Claire Fielden
FLDCLA001

CSC2002S TUTORIAL 2021

ASSIGNMENT 1: PARALLELISM

INTRODUCTION

The aim of this experiment was to parallelize a medium filtering algorithm using the Java Fork/Join Framework to create a parallel program that is faster than its sequential equivalent.

The Median Filtering algorithm operates by taking in an array of numbers as well as the width of a filter. A smaller array that matches the given width is then extracted from the original array. From the smaller array, a median is found and added to an "output array". The Java Fork/Join Framework was utilized to allocate the search for a median to individual threads in the hopes of speeding up the process.

The fork-join paradigm aids parallelism by providing light-weight threads and therefore less overhead. The "divide and conquer strategy" entails splitting up the duties of the coordinating thread amongst numerous "smaller" threads. A binary tree data structure is used as the pattern for the algorithm. Threads spawn other threads. The coordinating thread is no longer responsible for creating other threads, it will only create the first one.

Thus, the entire original array should be handed to a single coordinating thread, and from that moment the thread should split the array into smaller pieces and allocate each piece to a spawned thread. This process should be repeated until each thread has a small enough area that the sequential algorithm can be performed upon it. This is known as the sequential cut-off. Once the size of the problem for each thread becomes less than the sequential cut-off, the algorithm is performed sequentially, and the result returned to the thread that spawned it. The expected value for the minimum sequential cut-off will be the width of the filter, as this is the minimum length of an array. The algorithm traverses back up the binary tree after the leaf-threads have performed their duties. Once the root is reached, the result is returned to the main program.

It is common knowledge that any sequential program can be expected to have a time complexity of O(n), where n is the size of the array. Thus, as the length of the array increases, the sequential program will take longer to perform the algorithm. The parallel program, however, makes use of a binary tree data structure. Therefore, using a divide-and-conquer strategy, the total time is expected to be the height of the tree. Due to this, the overall calculation time as the threads step back through the problem is expected to be of O(log n) complexity for the parallel method.

The goal of this is experiment, consequently, is to find the optimal sequential cut-off that will allow the parallel program to perform the algorithm sequentially up until a certain array length is entered, at which point the algorithm should be performed parallelly to ensure maximum speed up.

METHOD

Overview of approach

Instead of heavy threads, the Fork Join Framework was used. A child class of RecursiveTask was created, which is a generic type. The object to be returned was specified to be a *float[]* array. This allowed the operations for the divide and conquer strategy to be associative as it did not matter which way the threads interleaved, each segment of the algorithm could be calculated at a separate time and the way they amalgamated into the final array did not matter. Instead of overriding the **run()** method, the **compute()** method was overridden, which returned an explicit *float[]* object. In the main class, a fork-join pool was created and invoked for the entire program. This was created as a static field so that it could be used as a global variable. The *RecursiveTask* subclass was used as a thread, with methods **fork()**, **join()** and **compute()**. The **fork()** method split away and gave the threads that took a portion of the array independent execution. The **compute()** method encapsulated the desired behaviour for the array, and the **join()** method waited for the completion of the other thread.

Full description of approach

First, the median filter algorithm was coded sequentially to make sure it worked correctly. The float data type was used because it has 6-7 total digits of precision and it only takes 4 bytes of computer memory, making it the cheaper option.

The parallel algorithm was created from the sequential starting point. The program begins by reading in the name of the data file from which to obtain the data set, the filter size as well as the name of the file the processed data should be written to. The input data is inserted into a *floatArr* via the **getInputArray()** method. This method takes the *BufferedReader* and *fileSize* in as arguments and returns the array of given floats. The filterSize is then double-checked, as the widths can only be of an odd size. This is done via the **checkFilter()** method, which takes in the *filterSize* as an argument, and repetitively asks the user to enter a correct *filterSize* until a satisfactory integer is provided. This integer is then returned to the main method.

A *medianArr* of type float is then initialized. The border is also initialized, which is determined by taking half the size of the filter width. Since the *filterSize* can only be an odd number, the border is automatically rounded down. This *medianArr* then has the front and back borders inserted into the relevant indices via the use of for loops, from the corresponding indices of the *floatArr*.

Once the border is obtained, **System.gc()** is called. This is an API provided in Java. When invoked, it will make its best effort to clear an accumulated unreferenced object (i.e. garbage) from memory. This is useful to be called just before the threading starts as it may minimize the likelihood of the garbage collector running during the execution of the threading. This would, consequently, have an influence of the time calculated for the parallel program to run.

