ASSIGNMENT 1
PARALLELISM

Introduction

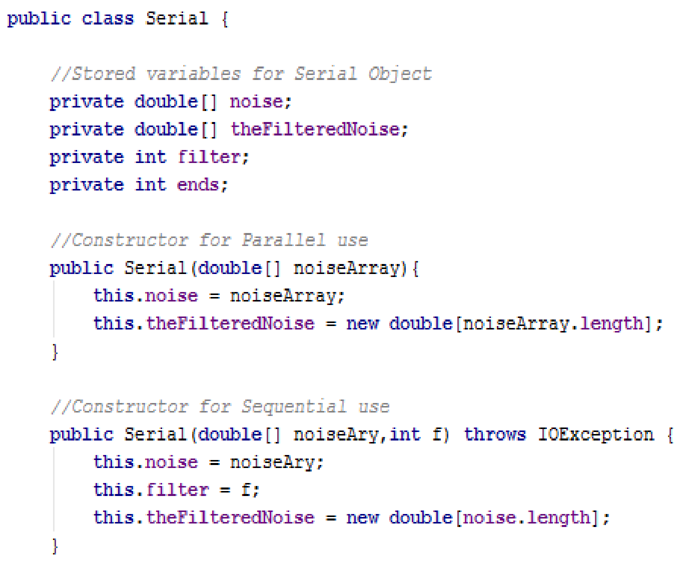
The problem presented in this assignment is to check if parallelism is more efficient when median filtering a 1-D array. Median filtering is a non-linear digital filtering technique that is often used to remove noise from a data set. To process median filtering on noise (an array), the noise is split into groups of a filter size given (eg.3) then the median (middle) value is found of that smaller group. The median found, from including the neighboring entries, replaces the value at the current index. Eg x = [7 95 1 45 6 2 47] at filter 3 transforms into y = [7 7 45 6 6 6 47].

This process is a hefty one and requires a decent amount of computational power so parallelizing the algorithm should prove helpful as the calculations could be spread over threads efficiently with the Java ForkJoin divide and conquer methodology. Especially since the Java ForkJoin Framework is geared towards the splitting of large computations into smaller threads. Hence the aim of this research assignment will be to find out if it is true that parallelizing a median filtering algorithm provides a more efficient faster route to the answer.

Methods

Solution Description

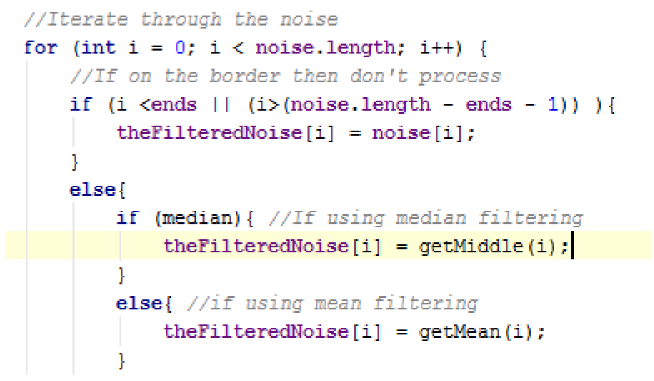
For the Sequential algorithm, an object class called Serial was created to manage the filter and noise array. The Object class was made to be reusable by the Parallel algorithm as the actual median filtering would have to be done sequentially but the Parallel algorithm would attempt to be more efficient by processing multiple smaller parts of the noise at the same time.



