CSC2002S

Assignment 1: Parallel Programming with the Java Fork/Join framework:
2D Median Filter for Image Smoothing

Table of Content

I. Introduction

This assignment is about implementing and comparing the performance of sequential algorithms on one side and parallel algorithms on the other side. The implementation of the parallel algorithm is executed using the Java Fork-Join framework.

The aim of this report is then consisting of:

- Implementing the algorithm of two serial filters (mean and median) for smoothing RGB colour images;
- Implementing the divide–and-conquer algorithm of two parallel filters (mean and median) by means of the Fork-Join Parallelism for smoothing RGB colour images;
- Benchmarking the parallel program to determine under which conditions parallelization is worth the extra effort involved

In the implemented algorithms, we will process an original and smoothed it with a mean filter which consists of setting each pixel in the image to the average of the surrounding pixels, and the same original will be smoothed with a median filter consisting of setting each pixel to the median of the surrounding pixels.

Using a sliding square window of a specified width w (w is an odd number >=3) for both the mean and median methods, the mean filter will simply average all the pixels within the window. However, the median filter will be sorting the pixels into numerical order, and then the target pixel will be replaced with the middle (median) value of the window.

The different versions implemented in this assignment are:

- Two serial programs to apply a mean and median filters to a specified image.
- Divide-and-conquer parallel programs using the Java Fork/Join framework in order to speed up the median and the mean filter process.

II. Different Classes

The filtering algorithm has been implemented sequentially (MeanFilterSerial, MedianFilterSerial Classes) first and it was kept modular so that most of the code could be reused for the future multithreaded (MeanFilterParallel Class, MedianFilterParallel Classes) implementation. Each class has its own main method in which the original image is read into a buffered image object using the ImageIO.read method. The buffered image class represents the image data such as the pixels, colour space and dimensions and it provides us with convenient methods to manipulate the pixels of the image. This class will also contain the result image object in which will be stored the output image.

III. Methods of Testing

III.1. Theoretical Number of cores used in testing

The performance testing in this report consisted of looking into the time of completion of the filtering processing task. This time was measured in millisecond time units.

The theoretical idea behind reducing the performance latency with parallel programming relied on the fact that we considered breaking the task, to be completed by a single thread sequentially within a certain time T, into multiple independent tasks. Then these subtasks were scheduled to run in parallel to each other in different threads.
Theoretically, what we wanted to achieve is the latency of T/N, where N is the number of subtasks. Therefore, the theoretical reduction of lacency by N had to be considered as the performance improvement factor. As shown in the figure 1 below.
The theoretical expectations were, on a general purpose computer, if N is the number of cores, and if nothing else of significance has to be run on the same computer, we expected the operating system to schedule every task on different cores, utilising its hardware to the best it can to provide optimal performance.

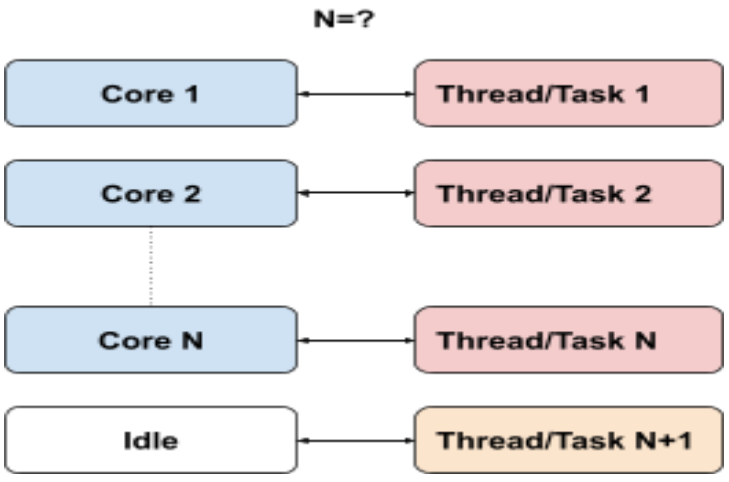

Figure 1: Operating system to schedule every task on different cores

It should be noted that the number of threads (number of cores) is optimal only if all threads are runnable and can run without interruption (no IO/ blocking calls /sleep, etc.). In reality this is rarely the case so we will be expecting the result to be rarely optimal but we can be close to that.

Our method of testing relied on the assumption that nothing else is running on the testing computers that can consume a lot of CPU. Which is almost never the case but in case of a dedicated server for our application, then the other processes and the operating system itself should have almost a negligible impact compared to our application threads. Again we will never achieve the optimal utilisation but we can get close to that.

