MWNMSI001

A study on the benefit of using parallel programming in image processing as compared to serial/sequential programming

Abstract

Parallel programming is programming strategy that aims to distribute individual tasks within a program to individual cores within a multicore computational device. This is done when the magnitude of the overall task of the program is enormous and would result in a very time expensive program execution. By using a parallelly programmed program, the execution can be reduced drastically. However, there is a cost that is paid as there is additional programming required to coordinate the distrusted tasks such that the final output is correct and usable.

Introduction

In this experiment, one of the applications of parallel programming will be investigated. An image processing program will be constructed both in a serial(normal) manner of programming and in a parallel manner of programming. Execution time will then be evaluated to determine whether there is in fact a benefit in using parallel programming in image processing. In particular, the programs will be executing the task of image blurring and image noise filtering.

Methodology

1. Parallelization algorithm

The parallelization algorithm used in this experiment makes use of the Java Fork-Join framework. This framework achieves parallelization by creating forks in the dataset of the original input file and then rejoining the forks after processing. This means that the original dataset is split recursively into two subsets until a certain the base case for the data size is reached. When the base case is reached, the resulting subset of data is then processed and recursively rejoined with other processed subsets to form the final processed dataset that is of the same size as the original unprocessed dataset.

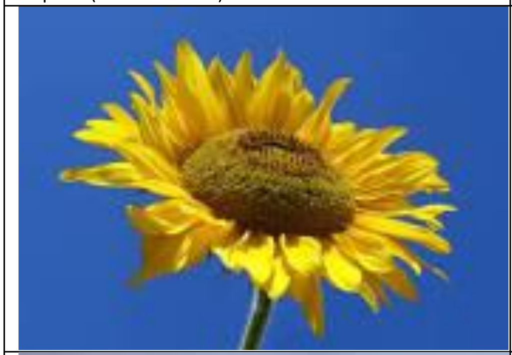
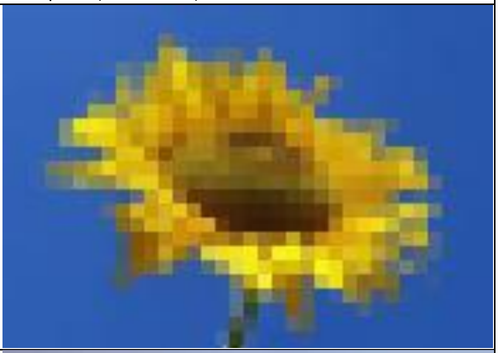
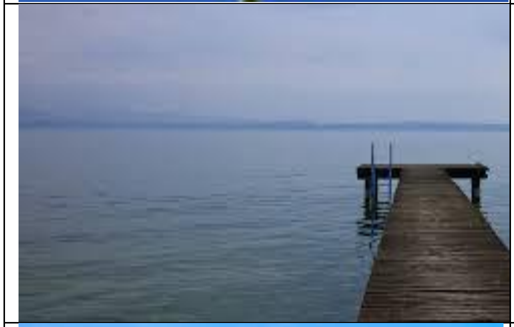
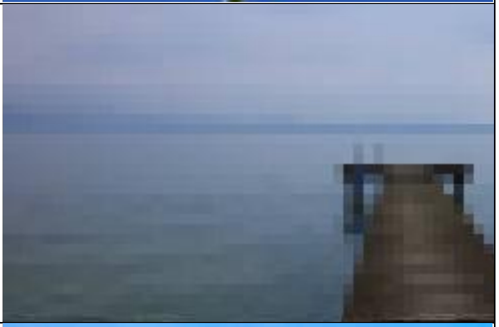
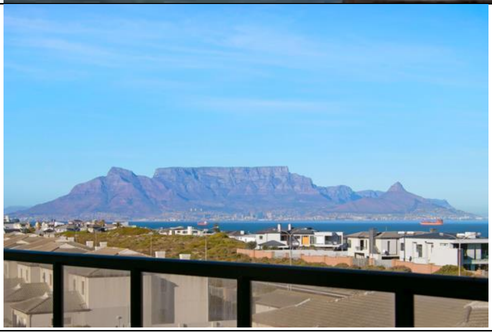
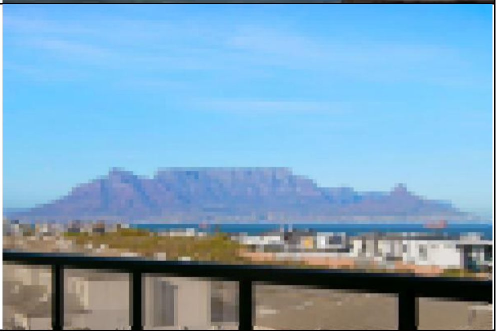
In this experiment, an image was read into the program followed by the extraction of its pixels. the pixels were then grouped into arrays according to the window size that would be used to filter the image. The grouped pixel arrays were then stored in an arraylist. During parallelization, forks were created in the arraylist and a finite number of grouped pixel arrays were assigned to each thread for processing. After processing, each thread saved the processed groups of pixels into arraylists. After all threads had completed execution these arraylists within rejoined to form the processed version of the original arraylist.