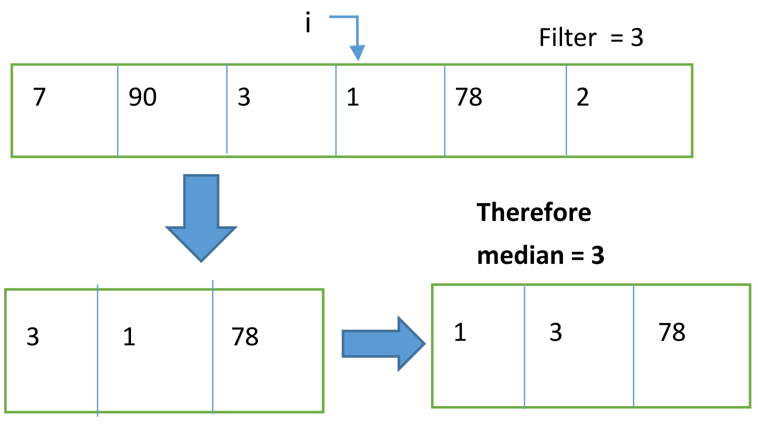
```java
public class Serial {

    //Stored variables for Serial Object
    private double[] noise;
    private double[] theFilteredNoise;
    private int filter;
    private int ends;

    //Constructor for Parallel use
    public Serial(double[] noiseArray){
        this.noise = noiseArray;
        this.theFilteredNoise = new double[noiseArray.length];
    }

    //Constructor for Sequential use
    public Serial(double[] noiseAry,int f) throws IOException {
        this.noise = noiseAry;
        this.filter = f;
        this.theFilteredNoise = new double[noise.length];
    }
```

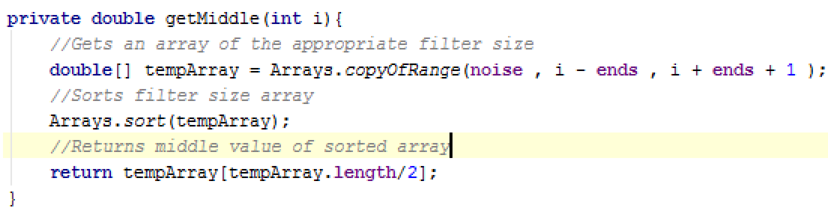
The FileUtil class will go through a given file and retrieve a double array which is the noise. This noise is then used to create a Serial Object which stores the noise and simultaneously creates the result array. There are two constructors, one is for when using the class fully sequentially where you would give both filter and noise at the same time then launch a method that calculates time to filter on object. The other method is for individual threads in the Parallel algorithm where time of individual median filterings is not timed, the whole process is timed.

The filtering of the noise is done by iterating through the noise and at each index then making a smaller copy with the appropriate filter size of the noise where the center is the element at the index. The smaller array copy is then sorted and the middle is found.



```
//Iterate through the noise
for (int i = 0; i < noise.length; i++) {
    //If on the border then don't process
    if (i <ends || (i>(noise.length - ends - 1)) ){
        theFilteredNoise[i] = noise[i];
    }
    else{
        if (median){ //If using median filtering
            theFilteredNoise[i] = getMiddle(i);
        }
        else{ //if using mean filtering
            theFilteredNoise[i] = getMean(i);
        }
```



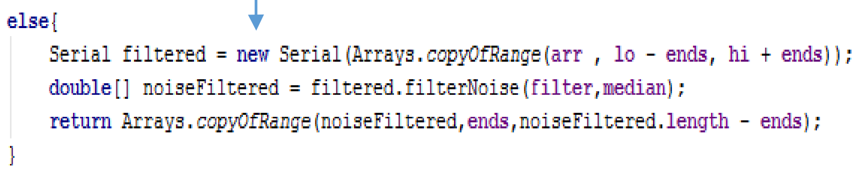


```
private double getMiddle(int i){
    //Gets an array of the appropriate filter size
    double[] tempArray = Arrays.copyOfRange(noise , i - ends , i + ends + 1 );
    //Sorts filter size array
    Arrays.sort(tempArray);
    //Returns middle value of sorted array
    return tempArray[tempArray.length/2];
}
```

So to sum up the sequential algorithm, the noise array is iterated through and each element follows the process described by the above diagram. Now to the Parallel!

