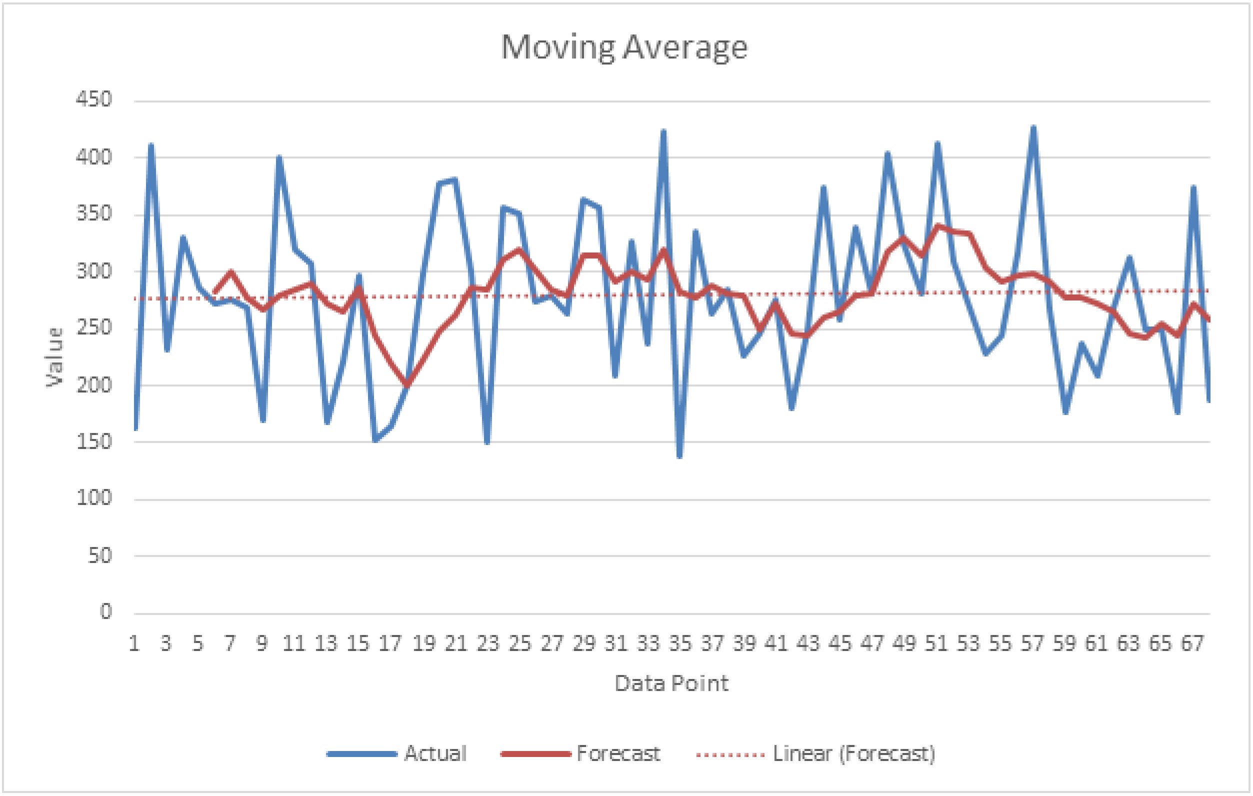
SKSVPX001

Introduction

Median filtering is a nonlinear digital filtering technique that is often used to remove noise from a data set, 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. Few or several data items considered to be filtered are within a filtering window of a certain size.



Aim and predictions

Use the Java Fork/Join framework to parallelize the median filter operation using a divide-and-conquer algorithm, where I filtered parts of the data input at the same time where work is distributed almost equally to unused cores. The naïve approach to achieving the aim is by sequentially going through all the data item-by-item utilizing one core, runnig order O(n) or not all the computer resources and this can be expensive when *n* ise very large, especially if the filter window is large enough. The parallel algorithm is to be correct and faster relative to serial operations, to give an O(log(n)) runtime.

Predictions:

1. Parallel program should be significantly faster than serial version with the same filter size.

2. An increase in filter size, parallel program should be slower approaching serial version pace.
3. A decrease in SEQUANTIAL CUTTOFF should increase speed of parallel program

Methods

Sequential

Designed a sequential median filtering program that takes the following parameters, an array of data items to be operated, filterSize window to utilize in finding median, lastly a lower & upper index on the array to start the filtering process.
A medianFilter method is applied to filter the data according to the given constructor parameters.

```
public SerialFilter(float[] file_data ,int filter_size, int start, int end){}
```

Then a medianFilter method is called and returns a filtered array from lower(start) to upper(end) index is returned.

Parallel

Large data items are divided into smaller ones, which median filtering can be applied to multiple process at the same time. The following compute method parallelizes the program.
The parallelization program constructor takes the same arguments but with one more, the sequential cuttOff, which is a range value within full data items.

```
ParallelFilter(float[] array, int l, int h, int filter_size, int seq_cuttOFF) {
...
...
}

protected float[] compute() {
    if ((hi - lo) <= SEQUENTIAL_CUTOFF) {
        return (new SerialFilter(arr, filter_size, lo, hi)).medianFilter();
    } else {
        ParallelFilter left = new ParallelFilter(arr, lo, (hi + lo)/ 2,
filter_size, SEQUENTIAL_CUTOFF);
        ParallelFilter right = new ParallelFilter(arr, (hi + lo) / 2, hi,
filter_size, SEQUENTIAL_CUTOFF);

        left.fork();
        float[] rightAns = right.compute();
        float[] leftAns = left.join();

        float[] both = Arrays.copyOf(leftAns, leftAns.length +
rightAns.length);
        System.arraycopy(rightAns, 0, both, leftAns.length, rightAns.length);
        return both;
}
```

1. Description of approach

The full data items are inspected whether they are more,less or equal to the sequential cuttoff according to upper(hi) and lower(lo) indx

```
if ((hi - lo) <= SEQUENTIAL_CUTOFF) {
```

