CSC2002S: Assignment 1 Report
Parallel 1 D Median Filtering

Adam Edelberg

August 2015

1 Introduction

Overview This investigation analyses the computation time of a 1 D median filter on an array of decimal data points using both a sequential and a parallel algorithm approach. A 1 D median filter is a nonlinear digital filtering process that is often used in reducing noise on a data signal.

The filter algorithm is designed to iterate over the whole array and captures the beginning and final element bounds of the data set. The middle entries are processed in small sections known a window. The window iterates over the whole array grabbing the median value of each window iteration. This value then replaces each element and increments until the whole array has been processed. The size of this window is known as the filter size and this determines how many elements are used to calculate the window median.

In this investigation, the program will compute arrays of varying lengths of data and uses the concepts of parallelization in the attempt that large data sets will be able to be filtered in a more efficient and faster execution time.

Why it is worth considering parallelization Parallelization is a useful technique in coding which aids in speeding up the computation time of otherwise comparative but (possibly slower) sequential algorithms. Whilst parallelization of algorithms can be advantageous in improving performance and theoretical computation times, parallelizing code can also lead to detrimental errors in code and possibly reduce efficiency depending on the actual parallelization viability of a specific sequential code.

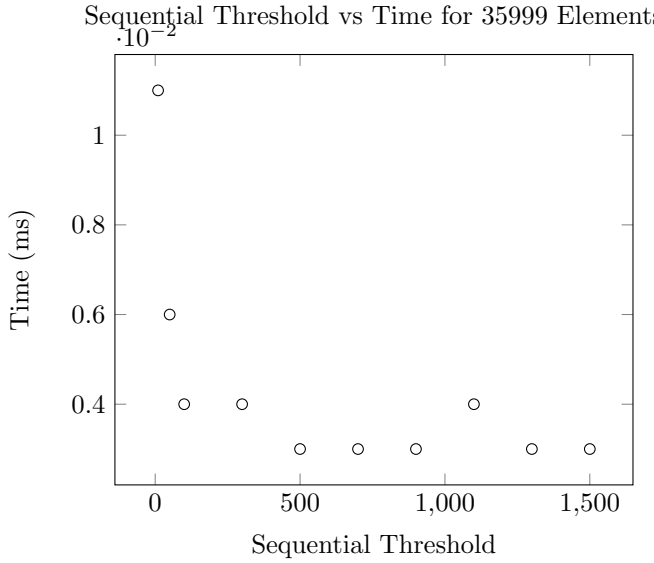
The computational benefits of parallelizing code can be roughly determined by the use of Amdahl's law which suggests that parallel code algorithms are essentially bottle-necked by sequential code within any given algorithm and whilst increasing the number of processors in a system can improve the computation time of a given algorithm, the increase of processors versus the improvement of computation time quickly adheres to the law of diminishing returns. These caveats are taken into account during the development and testing of this program.

