TAKUDZWA SHERPHERD MASEVA

MSVTAK002

Parallelisation algorithms

1. *MeanFilter.java*

This class extends the abstract class, Recursive Action that does not return a result. It is a subclass of ForkJoinTask that allows tasks to be executed on a separate core in a multicore system hence allowing for parallel tasks.
In this class, I override the protected compute method, which splits my program into small tasks that in turn will be handled by threads from the ForkJoinPool. If the total pixels of my image calculated as width x height of the image are less than my sequential cut off, then the program executes sequentially to filter portions of the image directly. Otherwise, if the total pixels of the image are bigger than the sequential cut off then the image is split into subsections which and will be filtered in parallel until all the subsections of the image have been filtered. This achieved by the 8 threads that l create to work on the divided image as well. Lastly I invoke all the threads that l have created in the compute method.
In the main method, l have allowed then image input and output as well as the window width to be passed as arguments on the terminal and if the window width is greater than or equal to three and is odd, then the blu image method is invoked which filters the image and is also where I create my pool of threads and invoke them.To read the image, l took use of the BufferedImage class, which is a subclass of the Image class and can be used to handle and manipulate the image data. After the image has been filtered it is written to a file.

2. *MedianFilter.java*

This class extends the abstract class, Recursive Action that does not return a result. It is a subclass of ForkJoinTask that allows tasks to be executed on a separate core in a multicore system hence allowing for parallel tasks.
In this class, I override the protected compute method, which splits my program into small tasks that in turn will be handled by threads from the ForkJoinPool. If the total pixels of my image calculated as width x height of the image are less than my sequential cut off, then the program executes sequentially to filter portions of the image directly. Otherwise, if the total pixels of the image are bigger than the sequential cut off then the image is split into subsections which and will be filtered in parallel until all the subsections of the image have been filtered. This achieved by the 8 threads that l create to work on the divided image as well. Lastly I invoke all the threads that l have created in the compute method.In the main method, l have allowed then image input and output as well as the window width to be passed as arguments on the terminal and if the window width is greater than or equal to three and is

odd, then the blu image method is invoked which filters the image and is also where I create my pool of threads and invoke them.To read the image, I took use of the BufferedImage class, which is a subclass of the Image class and can be used to handle and manipulate the image data. After the image has been filtered it is written to a file.

Validation of algorithms

How: To validate that my algorithms are correct, I filtered the given image using the serial versions of the different filtering techniques and compared those pictures with the parallel versions of the same filtering methods holding the window width constant.

Results:

1. **MedianFilterSerial(top) vs MedianFilterParallel(bottom)**

2. MeanFilterSerial(top) vs MeanFilterParallel(bottom)

Timing of Algorithms

How: To time my algorithms, l used the System.currentTimeMillies method which returns the current time in milliseconds. After creating the pool of threads in my algorithms, I then started my time there and stored it in a variable called start of the type long. After invoking the pool and writing the filtered image to a file l then created another variable to store the end time using the System.cureentTimeMillis method again. Now to determine how long it took for the process to complete, l subtracted the start time from the end time and then divided it by 1000.

The program was run 4 times each time noting the time taken to complete the process and then recording the smallest of all the values obtained as the time taken to complete the process.

Optimal Serial Threshold

How: To determine the optimal serial threshold or sequential cut off, l simply performed search by running my code multiple times varying the threshold value and each time noting the time it took to complete the filtering process. This was repeatedly done several times till the optimal serial threshold value was found with the lowest running time.

From the experiment it was noted that higher very high threashholds resulted in higher completion times as the program would become a sequential one. However very low threshold values would also cause the algorithm to not work properly.

