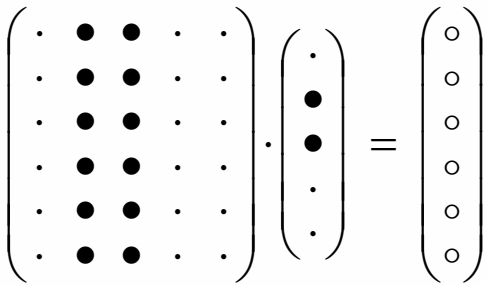
Matrix Operations

- 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
      - build and replicate entire vector c (gather-to-all, e. g.)
    - complexity of $O(N^2/P)$ multiplications / additions for P processes



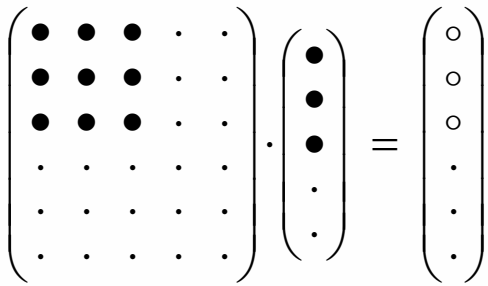
Matrix Operations

- matrix-vector multiplication (cont'd)
  - column-wise block-striped decomposition
    - less straightforward approach
      - each process gets some columns of A and respective elements of 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



Matrix Operations

- matrix-vector multiplication (cont'd)
  - 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)



Matrix Operations

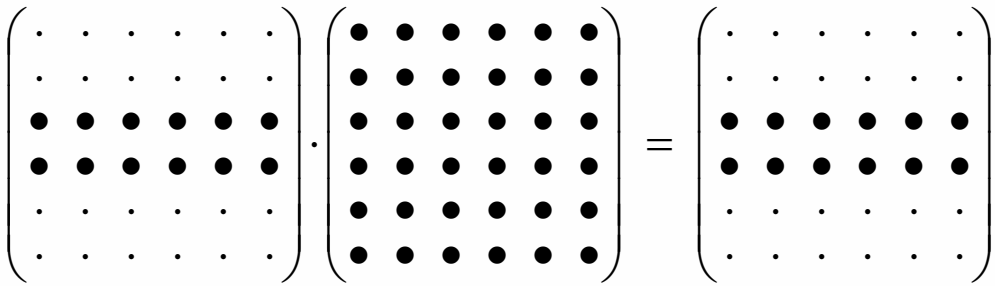
- matrix multiplication
  - appearances
    - computational chemistry (computing changes of state, e. g.)
    - signal processing (DFT, e. g.)
  - standard sequential algorithm for A, B $\in K^{N \times N}$

```
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];
        }
    }
}
```

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

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



  - 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

Matrix Operations

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

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

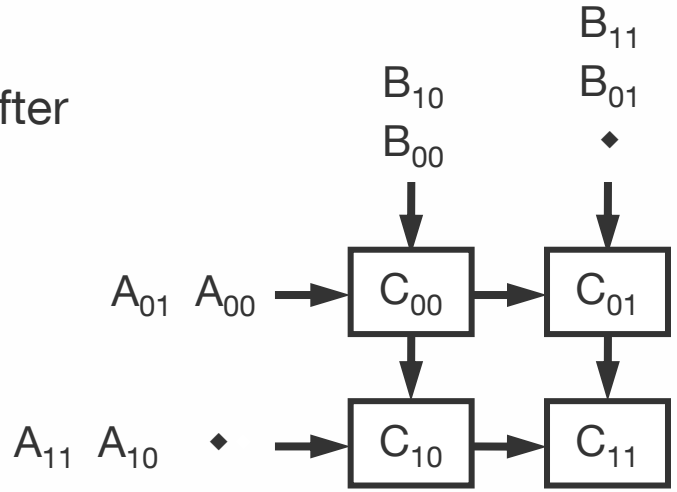
    - hence, the matrix multiplication can be computed as follows

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

    - 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)

Matrix Operations

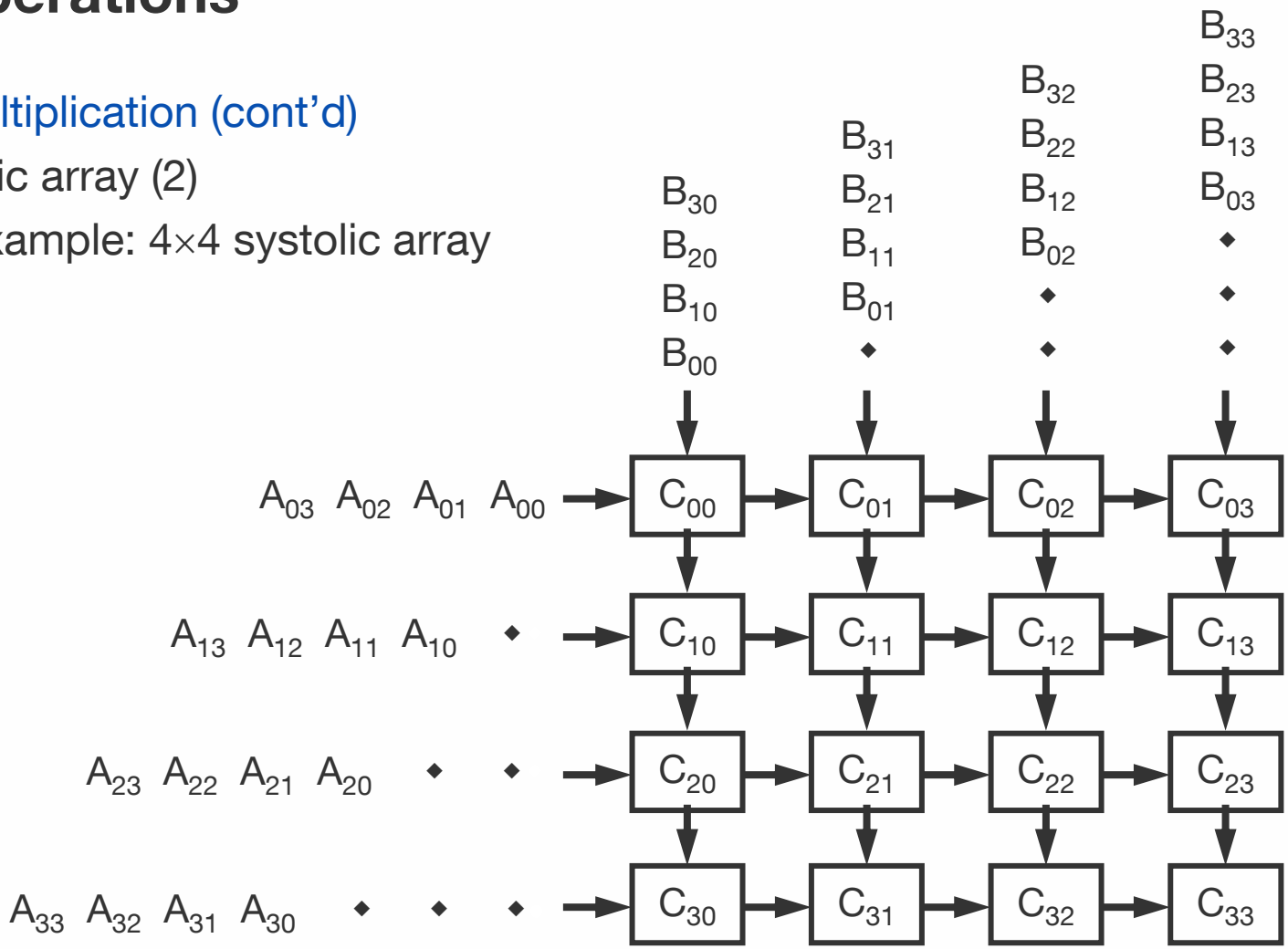
- matrix multiplication (cont'd)
  - systolic array (1)
    - again, matrices A and B are divided into submatrices
    - submatrices are pumped through a systolic array in various directions at regular intervals
      - data meet at internal nodes to be processed
      - same data is passed onward
    - drawback: full parallelisation only after some initial delay
    - example: 2×2 systolic array