After this, the function **tick()** is called, which will subsequently start the timer. Whilst the timer is running, the **fillMedianArr()** function is called. This function receives the following variables:

- medianArr - a float[] array that will be outputted as the final result. At this point, it has been preconditioned because it already contains the borders of the data set.
- floatArr - a float[] array containing all the input data.
- border - the calculated border length of the medianArr, which will act as the start point from which the threads need to start filling the medianArr.
- filterSize - the size of the filter as entered by the user.
- border - entered a second time because the border will be utilized by the threads. This value will remain unchanged whilst the “starting point” of the array will change as the threads are spawned during parallelism.

Details of parralelization

Within the **fillMedianArr()** method, the new length of *medianArr* is calculated, which will end just before the index containing the first border value. The Fork Join Pool is then invoked and the above-mentioned variables are passed on to the **fillArr()** class.

Within the constructor of the **fillArr()** class, the following variables are instantiated:

- medianArr - the same medianArr as aforementioned, which will receive different medians to be inputted to its indexes.
- floatArr - the original array of data as inputted by the user.
- lo - this variable will be the starting point of each subarray and is originally set to the value of the border.
- hi - this variable will be the end point of each subarray and is originally set to the value of the length of the medianArr with the border value subtracted from it.
- filterSize - the size of the filter as passed from the main program.
- border - the value of the border, as calculated by the main program.
- SEQUENTIAL_CUTOFF - a variable of type final int, which means this integer value cannot be changed by any of the threads. The value is set to the size of the filter incremented by 1.

Reason for my original sequential cut-off

Given that an array x = [2 80 6 3 1] and a median filter of size 3, as per the assignment brief. This array of size 5 could be split into a maximum of 3 arrays. One subarray to be searched would look as follows:

INDEX	0	1	2
VALUE	2	80	6

The subarray must have a length of 3, no more and no less, as this is the given filter size. In the above case, *lo* would be 0, *hi* would be 3, and the whole array would then be traversed through to find a median. The sequential cut-off, being set to (*filterSize*+1) would be 4 in this case. This array would only be assessed by threads when the portion received was (*hi*-*lo*) < 4. I.e. 3-0 = 3.

This, no doubt, worked well. The algorithm could perform in parallel. However, it had to be assessed whether it was vital for the array to be broken down into such small chunks to perform optimally. Perhaps allowing the array to remain a length of 5 and computing each median sequentially would be faster and have less overhead than breaking it up as above-mentioned.

Above the sequential cut-off

If the length of the array was not yet below the *SEQUENTIAL_CUTOFF*, the array was split by a fraction of ((*hi*+*lo*)+1)/2. With the above example in mind, the array would be split as follows:

Since lo is originally the length of the border: 1

And hi is the length of the array with the border subtracted: 4

hi+*lo* would be 5, however an array cannot be split by a fraction of 2.5 (to my knowledge), and therefore the sum was incremented by 1 to get a value of 6/2 = 3.

Left:

From index 1 to index 3: [80 6 3]

Right:

From index 3 to index 4: [3 1]

The split may seem strange at this point in the explanation, but this algorithm is all about the numbers. The numbers produce the correct result. The +1 needs to be there to ensure correct splitting of the array amongst threads otherwise a “Stack Overflow Error” will be incurred. So the left thread will have an array of index from the border to (((*hi*+*lo*)+1)/2, whilst the right thread will have an array over the span ((*hi*+*lo*)+1)/2 to *hi*. The length of the right data set is 2 and length of the left data set is 3, so both will be performed sequentially.

The **.fork()** method is then called for the left thread, which creates a separate address space for the child. The left child will then execute. Whilst the left child is executing, **.compute()** is called for the right child. This is a hand-optimization method whereby the thread does not sit idly whilst the other thread is executing. Instead, it does half the work whilst it waits. The **.join()** method forces the left thread to wait for the right thread to finish executing before the *medianArr* is returned to the parent thread.

Below the sequential cut-off

When the length of the array (*hi*-*lo*) is less than the *SEQUENTIAL_CUTOFF*, the sequential algorithm is performed upon the dataset.