Machine Architectures

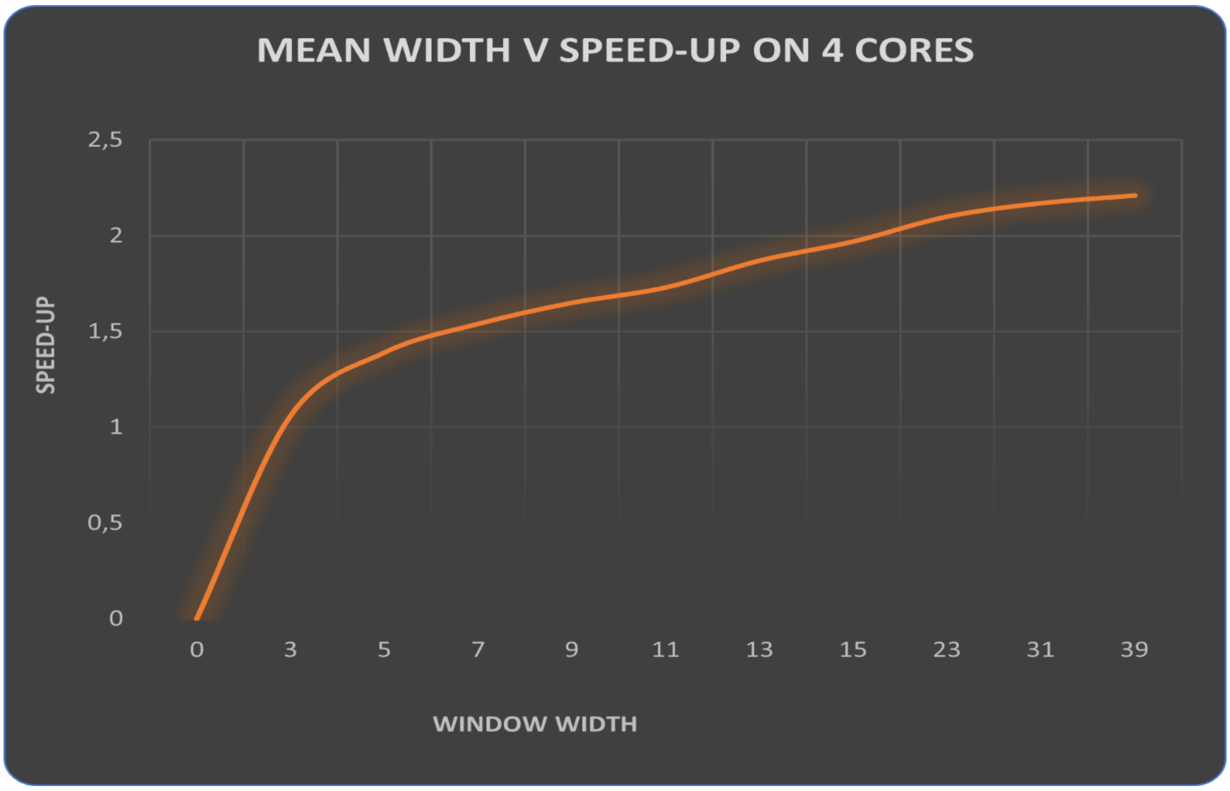
To test my programs, l tested on my 4 core machine first and l then doubled the cores to test on an 8 core machine in Ishango lab. During testing l encountered a few problems that include having to wait for a very long time to get the run time of some of the serial algorithms when for example the window size increased to a big one especially with the median filter serial algorithm. Another problem that I encountered was to determine the serial threshold for my parallel algorithms as l had to do a though trial and error to try and establish the optimal threshold. The other problem that l encountered was when l tried to use the nightmare server to try and use a different computer architecture but got results that were suspicious probably due to the overload on the server by many people trying to do the same.

Results

1. ***a) Mean Width Speed-up on 4 cores varying window width***

To get the speed-up, of this experiment, l ran several tests on each of the 11 test cases and recorded the smallest time l had obtained from the runs. This was run on a 4-core machine and for the parallel and serial program. For the serial program, it took more time to complete filtering an image than it took for a parallel algorithm to complete the process. However, for the first few window widths, the difference was narrow however for the big window sizes, it became evident that the parallel algorithm was very fast.

