PCP ASSIGNMENT ONE

AN ANALYSIS AND COMPARISON OF
SERIAL AND PARALLEL MEDIAN FILTERS

COMPLETED BY

DAVID PULLINGER

The University of Cape Town

CSC2002S
23 August 2021

TABLE OF CONTENTS

Introduction[1]

The aim of this project was twofold. Firstly, I had to create **correct** serial and parallel versions of a median filter, which is a digital filtering technique which removes noise (corrupted, or distorted data) from a dataset. Secondly, I had to compare the performance of the serial and parallel versions across a range of input sizes and filter widths for the median filter.

The parallel algorithm I used was the Fork/Join algorithm. This is expected to dramatically speedup the application, as the workload is being split into multiple smaller workloads which are able to run in parallel. However, the overhead from creating these threads could end up being slower than the potential speedup that splitting the workload would achieve. The limits of when parallel processing should stop, and serial processing should begin is investigated in this report.

Methods

Parallelization

The serial median filter simply iterates through an array of data and at any index, say `i`, it calculates the median of all elements in the range `i` to `(i+filterWidth)`. This process is repeated for the entire array of data. The parallelization was implemented by splitting up the iteration of the array, so that each thread would be running the serial median filter on a smaller subarray (Fork/Join approach). For example, if the array was 1 million elements long and we wanted the median filter to run on 4 threads, the 1st thread would run the median filter for indices 0 to 250 000, the 2nd thread for indices 250 000 to 500 000, etc.

Validation

<u>Serial median filter</u>
To validate the serial median filter, I had to test how well it had removed corrupted or distorted data. i.e., how well it had removed outliers. I tested this by calculating the variance of the data before and after a median filter had been applied. I tested this on all the data files (ranging from 100 to 1e7 data items), and I used filters of small, medium and large widths (7, 15 and 21 respectively). The results of these tests are given in the Results section below.

<u>Parallel median filter</u>

[1] Quick Note: For all data files referred to in the report, please refer to the README in the root directory of this project, which lays out where everything is.

To test the functionality of the median filter, I used the serial median filter application. Therefore, I was able to be certain that it worked as expected. So, in order to ensure that the parallel median filter was also working, I could simply run the Unix `diff` command, and pass it the output of the serial and parallel median filters; thus, ensuring that there was no difference in the output that the applications were producing. When I ran the `diff` command, there was no output, implying that the output of the median filters was indeed the same. I had thus validated that the serial and parallel median filters were running as expected.

Timing

Design

To compare the performance of the serial and parallel median filters, I timed the `serialMedianFilter()` and `parallelMedianFilter()` functions using `System.nanoTime()`, using a range of input and filter widths.

When timing the serial median filter, I tested it with input sizes which ranged from 100 to 1e7 in factors of 10. For each input size, I would run the median filter with 3 filter widths: 7, 15 and 21. This was to get broad categories of small, medium and large filter widths. Then, for each filter width, I would run the serialMedianFilter application 30 times, to get a reliable estimate of the application's running time with specific parameters.

When timing the parallel median filter, I essentially followed the same routine as with the serial median filter, except I had an extra loop which would run the parallelMedianFilter application with a different number of threads. So, for a given file size and a given filter width, the application would be run 150 times, 30 times for 2 threads, 30 times for 4 threads, and so on. I tested the application with 2,4,8,12 and 16 threads.

The results of these timings are given in the Results section below.

Scripting

Testing the median filter applications as described above is obviously quite laborious, so I wrote bash scripts (one for both the parallel and serial versions) which would run numerous `make` commands with the various parameters defined by where they were in the loop. I did this for the file sizes ranging from 100 to 100 000.

However, when it came to the 1e6 and 1e7 file sizes, it was not feasible to rerun the entire application, as this would require that the text file be read into an array each time the application was executed. This was feasible (in fact, even necessary. See Caching below) for the file sizes ranging from 100 to 100 000. However, reading in the 1e6 and 1e7 file sizes alone took about 1 minute and 5 minutes respectively. Therefore, I reran the serial and parallel median filter functions from within the Java application itself. This way, the text file could be read to an array once, and the median filter could be rerun as described in the Design above, with various parameters.