A for loop with variable *i* is initialized that will go through the whole *medianArr* from *lo* to just before *hi*. Therefore, with the above-mentioned left thread in mind, it will start at 1 and end at 2. A new float array, the *analysisArr*, is initialized to the length of the *filterSize*.

Another for loop with variable *j* is then initialized to fill the *analysisArr*. The index *j* of the *analysisArr* is filled according to the following code:

analysisArr[j] = floatArr[i-border+j];

Where *j* is the position in the *analysisArr*, and *(i-border+j)* is the position in the originally inputted *floatArr:* x = [2 80 6 3 1]. For the left thread, the *analysisArr* will be filled as follows:

analysisArr[0] = floatArr[1-1+0] = floatArr[0] = 2

analysisArr[1] = floatArr[1-1+1] = floatArr[1] = 80

analysisArr[2] = floatArr[1-1+2] = floatArr[2] = 6

Then the median will be calculated via the **findMedian()** method, which takes in the *analysisArr* and the length of the border. This method sorts the array using Java’s Arrays.sort() and finds the median according to the border. The border is assumed to be the index of the *median*.

The *median*, 6, is then placed into position *i*, 1, of the *medianArr* before *i* is incremented and the process begins again. The second iteration of the for loop will fill the *analysisArr* as follows: [80 6 3] and place 6 as the *median* into position 2 of the *medianArr*. The for loop will then be exited and the *medianArr* returned to the parent thread. The parent thread will then wait for the computation of the right thread.

For the above-mentioned right thread, [3 1], *lo* = 3 and *hi* = 4. The analysisArr is filled as follows:

analysisArr[0] = floatArr[3-1+0] = floatArr[2] = 6

analysisArr[1] = floatArr[3-1+1] = floatArr[3] = 3

analysisArr[2] = floatArr[3-1+2] = floatArr[4] = 1

The *median*, 6, will be placed into the third position of the *medianArr*. Then the for loop will be exited and the *medianArr* returned to the parent thread. However, since the parent thread is actually the coordinating thread, the *medianArr* is passed back to the main program.

Now that the main program has the filled *medianArr*, **tock()** is called to calculate the total amount of time the parallel algorithm took to run.

Timing of algorithms

The static variable, *startTime*, was declared as a long at the beginning of the parallel algorithm and initialized to zero. When **tick()** was called, this *startTime* variable was initialized to the current value of the most precise available system timer, in nanoseconds using **System.nanoTime()**. This method was used instead of **System.currentTimeMillis()** because it is more accurate than measuring time in milliseconds due to the smaller units.

When **tock()** was called, **System.nanoTime()** was referenced again and the difference between the result and the initial *startTime* obtained. This resuly was then divided by 1000 000 to get the answer in milliseconds and returned to the main program. The result was then stored in a float variable, *time*, for reference.

Speed up measurement

One the time taken to run the parallel algorithm was stored, a *PrintWriter* was initialized to print to a file named *ParallelTimer.txt*. The following variables were then written to the *ParallelTimer.txt* file:

- fileSize - the size of the inputted array.
- filterSize - the width of the filter.
- time - time taken to complete the parallel computation.

This time data could then be used later for analysis of time for different array sizes (*fileSize*) and sequential cut-offs (*filterSize*).

Algorithm validation

Once the timing data was written to the file, the **writeToFile()** method was called. This method takes in the name of the output file, *fileOut*, as previously entered by the user as well as the *medianArr*. The contents of the *medianArr* is then written to the file in the same format as the input file.

I utilized this output file to ensure the parallel program worked properly by doing the following:

1. Comparing the output to the sequential program's output. Since the sequential program was created before the parallel version, I tested each of the given data sets on the sequential version first to ensure it worked correctly.
2. Ensured the sequential algorithm's output was correct for each data set by finding the median of a set of numbers in the beginning, middle and end of each text file. This was done for varying filter widths and an online *Mean, Median, Mode, Range Calculator* was used from *Calculator.net*.

Machine architecture

I checked my system architecture using the following code:

```
String operatingSystemArchitecture = System.getProperty("os.arch");

Runtime runtime = Runtime.getRuntime();

int numberOfProcessors = runtime.availableProcessors();

System.out.println("Architecture: "+operatingSystemArchitecture);

System.out.println("Available processors: "+numberOfProcessors);
```

Which returned the following results:

Architecture: amd64

Available processors: 4

<u>Problems and difficulties</u>

The main problem I encountered was a stack overflow error when coding the system in parallel. The system kept returning this error many times with many errors I did not understand such as "unintentional recursion" and at one point the keyword "dangerous" was used, which made me quite fearful. I also did not want to throw the exception as one of the tutors said this could be very harmful to my computer, and my computer is on its last legs as it is.

