# Parallel Assignment Report

## Introduction

The aim of this project was to create a Median Filter to "smooth" out a one-dimensional data set. Median Filtering is a nonlinear digital filtering technique that is often used to remove noise from a data set.

The Median Filter algorithm is to be compared to a sequential version, which will present the benefits and limitations of parallel programming.

In this this parallel algorithm, a divide and conquer approach and the Java Fork Join Framework are used to implement the solution.

The algorithm uses a naive approach to Median Filtering whereby a filter window size of factor three or greater is sorted to find the median which is then placed in the middle position. This is done successively over a data set of random numbers.

The parallel median filter was tested over three different architectures namely; a personal laptop, UCT desktop and UCTs nightmare server.

The Java Fork Join Framework provides an expected time bound. Where the time for P processors or $T_p$ is given by $O(\frac{T_1}{P} + T_\infty)$ .Where $T_1$ is the work and $T_\infty$ is the span or critical path. However, this bound only holds under some assumptions.

The assumptions are both related to load misbalancing. The first assumption is that threads created do approximately the same amount of work, as over worked threads will delay the overall completion of the task. The second is that threads do a small but not negligible amount, this similarly helps with load balancing.

If these assumptions hold, we can see that with a small number of processors (P). $T_1/P$ is likely to dominate and we can expect roughly linear speed-up (i.e. doubling processors halves running time). As the number of processors increases the span becomes more relevant and the limit on runtime is more influenced by $T_\infty$.

Amdahl's Law is a mathematical principle showing the drastic reduction in speedup as a result of, doing sequential work within a parallel algorithm.

$$Speedup = \frac{1}{1 - P + \frac{P}{N}}$$

Where 1-P is the sequential fraction, P is the parallel fraction and N is the number of processors. This law will be further considered in the results section.

## Methods

1. Framework and code

The Java Fork Join Framework was used to implement the Median Parallel Filter. More specifically, the Parallel Filter class was extended with the RecursiveAction class.

This method involves creating a Fork Join Pool and using the invoke method on it. The parallelism is started by calling this invoke method.

In the Parallel Filter class the compute method is used to filter the data set passed into the array. This is done until the array size is equal to (or less than) the sequential cut-off. When sequential cut-off is reached, the algorithm switches to the sequential computation.

```
//Method does computation in parallel until sequential threshold is reached
protected void compute()
{
    if (hi - lo <= seq_thresh)
    {
        for (int i = 0; i < medianPos; i++) {
            cleanArray.add(noiseArray[i]);
        }
        //Populate temporary array with filtered values
        for (int j = 0; j <= filterAdjust; j++) {//<=
            ArrayList temp = new ArrayList();

            for (int k = 0; k < filterSize; k++) {
                temp.add(noiseArray[j + k]);
            }

            Collections.sort(temp);
            Object medianVal = temp.get(medianPos);
            cleanArray.add(medianVal);
        }
    }
    else{
        ParFilter left = new ParFilter(noiseArray, filterSize, lo, (hi + lo) / 2);
        ParFilter right = new ParFilter(noiseArray, filterSize, (hi + lo) / 2, hi);
        left.fork();
        right.compute();
        left.join();
    }
```

2. Validation of algorithm and timing

The algorithm was verified using JUNIT testing, whereby a filtered data set (manually computed) was compared to the array of filtered values.

To measure timing the System.currentTimeMillis() method was used to get a precise time result.

System.gc() was called before the timing block to minimize the likelihood that the garbage collector would run during execution.

The time to filter the data set using the sequential algorithm and parallel algorithm were then compared. Speedup was calculated by taking the sequential time divided by the parallel time.

3. Machine Architectures

The three architectures below were used to test the code.

a. Personal Laptop

```
david@david-HP-Pavilion-15-Notebook-PC:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    2
Core(s) per socket:    2
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 69
Stepping:              1
CPU MHz:               759.000
BogoMIPS:              4589.39
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              3072K
NUMA node0 CPU(s):     0-3
```

b. UCT desktop

```
frndav011@sl-dual-127:~/NetBeansProjects/MyRepo/Ass_1$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    2
Core(s) per socket:    2
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 58
Model name:            Intel(R) Core(TM) i3-3220 CPU @ 3.30GHz
Stepping:              9
CPU MHz:               1929.968
CPU max MHz:           3300.0000
CPU min MHz:           1600.0000
BogoMIPS:              6599.99
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              3072K
NUMA node0 CPU(s):     0-3
```

