Assignment 1 – Parallel Programming

CSC2002S

Dylan Tasdhary

TSDDYL001

11 August 2022

Aim

To determine whether parallelization is worth using for Median and Mean filter image rendering in Java.

Methods

Parallelisation Algorithms

The basic idea behind the parallel algorithms was to split the given `input` image instance variable into vertical strips and allow each thread to then render an individual strip. This was done using the `ForkJoinPool` framework and allowing the parallel classes to inherit from `RecursiveAction`. The `ForkJoinPool` creates the same number of threads as logical processors to ensure that the maximum parallelism is achieved for each specific architecture. A blank `output` image instance variable was used, and each strip would retrieve the `input` pixels from the original input image and set the respective `output` image pixel to the updated pixel value based on the algorithm applied – either Mean or Median. The optimal sequential cutoff (OSC) was the width of the vertical strip being processed. The width of the strip was an instance variable in both parallel classes called `stripWidth`. Each time `compute()` was called and the `stripWidth` was above the OSC, the image was halved. A new instance of the respective class was created for each half. A left instance was created and as well as a new thread with the `fork()` method to handle the left instance. The right halve was handled by the main thread or the current thread that was busy with that instance of the class, but this would also then be subdivided recursively by the same procedure. The left half's processing would start at the same x-value as the instance in which it was created, whereas the right's would start at the halfway point of the instance in which it was created. This process would happen until such point that `stripWidth` was equal to or less than the OSC. Once this condition was met, the `applyfilter()`method was called, which then begins the image rendering process for the given strip.

Algorithm Validation

Test Cases

Please note that these are not the data inputs but rather test cases to illustrate functioning algorithms.

In each case, the same 3 following test images were used with a 5x5 window:

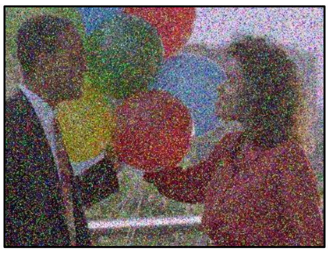


Image 1

Image 2

Image 3

<u>Median</u>

This algorithm is used to remove noise from an image. Due to this, the test cases contain noise to ensure the purpose is met by inspecting the rendered image.

Serial Output:

Rendered Image 1

Rendered Image 2

Rendered Image 3

Parallel Output:

Rendered Image 1

Rendered Image 2

Rendered Image 3

By inspection, the noise is removed and so the functionality of the algorithm is validated.

<u>Mean</u>

This algorithm is also used to remove noise from an image; however, its noise reduction capabilities are limited. The algorithm also results in a blurred image – which is the main indication of the filter working.

Serial Output:

Rendered Image 1

Rendered Image 2

Rendered Image 3

Parallel Output:

Rendered Image 1

Rendered Image 2

Rendered Image 3

Again, by inspection, the noise is poorly reduced, and the rendered images have become blurred, hence, the functionality of the algorithm is validated.

However, this was not enough to ensure that both the serial and parallel algorithms produce the same output i.e. the images are identical. To further validate the algorithms, the java class *ImageChecker.java* was created.

ImageChecker.java

This class receives an image name argument and the type of filter being applied argument. This then obtains an array representation of the pixel RGB values for each image, and then checks that each element in the respective arrays is equal. This class is in the Assignment 1 folder.

Each test case output matched with its respective duo (MedianParallel matched with MedianSerial, and likewise for Mean) which then further validates that the algorithms perform the functionality (the algorithms work), and that both the parallel and serial algorithms render the same image (the algorithms are different yet yield the same result).

Algorithm Timing

Two variables, namely `begin` and `runTime`, of type `long` were used.
`begin` is given the system's current time in milliseconds.
`runTime` is given the system's current time in milliseconds minus `begin`.

Serial

Inside `applyFilter()` in each serial class, the `begin` is set before the iterations that "slide" the focus pixel across the entire image i.e. the actual filtering process, begin. At the end of the filter processing (when the focus pixel iterations end), `runTime` is calculated and stored.

Parallel

Inside `render()` in each parallel class, the `begin` is set before the `ForkJoinPool` is invoked with `invoke()`. Immediately after `invoke()` is completed, `runTime` is then calculated and stored.

Establishing Optimal Serial Threshold

A script to automate this process was created, namely *optimalSeqFinder.py*, which can be found in the *Assignment 1* folder. The script compiles and runs both the Median and Mean parallel files by passing the three (3) input images as command-line arguments. The same window width of 5 was used for all sequential cutoff optimisations. It must be noted that this script cannot be run without making alterations to the main method – this was just done to obtain the values which are stored in the *data* folder.