I tried removing the **findMedian()** method from the **fillArr** class as I thought maybe multiple threads were trying to access the method at the same time, however this did not work. I then tried changing the **findMedian()** method from a bubble sort method to an **Array.sort()** method, which did not work either but I decided to keep this change because it uses less memory and we're all about saving resources.

Finally, I went through the program with a pen and paper. The Stack Overflow error is all about a thread's stack size growing beyond the allocated memory limit. I tried looking for the bug and realized:

a) My SEQUENTIAL_CUTOFF was too low and changed is to *filterSize+1*.
b) I was splitting my threads in half, which did not work with the *filterSize* being odd. I believe this was the main contributing factor to my Stack Overflow Error.

<u>Obtaining results</u>

My method for calculating the limits for which the parallel program will be faster than the sequential program consisted of running both the sequential and parallel program for the same data set length and filter widths. The following filter widths were analysed:

- 3
- 11
- 15
- 21

For array sizes of:

- 100
- 1000
- 10 000
- 100 000
- 1 000 000
- 10 000 000

First, the Java library was warmed up by running the first problem (array size 100 with a filter width of 3) 5 times before beginning to experiment. The first few runs are always slower due to the initialization of memory allocation and optimization in the background.

Once the results for the parallel and sequential program were obtained in the above cases, the time data could be analysed to draw conclusions.

RESULTS AND DISCUSSION

Effect of data sizes

The relationship between array length and time is displayed in Table 1 and 2 below:

Table One:
The Relationship Between Array Length and Average Time for the Parallel Program

Array length	Average Time
100	2,8
1000	4,76
10000	13,89
100000	32,13
1000000	198,78
10000000	742,16

Table Two:
The Relationship Between Array Length and Average Time for the Sequential Program

Array length	Average Time
100	0,52
1000	2,8
10000	7,81
100000	25,83
1000000	193,93
10000000	1864,59

The data for Table 1 and 2 are represented by Figure 1 and 2, respectively, below.

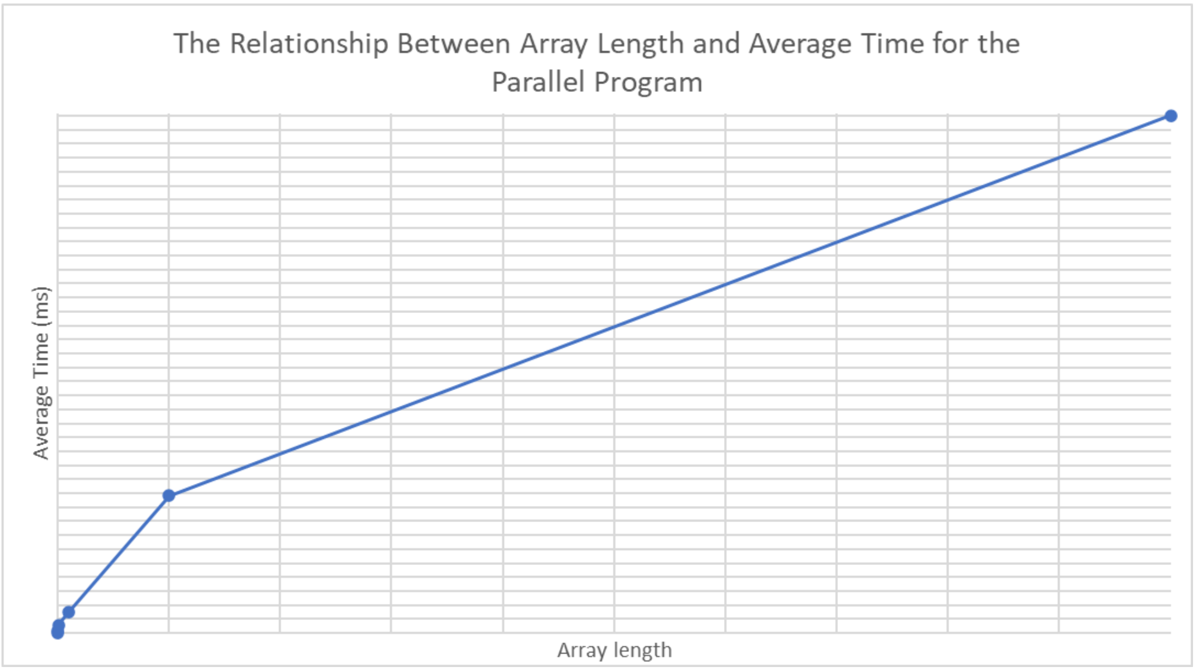

Figure 1 Graph displaying the relationship between average time and array length for the parallel program

