CSC2002S Assignment PCP1 2022

set by M. Kuttel

Parallel Programming with the Java Fork/Join framework:

2D Median Filter for Image Smoothing

The aim of this assignment is to give you experience in programming a parallel algorithm using the Java Fork-Join library as well as benchmarking the parallel program to determine under which conditions parallelization is worth the extra effort involved.

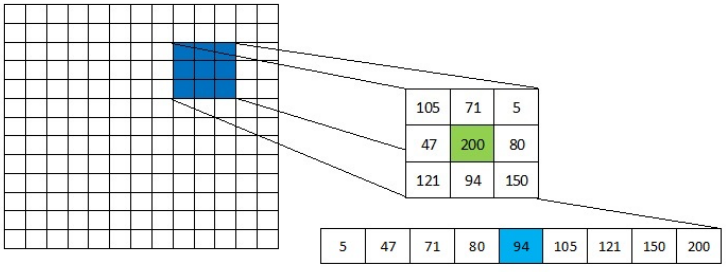
Specifically, your task is to implement two parallel **filters** for smoothing **RGB** colour images. The pictures below show an original image on the left, the same image smoothed with a *mean filter* (middle) and with a **median filter** (right). A *mean filter* sets each pixel in the image to the *average* of the surrounding pixels, whereas a **median filter** sets each pixel to the **median** of the surrounding pixels. A median filter has the advantage that it is much better at preserving edges after smoothing (compare the skylines in the three images).


original


mean filter


median filter

Both methods use a **sliding square window** of a specified width *w* (*w* is an odd number >=3) that defines the neighbouring pixels that are used to calculate the mean or the median. The mean filter simply averages all the pixels within the window. However, to calculate the median, the pixels are sorted into numerical order, and then the target pixel is replaced with the middle (median) value of the window. The figure below illustrates this.



A 3x3 **median filter**: the pixels in the window are sorted and then the middle/median pixel replaces the target pixel.

This is actually a bit more complicated for RGB colour images, as follows. The filter calculation considers the three colour components of each pixel (red, green, and blue)

separately – and each pixel is set to the mean/median of the red, green, and blue values of the surrounding pixels.

Although it has clear advantages, an median filter is much more **computationally expensive** than a mean filter, because of the sorts required. This relative computational cost increases with the size of the window. Therefore this is a nice example for illustrating when parallelisation is worth the effort

In this assignment, you will do the following.

1. Write two **serial programs** to apply a mean and median filters to a specified image.
2. Write divide-and-conquer **parallel programs** using the Java Fork/Join framework in order to speed up the median and the mean filter process.
3. **Benchmark** your programs experimentally, **timing** the **execution** on at least two machines (e.g. your laptop and the departmental server) with **different image sizes** and **different filter sizes** (e.g 3x3,5x5,11x11,15x15), generating **speedup graphs** that show when you get the best parallel **speedup**. (Note that you should also experiment to determine the point at sequential processing should begin in your fork/join algorithm).
4. Write a **short report** including the graphs with an explanation of your findings.

Note that parallel programs need to be both correct and faster than the serial versions. Therefore, you need to demonstrate both correctness and speedup for your assignment. If speedup is not achieved, you need to explain why.

Assignment details and requirements

1) Program files

You must submit four Java program files, named `MeanFilterSerial.java`, `MeanFilterParallel.java`, `MedianFilterSerial.java` and `MedianFilterParallel.java`, respectively. Use these **exact names**.

2) Input and Output

Your programs must take the following command-line parameters (in order):

```
<inputImageName> <outputImageName> <windowWidth>
```

where

`<inputImageName>` is the name of the input image, including the file extension;
`<outputImageName>` is the name of the input image, including the file extension;
`<windowWidth>` is the width of the square filter window. This must be an odd, positive integer >=3.

- Invalid input should be handled gracefully, exiting without crashing.
- You can assume that all files are in jpeg format, although the **`java.awt.image.BufferedImage`** class makes it easy to deal with a range of image file types.

3) Benchmarking

- The first step in optimising your fork/join parallel code is to establish a good value for the serial cutoff for both the mean and the median filter programs. Plot graphs to show what this is in each case.
- Then, to benchmark your parallel algorithms you must time the execution of both your serial algorithms and your parallel algorithm across a range of image sizes. You must then use this data to plot parallel speedup graphs showing the speedup relative to the serial implementations. You should do this on at least **two different machine architectures** (at least one of which must be a multi-CPU/multi-core machine) – e.g. your laptop and the departmental server.