The `experiment`*()* method in the parallel classes were called. This method iterates from 10 inclusive to 500 exclusive. These were passed as sequential cutoff values. These values were then used to filter the input and record the times. Each sequential cutoff value was tested 5 times before iterating to the next, and the minimum runtime was tracked throughout every iteration. This minimal sequential value for each respective image was then stored in a text file at the end of all iterations.

Testing on Various Machine Architectures

Architecture 1 (A1): Intel(R) Core (TM) i5-8250U CPU @ 1.60GHz 4 Cores 8 Logical Processors

Architecture 2 (A2): Intel Core i5 CPU @ 2.50 GHz 2 Core 4 Logical Processors

Arising Problems

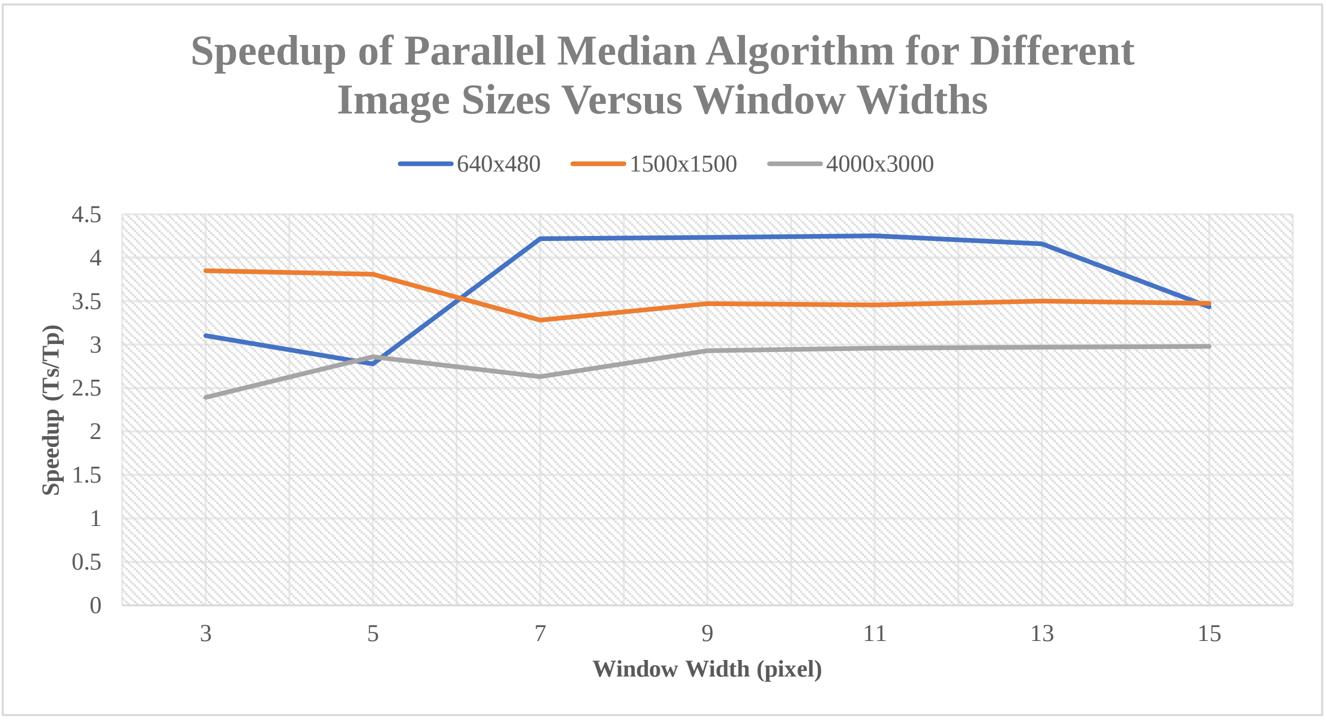
The overheating or rise in temperature as data was being gathered, skewed the results as the CPU performance must be hindered for it to cool. Accurate data was difficult to achieve, however, with this factor being ubiquitous the data gathering on both architectures, the data is still consistent with expected behaviour.

Finding noisy images with suitable sizes for testing validation and gathering data was difficult. These images had undergone lossy compression so visually it was difficult to determine whether the noise from the image had been completely removed which is an indication of a working algorithm in terms of validation testing. After validation testing occurred, regular images were used for the data input due to the need for larger sizes.