Measuring Speedup

I measured the speedup by simply plotting the fastest parallel median filter time (the one with the fastest thread configuration) against the serial median filter time for a given file size and filter width. Note that these plots had 30 data points as the applications were run 30 times as aforementioned. I then compared the means of the parallel and serial median filters for each file size and filter width configuration that was possible (i.e., 100 and 7, 100 and 15, …, 1e7 and 15, 1e7 and 21). To attain the relative speedup, I simply divided the larger of the two means by the smaller.

My Machine Architecture

Hardware

The hardware specification of the machine I ran the parallel and serial median filters is summarized in the table below:

Device Name	Apple MacBook Pro
Chipset	Apple Silicon M1
Cores (physical)	8 (4 “high-performance” and 4 “high-efficiency”)
Cores (logical)	8 (there is no hyperthreading on the M1 chipset)
L1 Cache size	320 KB (per “performance core”), 192KB (per “efficient core”)
L2 Cache size	12MB (“performance cores”), 4MB (“efficiency cores”)
RAM size	8GB

Software

I edited the code in VSCode and then compiled and ran the application via the terminal. Therefore, any optimizations that could be made by a Java IDE were absent and I could be sure that the applications were compared on equal footing. I used Bash and Python to write scripts which would aid in compiling and visualizing the data.

Interesting results

Caching

When I tried running the parallel and serial median filters from within the Java application itself for the smaller file sizes (100 to 100 000 data items), I occasionally found that the serial and parallel median filters ran in a time of 0 nanoseconds. I did a bit of research and could only conclude that the previous run of the median filter had cached the result, and when I made the exact same function call exactly after the previous one had finished, it simply returned the result of the previous run.

However, I was not interested in comparing which application would cache these results better (I assumed the serial application would, as there would be less memory consumption than the parallel application, which allocates cache to a thread). Therefore, for the smaller file sizes, I opted to rerun the entire application in the hope that such effects would be minimized.

However, for the smallest file size (100 data items), the reading in of the text file was so quick that the serial median filter was **still** able to run in a time of 0 nanoseconds (it was still able to use the cached results from the last time the entire application was run). I did not see the same results for the parallel median filter, possible because of the reason mentioned above.

I did not exclude these results when comparing the serial median filter and parallel median filter. The caching was a clear advantage of using a serial median filter and it would be unfair and inaccurate to exclude it.

Results and Discussion

Validation

The first results I collected were the results of the validation of the serial and parallel median filters. These are given in the table below:

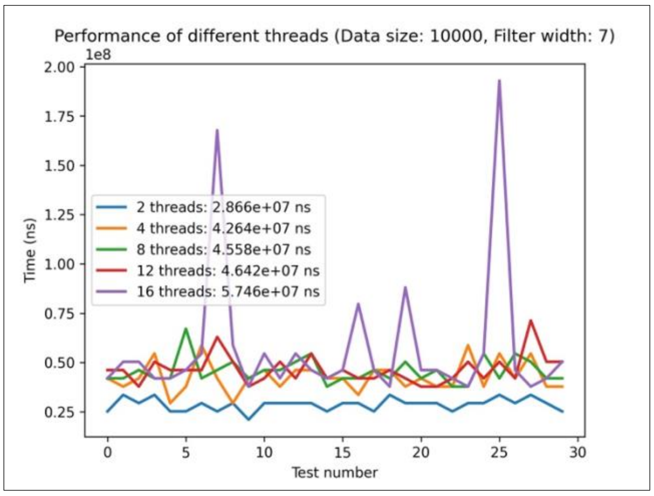
		Variance after applying a median filter of width		
Data size:	Initial variance:	7	15	21
100	2.069	0.917	0.846	0.811
1000	2.047	0.720	0.384	0.301
10000	2.090	0.693	0.365	0.270
100000	2.077	0.694	0.374	0.279
1000000	2.086	0.694	0.367	0.271
10000000	2.083	0.694	0.368	0.272

