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
OpenMP Tutorial:	https://computing.llnl.gov/tutorials/openMP/
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

Exercise 4: Parallel Median Filtering of Image Sets using OpenMP

The goal of this exercise is to develop an application to denoise a set of grayscale images using *median filtering*. The application should be able to work work both serially and in parallel; to achieve the parallelism, you should make use of the directives and routines provided by the OpenMP API.

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 known 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 amounts 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 its pixels.

Your task is to write a C++ application that, given as command-line arguments an integer **s** (corresponding to a window size) and a list of names of text files containing image matrices, will perform the median filtering of the images using a window of size **s**. Your application will start by reading the images from the files into memory. Each 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 matrices, the actual filtering of the images will start. Your application must be able to run in one of the following possible modes:

1. **serial**: there will be a single thread of execution; the images will be processed one after another and, for each image, its pixels will be processed sequentially;
2. **parallel, at image level**: there will be multiple threads of execution, running in parallel; each thread will process one or more images, so that the set of input images is split among the available threads; each thread will process the images it has been assigned one after another, filtering the pixels sequentially;
3. **parallel, at pixel level:** the images will be processed sequentially one after another, but for each image its pixels will be processed in parallel by multiple threads of execution, splitting the set of pixels to be processed among the available threads;
4. **benchmark**: the processing will be performed in all 3 modes listed above, one after another; your application will keep track of the time spent in each mode (using the functions in the

OpenMP runtime library) and print a summary of the processing time when all 3 modes have finished.

The number of threads to be created must be passed to your application as a command-line argument. For modes 2. and 3. you should devise a parallelisation strategy that ensures that the workload is balanced among the available threads.

Note that you should pre-allocate separate image matrices to store the results of the processing. This way, 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.

If the application is running in one of the modes 1. to 3., each filtered image matrix must be output to a text file, formatted like the input files (see description above). The name of each output file will be the name of the corresponding input file preceded by the string `OUT_` . For example, the filtered version of the matrix contained in the input file `image1.txt` must be named `OUT_image1.txt` .

Some additional comments about this task:

- the running mode of your application should be selected based on the value of a command-line argument (value = 0 -> serial, value = 1 -> parallel at image level, value = 2 -> parallel at pixel level, value = 3 -> benchmark);
- the overall sequence of command-line arguments must be the following (in this order): an integer corresponding to the window size; an integer corresponding to the number of threads to be created; an integer that controls which run mode to execute; a list of strings corresponding to the filenames of the input image matrices; for instance, invoking the executable of your application with the following list of parameters

  3 8 2 image1.txt image2.txt image3.txt

  will result in the filtering of the 3 images stored in files image1.txt, image2.txt, image3.txt using a window of size 3 and parallelised at an image level using 8 threads;
- when filtering an image, 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, 26th.

Notes:

- you are provided with some sample source files and with a simple Makefile; you may 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: 8th November, 2015 at 23.59h.

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 and the Parallel Computing books.