CSC2001F
Assignment 1 – Parallelization 1

Nkosingiphile Gumede
GMDNKO003

1.2) Report

1.2.1) Introduction:

This report aims to explore the effects of parallel computing in the java programming language with the built-in java fork/ join framework. This framework will be applied to median filtering a 1D array with variable median sizes, introducing other variables of interest along the way. The fork join algorithm is essentially used to split the work required to be computed into a number of processes. This algorithm had it's foundations rooted in the necessity to run a number of process. Without which, things like operating systems and graphical user interfaces could not exist.

1.2.2) Methods:

A sequential solution will consistently be pitted against a parallel one to derive a speed-up value. This speed-up value equates to the ratio of the times given to compute the sequential and parallel algorithms respectively, given by the equation: time of sequential algorithm divided by time of parallel algorithm. The variables of change include filter size, sequential cut-off and the number of data values in the input file. The number of processors/ cores will also be a variable of interest in understanding the comparison between sequential and parallel algorithms.

The filter size represents the range of values in the input file to which median filtering will be applied. These values have been pre-determined to range between 3 and 21. The values 3, 15 and 21 a have been chosen to represent changes in the filter size.

The sequential cut-off represents the size that the sub-array is required to be to compute sequentially. If this array is greater in length than the sequential cut-off, the original array will split at the mid-point index using the fork-join framework. The sub-arrays will follow the same rules and so on. This implies the the value of the sequential cut-off will represent the number of threads the algorithm will generate. A sequential cut-off value equal to the size of the original/ input array is essentially just the sequential method. This yields only on thread. Any sequential cut-off less than the size of the original array will yield more than one thread, the point at which parallelization occurs. This can be best understood via the following code extract:

```
if ((hi-lo) < SEQUENTIAL_CUTOFF) { //the size of the sub-array is less than the sequential cut-off
        //median filter the array sequentially...
}
else {
        ParallelSolution left = new ParallelSolution(arr,lo,(hi+lo)/2,filterSize);
        ParallelSolution right= new ParallelSolution(arr,(hi+lo)/2,hi,filterSize);
        left.fork();
        right.compute();
        left.join();
 }
```

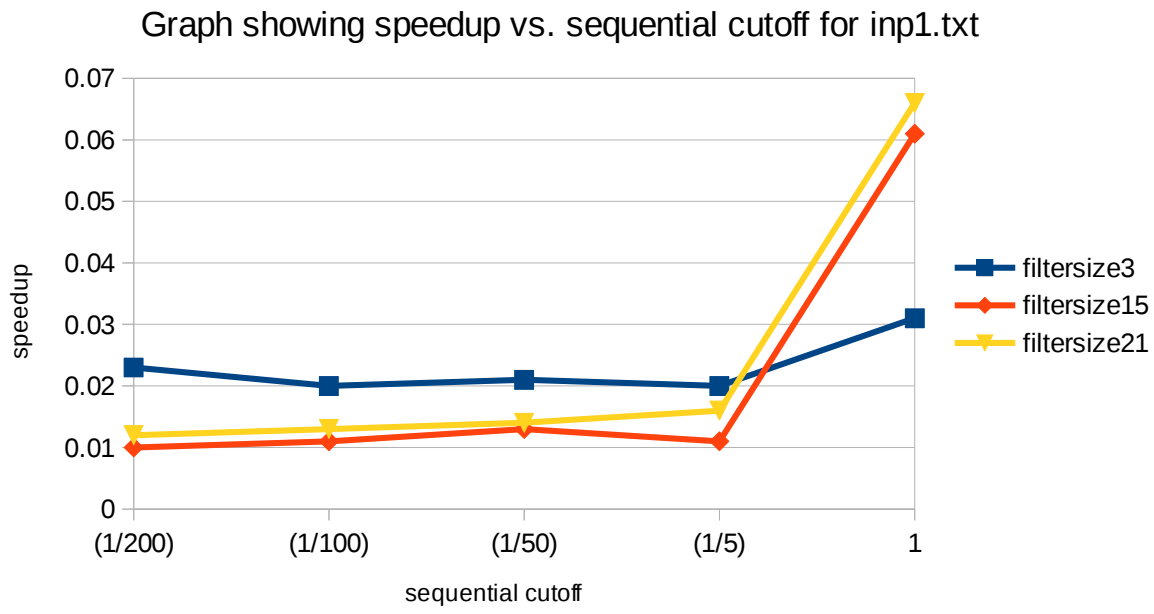
The number of data values to which median filtering will be determined by the input data files. The data files in question have been given as inp1.txt, inp2.txt, inp3.txt and inp4.txt which contain 35999, 100001 440001 and 1999886 data values respectively. These allow us to observe any

observable trend in the times of sequential and parallel algorithms for the given increase in data input.