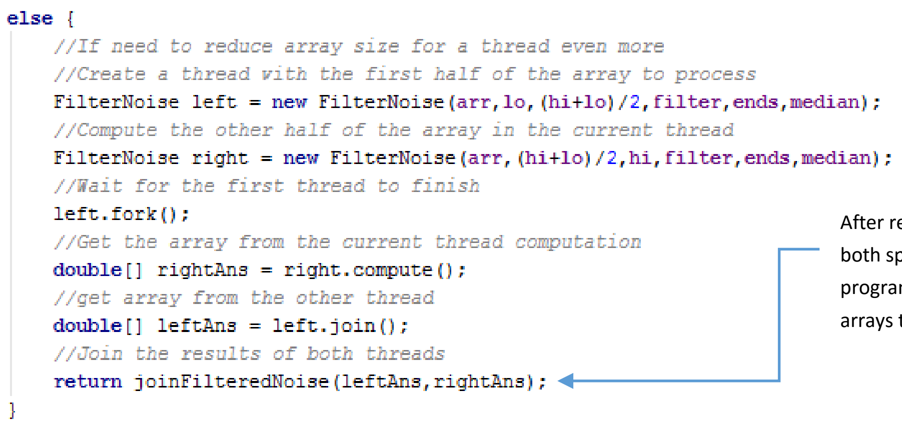
The Parallel algorithm uses the Fork/Join Framework and the divide and conquer methodology to keep splitting the noise array into smaller parts. A new thread is assigned to each smaller part created. The threads will keep splitting until the array size reaches the sequential cutoff limit which is where the sequential part of the thread will be done. The sequential part of the thread will create a Serial object from above and then filter noise using the same code as the sequential algorithm. The Parallel algorithm essentially splits the noise into segments of the size of the sequential cutoff (eg. 1000) and then simultaneously processes (filters) each segment of the noise separately.

Threads processing segment sequentially



```
else{
    Serial filtered = new Serial(Arrays.copyOfRange(arr , lo - ends, hi + ends));
    double[] noiseFiltered = filtered.filterNoise(filter,median);
    return Arrays.copyOfRange(noiseFiltered,ends,noiseFiltered.length - ends);
}
```

Fork/Join threading to create left and right threads each with one half of the noise array

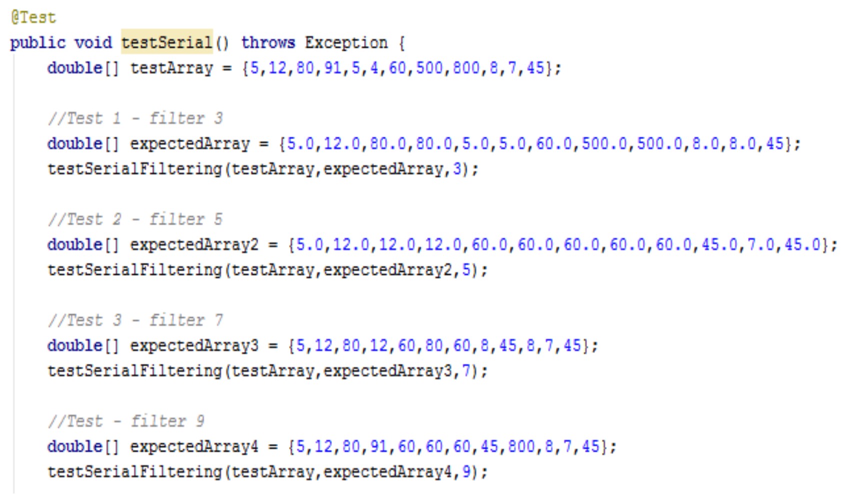


```
else {
    //If need to reduce array size for a thread even more
    //Create a thread with the first half of the array to process
    FilterNoise left = new FilterNoise(arr,lo,(hi+lo)/2,filter,ends,median);
    //Compute the other half of the array in the current thread
    FilterNoise right = new FilterNoise(arr,(hi+lo)/2,hi,filter,ends,median);
    //Wait for the first thread to finish
    left.fork();
    //Get the array from the current thread computation
    double[] rightAns = right.compute();
    //get array from the other thread
    double[] leftAns = left.join();
    //Join the results of both threads
    return joinFilteredNoise(leftAns,rightAns);
}
```