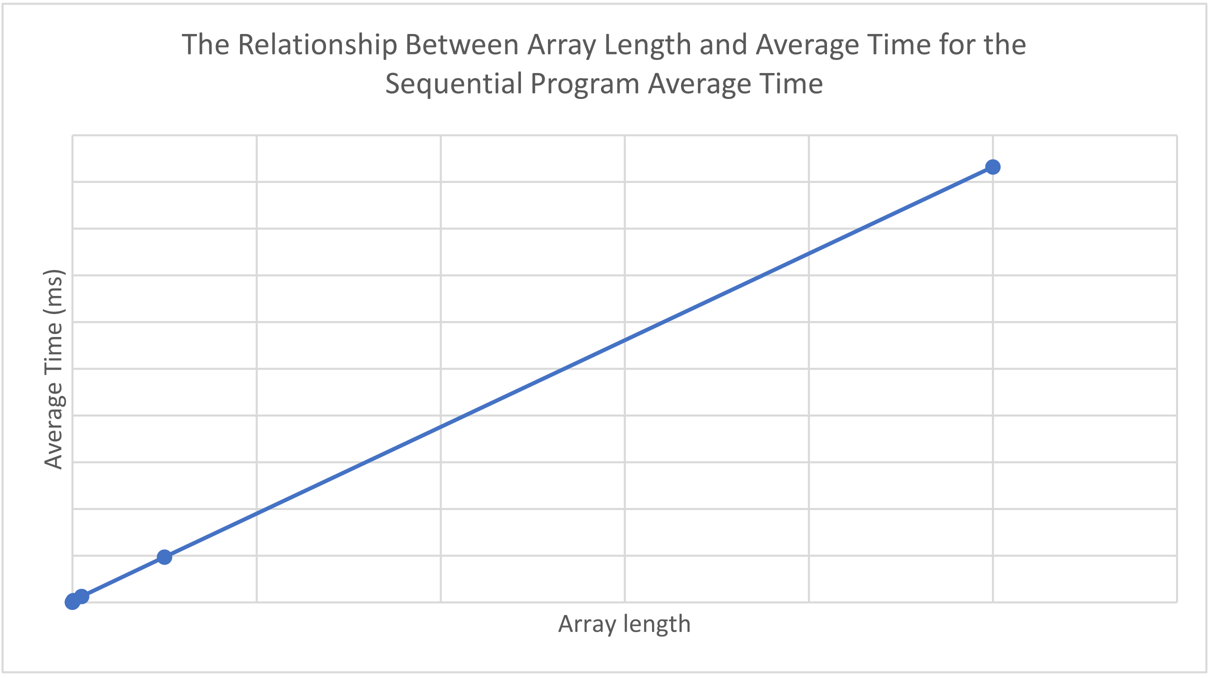

Figure 2 Graph displaying the relationship between average time and array length for the sequential program

I have chosen not to include numbers on the axes above to highlight the different relationships for each program, as the numbers cannot be displayed well on such a small page, especially for the logarithmic graph. It is clear, as per Figure 1 and 2, that there is a logarithmic relationship between time and array size when experimenting with the parallel program. However, the sequential equivalent has a linear relationship between time and array size. Thus, the sequential version of the algorithm takes longer to perform median filtering on a small array when compared to the linear algorithm. However, when the data set becomes larger, the curve of the parallel relationship falls beneath the line of the sequential relationship. Thus, as the data set becomes larger and exceeds the one million length marker, parallel programming will become more efficient. Any length of data below this will be useless to run the parallel program on because the creation of threads to achieve a result is extremely expensive and time-consuming, and the task becomes less beneficial than its sequential alternative.