Another note to consider is that most computers today use hyperthreading (virtual cores vs physical cores). It means that a single physical core can run two threads at the same time. That is achieved by having some hardware units in a physical core duplicated so the two threads can run in parallel and some hardware units shared. So we can never run all threads a hundred percent in parallel to each other but we can get close to that.

Cost of multithreading was another fact to consider in the methodology of testing. If the task had easily been broken into multiple, we had to accept some inherent costs associated with the procedure. It was going to be the cost of doing some calculations in breaking the task into smaller segments, then the cost of creating the threads, passing the tasks to them and starting them, then there is the time that it takes for the threads to actually be scheduled by the operating system to start running.
Not all the sub tasks may be the same length and not all of them start at the same time. Also again, we have the time until the aggregating thread gets the signal and starts running again. Lastly, aggregating the subtasks into a single artefact also has some cost. This is illustrated in the figure 2 below:

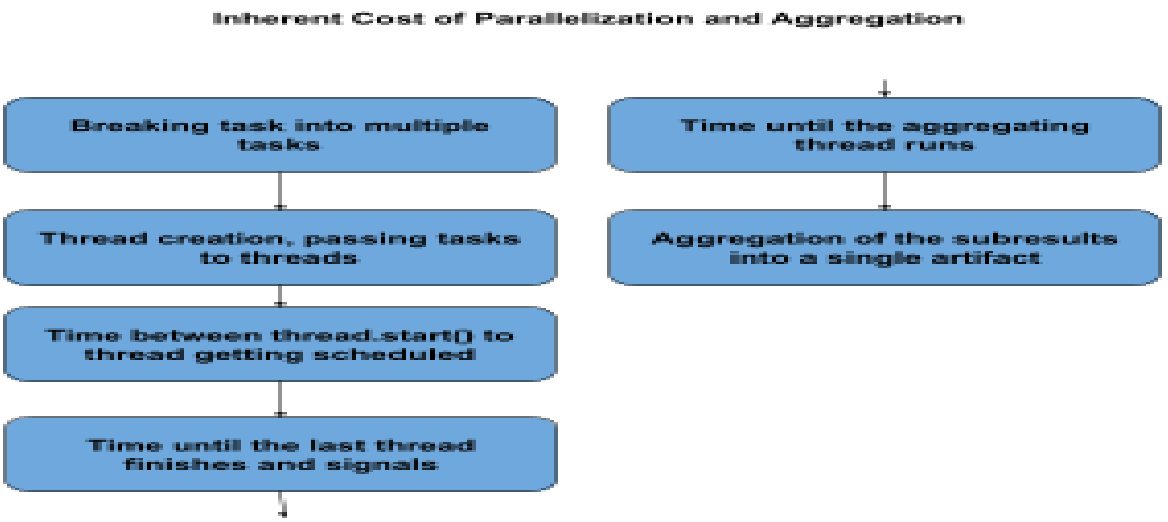

Figure 2: Cost of Parallelization and Aggregation

III.2. Optimising the Fork/Join parallel code

The fork() method, from the Fork-Join framework, had to be implemented to asynchronously execute the given task of image processing from the mean and median multithreaded filters. With this method, using the RecursiveTask<> class, we computed the filtering image processing by stating that if the task is too large, that is if we are dealing with more than one thread, then we split it and execute it in parallel manner. In case of one thread to be processed then, no need to implement parallelization.

So if we plot the theoretical latency of our solution versus the original taks latency (see graph below in figure 3), we can see the point where we need to decide if the task is worth parallelizing, is the intersection of the multithreaded solution and the single threaded solution. The key takeaway from this graph is that small and trivial tasks are just not worth breaking and running in parallel.

Original Task Latency (ms)	Single threaded (ms)	Multithreaded solution
0	0	12.5
25	25	17.5
50	50	25
75	75	30
100	100	37.5