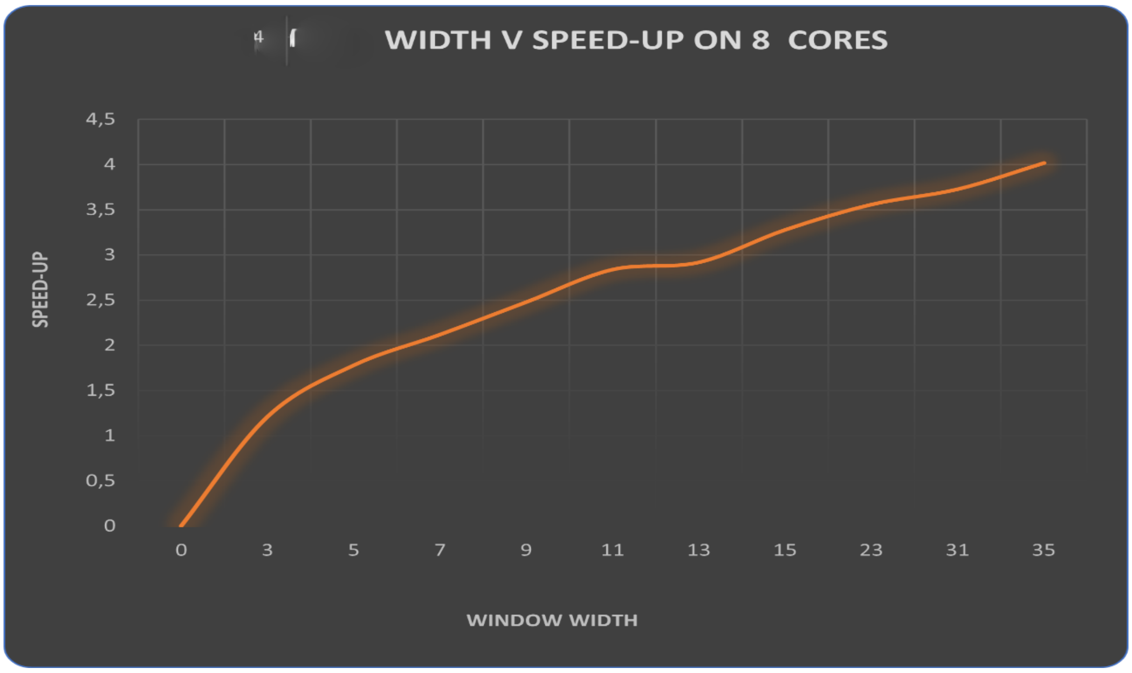
The speed up was then calculated as serial time/ parallel time. The optimal sequential cut off for the parallel algorithm was 10 000.



From the graph above, it can be noted that the speed-up was relatively good from a window width of 3 right up to the end with a high speed of around 2,25. However there were some anomalies that were noted when the window size kept on increasing and the speedup went above what is expected, it kept on increasing and there were also minor dips. In conclusion it is worthy it using parallelization to tackle this problem in java if it's going to be applied on images with a big window size because of the efficiency it comes with even when the window size becomes big.

b. ***<u>Mean Width Speed-up on 8 cores varying window width</u>***

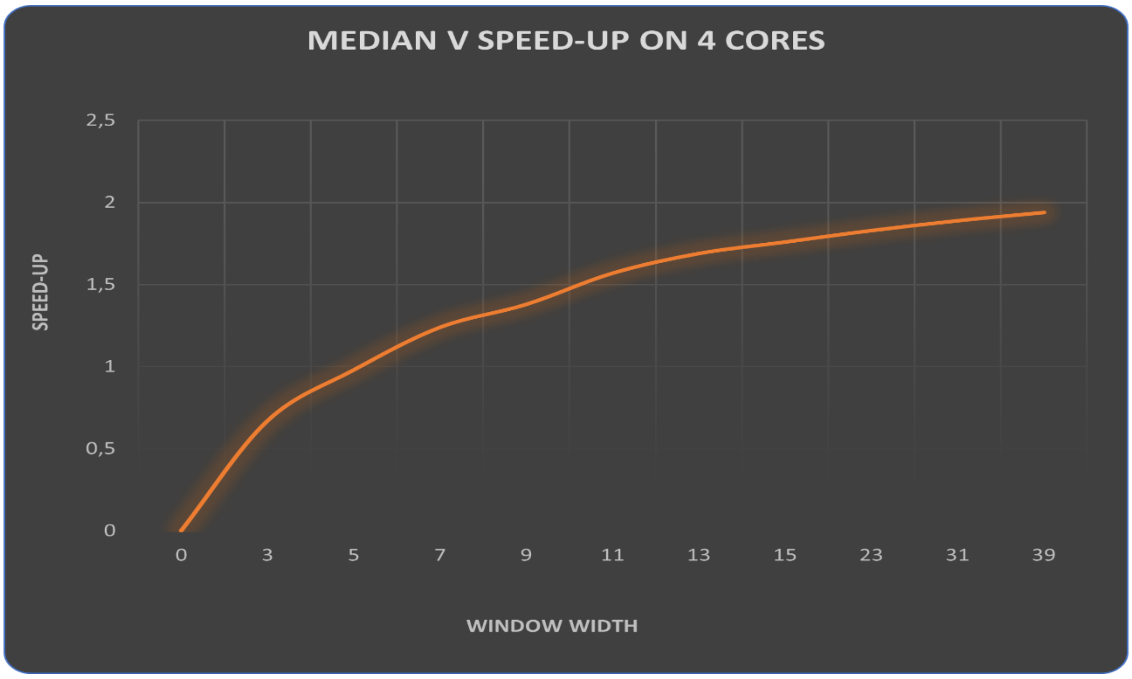
The same experiment was then repeated, but this time on an 8-core machine to see how the speed-up would improve on a bigger machine.