If it is less that sequential cuttoff or equal to, then the sequantial(median) filtering is applied to that *piece* of data and a filtered piece is returned by the sequential filtering program,

```
return (new SerialFilter(arr, filter_size, lo, hi)).medianFilter();
```

if it they are more than the sequential cuttoff, the data is cut into 2 half pieces, by creating the *left* and *right* objects.

```
} else {
    ParallelFilter left = new ParallelFilter(arr, lo, (hi + lo)/ 2,
filter_size, SEQUENTIAL_CUTOFF);
    ParallelFilter right = new ParallelFilter(arr, (hi + lo) / 2, hi,
filter_size, SEQUENTIAL_CUTOFF);
```

and the 2 half pieces each run on two separate threads from the main thread where the left object has independant excution [fork()] from the main thread, and the right thread is run on onother separate thread[right.compute()], then the left thread waits for the right thread to finish [left.join()].
and repeat the same process of being inspected(at the top) whether each is more,less or equal to the sequential cuttoff. This is a recursive process.

```
left.fork()
float[] rightAns = right.compute();
float[] leftAns = left.join();
```

since the sequential median filtering program only returns the filtered output from lower and upper index given, i.e if data elements are not on a range to be filtered, then that whole range is not retuned)
The two half pieces returned are then pieced together after being filtered, by combining them together, joining the right piece to the right.

```
float[] both = Arrays.copyOf(leftAns, leftAns.length + rightAns.length);
System.arraycopy(rightAns, 0, both, leftAns.length, rightAns.length);
return both;
```

2. Speedup was measured by dividing the parallel program runtime by the sequential runtime for each experimental input.

3. Validating the Algorithm

    - Debugged the algorithm, inspecting it line by line as to wheather it is taking the correct actions when filtering some input data.
    - Made use of small data inputs, then increased to large data input.
    - Made use of a small data items and confimed by hand(manually) for all filters windows(3 to 21) and varying the sequential cuttOff that the filtering was correct, increased input data items size after some time.
    - Applied median filtering to half(the left or the right) of input data items, e.g when there's 50 data items to filter ranging from zero, I input lower index as 25 and upper as 50, and let the parrallel program filter the upper half(25-50) without changing the lower half, also vice versa.

4. Timing the Algorithm.

    Timing was restricted to the median filter operation (only), excluding the time to read in the file with data input beforehand, and print the cleaned data afterwards, as well as other unnecessary operations.

    Two timing methods were created *tick()* and *tock()*.
    tick() records the current time translated to milliseconds.

    ```
    private static void tick(){startTime = System.currentTimeMillis();}
    ```

tock() also records the current time, but subtracts it from *startTime* so to get the difference, divided by 1000 to return the time is seconds.

```
private static float tock(){return (System.currentTimeMillis()-startTime)
/ 1000.0f;}
```

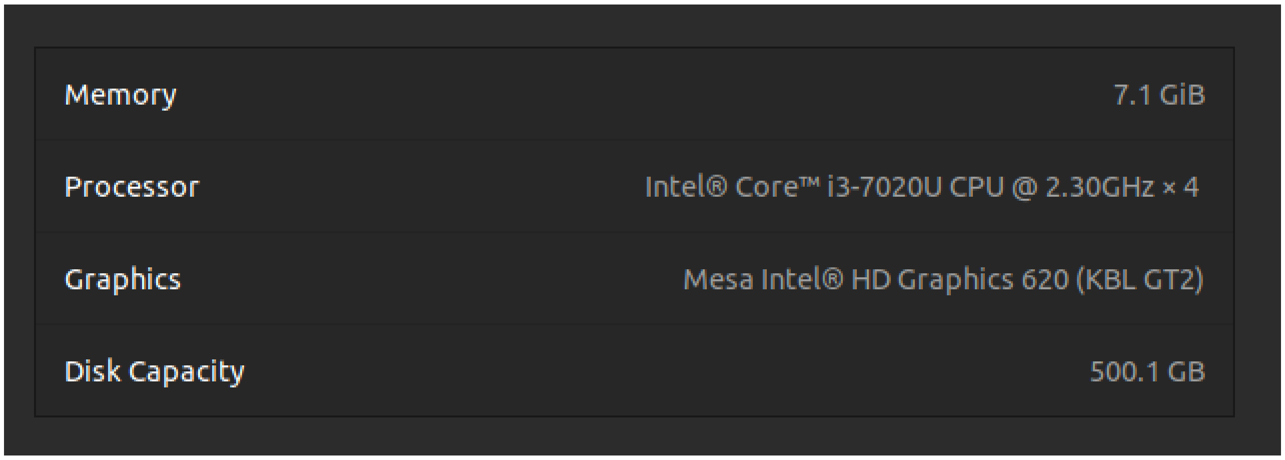
The algorith was run 20 times, finding timing avarage of the 20 runs across a range of data sizes and number of threads (sequential cut-off).

```
for (int indx = 0; indx < 20; indx++) {
    System.gc();
    tick();
    float[] b = toParallelFilter(file_data, filter, seq); //
seq=sequentiallCuttOff
    a = a + tock();
}
a = a / 20;
```

System.gc() is called to minimize the likelihood that the garbage collector will run during the execution.

A *Experiment.java* program ran the algorithm 20 times for a each 5% increments from data input length of seqential cut-off for all the filter windows i.e 3-21.
e.g If data input with 100 items is ran on the algorithm, then a sequential cut-off start at 5% which is 5, and runs the algorithm 20 times for each filter size avaraging timing for that filtersize with that sequential length, then increment sequential with another 5% which will be 10, then again runs the algorithm 20 times for each filter size avaraging timing for that filtersize with that sequential length.