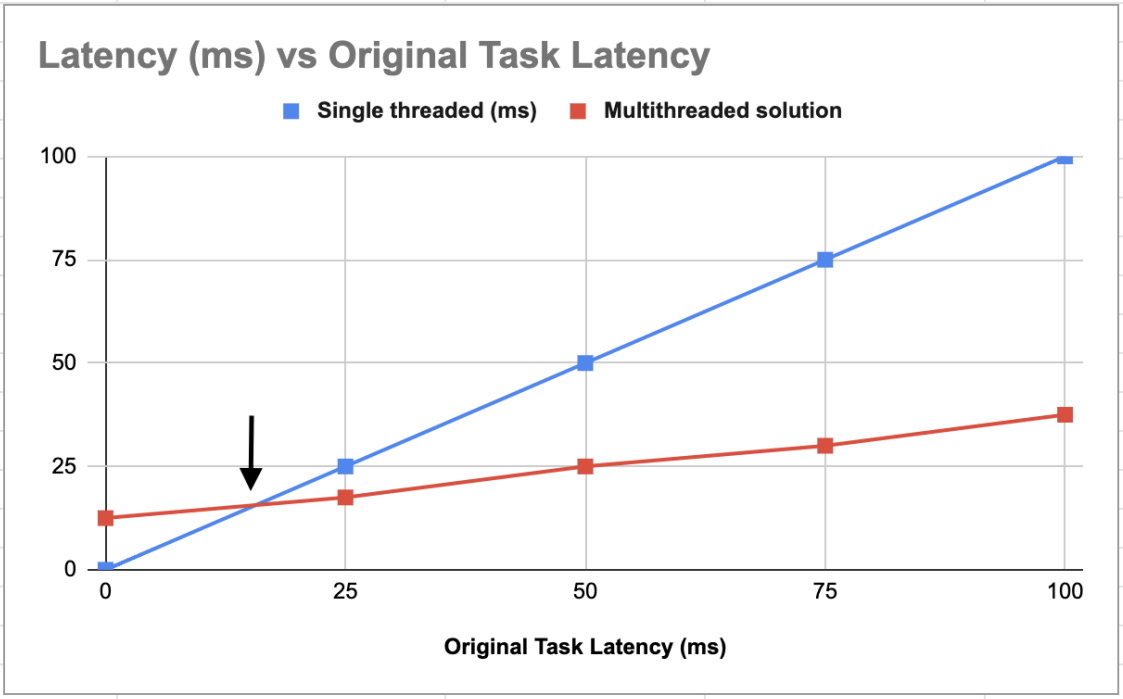

Figure 3: Serial cutoff for both the mean and the median filter programs

IV. Testing and Results

IV.1. Benchmark of Algorithms

Testing was executed by comparing the two solutions (serial and parallel implementation) for both the mean and the median filters.

In the main method of each class, we captured the timestamp of the start of the image processing operation and stored it in a startTime type long variable. Then similarly we captured the end of the image processing operation, and stored it in an endTime variable. Then we calculated the duration and printed it out to the user

After running the single threaded mean and median filter first, we noticed that it took us about 1577 and 3283 milliseconds respectively to process a 3036 x 4048 pixels image by our single main thread.

Then after switching to the mean and median multithreaded filter implementation and running it with a single thread, it actually took us a little longer. That is about 1750 and 3613 milliseconds respectively. This was expected as we just added more overhead of creating a new thread and did not get any benefit out of it.

Then when ramping the mean and median multithreaded filter implementation to two threads, We noticed that it took us almost half the time, about 926 and 1187 milliseconds respectively and the result image looked exactly as expected.

And increasing the number of threads to six, the time it took to complete the processing was even smaller.

So we could clearly see a big performance improvement in a form of shorter latency.

We went ahead and ran the program many times for each number of threads on a six cores MacBook Pro, which has 1 processor and a 2.6GHz processor speed. The same procedure happened on a windows machine server with 4 logical processors and 2 cores. See the two machines specs in figure 4 and 5 below:

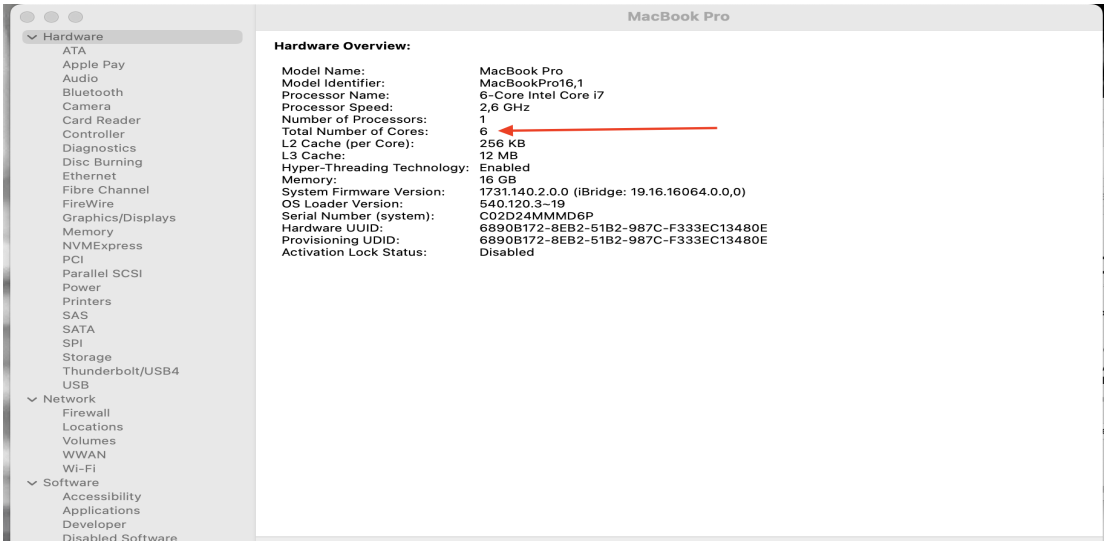

Figure 4: MacBook Pro hardware details

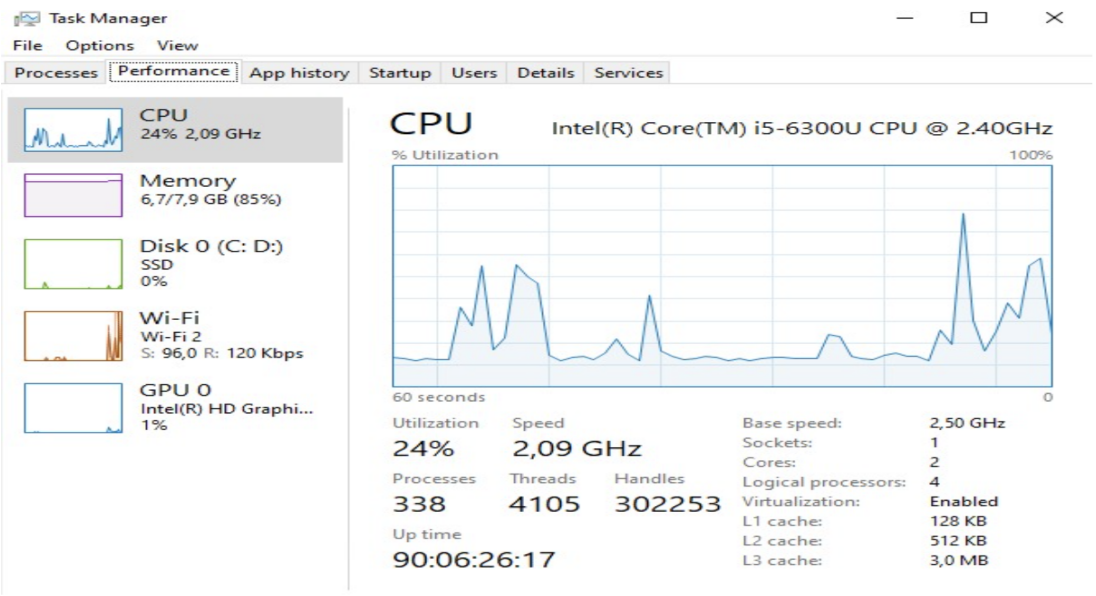

Figure 5: Windows machine hardware details

IV.1.1. Test on a 6 cores MacBook OS machine

After running our tests on the 6 cores MacBook machine, we noticed that, as we increased the number of threads towards the number of physical cores, the latency decreased significantly. As we keep increasing the number of threads towards the number of virtual cores, we could see a slow down in latency improvement and somewhere in the middle of it, we could see it becoming counterproductive . That's because both pairs of cores share some resources among themselves and also because the computer we were running the benchmark on, was not fully dedicated for our application. So there were still some CPU resources consumed by other processes and other background applications.