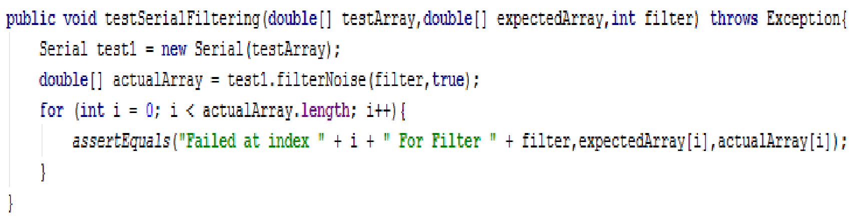
After retrieving results from both split threads, the program will add the double arrays together

Validation

The algorithms were validated with Junit testing. The Serial algorithm was given multiple test cases that was worked on individually (by me :)) and tested against the results from the Serial algorithm. The test cases tested the Serial algorithm with different array sizes and different filters until the serial algorithm was considered to be correct. The Junit tests will iterate through each element in the expected array and compare the expected element with the actual element produced from the Serial class.



```java
@Test
public void testSerial() throws Exception {
    double[] testArray = {5,12,80,91,5,4,60,500,800,8,7,45};

    //Test 1 - filter 3
    double[] expectedArray = {5.0,12.0,80.0,80.0,5.0,5.0,60.0,500.0,500.0,8.0,8.0,45};
    testSerialFiltering(testArray,expectedArray,3);

    //Test 2 - filter 5
    double[] expectedArray2 = {5.0,12.0,12.0,12.0,60.0,60.0,60.0,60.0,60.0,45.0,7.0,45.0};
    testSerialFiltering(testArray,expectedArray2,5);

    //Test 3 - filter 7
    double[] expectedArray3 = {5,12,80,12,60,80,60,8,45,8,7,45};
    testSerialFiltering(testArray,expectedArray3,7);

    //Test - filter 9
    double[] expectedArray4 = {5,12,80,91,60,60,60,45,800,8,7,45};
    testSerialFiltering(testArray,expectedArray4,9);
```



```java
public void testSerialFiltering(double[] testArray,double[] expectedArray,int filter) throws Exception{
    Serial test1 = new Serial(testArray);
    double[] actualArray = test1.filterNoise(filter,true);
    for (int i = 0; i < actualArray.length; i++){
        assertEquals("Failed at index " + i + " For Filter " + filter,expectedArray[i],actualArray[i]);
    }
}
```

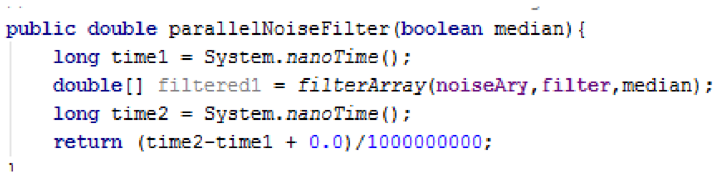
Once the Serial code was validated to the point where it could be considered correct, the Parallel algorithm could now be compared with the Serial to check if the results were correct. This was done to reuse code and be able to create more dynamic test cases with Parallel.

This is still not the best method to test the Parallel algorithm and is certainly not full proof. A Parallel algorithm provides difficulty with testing as threads do not finish in an order so threads process methods on the same data simultaneously which could cause unforeseen errors. These are hard to find as there are many possibilities in how threads can be ordered.

Timing and Speed-Up

The filtering of noise with Sequential and Parallel algorithms were timed individually so as to be able to compare the two. The timing of the algorithms is a huge indicator of which algorithm is able to process the noise more efficiently thus could contribute to this research assignment.

