# Parallel Programming & Heterogeneous Computing

Summer Term 2019

*Assignment 1 (Submission deadline: 2019-05-11, 23:59 CEST)*

The first assignment covers the usage of basic synchronization primitives in a thread-based shared memory system. You have to solve the given programming exercises in C / C++ with the POSIX PThread API[1]. Additional threading libraries are not allowed for this assignment.

Furthermore, you will be introduced to our heat map example. You will re-use the same scenario in future assignments about shared memory programming, distributed memory programming and accelerator programming.

## General Rules

The assignment solutions have to be submitted at:

https://www.dcl.hpi.uni-potsdam.de/submit/

Our automated submission system is intended to give you feedback about the validity of your file upload. A submission is considered as accepted if the following rules are fulfilled:

- You did not miss the deadline.
- Your file upload can be decompressed with a zip / tar decompression tool.
- Your submitted solution contains only the source code files and a Makefile for Linux. Please leave out any Git clones, backup files or compiled executables.
- Your solution can be compiled using the “make” command, without entering a separate sub-directory after decompression.
- You program runs without expecting any kind of keyboard input or GUI interaction.
- **Our assignment-specific validation script accepts your program output / generated files.**

If something is wrong, you will be informed via email (console output, error code). Re-uploads of corrected solutions are possible until the deadline.

**50%** must be solved correctly in order to pass each assignment. Documentation should be done inside the source code.

Students can submit solutions either **alone or as team of max 2 persons**.

[1] `man pthread`

## Task 1.1

Implement a program that sums up a range of numbers in parallel. The general algorithmic problem is called “parallel reduction”.

### Input

Your executable has to be named `parsum` and accept three parameters: The *number of threads to use*, the *start index* and the *end index* (64bit numbers) of the range to compute. For example, the command line

```
Example: ./parsum 30 1 10000000000
```

has to result in a parallel summation of the numbers 1,2,...,10.000.000.000, based on 30 threads running in parallel.

### Output

Your program has to produce a single line on the standard output, that contains only the computed sum.

### Validation

The solution is considered correct if a true parallelized computation takes place (no Gauss please), if the solution scales based on the number of threads, and if the application produces correct results for all inputs. We will evaluate your solution with different thread counts / summation ranges.

## Task 1.2

Implement the dining philosophers with a freely chosen deadlock-free solution strategy[2].

Each philosopher has to be represented by a thread. You are free to map also other stake holders (e.g. waiters) or resources (e.g. forks) to threads if necessary.

### Input

Your executable has to be named `dinner` and accept two parameters, the *number of philosophers* resp. forks (min. 3) and the *maximum run time* in seconds.

```
Example: ./dinner 3 10
```

### Output

After the given run time is exceeded, the program must terminate with exit code 0 and produce an output file with the name of `output.txt` in the same directory. This file has to contain nothing but the number of “feedings” per philosopher as semicolon separated values.

```
Example: 5000;5000;5000
```

[2] https://en.wikipedia.org/wiki/Dining_philosophers_problem

## Task 1.2

Implement a program that simulates heat distribution on a two-dimensional field. The simulation is executed in rounds. The field is divided into equal-sized blocks. Initially some of the blocks are cold (value=0), some other blocks are active hot spots (value=1). The heat from the hot spots then transfers to the neighbor blocks in each of the rounds, which changes their temperature value.

The new value for each block per round is computed by getting the values of the eight direct neighbor blocks from the last round. The new block value is the average of these values and the own block value from the last round. Blocks on the edges of the field have neighbor blocks outside of the fields, which should be considered to have the value 0. When all block values are computed in a round, the value of the hot spot fields may be set to 1 again, depending on the live time of the hot spot during a given number of rounds.

You have to develop a parallel application for this simulation in C or C++, which only uses the POSIX. Additional threading libraries, such as OpenMP, or the C++ 11 concurrency features are not allowed. The goal is to minimize the execution time of the complete simulation. Specific optimizations for the given test machine (such as a fixed number of pinned threads) are not allowed and will lead to a fail grade in the assignment.

### Input

Your executable has to be named `heatmap` and needs to accept five parameters:

- The width of the field in number of blocks.
- The height of the field in number of blocks.
- The number of rounds to be simulated.
- The name of a file (in the same directory) describing the hotspots.
- *Optional parameter*: The name of a file (in the same directory) containing coordinates. If it is passed, only the values at the indicated coordinates (starting at (0, 0) in the upper left corner) are to be written to the output file.

```
Example:      ./heatmap 20 7 17 hotspots.csv

              ./heatmap 20 7 17 hotspots.csv coords.csv
```

The *hotspots* file has the following structure:

- The first line can be ignored.
- All following lines describe one hotspot per line. The first two values indicate the position in the heat field (x, y). The hot spot is active from a start round (inclusive), which is indicated by the third value, to an end round (exclusive!), that is indicated by the last value of the line.

```
Example hotspots.csv:

x,y,startround,endround
5,2,0,20
15,5,5,15
```

```
Example coords.csv:

x,y
5,2
10,5
```

## Output

The program must terminate with exit code 0 and has to produce an output file with the name `output.txt` in the same directory.

If your program was called without a coordinate file, then this file represents the resulting field after the simulation terminated. Each value in the field is encoded the following way:

- A block with a value larger than 0.9 has to be represented as "*X*".
- All other values are incremented by 0.09. From the resulting value, the first digit after the decimal point is added to the output picture.

```
Example content of output.txt without coordinate file

11112221111111111100
11123432111111111110
11124X42211111111111
11124442111111222111
11122222111112222211
11111211111112232211
01111111111111222111
```

If your program was called with a coordinate file, then this file simply contains a list of exact values requested through the coordinate file.

```
Example content of output.txt with coordinate file

1.0
0.030563410733335933
```