From the graph above it can be noted that the speed-up on an 8-core machine is higher than it is on a 4-core machine. This is due to the increase in the number of processors on the 8-core machine, hence now different parts of a program can be executed in parallel. There are also anomalies in the results as the speed-up on bigger window size almost doubled which is not supposed to be the case as more time is lost as threads wait for another to complete before continuing. In conclusion, it is still worthy it to use java to tackle this problem if the window size is going to be big.

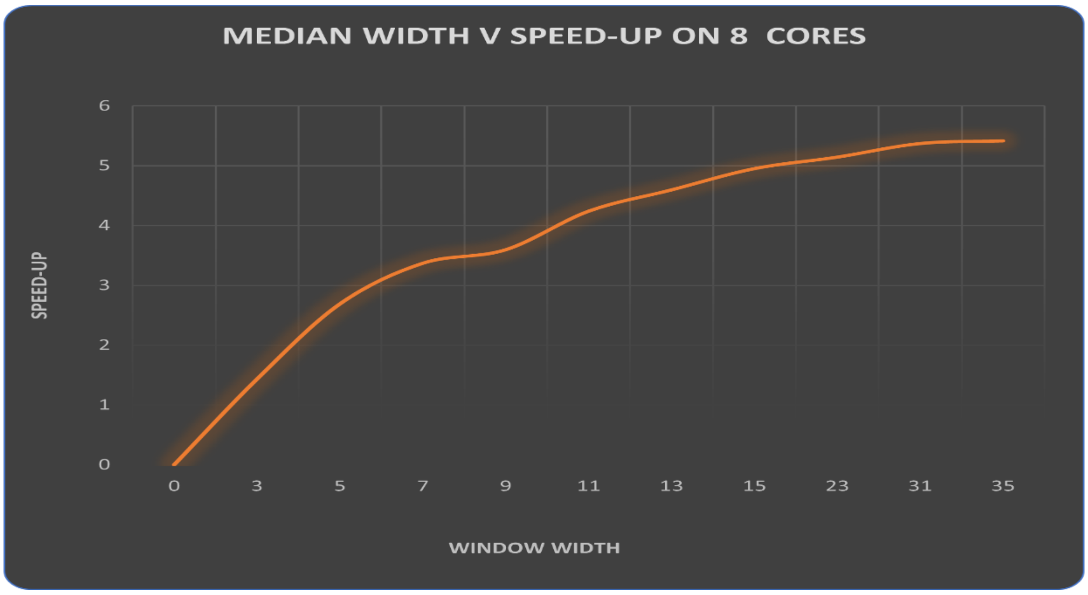
2. a) ***Median Width Speed-up on 4 cores, varying window width***

This experiment was also done by running several tests on each window width before taking note of the smallest time taken. This was run on both a 4-core machine and on an 8-core machine. The optimal sequential cut-off for both was 10 000. This was performed on a set of data set ranging from a window size of 3 up to 39.



On a 4-core machine the speed-up was relatively good and within the expected speed-up. The highest speed-up recorded was about 1,97. The highest speed-up recorded for this median filter was lower than the highest recorded for the mean, of 2,25 mainly because median is heavy as it involves finding the average of the pixels and sorting of the pixels.

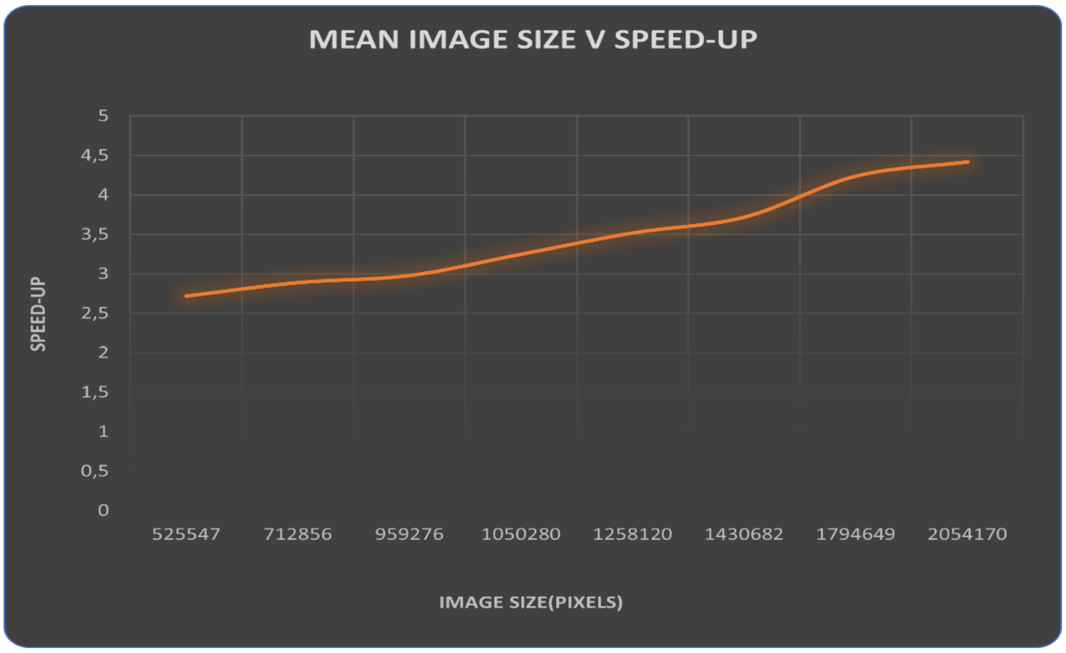
On the 8-core machine, the speed-up was much higher than on the 4-core machine with a high speed of around 5,24 which is also an anomaly because it also almost doubled with a change in cores.