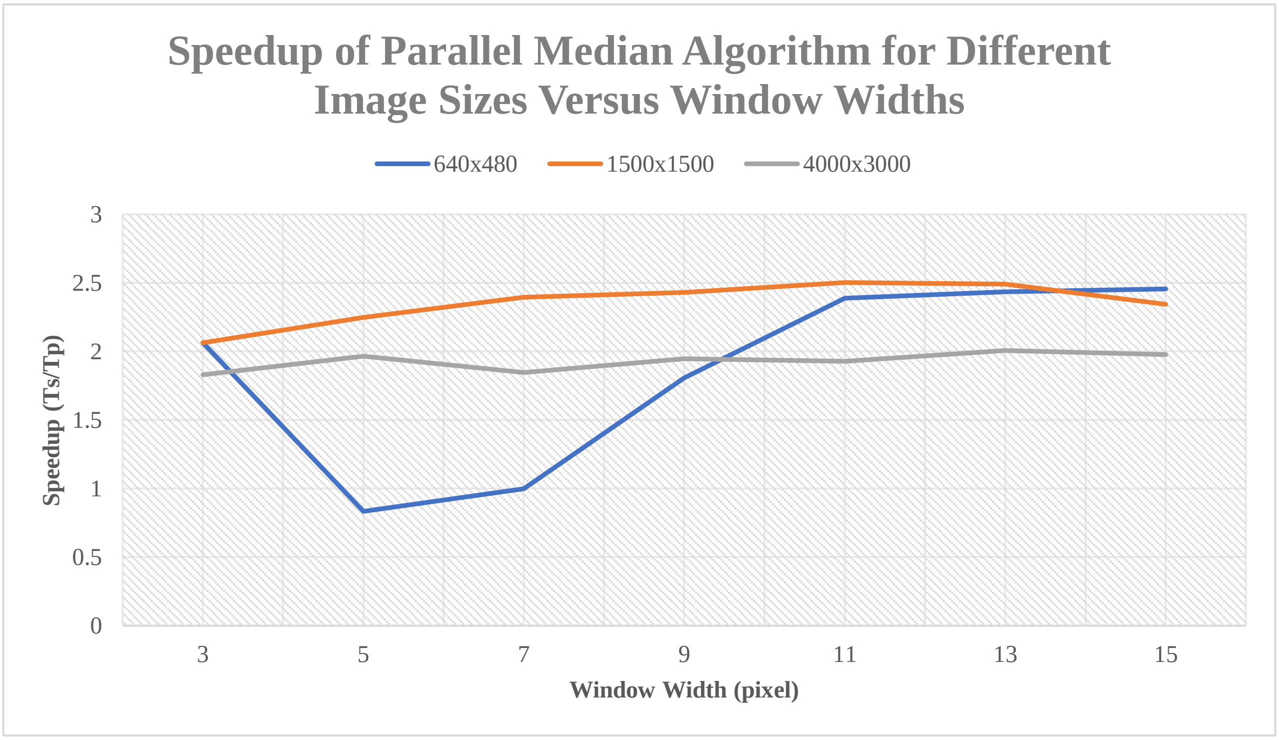
Results

Ts – serial runtime in milliseconds; Tp – parallel runtime in milliseconds

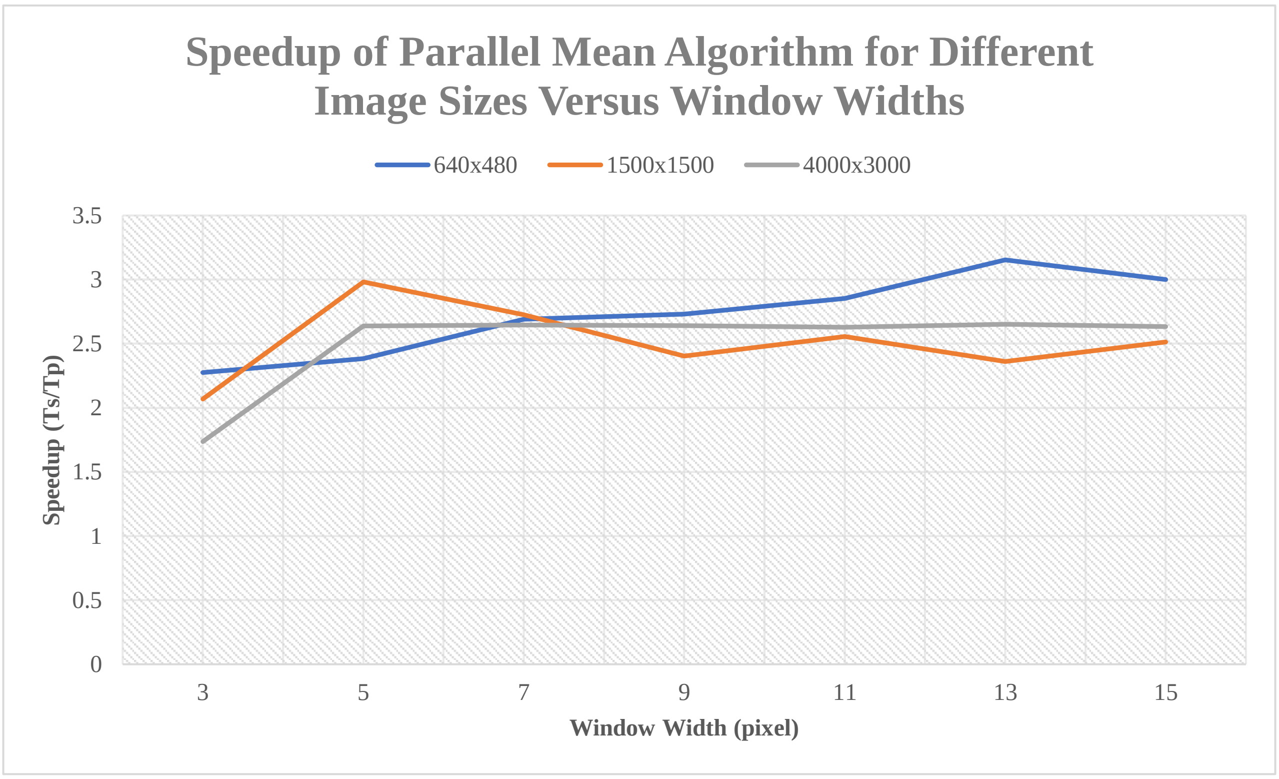
Graph 1: Architecture 1



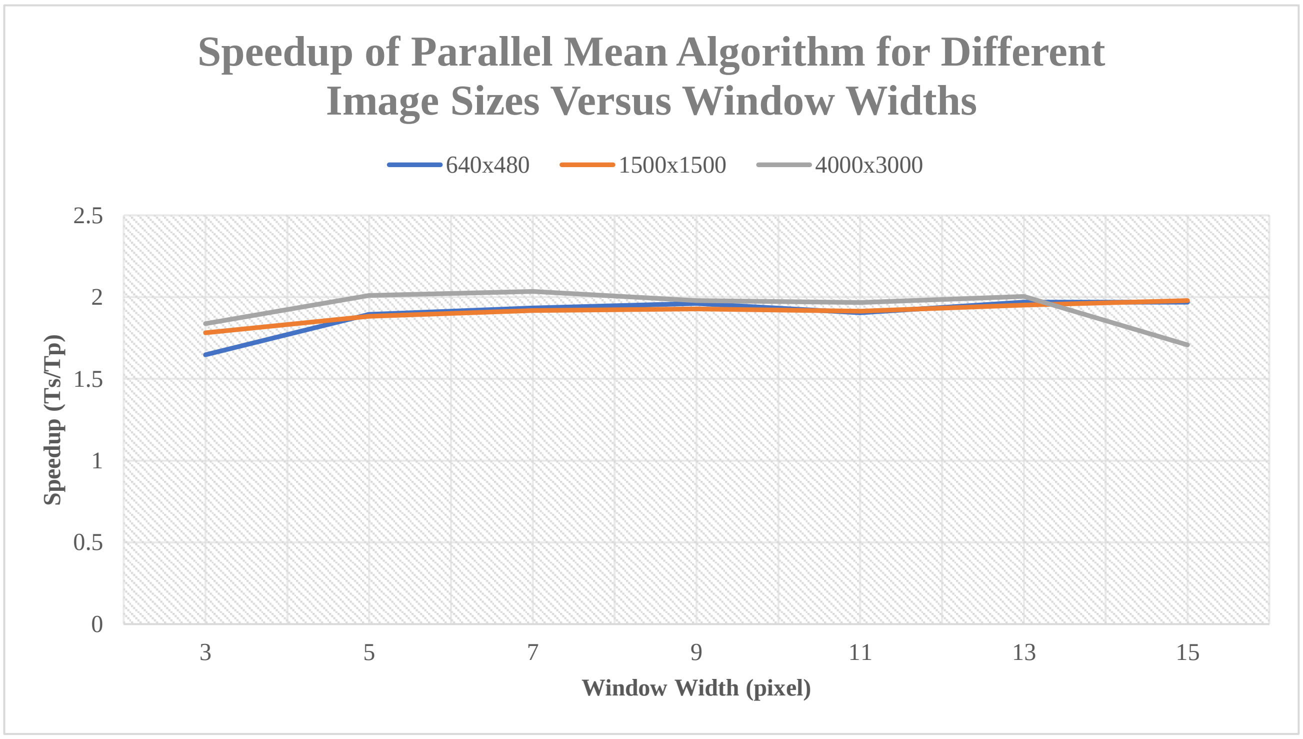
Graph 2: Architecture 2



Graph 3: Architecture 1



Graph 4: Architecture 2



Discussion

Firstly, as mentioned, the OSCs for each parallel program were determined via a script. These changed as the images increased in size. The input image sizes are: 640x480; 1500x1500; 4000x3000. The median OSCs for increasing image size were: 110, 20, 10. The mean OSCs for increasing image size were: 100, 50, 10. These OSCs are almost the same for both mean and median. The decrease in OSC is there due to the image size increasing. As the image size increases, there are more computations to be done, and so having a higher OSC means that the image is split into more strips which the threads and processors can handle i.e. spreading more work across the processors makes for faster runtimes.

The parallel algorithms perform better than the sequential algorithms in general – this is due to sizes of the images in the data set; if the image sizes were smaller, the sequential would perform better. They outperform the sequential for every window width passed – these ranged from 3 to 15 pixels wide – as the difference in efficiency between the sequential and parallel becomes greater the larger the window width becomes. Again, more work to split over the processors results in faster runtimes than having one core do all the computations. For the larger image sizes, and specifically larger window widths, the parallel algorithm is the far better option.