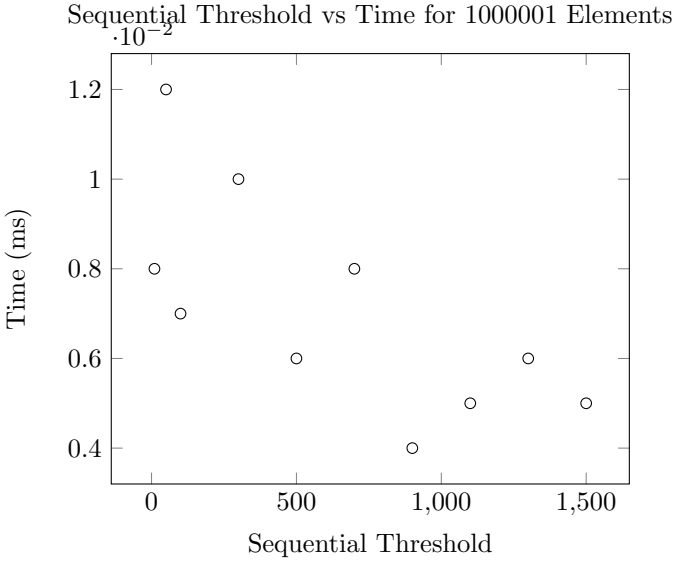
2 Methods

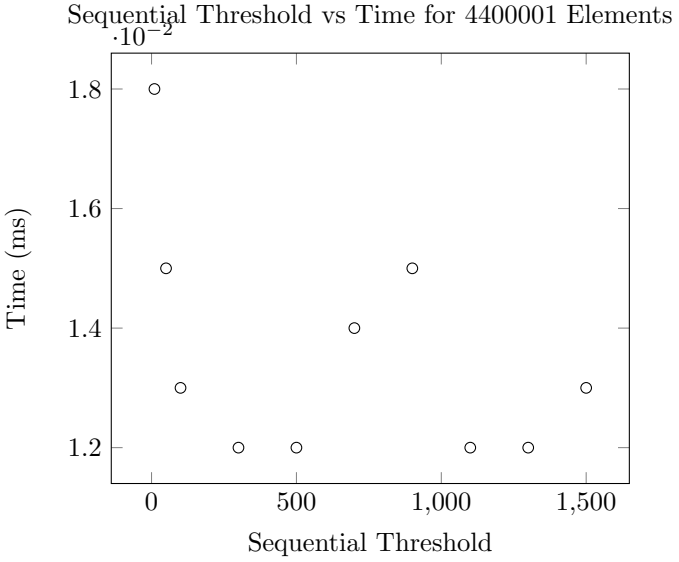
Methodology To create the parallel code of the median filter program, a concept sequential program was first coded in order to provide a working prototype as well as a control and base line for further comparative tests. The parallelization of the sequential version makes use of Java's Fork/Join framework which helps take advantage of multiple processors by means of a dividing and conquering work. This programming approach allows the parallel version of the filter to use a thread pool where the division of work is distributed to worker threads within the pool.

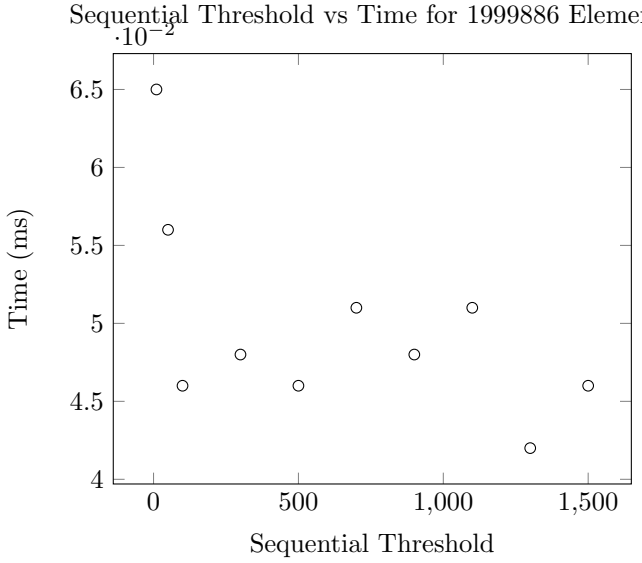
To determine when threads are created for the distribution of work, a sequential threshold needs to be considered. This sequential threshold determines whether the work is to be done directly or split and processed by separate threads. To find the sequential threshold suitable for the median filter, a series of tests using different sequential thresholds and data sizes was executed. The graphs below indicate the various test ranges over different data sets to help determine where the most efficient sequential threshold lies.

Figures 1 - 4: Determining sequential threshold using different data sets.




Sequential Threshold vs Time for 1000001 Elements
$\cdot 10^{-2}$
1.2
1
0.8
0.6
0.4
Time (ms)
0
500
1,000
1,500
Sequential Threshold


Sequential Threshold vs Time for 4400001 Elements
$\cdot 10^{-2}$
1.8
1.6
1.4
1.2
Time (ms)
0
500
1,000
1,500
Sequential Threshold



Validation and Timings To calculate the precise time elapsed by the *filter()* method within both the sequential and parallel classes, the *start()* method is only initiated once the *filter()* method is called and *stop()* is called thereafter. This is done to ensure that only the filter time is being recorded and not the extraneous processing time required for the reading and parsing of the various test files:

```
// code that processes file contents...
// start timer
start();
pool.invoke(filter);
// stop timer
float end = stop();
// code that will print results etc...
```

A performance class was created to conduct the performance testing of the parallel algorithm. The performance class accesses different constructor methods within each respective filter classes. These alternative methods allow for the performance class to pass through test parameters including a user determined sequential threshold parameter not available within the default driver constructor. Furthermore, to improve accuracy these methods also by default loop for ten iterations and return an average of the iteration time.

In order to analyse the speed up of the parallel algorithm, the testing class also invokes the timings from the sequential class and calculates both algorithm computation times. These results can then be used to calculate the parallelization speed up.

The performance class outputs two files, *pOut.txt* and *sOut.txt* with the filtered results and execution time. These two files can be compared and their results allows for both filtering algorithms to be verified and execution times validated.

Furthermore, calling the automatic test feature in the performance class will automatically test all the data sets with the filter sizes, 3, 5, 7, 9 and 11 as well as testing all these instances with sequential limits; 50, 200, 500, 1000 and 1500. These test results are outputted to a file *auto.txt* and the highest recorded speedup for the test sequence is recorded.