The timing was retrieved at a nano second measurement for the best accuracy. The timing was only done when the data was being filtered so as to be as fair as possible. On the right, we can see that the time was retrieved at the start of the method, then the filtered noise was retrieved by running the parallel algorithm, the final time was captured and the difference was returned by the method. This method of timing has been tested to be accurate in this circumstance.



```java
public double parallelNoiseFilter(boolean median){
    long time1 = System.nanoTime();
    double[] filtered1 = filterArray(noiseAry,filter,median);
    long time2 = System.nanoTime();
    return (time2-time1 + 0.0)/1000000000;
```

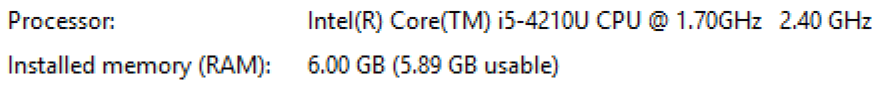
The time of one individual run is surely not enough (due to warm ups) to maintain a good estimate of how long the sequential and parallel algorithms actually take. With this in mind, a csv file (Excel) was produced with the times of the first 1000 runs of the algorithm. The average is found from these times for use in this research assignment. This average provides a better estimate of where about the timing of a Parallel or Sequential algorithm differ.

The Speed-Up was calculated in a similar sense, the time of the Serial code divided by the time of the parallel code which produces how much faster the parallel algorithm is from the sequential algorithm. The program was run a large amount of times with the speed-up and times marked down in a csv file. The csv file (Excel) is then used to find the average speed-up of the parallel code and the csv file is used to generate intuitive graphs.

$$\boxed{\text{Speed-up} = \frac{T_1}{T_P}}$$

Machine Architectures

1) Windows 10 - 2 Cores



Processor: Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz 2.40 GHz
Installed memory (RAM): 6.00 GB (5.89 GB usable)

2) Ubuntu – 4 Cores (8 logical processors but 4 physical cores) - Nightmare Server (Used in early mornings with presumably low traffic)

Also, for more results cores were disabled for java on some of the architectures when the program was running to simulate different architectures. A 1 Core (Done to see if speed up is = 1) and a 3 core architecture was simulated for more results.

Difficulties

Problems that arose during the parallelizing of the median filtering algorithm was the border processing of the broken up segments. When passing a middle segment of the noise to be processed, the borders of that segment would need to access elements outside their segment to find the median. This proved difficult in code as now the segments had to be allowed access extra elements thus extending the size of the segment passed.

Furthermore, the array copies that were made in threads could result in expensive computations in each thread which would slow down the speed-ups thus the results of this experiment.

Please Turn Over

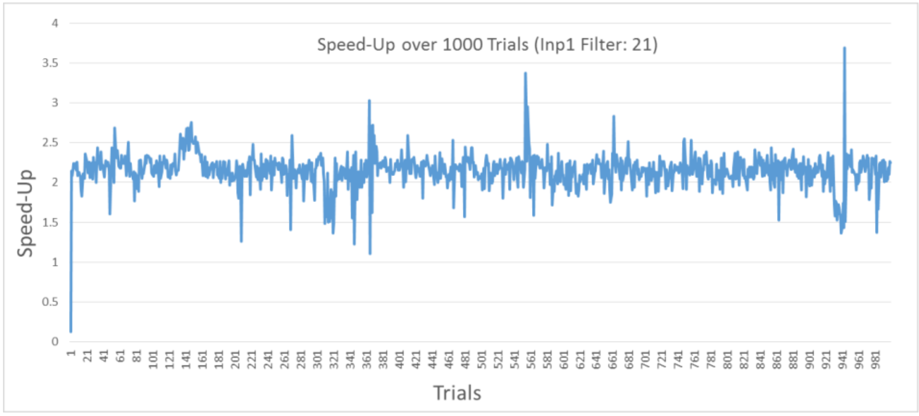
Results and Discussion

Now for the data! There are essentially three variables that affect the speed of the parallel algorithm. The filter size, data size and the sequential cut-off (limit). Each variable will be tested to find the best condition in which the parallel algorithm is most efficient. We will start with data sizes.