◆ means one block delay

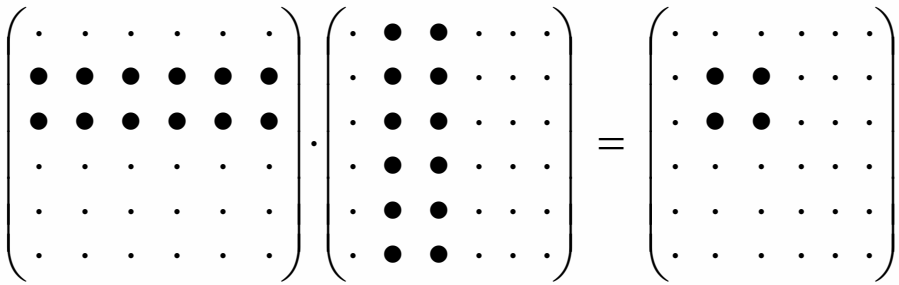
Matrix Operations

- matrix multiplication (cont'd)
  - systolic array (2)
    - example: 4×4 systolic array



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
    - 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



Overview

- matrix operations
- JACOBI and GAUSS-SEIDEL iterations
- sorting

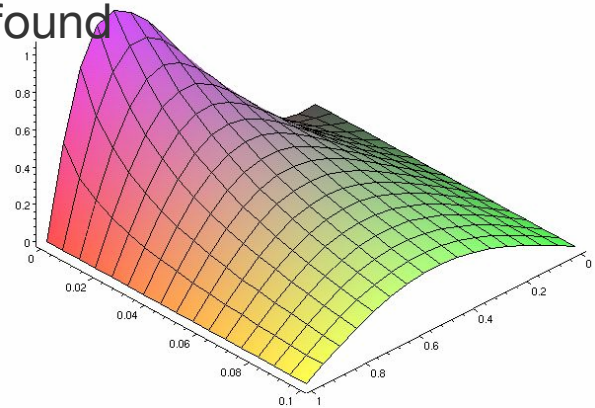
JACOBI and GAUSS-SEIDEL Iterations

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

(1) $$-\Delta u(x,y) = -\frac{\partial^2 u(x,y)}{\partial x^2} - \frac{\partial^2 u(x,y)}{\partial y^2} = f(x,y) \qquad \text{for } (x,y) \in \Omega$$

on the unit square $\Omega =]0,1[^2$ 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
    - …



JACOBI and GAUSS-SEIDEL Iterations

- discretisation
  - for solving our example PDE, a discretisation is necessary
  - typical discretisation techniques
    - finite differences
    - finite volumes
    - finite elements
  - finite difference discretisation (forward and backward differences) for the second derivatives in (1) for a mesh width h leads to

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

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

Jacobi and Gauss-Seidel Iterations

- discretisation (cont'd)
  - for computational solution of our problem a 2D grid is necessary
    - equidistant grid with (N+1)×(N+1) grid points, N = 1/h
    - $u_{i,j} \approx u(i \cdot h, j \cdot h)$ with i, j = 0, …, N
  - hence, (2) can be written as

(3) $$\frac{\partial^2 u(x,y)}{\partial x^2} \approx \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2}, \qquad \frac{\partial^2 u(x,y)}{\partial y^2} \approx \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2}$$

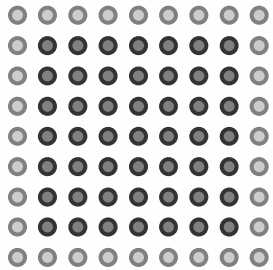
  - with (3) and an appropriate discretisation of f(x,y) follows

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

  - resulting equation on the boundary

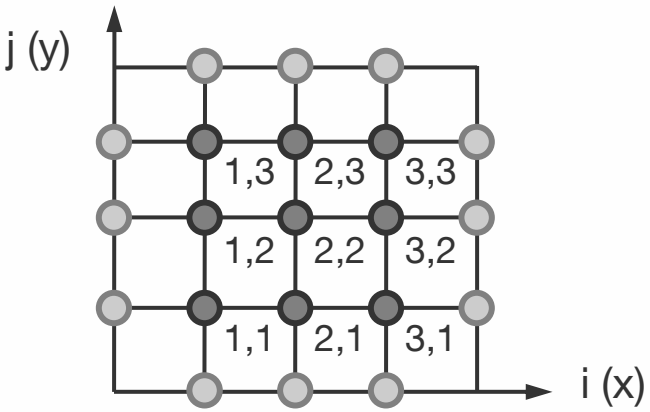
$$u_{i,j} = g_{i,j} \qquad i, j = 0 \text{ or } i, j = N$$

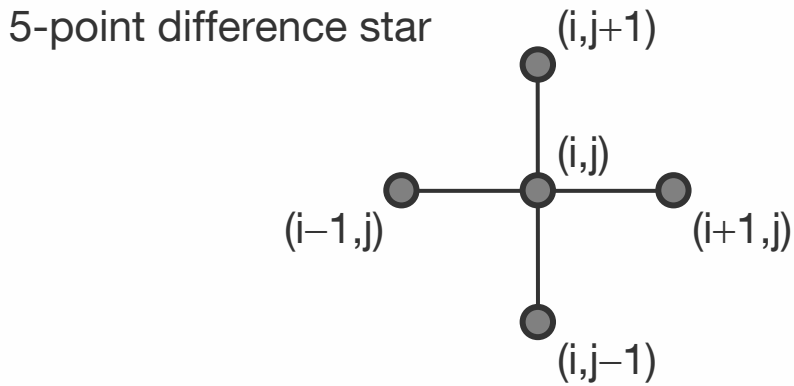
- inner point
- boundary point



JACOBI and GAUSS-SEIDEL Iterations

- system of linear equations
  - for each inner point there is one linear equation according to (4)
  - equations in points next to the boundary (i. e. i, j = 1 or i, j = N–1) access boundary values $g_{i,j}$
    - these are shifted to the right-hand side of the equation
    - hence, all unknowns are located left, all known quantities right
  - assembling of the overall vector of unknowns by lexicographic row-wise ordering





Jacobi and Gauss-Seidel Iterations

- system of linear equations (cont'd)
  - this results to a system of linear equations $A \cdot x = b$
    - with $(N-1)^2$ equations in $(N-1)^2$ unknowns
    - matrix A has block-tridiagonal structure and is sparse