Test Hardware The code was executed on a Macintosh UNIX architecture running with an Intel Core i5 running at 2.6GHz with four processor cores.

Problems

3 Results

Effects of Variables Analysing the sequential threshold versus time graphs (Figures 1 - 4 above), it is evident that the interval including values for a sequential threshold from 1000 to 1500 is the most time efficient. Using this sequential threshold as a constant, a series of speedup graphs comparing sequential versus parallel execution times can be calculated. These results and speedup percentages are tabulated below:

Data Set	Sequential Time	Parallel Time	Speedup
35999	0.013	0.003	25.4%
100001	0.016	0.008	47.5%
440001	0.047	0.013	28.9%
1999886	0.108	0.059	54.7%

Table 1: Sequential Limit: 1000 - Speedup

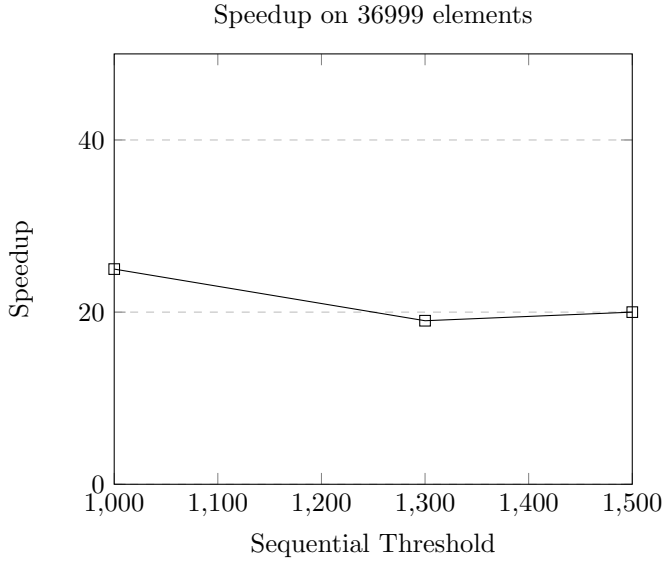
Data Set	Sequential Time	Parallel Time	Speedup
35999	0.012	0.002	19.5%
100001	0.017	0.006	37.1%
440001	0.044	0.017	38.8%
1999886	0.11	0.043	38.8%

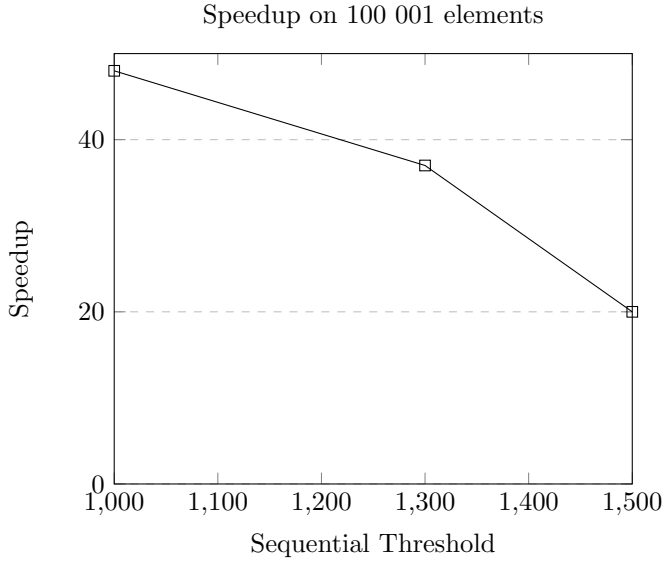
Table 2: Sequential Limit: 1300 - Speedup

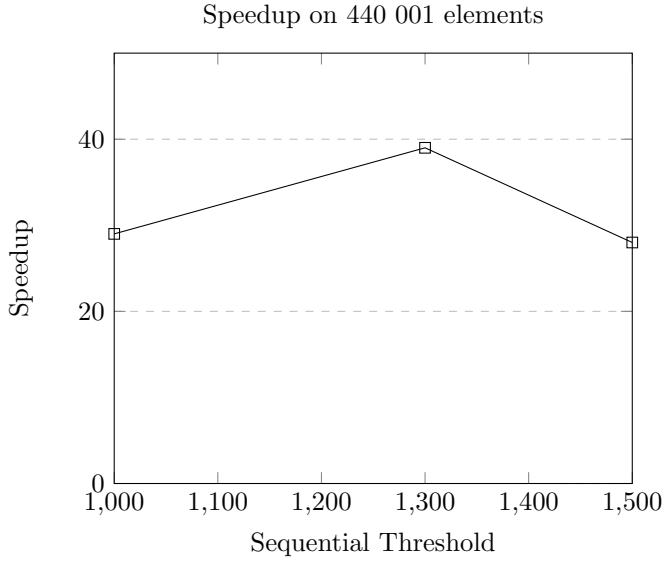
Data Set	Sequential Time	Parallel Time	Speedup
35999	0.014	0.003	20.3%
100001	0.016	0.006	37.4%
440001	0.042	0.012	28.4%
1999886	0.107	0.047	43.8%