2. Validation

To validate the operation of the programs that we used in this experiment, multiple unprocessed and/or noisy images were fed into the programs and the outputs were observed. The same inputs were provided to the serial and parallel versions of each program and the same outputs were observed.

a. <u>Mean Filter Validation</u>

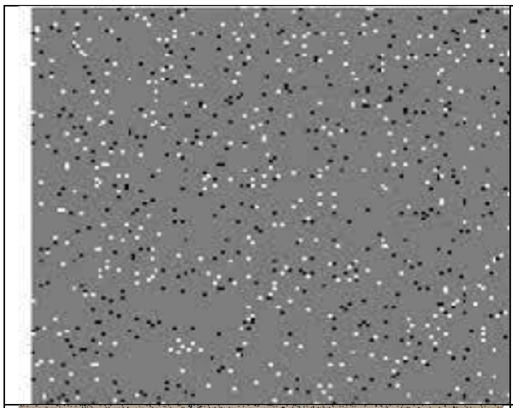
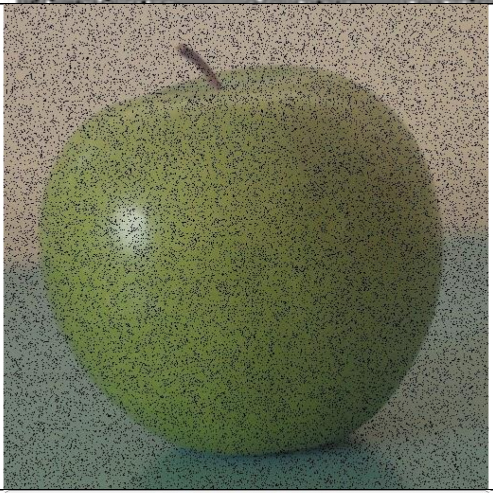
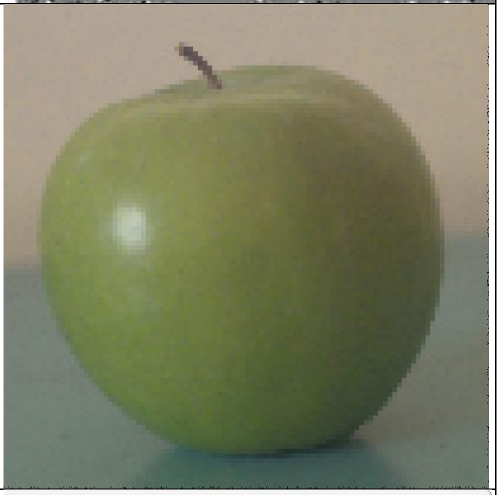
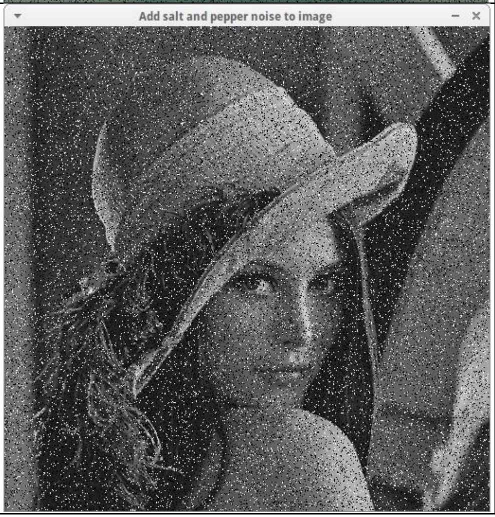
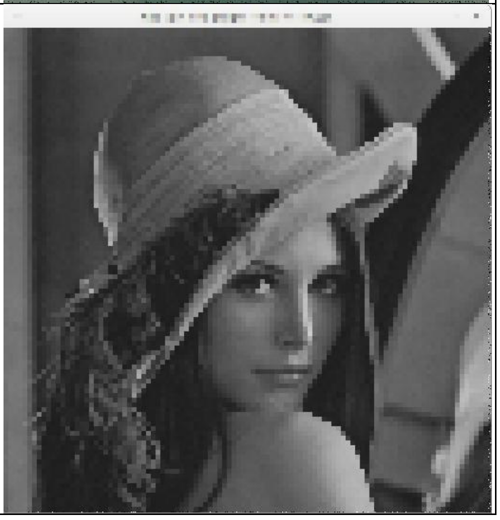
The expected behavior of the mean filter is to blur an image therefore a noise-free clear image is used as an input and a blurry image is expected at the output.