<u>The effect of different numbers of threads</u>

The number of threads is directly proportional to the number of leaves in the binary tree. Thus, if the size of the array is much larger than the SEQUENTIAL_CUTOFF, many more threads will be spawned until the lower leaves are reached. Due to this, the number of threads is directly proportional to the size of the array in the parallel case. However, in the sequential case, there is only one thread.

Looking at the relationship between array length and average time for the parallel program, one can see that when the number of threads becomes large enough, the performance of the parallel algorithm will be better than the sequential alternative.

<u>The effect of differing sequential cut-offs</u>

For the both the sequential and parallel algorithm, the time taken for each filter size differs. As the sequential cut-off for each run of the algorithm was decided by the filter size, the filter size had an impact on the time taken to find the median filter. The relationship between time, sequential cut-off and array length for the sequential program is outlined in Table 3 below.

Table Three:
The Relationship Between Filter Size, Sequential Cut Off, Array Size and Time for the Sequential Algorithm

		Time (ms)					
	Array Length	**100**	**1000**	**10 000**	**100 000**	**1 000 000**	**10 000 000**
Filter Size	3	0,9358	0,9358	3,3504	11,5551	36,998	267,561
	11	0,4608	2,4368	6,8613	20,45	160,39569	1509,6392
	15	0,5217	3,4916	9,3583	28,914	231,04631	2268,0017
	21	0,6705	4,3281	11,6631	42,3969	347,26453	3413,1743

This information is summarized in Figure 3 below.

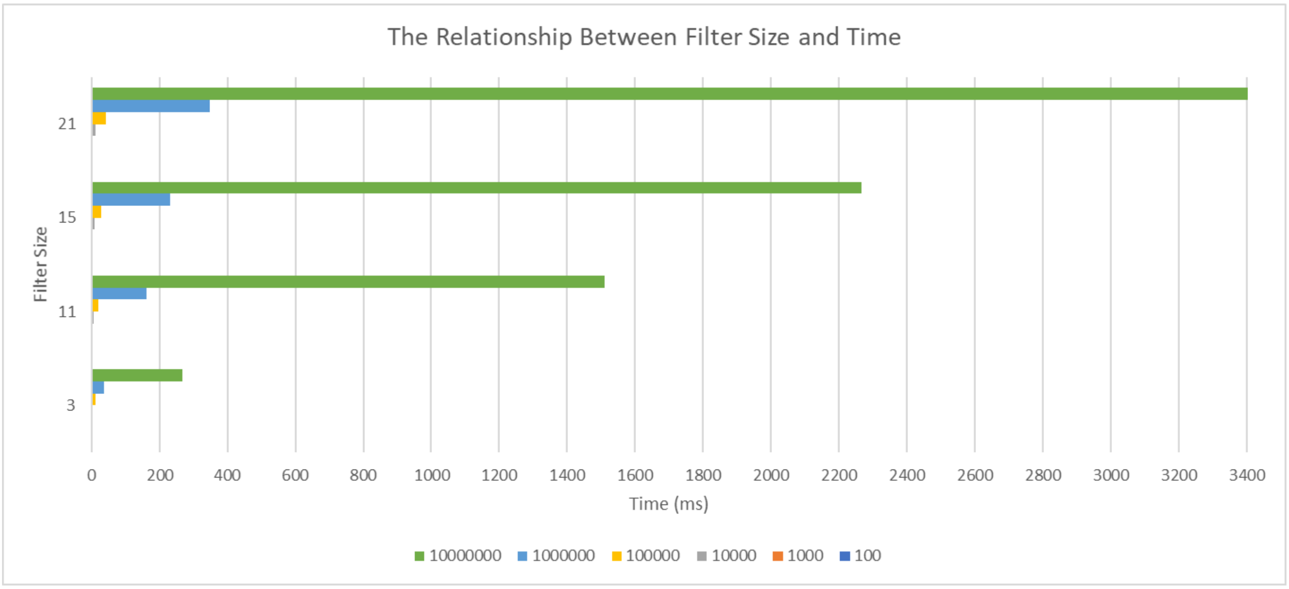

Figure 3 The Relationship Between Filter Size and Time for the Sequential Algorithm

This can be contrasted against the time taken for the parallel algorithm for each filter size, as seen in Table 4 below.