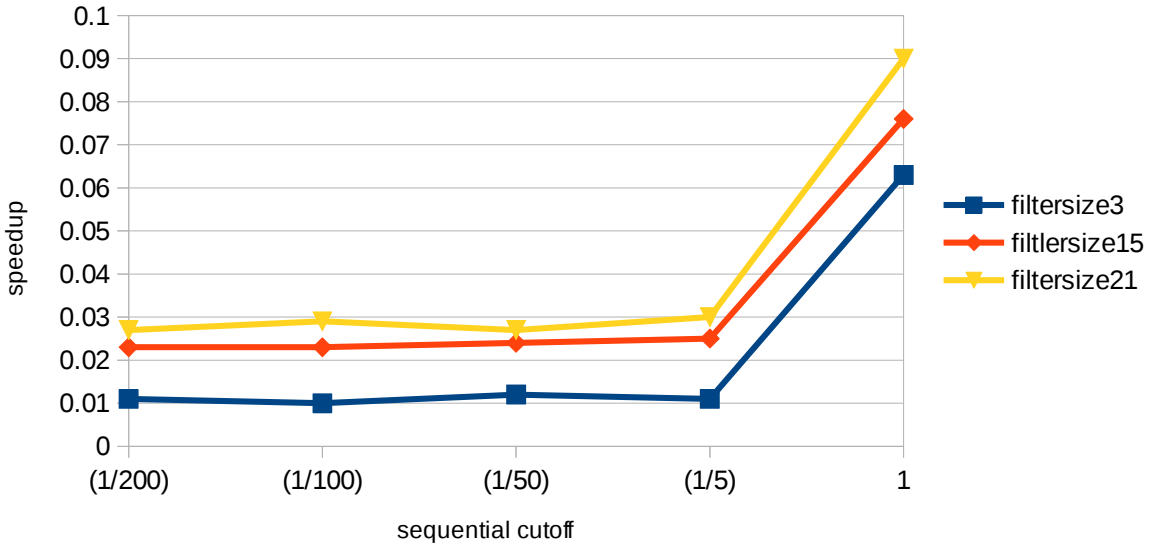
To ensure that our sequential and parallel algorithms work and are correct, one must check to ensure that they both yield the same results for each iteration of median filtering. This must correspond to what we expect when looking at the original values and applying median filtering manually. To do so, an output text file will be supplied. This will supply information on relevant data necessary to ensure the accuracy of our output. It is particularly difficult to manually check for discrepancies in such large data output therefore values for any random index can be juxtaposed to ensure/ assume at least a certain degree of accuracy.

1.2.3) Results and Discussions:

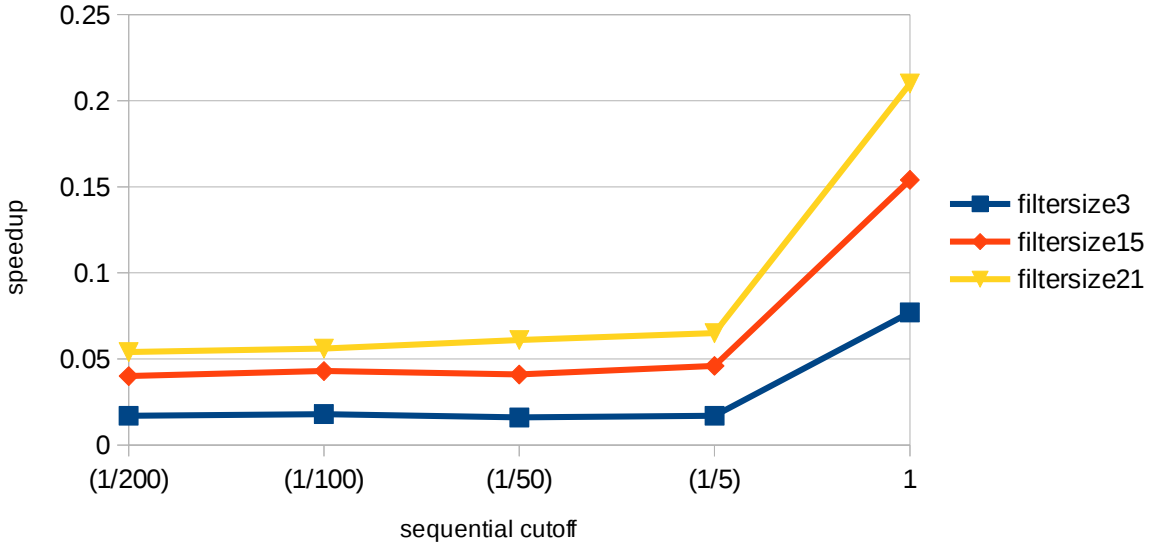
By running the relevant tests numerously over a variation of changing variables, it is possible to establish any observation trends in which variables relate to the output data values we derive. The outputs of a number of relevant relationships have been graphed and discussed below.

