CSC2002S
Assignment 1: Parallel Programming with the Java Fork/Join framework: 1D Median filter

Zach Nudelman

NDLZAC001
UCT

I. Introduction

A. Problem

A Median Filter is a technique used to remove noise from a set of data. It iterates through the data items and replaces each item with the median of itself and its neighbouring entities. The number of items used to find the median is know as the filter size. This problem is quite time-consuming since each data item must be processed.

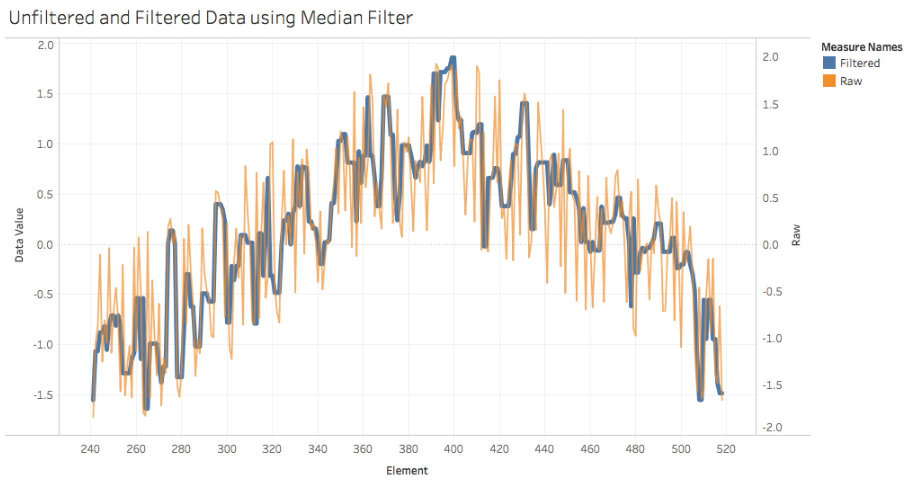


Fig. 1: The effects of Median Filtering on 1D Data with a Filter Size of 3

B. Aim

The aim of this assignment is to speed up this problem through parallelizing the algorithm. This parallel algorithm will effectively be a parallel map since each thread will perform an operation on each element independently (or rather on subarrays) [1]. In the Java ForkJoin framework the expected speedup for this type of problem is $O(n)/O(logn)$ .Thus the aim of this project is to create an exponential speedup of the parallel algorithm relative to the sequential one.

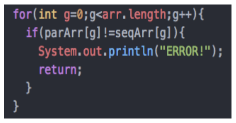
II. Methods

A. The Parallel Algorithm

As stated in the Aim subsection, the parallelization of this algorithm resulted in a parallel map as follows:

1) The method was called which invoked the main thread.
2) Two new threads objects were invoked
   a) The left thread accepted the beginning index and the middle index of the original array
   b) The right index accepted the middle index and the end index of the original array
   c) The left thread object called the *fork()* method to invoke a new thread
   d) The right thread object called the *compute()* method to use the current thread to work
   e) The left thread called the *join()* method to ensure that the thread had completed its work before returning
3) For every new thread created, if the subarray was larger than some predefined Sequential Cutoff, the thread forked and the processed recursed (from 2).
4) Once the Sequential Cutoff was reached, a similar algorithm as the sequential one was invoked as explained in I.

The parallel algorithm was validated by simply iterating through the elements, checking if each element was equal to that within the sequential method's output array and if there was some discrepancy returning and printing "ERROR!" to the console.



```
for(int g=0;g<arr.length;g++){
  if(parArr[g]!=seqArr[g]){
    System.out.println("ERROR!");
    return;
  }
}
```