(5)

$$\underbrace{\begin{pmatrix} 4 & -1 & & -1 & & & \\ -1 & 4 & -1 & & & & \\ & & & & \ddots & & \\ -1 & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & -1 \\ & \ddots & & & & & \\ & & & & -1 & 4 & -1 \\ & & & -1 & & -1 & 4 \end{pmatrix}}_{=A} \cdot \underbrace{\begin{pmatrix} u_{1,1} \\ \vdots \\ u_{N-1,1} \\ u_{1,2} \\ \vdots \\ \vdots \\ u_{N-1,N-1} \end{pmatrix}}_{=x} = \underbrace{\begin{pmatrix} f_{1,1} \\ \vdots \\ f_{N-1,1} \\ f_{1,2} \\ \vdots \\ \vdots \\ f_{N-1,N-1} \end{pmatrix}}_{=b}$$

JACOBI and GAUSS-SEIDEL Iterations

- solving large sparse SLEs
  - schoolbook method for solving SLEs: GAUSSian elimination
    - direct solver that provides the exact solution
    - has a complexity of $O(M^3)$ for M unknowns (!)
    - does not exploit sparsity 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
    - has a complexity of $O(M)$ operations for a single iteration
    - typically much less than $O(M^2)$ iteration steps needed ➔ ideal case of $O(1)$ steps for *multigrid* or *multilevel* methods
    - basic methods (number of steps depending on M)
      - relaxation methods: JACOBI, GAUSS-SEIDEL, SOR
      - minimisation methods: steepest descent, CG

JACOBI and GAUSS-SEIDEL Iterations

- JACOBI iteration
  - decompose matrix A in its diagonal part D_A, its upper triangular part U_A, and its lower triangular part L_A

$$A = L_A + D_A + U_A$$

  - starting with

$$b = A \cdot x = D_A \cdot x + (L_A + U_A) \cdot x$$

    and writing $b = D_A \cdot x^{(T+1)} + (L_A + U_A) \cdot x^{(T)}$ with $x^{(T)}$ denoting the approximation to x after T steps of the iteration leads to the following iterative scheme

$$x^{(T+1)} = -D_A^{-1} \cdot (L_A + U_A) \cdot x^{(T)} + D_A^{-1} \cdot b = x^{(T)} + D_A^{-1} \cdot r^{(T)}$$

    where the residual is defined as $r^{(T)} = b - A \cdot x^{(T)}$

JACOBI and GAUSS-SEIDEL Iterations

- JACOBI iteration (cont'd)
  - algorithmic form of the JACOBI iteration

```
for (T = 0, 1, 2, ...) {
    for (k = 1; k ≤ M; ++k) {
        x_k^(T+1) = 1/A_k,k *(b_k − Σ_j≠k A_k,j * x_j^(T));
    }
}
```

  - for our example with matrix A according to (5) this means

```
for (T = 0, 1, 2, ...) {
    for (j = 1; j < N; ++j) {
        for (i = 1; i < N; ++i) {
            u_i,j^(T+1) = 1/4*(u_i−1,j^(T) + u_i,j−1^(T) + u_i+1,j^(T) + u_i,j+1^(T) − h² * f_i,j);
        }
    }
}
```

JACOBI and GAUSS-SEIDEL Iterations

- GAUSS-SEIDEL iteration
  - same decomposition of matrix A as for JACOBI

$$A = L_A + D_A + U_A$$

  - starting with

$$b = A \cdot x = (D_A + L_A) \cdot x + U_A \cdot x$$

and writing $b = (D_A + L_A) \cdot x^{(T+1)} + U_A \cdot x^{(T)}$ leads to the following iterative scheme

$$x^{(T+1)} = -(D_A + L_A)^{-1} \cdot U_A \cdot x^{(T)} + (D_A + L_A)^{-1} \cdot b = x^{(T)} + (D_A + L_A)^{-1} \cdot r^{(T)}$$

where the residual is defined as $r^{(T)} = b - A \cdot x^{(T)}$

JACOBI and GAUSS-SEIDEL Iterations

- GAUSS-SEIDEL iteration (cont'd)
  - algorithmic form of the GAUSS-SEIDEL iteration

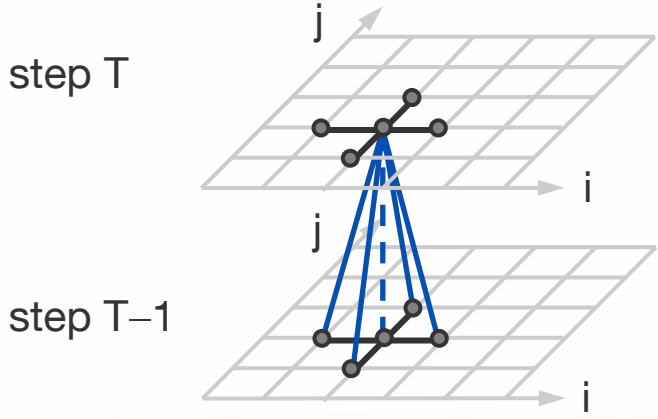
```
for (T = 0, 1, 2, ...) {
    for (k = 1; k ≤ M; ++k) {
        x_k^(T+1) = 1/A_k,k *(b_k − Σ_j=1^k-1 A_k,j * x_j^(T+1) − Σ_j=k+1^M A_k,j * x_j^(T));
    }
}
```

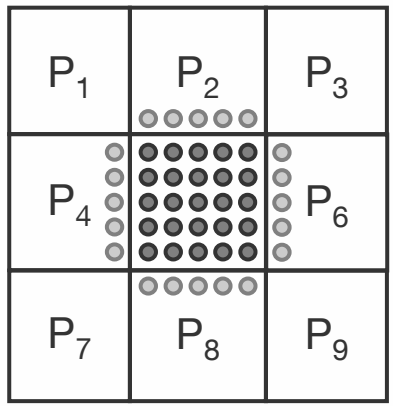
  - for our example with matrix A according to (5) this means

```
for (T = 0, 1, 2, ...) {
    for (j = 1; j < N; ++j) {
        for (i = 1; i < N; ++i) {
            u_i,j^(T+1) = 1/4*(u_i−1,j^(T+1) + u_i,j−1^(T+1) + u_i+1,j^(T) + u_i,j+1^(T) − h² * f_i,j);
        }
    }
}
```

JACOBI and GAUSS-SEIDEL Iterations

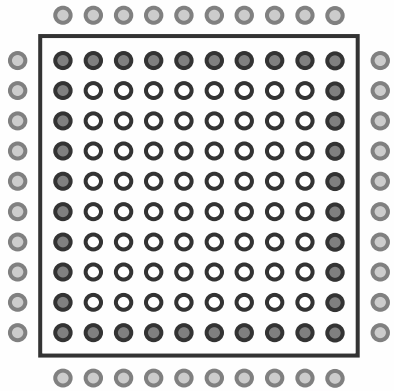
- parallelisation of JACOBI iteration
  - neither JACOBI nor GAUSS-SEIDEL are used today very frequently for solving large SLEs (they are too slow)
  - nevertheless, the algorithmic aspects are still of interest
  - a parallel JACOBI is quite straightforward
    - in iteration step T only values from step T–1 are used
    - hence, all updates of one step can be made in parallel
    - furthermore, subdivide the domain into strips or blocks, e. g.