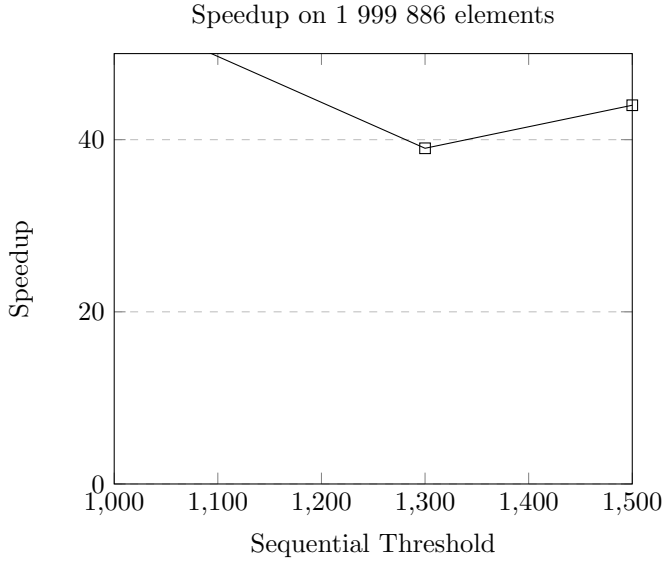
Table 3: Sequential Limit: 1500 - Speedup

Using these results we can graph the speedup percentages to gain a better perspective of the efficiency and success of the parallel algorithm.




Speedup on 100 001 elements
Speedup
0
20
40
1,000
1,100
1,200
1,300
1,400
1,500
Sequential Threshold


Speedup on 440 001 elements
Speedup
0
20
40
1,000
1,100
1,200
1,300
1,400
1,500
Sequential Threshold



Is it worth using parallelization to tackle this problem in Java? Using paralellization to filter the test data sets clearly shows its worth where all test cases proved to produce faster run times than the sequential counterparts. It does however need to be noted that the actual process for creating threads in the Java Fork/Join framework does take time on the first few iterations. Taking this into account the speedups gained from the parallelization does decrease ten fold. Allowing for a few runs and discarding the highest timed run set does however offset this timing issue with the framework.

What range of data set sizes and filter sizes does the parallel program perform well? Under testing, the parallel program did experience a few errors and some interesting conclusions can be drawn. It is obvious to note that a big factor on computation times is the filter size. In the testing output file, the execution time for the larger filter values increases exponentially but it is also interesting to note that it also at these larger filter values, the parallel program speedup results remain the most consistent with average speedups of around 50%. The parallel program really shows its advantages here as the bigger workloads of the increased filter size is able to be shared across multiple threads and thus provides a far more efficient running time.

What is the maximum speedup obtainable with your parallel approach? How close is this speedup to the ideal expected? The theoretical maximum ideal speed up on the test system with four processors can be calculated as $\frac{T1}{T4}$. This equates to a maximum possible speedup of 400% over the sequential algorithm. The highest speedup recorded by the performance class is 153.57%.

This result is a more representative speedup value as the testing environment is not under ideal conditions due to there being multiple programs and processes running in the background which would only be detrimental to the running time of the testing code.

4 Conclusions

The results obtained from this investigation are not to be taken as an incontrovertible example of the benefits of parallel computing. Whilst the parallel code did predominantly run faster than the sequential revision and was more efficient with even very large data sets, the actual coding of a parallel program proved to be more complex and time consuming than the simpler sequential counterpart. The very fact that this investigation covers the analysis of certain variables pertaining to the performance of the parallel code is evidence of this.

It is without a doubt that with all things being equal and under ideal conditions, with more processing cores, sequential code that is well parallelized will in fact run faster as the results and timings obtained above do prove and more importantly the principles behind parallel computing are based.

Ideally this program needed to undergo more stringent testing for a much longer period of time and for a wider range of test input values and on different machine architectures. As the graphs (Figures 1 - 4) to point out, there is a somewhat haphazard result when it comes to timings. Further, the Java Fork/Join framework is required to be run a few times before actually timing as the increase in speed after a few runs is exponential. More granular testing would provide a more insightful and accurate result to draw conclusions and determine the "real world" improvements of parallel computing, certainly in this case.

What really needs to be considered before coding a parallel program is how much the speedup will benefit the specific application as according to the aforementioned results, under some load conditions, the performance increase is not as desirable as one might think, especially considering the efforts taken into account to actually create a parallel coded version to begin with.