Table Four:
The Relationship Between Filter Size, Sequential Cut Off, Array Size and Time for the Parallel Algorithm

		Time (ms)					
	Array Length	**100**	**1000**	**10 000**	**100 000**	**1 000 000**	**10 000 000**
Filter Size	3	2,5773	4,0594	29,8989	26,1257	156,40489	324,0034
	11	2,8372	4,489	10,7277	30,3468	184,6037	673,8521
	15	3,0676	5,125	13,3103	32,9325	230,6947	891,4544
	21	2,5255	5,349	14,9489	39,1136	223,407	1079,306

The corresponding graph for Table 4 can be seen in Figure 4 below.

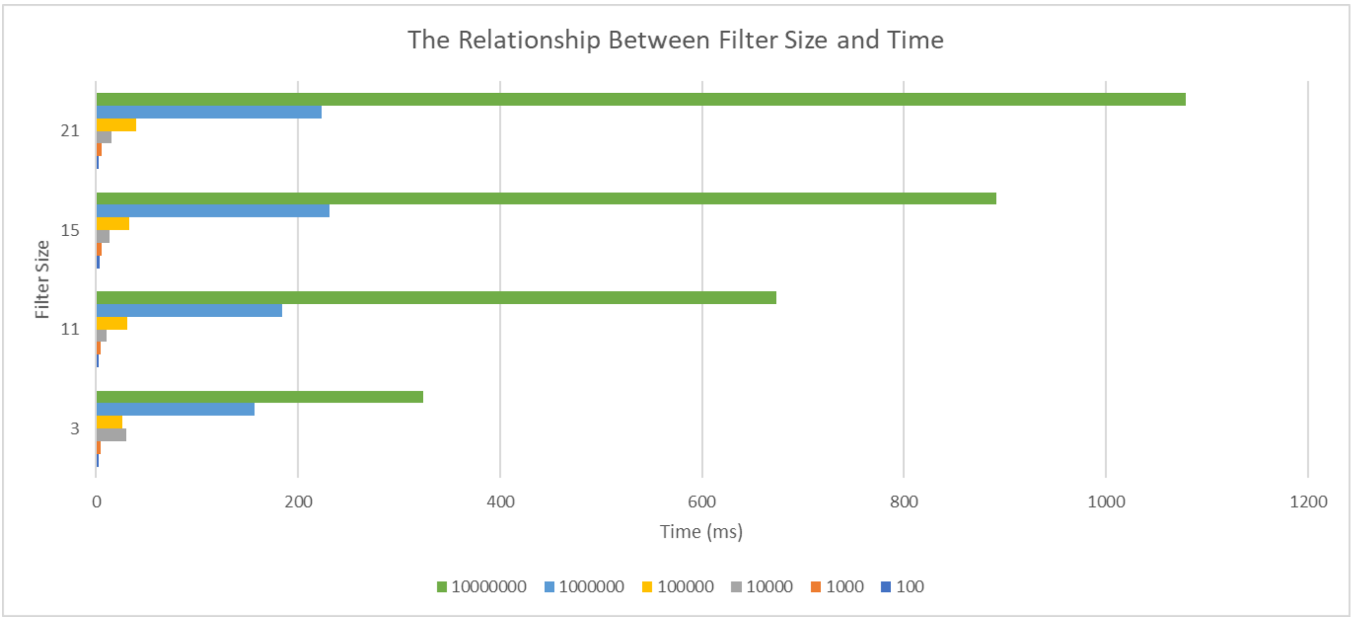

Figure 4 The Relationship Between Filter Size and Time for the Parallel Algorithm

A few trends can be identified through the analysis of Figure 3 and 4. Firstly, the sequential algorithm's time is largely dependent on the size of the filter/sequential cut-off. As the size of the filter increases, the time taken to complete the program increases by a large magnitude. However, for the parallel program, time does not vary as greatly between the minimum filter width (3) and the maximum filter width (5).

Additionally, one can identify that the parallel program becomes more efficient than the sequential program as the filter width increases, given that the array length is more than or equal to 1 million.

<u>Optimal number of threads for this architecture</u>

Since my architecture has 4 cores, the optimal number of threads would be one per core at a time. Since the number of threads are equivalent to the number of leaves in the binary tree, the optimal number of threads would be:

(n - 14)/4 threads per core.

Where n is the length of the inputted data file.

<u>Optimal sequential cut off</u>

In order to find the bull's eye, i.e. the point at which the program should switch from performing the median filter algorithm sequentially to parallelly, is therefore determined by both the length of the array and the size of the median filter. It is clear, by looking at Figures 1-4, that this "sweet spot" can be found between array lengths 1000 000 and 10 000 000. Additionally, the favourable filter is between the number 11 and 21.