B. Algorithm Timing

To time the different algorithms, two methods *tick* and *tock()* were used. *tick()* would save the current system time as a float and *tock()* would return the difference between the subsequent current system time and the previous one (divided by 1000 to return seconds and not milliseconds). Before each call of *tick()*, the Java garbage collector *System.gc()* was called.
For the sequential algorithm, timing began before creating the filter object and terminated after the *filter()* method returned. Similarly, the parallel algorithm was timed by calling *tick()* before the *ForkJoinPool()* was invoked and terminated upon the *invoke()* method's return.
The experiment used a Java class *ExperimentMain* which ran each algorithm 10 times, and found the average of the last 9 runs of each algorithm (since the first run would have extrapolated the data). The average time was then saved in a file along with the number of processors, the filter size, the Sequential Cutoff and the size of the array (number of elements to filter). A bash script was developed which ran a test for increasing values of:

1) The sequential cutoff from 500 to 5000 in increments of 500
2) The filter size from 3 to 21 in increments of 2
3) The size of the Array to filter from 100,000 to 1,000,000 in increments of 100,000

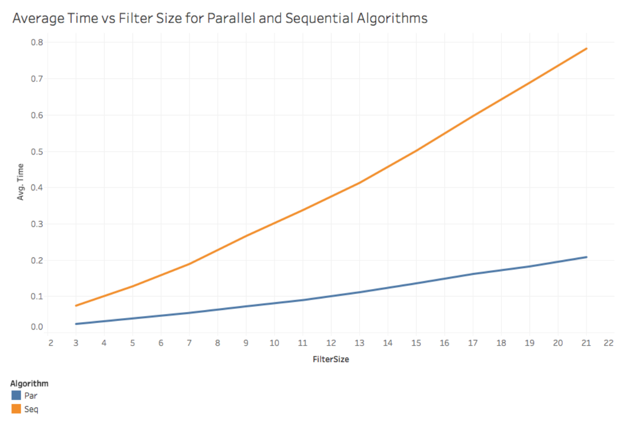
using nested loops. During testing, no other applications were open and the machine had come out of a fresh restart. For the first set of tests my own laptop was used with 4 cores. For the second set, the UCT nightmare server was used with 8 processors. Speedup was tested by dividing the parallel time by the sequential time for each experimental output for each core. I.e.,

$$S/T_p$$

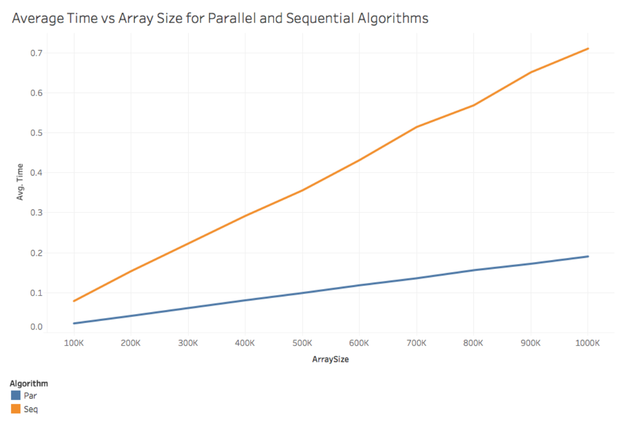
, where S is the runtime for the sequential algorithm and T_p is the runtime for the parallel algorithm. It is expected that the 8 core machine will produce a speedup double the size of the 4 core machine.

III. Results & Discussion

It was found that for varying filter sizes and array sizes, the parallel algorithm outperformed the sequential one in terms of runtime as seen in figures 2a and 2b below.



(a)



(b)

Fig. 2: Average Times for Parallel vs Sequential Algorithms for varying filter and array sizes

Hence, it is worth using parallelization in Java for this particular problem in general. However, as can be seen in the figures above, the speedup was not very significant for very low array sizes (<= 10000) and very low filter sizes(<= 3). In fact, the lowest speedup gained was only $0.45x$ for the 4 core machine and $0.88x$ for the 8 core machine. In both these instances, the filter size was 3 and the array size was 10,000. Additionally, the maximum speedups found was $5.05x$ with a filter size of 11 and an array size of 800,000 for the 8 core machine and $2.56x$ for the 4 core machine with a filter size of 9 and an array size of 500,000. Since this problem fits the Map Pattern, it is embarrassingly parallel and thus there is effectively no limit to how many processors could result in a speedup.