And as we increased the number of threads beyond the number of virtual cores, we saw no benefits at all. Just as it was expected:

Mean Filter Parallel	
Number of Threads	Latency (ms)
2	926
3	700
4	675
6	660
8	671
9	670
10	675
11	675
12	672
13	675
14	674
15	675
16	673

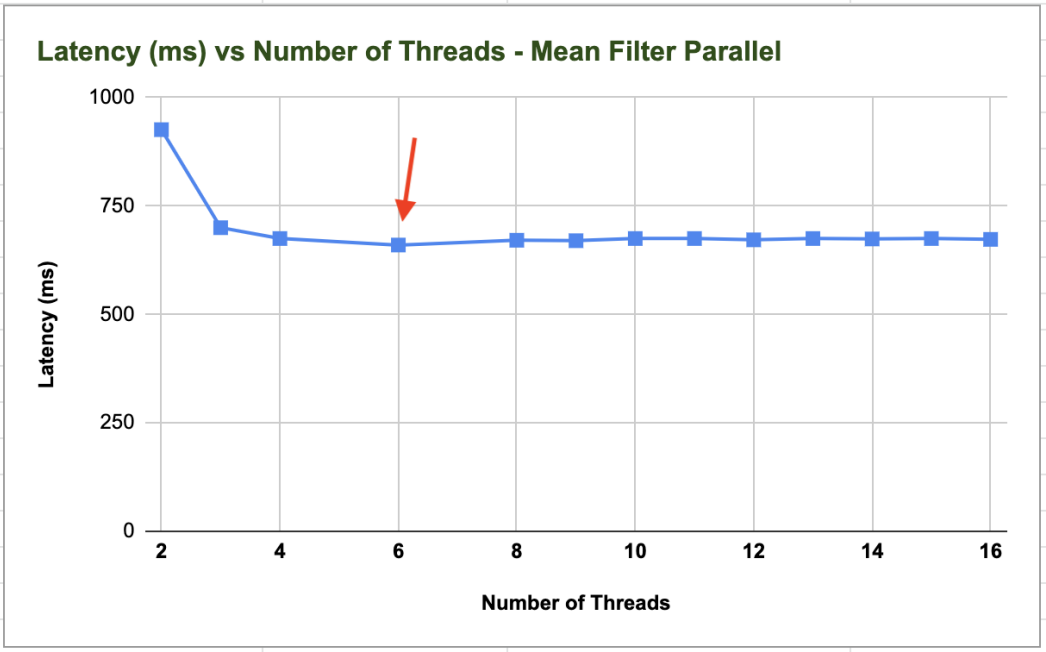

Figure 6: Performance Analysis Mean filter Parallel- Latency vs # Threads

Median Filter Parallel	
Number of Threads	Latency (ms)
2	1187
3	906
4	775
6	751
8	779
9	767
10	756
11	749
12	758
13	754
14	771
15	764
16	769

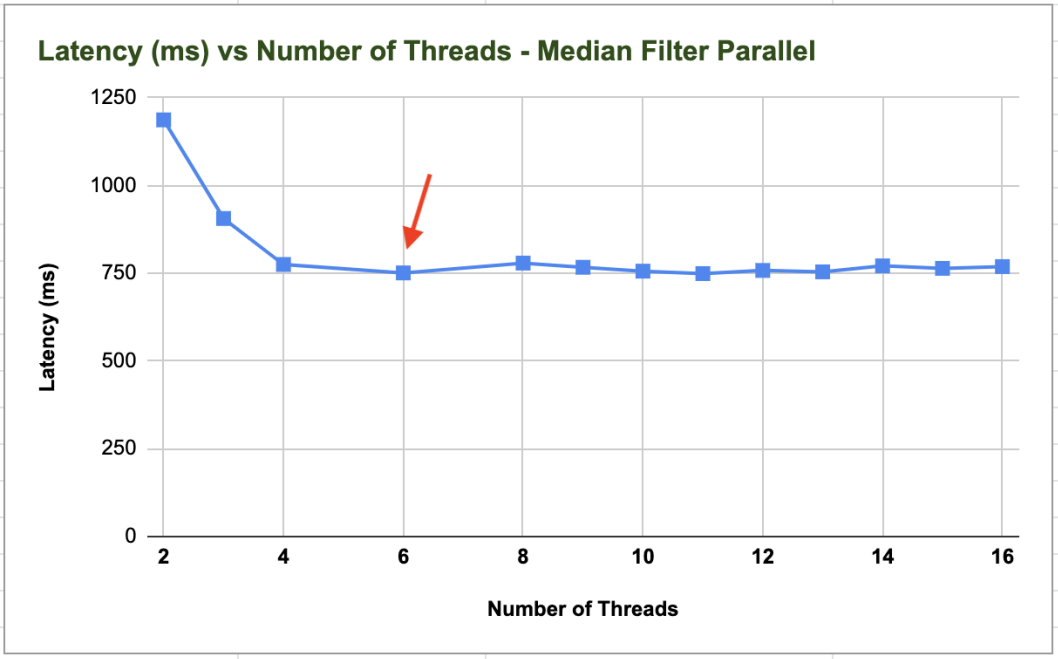

Figure 7: Performance Analysis Median filter Parallel - Latency vs # Threads

To compare how much speedup we get from partitioning the image and processing it by 6 versus 1 thread, we resized the original image into smaller resolutions and ran the application on each of those images. We could see that as the source image got smaller in resolution, in other words, less pixels to process, we started getting less and less benefit from processing that image by multiple threads. In fact for images smaller than 1138 x1517 pixels, we started navigating into negative speedup as illustrated in figure 8 and 9 below. In this case the multithreaded solution became slower than the sequential and that's due to the cost we mentioned above. We pay for running an algorithm in parallel.

