Systems Software

VISUALIZATIONANDMULTIMEDIALAB

BINF31aa (BINF3101)

Prof. Dr. Renato Pajarola

Exercise Completion Requirements

- Exercises are mandatory, at least 3 of them must be completed successfully to finish the class and take part in the final exam. The first exercise serves as an introduction to C/C++ and does not count towards admission to the final exam.
- Exercises are graded coarsely into only two categories: ***fail*** or ***pass***
    - if all the exercises (without considering the first introductory exercise) are completed successfully, then a bonus is carried over to the final exam[1]
- Turned in solutions will be scored based on functionality and readability
    - a solution which does not compile, run or produce a result will be a ***fail***
    - the amount of time spent on a solution cannot be taken into account for the score
- Exercises can be made and submitted in pairs
    - both partners must fully understand and be able to explain their solution
- Students may randomly be selected to explain their solution
    - details to be determined during exercises by the assistant

Exercise Submission Rules

- Submitted code must compile without errors.
- Code must compile and link either on Mac OS X or on a Unix/Linux/Cygwin environment using a Makefile
- The whole project source code (including Makefile) must be submitted via OLAT by the given deadline. Include exactly the files as indicated in the exercise.
- We will use MOSS to detect code copying or other cheating attempts, so write your own code!
- The whole project (including Makefile) must be zipped, and the .zip archive has to be named: ss_ex_EXERCISENUMBER_MATRIKELNUMBER.zip (ss_ex_3_01234567.zip - one student, Systems Software exercise number 3; ss_ex_1_01234567_02345678.zip - two students, Systems Software exercise number 1).
- Submit your code through OLAT course page.
- Teaching Assistant: Claudio Mura (claudio@ifi.uzh.ch)

Help with C++, Makefiles and UNIX programming

C++ Tutorial:	http://www.cplusplus.com/doc/tutorial/
C++ Library Reference:	http://www.cplusplus.com/reference/
C++ STL:	http://www.sgi.com/tech/stl/table_of_contents.html
GNU Make Tutorial:	http://www.gnu.org/software/make/manual/make.html
Reference book:	"Advanced Programming in the UNIX® Environment, 2nd edition" *W. R. Stevens, S.A. Rago,* Addison Wesley

[1] the bonus will be in the order of 10% of the total number of points achievable in the final exam

NOTE: for this exercise it is assumed that you have acquired sufficient familiarity with the C++ language and with Makefiles. Before you start writing the solution make sure you have fully understood the relevant theory chapters from the Operating Systems book. You might find it useful to check the man pages of `pthread` and to read the resources listed above.

Exercise 3: Parallel Image Denoising using PThreads

In this exercise you will implement a multi-threaded version of an image denoising algorithm based on *median filtering*. To do so, you will make use of the thread management routines provided by the PThreads standard.

A grayscale image can be represented as a matrix of *intensity* values in the range **[0,1]**, each of them corresponding to the shade of grey of a *pixel* in the image. During the image acquisition process, a value is set for every pixel of the image. However, due to defects in the sensor of the camera, some of the pixels may acquire wrong values. After acquiring an image, it is desirable to automatically modify the image so that the wrong values are removed. Since it is not know which pixels values are correct and which are not, one common approach is to replace the value of *every* pixel with a value that is likely to be correct. One way of doing this is to replace the value of every pixel with the median of the values of its surrounding pixels. This algorithm is called *median filtering*.

To better describe the algorithm, let's introduce some notation. Let **p(r,c)** denote the value of a pixel at position **(r,c)** in the image, that is, the pixel at row **r** and column **c** of the image matrix. The pixels in a *window* of size **s** around **(r,c)** are identified by the set of indices $\mathbf{W_{r,c} = \{ (r + i, c + j) \}}$, with **i** and **j** taking values in the range **[floor(-s/2), floor(s/2)]**. For the sake of practicality, **s** is often restricted to be an odd number.
Performing the median filtering on a pixel **(r,c)** then boils down to computing the median $\mathbf{v_{med}}$ of the pixels whose positions are contained in $\mathbf{W_{r,c}}$ and replacing the value **p(r,c)** with $\mathbf{v_{med}}$. In order to de-noise a whole image, one simply repeats the operation described above for all the pixels.

Your task is to write a C++ application that, given as command-line arguments an integer **s** (corresponding to a window size) and the name of a text file containing an image matrix, will perform the median filtering of the image stored in the file using a window of size **s**. Your application will start by reading the image from the file into memory. The text file will be formatted as follows:

```
n_rows
n_cols
M[ 0, 0 ] M[ 0, 1 ] ... M[ 0, n_cols-1 ]
M[ 1, 0 ] M[ 1, 1 ] ... M[ 1, n_cols-1 ]
......
M[ n_rows-1, 0 ] M[ n_rows-1, 1 ] ... M[ n_rows-1, n_cols-1 ]
```

Note that `n_rows` is the *height* of the image and `n_cols` is its *width*.

After reading the input matrix, your application will start with the actual filtering of the image. You must provide both a serial (i.e., non-parallel and single-threaded) and - more importantly - a parallel version of the filtering algorithm, in which *n* threads (with *n* passed as additional command-line parameter) process in parallel different parts of the image. For the parallel version you must devise a partitioning strategy for the problem and assign each thread a range of indices corresponding to the pixels it must process. In doing so, you must make sure that the workload is as much as possible balanced among the available threads.
Also note that, provided that you use a separate matrix to store the results of the processing, the parallelisation will be free of read-write conflicts, that is, it will never occur that a thread tries to read an item that could be modified by another thread.

After computing the filtered matrix, you must write it to a text file named `filtered.txt`, which should be formatted like the input file.

Some additional comments about this task:

- your application should run either the serial or the parallel version of the algorithm based on the value of a command-line argument (value = 0 -> serial, value = 1 -> parallel);
- overall, your application will take 4 command-line arguments (in this order): a string corresponding to the full path of the file containing the input image; an integer corresponding to the window size; an integer corresponding to the number of threads to be created; an integer that controls whether to run the serial or the parallel version (see previous bullet point);
- make sure to avoid accessing locations that are outside the matrix! This could happen when you are processing pixels on the boundaries (i.e., pixels **p(r,c)** such that **r < s/2** or **r >= n_rows - s/2** or **c < s/2** or **c >= n_cols - s/2**. For these pixels, simply consider the locations of $\mathbf{W_{r,c}}$ that fall inside the image.

Further clarification on this exercise will be provided during the tutorial session of October, 12th.

Notes:

- you are provided with some sample sources files and with a simple Makefile; use them as a basis for your solution.

Deliverables:

Electronically submit your project files (i.e. source files and Makefile) and a README file that briefly explains your program, all in a single ZIP file.

Deadline: 25th October, 2015 at 23.59h.