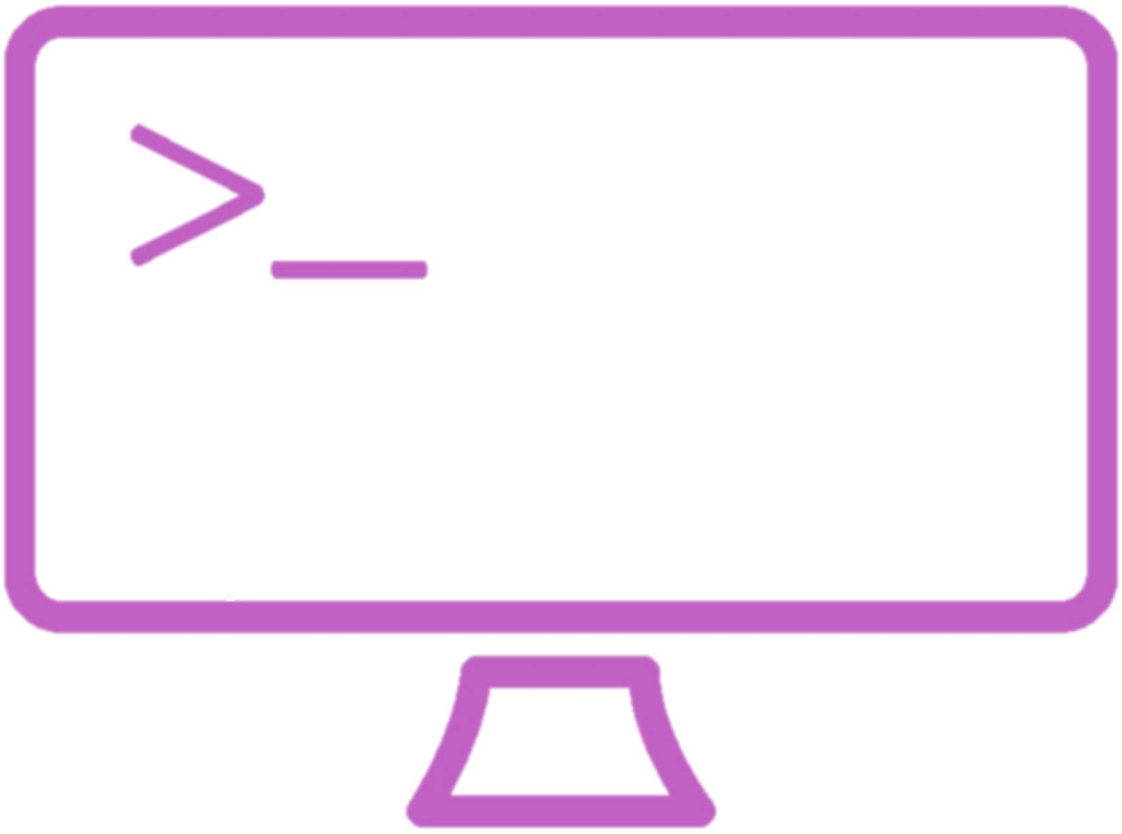
Assignment 1

Cameron Rebelo RBLCAM001



Introduction

The aim of the project is to use java fork join framework to speedup the process of applying a median filter on a set of data. We are using this framework in order to decrease the time that it would take process the data by making use of threads to run in parallel.

The parallel algorithm that my project uses it described as follows:

> The data is put into an array and given to a recursive task function. The array then recursively splits into left and right nodes until the length of the array in the node is less that the sequential cut off. At that point the array then takes each individual element of the array between the boundaries and creates a filter around it (based on the given size of the filter) and chooses a median value. This value is then parsed to an output array object at the same index as the original piece of data. Every time that the array recurses a new thread is created in the fork join pool framework which then runs in parallel thus the program is parallelizable.

The machine that the program is being tested on has a i5 @ 2,3Ghz that has 2 cores. We then know that in an ideal situation, the speedup of the program is going to be 2. In a more real-world sense, we should probably expect a speedup less than that closer to a value of 1,8.

