Experimental and Theoretical Speedup Prediction for Serial program and Parallel implemented programs

Mholi Mncube MNCMHO001
MNCMHO001@myuct.ac.za

Introduction

The aim of this experiment is to do a performance between two algorithms such as sequential and parallel. The main function is to filter data set that is one dimension to get a median, such as if you given a set X such that is has elements 2,80,6,3,1, (X = {2, 80, 6, 3, 1}) the filtered median elements in set Y are 2, 6, 6, 3, 1, (Y = {2,6,6,3,1}).

A sequential algorithm is based on operations being done sequentially, self-explanatory. This is done by comparing the data items to find the relative order or output, hence this is achieved by comparing two items at a time, in the case of median filter 3 data items to find the median between them. Parallelizing algorithm or sorting algorithms repeatedly divides the original list into two sub-lists and compares these sub-lists at the same time. These elements in these sub-lists are then merged together at the end to build the sorted output list.

A parallel algorithm is one which can be executed at a time on many different processing devices, and then merged together again at the end to get desired outcome (ComSIS Vol. 10, No. 3, June 2013). Parallel algorithms or programs are expected to produce improved performance on different types of architectures. There should be a speedup if an algorithm or program is paralyzed. We define **speedup** as this ratio: **serial execution time** / **parallel execution time.** This explains how the efficiency or times a parallel program does work faster than the serial program to solve a certain program.

Using Amdahl's Law & parallel processing the theoretical speedup prediction for parallel processing algorithm when estimating speedup vs serial process can be defined again in the following equation:

Let T(N) be the time required to complete the task on N processors. The speedup S(N) is the ratio:

S(n) = T(1)/T(n)

In different instances the time T(1) has, as noted above both serial portion Ts (Time serial) and parallelizable portion Tp(Time parallel). The serial time does not reduce when the parallel part is split up. If one is “optimally” fortunate, the parallel time is decreased by a factor of 1/n .
The expected speedup one can expect is thus **S(n) = T(1)/T(n) = (Ts + Tp) / [Ts +(Tp/N)]**
This elegant expression is expressed as Amdahl’s Law [Amdahl] and hence expressed as an equality. In some cases the best speedup one can achieve is by doing work in parallel, so the read speed up S(n) is less than or equal to this equality.

Methods

In this section we focus on description of our approach in validating algorithm to the solution with details of parallel program.

We will look at the algorithm code below, the compute medthod:

```
        protected ArrayList<Double> compute() {
1.              if((hi - lo)<SEQUENTIAL_CUTOFF){
                        return MedianFilter(x,Filtersize) ;
                }
2.              else{

                        MedianFilterParallel left = new MedianFilterParallel(this.x, this.Filtersize,
this.lo, ((this.lo + this.hi)/2));
                        MedianFilterParallel right = new MedianFilterParallel(this.x, this.Filtersize,
((this.hi + this.lo)/2), this.hi);


                        left.fork() ;

                        ArrayList<Double> leftArr = new ArrayList<Double>();
                        ArrayList<Double> rightArr = new ArrayList<Double>();

                        rightArr = right.compute();
                        leftArr  = left.join()    ;

                        ArrayList<Double> output = new ArrayList<Double>();
                        output.addAll(leftArr);
                        output.addAll(rightArr);
                        return output ;
                }
```

When looking at number one we look at the sequential program which is called when you know it’s sequential algorithm from median filter explained above.

Now when we improve the algorithm for parallelization we look how we split tasks by creating different processes that is splitting the lists using fork/join implementation we have left part of the list and right part both split to get the desired output for the segment of the lists. Then later on join them when we done getting desired output. This is called multi-threading java has a built in methods in its library which allow us to do this. Now to check whether the parallel part of the parallel algorithm we look at the results acquired as a measure of time taken to get results then plot them on graph draw conclusions. We can also use the speedup equation to calculate the speed up in a random run we get that time taken for serial run is 0.168s and for fork/join is 0.18s and filterSize = 3 with samplesize of 10000000 of data.
With that we clearly see speeds are improved by speedup formula S(n) = T(1)/Tp = 0.179/.0.158 = 1.1329. Our desired speedup should be above 2 > to get optimal speedup and conclude algorithm is correct. Also when graphing the graph should show a speedup linear display.

This code is found in the SerialvsParallel class

```
tick();
MedianFilter filter = new MedianFilter(inputs,FilterSize);
float time = toc();
System.out.println("Time taken for typical filter "+ time +" seconds");

tick();
Parallel(inputs,FilterSize);
time = toc();
System.out.println("Time taken for fork/join filter "+ time +" seconds");
```

Here we measure the time each of our different methods yields results. We then sample it by getting average of 5 trial times and record. Then we make conclusions. The programs will have difficulty processing small data with parallel sorting due to the fact that serial has better times hence an analogy was used in class for chef when task was small and having a lot of staff will crowd kitchen making it unproductive. Other problems encountered is some architectures are quite faster than others so at times you think your parallel program works but not at its best.