5. Machine architecture.

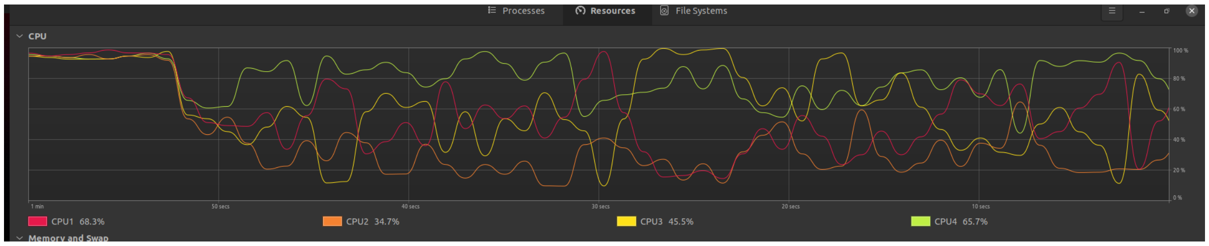


Memory	7.1 GiB
Processor	Intel® Core™ i3-7020U CPU @ 2.30GHz × 4
Graphics	Mesa Intel® HD Graphics 620 (KBL GT2)
Disk Capacity	500.1 GB

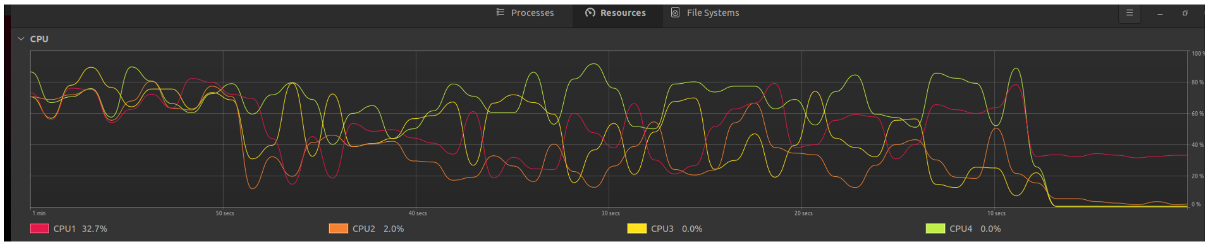
6. Problems encountered.

   1. Nondeterminism, recieved a huge timing difference results for the same data input size, with the same sequential cuttoff and the same filtersize, even after I have warmed up the computer by running warm up test, as much as this is an research assignment, the timing difference should not be too much off, there's a small range of allowed difference. After investigation, my algorithm was not the problem, I found difference in cpu spikes.

      For example *(these images are not original to when i discovered the problem but relate)*.
      At first time running the experiment, I had Visual Studio Code, and Firefox browser(watching YouTube video) running with many tabs open, thus it was using a lot of cpu, and slowing down the the algorithm making it slow.



      Later that day, after powering on my computer I ran the experiment again, with no other program running, the results dramatically changed.

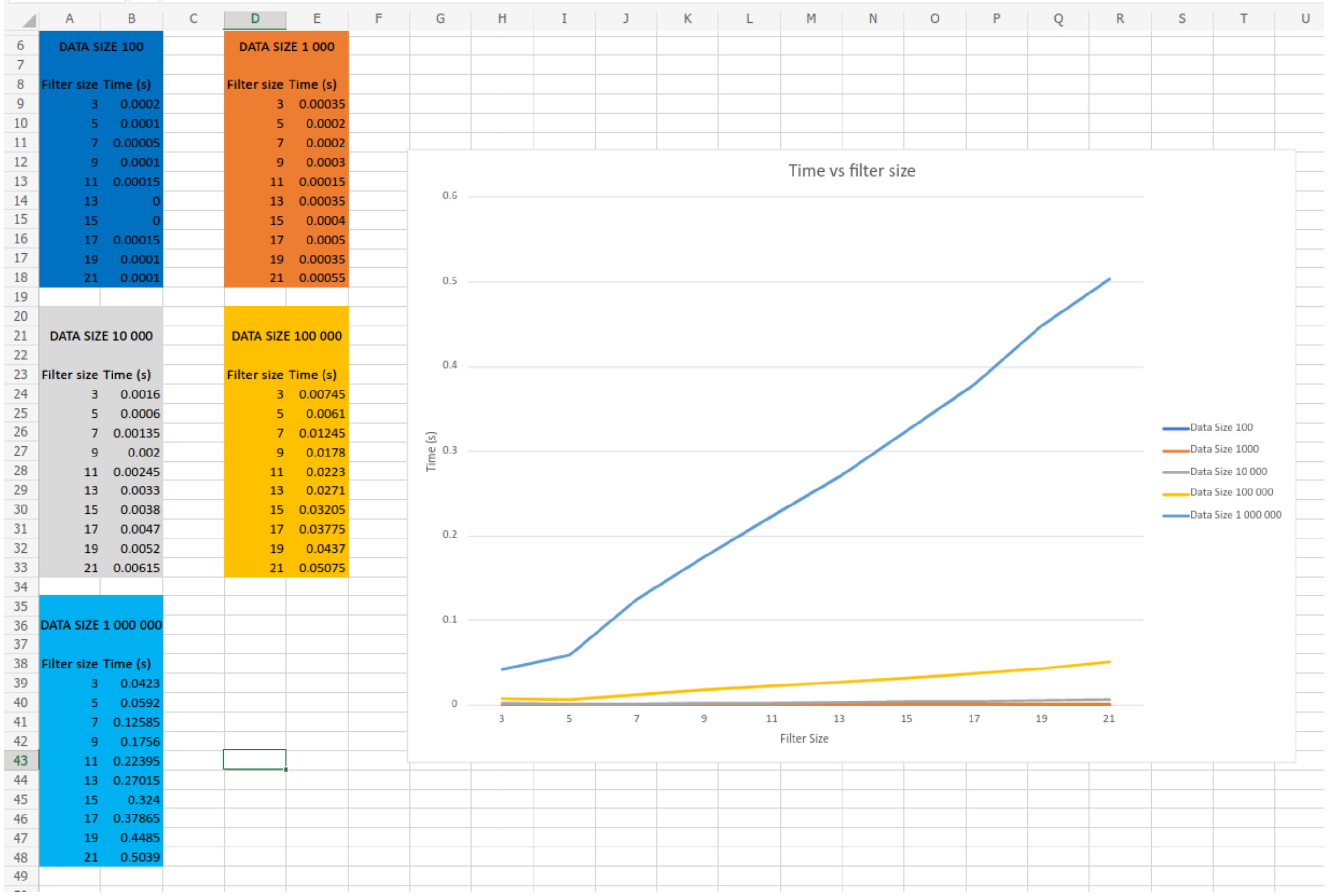


Experimental Results

Serial Program

For every data size, every filter window size that program has been run 20 times and the time of execution has been averaged.