Methods

Approach

This program's main purpose is to make use of threads to apply a median filter to a dataset. The solution works by first reading a command argument that gives the name of the input file, the filter size, and the output file name. Using this the data is read in from the input file into an array of floats. At this point the program also creates an instance of an outArr object that will store the values of the medians determined to be output later. This is when the parallel algorithm of the solution begins so a method is run to start recoding the time it will take to run. The data is then parsed to the thread class by invoking a fork join pool. Withing the thread class, the data is run through the compute() function. It checks whether the size of the array is less than the sequential cut off point. The array is divided recursively (also creating new threads every time) until the appropriate size is met at which point the data is processed by creating filters of all the elements within the array. The median is then obtained from each filter and parsed into the outArr instance with their correct index. The .fork and .join methods are used here as well to make sure the order at which the leaves of the recursive function is run correctly. Once every piece of data has been processed a method is run that records how long it took (in milliseconds) since the program started the timer. The output array of floats is then formatted correctly and written in the output file.

Testing

This program was tested by counting the time that it takes to run the invoke command of the fork join pool. This therefore measures the time that it takes to compute the median filter while making use of threads. Tests were then then done for every filter size between 3 and 21 with a sequential cut-off of 100. These times were then recorded 20 times in order to find an average for every filter size for every sampleInput size. Tests were then run where the sequential cut-off was set to be equal to the size of the respective datasets so that a sequential timing could be recorded for that data set, this was done 20 times to obtain an average as well. The speedup of the program was then calculated by dividing the sequential time by the parallel time; the expected speedup of the machine that the tests were run on should be 2 as the machine had 2 cores. The data was also put through tests to determine the optimal sequential cut off by running the program through many different values for the cut off. The data was validated to be correct by taking samples of subsets of data from the input and output files and doing some manual calculations to ensure that the median was indeed found and put in the correct index.

medianFilter.class

This is the main class of the program. When this program is run, it takes an argument from the command line that provides the program with an input file, filter size and output file. The program then sends the populated float array with the relevant data to be processed through the median filter classes. This class also creates an instance of an output Array object that serves as the final output array of the median filter data. This class also handles all the output algorithms to write the outputted data in the correct format to the desired output file.

tick()

This method starts the clocks for measuring time.

tock()

This method returns the amount of time since the tick() method was run.

prune()

Parses the array of floats to the recursive methods with a determined filter as well. This method also times how long it would take the parallel method to run.

main()

The main method of the program that takes in input from the command line including the input file name, the filter, and the output file name. The method scans the input file and places the data of all the floats into an input float array. This method calls the prune() method with the array full of floats and that returns the time that it takes to sort through this data with a median filter.

setOutputArrayElement()

At a particular index in the output array object, set the value.

outArr.class

The purpose of this class is to protect the output array by removing it from the thread class so that it may act as a global variable for all threads. When a median value is calculated in a thread then the value to be outputted is rather sent to this variable to be protected. This solution was used because when the output array was within the thread class, the value at index would not update for all arrays across all threads and thus when the output array was returned, it would be essentially empty.

outArr()

Constructor for class.

getOutput()

Getter method.

setOutput()

Setter method.

medianThread.class

This is the thread class of the program that is responsible for recursively splitting the float array into threads that will calculate the median values of the filters. The class is used in the fork join pool framework where each thread is then put into a pool of threads that will be allocated to the different cores of the CPU in the most efficient way possible.

medianThread()

Constructor method for the class

compute()

The overloaded method from the RecursiveAction parent class. This method checks if the sequential cut-off is met; if yes then it changes all values in that array into their median filter version then sends those to the output Array object to be placed in their respective indexed place. If no, then two new objects of medianThread() will be created where the array is split in half. The method then makes use of the .fork() and .join() methods to run the threads in the program in the correct order.

sort()

This is a helper method that sorts an array parsed to it.

median()

Helper method that returns the median float when given an array of floats

Results and Discussions

Results

For a full range of all the result values obtained, please refer to the excel spreadsheet that is included in the project folder. Below is listed the average values obtained from running 20 tests on each filter between 3 and 21 (inclusive) performed on 6 different sample input data ranging between 100 and 10000000 elements with a sequential cut off of 100. The average obtained from running the data sequentially and the speedup are also listed in the table