To determine the bull's eye, the sequential cut-off was first identified. This was done using the data set of length 1 000 000, since one can see that the parallel program is not even 1ms faster than the sequential version in Figure 3 and 4.

Table 5 displays the data obtained in searching for the optimal filter size. This data was obtained by finding the time for each filter 5 times and then taking the average.

Table Five: **Analysis of Average Times for Filters Using Array length 1000 000**		
Filter Input	**Average Time Taken for Parallel Program**	**Average Time Taken for Sequential Program**
11	178.69	159.144
13	191.53	196.23
15	203.84	227.37
17	207.73	256.57
19	247.29	296.29

Therefore, it can be assumed that the filter size at which the sequential program should be performed in parallel is 13. Thus, the SEQUENTIAL_CUTOFF should be 14.

Taking this sequential cut-off, the array length over which the program should switch from sequential to parallel execution is 1 000 000, as Figure 3 and 4 displays that this is the point when - above the sequential cut-off - the sequential algorithm becomes slower than the parallel algorithm.

Is it worth using parallelization for this problem?

For this problem, parallelization is worth it as it allows an exponential speed up of processing with files 1 000 000 lines in length and longer, with filter sizes larger than 11. Thus, my program performs best for problems with the following specifications:

- File size < 1 000 000 (sequentially)
- File size >= 1 000 000 and filter size>=13 (parallelly)

This leaves a small window of files that are less than 1 000 000 in size with a filter width larger than 13. These files will not experience parallelism, however the difference between time is a maximum of 4 ms, which is a small price to pay for a speedup of 2 minutes as seen for a file of length 100 000 000 and a filter width of 21 (Figures 3 and 4).

Maximum speedup

The maximum speedup of the parallel processing of a file was expected to be T_1/T_∞, where T_1 is the amount of time for the algorithm to iterate sequentially, and T_∞ is how long it would take an infinite amount of processors to finish the problem. Amdahl's law cannot be applied to calculate speed up because the ration of sequential processing to parallel processing changes with each file length and filter size. Thus, we apply Gustafson's law, which bases the problem size upon the number of processors. This implies that the speedup of the parallel program is guaranteed to be $(T_1/P) + O(T_\infty)$.

Taking the example of file size 10 000 000 and filter size 21, we can analyse the following:

T_∞ is the longest dependence change, for this array of n numbers, the T_∞ will be log(n) as the divide-and-conquer strategy has been made use of. The overall calculation time as the thread step back through the problem is proportional to the height of the tree, which is of log(n) complexity.

- Log(10 000 000) = 7
- T_1 is the time taken to execute the algorithm sequentially, and is thus = 3413.1743
- P is the number of processors = 4

The expected speed up for this problem would be as follows:

(3413.1743/4) + 7 = 860 ms

Therefore, the fork-join framework guarantees an expected time of 860ms per processor. This is an assumption that is asymptotically optimal and ignores memory-hierarchy issues.

The actual speedup obtained for this same problem was as follows:

3413,1743 - 1079,306 = 2333.86 ms

This falls below the (860*4) = 3440 ms mark, and thus the results from my program falls within the bounds predicted by Gustafson's Law.

CONCLUSIONS

The median filtering technique used to remove noise from a data set can be done sequentially to output a data set of small size. However, a data set of 1 000 000 items or more will become expensive to apply the technique to for a filter of width 13 or more. Thus, parallel processing can be used to achieve a speed up of up to 3440 milliseconds, as per Gustafson's Law. The results obtained are reliable due to the achievement of averages amongst 5 tests for each timing attained, per filter, per data set length. The speedup is independent of the architecture of the machine being used to apply the technique; however, an increased number of processors may significantly increase processing time.

The divide-and-conquer strategy was chosen for this experiment, as it creates threads to split the work up between them until the sequential cut-off is reached. When the thread area is below 14, the algorithm is performed sequentially to further supply the algorithm with speedup and reduce overhead. Additionally, the fork-join paradigm allows light-weight threads to handle the problem, which allows a lesser overhead than the more heavy-weight Java threads.

Henceforth, not only does the implementation of parallelism on the median filtering technique reduce processing time for larger data files, it also aids in the reduction of computer resource usage.