As can be seen in 3 and 4 below, with a greater number of processors, the curves follow the expected speedup of $O(n)/O(logn)$. Additionally, the average, minimum and maximum speedups for the 4 core and 8 core machine are at a 2:1 ratio. I.e., the 8 core machine provides twice the speedup that the sequential algorithm does.

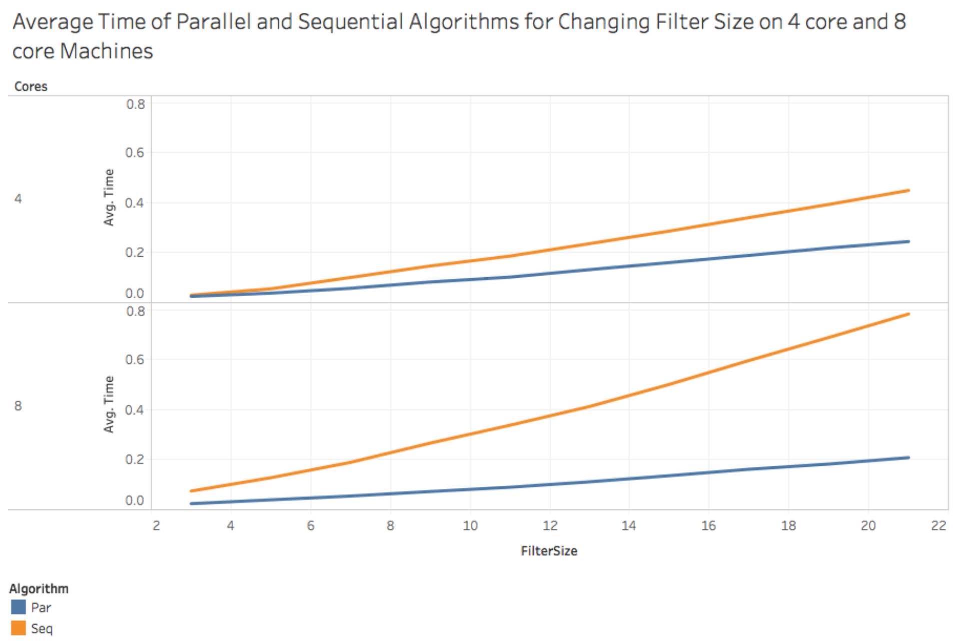


Fig. 3: Average Times for 8Core vs 4Core Algorithms for Varying Filter Sizes

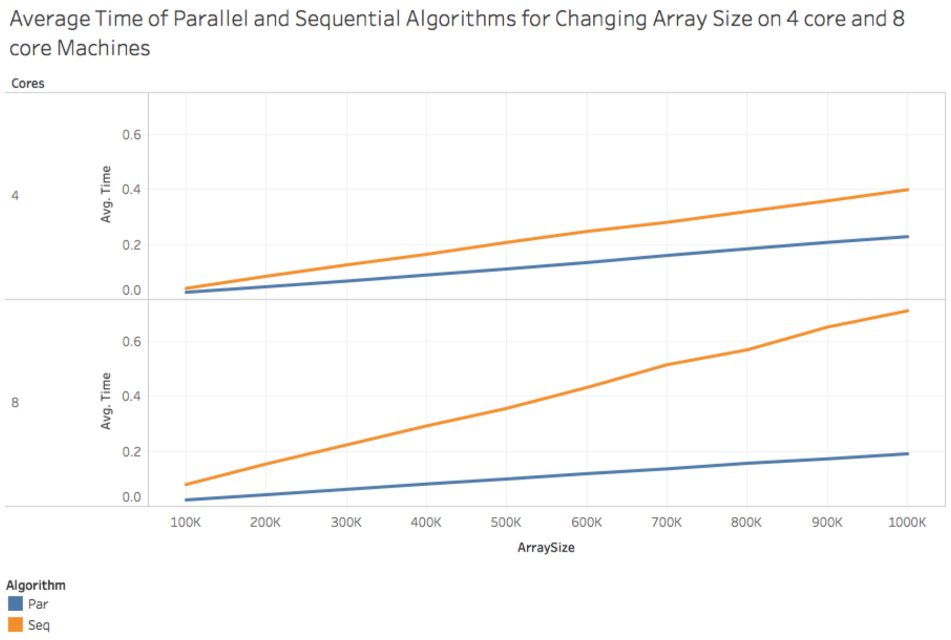


Fig. 4: Average Times for 8Core vs 4Core Algorithms for Varying Array Sizes

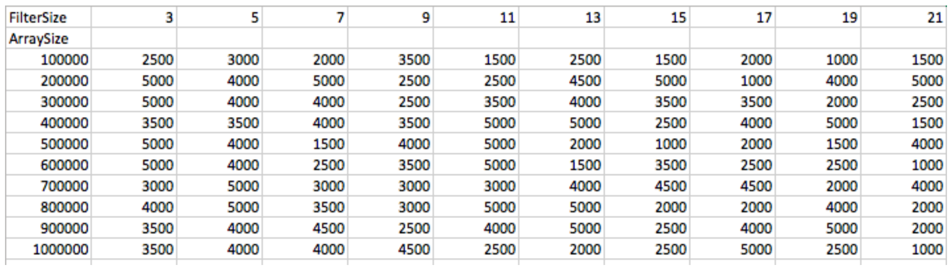
The optimal sequential cutoff and threads depends on the filter size and the size of the array being checked. Appended is a table of optimal sequential cutoffs for the 4 core machine. The number of threads would be a function of the sequential cutoff, the array size and