Parallel Programming Concepts WS 2013 / 2014

Assignment 3 (Submission deadline: Dec 8th 2013, 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, 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 the assignment. Documentation should be done inside the source code.

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

Assignment 3

The third assignment covers OpenMP in shared memory systems. OpenMP-enabled compilers should be available in all modern development systems, such as with the default compiler under Linux and MacOS X (gcc –fopenmp)

Two out of three tasks must be solved correctly in order to pass the assignment. Documentation should be done inside the source code.

Task 3.1: Heat Map with OpenMP

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.

A round is computed as follows:

1. 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.
2. 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.

You have to develop a parallel application for this simulation in C / C++ or Fortran, which only uses OpenMP. 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, since we may have to opportunity to run your code on some larger computer cluster for the performance comparison.

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
11124444211111222111
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
0.030563410733359 33
```

The student achieving the lowest average runtime will be announced in the lecture.

Task 3.2: Decrypt with OpenMP

Develop an OpenMP-based command line tool that performs a brute-force dictionary attack on Unix crypt(3) passwords. An example password file to be attacked is available at:

http://www.dcl.hpi.uni-potsdam.de/teaching/parProg/taskCryptPw.txt

Each line of the password file contains the username and the encrypted password, separated by the character ":". Your program can use the example dictionary file available at:

http://www.dcl.hpi.uni-potsdam.de/teaching/parProg/taskCryptDict.txt

One of the users has a password exactly matching one dictionary entry. A second user has a password build from one of the dictionary entries plus a single number digit (0-9) attached, e.g. "Abakus5".

It is recommended to start with a serial version of your program, and add the OpenMP parallelization as the last step.

Please note that the first two characters of the encrypted password string in *taskCryptpw.txt* are the salt string used in the original encryption process. A correct solution therefore splits the encrypted password string into salt and encryption payload, calls some *crypt(3)* implementation with the salt and all of the dictionary entries, and checks if one of the crypt results matches with an entry from the user list.

Input

Your program has to be named "decrypt" and has to take two arguments, the name of the *password file* as the first and the name of the *dictionary file* as the second command line argument.

```
Example: ./decrypt ../../taskCryptPw.txt ./../taskCryptDict.txt
```

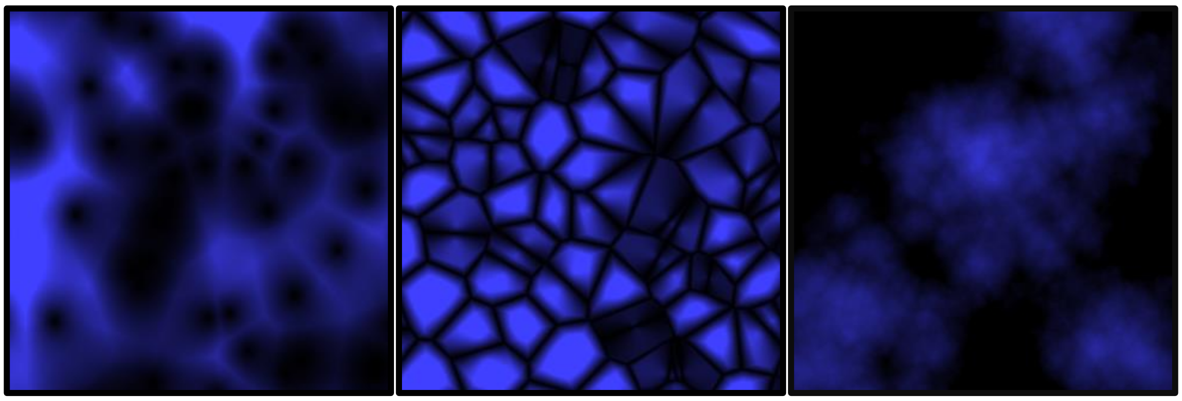
Output

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 users whose passwords could be decrypted with the dictionary. Each line of the result file has to be a combination of username and decrypted password, separated by semicolon:

```
User01;pass
User02;Abakus5
```

Submit a compressed archive with the OpenMP sources as a solution. Beside the source code, the archive can also contain a file named "taskCryptSolution.txt" with the cracked users for the example data. In this case the validation step will tell you if you found the right ones. Please do not let the validation machine perform the cracking of the example data, since this may take several hours.

Task 3.3: Worley Noise with OpenMP



Besides Perlin Noise and Simplex Noise, Worley Noise[1] aka “Cell Noise” is one of the fundamental ways to create beautiful realistic textures for games and simulations.

Your task is to use OpenMP to parallelize this Open Source Cell Noise implementation:

https://code.google.com/p/fractalterraingeneration/downloads/detail?name=CellNoise.1.0.cpp

Develop an OpenMP-based command line tool that performs a Worley Noise calculation and saves the resulting picture to a Bitmap-File. (The code that can be found at the Url already produces such a file. All you have to do is to parallelize it using OpenMP and use the parameters that we are passing to your program.)

Input

Your program has to be named “worley” and has to take two arguments, the *width* and the *height* of the image that is produced.

```
Example: ./worley 1000 1000
```

Output

The program must terminate with exit code 0 and produce an output file with the name “output.bmp” in the same directory.

Further Remarks

In order to allow for larger picture sizes, you need to put the map on the heap instead of the stack:

```
float* map = (float*)malloc(sizeof(float)*hgrid*vgrid);
…
void fillMap(float* map, float &min, float &max)
…
map[x + y * hgrid] = total;
map[j + i * hgrid] -= min;
…
```

[1] If you want to toy around with a noise generator, check out: http://aftbit.com/cell-noise-2/