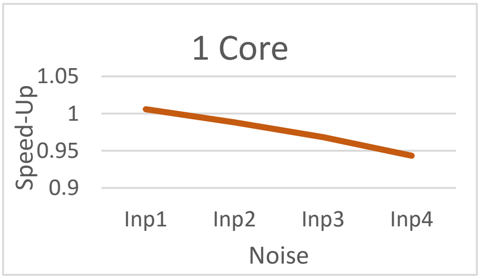
Data Sizes

First it is to be established that an average is used with the speed-up values, so 1000 speed-ups are calculated on each architecture then the final speed-up displayed in the graphs would be the average. This is done for accuracy as there is some variation in the results. As an example the graph to the right is the speed-up of the inp1.txt noise run at filter 21, 1000 times to produce:

Average Speed-Up = 2.152

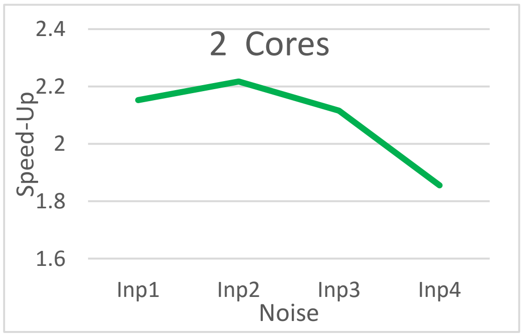


Each file with different data sizes are processed and compared below. The Filter 21 and sequential cut-off of 10000 will be kept constant throughout these comparisons.

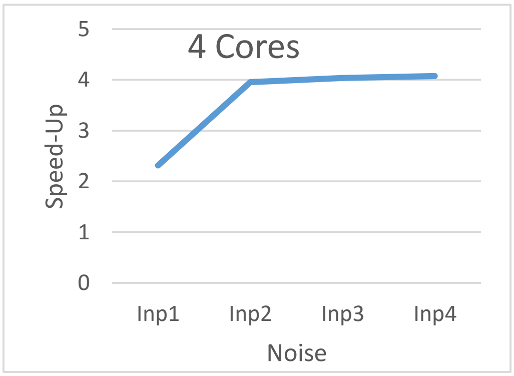


The 1 Core artificial architecture was done more out of curiosity of whether theory would show practically. In theory, a parallel algorithm should run at near the same speed as sequential code as seen on the left but with larger files more threads are created thus more computational overhead. This means that the parallel algorithm would get slower than the sequential algorithm with larger files which is observed.

From the graphs below, it can be seen that the parallel algorithm peaks with speed-ups at around the data size of inp2 and inp3 which are medium sized noise (100 000 to 400 000 elements). This also shows that the parallel algorithm performs especially well with larger files provided there are a several cores. The 2 Core architecture peaks early and then slows down as the file size gets to be too high for just two cores to be that much faster than one. The 4 Core architecture can handle larger data sizes more efficiently than the 2 core as the work gets split more amongst the processors hence a higher speed-up attained. There is no dip in the 4 core architecture similar to the 2 core architecture as the 4 core can handle the large data size of inp4 but would most likely start decreasing with an even larger data size.



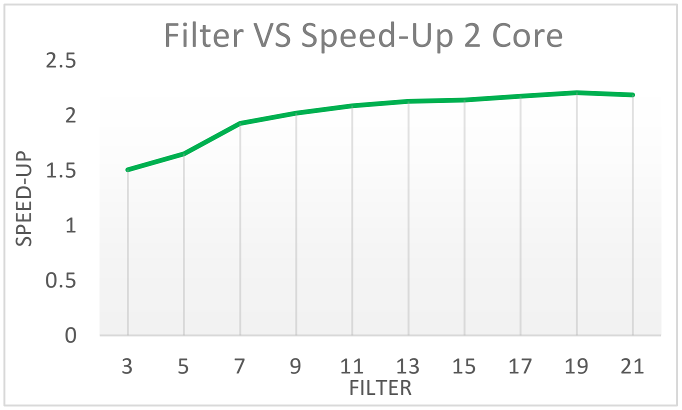
Therefore as the noise of **inp2** yields the fastest results for the parallel algorithm, we shall use it as the optimal data size.

