Laboratory for Parallel Programming
Prof. Dr. Felix Wolf

TECHNISCHE UNIVERSITÄT DARMSTADT

Aamer Shah, shah@cs.tu-darmstadt.de

# Large-Scale Parallel Computing (WS 15/16)

# Exercise 5

This is a hands on exercise. The solution will be developed by the tutor and the students together in the class on January 19th, 2016. However, the students are encouraged to solve the problem before attending the session. Skeleton code of the tasks are provided along with the task sheet, and can also be copied from `/home/as65huly/public/ex05.tgz` on the Lichtenberg cluster.

## Task 1

Prime numbers are numbers that can be divided by 1 and by themselves only. A trivial algorithm to find if a number $N$ is a prime number or not is to try dividing it from 2 to $\frac{N}{2}$. In the task, a serial program is given, that takes a range of numbers as command line input and finds the total number of prime numbers in the range.
The aim of the exercise is to make a parallel version of the program. Perform the following subtasks:

**a)** Initialize MPI at the beginning and get process rank and communicator size.

**b)** Rank 0 distributes the number range among processes. Each process gets an equal contiguous range.

**c)** Each process finds the total number of prime numbers in its range.

**d)** Each process also measures the time it takes to find primes in its range

**e)** Apply reduction operations to find the total number of prime numbers at rank 0.

**f)** Apply reduction operations to find maximum, minimum and average time took by processes to find the prime numbers.

**g)** Find the percentage difference between maximum, minimum and average time. How do they compare?

**h)** Come up with a scheme to better distribute the number range among processes and to reduce the difference in percentage time.

## Task 2

This task uses virtual topology. In this task, a parallel, memory-efficient algorithm will be developed to sort values among processes. The scenario is that each process has a single unique value. What is required is that a sorting algorithm is applied such that after sorting, the smallest value among the processes is with rank 0, the second smallest with rank 1, and so on. As a result, the values are rearranged among the processes in ascending order. A simple naive approach can be to gather all the values at rank 0, apply a sorting algorithm and then scatter the values among the processes. However, such an algorithm does not scale and requires large amount of memory and collective communication.
We propose a ring based counting algorithm, which works as listed in Algorithm 1. **Note:** This is pseudo-code, the function syntax are not accurate.
The main idea of the algorithm is that, for a given process, count the number of values that are less than or equal to the process's own value. This count gives that rank of the process where the value should end up in sorted order. The algorithm works by first initializing each process's value. Then each process enter a loop to compare its original value to other processes' values. This is done through the ring topology, where each process gets a new value from its right neighbor, and sends the old value to its left neighbor.

**Algorithm 1** Ring-based sort algorithm

```
1:  procedure RING COUNT
2:      original_val ← drand48()
3:      send_val ← original_val
4:      //Loop from 1 to comm_size, to ignore own value
5:      for i = 1 to comm_size do
6:          //Use ring topology to shift values among the processes
7:          left_neighbor ← get_ring_left_neighbor()
8:          right_neighbor ← get_ring_right_neighbor()
9:          //Send value to left, receive value from right
10:         MPI_Sendrecv(left_neighbor, send_val, right_neighbor, recv_val)
11:         if original_val <= recv_val then count++
12:         send_val ← recv_val
13:     //Send own value to count, the rank where it should end up
14:     //Receive from any process, and store it as the sorted value
15:     MPI_Sendrecv(count, original_val, MPI_ANY, sorted_val)
```

The new value is then compared to the process's original value. At the end, the count is the rank where the process's original value should end up in sorted order.
The naive sorting algorithm is provided in the archive along with the task sheet. The aim of the task is to implement the ring-based sorting algorithm.