JACOBI and GAUSS-SEIDEL Iterations

- parallelisation of JACOBI iteration (cont'd)
  - for its computations, each processor P needs
    - a subset of boundary values (if P is adjacent to the boundary)
    - one row / column of values from P's neighbouring processes
    - a global / local termination condition
  - hence, each processor has to execute the following algorithm
    1) update all local approximate values $u_{i,j}^{(T)}$ to $u_{i,j}^{(T+1)}$
    2) send all updates (●) to the respective neighbouring processes
    3) receive all necessary updates (◌) from neighbouring processes
    4) compute local residual values and perform a reduce-all for global residual
    5) continue if global residual is larger than some threshold value ε

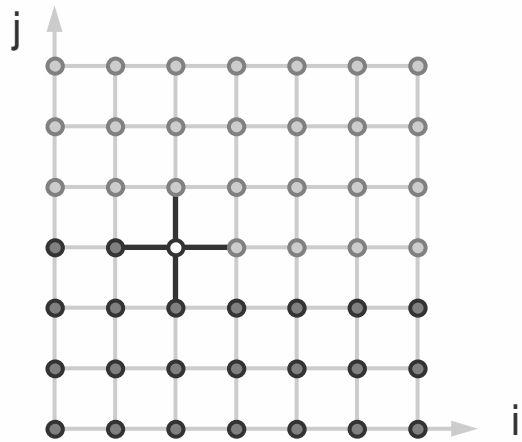


JACOBI and GAUSS-SEIDEL Iterations

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

updated values (step T)

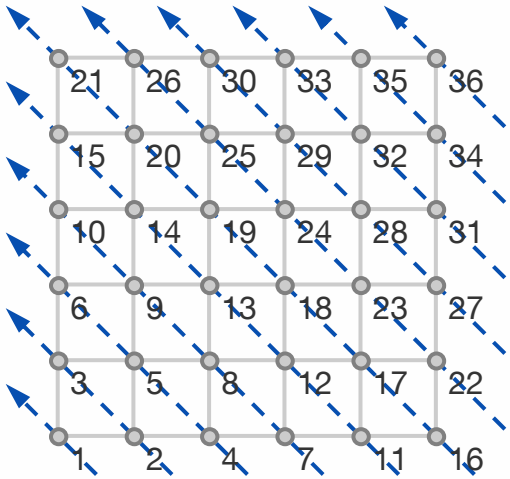
old values (step T–1)



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

JACOBI and GAUSS-SEIDEL Iterations

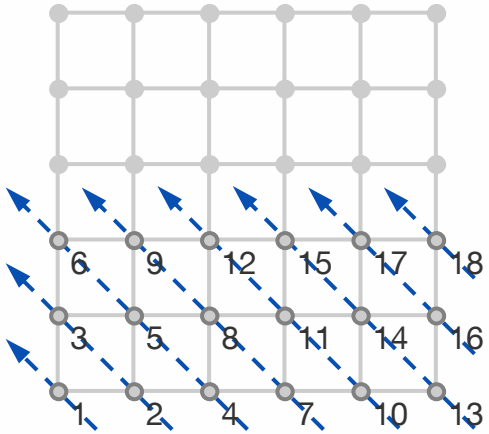
- parallelisation of GAUSS-SEIDEL iteration (cont'd)
  - wavefront ordering (1)
    - diagonal ordering of updates ➔ all values along a diagonal line can be updated in parallel; single diagonal lines still have to be processed sequentially
    - problem: for P = N processors there are P^2 updates that 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					

JACOBI and GAUSS-SEIDEL Iterations

- parallelisation of GAUSS-SEIDEL iteration (cont'd)
  - wavefront ordering (2)
    - 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 $K \cdot P/(K+1)$

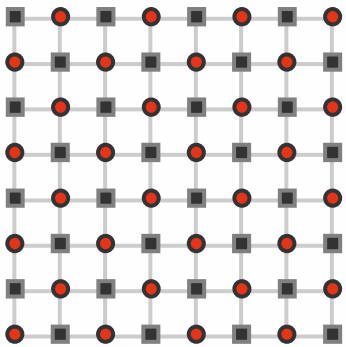


P_1:	1	2	4	7	10	13	16	18
P_2:		3	5	8	11	14	17	
P_3:			6	9	12	15		

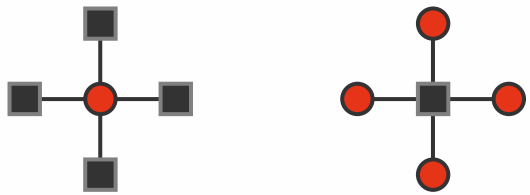
here, K = 2 ➔ speed-up S(p) = 2P/3

JACOBI and GAUSS-SEIDEL Iterations

- 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 and some black points ➔ two sequential steps necessary, but perfect parallelism within each of them



red-black ordering



5-point star for red (left) and black (right) grid points

Overview

- matrix operations
- JACOBI and GAUSS-SEIDEL iterations
- sorting

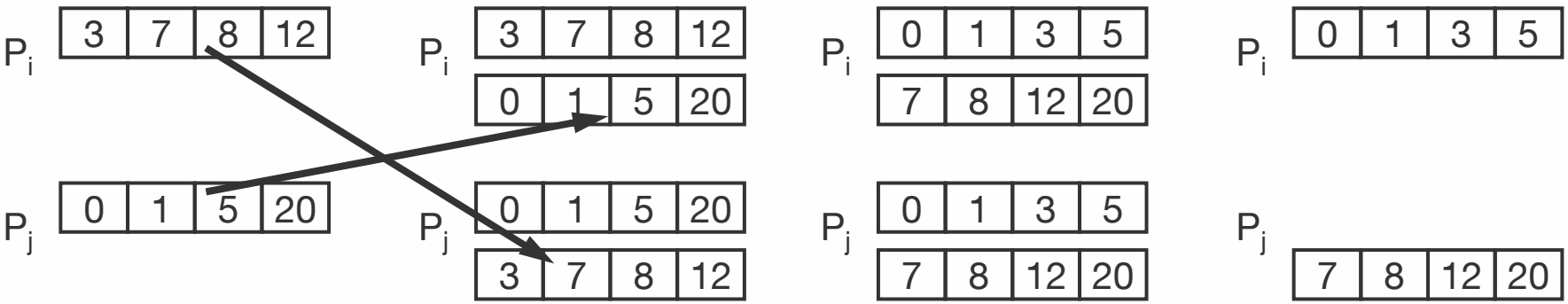
Sorting

- 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 $O(N \cdot \log N)$

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



P_i sends block A_i to P_j and vice versa | merge | split

Sorting

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

```
for (i = N-1; i ≥ 1; --i) {
    for (j = 0; j < i; ++j) {
        compare-exchange (a[j], a[j+1]);
    }
}
```