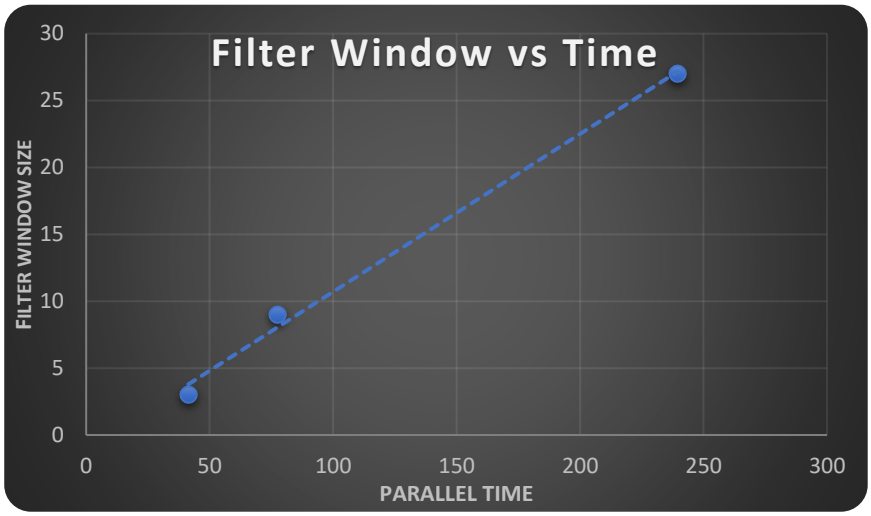
c. UCT Nightmare Server

```
frndav011@nightmare:~/NetBeansProjects/MyRepo/Ass_1$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 44
Stepping:              2
CPU MHz:               1600.000
BogoMIPS:              4788.23
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              12288K
NUMA node0 CPU(s):     0-7
```

## Results and Discussion

**Filter Window Size:**

| Parallel Time | Filter Window Size |
|---|---|
| 41.338 | 3 |
| 77.433 | 9 |
| 239.594 | 27 |




Comments:

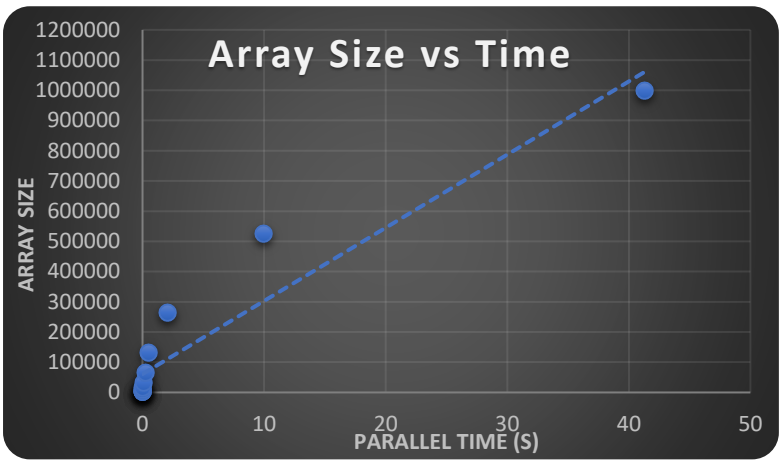
To compare the filter window size, sequential cut-offs of 4096 and an array size of 1 000 000 were kept constant. The filter window size was then varied three times at sizes of; 3, 9 and 27.

The results showed that a filter window size of 3 is the most effective. This could be because with a window size of 3, the median can be calculated more quickly. Resulting in a faster computation.

The graph above clearly shows that with an increase in filter window size, we have a longer parallel computation time.

**Input Array Size:**

| Array Size | Parallel Time | Speedup | Threads |
|---|---|---|---|
| 4096 | 0.009 | 0.889 | 1 |
| 8192 | 0.021 | 0.714 | 2 |
| 16384 | 0.045 | 0.422 | 4 |
| 32768 | 0.118 | 0.186 | 8 |
| 65536 | 0.288 | 0.122 | 16 |
| 131072 | 0.538 | 0.115 | 32 |
| 262144 | 2.104 | 0.051 | 64 |
| 524288 | 10.004 | 0.012 | 128 |
| 1000000 | 41.338 | 0.012 | 256 |




Comments:

To investigate the effect of array sizes, a filter window size of three and a sequential cut-off, of 4096 was used.

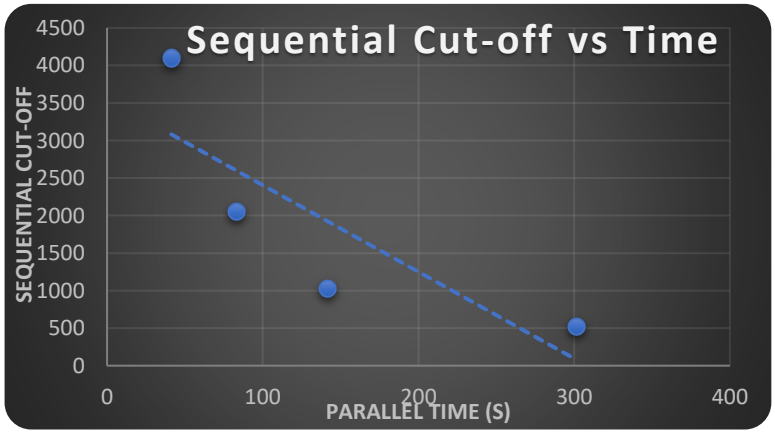
The data and graph show that with an increase in array size we have an increase in parallel computation time, as expected. Similarly, as the array size increases we have a decrease in speedup.