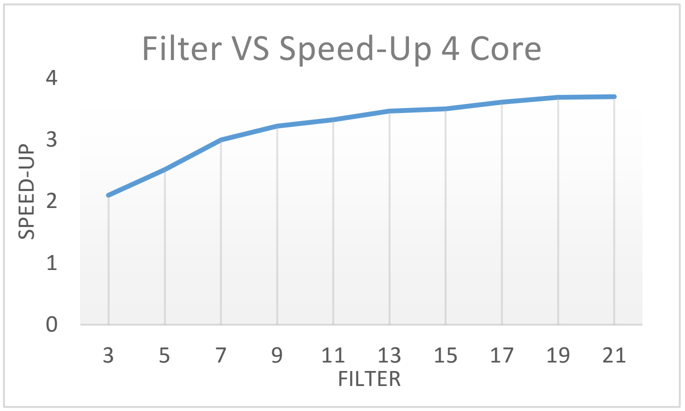


Filter Size

All filters shall be tested at the noise data size of inp2.txt and sequential cutoff of 10 000 for fairness amongst architectures. The parallel algorithm optimal filter size will attempt to be found. Functionality was added to code to be able to run through every filter and collect speed-up results as shown here.

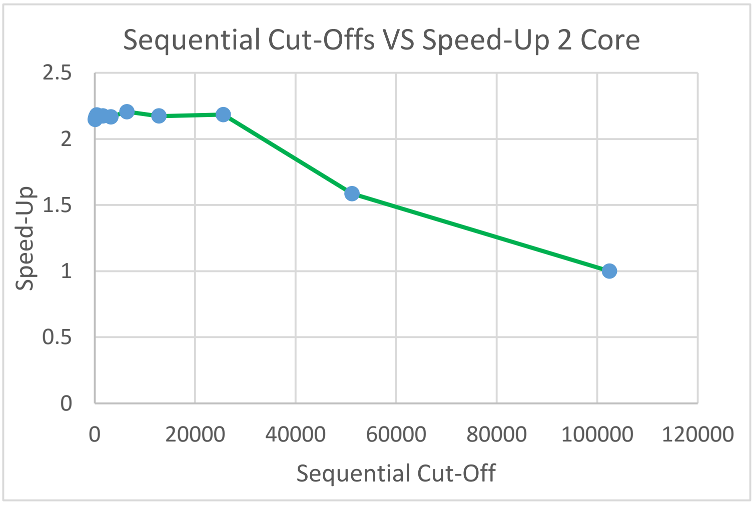
As the filter increases, the computation becomes more expensive hence a parallel algorithm is more suited for the job. This trend is observed below as the filter increases, the speed-up increases and peaks slightly at filter 19 hence we shall use the filter 19 as the optimal filter for the parallel algorithm.

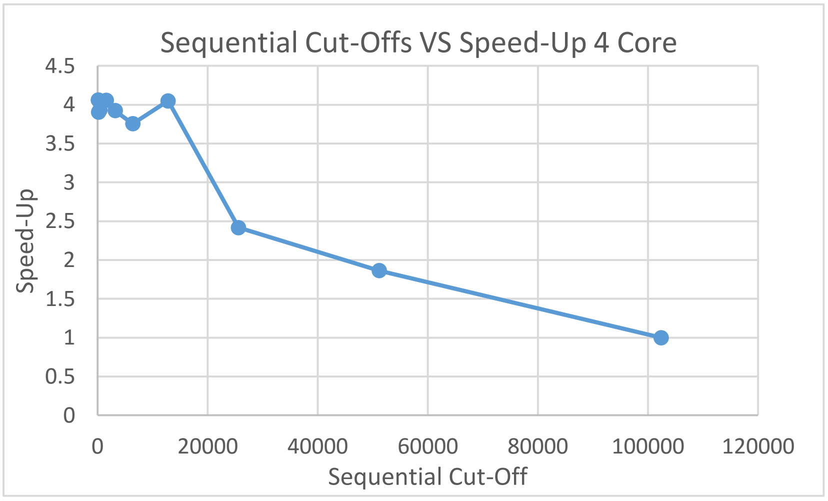




Sequential Cut-Off

To determine the best possible sequential cut-off, different cut-offs were used against each other on the 2 and 4 core architecture with filter size and data size constant at 19 and inp2.txt respectively.