4) Hints and tips

- The `java.awt.image.BufferedImage` class will handle all the image stuff for you.
- Use the `System.currentTimeMillis` method to do the timing measurements. Timing should be done at least 5 times and the smallest value from the set used as each data point (*why the smallest?).* Please note that if the server is under heavy load, it will affect your timing. Be sensible about how you do the timing – you should only time the **core work** of doing the median filter.
- You can use the very simple **Fork/Join blur example** in this Oracle tutorial https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html as a starting point for your parallel solutions. However, please note that this is not a mean filter. Also, do not contravene our plagiarism rules – **any code you submit must be your own code.** We will be checking this.
- There are standard operations for extracting the red/green/blue components of a pixel that are faster than using the class methods. You may find this site useful https://dyclassroom.com/image-processing-project/how-to-get-and-set-pixel-value-in-java

5) Report

You must submit a short assignment report **in pdf format** (please do not submit work documents). Your **concise** report must contain the following:

- A **brief** *Methods* section describing:
  - your **parallelisation algorithms**
  - how you **validated** your algorithms (showed that they are correct),
  - how you **timed** your algorithms accurately,
  - how you established the **optimal serial threshold** for your fork/join algorithms
  - the **machine architectures** you tested on and
  - any **problems/difficulties** you encountered.
- A *Results* section, with **speedup graphs**. Plot graphs to show how the parallel algorithms scale with **image size**, the **size of the median filter window**, and on (at least 2) different computers. **Graphs** should be clear and **labelled** (title and axes). This section should **include a brief discussion** that answers following questions:
  - What is an optimal sequential cutoff for both parallel algorithms? (Note that the optimal sequential cutoff can vary based on dataset size.)
  - For what range of data set sizes/ filter sizes do your parallel programs perform well?
  - What is the maximum speedup obtainable with each parallel algorithm? How do they differ and why? How close is the speedup to the ideal expected?
  - How reliable are your measurements? Are they any anomalies and can you

explain why they occur?

- A *Conclusions* (note the plural) section where you say whether it is worth using parallelization (multithreading) to tackle this problem in Java.

Please do NOT ask for the recommended numbers of pages for this report. It should be short, with clear graphs. Say what you need to say: no more, no less.

1.4 Assignment submission requirements

You will need to submit an **assignment archive**, named with your **student number** and the **assignment number e.g. KTTMIC004_CSC2002S_PCP1.** Your submission archive must contain

- a **short report (in pdf format)**
- **your parallel and serial programs**
- a **Makefile** for compilation
- **a GIT usage log (**as a .txt file, use `git log –all` to display all commits and save).

Upload the archive file and **then check that it is uploaded.** It is your responsibility to check that the uploaded file is correct, as mistakes cannot be corrected after the due date.

The deadline for marking **queries** on your assignment is **one week after the return of your mark.** After this time, you may not query your mark.

1.6 Assignment marking

A draft marking guide is included below.

Marking Guide: CSC2002S 2022 Assignment PCP1		Max. Mark
1. Code:	**conforms to specifications (input, output etc), code correct , timing, style and comments.**	
	Command-line parameters work as specified, programs exits gracefully with incorrect filename or window width	1
	Serial and parallel code produce identical (and correct) output across all test cases	2
	Timing implemented correctly - only the core work timed for both algorithms	1
	Good clear code style and organization, including comments, espcially for divide-and-conquer algorithms	1
2. Report	**all aspects required in the assignment brief are covered**	
	Methods section with all detail included	2
	Clear, **labelled** (title and axes) graphs of parallel speedup versus image size on **two different architectures** .	3
	Discussion of the optimal/expected performance of the algorithms, comparisons between them	4
	Conclusions section	1
	Total	15

Penalties	
Code does not execute/run.	**-5**
no GIT repository.	**-2**
GIT repository, but no regular updates (commits on at least on 5 separate days).	**-1**
No makefile.	**-1**
Late penalty (10% per day or part thereof,).	**-1.5 x d**

Note: submitted code that does not run or does not pass standard test cases will result in a mark of zero. **Any plagiarism, academic dishonesty, or falsifying of results reported will get a mark of 0 and be submitted to the university court**.