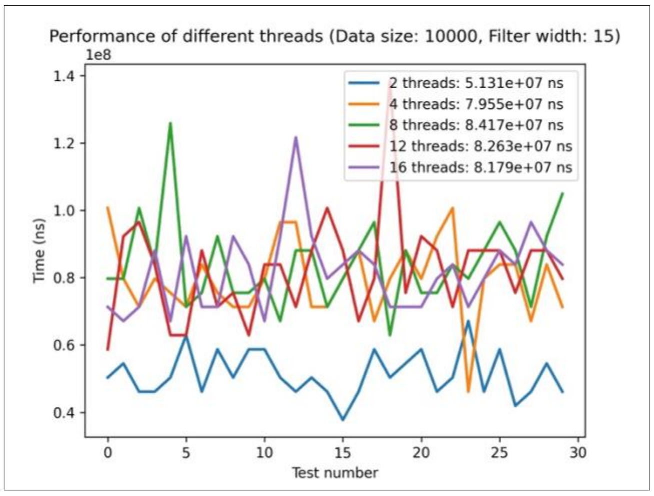
As one can see, the median filter was able to reduce noise (which I have measured as variance) by as much as a factor of 2.090/0.270 = 7.74. This is a dramatic improvement and indicates relatively strongly that the median filters work as expected.

Timing

Note
The times found on the legends of the graphs are the average execution time.

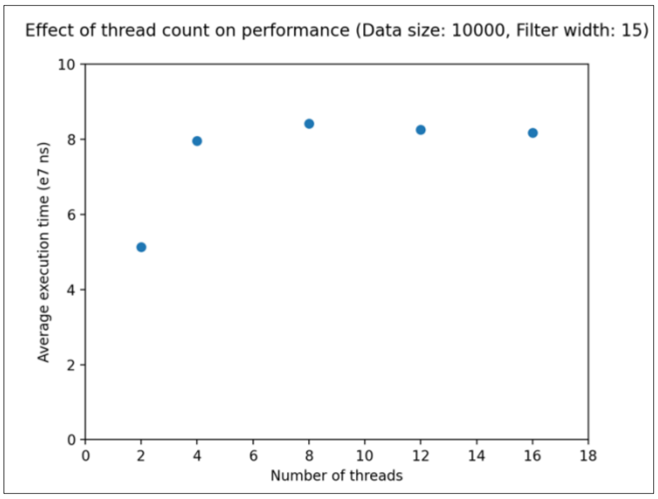
Parallel Threads (Small files)
Secondly, I collected the results from timing. The design of my timing experiments has been outlined above. I plotted graphs of the performance of all the threads for each filter width and file size, such as the ones below (the times on the y-axis are the execution times of the parallel median filter):





I have omitted most of these graphs since I produced 6*3*5 = 90 graphs, and these can easily be found in the projects testing/ folder, as outlined in the README. Also, I did not think it necessary to include all of them, as the two above represent what the graphs of the small data files (100 to 100 000 data items) look like.

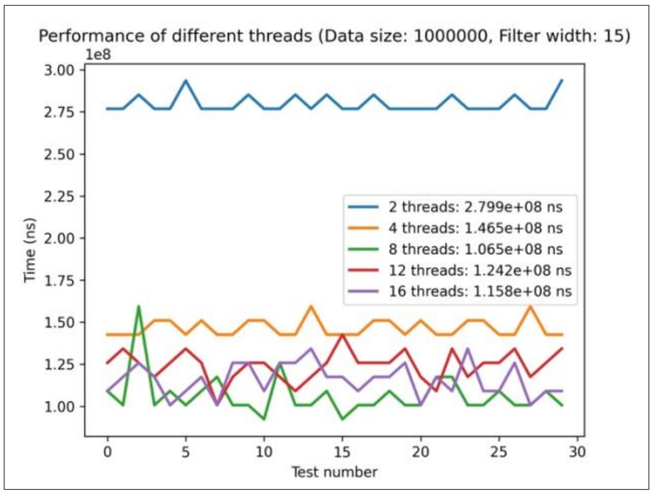
While these graphs may be a bit difficult to understand, they conveyed one fact very clearly (**for small data files**): the parallelization of the problem was only hurting performance. As the number of threads increased, the program became slower, with the running time growing logistically, as can be seen in the speed up graph below:

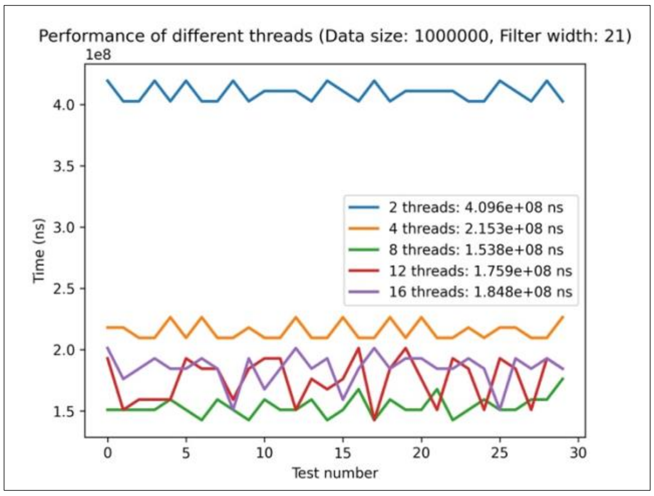


Therefore, it was clear that the best number of threads to use for the small data files would be as little as possible, in this case 2 (i.e., the sequential cutoff for the small text files would be dataLength/2). Ideally, I would use 1, but then there would be no parallelization.

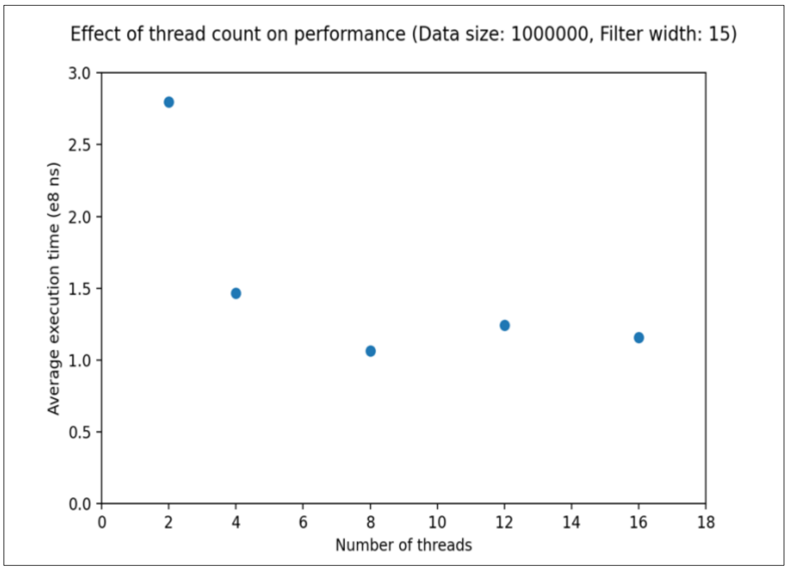
Parallel Threads (Large files)

I tested the large text files (1e6 and 1e7 data items) in a different way to the smaller data files, as described in the experiment design above. Again, I plotted graphs of the performance of all the threads for each filter width and file size, such as the ones below (the times on the y-axis are the execution times of the parallel median filter):