  - example: iterations i = 1, 2, 3 for sorting A = ⟨3, 2, 3, 8, 5, 6, 4, 1⟩

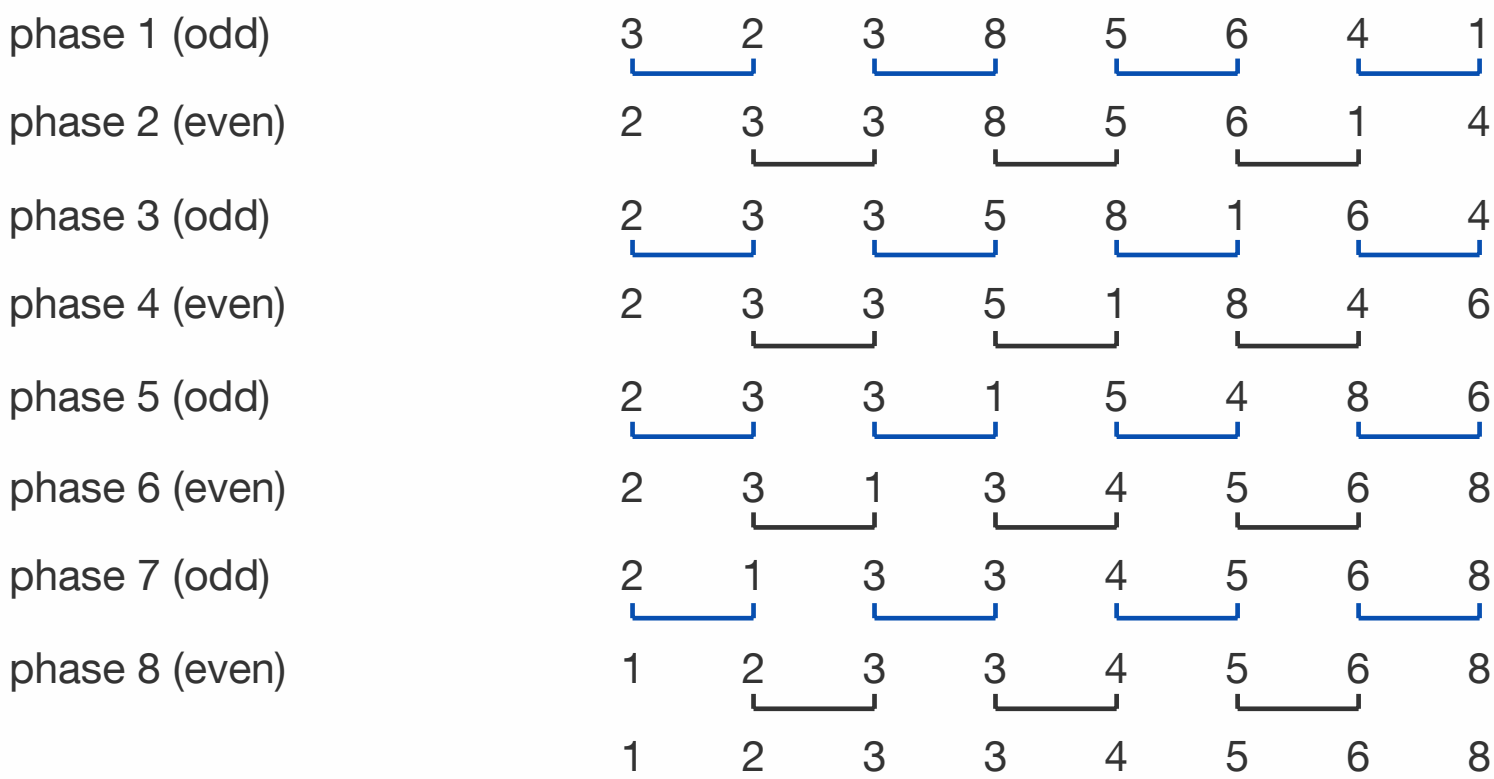
initial setup	3	2	3	8	5	6	4	1
1st iteration	2	3	3	5	6	4	1	8
2nd iteration	2	3	3	5	4	1	6	8
3rd iteration	2	3	3	4	1	5	6	8

Sorting

- bubble sort (cont'd)
  - standard algorithm not very suitable for parallelisation ➔ partition of A into blocks of size N/P elements (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, called odd and even phase
    - 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), …, (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), …, (a_{N-2}, a_{N-1})
    - after N phases of odd-even-exchanges, sequence A is sorted

Sorting

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



Sorting

- bubble sort (cont'd)
  - parallelisation of odd-even-transposition
    - each process is assigned a block of N/P elements, which are sorted internally (using merge sort or quicksort, e. g.) with a complexity of $O((N/P) \cdot \log (N/P))$
    - 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 $A_i \leq A_j$ for $i < j$)
    - during each phase $O(N/P)$ comparisons are performed, thus, the total complexity of the parallel sort can be computed as

$$\underbrace{O((N/P) \cdot \log (N/P))}_{\text{local sort}} + \underbrace{O(N)}_{\text{comparisons}} + \text{communication}$$

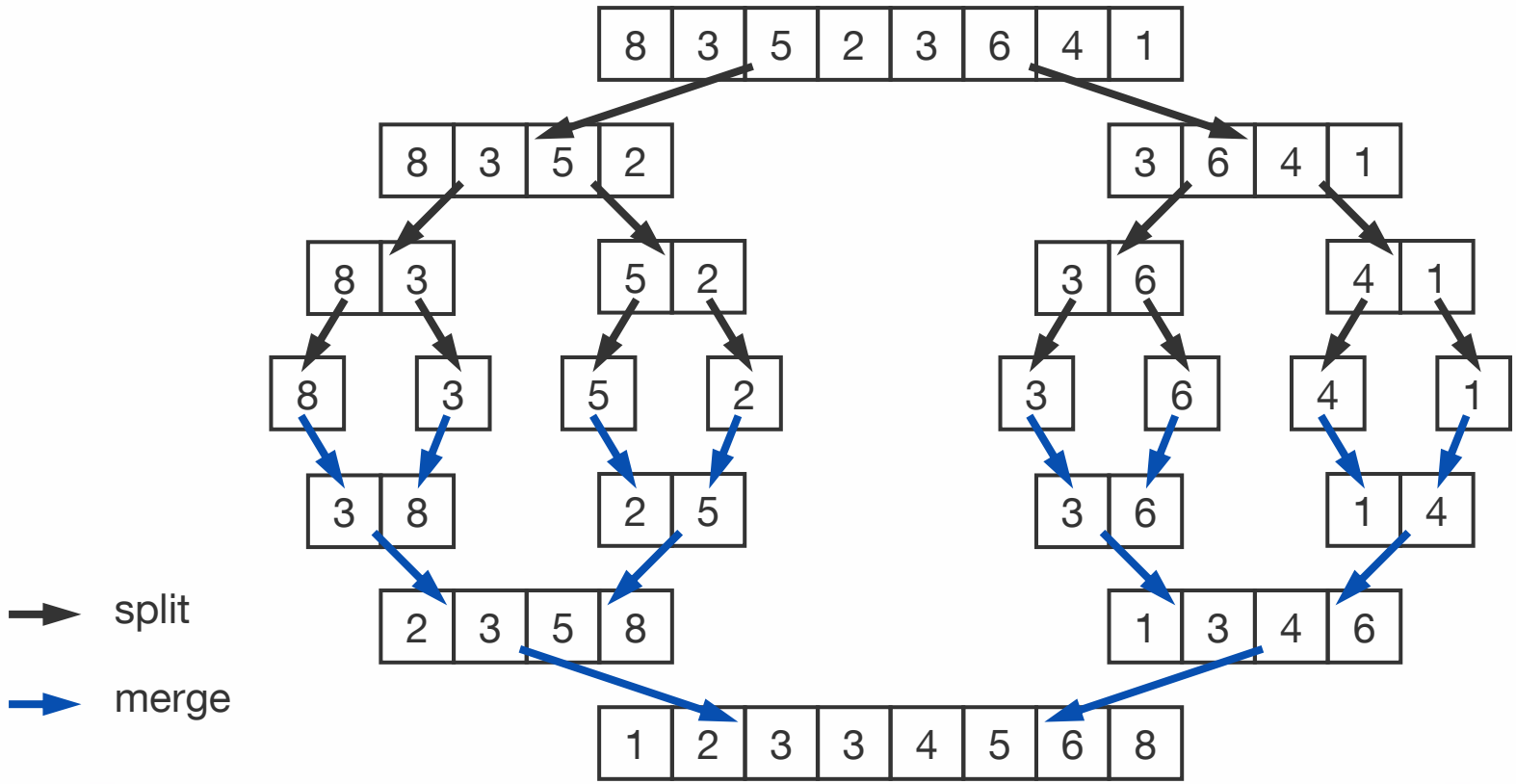
Sorting

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

```
function mergesort (list L) {
    if (size of L == 1) return L;
    else
        divide L into left and right list;
        left = mergesort (left);
        right = mergesort (right);
        result = merge (left, right);
        return result;
}
```