The data also shows that the highest speedup was achieved at an array size of 4096. Since we have chosen a sequential cut-off of 4096, we have hence created one thread to do computation.

The speedup decreases with more threads because there is too little computational work to warrant the overhead involved in creating multiple threads.

**Sequential Cut-off:**

| Parallel Time | Sequential Cut-off |
|---|---|
| 41.338 | 4096 |
| 83.198 | 2048 |
| 141.642 | 1024 |
| 301.651 | 512 |



Comments:

Using a filter window size of 3 and an array size of 1 000 000, the sequential cut-off was varied. The results show that a sequential cut-off, of 4096 gives the shortest time.

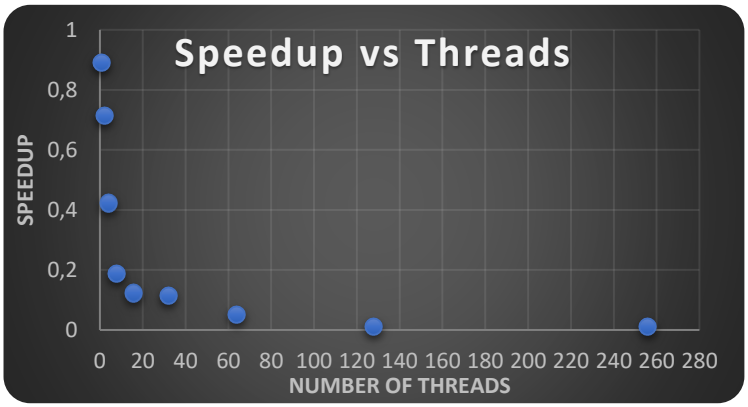
This is because the data set of 1 000 000 is large enough to warrant the creation of 244 threads. In other words, there is enough computational work for the pool of threads to be useful.

## Machine Architectures:

The parallel median filter was tested over three different architectures. Each architecture was run with a filter window size of three and a sequential cut-off, of 4096.

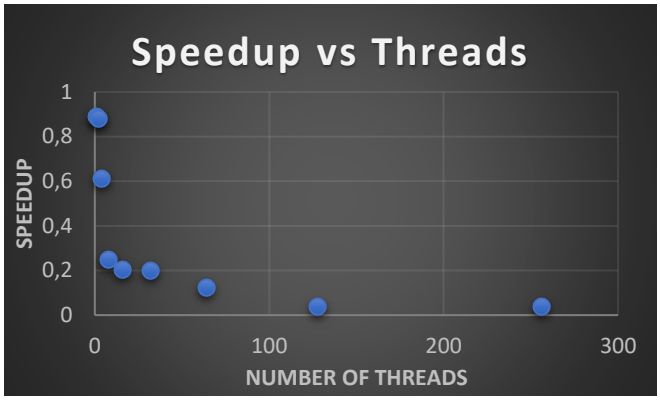
**Laptop:**

| Array Size | Parallel Time | Speedup | Threads |
|---|---|---|---|
| 4096 | 0.009 | 0.889 | 1 |
| 8192 | 0.021 | 0.714 | 2 |
| 16384 | 0.045 | 0.422 | 4 |
| 32768 | 0.118 | 0.186 | 8 |
| 65536 | 0.288 | 0.122 | 16 |
| 131072 | 0.538 | 0.115 | 32 |
| 262144 | 2.104 | 0.051 | 64 |
| 524288 | 10.004 | 0.012 | 128 |
| 1000000 | 41.338 | 0.012 | 256 |



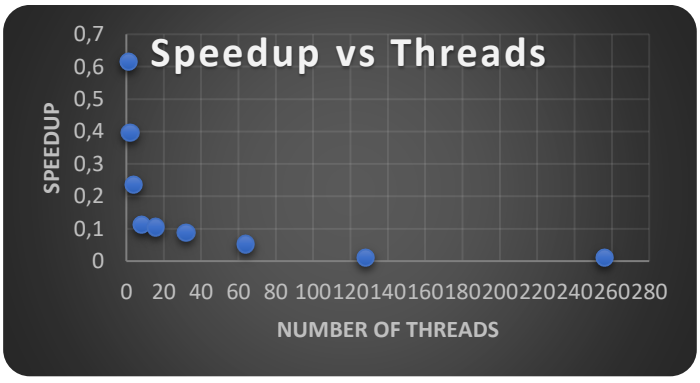

**UCT Desktop:**

| Array Size | Parallel Time | Speedup | Threads |
|---|---|---|---|
| 4096 | 0.009 | 0.889 | 1 |
| 8192 | 0.017 | 0.882 | 2 |
| 16384 | 0.031 | 0.613 | 4 |
| 32768 | 0.089 | 0.247 | 8 |
| 65536 | 0.171 | 0.205 | 16 |
| 131072 | 0.315 | 0.197 | 32 |
| 262144 | 0.88 | 0.122 | 64 |
| 524288 | 3.07 | 0.038 | 128 |
| 1000000 | 12.675 | 0.038 | 256 |




