ELNRAE001

Introduction

The median filter is a nonlinear digital filtering technique, often used to remove noise. This report aims to show how parallel programming can be a useful and effective tool when dealing with large data. What our programme should to is, take in an array and through the process of median filtering produce an output array that contains the medians of a given subset on the original array. This can be easily done sequentially, but due to the nature of the modern laptop can computer, which has multiple cores(and it would be a real waste not to use them) we can create a parallel program that produces the same output as the sequential, but faster.

Methods

A very simple mapping parallel algorithim was used. Given a filtering size, I was able to split the array using a divide and conquer algorithim. Each time the array would go through a sequential test to determine if a parallel split was required. If it was, I would create a new object of type ParallelFiltering and pass parameters into the ParallelFiltering construct to tell the programme where to look in the original array inorder to calculate the median values and calculate where in the median(ouput list) array to put the calculated value. This all was done in the compute() method in the ParallelFiltering class.
ParallelFiltering(float[] array, float[] medianList, int start, int end, int filterSize)
start and end would tell the programe where to look and:
startIndex = start + (int) Math.floor(filterSize / 2);
endIndex = end - (int) Math.floor(filterSize / 2);
would tell the programme where to insert the calculated median value.

The Sequential method first inserted the boundry values into our new array. It went through the array sequentially using the given filter size to produce a new array of length filterSize. This array was then used to get the median value and store it in the medianList.

A compare method: compare() was also created in a Test class to check if the lists produced by the ParallelFiltering class and SequentialFiltering class is the same. The Test class also holds the two median lists and it is where the run time of each algorithim is calculated.

Testing algorithims was created in the driver class to test the time taken, given different filtering sizes. The time taken given a sequential limit and to compare the sequential and parallel codes.

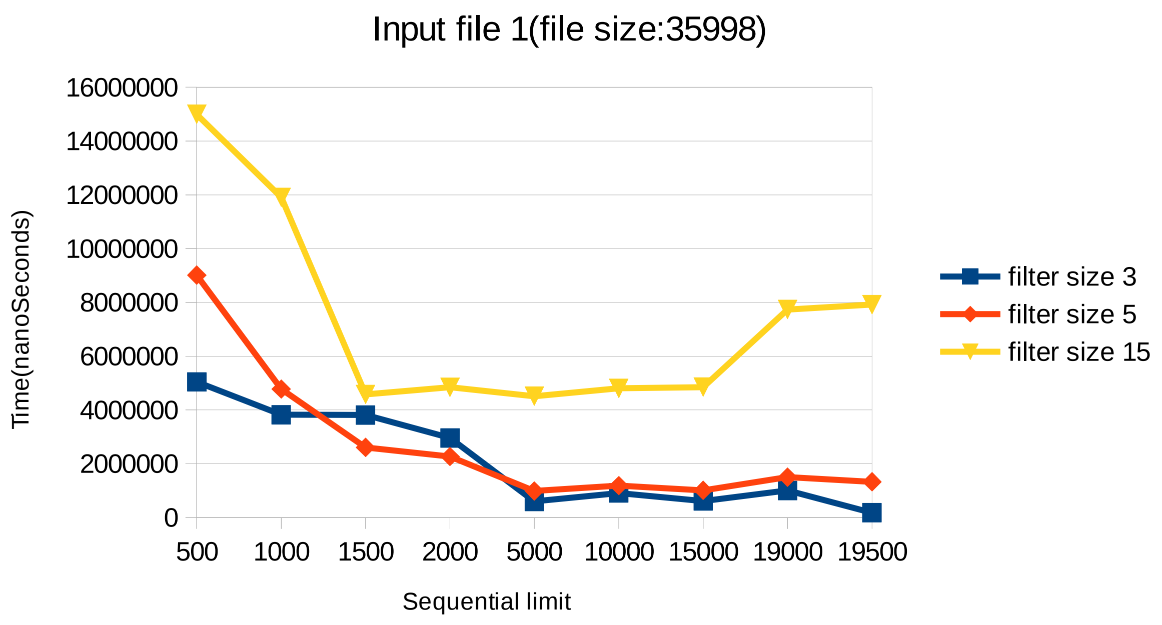
When timing the programme I generated 11 outputs and discarded the first time and averaged the rest to obtain a more accurate time.