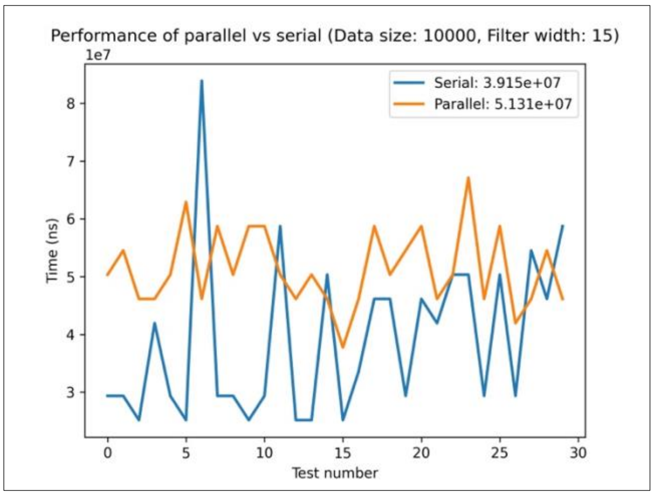
Fortunately, these graphs are a bit easier to understand than the previous ones. Clearly, the threads were now having the opposite (and expected) effect of speeding up the median filter by splitting up the huge workload into smaller, manageable tasks. As the threads increased, so the execution time decreased. The execution time decreased exponentially, as can be seen in the speed up graph below:

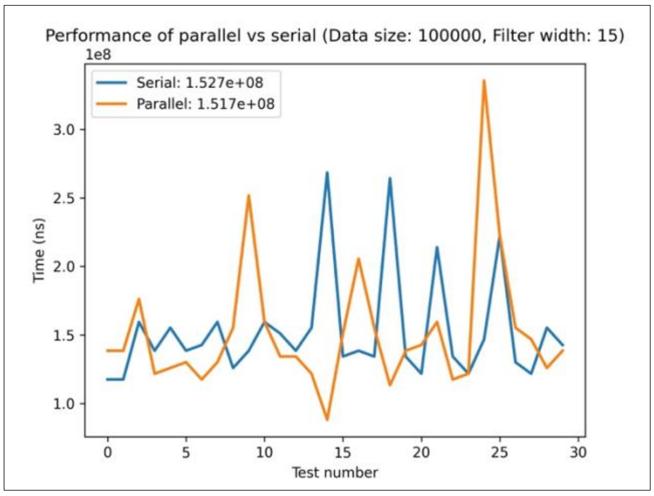


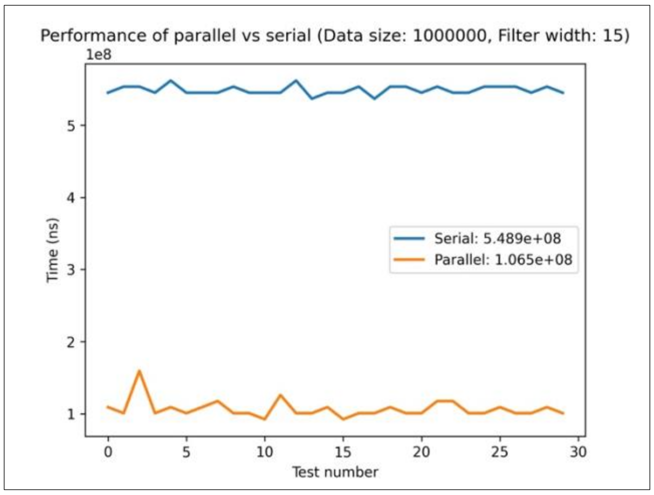
I noticed that, for the large file sizes, the application ran especially quickly when it was running on 8 and 16 threads. In fact, the median filter was fastest for the 1e6 and 1e7 data sizes when it ran on 8 and 16 cores respectively (regardless of the filter width). I believe this may be because my machine has 8 physical cores and running 1 or 2 threads on each core was the most efficient use of the cores. I would have liked to run the parallel median filter on thread counts of other multiples of 8, such as 24 or 32. It is fair to assume then, that the sequential cutoff for the 1e6 and 1e7 data sizes are dataLength/8 and dataLength/16, respectively.