Results

filterSize	DataSize	Ts	Tp
3	100	0.002	0.004
3	100	0.002	0.003
3	100	0.001	0.003
3	100	0.003	0.003
3	100	0.002	0.005
3	100	0.002	0.003

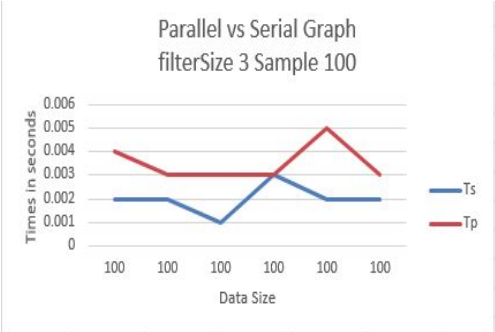
filterSize	DataSize	Ts	Tp
3	1000	0.009	0.007
3	1000	0.009	0.006
3	1000	0.006	0.007
3	1000	0.008	0.007
3	1000	0.008	0.006
3	1000	0.007	0.005

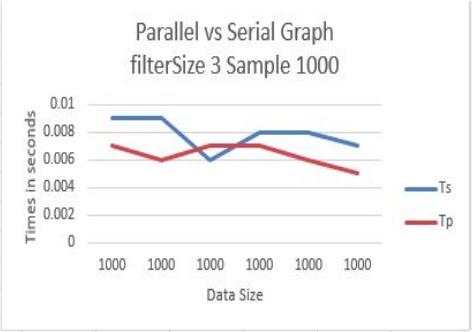
filterSize	DataSize	Ts	Tp
3	100000	0.061	0.05
3	100000	0.072	0.052
3	100000	0.049	0.048
3	100000	0.076	0.045
3	100000	0.06	0.046
3	100000	0.064	0.054

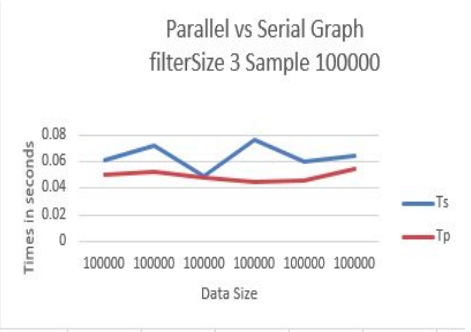
filterSize	DataSize	Ts	Tp
21	100	0.005	0.004
21	100	0.005	0.006
21	100	0.004	0.006
21	100	0.004	0.004
21	100	0.005	0.004
21	100	0.004	0.008

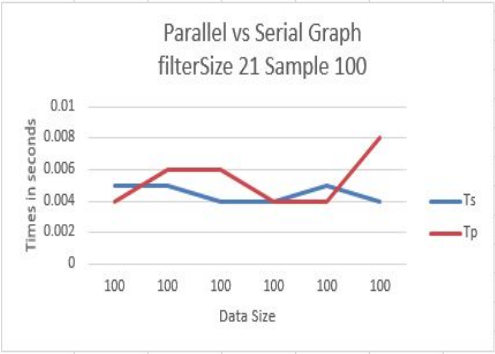
filterSize	DataSize	Ts	Tp
21	1000	0.019	0.01
21	1000	0.018	0.007
21	1000	0.045	0.016
21	1000	0.014	0.009
21	1000	0.02	0.007
21	1000	0.021	0.011

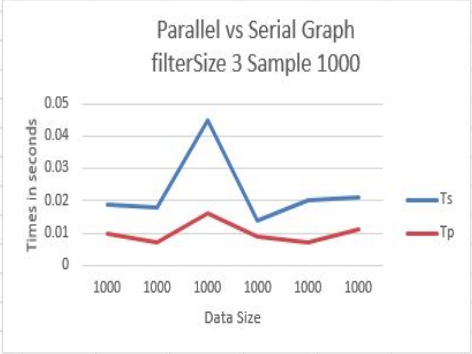
filterSize	DataSize	Ts	Tp
21	100000	0.207	0.147
21	100000	0.208	0.146
21	100000	0.192	0.152
21	100000	0.215	0.143
21	100000	0.223	0.155
21	100000	0.215	0.151