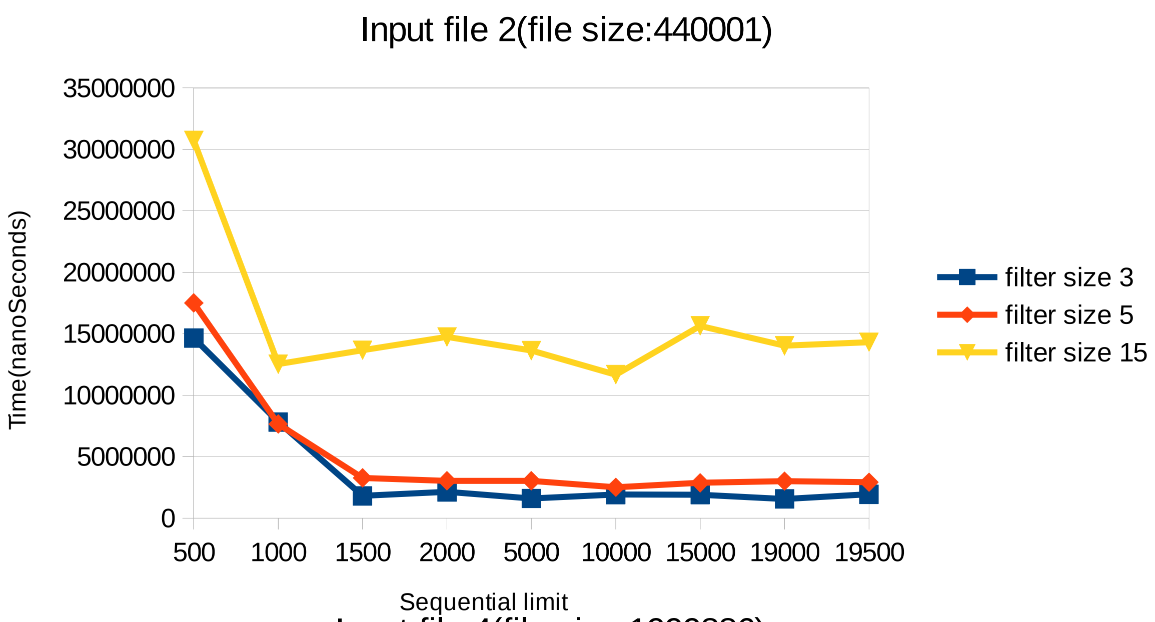
Hardware used

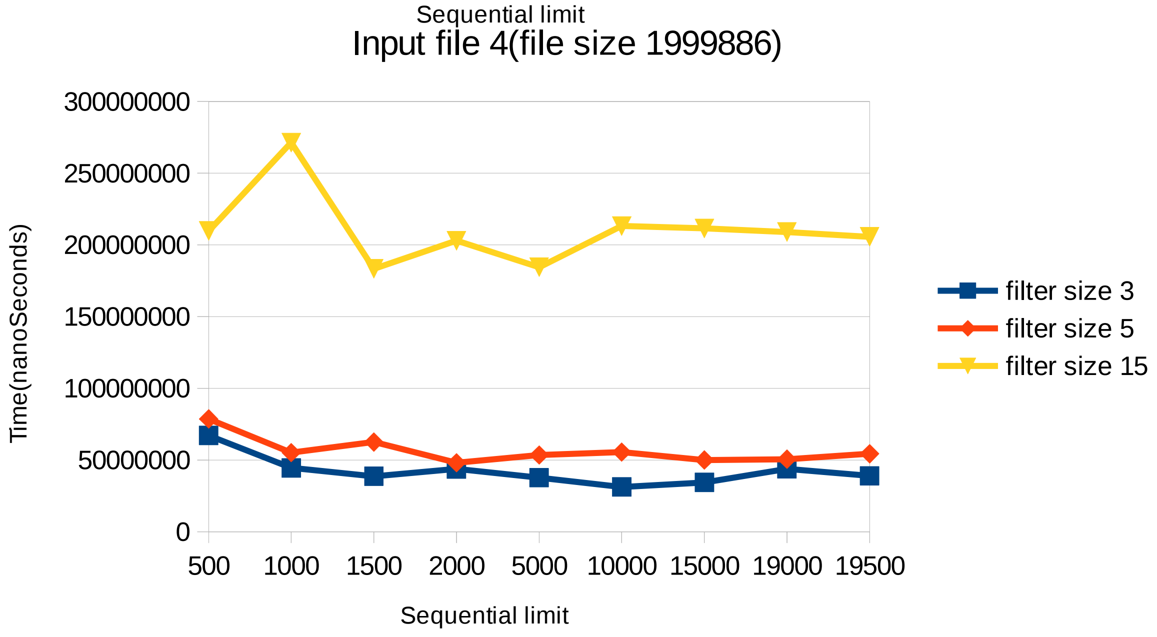
Two laptops was used to conduct the tests:
intel i3 4-core (each 2.3GHz), 4GB RAM, running ubuntu 14.04.
intel celeron 2-core(each 2.13GHz), 2GB RAM, running windows 8.1.