Speedup - Mean Filter Parallel			
Different Resolutions (pixels)	1 thread	6 threads	Speedup
3036x4048	1586	1447	139
1518x2024	184	119	65
1138x1517	46	56	-10
759x1012	44	53	-9
379x505	10	18	-8
189x252	9	15	-6

Different Resolutions (pixels)	Speedup of 6 vs 1 thread
3036x4048	139
1518x2024	65
1138x1517	-10
759x1012	-9
379x505	-8
189x252	-6

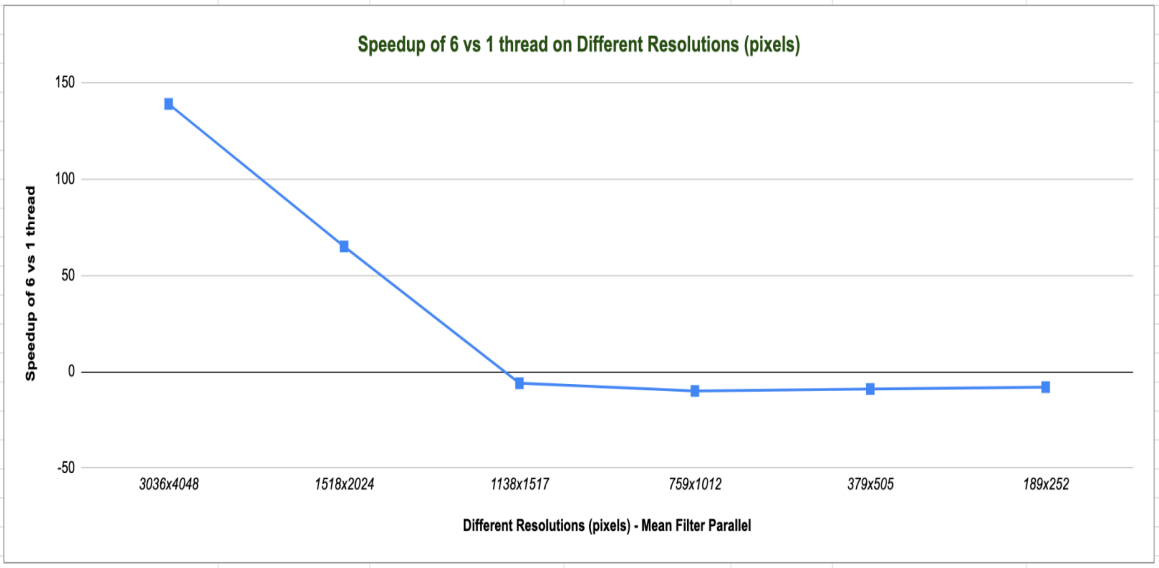

Figure 8: Speedup Mean Serial vs Mean Parallel for different screen resolutions

Speedup - Median Filter Parallel			
Different Resolutions (pixels)	1 thread	6 threads	Speedup
3036x4048	3131	1187	1944
1518x2024	309	148	161
1138x1517	88	87	1
759x1012	19	20	-1
379x505	16	18	-2
189x252	10	14	-4

Different Resolutions (pixels)	Speedup of 6 vs 1 thread
3036x4048	1944
1518x2024	161
1138x1517	1
759x1012	-1
379x505	-2
189x252	-4

