Assignment 1: Parallel Programming -Median Filter Method

Introduction:

The aim of this project was to create a sequential and parallel (multi-threaded) nonlinear digital filtering technique known as median filtering. This median filter method is used to clean data for later processing. Although the code was a large part of this assignment, the ultimate goal of this project was to understand and learn about the difference between sequential programming on processors vs parallel programming on processors.

The testing of the parallel and sequential program was done across a range of data array sizes and number of threads which is indirectly determined by the sequential cutoff in the recursive parallel algorithm. Timing was done 5 times using the System.currentTimeMillis to carry out the measurements. The code was tested on 2 different machine architectures. One being windows machine while the other being ubuntu running on the same machine through a virtualbox. To ensure that timing was accurate System.gc() was called to minimize the likelihood that the background garbage collector would not run during the execution blocks of both the sequential and parallel algorithms.

The parallel algorithm used implemented a divide-and-conquer method by extending the recursive action class which is used in combination with Java’s built-in ForkJoin framwork. This framework allows coders to create a pool of recursive threads which execute simultaneously, where the number of threads being executed simultaneously is determined by the number of processors. The parallel algorithm is simply a basic recursive if else statement.

The ideal speed up on the quad-core machine that the program was coded on is 4x faster than the sequential runtime of the algorithm. Unfortunately parallelization comes with costly overheads that greatly reduce the speedup of the parallel algorithm.

Method:

To understand the parallel alogrithm we must first go through the sequential algorithm. The sequential algorithm starts by initializing a Double array that is the same size as the input array list size. The benefit of intializing an array instead of an arraylist is that the array has values equal to null while the arraylist is simply initialized and one has to add values from the start of the array. The code then loops over the input list from start to finish. A temporary arralist is created to store the median windows. Within the for loop there is an if-else block. The if statement checks to see if the index of the value is outside the border ((filter-1)/2). If the value is outside the border it simply places the value, with its corresponding index (outputArray[i]=inputList.get(i)) into the output array. The array with all values initialized to null allows us to use the same index to place the value into the outputArray. The else catches those values within the borders. Within the else block, a for loop loops from the start of the median window to the end window. It adds these values to a temporary array. This temporary array then uses the sort method within the inherited collection method to sort the temporary array in ascending order. The middle value of the temporary array is then the median which is placed in the outputArray at its corresponding index. The temporary array is then cleared outside the else statement to ensure that the temporary array does not accumulate values.

The fork/join framework provides coders with an easy way to solve recursive problems in parallel. The fork/join framework is implemented through its subclass RecursiveAction which does some action and has a return field of void. RecursiveAction breaks down the work into multiple threads that, when small enough, performs some task. The RecursiveAction class comes with a method (compute()) that must be overriden. It is within this method that the threads are recursively made. The overridden compute follows the same structure as a standard recursive method. An if statement that will be executed if the work has been divided into a small enough portion, and an else block that creates new threads of the same parallel class. The if statement for the parallel median filter has a cutoff that is called the sequential cutoff. The sequential cutff essentially says that if the work is smaller than a certain size, then the work must be done using the sequential algorithm. Within the else block, a left and right thread are created splitting up the array list in half until the the length of the halved arrays is shorter than the sequential cutoff. The most useful part of the Fork/Join framework is the the fork() and join() methods. The fork method recursively breaks the task into smaller parts until it is small enough to be completed asynchronously. The join method of the RecursiveAction class waits until every subtask that has been recursively created has been executed.

The threads that have been recursively created are “stored” within the ForkJoinPool which oversees how the threads are executed.

In order to validate the algorithm, I needed to time how long it took the sequential algorithm to sort an input file in comparison to the parallel algorithm. I started with a fixed filter size and a small array size. The sequential algorithm took quicker to perform median filtering on the data. This is expected as the cost of filtering the input list in parallel comes with costly overheads. So to validate that the parallel algorithm is in fact faster than the sequential I increased the size of the input array list. After increasing the size of the array list and running both algorithms the parallel was on average 2x times faster than sequential. This shows that my parallel method did in fact speedup the median filtering.