Results

Different sequential limits was tested on the 4-core,i3 pc to obtain an optimal limit for further testing. The sequential limits ranged from 500 to 20000 and were tested using three different file sizes. The graphs for Time vs Sequential limits are show below. Note that I used nano seconds, by using nano seconds I was able to clearly see the change in time given different sequential limits as appossed to using milli seconds.



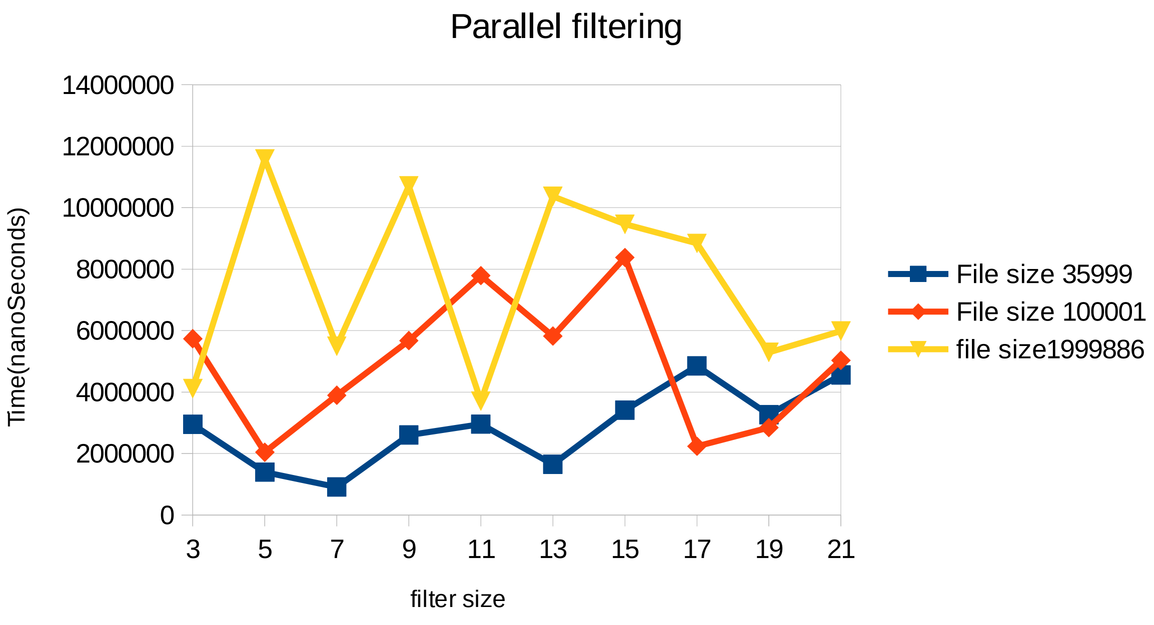


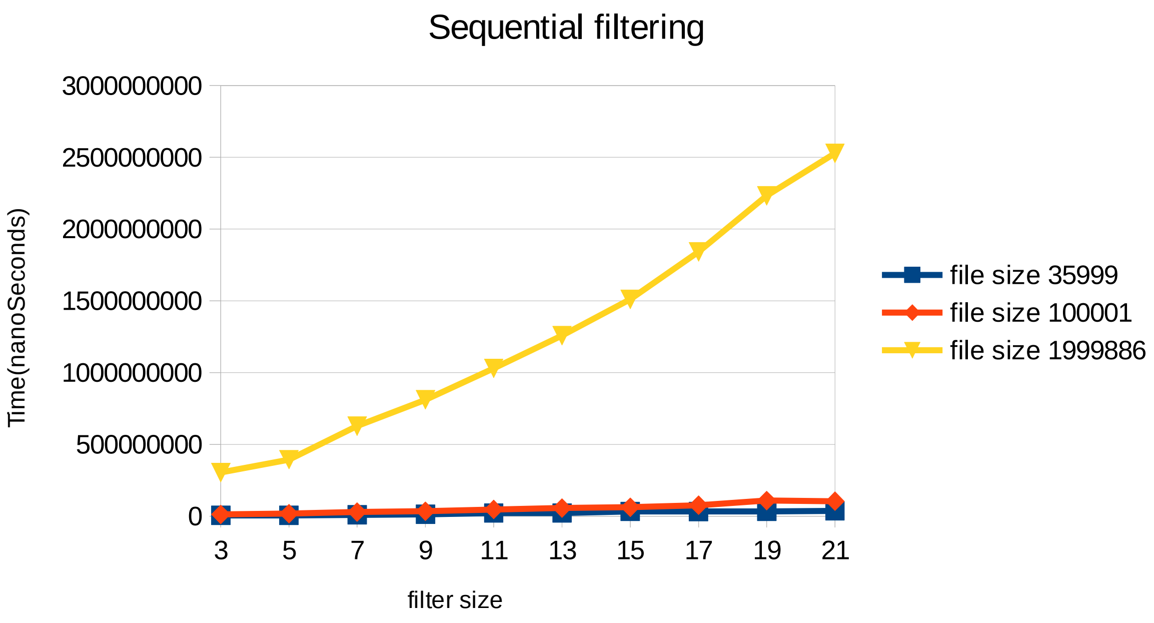


In all three graphs we could see a drop around the 1500 limit given different filter sizes. However for the smaller input size graph the biggest drop occurred at the 5000 limit for filter sizes 5 and 3. This could be due to the size of the file and the low filter sizes. The lower the filter size is the better the sequential programme runs and the higher the sequential limit our parallel code tends to become like our sequential programme. Therefore I will be using a sequential limit of 1500 to test different filter sizes vs run time on the two different machines.