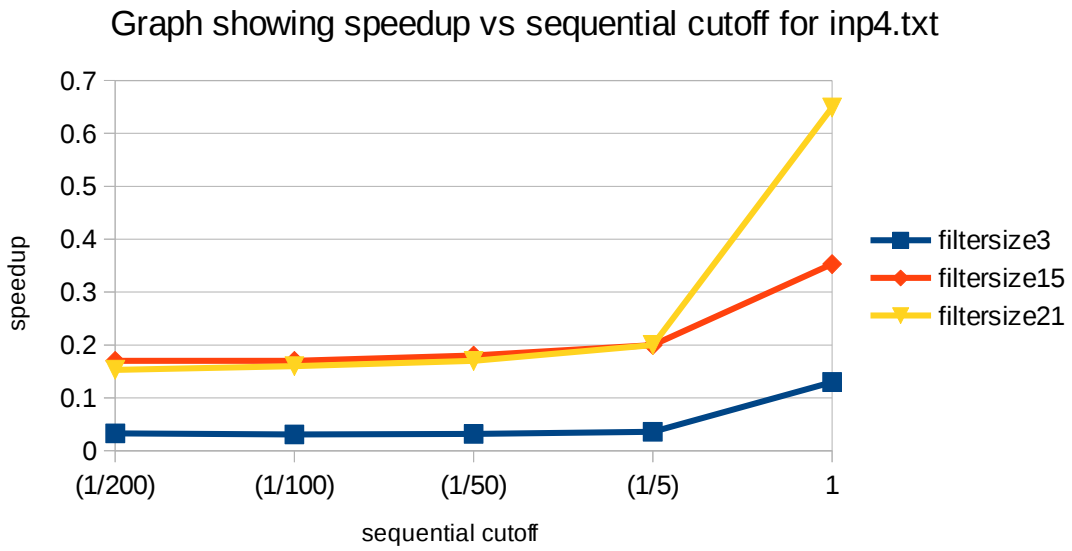


Graph showing speedup vs sequential cutoff for inp2.txt



Graph showing speedup vs sequential cutoff for inp3.txt





As visualized in the graphs above, there is big difference between the speedup times of sequential and parallel algorithms. In these graphs; sequential cut-off is given as a function of the input array on the x-axis. Speedup times are given on the y-axis. There is also an observable relationship between speedup times and filter size. The lower the filter size, the lower the speedup.

Is it worth using parallelization/multithreading to tackle this problem in Java?

Yes, it is clear from the given speed up values (which are all greater than one) than parallelization is always faster than serialization in median filtering. Multi-threading is clearly efficient and well worth implementing to speed up the computation of programs in such cases.

For what range of data set sizes and filter sizes does parallel programming perform well?

Generally speaking, parallel programming performs exceptionally superiorly with smaller filter sizes. Based on the data from the Excel spreadsheet, the data set size has an optimal peak in the region of 400000. This could be due to the fact that either data set sizes do not correlate with speed up or that more variables other than data set sizes offer multi-collinearity and have an influence on the performance of parallelization.

What is the maximum speedup obtainable with your parallel approach?

As given by the data above, the maximum speedup obtainable was 5.7. The were no identifiable trends in the maximum speedup obtainable, therefore from this output data it is not possible to establish an actual maximum given that this variable is so varied. Theoretically speaking, doubling the number of threads should half the time taken to compute the parallel solution.

How close is this speedup to the ideal expected?

The expected speedup can be given by the mean of all speedup values obtained. This expected speedup was given as 3.96. The standard deviation of which was given as 1.4. The maximum has differed from the expected by 1.74. This shows that parallelization has a tendency of varying desired output speeds quite drastically, by almost up to 6 times.

What is an optimal sequential cutoff for this problem?

It is important to note that the optimal sequential cut-off can vary on the dataset size and the median filter size. For the reason, it would be ideal to formulate a general equation for the optimal cut-off according to any correlated variables. For example, the optimal cut-off variable could well be a function of the size of input data or be dependent on the number of processors/ cores your computer has.

From the findings of report as processed from the Excel spreadsheet attached and supplementary findings of running the algorithms on a variation of processors, we can conclude that the optimal sequential cut-off depends on both. Generally, the higher the sequential cut-off, the faster the program runs up to a certain point at which this time starts to diminish depending on the size of input file. This is due to the fact that up to a certain point, it becomes difficult for the computer to efficiently compute a larger number of threads. Also, the number of threads your computer can handle (increased by decreasing the sequential cut-off) depends of the number of processors it has.

1.2.4) Conclusion:

From the above-mentioned finding it is possible to conclude that parallel programming is effective in producing more efficient algorithms which aid in the implementation of relatively lengthy sequential programs and innovation in the field of computer science. The number of processors a computer possesses further enhanced the efficiency of parallel algorithms. This is especially useful with large data sets. Variables such as sequential cut-off, which dictated the number of threads/ processors; and filter sizes have also had an observable effect on the effectiveness of parallelization.