This problem is also worthy it solving it using parallelization regardless of the window size since the mean filtering becomes heavy from even a small window size when tackled in serial as it is involves sorting and averaging.

3. Varying Image Size

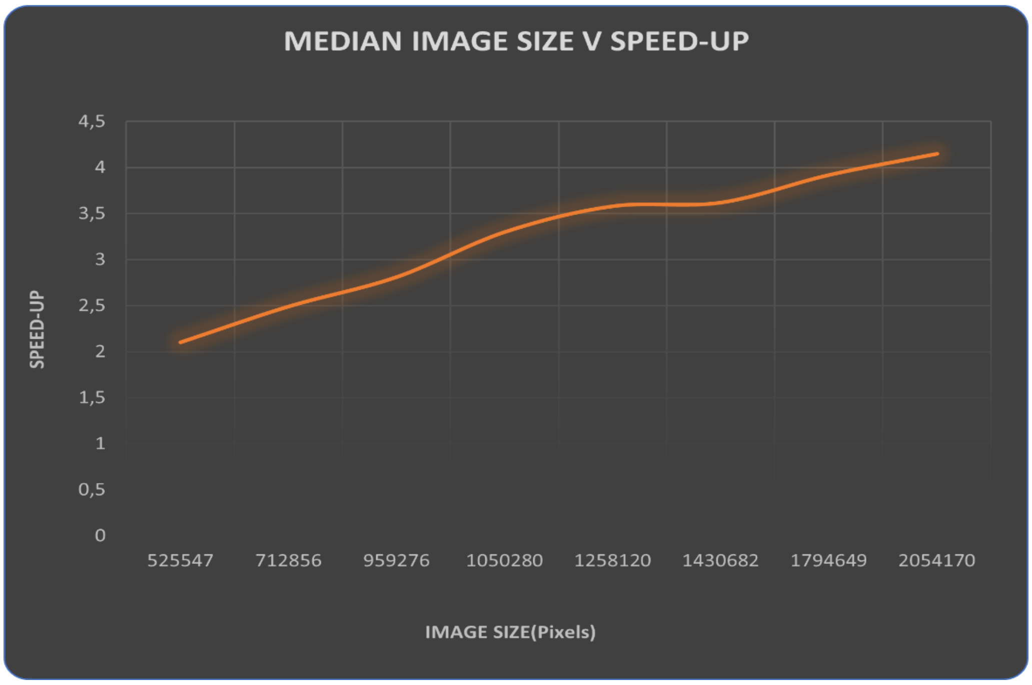
a. Mean Filtering



The optimal sequential cut-off for both algorithms was still maintained at 10 000. The dataset used ranged from 525 547 image pixels to around 2 050 000 pixels. The maximum speed-up that was recorded from this experiment was around 4,43. The speed-up increases slowly as the number of image pixels increase due to the efficiency of the parallel algorithm because it takes less time to filter a big image. It can be concluded that parallelization is worthy it to tackle this problem with java as it has a high speed-up even with big pictures

b. Median Filtering

For median filtering, the optimal threshold was maintained at 10 000. The same dataset was used as for the mean filtering above. The maximum speed-up that was recorded form this experiment was around 4,24 which is slower than the mean filter. The speed – up also increases as the total number of pixels increase due to the efficiency of the parallel algorithm to filter a big image. It can also be concluded that parallelization is worthy it to tackle this with java if there are going to be big images.



Conclusion

From the speed-up graphs, it was indeed worth it to