Input (Unfiltered)	Output (Filtered)
	
	
	

b. <u>Median Filter Validation</u>

The expected behavior from a median filter is to remove noise from an image. Some blurring is also expected from such a filter; however, noise filtering is expected to be the main behavior therefore a noisy image is used as an input and a noise-free image as expected at the output.

Input (Unfiltered)	Output (Filtered)

3. <u>Timing</u>

   To time each iteration offer filters execution the ***currentTimeMillis()*** method found in the Java System class was used i.e. ***System.currentTimeMillis()***.This method returns a value of type ***long*** of the timestamp at the point at which it is called in the program. In this experiment, the method is called at the beginning of execution and the timestamp at the end of the program's execution. The difference between these two timestamps is then calculated as the program's execution time. The unit of measurement is milliseconds.

```
long start = System.currentTimeMillis();


********Program executes*********


long finish = System.currentTimeMillis();

long executionTime = finish – start;
```

4. Optimal serial threshold selection

To find the optimal serial threshold, 10 iterations of the program were executed and incremented at intervals or 10% of the original size of the input data. The threshold with the quickest average execution time across different input image sizes, was chosen as the optimal serial threshold.

Mean Filter				
PercentageSize[%]	Time (flower.jpg) [ms]	Time (bridge.jpg) [ms]	Time (cpt.jpg) [ms]	Average Time
10	105	103	175	128
20	93	162	174	143
30	95	110	172	126
40	96	111	184	130
50	93	121	183	132
60	90	152	174	139
70	251	263	157	224
80	93	132	164	130
90	102	146	218	155
100	128	119	170	139

Because 30% yielded the quickest average execution time across different image sizes, an optimal threshold of 30% of the original data size was chosen for the parallel Mean filter.

Median Filter				
PercentageSize[%]	Time (blank.jpg) [ms]	Time (apple.jpg) [ms]	Time (lady.jpg) [ms]	Average Time
10	476	1151	1388	1005
20	315	839	811	655
30	197	607	629	478
40	234	621	587	481
50	245	578	574	466
60	165	434	395	331
70	280	421	448	383
80	157	471	402	343
90	217	407	397	340
100	228	418	395	347

Because 60% yielded the quickest average execution time across different image sizes, an optimal threshold of 60% of the original data size was chosen for the parallel Median filter.