Sorting

- merge sort (cont'd)
  - example: merge sort for sequence A = ⟨8, 3, 5, 2, 3, 6, 4, 1⟩

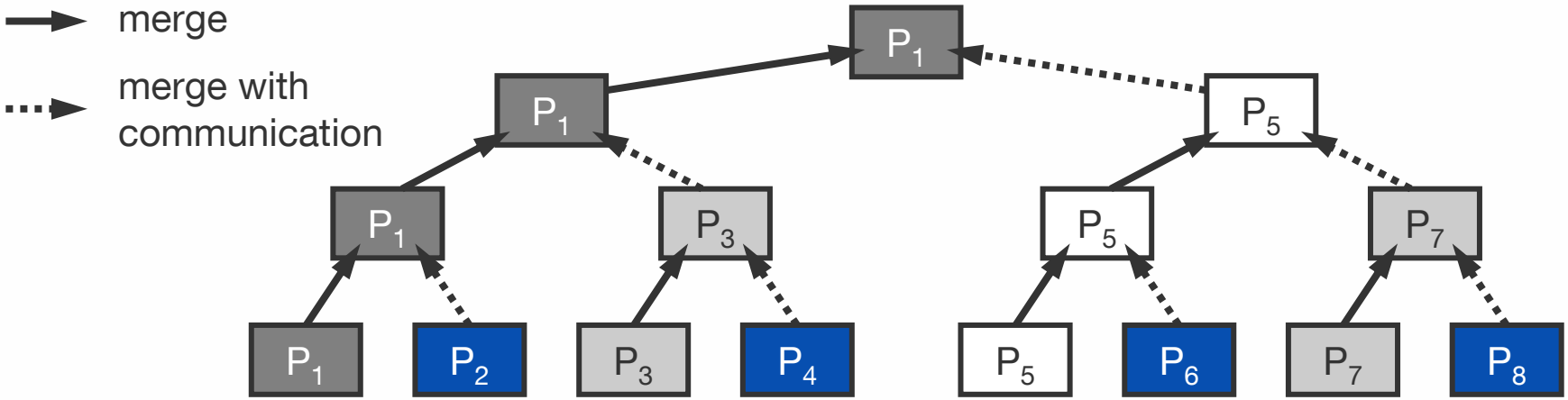


Sorting

- merge sort (cont'd)
  - parallelisation of merge sort: *naïve approach*
    - construct a binary processing tree of L leaf nodes and assign P processors, $P \geq L$, to tree nodes (i. e. inner and leaf nodes)
    - divide sequence A into blocks A_i of size N/L and store them at leaf nodes
    - parallel sort of blocks A_i via sequential merge sort with a complexity of $O((N/L) \cdot \log (N/L))$
    - repeatedly parallel merge of sorted sublists from child nodes and sending result upstream to parent node with a total complexity of $O(N)$ (costs of merge operation at the different tree levels are $N + N/2 + N/4 + \ldots + N/L$ comparisons)
  - problem: sending of sublists might induce heavy communication
  - hence, an appropriate mapping of processors to tree nodes is indispensable

Sorting

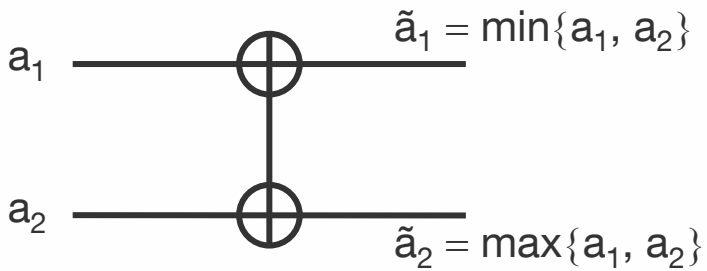
- merge sort (cont'd)
  - example: mapping of processors to tree nodes for P = L = 8

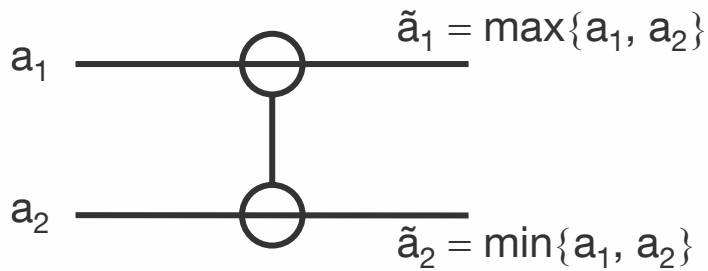


  - observations: not to expect very good speed-up values for parallel merge of sublists (amount of used processors is halved in every step ➔ part 1: parallel index and estimate of MINSKY)
  - hence, different strategy for parallel merge necessary

Sorting

- sorting networks
  - sorting networks are based on a comparison network model, that sort N elements in significantly smaller than $O(N \cdot \log N)$ 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, where each column performs a permutation, thus the final column is sorted in increasing / decreasing order (➔ permutation networks)





Sorting

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

    A sequence $S = \langle a_0, a_1, \ldots, a_{N-1} \rangle$ is bitonic iff

    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 $\langle 0, 4, 8, 9, 2, 1 \rangle$