Each data size has a different colour relating to the data table on the left.



DATA SIZE 100

Filter size	Time (s)
3	0.0002
5	0.0001
7	0.00005
9	0.0001
11	0.00015
13	0
15	0
17	0.00015
19	0.0001
21	0.0001

DATA SIZE 1 000

Filter size	Time (s)
3	0.00035
5	0.0002
7	0.0002
9	0.0003
11	0.00015
13	0.00035
15	0.0004
17	0.0005
19	0.00035
21	0.00055

DATA SIZE 10 000

Filter size	Time (s)
3	0.0016
5	0.0006
7	0.00135
9	0.002
11	0.00245
13	0.0033
15	0.0038
17	0.0047
19	0.0052
21	0.00615

DATA SIZE 100 000

Filter size	Time (s)
3	0.00745
5	0.0061
7	0.01245
9	0.0178
11	0.0223
13	0.0271
15	0.03205
17	0.03775
19	0.0437
21	0.05075

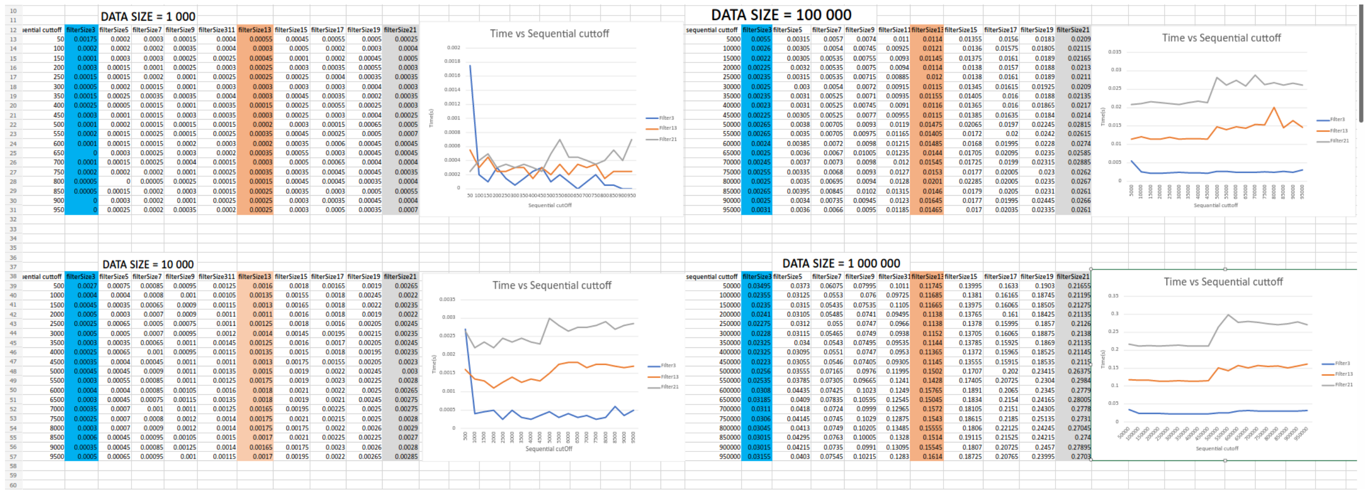
DATA SIZE 1 000 000

Filter size	Time (s)
3	0.0423
5	0.0592
7	0.12585
9	0.1756
11	0.22395
13	0.27015
15	0.324
17	0.37865
19	0.4485
21	0.5039

The extrapolated graphs show that for every data size, an increase in the filtering window increases the runtime of filtering of data.

Parallel Program

The sequential cut-off of every data size has been made of increments of 5% of the full data size, and for every sequential cut-off , every filtering window size has been run 20 times and the time of execution has been averaged, i.e for every sequential cut-off and filtering window size the program has run 20 times, then the timing was avaraged.



DATA SIZE = 1 000

ential cuttoff	filterSize3	filterSize5	filterSize7	filterSize9	filterSize311	filterSize13	filterSize15	filterSize17	filterSize19	filterSize21
50	0.00175	0.0002	0.0003	0.00015	0.0004	0.00055	0.00045	0.00055	0.0005	0.00025
100	0.0002	0.0002	0.0002	0.00035	0.0004	0.0003	0.0005	0.0002	0.0003	0.0004
150	0.0001	0.0003	0.0003	0.00025	0.00025	0.00045	0.0001	0.0002	0.00045	0.0005
200	0.0003	0.00015	0.0001	0.00025	0.0003	0.00025	0.0003	0.00035	0.00055	0.0003
250	0.00015	0.00015	0.0002	0.0001	0.00025	0.00025	0.00025	0.0004	0.00035	0.00035
300	0.00005	0.0002	0.00015	0.0001	0.0003	0.0003	0.0003	0.0003	0.0004	0.0003
350	0.00015	0.00025	0.00035	0.00035	0.0004	0.0003	0.00045	0.00035	0.0002	0.00035
400	0.00025	0.00005	0.00015	0.0001	0.00035	0.00015	0.00025	0.00055	0.00025	0.0003
450	0.0003	0.0001	0.00015	0.0003	0.00035	0.0003	0.00025	0.0003	0.0004	0.00025
500	0.0001	0.0002	0.00015	0.00015	0.00015	0.0002	0.0003	0.00015	0.00065	0.0005
550	0.0002	0.00015	0.00025	0.00015	0.00025	0.00035	0.00045	0.00025	0.0005	0.0007
600	0.0001	0.00015	0.00015	0.0002	0.0003	0.0002	0.00035	0.0006	0.00045	0.00045
650	0	0.0003	0.00025	0.0003	0.0002	0.00035	0.00055	0.0003	0.00045	0.00045
700	0.0001	0.00015	0.00025	0.0004	0.00015	0.0003	0.0005	0.00065	0.0004	0.0004
750	0.0002	0.0002	0.0002	0.0001	0.00025	0.00035	0.00035	0.00045	0.00045	0.00035
800	0.00005	0	0.00005	0.00025	0.00015	0.00015	0.00045	0.00045	0.00035	0.0004
850	0.00005	0.00015	0.0002	0.0003	0.00015	0.00025	0.00035	0.0003	0.0005	0.00055
900	0	0.0003	0.0002	0.0001	0.00025	0.00025	0.0003	0.00035	0.00045	0.0004
950	0	0.00025	0.0002	0.00035	0.0002	0.00025	0.0003	0.0005	0.00035	0.0007