Dataset 100			
Filter	Sequential Time	Parallel Time	Speedup
3	13,31555	12,80990	1,03947
5	5,69630	4,85680	1,17285
7	7,11200	6,34615	1,12068
9	4,94630	4,53155	1,09152
11	7,51470	3,35875	2,23735
13	5,70250	4,08075	1,39741
15	5,70250	3,81835	1,49345
17	3,48790	2,90665	1,19997
19	4,22165	3,34040	1,26382
21	4,19960	11,96695	0,35093

Dataset 1000			
Filter	Sequential Time	Parallel Time	Speedup
3	25,61050	7,12575	3,59408
5	9,14155	3,7735	2,42257
7	8,66635	6,92305	1,25181
9	12,72770	5,2103	2,44280
11	10,27685	6,36975	1,61338
13	7,73280	5,48	1,41086
15	9,05460	7,00145	1,29325
17	10,72520	9,4142	1,13926
19	10,44955	8,6982	1,20135
21	11,06905	9,8101	1,12833

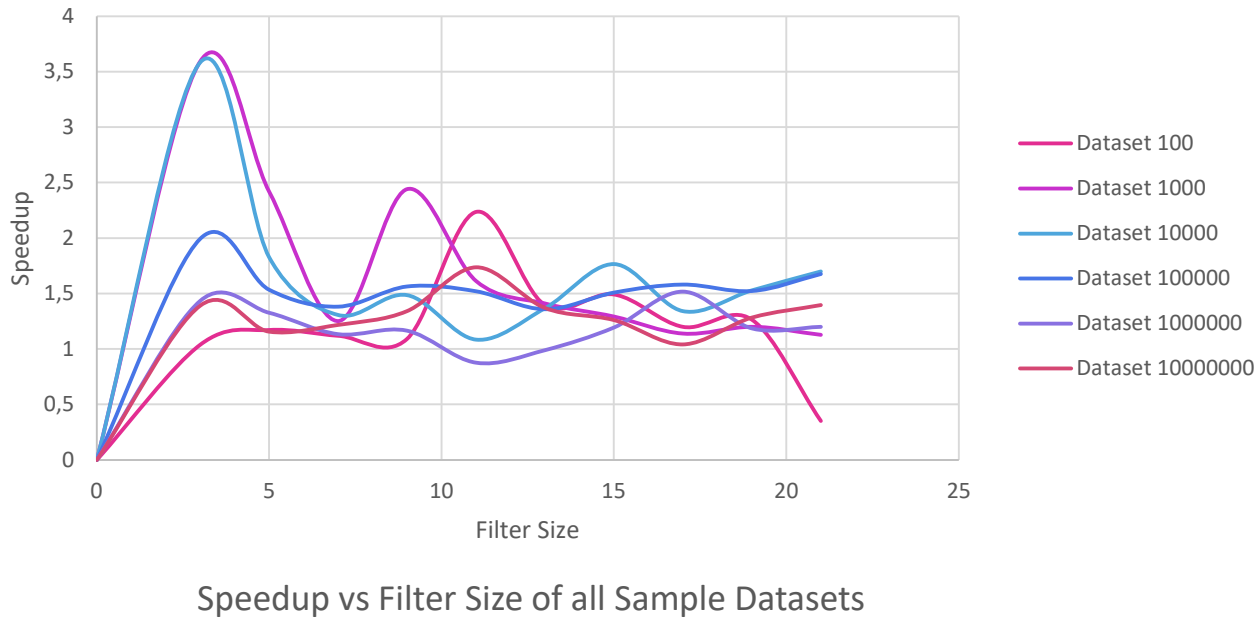
Dataset 10000			
Filter	Sequential Time	Parallel Time	Speedup
3	85,56805	23,8634	3,58574
5	28,38285	15,501	1,83103
7	24,24320	18,56155	1,30610
9	28,18060	18,94545	1,48746
11	29,16790	26,854	1,08617
13	38,82080	28,39	1,36763
15	55,59710	31,4972	1,76514
17	50,56720	37,67885	1,34206
19	59,44140	38,8414	1,53036
21	70,17055	41,275	1,70007

