Project Seminar:
Parallel and Distributed Systems

Assignment 1 (Submission deadline: Nov 02th 2015, 23:59 CET)

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 2.6 64-bit. Please leave out any Git / Mercurial repository clones or SVN / CVS meta-information.
- 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 with your submission, you will be informed via email (console output, error code). Re-uploads of corrected solutions are possible until the deadline.

All tasks must be submitted accordingly in order to pass the assignment.

Assignment 1

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; they will be covered in later assignments.

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 application 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
```

[1] `man pthread`

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 an output file with the name "output.txt" in the same directory. This file has to contain 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.

Performance results will be announced in the lecture.

Task 1.2: pargrep

Develop an C/C++-based command line tool that searches a file for given strings. The program gets two files as command-line arguments. The first file contains the list of search strings, the second file contains the data to be analyzed.

The first step is to read both files completely into memory (yes, normally you wouldn't do that). After that, the program spawns a number of threads. These threads count the number of occurrences of the given strings in the text.

Input

Your program has to be named as "pargrep" and take two arguments, a search string and a file path:

```
Example: ./pargrep /tmp/strings.txt /tmp/input.txt
```

Output

The program must terminate with exit code 0 and produce an output file with the name "output.txt" in the same directory. This file has to contain nothing but the *search strings* and number of *their occurrences* in the input document as semicolon separated values:

```
abc;3
def;10
```

Task 1.3: Heat Map with Threads

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/C++. The goal is to minimize the execution time of the complete simulation.

Input

Your application 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.*
- The name of a file (in the same directory) containing *coordinates*. This is the only optional parameter. 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 content of hotspots.csv:
x,y,startround,endround
5,2,0,20
15,5,5,15
```

With such an input file, you have to run a simulation of 17 rounds on a 20x7 field with two hotspots. The first one will be located at the coordinates (5, 2) and will be active from the first round to last round (and beyond). The second hotspot will be located at the coordinates (15, 5) and will be active starting from round 5. Starting from round 15, it will no longer be active. The starting round is inclusive, the final round is exclusive. We start counting at 0. So the first hotspot will be active at round 0,1,2…18,19; the second hotspot will be active at round 5,6,7,…13,14.

```
Example content of coords.csv:
x,y
5,2
10,5
```

With such a coordinate file, you only have to provide the values at the coordinates (5, 2) and (10, 5) as part of the output file.

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 simulation termination. The values in the field are encoded in the following way:

- A block with a value larger than 0.9 has to be represented as "*X*".
- All other values must be increased 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 represents the list of exact values requested through the coordinate file.

```
Example content of "output.txt" with coordinate file
1.0
0.030563410733335933
```

Performance results will be announced in the lecture.