DATA SIZE = 10 000

ential cuttoff	filterSize3	filterSize5	filterSize7	filterSize9	filterSize311	filterSize13	filterSize15	filterSize17	filterSize19	filterSize21
500	0.0027	0.00075	0.00085	0.00095	0.00125	0.0016	0.0018	0.00165	0.0019	0.00265
1000	0.0004	0.0004	0.0008	0.001	0.00105	0.00135	0.00155	0.0018	0.00245	0.0022
1500	0.00045	0.00035	0.00065	0.0009	0.00115	0.0013	0.00165	0.0018	0.0022	0.00235
2000	0.0005	0.0003	0.0007	0.0009	0.0011	0.0011	0.0016	0.0018	0.0019	0.0022
2500	0.00025	0.00065	0.0005	0.00075	0.0011	0.00125	0.0018	0.0016	0.00205	0.00245
3000	0.0005	0.0005	0.0007	0.00095	0.0012	0.0014	0.00145	0.00195	0.00215	0.00235
3500	0.0003	0.00035	0.00065	0.0011	0.00145	0.00125	0.0016	0.0017	0.00205	0.00245
4000	0.00025	0.00065	0.001	0.00095	0.00115	0.00135	0.0015	0.0018	0.00195	0.00235
4500	0.00035	0.0004	0.00045	0.0011	0.0011	0.0013	0.00175	0.00155	0.00205	0.0023
5000	0.00045	0.00045	0.0009	0.0011	0.00135	0.0015	0.0019	0.0022	0.00245	0.003
5500	0.0003	0.00055	0.00085	0.0011	0.00125	0.00175	0.0019	0.0023	0.00225	0.0028
6000	0.0004	0.0004	0.00085	0.00105	0.0016	0.0018	0.0021	0.0022	0.0025	0.00265
6500	0.0003	0.00045	0.00075	0.00115	0.00135	0.0018	0.0019	0.0021	0.00245	0.00275
7000	0.00035	0.0007	0.001	0.0011	0.00125	0.00165	0.00195	0.00225	0.0025	0.00275
7500	0.00025	0.0007	0.0008	0.0012	0.0014	0.00175	0.0021	0.00215	0.0025	0.0028
8000	0.0003	0.0007	0.0009	0.0012	0.0014	0.00175	0.00175	0.0022	0.0026	0.0029
8500	0.0006	0.00045	0.00095	0.00105	0.0015	0.0017	0.0021	0.00225	0.00225	0.0027
9000	0.00035	0.00045	0.00085	0.00125	0.0014	0.00165	0.00175	0.0023	0.0026	0.0028
9500	0.0005	0.00065	0.00095	0.001	0.00115	0.0017	0.00195	0.0022	0.00265	0.00285

DATA SIZE = 100 000

sequential cuttoff	filterSize3	filterSize5	filterSize7	filterSize9	filterSize11	filterSize13	filterSize15	filterSize17	filterSize19	filterSize21
5000	0.0055	0.00315	0.0057	0.0074	0.011	0.0114	0.01355	0.0156	0.0183	0.0209
10000	0.0026	0.00305	0.0054	0.00745	0.00925	0.0121	0.0136	0.01575	0.01805	0.02115
15000	0.0022	0.00305	0.00535	0.00755	0.0093	0.01145	0.01375	0.0161	0.0189	0.02165
20000	0.00225	0.0032	0.00535	0.0075	0.0094	0.0114	0.0138	0.0157	0.0188	0.0213
25000	0.00235	0.00315	0.00535	0.00715	0.00885	0.012	0.0138	0.0161	0.0189	0.0211
30000	0.0025	0.003	0.0054	0.0072	0.00915	0.0115	0.01345	0.01615	0.01925	0.0209
35000	0.00235	0.0031	0.00525	0.0071	0.00935	0.01155	0.01405	0.016	0.0188	0.02135
40000	0.0023	0.0031	0.00525	0.00745	0.0091	0.0116	0.01365	0.016	0.01865	0.0217
45000	0.00225	0.00305	0.00525	0.0077	0.00955	0.0115	0.01385	0.01635	0.0184	0.0214
50000	0.00265	0.0038	0.00705	0.0093	0.0119	0.01475	0.02065	0.0197	0.02245	0.02815
55000	0.00265	0.0035	0.00705	0.00975	0.01165	0.01405	0.0172	0.02	0.0242	0.02615
60000	0.0024	0.00385	0.0072	0.0098	0.01215	0.01485	0.0168	0.01995	0.0228	0.0274
65000	0.0025	0.0036	0.0067	0.01005	0.01235	0.0144	0.01705	0.02095	0.0235	0.02585
70000	0.00245	0.0037	0.0073	0.0098	0.012	0.01545	0.01725	0.0199	0.02315	0.02885
75000	0.00255	0.00335	0.0068	0.0093	0.0127	0.0153	0.0177	0.02005	0.023	0.0262
80000	0.0025	0.0035	0.00695	0.0094	0.0128	0.0201	0.02285	0.02005	0.0235	0.0267
85000	0.00265	0.00395	0.00845	0.0102	0.01315	0.0146	0.0179	0.0205	0.0231	0.0261
90000	0.0025	0.0034	0.00735	0.00945	0.0123	0.01645	0.0177	0.01995	0.02445	0.0266
95000	0.0031	0.0036	0.0066	0.0095	0.01185	0.01465	0.017	0.02035	0.02335	0.0261