5. Machine architectures

The programs were tested on the following devices:

a. Device 1
   i. Name: Dell Latitude 5490
   ii. Processor: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz   1.80 GHz
   iii. No. of cores: 4
   iv. RAM: 8GB
b. Device 2
   i. Name: sl-dual-197
   ii. Processor: Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz x 4
   iii. No. of cores: 4
   iv. RAM: 8GB

6. Problems/difficulties

a. Ensuring that that the filter window captures the edge pixels of the image accurately.
b. Establishing a solid relationship between the independent and dependent variables in the experiment due to the uncertain nature of thread execution coupled with the general presence of outliers.

Graphs and Tables

1. Device 1

Mean Filter				
Image Size (Dimensions)	Image Size (Area)	Serial Time [ms]	Parallel Time [ms]	Speedup
173 x 121 (flower.jpg)	20933	110	202	0,544554455
318 x 159 (bridge.jpg)	50562	156	151	1,033112583
600 x 400 (cpt.jpg)	240000	441	188	2,345744681
Median Filter				
Image Size (Dimensions)	Image Size (Area)	Serial Time	Parallel Time	Speedup
253 x 199 (blank.jpg)	50347	162	232	0,698275862
512 x 512 (apple.jpg)	262144	506	403	1,255583127
514 x 537 (lady.jpg)	276018	500	532	0,939849624

Figure 1: Table showing the speedup in the Serial Mean Filter vs Parallel Mean Filter and the Serial Median Filter vs Parallel Median Filter as image size increases

