Name: Fardoza Tohab Student number: THBFAR002 Date: 23 August 2021

CSC2002S: ASSIGNMENT 1

Parallel Programming with the Java Fork/Join framework: 1D Median Filter

INTRODUCTION

Median filtering is a nonlinear digital filtering technique that is often used to remove noise from a data set. Noise reduction is usually a pre-processing step that improves or "cleans" the data in preparation for later processing. In this method, a median filter slides over the data item by item, generating a new data set, where each item is replaced by the median of neighboring entries. The aim of this assignment is to simulate this process and compare the speed up using two different techniques of programming namely a sequential method and a parallel method using the Java Fork/Join framework to parallelize the median filter operation. The expected speedup is measured by: speedup = time for 1 process/time for *p* processes

= T1/Tp

The ideal speed up for my machine is approximately 2.

The expected speedup for this experiment is that there will be a significant increase in the speedup as more processes are used. In other words, the parallel implementation must be faster than the sequential implementation of the median filter operation.

(Refer to the method section for more details on the parallel algorithms).

METHOD

SEQUENTIAL

The sequential approach for the median filter operation consists of a MedianFilter.class with a static method (median) that calculates the median given a subarray and a filter size. SubArrays are created using the Arrays.copyOfRange() method and passed to the median method.

PARALLELIZATION

In this assignment, to parallelize the median filter process, I used two classes: MFArray.class and ParallelMF.class.

MFArray.class

This is where the divide and conquer algorithm is implemented to parallelize the median filter operation. It extends RecursiveAction from the java.util.concurrent library. It takes in a set of parameters namely the starting, ending, raw data and filter size. I overrided the compute() function of type void where recursion is used. The base case basically says that when the sequential cut-off is reached, the program should transition into doing the calculations sequentially. (If the size of the list is less than the threshold, the median filter operation is computed directly, and the result is printed). Otherwise, the list is divided into two MFArray

tasks (left and right). Then, the first task is forked while the result of the second is computed (this is the recursive call until the condition of the base case is fulfilled) and after that, we wait for the first task result.

ParallelMF.class

This class has the main method where I read in from the input file and split and store the contents into an array. I also imported the ForkJoinPool class from java.util.concurrent library and created a static instance of it. The data is then passed to an MFArray instance together with the filterSize, window and length of array, which in turn is passed to a new ForkJoinPool instance to be executed.

My algorithm for finding the median filter is correct since I have implemented it in both the sequential and parallelized version and they both output the same results (this can be tested if need be). I also created a separate text file with the sample input given in the assignment brief and checked for both versions if they have the same output as given in the assignment brief.

TESTING PERFORMANCE OF ALGORITHMS

I created a Performance.class to measure the time taken for both the sequential and parallelized implementation of the median filter operation. It takes in the filter size, and raw data. I created a main method to read in the data and two separate methods to measure the timing for each algorithm namely: testForkJoin and testSequentially. I timed the algorithms with different array input sizes. I did this manually on jGrasp where I used a scanner object to take in the parameters that I wanted to keep constant and the parameters I wanted to change and got the timing as output. I ran each entry 20 times and used the 20^{th} run time to plot the graph. I measured the speedup by dividing the sequential time over the parallel time and used this to plot three different graphs namely: Data size vs speedup, Number of threads vs speed up and filter size vs speed up.