DATA SIZE = 1 000 000

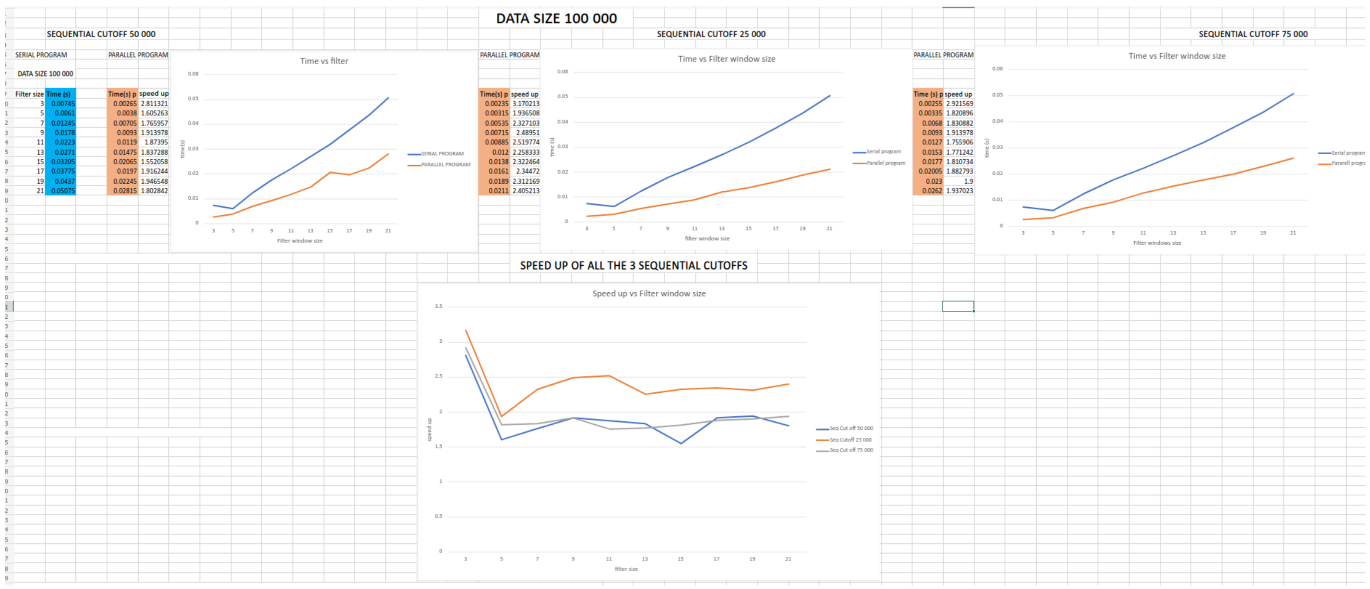
sequential cuttoff	filterSize3	filterSize5	filterSize7	filterSize9	filterSize31	filterSize13	filterSize15	filterSize17	filterSize19	filterSize21
50000	0.03495	0.0373	0.06075	0.07995	0.1011	0.11745	0.13995	0.1633	0.1903	0.21655
100000	0.02355	0.03125	0.0553	0.076	0.09725	0.11685	0.1381	0.16165	0.18745	0.21195
150000	0.0235	0.0315	0.05435	0.07535	0.1105	0.11665	0.13975	0.16065	0.18505	0.21275
200000	0.0241	0.03105	0.05485	0.0741	0.09495	0.1138	0.13765	0.161	0.18425	0.21135
250000	0.02275	0.0312	0.055	0.0747	0.0966	0.1138	0.1378	0.15995	0.1857	0.2126
300000	0.0228	0.03115	0.05465	0.0749	0.0938	0.1152	0.13705	0.16065	0.18875	0.2138
350000	0.02325	0.034	0.0543	0.07495	0.09535	0.1144	0.13785	0.15925	0.1869	0.21135
400000	0.02325	0.03095	0.0551	0.0747	0.0953	0.11365	0.1372	0.15965	0.18525	0.21145
450000	0.0223	0.03055	0.0546	0.07405	0.09305	0.1145	0.13555	0.15915	0.18535	0.2115
500000	0.0256	0.03555	0.07165	0.0976	0.11995	0.1502	0.1707	0.202	0.23415	0.26375
550000	0.02535	0.03785	0.07305	0.09665	0.1241	0.1428	0.17405	0.20725	0.2304	0.2984
600000	0.0308	0.04435	0.07425	0.1023	0.1249	0.15765	0.1891	0.2065	0.2345	0.2779
650000	0.03185	0.0409	0.07835	0.10595	0.12545	0.15045	0.1834	0.2154	0.24165	0.28005
700000	0.0311	0.0418	0.0724	0.0999	0.12965	0.1572	0.18105	0.2151	0.24305	0.2778
750000	0.0306	0.04145	0.0745	0.1029	0.12875	0.1543	0.18615	0.2185	0.25135	0.2731
800000	0.03045	0.0413	0.0749	0.10205	0.13485	0.15555	0.1806	0.22125	0.24245	0.27045
850000	0.03015	0.04295	0.0763	0.10005	0.1328	0.1514	0.19115	0.21525	0.24215	0.274
900000	0.03015	0.04215	0.0735	0.0991	0.13095	0.15545	0.1807	0.20725	0.2457	0.27895
950000	0.03155	0.0403	0.07545	0.10215	0.1283	0.1614	0.18725	0.20765	0.23995	0.2703

The extrapolated graphs show that for every data size,

1. the lower the sequential cut-off and the lower the filtering window size then the shorter the amount of time the program takes to filter the data, i.e the program speed increase.
2. Once the sequencial cut-off has reached half of the data size, there's spike(increase) in the time the program filters the data regardless of what the filter window is.

Speed up

To determing the speed up, a serial program and parallel program were run on two data sizes, but for a parallel program the 100 000 data size with sequential cut-off of 25 000, 50 000 & 75 000 and for a data size of 1 000 000 with sequential cut-off of 250 000, 500 000 & 750 000.