the filter size. For the 4 core machine, the optimal thread count varied between 5 and 7 and for the 8 core machine, between 7 and 9.

IV. Conclusions

In conclusion, Parallelization is very effective in the Median Filtering technique. Particularly for larger array sizes and filter sizes. It can also be concluded that this algorithm is perfectly parallelizable since it follows the map pattern. Thus the speedup will grow linearly with a growing number of processors. These conclusions can be made with moderate certainty since the experiments were performed under strict conditions. Moreover, the relevant correlations could be made even though the 4 core machine was faster than the 8 core machine since in both cases, the parallel algorithm was compared with its respective sequential algorithm's performance.

V. Appendices

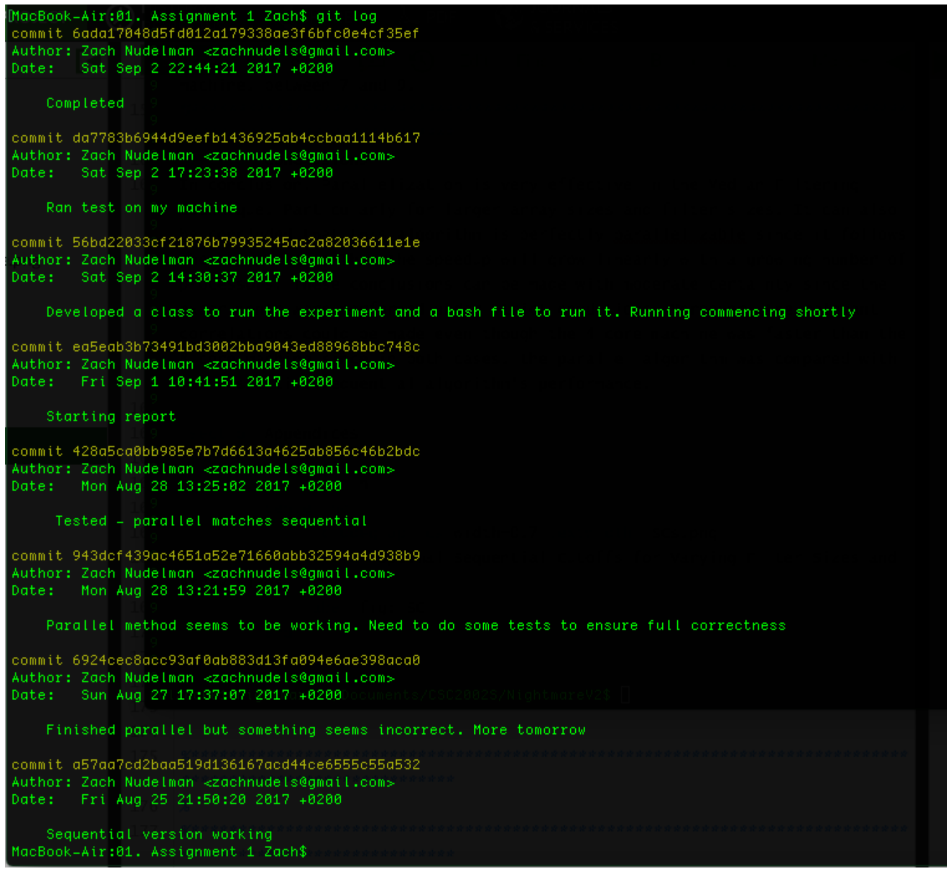
A.