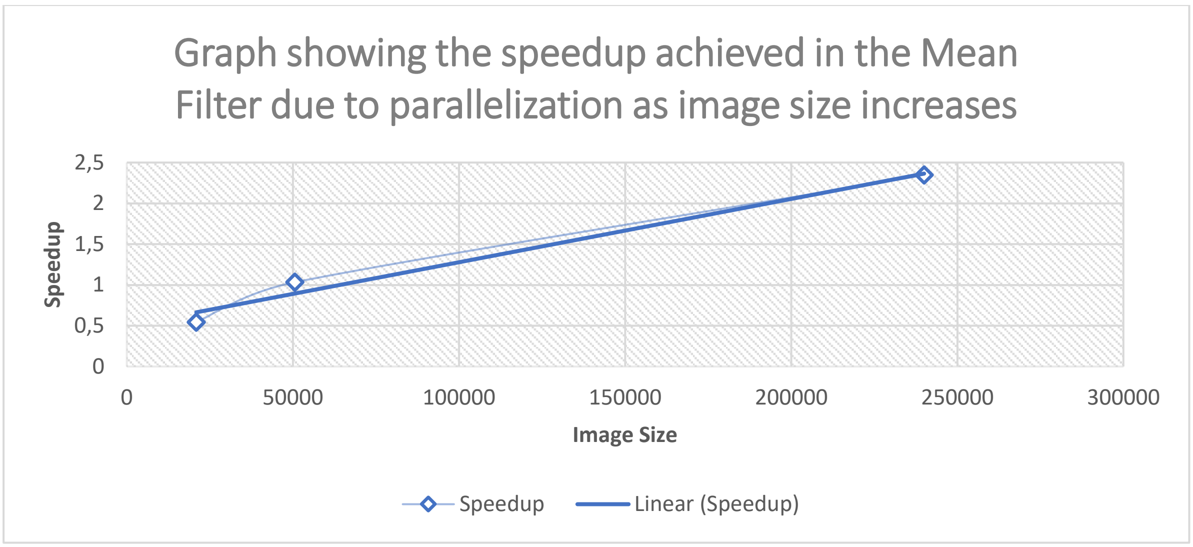


Figure 2: Graph of the Mean Filter speedup

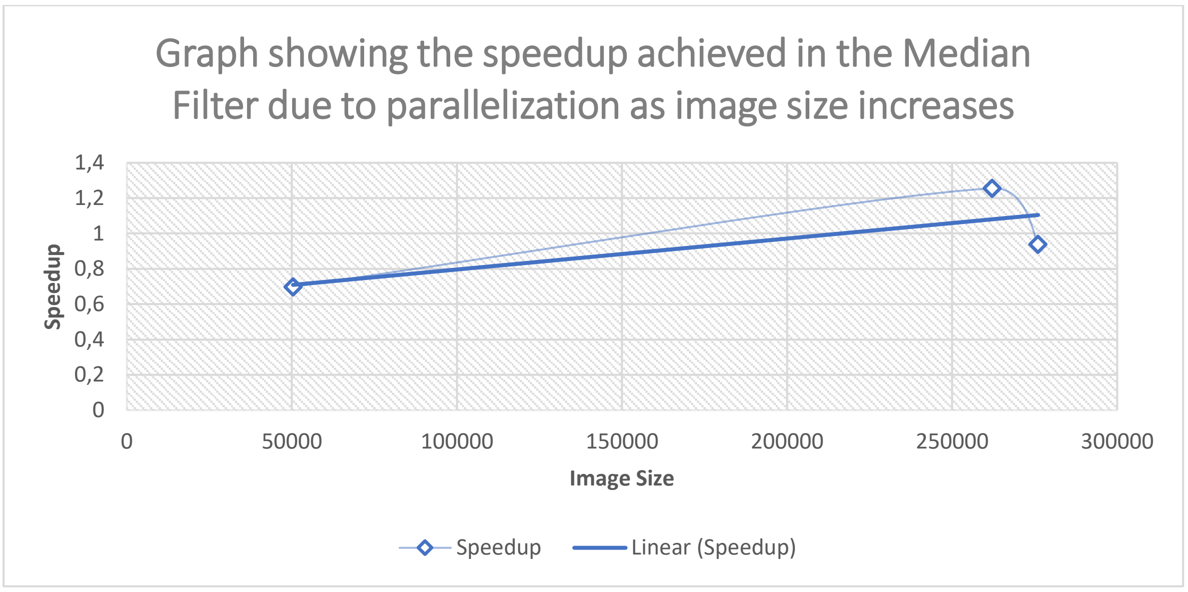


Figure 3: Graph of the Median Filter speedup

Median Filter				
Window Size (Dimensions)	Window Size (Area)	Serial Time [ms]	Parallel Time [ms]	Speedup
3 x 3	9	295	571	0,51664
5 x 5	25	500	532	0,93985
7 x 7	49	383	441	0,86848
9 x 9	81	450	494	0,91093
11 x 11	121	428	356	1,20225

Figure 4: Table showing the speedup achieved in the Median Filter as the of the filtering window increases

