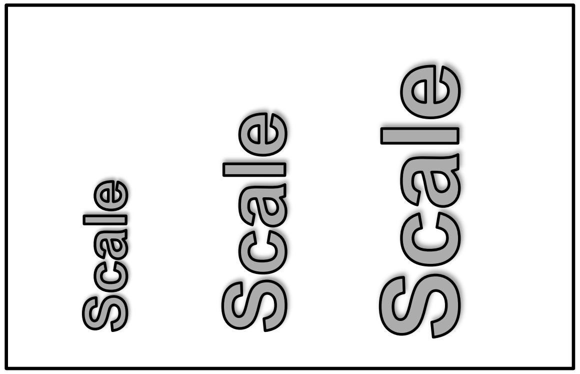
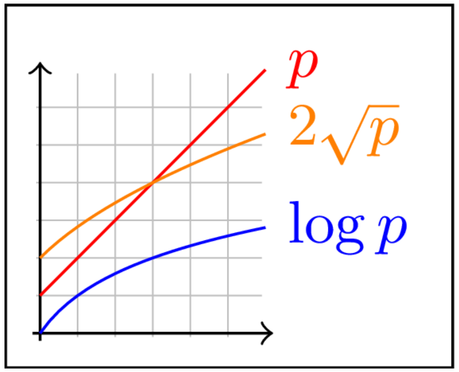
# SPP '16 – MPI lab tasks

Sergei Shudler

TECHNISCHE UNIVERSITÄT DARMSTADT





## Overview – parallel sorting problem

- Lab purpose: Implement two parallel sorting algorithms and analyze the performance of the second one (the better one)
- Input:
  - *N* integers equally distributed among *p* processes
  - Process *i* has *n* = *N* / *p* integers
- Output:
  - Permutation of the input numbers such that the local part in each process *i* is sorted
  - Each integer in process *i* is smaller or equal to all the other integers at a higher-ranked process *j* (*j* > *i*)

## General remarks

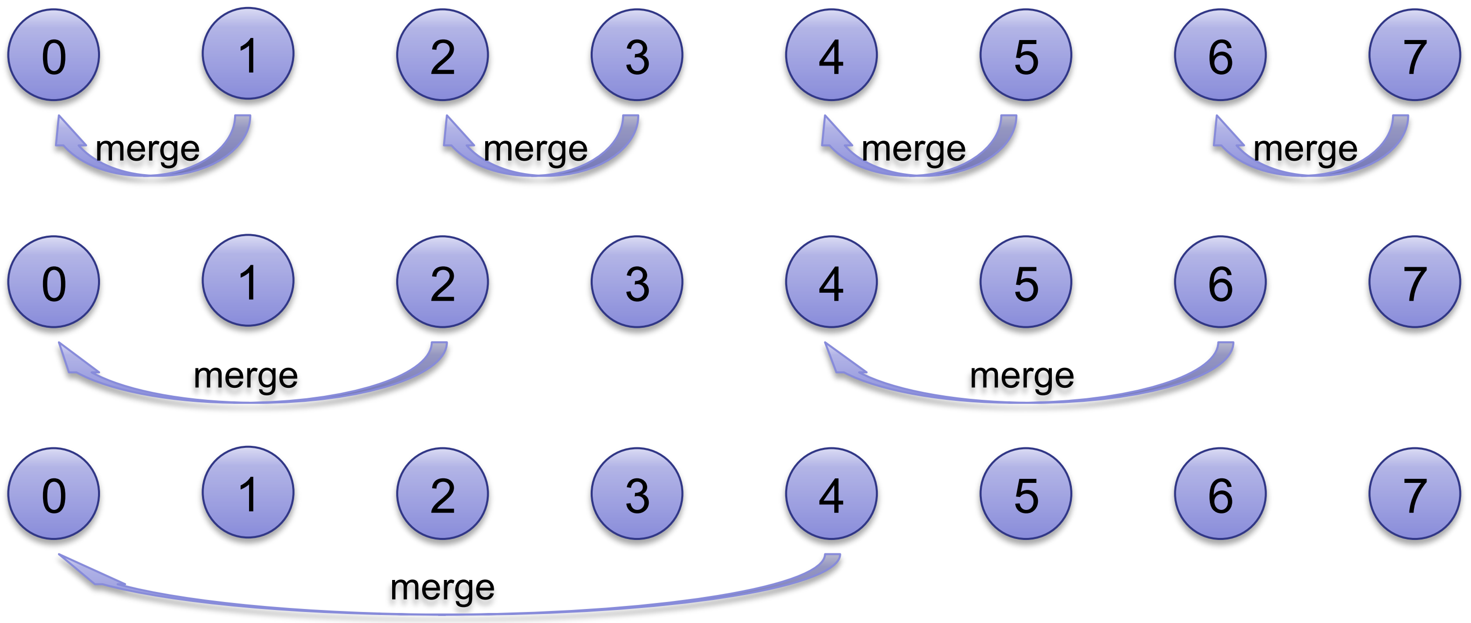
- We provide header files and source templates
  - You need to implement specific functions
- Makefile and job script is provided
- Implementation has to be in C

## Tasks

- Task 1 – Simple parallel sort
- Task 2 – Splitter-based parallel sort
- Task 3 – Complexity analysis
- Task 4 – Performance modeling