Sorting

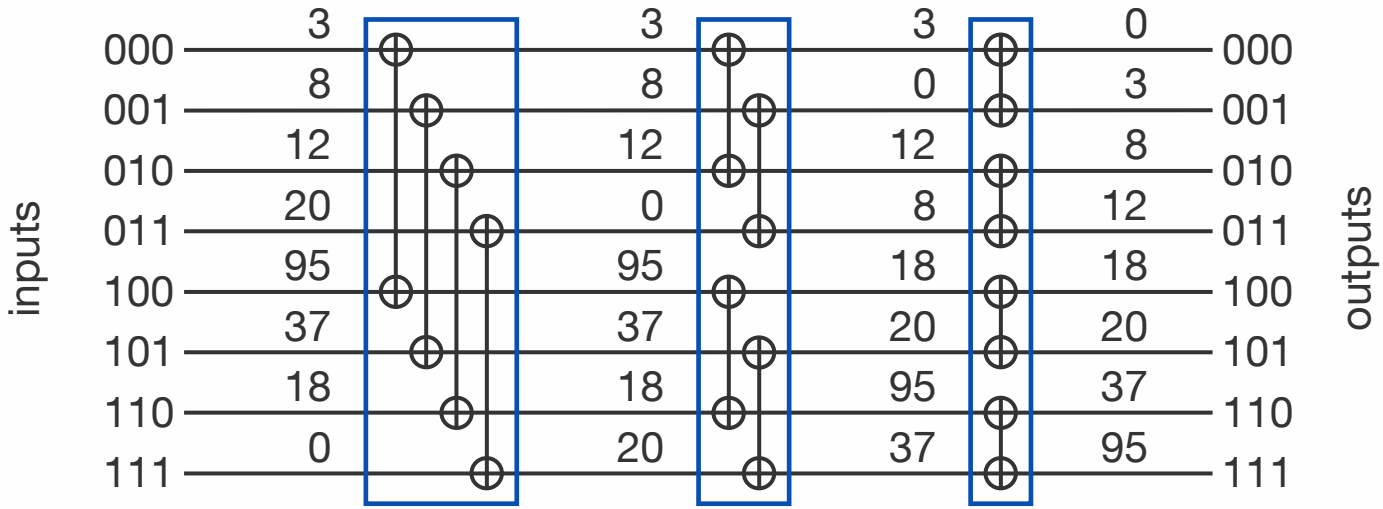
- bitonic sort (cont'd)
  - let $S = \langle a_0, a_1, \ldots, a_{N-1} \rangle$ be a bitonic sequence such that
    - $a_0 \leq a_1 \leq \ldots \leq a_{N/2-1}$ and
    - $a_{N/2} \geq a_{N/2+1} \geq \ldots \geq a_{N-1}$
  - consider the following subsequences of S
    - $S_1 = \langle \min\{a_0, a_{N/2}\}, \min\{a_1, a_{N/2+1}\}, \ldots, \min\{a_{N/2-1}, a_{N-1}\} \rangle$
    - $S_2 = \langle \max\{a_0, a_{N/2}\}, \max\{a_1, a_{N/2+1}\}, \ldots, \max\{a_{N/2-1}, a_{N-1}\} \rangle$
  - in sequence S_1, there is an element $s_i = \min\{a_i, a_{N/2+i}\}$ such that all elements before s_i are from the increasing part of S and all elements after s_i are from the decreasing part of S
  - also, in sequence S_2, there is an element $\hat{s}_i = \max\{a_i, a_{N/2+i}\}$ such that all elements before $\hat{s}_i$ are from the decreasing part of S and all elements after $\hat{s}_i$ are from the increasing part of S
  - hence, sequences S_1 and S_2 are bitonic sequences

Sorting

- bitonic sort (cont'd)
  - furthermore, $S_1 \leq S_2$ because s_i is greater than or equal to all elements of S_1, $\hat{s}_i$ is less than or equal to all elements of S_2, and $\hat{s}_i$ is greater than or equal s_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 S_1 and S_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*

Sorting

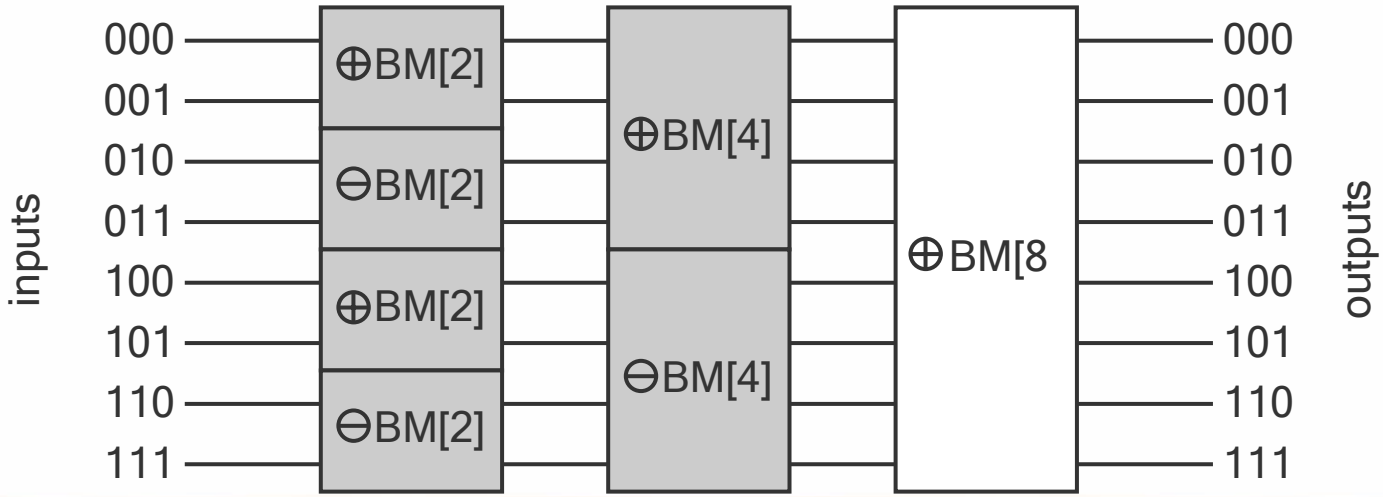
- bitonic sort (cont'd)
  - example: bitonic merging network with 8 inputs (⊕BM[8])



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

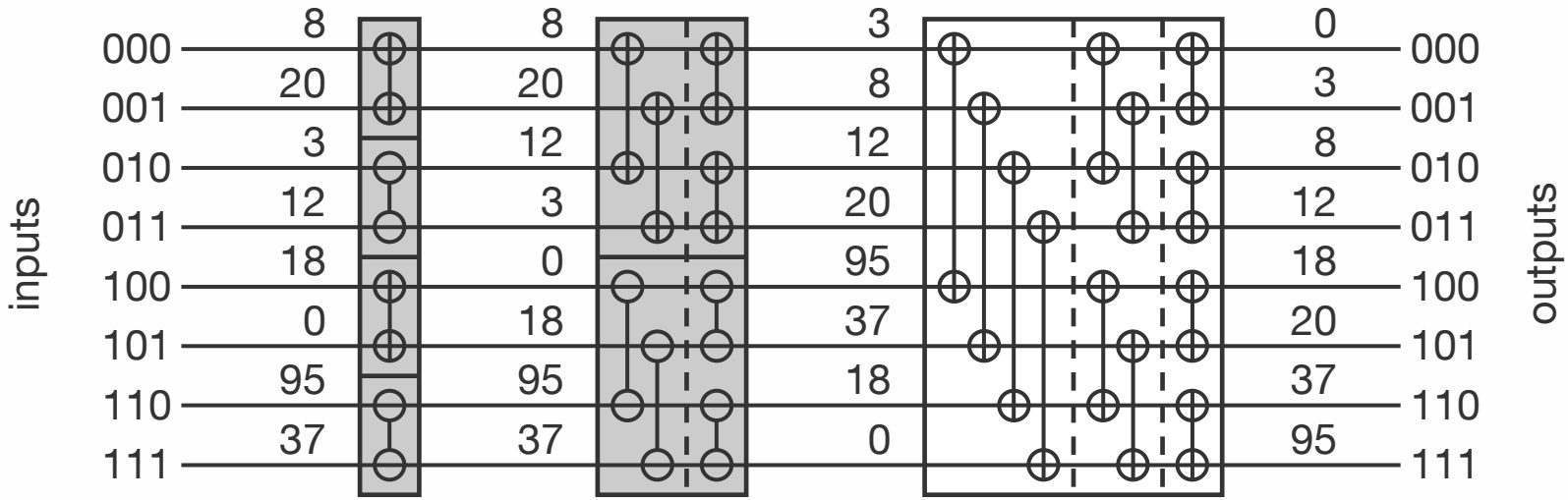
Sorting

- 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 $S = \langle a_0, a_1, \ldots, a_{N-1} \rangle$ out of N unordered elements
  - construct S by repeatedly merging bitonic sequences of increasing length (here, the last bitonic merge (⊕BM[8]) sorts the input)