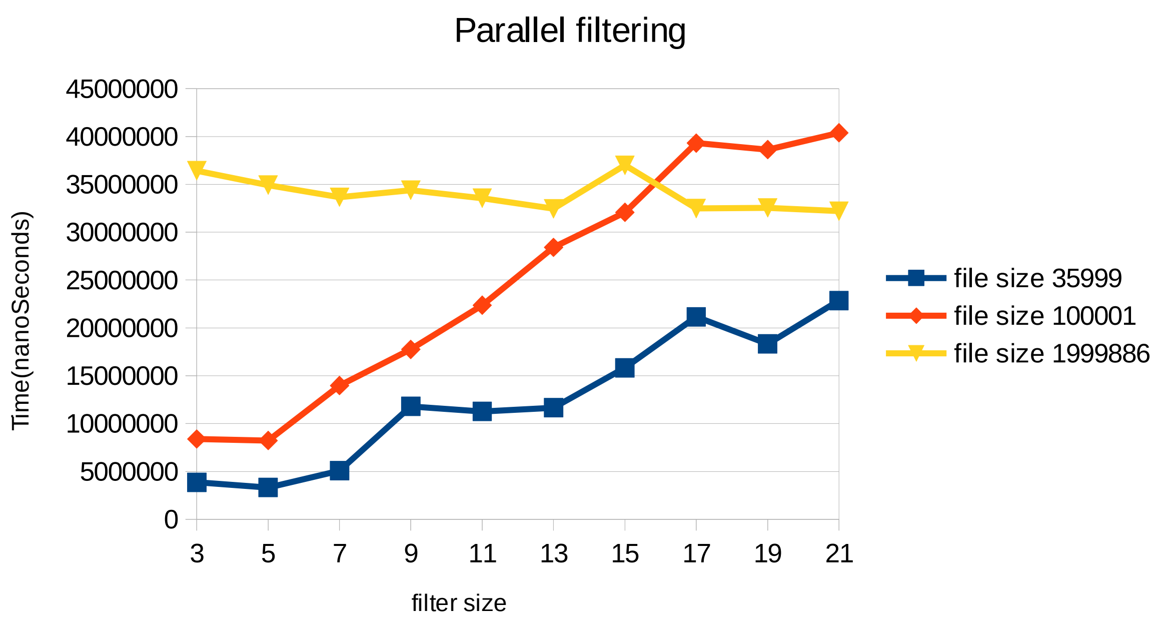
Filter sizes on the i3

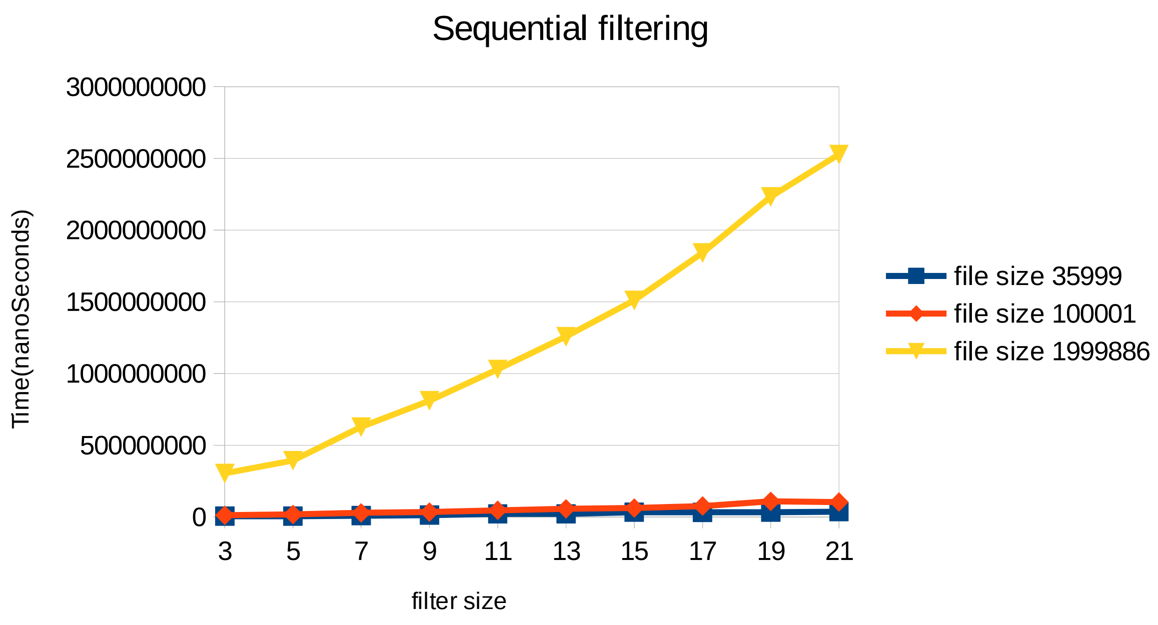
Different filter sizes and file sizes was used to test the performance of parallel filtering and sequential filtering on each machine.





Filters on the intel celeron





(nano seconds was used to analyze the parallel data better)

Conclusion

For sequential filtering the bigger the filter size and the file size the slower the programme runs. In both cases we obtained the same pattern for seqential filtering.

Parallel filtering on the intel celeron laptop tends to follow a linear pattern when dealing with smaller input sizes and higher filter sizes. This is a consequence of only having two processors since we are creating too many threads for two processors to handel. Having a large input size still takes quite some time to filter, but the run time is fairly constant across a range of different filtering sizes.

Parallel filtering on the i3 tends to fluctuate across the range of different filter sizes. However if given a wider range of filtering sizes we would observe that the line of best fit would be constant. Meaning that the average time taken on each filter size would be constant.

The lower the filter size and file size the better the sequential filtering will run.

The parallel filtering programme depends on the pc running it. If you have a fast enough pc (4 cores and above) the filter size should not be an issue, only the sequential limit, which I have determined it to be 1500 for optimal performance and the file size. However if you are running a slower/older pc, a filter size of 3 should be the best solution.

Recommendations

More accurate results could have been obtained by running the programme on faster computers and providing much smaller and much larger file sizes.