The timing was done using System.currentTimeMillis(). A garbage collector was run before the sequential and parallel algorithms ran to ensure that the time given was as accurate as possible. The speedup was calculated by dividing the time it took the sequential algorithm to run by the time it took the parallel algorithm to execute. The maximum speed up on a quad core machine assuming no hyper-threading is taking place is 8x times faster.

I tested my code on 2 different machine architectures. The first was an HP Envy laptop running Windows 10 with the following processors: Processor Intel(R) Core(TM) i7-4700MQ CPU @ 2.40GHz, 2401 Mhz, 4 Core(s), 8 Logical Processor(s).

The seconds machine was on the same laptop except through virtualbox. This virtualbox has 1 core and is running Ubuntu.

Although the laptop has 4 cores it uses hyperthreading which allows it to reach speedups of close to 8x.

Results:

Quad-Core

Filter size: 3
Array size: 100
Sequential Cutoff: 20

Sequential	0.001	0.002	0.001	0.001	0.001
Parallel	0.003	0.004	0.003	0.004	0.003
Speedup	0.33 (Slower)	0.5 (Slower)	0.33 (Slower)	0.25 (Slower)	0.33 (Slower)

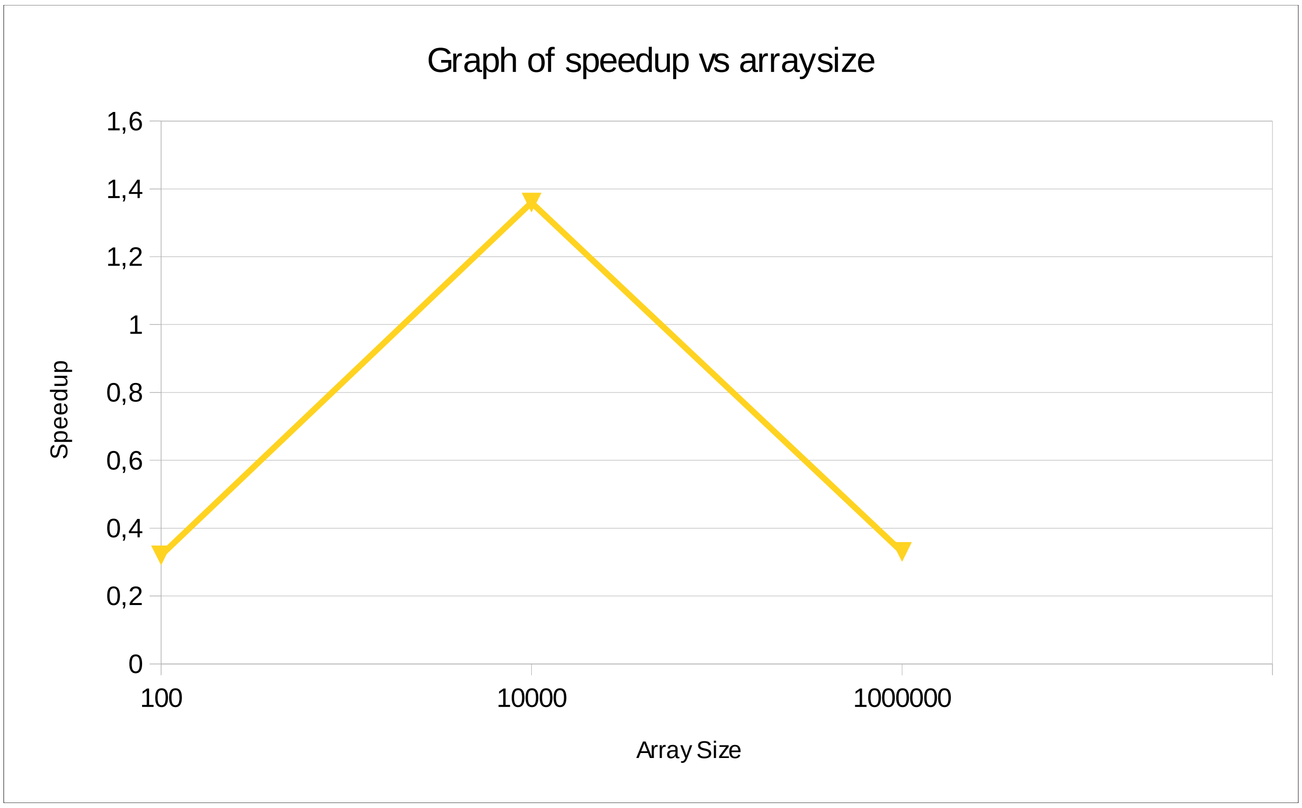
Filter size: 3
Array size: 10000
Sequential Cutoff: 20