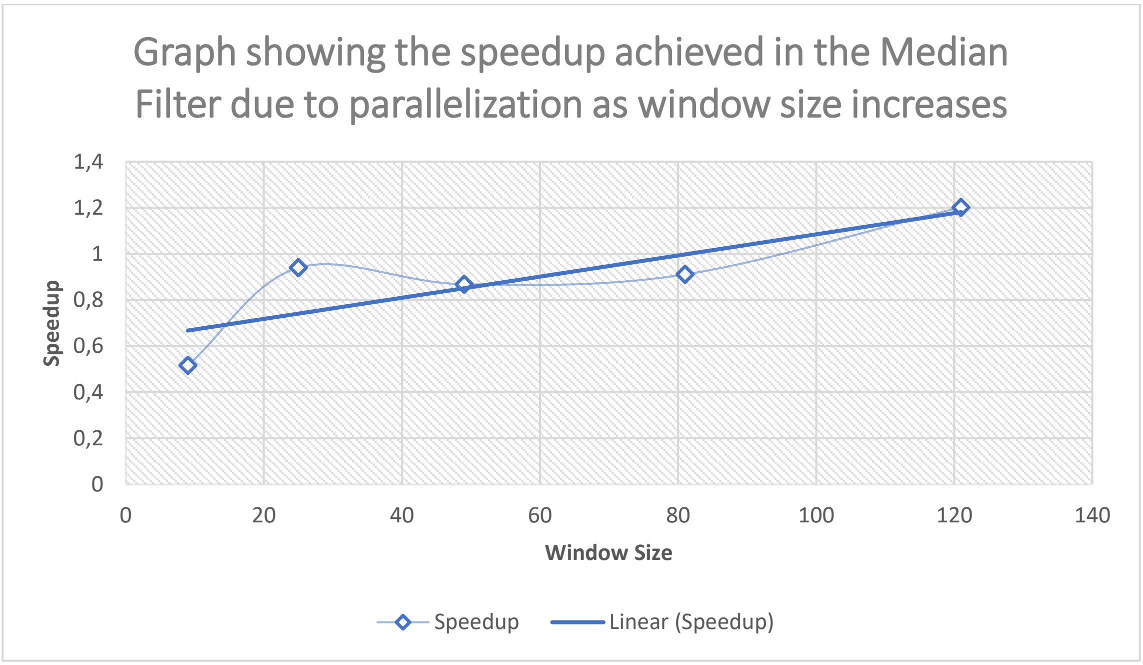


Figure 5: Graph of the Median Filter speedup with varying window size

2. Device 2

Mean Filter				
Image Size (Dimensions)	Image Size (Area)	Serial Time [ms]	Parallel Time [ms]	Speedup
173 x 121 (flower.jpg)	20933	315	193	1,632124352
318 x 159 (bridge.jpg)	50562	336	199	1,688442211
600 x 400 (cpt.jpg)	240000	265	150	1,766666667
Median Filter				
Image Size (Dimensions)	Image Size (Area)	Serial Time	Parallel Time	Speedup
253 x 199 (blank.jpg)	50347	255	197	1,294416244
512 x 512 (apple.jpg)	262144	518	359	1,442896936
514 x 537 (lady.jpg)	276018	582	364	1,598901099

Figure 6: Table showing the speedup in the Serial Mean Filter vs Parallel Mean Filter and the Serial Median Filter vs Parallel Median Filter as image size increases