In theory, the maximum speedup obtainable by A1 (4 cores) is 4, and for A2 (2 cores) is 2 as the work/computations can be equally split across all cores. For both Mean and Median filters, A1 has a higher speedup than A2 for example, the median for A1 had an average speedup of around 3.5 whereas A2 had an average speedup of around 2. For both Mean and Median, A1's average speedup was around 2.5 to 3.5. It is expected that A1's speedup should be higher than A2's which is the case for almost all image sizes across the different window widths for both Mean and Median. A1's peak speedup is also much higher than A2's peak speedup – due to A1 having double the number of cores – and this is seen in Median where A1 peaks at just under 4 (the 4.5 is ignored as this is an anomaly to be addressed later) whereas A2 peaks at 2 (again the values over 2 are ignored as these are anomalies). For Mean, A1 peaks at just over 3 whereas A2 peaks at around 2. These peaks come very close to the ideal speedup. The speedups also vary between 2 to 4 for A1 and 1 to 2 for A2 as there are overhead costs like creating the threads, etc. which slow down the runtime.

It should be noted that there are anomalies in the data. This comes in where the speedup is larger than the number of cores. These anomalies arise due to the way the data was collected. The parallel versions were collected first, and then the serial versions via a script. As the parallel versions often utilized 100% of the CPU, this led to the CPU rising in temperature which caused thermal throttling. Thermal throttling is where the CPU reduces its performance to allow it to cool down and prevent overheating. This can be seen in A1's Median graph, as the largest image had a speedup of 3 which should be approaching 4 instead. The largest image was rendered in parallel, and directly after the serial version was run. This led to the serial version's runtimes being longer than they should have, which then affects the speedup formula. The numerator was larger than it ideally should have been, which results in a larger speedup. There are also speedup drops like in A2 Median's graph where the speedup drops below 1. Other tasks in the background were running which affects CPU utilization and this then affects the runtime.

Conclusions

Using parallelization for the Median and Mean filter image rendering in Java is worth it. The large number of computations needed for the process, especially as the image sizes and window sizes grow, can be easily dealt with by the different cores to allow for faster runtimes while still producing the same output as the serial versions. Also, increasing the number of cores with which one would do this, would amplify these benefits. However, this should be done when the CPU is not needing to run any other background tasks as this takes away from the effect of parallelization.