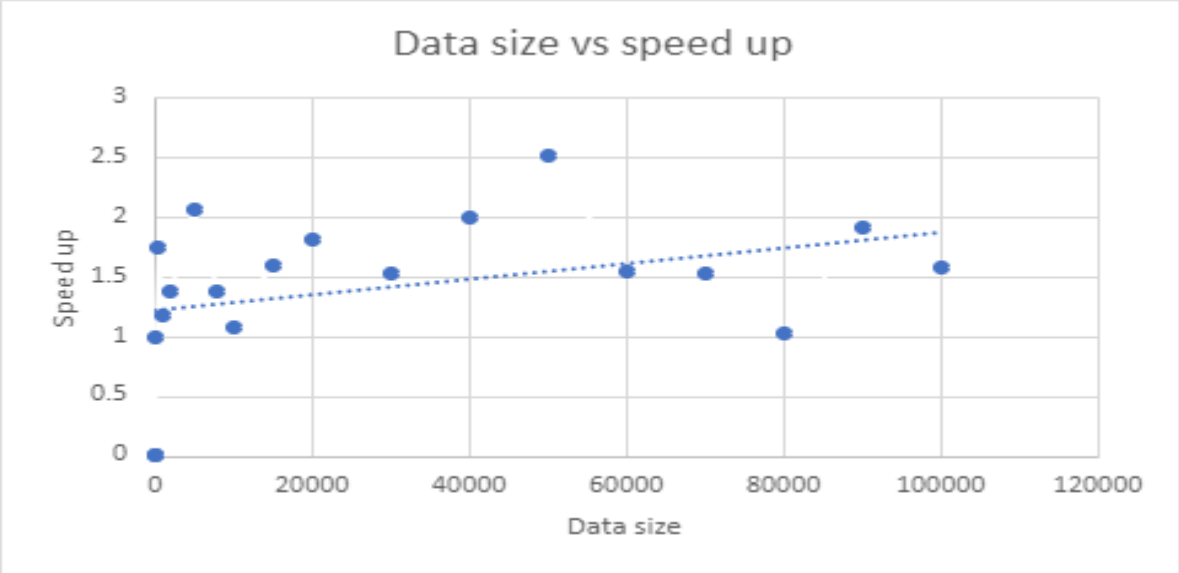
The machine architecture I used has 2 cores and 4 logical processors. This has limited the amount of memory that I had available and thus could not test my programs against very large data sets.

RESULTS

Keeping number of threads constant (100) and varying data size (Using a filter size of 5).

Data Sizes	Number of Threads	Filter Size	Parallel Time (ms)	Serial Time (ms)	Speed up (ms)
5	4000	5	0	0	0
10	4000	5	0	15	0
100	4000	5	3	3	1
500	4000	5	4	7	1.75
1000	4000	5	11	13	1.18
2000	4000	5	13	18	1.38
5000	4000	5	15	31	2.07
8000	4000	5	8	11	1.38

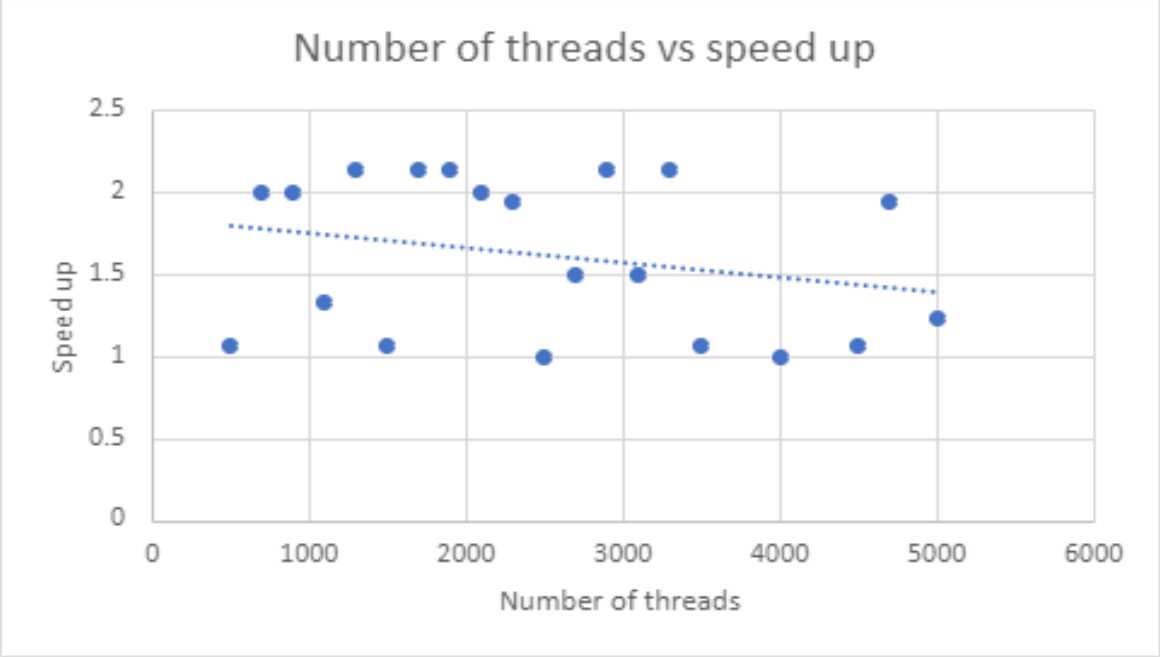
10000	4000	5	15	16	1.07
15000	4000	5	15	24	1.60
20000	4000	5	16	29	1.81
30000	4000	5	31	47	1.52
40000	4000	5	31	62	2.00
50000	4000	5	31	78	2.52
60000	4000	5	52	80	1.54
70000	4000	5	62	94	1.52
80000	4000	5	62	125	1.02
90000	4000	5	78	150	1.92
100000	4000	5	86	136	1.58
1000000	4000	5	1109	1375	1.24