## Simple parallel sort

- In Task 1 you should implement a simple parallel sort:
  - Each process sorts its local part of the array using std. C library
  - All *p* subarrays should be merged iteratively in parallel and the root should hold the sorted array in the end, for example (*p* = 8):



## Simple parallel sort

- Results verification is provided

- Template: task1.c

## Tasks

- Task 1 – Simple parallel sort
- Task 2 – Splitter-based parallel sort
- Task 3 – Complexity analysis
- Task 4 – Performance modeling

## Splitter-based parallel sort - example

- Assume we have $p = 4$, $N = 32$, $n = 8$:

P0: 27 13 11 42 44 27 2 61

P1: 40 25 16 16 38 26 47 3

P2: 13 20 36 39 48 63 32 18

P3: 50 54 39 63 28 50 45 14

- Step 1 (sort locally):

P0: 2 11 13 27 27 42 44 61

P1: 3 16 16 25 26 38 40 47

P2: 13 18 20 32 36 39 48 63

P3: 14 28 39 45 50 50 54 63

## Splitter-based parallel sort - example

- Step 2 (splitters – red boxes):

P0: 2 11 $\boxed{13}$ 27 $\boxed{27}$ 42 $\boxed{44}$ 61

P1: 3 16 $\boxed{16}$ 25 $\boxed{26}$ 38 $\boxed{40}$ 47

P2: 13 18 $\boxed{20}$ 32 $\boxed{36}$ 39 $\boxed{48}$ 63

P3: 14 28 $\boxed{39}$ 45 $\boxed{50}$ 50 $\boxed{54}$ 63

- Step 3 (new splitters – red boxes):

13 16 20 $\boxed{26}$ 27 36 $\boxed{39}$ 40 44 $\boxed{48}$ 50 54

# Splitter-based parallel sort - example

- Step 4 (splitters = 26, 39, 48):

P0: $\boxed{2\ \ 11\ \ 13}\ \boxed{27\ \ 27}\ \boxed{42\ \ 44}\ \boxed{61}$

P1: $\boxed{3\ \ 16\ \ 16\ \ 25}\ \boxed{26\ \ 38}\ \boxed{40\ \ 47}$

P2: $\boxed{13\ \ 18\ \ 20}\ \boxed{32\ \ 36}\ \boxed{39}\ \boxed{48\ \ 63}$

P3: $\boxed{14}\ \boxed{28}\ \boxed{39\ \ 45}\ \boxed{50\ \ 50\ \ 54\ \ 63}$

## Splitter-based parallel sort - example

- After the exchange:

P0: $\boxed{2\ \ 11\ \ 13}$ $\boxed{3\ \ 16\ \ 16\ \ 25}$ $\boxed{13\ \ 18\ \ 20}$ $\boxed{14}$

P1: $\boxed{26\ \ 38}$ $\boxed{27\ \ 27}$ $\boxed{32\ \ 36}$ $\boxed{28}$

P2: $\boxed{39}$ $\boxed{42\ \ 44}$ $\boxed{40\ \ 47}$ $\boxed{39\ \ 45}$

P3: $\boxed{50\ \ 50\ \ 54\ \ 63}$ $\boxed{61}$ $\boxed{48\ \ 63}$

- Step 5 (local merge / sort):

P0: 2 3 11 13 13 14 16 16 18 20 25

P1: 26 27 27 28 32 36 38

P2: 39 39 40 42 44 45 47

P3: 48 50 50 54 61 63 63

## Splitter-based parallel sort - remarks

- Details are in the exercise description
  - Use MPI_Gather, MPI_Bcast, MPI_Alltoall, and MPI_Alltoallv
  - Other collectives are possible, but not strictly necessary
- Implement an efficient results verification *verify_results_eff* [Hint: Every process can send the biggest (local) element to the next neighbor]
- Template: task2.c

## Tasks

- Task 1 – Simple parallel sort
- Task 2 – Splitter-based parallel sort
- Task 3 – Complexity analysis
- Task 4 – Performance modeling

## Complexity analysis

- Analyze the complexity of the execution time and the memory of your implementation in Task 2
- Complexity of all the mandatory MPI collectives is provided:
  - MPI_Bcast: O(m*log p), m – size of bcasted data
  - MPI_Gather: O(log p + m), m – the size of all the gathered data
  - MPI_Alltoall/v: O(p + m), m – total amount of data sent from one process to all the others

## Complexity analysis

- Simplifying assumption:
  - Every process has *n* integers
  - *n* is much bigger than *p*, so the terms with *n* will dominate over the terms in *p*
- Should not be too fine grained, but at the level of the algorithm steps provided in Task 2

## Tasks

- Task 1 – Simple parallel sort
- Task 2 – Splitter-based parallel sort
- Task 3 – Complexity analysis
- Task 4 – Performance modeling

## Performance modeling

- Purpose: use Extra-P to create a performance model for the runtime as a function of *n*
- You are provided with the job script *perf_analysis.sh*:
  - Runs the algorithm on 5 different values of *n* (on 32 processes)
  - Writes the output to *input.res*
- You have to run the script and run Extra-P on the *input.res*
- Output is produced in an *.xtrap* file, report the model you get in this file and compare to asymptotic complexity from Task 3