Sorting

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



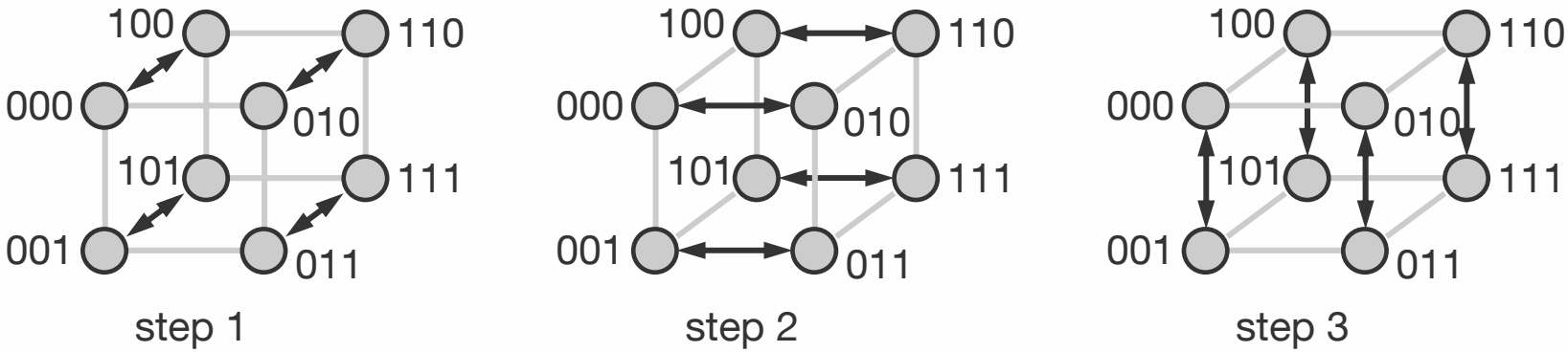
  - depth d(N) of a bitonic sorting network with N inputs can be computed by the following recursion

$$d(N) = d(N/2) + \log N$$

  - hence, $d(N) = \sum_{i=1}^{\log N} i = (\log^2 N + \log N) / 2 = O(\log^2 N)$

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 one bit, i. e. within step D processes whose (binary) labels differ in the D^{th} bit compare-exchange their elements



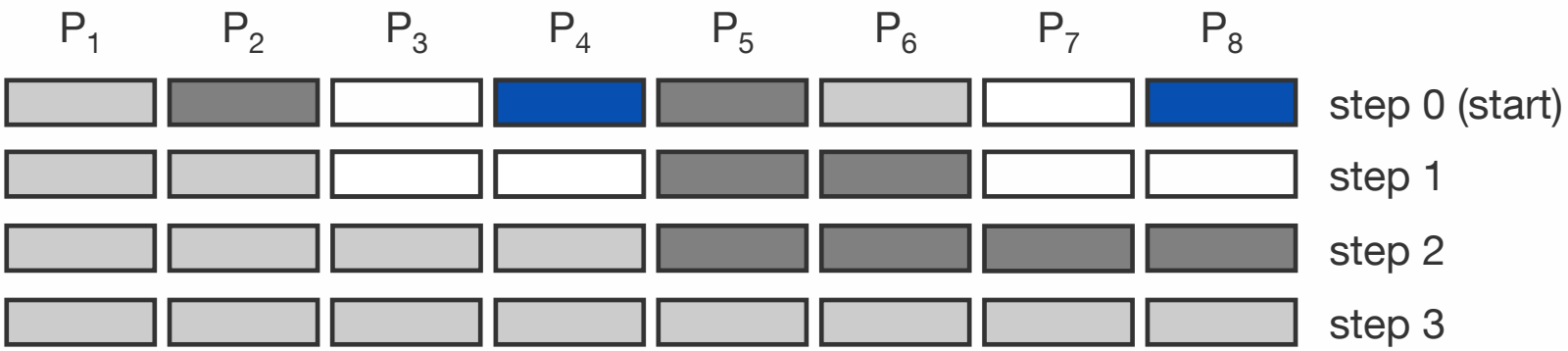
    - *mesh*: there exist several possibilities for a proper mapping

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
    - parallel sort of blocks A_i via sequential merge sort with a complexity of $O((N/P) \cdot \log (N/P))$
    - parallel merge
      1) starting point: P sorted sublists each distributed over one processor
      2) merging two sublists using compare-split operations leads to P/2 sorted sublists each distributed over two processors
      3) repeatedly executing step 2 finally leads to one sorted list distributed over P processors

Sorting

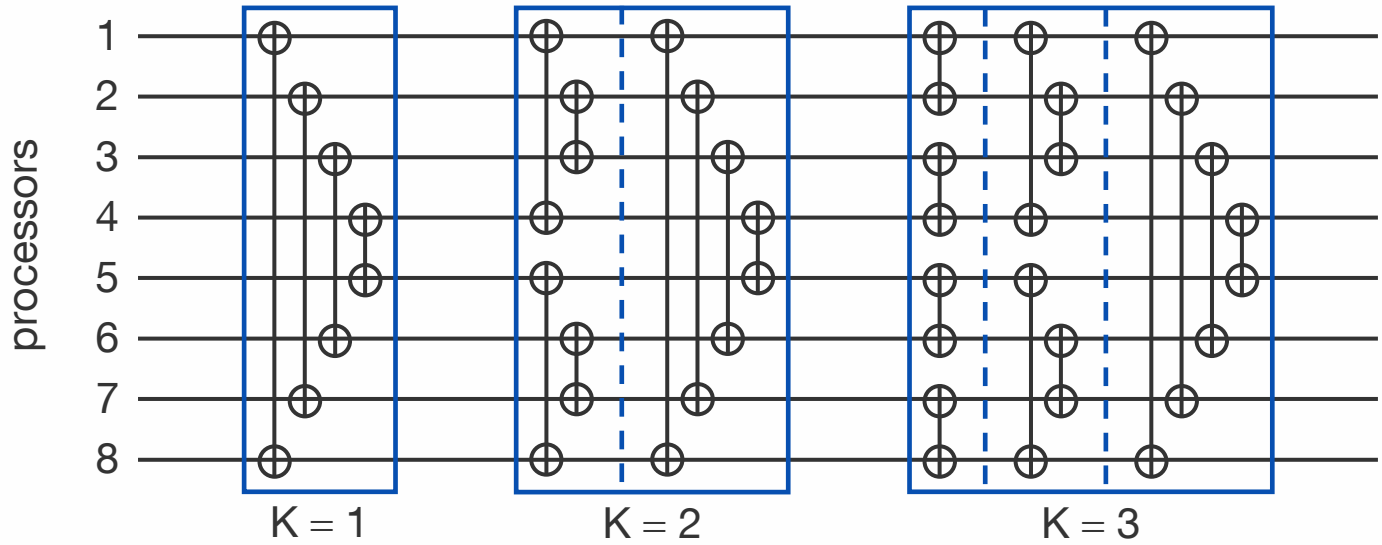
- again: merge sort (cont'd)
  - example: repeated merge of sorted sublists for $P = 8$



  - there is a total of log 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 $O(\log^2 P)$ compare-split operations

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

$$\underbrace{O((N/P)\cdot\log(N/P))}_{\text{local sort}} + \underbrace{O((N/P)\cdot\log^2 P)}_{\text{comparisons}} + \text{communication}$$