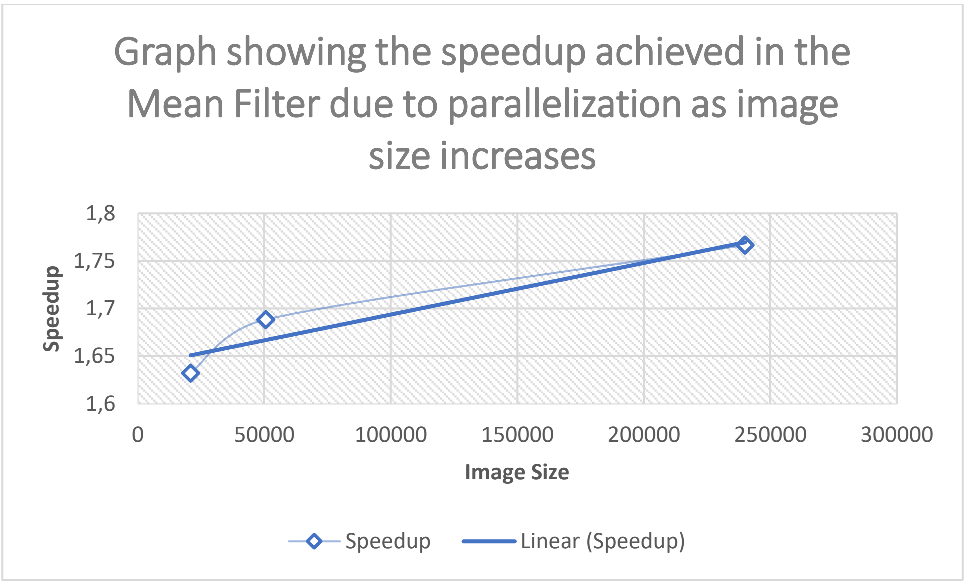


Figure 7: Graph of the Mean Filter speedup

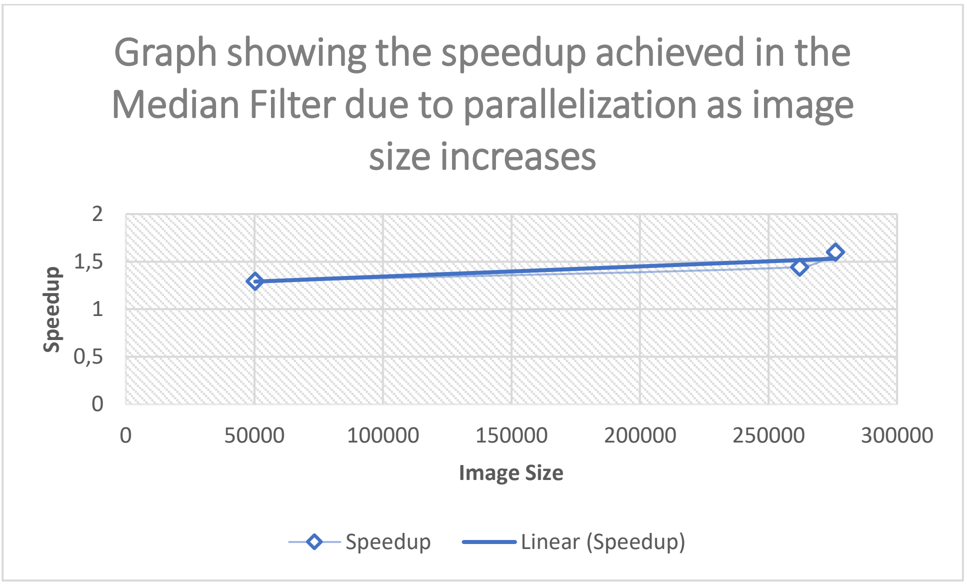


Figure 8: Graph of the Median Filter speedup

Median Filter				
Window Size (Dimensions)	Window Size (Area)	Serial Time [ms]	Parallel Time [ms]	Speedup
3 x 3	9	103	453	0,227373068
5 x 5	25	114	453	0,251655629
7 x 7	49	278	155	1,793548387
9 x 9	81	124	210	0,59047619
11 x 11	121	218	153	1,424836601

Figure 9: Table showing the speedup achieved in the Median Filter as the of the filtering window increases