Sequential	0.019	0.021	0.025	0.02	0.018
Parallel	0.016	0.011	0.017	0.016	0.018
Speedup	1.19 (Faster)	1.91 (Faster)	1.47 (Faster)	1.25 (Faster)	1 (Same)

Filter size: 3
Array size: 1000000
Sequential Cutoff: 20

Sequential	0.15	0.145	0.164	0.144	0.16
Parallel	0.45	0.45	0.491	0.436	0.447
Speedup	0.33 (Slower)	0.32 (Slower)	0.33 (Slower)	0.33 (Slower)	0.35 (Slower)

These results reveal that parallel will speed up as input array size increases but in order to continue speeding up the sequential cutoff and filter size will have to vary with input size.


Graph of speedup vs arraysize
Speedup
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
100
10000
1000000
Array Size

The following results will display the same variables except sequential cutoff will vary.

Filter size: 3
Array size: 100
Sequential Cutoff: 20

Sequential	0.001	0.002	0.001	0.001	0.001
Parallel	0.003	0.004	0.003	0.004	0.003
Speedup	0.33 (Slower)	0.5 (Slower)	0.33 (Slower)	0.25 (Slower)	0.33 (Slower)

Filter size: 3
Array size: 10000
Sequential Cutoff: 4000

Sequential	0.018	0.022	0.025	0.023	0.019
Parallel	0.008	0.012	0.016	0.007	0.008
Speedup	2.25 (Faster)	1.83 (Faster)	1.56 (Faster)	3.29 (Faster)	2.38 (Faster)

Filter size: 3
Array size: 1000000
Sequential Cutoff: 200000

Sequential	0.022	0.015	0.017	0.017	0.026
Parallel	0.015	0.016	0.012	0.012	0.014
Speedup	1.46 (Faster)	1.07 (Faster)	1.42 (Faster)	1.42 (Faster)	1.85 (Faster)

We see that the highest speedup was achieved with an array size of 10000. The parallel algorithm still sped up the 1000000 size array but just not to the same extent. This suggests that in order to reach maximium speeup both filter size and sequential cutoff needs to be increased as input size increases.

Filter size: 3
Array size: 100
Sequential Cutoff: 50

Sequential	0.001	0.001	0.001	0.001	0.002
Parallel	0.005	0.006	0.003	0.003	0.006
Speedup	0.2 (Slower)	0.17 (Slower)	0.33 (Slower)	0.33 (Slower)	System.out.println("Sequential timing: " + time+"s");0.5 (Slower)

Filter size: 333
Array size: 10000
Sequential Cutoff: 50

Sequential	0.101	0.092	0.091	0.1	0.084
Parallel	0.082	0.085	0.081	0.086	0.076
Speedup	1.23 (Faster)	1.08 (Faster)	1.07 (Faster)	1.16 (Faster)	2.38 (Faster)

Filter size: 3333
Array size: 1000000
Sequential Cutoff: 50

Sequential	37.764	37.03	36.11	41.97	33.93
Parallel	17.64	17.85	16.55	16.27	15.72
Speedup	2.14 (Faster)	2.09 (Faster)	2.18 (Faster)	2.58 (Faster)	2.16 (Faster)

These results show that a constant sequential cutoff, and an increasing input array size, an increasing filter size has a faster speedup as array size increases than increasing the sequential cutoff while keeping filter size constant. This is expected as a larger filter size creates a larger median window that needs to be sorted. The sequential thus takes increasingly longer to sort this growing window in comparison to the parallel sorting.