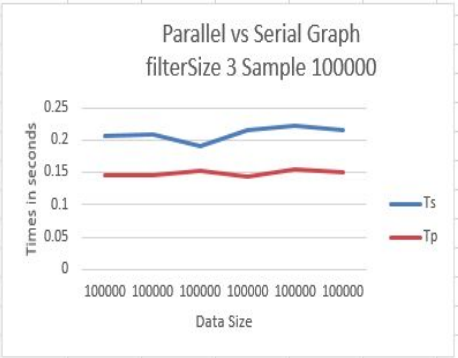












Results captured on an intel i3 processor 2.24GHz
3Gig Ram

DataSize	Ts	Tp
100	0	0,015
100	0	0
100	0	0
100	0,015	0
100	0	0,015
100	0	0

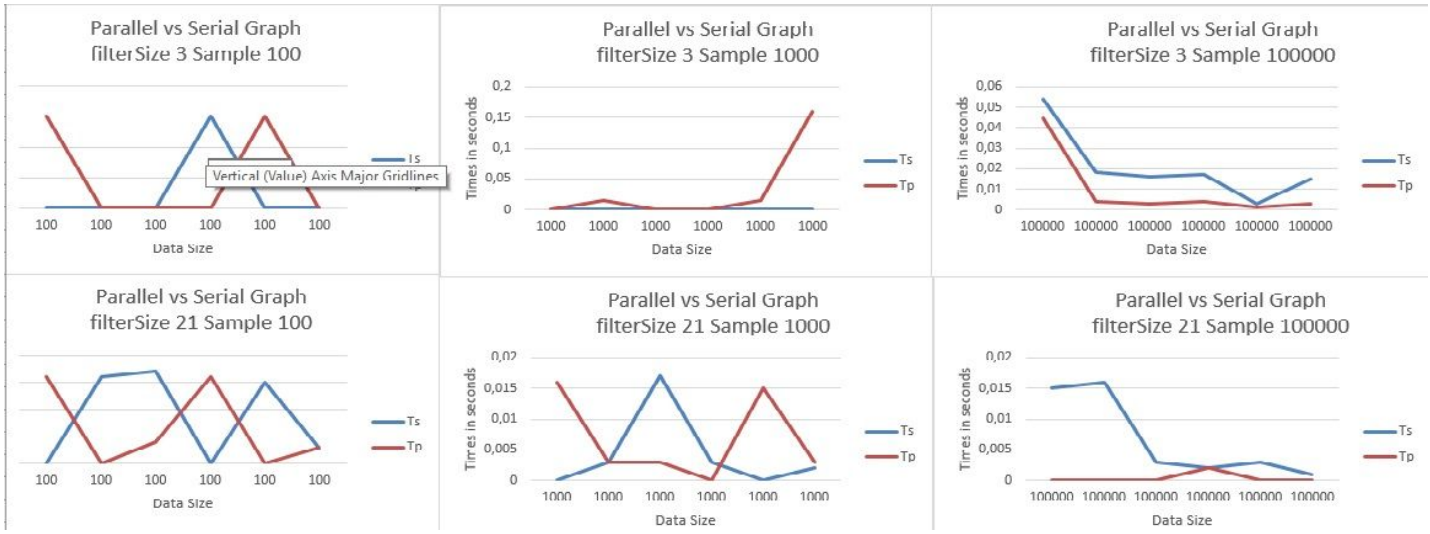
filterSize	DataSize	Ts	Tp
3	1000	0	0
3	1000	0	0,016
3	1000	0	0
3	1000	0	0
3	1000	0	0,015
3	1000	0	0,16

filterSize	DataSize	Ts	Tp
3	100000	0,054	0,0446
3	100000	0,018	0,004
3	100000	0,016	0,003
3	100000	0,017	0,004
3	100000	0,003	0,001
3	100000	0,015	0,003

DataSize	Ts	Tp
100	0	0,016
100	0,016	0
100	0,017	0,004
100	0	0,016
100	0,015	0
100	0,003	0,003

filterSize	DataSize	Ts	Tp
21	1000	0	0,016
21	1000	0,003	0,003
21	1000	0,017	0,003
21	1000	0,003	0
21	1000	0	0,015
21	1000	0,002	0,003

filterSize	DataSize	Ts	Tp
21	100000	0,015	0
21	100000	0,016	0
21	100000	0,003	0
21	100000	0,002	0,002
21	100000	0,003	0
21	100000	0,001	0



Results captured using Intel i3 PC with 3.3GHZ
3GigRam

Based on these results we can clearly see that using parallelization which is multiple thread to solve a problem of big data is efficient for huge processes and costly for small data problems. It is worthy to use parallel approach to solve this problem in Java

The program performs well for filterSize 21 and 100000-10000000 data set, that is for i3 laptop and for i3 Pc filterSize 21 and 10000000 works well, because you produce better results.

Some calculations on speedup based on well performed times by two different architectures

filterSize 21		data set 1x10^6	
i3 laptop	AVG Ts	AVG Tp	Speedup
	0.21	0.149	1.409396

filterSize 21		data set 1x10^6	
i3 PC	AVG Ts	AVG Tp	Speedup
	0.006667	0.000333	20

These calculations are both done on excel.

This was expected on from my understanding that PC tower computers have more processing speed and memory so they have more threads waiting to execute and if they have GPU they allocate certain processes to different parts of operating system so they bound to perform better than a laptop.

The optimal sequential cutoff was 1000000 because anything smaller caused the program to crash and not process well with the sequential program.

The optimal number of threads for i3 acer Aspire e-571 is 4 threads
The optimal number of threads for i3 Dell Inspiron 3464 is 4 threads

Conclusion

This is an interesting topic and can somewhat be revolutionary to the age of computers if algorithms can be optimized and made to be more efficient. I personally learned a how processing works because I always wondered how processes occur and theory behind it, since we learning concurrency aswell should be more interesting. Such experimental assignments build curiosity and directions as to which field one should focus on in furthering their studies.