**UCT Nightmare Server:**

| Array Size | Parallel Time | Speedup | Threads |
|---|---|---|---|
| 4096 | 0.013 | 0.615 | 1 |
| 8192 | 0.038 | 0.395 | 2 |
| 16384 | 0.081 | 0.235 | 4 |
| 32768 | 0.194 | 0.113 | 8 |
| 65536 | 0.335 | 0.104 | 16 |
| 131072 | 0.693 | 0.089 | 32 |
| 262144 | 2.071 | 0.052 | 64 |
| 524288 | 9.95 | 0.012 | 128 |
| 1000000 | 29.723 | 0.016 | 256 |

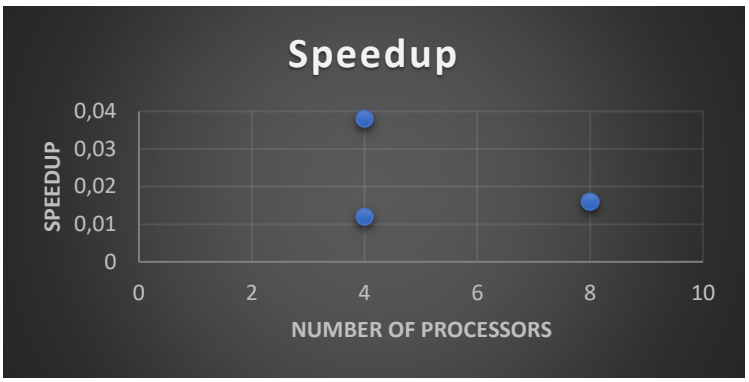



Investigating speedup with an increased number of processors.

| | Number of cores | Speedup |
|---|---|---|
| **Laptop** | 4 | 0.012 |
| **Desktop** | 4 | 0.038 |
| **Nightmare** | 8 | 0.016 |

Investigating effect of clock speed.

| | Number of cores | Clock Speed |
|---|---|---|
| **Laptop** | 4 | 759.000MHz |
| **Desktop** | 4 | 1929.968Mhz |
| **Nightmare** | 8 | 1600MHz |




Comments:

The best speedups were achieved on my personal laptop and the UCT desktop, both speedups were approximately 0.889.

This speedup was achieved with one thread and a data size of 4096. The sequential cut-off was set at 4096 which means that the entire computation was done is parallel. This supports Amdahl's Law, as the best speedup is consistently achieved without any sequential computation.

In comparing the three architectures, the desktop was found to give the best performance. Initially it was expected that the machine with 8 cores (Nightmare server), would give the best results.

Possible reasons for nightmare not performing well can be attributed to the users who were interacting with the server whilst the experimental timing was performed.

Furthermore, for the size of tasks that were performed. An 8-core machine may be unnecessary, since there is no way of controlling how the scheduler of the operating system will distribute CPU time to tasks. In other words, we cannot be sure that our code will use the full processing power of the machine.

Finally, when comparing the clock-speeds of the machines, the desktop clearly has the highest at 1929.968Mhz. This could also contribute to it being the most efficient.

**Conclusions:**

Through investigating the limitations and benefits of parallel algorithms (specifically the Median Filter). It was clear that serialising part of the computation reduced the effectiveness of speedup within the parallel algorithm. This was expected and can be confirmed by Amdahl's Law.

In varying the filter size, it was found that the most effective filter size for the data sets chosen was a size of three. This is rationalised in the fact that computation is much faster is a smaller window.

As expected with an increase in array sizes (data sets), the speedup was reduced. It was also shown however that with a large enough array and a reasonable sequential cut-off (number of threads), that we could improve speedup.

Linear speedup was not achieved when comparing the 4-core machines to the 8-core machine. This confirms that there are many trade-offs to consider when choosing the architecture to run parallel algorithms on.

Some of the architecture trade-offs to consider are clock speeds and memory hierarchy.

Finally, parallel programming can be an effective tool in speeding up computation. However, there are many trade-offs to take into consideration. As a programmer using the framework it is important to write efficient code and allow the Fork Join Framework scheduler to perform its tasks.

## References

Grossman, D. (2015). *A Sophomoric* Introduction to Shared-Memory Parallelism and Concurrency.*