Parallel Programming & Heterogeneous Computing

Summer Term 2019

Assignment 2 (Submission deadline: 2019-06-01, 23:59 CEST)

The second assignment covers OpenMP in shared memory systems. OpenMP-enabled compilers should be available in all modern development systems, such as with GCC under recent Linux distributions (–fopenmp). On macOS, you should consider installing GCC manually via e.g. Homebrew..

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**.

Task 2.1: 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.

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++ 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 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

1111222111111111100
1112343211111111110
11124X4221111111111
1112444211111222111
1112222111112222211
1111121111112232211
0111111111111222111
```

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

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

1.0000
0.0306
```

Task 2.2: 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:

https://www.dcl.hpi.uni-potsdam.de/teaching/parProg/assignments/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:

https://www.dcl.hpi.uni-potsdam.de/teaching/parProg/assignments/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 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 executable "decrypt" 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 print a line by line a list of the successfully cracked combinations of username and decrypted password, both separated by semicolon:

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

(This is not the solution). Submit a compressed archive with the OpenMP sources. Additionally, the archive can also contain a file named "taskCryptSolution.txt" with the cracked users for the above example data (include a trailing new-line). In this case the validation step will tell you if you found the right ones. The validation machine will not crack the example data, since this may take several hours.

Task 2.3: Hashed & Ordered Index with OpenMP

Develop an OpenMP-based program, that generates a range of 512 bit blocks, applies the MD5-hash to them and returns the given *i* numerically smallest hash(es).

The first block is generated from a hexadecimal noted byte sequence supplied as the first command line argument. Missing bytes at the block's end are padded with zero. Each following blocks of the block range is incremented by one, acting like a 512-bit big-endian integer.

Your program then applies the MD5-hash to all blocks. Afterwards, the user can query for the *i* smallest hash(es) in the resulting list of hashes.

Input

Your executable "hoi" takes three or more arguments: The initial block represented as hexadecimal number, the number of blocks and (multiple) queried indices.

```
Example: ./hoi c0ffee 4 0 2
```

In this example the following blocks and hashes are generated:

```
md5(c0ffee00[0...]00) = ec100b976988b81e9465a680afaadecd
md5(c0ffee00[0...]01) = 210542482275b82b18d71deeca954acf
md5(c0ffee00[0...]02) = f323610f90a5bf62e51e0f3e44726b9f
md5(c0ffee00[0...]03) = 1a640827a901ae4c617f6893c16d0398
```

Output

The program must terminate with 0 and print line by line the requested smallest hashes. The above example would print the bytes of the smallest and third-smallest of the four hashes (index 0 and 2) in hexadecimal representation:

```
1a640827a901ae4c617f6893c16d0398
ec100b976988b81e9465a680afaadecd
```

Task 2.4 [optional]: Parallel Grep with Java Monitors

Develop a Java-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. The result is then written to a shared data structure, the output list.

Use the Monitor concept and condition variables in order to realize this solution. The idea is that there is a central class with a critical method that looks like this, executed by each thread:

```
void lookforit() {
        // get string to search for
        // look for the string in buffer
        // write string to result list
}
```

Input

Your program has to be named “pargrepmon” and has to take two arguments, a search *string file path* and a *input data file path*:

```
java -jar pargrepmon.jar /tmp/strings.txt /tmp/input.txt
```

The search strings file contains one search string per line.

Output

The program must terminate with exit code 0 and print line by line the search strings and number of their occurrences in the input document as semicolon separated values:

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