Dataset 100000			
Filter	Sequential Time	Parallel Time	Speedup
3	113,38400	56,8618	1,99403
5	108,49630	70,68465	1,53493
7	148,40945	107,37055	1,38222
9	236,02515	150,9446	1,56365
11	270,76220	177,9732	1,52137
13	344,50995	253,68	1,35804
15	351,09145	232,77965	1,50826
17	414,66140	262,0272	1,58251
19	482,09035	316,5481	1,52296
21	577,50610	344,21935	1,67773

Dataset 1000000			
Filter	Sequential Time	Parallel Time	Speedup
3	484,84160	338,0654	1,43417
5	738,66730	556,66595	1,32695
7	1072,84491	945,8094055	1,13431
9	1400,62489	1201,642812	1,16559
11	1699,69985	1939,750515	0,87625
13	1992,63547	2010,84	0,99095
15	2252,24224	1883,85001	1,19555
17	2856,39025	1882,96478	1,51696
19	3149,70143	2656,50484	1,18566
21	3279,08112	2730,48958	1,20091

Dataset 10000000			
Filter	Sequential Time	Parallel Time	Speedup
3	3120,69255	3750,470465	0,83208
5	4376,93792	5510,483885	0,79429
7	7326,86593	8527,760125	0,85918
9	9435,33525	11443,15648	0,82454
11	12154,50280	14504,85835	0,83796
13	14901,37650	17846,41	0,83498
15	17486,58965	21445,97185	0,81538
17	21209,21490	23905,8327	0,88720
19	24655,67705	20982,0772	1,17508
21	27311,47085	23279,67745	1,17319

Graphs



Speedup vs Filter Size of all Sample Datasets

Figure 1

In this graph we observe how the speedup of the program changes over the different size filters that are used on the various datasets. From the results we can observe that the density of the speedup values lies around the 1,5 mark. The expected ideal speedup is known to be 2, therefore what we see is expected real world results. The speedup of a program doesn't always reach the ideal speedup due to other factors such as optimisation of threads issues and waiting on other threads from other programs to run as well.