Keeping data size constant and varying number of threads (Using a filter size of 5).

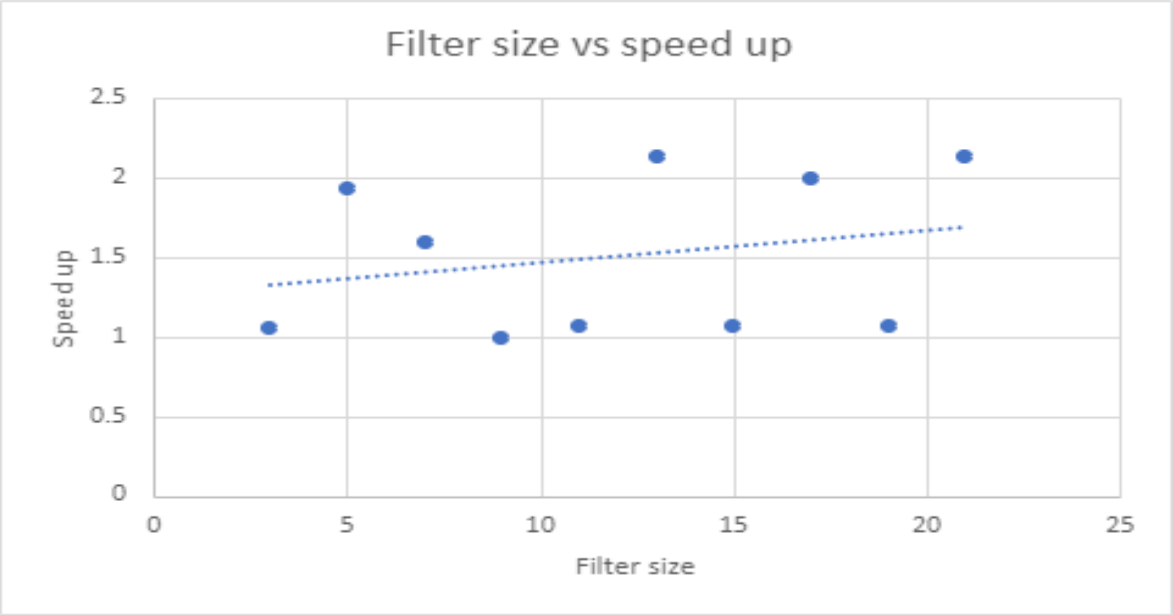
Data Sizes	Number of Threads	Filter Size	Parallel Time (ms)	Serial Time (ms)	Speed Up (ms)
10000	500	5	15	16	1.07
10000	700	5	8	16	2.00
10000	900	5	16	32	2.00
10000	1100	5	12	16	1.33
10000	1300	5	15	32	2.13
10000	1500	5	15	16	1.07
10000	1700	5	15	32	2.13
10000	1900	5	15	32	2.13
10000	2100	5	8	16	2.00
10000	2300	5	16	31	1.94
10000	2500	5	16	16	1.00
10000	2700	5	16	24	1.50