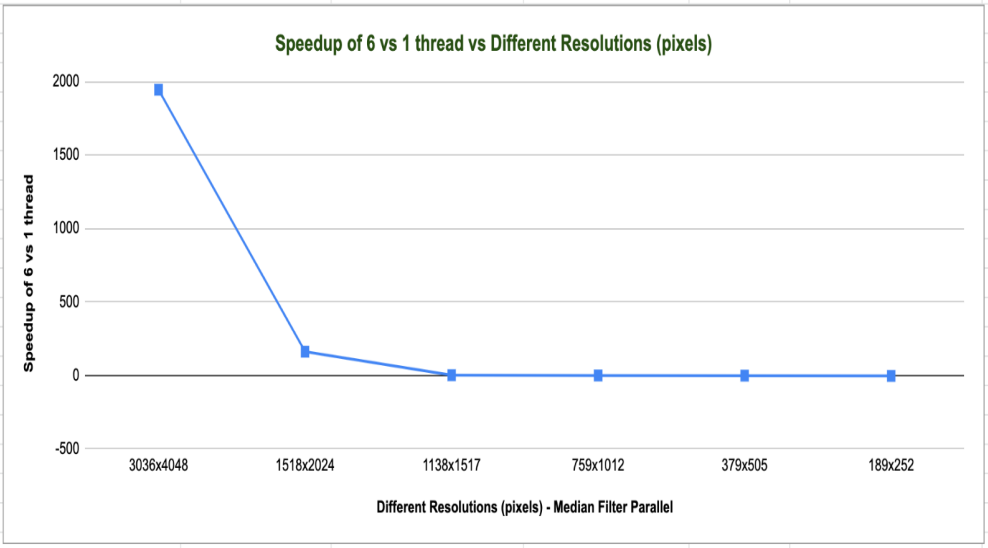

Figure 9: Speedup Median Serial vs Median Parallel for different screen resolutions

IV.1.2. Test on a 2 cores Windows OS machine

The same above process was executed on a 4 logical processors and 2 cores windows machine and we noticed that the latency significantly decreased to almost a third for the same number of threads processed on a 1 processor machine.

After running the single threaded mean and median filter first, we noticed that it took us about 518 and 909 milliseconds respectively to process a 3036 x 4048 pixels image by our single main thread.

Then when testing the mean and median multithreaded filter implementation with two threads, We noticed that it took us slightly more time than the single threaded case for the mean filter and slightly less time for the median filter , about 715 and 724 milliseconds respectively. Though the image looked exactly as expected, the multithreaded mean filter time increase wasn't expected. As mentioned in the theoretical part, this could be explained by the fact that most computers today use hyperthreading (virtual cores vs physical

cores). Meaning that a single physical core can run two threads at the same time. That is achieved by having some hardware units in a physical core duplicated so the two threads can run in parallel and some hardware units shared.

And as we increased the number of threads beyond the number of virtual cores, we saw no benefits at all. This is illustrated in figure 10 and 11 below.