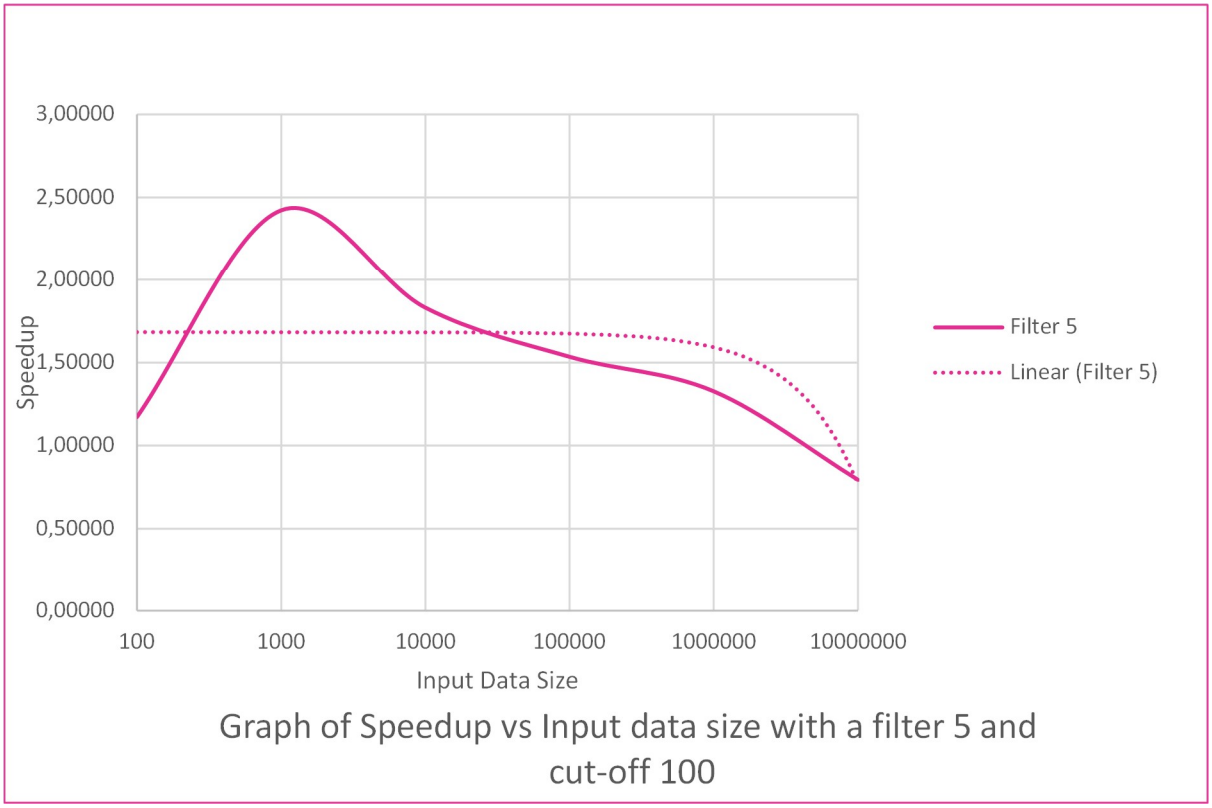


Figure 2

In figure 2 we observe the speedup across various input data sizes with a filter of 5 and a sequential cut off of 100. This graph is useful to us to show which sizes of data run better with the program at this cut off point. We can observe here that the 1000 data set exceeds the expected ideal speedup case (this is likely due to other factors influencing a slightly skewed result). This tells us that this filter works well for a data set size of 1000 but begins to show inefficiencies at sizes greater than that. One of the most important factors of this graph is the general trend line which is constant at a speedup of about 1,7 until we reach much higher datasets. This reflects the real world expected output of a speedup of 1,8. The program becomes much less efficient at much larger datasets, however.