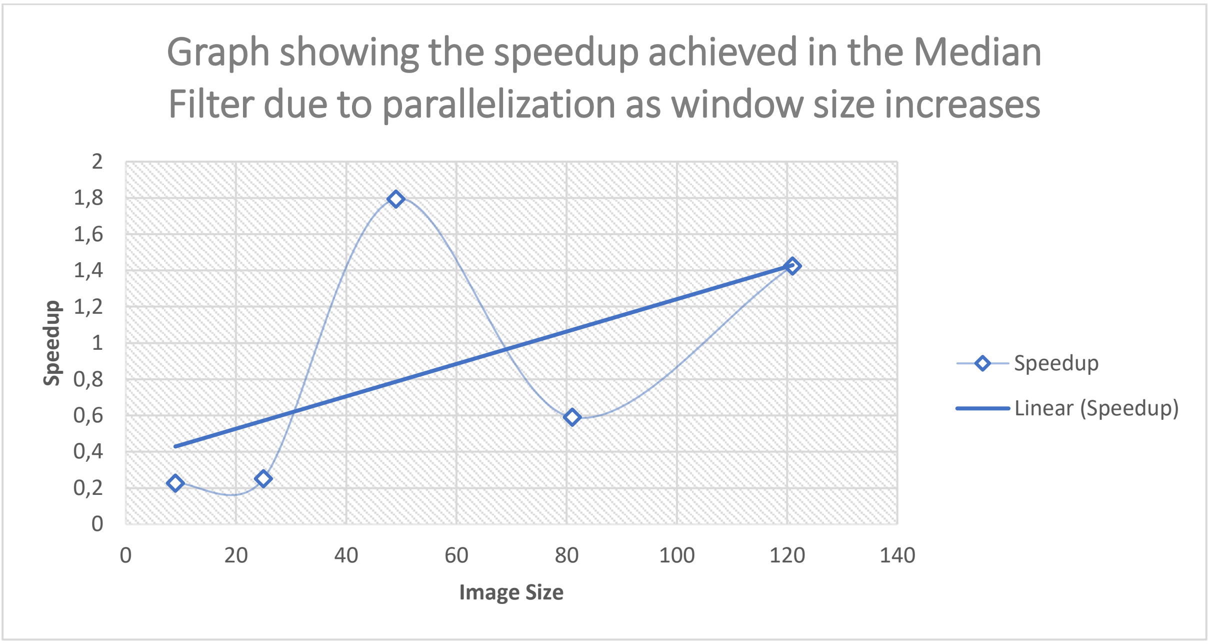


Figure 10: Graph of the Median Filter speedup with varying window size

Theoretical Analysis

The comparison of serial and parallel programs is characterized by the notion of speedup. This is defined as the ratio between the execution time of a program running serially and the execution of time of the same program running parallelly i.e.

$$Speedup = \frac{T_{serial}}{T_{parallel}} \quad [1]$$

What is expected is a program to achieve positive linearly increasing speedup as input data size increases. This means that as input data size increases, the parallel version of a program should always take less time to complete execution when compared to the serial version of that program. This is what motivates the use of parallelization in programs.

Analysis of experimental Results

- An optimal sequential cutoff of 30% of the original dataset size was chosen for the parallel mean filter and an optimal sequential cutoff of 60% of the original dataset size was chosen for the parallel mean filter. The reasons for choosing these values were discussed in the methodology section above.
- The parallel programs depicted the expected behavior of running faster than the serial program, with a more prevalent execution time benefit occurring at image sizes greater than 173 x 121 (parallel mean filter) and 253 x 199 (parallel median filter).
- In this experiment a maximum speedup of 2,35 was obtained for the Mean filter and a maximum speedup of 1.6 was obtained for the Median filter.

- The maximum speedup of Mean filter and the maximum speedup of the Median filter differ by a factor 0.68. The maximum speedup was lesser in the Median filter as the sorting algorithms from ***Java.util.ArrayList.sort()*** used to sort the pixels being filtered have a time complexity of nlog(n). This increases execution and results in a lesser time benefit.
- The results showed the expected general trend with a few anomalies due to experimental noise and the uncertain nature of thread execution. However, the general trend is inline with the theoretical definition of speedup. The results also showed a similar trend across different devices thus showing that the results are precise and regular.

Conclusion

The speedup obtained in the Mean filter was positive and linearly increasing. This is in keeping with the expected behaviour as defined in the theoretical analysis. This behaviour was observed across different devices an thus can be accepted. The speedup obtained in the Median filter was positive and linearly increasing. This is in keeping with the expected behaviour as defined in the theoretical analysis. This behaviour was observed across different devices an thus can be accepted. The speedup due parallelism was more prevalent in larger datasets and proved to be more time expensive in smaller datasets. This was because of the additional computation that is involved in multithreading. This means that it is more worth it to use parallel computing for larger datasets and that it is more expensive to run parallel programs for smaller datasets.

References

[1] Slideplayer.com. 2022. *Parallel Computing Lecture - ppt download*. [online] Available at: <https://slideplayer.com/slide/13453241/> [Accessed 16 August 2022].