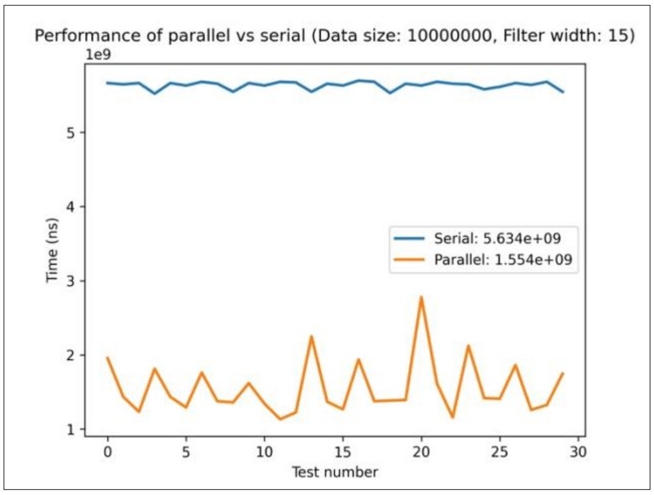
Serial vs. Parallel

When comparing the serial and parallel median filters, I plotted the serial median filters results against the fastest (best configured) of the parallel median filters results. So, for the small file sizes, I compared the 2 threaded parallel program and for the 1e6 and 1e7 file sizes, I compared the 8 and 16 threaded parallel programs, respectively. A selection of these graphs is given below:









As one can see, for the small file sizes, the serial median filter is, as expected, faster than the parallel median filter. However, as the file size increased, the parallel median filter began to close on the serial median filter's performance and by the time the file size was 100 000 and the filter width was 15, the parallel median filter had overtaken the performance of the serial median filter.

This not only indicates that a larger data size favors the parallel median filter but also a larger filter size, as the serial median filter is still faster than the parallel median filter at the 100 000 data size using a filter width of 7.

When the file size was increased to 1e6 and 1e7, the true power of the parallel median filter came to light, and this is abundantly clear in the timing graphs above. The parallel median filter is roughly 5 times faster than its serial counterpart. This is a huge difference, especially if the median filter function had to be regularly used by a user.

Conclusions

Efficacy of median filter

The first and foremost aim of this project was to produce a median filter which helped to reduce noise in 1D data. As can be seen in the results that were compiled, the variance of all the data decreased quite significantly, indicating quite strongly that the median filter was very effective.

I attempted to produce some sort of visualization of the noise removal in the form of a line graph to no avail. There were simply too many data points to produce a decent looking graph. Hopefully the demonstration of the variances is sufficient.

Finetuning of parallel median filter

The second aim of the project was to compare the performance of the parallel and serial median filters. Before I could do this directly, I had to finetune the parallel median filter by experimenting with various numbers of threads to determine which configuration would be best suited for which parameters.

I concluded that for small file sizes (100 000 and below), it would be best to use as few threads as possible, and so I chose to use 2. However, for large file sizes (1e6 and 1e7), it seemed that the more the better, especially if they were a multiple of 8 (this is, however, specific to my machine). So, I settled on 8 threads for the 1e6 file and 16 threads for the 1e7 file.

Comparison of serial and parallel median filter

Finally, the crux of the report. Once I had finetuned the parallel median filter, I was able to do an apples-to-apples comparison between the parallel and serial median filters. The conclusion was clear:

For small file sizes, (10 000 and below) the best option was the serial median filter; the parallel median filter had too much overhead to justify the penalization of the problem.

For medium file sizes (specifically 100 000 in this case), the best option depended on the filter width. For smaller filter widths, the serial median filter was quicker while for medium and larger filter widths, the parallel median filter was quicker. This is because when medium and large filter widths are used, the actual median computation is longer since more data needs to be sorted. If a thread is doing more work, it is “worth” the overhead, and so we saw the slight advantage that the parallel median filter had for larger filter widths.

For large file sizes (1e6 and higher), the best option was by far the parallel median filter. The serial median filter’s performance did not compare with that of the parallel median filter. It is fair to conclude that implementing this application in parallel is definitely worth it.