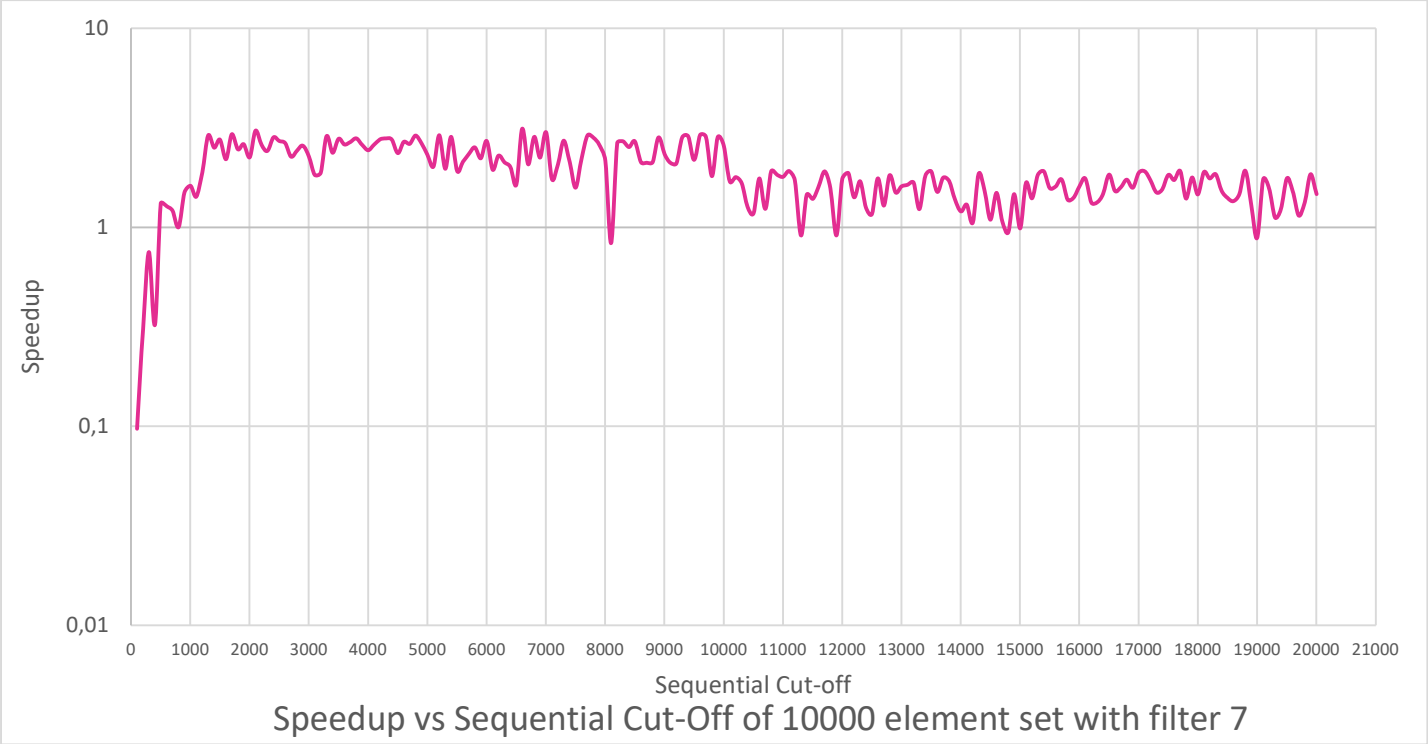


Figure 3

In figure 3 we observe a graph of speed up across various sequential cut offs. This was all done on the sample dataset with 10000 elements in it and a filter size of 7. What we observe in the graph is very little speed up at the lower cut-offs until about 1500, at that point it begins to level out and at a constant speed up of about 2 and then dips down a speedup closer to 1 after 10000. From this we can see that it aligns with what was expected as the program takes finds an optimal cut-off around 1500 and then drops to almost the same speed at the sequential version when it reaches the cut-off with the same number of elements as the dataset.

Discussion

The results of running tests on this data show that it is indeed worth using parallelization to compute data with a median filter. However, at small data sets the parallelisation doesn't affect the speed much. The speedup that was produced across many datasets is within range of what was expected on the machine that the data was tested on. The 2-core machine produced optimal speedup results of around the expected value of 1,8. The larger datasets especially benefit from parallelisation to handle their large quantities of data at the same time, this of course reaching its best case when the optimal sequential cut-off is reached. The data sets with elements of 1000-100000 run especially well on the machine when they are given a filter size of 13-17. In these conditions they run close to the 1,8 expected values. This is likely because the computer doesn't have to handle too much data or divide too much so a very quick result is produced. While the program produced speedup that was above the expected value, these values should be treated as outliers as other conditions likely affected the result of the tests and therefore produced behaviour that was not expected. The maximum speedup value obtained therefore was 1,99. This value is very close to the ideal case of 2 as the speedup. Each dataset has a different optimal sequential cut-off but for the values that I tested I can determine the best cut-offs for a data size of 10000 is 1500 and the best cut off for a dataset of size 100 is 100. Because the second set is so small, the sequential and parallel programs are very close in output times as the parallelisation doesn't affect the result much. Based on this we can say that the optimal number of threads for the 10000 datasets would be 3000 and for the 100 dataset it would be 200. We can also see an interesting trend in most of the recorded data, the first element tested usually takes longer than the rest of the trials. This is most likely due to the core getting faster at computing the same thing repeatedly as it expects the same orders to come through again. Therefore, the initial longer time could be observed as the CPU learning and warming up to the instructions that it is given.

Conclusion

To conclude we can determine a few things about median filtering and parallel programming. Firstly, we know that median filtering does in fact benefit from parallel programming as the data produced speedup that was close to what was expected on the machine. We also determine that each data size has its own unique optimal cut off point at which it handles the data the fastest. We also observe that the smaller the dataset, the less of an impact parallel programming has on the speedup. Parallel programming is a powerful tool that can be used to greatly decrease the time that it takes to perform tasks on large pieces of data and this project has taught me many new techniques and solutions to problems that I wouldn't have been able to solve before.