DATA SIZE 100 000

SEQUENTIAL CUTOFF 50 000

SERIAL PROGRAM — PARALLEL PROGRAM

DATA SIZE 100 000

Filter size	Time (s)	Time(s) p	speed up
3	0.00745	0.00265	2.811321
5	0.0061	0.0038	1.605263
7	0.01245	0.00705	1.765957
9	0.0178	0.0093	1.913978
11	0.0223	0.0119	1.87395
13	0.0271	0.01475	1.837288
15	0.03205	0.02065	1.552058
17	0.03775	0.0197	1.916244
19	0.0437	0.02245	1.946548
21	0.05075	0.02815	1.802842

SEQUENTIAL CUTOFF 25 000

PARALLEL PROGRAM

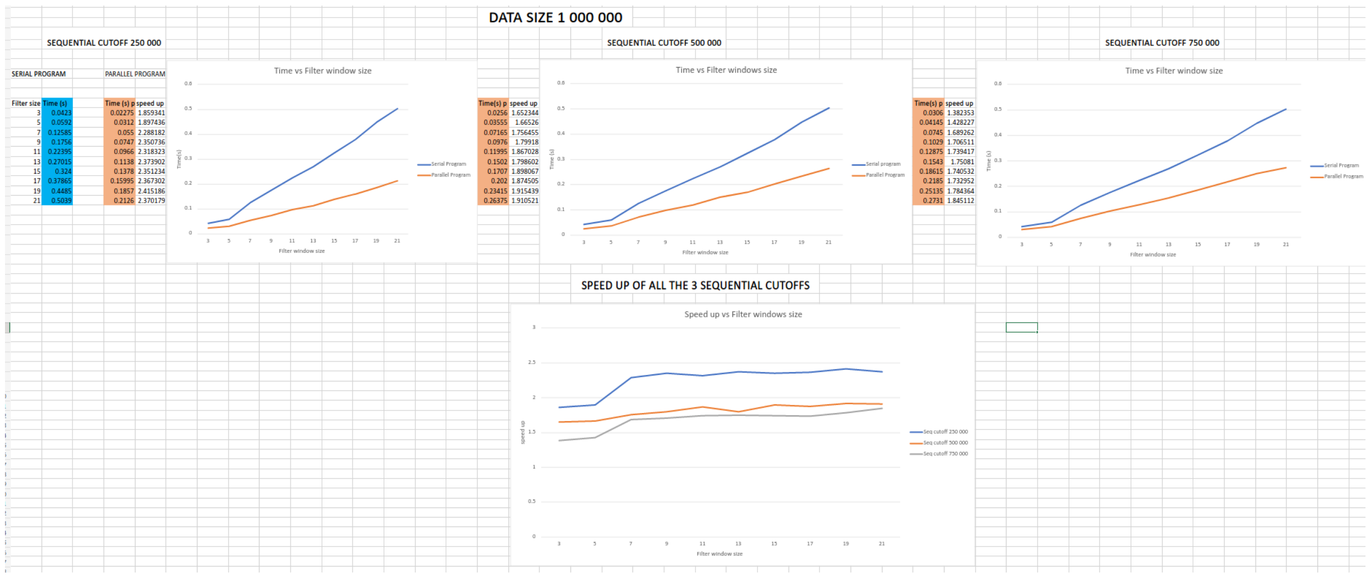
Time(s) p	speed up
0.00235	3.170213
0.00315	1.936508
0.00535	2.327103
0.00715	2.48951
0.00885	2.519774
0.012	2.258333
0.0138	2.322464
0.0161	2.34472
0.0189	2.312169
0.0211	2.405213

SEQUENTIAL CUTOFF 75 000

PARALLEL PROGRAM

Time (s) p	speed up
0.00255	2.921569
0.00335	1.820896
0.0068	1.830882
0.0093	1.913978
0.0127	1.755906
0.0153	1.771242
0.0177	1.810734
0.02005	1.882793
0.023	1.9
0.0262	1.937023

SPEED UP OF ALL THE 3 SEQUENTIAL CUTOFFS


DATA SIZE 1 000 000

SEQUENTIAL CUTOFF 250 000

SERIAL PROGRAM		PARALLEL PROGRAM	
Filter size	Time (s)	Time (s) p	speed up
3	0.0423	0.02275	1.859341
5	0.0592	0.0312	1.897436
7	0.12585	0.055	2.288182
9	0.1756	0.0747	2.350736
11	0.22395	0.0966	2.318323
13	0.27015	0.1138	2.373902
15	0.324	0.1378	2.351234
17	0.37865	0.15995	2.367302
19	0.4485	0.1857	2.415186
21	0.5039	0.2126	2.370179

SEQUENTIAL CUTOFF 500 000

Time(s) p	speed up
0.0256	1.652344
0.03555	1.66526
0.07165	1.756455
0.0976	1.79918
0.11995	1.867028
0.1502	1.798602
0.1707	1.898067
0.202	1.874505
0.23415	1.915439
0.26375	1.910521

SEQUENTIAL CUTOFF 750 000

Time(s) p	speed up
0.0306	1.382353
0.04145	1.428227
0.0745	1.689262
0.1029	1.706511
0.12875	1.739417
0.1543	1.75081
0.18615	1.740532
0.2185	1.732952
0.25135	1.784364
0.2731	1.845112

SPEED UP OF ALL THE 3 SEQUENTIAL CUTOFFS

From the data collected and graphs extrapolated, and for all filter sizes, the parallel program is more efficient relative to the serial program, from the speed up graph below the parallel program is twice faster than the serial program, with the highest speed up value of 3.17 on the 100 000 data size collected data, this result is far from the expected speed up, the highest speed up can be obtained when the data size is low as possible. This suggest that an increase in data size, will slow the program for both the seria and the parallel program.