Mean Filter Parallel	
Number of Threads	**Latency (ms)**
2	715
3	608
4	603
6	599

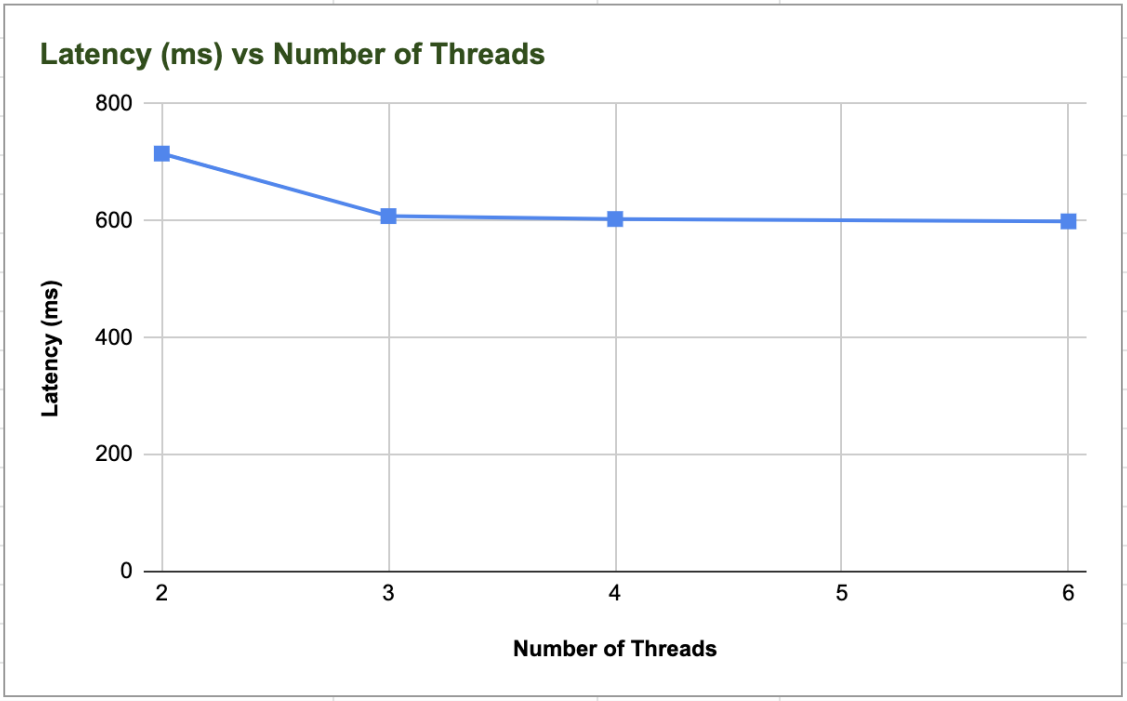

Figure 10: Performance Analysis Mean filter Parallel- Latency vs # Threads

Median Filter Parallel	
Number of Threads	Latency (ms)
2	724
3	613
4	604
6	611

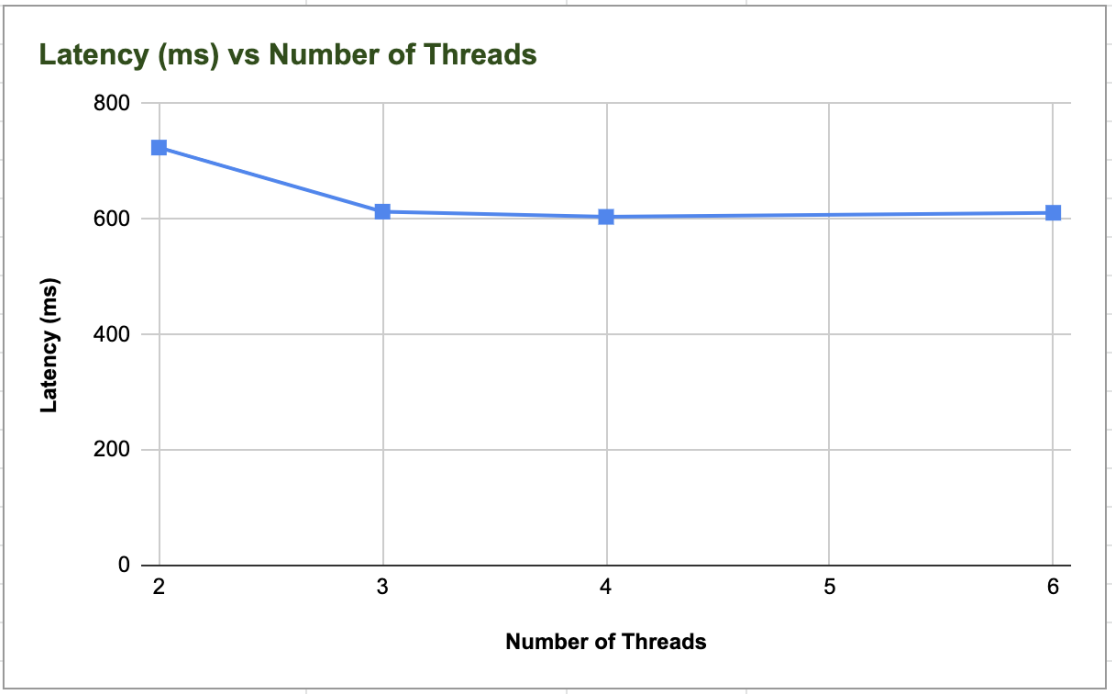

Figure 11: Performance Analysis Mean filter Parallel- Latency vs # Threads

V. Conclusion

Throughout this assignment and after all the implementations and testings, we conclude that concurrent programs have their own limitations. We can also conclude that parallel programming is not always faster; therefore choosing the right algorithm for a given problem remains the key to achieve better results within a reasonable time.

It is true that we can get a speed up if we partition a problem into multiple sub-problems. However, more threads than cores is counterproductive, we proved that during our testing on the 4 logical processors and 2 cores windows machine where the multi threaded implementation using two threads took more time to process the task than the single threaded implementation. Therefore, it is safe to say, there is an inherent cost for running an algorithm by multiple threads.The problem or task to be partitioned needs to be large enough to make the effort worthwhile.

We did get some useful information from the internet and this project took us a long time since it was quite a huge one, the most difficult part was determining and plotting the speedup for different resolutions on the 2 cores windows machine.

Link to the Git repository: *https://github.com/teddy-muba/CSC2002S_Tasks/tree/master*