10000	2900	5	15	32	2.13
10000	3100	5	16	24	1.50
10000	3300	5	15	32	2.13
10000	3500	5	15	16	1.07
10000	4000	5	16	16	1.00
10000	4500	5	15	16	1.07
10000	4700	5	16	31	1.94
10000	5000	5	13	16	1.23



Keeping number of threads and number of elements constant and varying filter size.

Data Sizes	Number of Threads	Filter Size	Parallel Time (ms)	Serial Time (ms)	Speed Up (ms)
10000	4000	3	15	16	1.06
10000	4000	5	16	31	1.94
10000	4000	7	15	24	1.60
10000	4000	9	16	16	1.00
10000	4000	11	15	16	1.07
10000	4000	13	15	32	2.13
10000	4000	15	15	16	1.07
10000	4000	17	8	16	2.00
10000	4000	19	15	16	1.07
10000	4000	21	15	32	2.13



DISCUSSION

- Varying the data and keeping everything else constant. (Graph)

From the results I've noticed that there is a significant increase in the speedup as I increase the data size. The parallel program almost always outperformed the sequential program. However, I had to run my programs at least 20 times using a for loop to get the parallel program to run faster than the sequential program.

- Varying number of threads (sequential cutoffs) and keeping everything else constant. (Graph)

There was a significant decrease in the speed up as I increased the sequential cutoff. Usually, a low threshold impacts performance due to the overhead of creating many small subtasks. This is probably because the thresholds I used to conduct the experiment were too small compared to the data size used and thus created a decrease in the speedup of my programs.

- Varying the filter size and keeping everything else constant. (Graph)

Increasing the filter size had an increase in the speedup of the programs. Increasing the filter size increases the size of the subarrays and parallel programs are ideal when considering huge data sizes. Thus, resulting in a speedup since multiple threads tackle each subarray and produce an answer faster.

The machine architecture I used had only 2 cores with 4 logical processes. This limited the amount of memory I had available and could not test for large data sizes. Using a machine

architecture with more cores helps with speed up since there are more cores available to do work and the tasks can be carried out much faster.

On a more general note, it is ideal to use parallel programming for the median filter operation only if we are dealing with large data sizes. On a rather relatively small data size, the two programs performed almost exactly the same, however, the parallel program outperformed the sequential when it came to large data sizes. So, if the task is small, it is better to do it sequentially.

My parallel program performed well for larger data sizes ranging from 10 000 to 1000000 elements and filter sizes ranging from 13 to 21.

The maximum possible speedup for my parallel approach is approximately 2.52. The ideal speed up is 2. Differences in the two speedups could be caused by the thread overhead of creating small subtasks.

There are a few sequential cutoffs that give maximum speed up. 1300, 1700, 1900, 2900, 3300 all give a maximum speed up of 2.13 when varying the sequential cutoff. However, when varying the data sizes 4000 becomes an ideal sequential cutoff.

The ideal number of threads is 1 thread per core. So, on this machine architecture the optimal number of threads is 2.

CONCLUSIONS

- For larger data sizes or filter sizes, the speed up increases significantly when the operation is parallelized.
- If the task is simple, it is better to do it sequentially rather than parallel since the cost of splitting and queuing the tasks adds up to exacerbate the runtime.
- A low sequential cut-off relative to the data size impacts performance due to the overhead of creating many small subtasks.