FilterSize	3	5	7	9	11	13	15	17	19	21
ArraySize										
100000	2500	3000	2000	3500	1500	2500	1500	2000	1000	1500
200000	5000	4000	5000	2500	2500	4500	5000	1000	4000	5000
300000	5000	4000	4000	2500	3500	4000	3500	3500	2000	2500
400000	3500	3500	4000	3500	5000	5000	2500	4000	5000	1500
500000	5000	4000	1500	4000	5000	2000	1000	2000	1500	4000
600000	5000	4000	2500	3500	5000	1500	3500	2500	2500	1000
700000	3000	5000	3000	3000	3000	4000	4500	4500	2000	4000
800000	4000	5000	3500	3000	5000	5000	2000	2000	4000	2000
900000	3500	4000	4500	2500	4000	5000	2500	4000	5000	2000
1000000	3500	4000	4000	4500	2500	2000	2500	5000	2500	1000

Fig. 5: Optimal Sequential Cutoffs for Varying Filter Sizes and Array Sizes

B.



```
MacBook-Air:01. Assignment 1 Zach$ git log
commit 6ada17048d5fd012a179338ae3f6bfc0e4cf35ef
Author: Zach Nudelman <zachnudels@gmail.com>
Date:   Sat Sep 2 22:44:21 2017 +0200

    Completed

commit da7783b6944d9eefb1436925ab4ccbaa1114b617
Author: Zach Nudelman <zachnudels@gmail.com>
Date:   Sat Sep 2 17:23:38 2017 +0200

    Ran test on my machine

commit 56bd22033cf21876b79935245ac2a82036611e1e
Author: Zach Nudelman <zachnudels@gmail.com>
Date:   Sat Sep 2 14:30:37 2017 +0200

    Developed a class to run the experiment and a bash file to run it. Running commencing shortly

commit ea5eab3b73491bd3002bba9043ed88968bbc748c
Author: Zach Nudelman <zachnudels@gmail.com>
Date:   Fri Sep 1 10:41:51 2017 +0200

    Starting report

commit 428a5ca0bb985e7b7d6613a4625ab856c46b2bdc
Author: Zach Nudelman <zachnudels@gmail.com>
Date:   Mon Aug 28 13:25:02 2017 +0200

     Tested - parallel matches sequential

commit 943dcf439ac4651a52e71660abb32594a4d938b9
Author: Zach Nudelman <zachnudels@gmail.com>
Date:   Mon Aug 28 13:21:59 2017 +0200

    Parallel method seems to be working. Need to do some tests to ensure full correctness

commit 6924cec8acc93af0ab883d13fa094e6ae398aca0
Author: Zach Nudelman <zachnudels@gmail.com>
Date:   Sun Aug 27 17:37:07 2017 +0200

    Finished parallel but something seems incorrect. More tomorrow

commit a57aa7cd2baa519d136167acd44ce6555c55a532
Author: Zach Nudelman <zachnudels@gmail.com>
Date:   Fri Aug 25 21:50:20 2017 +0200

    Sequential version working
MacBook-Air:01. Assignment 1 Zach$
```

Fig. 6: Git Log

References

[1] D. Grossman, "A sophomoric introduction to shared-memory parallelism and concurrency."