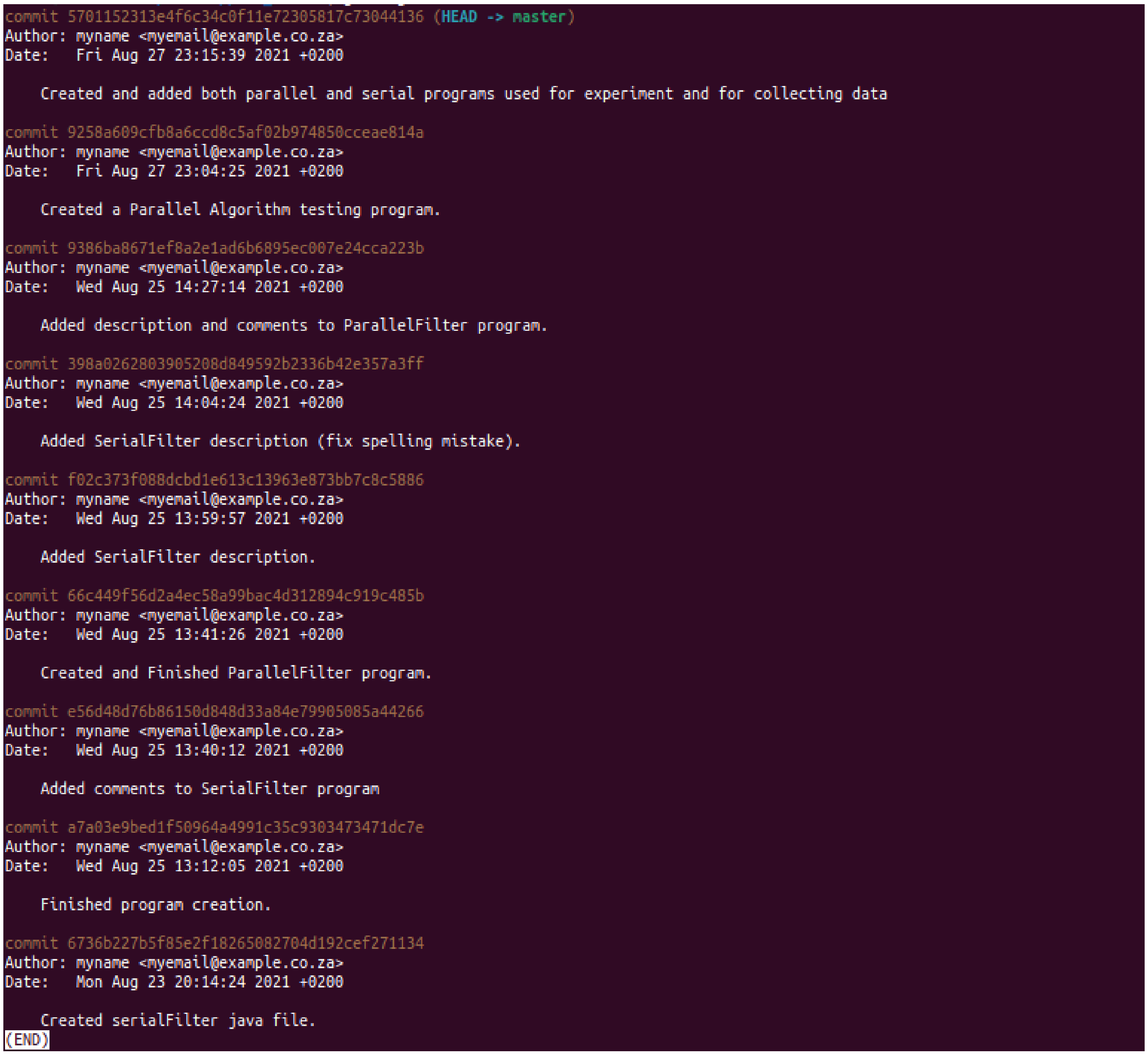
The optimal sequential cuttoff for the 100 000 data set is 50 000, and 250 000 for the 1 000 000 data set. This shows that the more the work is distributed up and processed at the same time equally, the faster the program becomes for all data sizes but the optimal sequential cut-off will vary.

Conclusions

- My first prediction was wrong, the parallel program did not run on order O(log(n)), infact the obtained results grew lineary from the serial program.
- The lower the filter and with a suitable sequential cut-off results in the hight peak of the speed up of the program, the parallel program is almost 3x efficient than the serial program when the data size is not very much large, which is in the line of Amdahl's law, speed up isn't much when problem size is small.
- An increase in data size, results in a little decrease in speed up of the program to when the data size is almost 10x smaller.
- Parallelising is effective but not signicficatly, an increase in computer cores does not significantly improve the speed.

Appendices

GIT LOG



```
commit 5701152313e4f6c34c0f11e72305817c73044136 (HEAD -> master)
Author: myname <myemail@example.co.za>
Date:   Fri Aug 27 23:15:39 2021 +0200

    Created and added both parallel and serial programs used for experiment and for collecting data

commit 9258a609cfb8a6ccd8c5af02b974850cceae814a
Author: myname <myemail@example.co.za>
Date:   Fri Aug 27 23:04:25 2021 +0200

    Created a Parallel Algorithm testing program.

commit 9386ba8671ef8a2e1ad6b6895ec007e24cca223b
Author: myname <myemail@example.co.za>
Date:   Wed Aug 25 14:27:14 2021 +0200

    Added description and comments to ParallelFilter program.

commit 398a0262803905208d849592b2336b42e357a3ff
Author: myname <myemail@example.co.za>
Date:   Wed Aug 25 14:04:24 2021 +0200

    Added SerialFilter description (fix spelling mistake).

commit f02c373f088dcbd1e613c13963e873bb7c8c5886
Author: myname <myemail@example.co.za>
Date:   Wed Aug 25 13:59:57 2021 +0200

    Added SerialFilter description.

commit 66c449f56d2a4ec58a99bac4d312894c919c485b
Author: myname <myemail@example.co.za>
Date:   Wed Aug 25 13:41:26 2021 +0200

    Created and Finished ParallelFilter program.

commit e56d48d76b86150d848d33a84e79905085a44266
Author: myname <myemail@example.co.za>
Date:   Wed Aug 25 13:40:12 2021 +0200

    Added comments to SerialFilter program

commit a7a03e9bed1f50964a4991c35c9303473471dc7e
Author: myname <myemail@example.co.za>
Date:   Wed Aug 25 13:12:05 2021 +0200

    Finished program creation.

commit 6736b227b5f85e2f18265082704d192cef271134
Author: myname <myemail@example.co.za>
Date:   Mon Aug 23 20:14:24 2021 +0200

    Created serialFilter java file.
(END)
```