On this multicore machine, it is shown that parallelization is definitely worth it. As both array size and filter size increase, solving the problem sequentially becomes increasingly costly whereas the parallel algorithm reaches speedups of up to 3.5x faster.

The following results are used to identify at what array size the parallel method starts speeding up the sorting:

Filter size: 50
Array size: 210
Sequential Cutoff: 50

Sequential	0.004	0.005	0.005	0.003	0.003
Parallel	0.005	0.006	0.008	0.004	0.004
Speedup	1.23 (Slower)	0.83 (Slower)	0.62 (Slower)	0.75 (Slower)	0.75 (Slower)

Filter size: 50
Array size: 230
Sequential Cutoff: 50

Sequential	0.003	0.005	0.004	0.005	0.004
Parallel	0.004	0.005	0.004	0.004	0.003
Speedup	0.75 (Slower)	1 (Same)	0.62 (Same)	0.75 (Faster)	1.33 (Faster)

At a constant sequential cutoff of of 50 and constant filter size of 50, the parallel program begins to speedup when the data set size is roughly equal to or more than 230. Therefore the data set size range that this parallel program starts performing well starts at 250 and continues to infinity.

The following results are used to show what the range of filter sizes that

Filter size: 50
Array size: 230
Sequential Cutoff: 50

Sequential	0.003	0.005	0.004	0.005	0.004
Parallel	0.004	0.005	0.004	0.004	0.003
Speedup	0.75 (Slower)	1 (Same)	0.62 (Same)	0.75 (Faster)	1.33 (Faster)

The maximum speedup I obtained was 3.29. This 3.29 is compared to the unobtainable speedup of 8x faster.

Single-Core

Filter size: 3
Array size: 100
Sequential Cutoff: 20

Sequential	0.001	0.002	0.003	0.001	0.001
Parallel	0.003	0.004	0.002	0.004	0.004
Speedup	0.33 (Slower)	0.5 (Slower)	0.67 (Slower)	0.25 (Slower)	0.25 (Slower)

Filter size: 3
Array size: 10000
Sequential Cutoff: 20

Sequential	0.016	0.015	0.017	0.011	0.007
Parallel	0.022	0.019	0.017	0.021	0.024
Speedup	0.72 (Slower)	0.79 (Slower)	1 (Slower)	0.52 (Slower)	0.29 (Slower)

Filter size: 3
Array size: 1000000
Sequential Cutoff: 20

Sequential	0.116	0.155	0.117	0.125	0.112
Parallel	0.248	0.25	0.315	0.194	0.215
Speedup	0.47 (Slower)	0.62 (Slower)	0.37 (Slower)	0.64 (Slower)	0.52 (Slower)

As expected, the parallel algorithm exeuted on a single-core machine has no speedup and is always slower than its sequential counter-part because threads cannot be executed in parallel. The results show that as input size increases the parallel algorithm never speeds up the filtering process.

Therefore it is only worth it to use the parallel method on a machine with multiple cores so that the threads can actually be executed simultaneously.

Conclusions

Parallel algorithms are extremely useful in utilising the full processing power of a computer. Unfortunately the speedup obtained from the parallel algorithm in comparison to the sequential does not come close to the perfect speedup.

The speedup of parallel algorithms are heavily dependant on the variables used in the method. Thus it is requires a lot of testing to determine exactly what range of data set sizes, filter sizes and sequential cutoffs should be filtered in parallel.

The parallel median filter method is definitely worth it for this project as sequential filtering becomes increasingly expensive and for large data sizes it is simply not a viable filtering method.