It can be seen that at only significantly large sequential cut-offs does the program start losing its speed-up. This happens as a thread is given too much work with a large sequential cut-off therefore threads take longer to process their data. The optimal sequential cut-off to be observed from the graph is a cut-off within the range of 500 - 20 000. Since there seems to be a peak at the 10 000 cut-off, we shall use it as our optimal sequential cut-off for the parallel algorithm.

Discussion

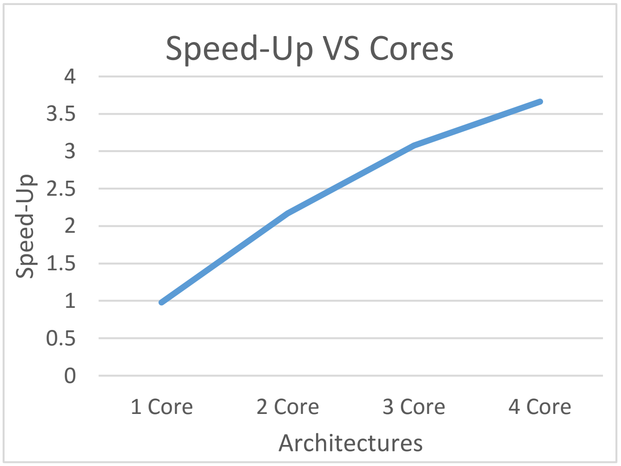
A trend can be observed from the above results showing that the parallel algorithm has a significant speed-up when compared to its sequential counterpart. Also, it can be observed that with an increase in the number of cores comes an increase in speed-up in the parallel algorithm. These two points show the worth in parallelizing the median filtering algorithm as there is significant speed-ups leading to an efficient outcome. It can also be said that since there is a trend in speed-up and number of cores, if more cores are introduced into computers in the future this algorithm would lead to even greater efficiency.

The optimal range for data sizes to be most efficient with the parallel algorithm is between 100 000 and 400 000 elements (inp2 noise) at a 2 core architecture but with a higher number of cores, a higher data size can provide a higher speed-up as seen with the 4 core architecture.

For filter sizes on the other hand, it was found that the optimal range is between 17 and 21 as the speed-ups peaked at this range. This could be due to increased complexity in the calculation with a higher filter, a higher filter causes more values to be sorted which is very expensive hence the sequential algorithm becomes slower whilst the parallel algorithm successfully splits the work.

The optimal sequential cut-off was also sought after. It was found that larger cut-offs lead each thread to be doing too much work hence taking longer times to process the median filter. Smaller sequential cut-offs however lead to more efficient results and thus it was determined that 10 000 was the optimal cut-off to use.

Therefore if we run the parallel algorithm with its variables set to their most optimal position on all architectures (1, 2, 3 and 4 Cores) then we obtain.



A trend of increasing speed up being proportional to the number of cores can be observed from the graph on the left, showing that it is beneficial to run the algorithm with more cores. It can also be seen that the speed-up produced by the parallel algorithm is approximately equal to the number of cores of the architecture the algorithm was run on. This shows that the median filtering problem is highly parallelisable and can lead to significant increases in efficiency.

The ideal expected speed-up for this algorithm, and really any parallel algorithm, is for a linear speed-up. That is as number of processors increases, the time taken for algorithm decreases proportional to the number of processors. Hence the ideal speed-up for the 1, 2, 3 and 4 Core architecture is 1, 2, 3 and 4 speed-up respectively. The actual speed-ups attained for each architecture is summarised in the table below.

$\boxed{P \uparrow \text{ therefore } \frac{T_1}{P} \downarrow}$

Number of Cores	Expected Speed-Up	Actual Speed-Up
1	1	0.977635922
2	2	2.170594026
3	3	3.076983568
4	4	3.662176651

The actual speed-ups obtained are relatively close to the expected speed-up but when they do fall short of the expected value, it is due to the sequential portion of the code. An algorithm can't be fully parallelisable as there has to be a sequential part to a thread hence there must be parallelism (maximum possible speed-up).

The parallelism can be found with $\frac{T_1}{T_\infty}$, this will tell us the maximum speed that is attainable with this median filtering algorithm. We can use Amdahl's Law that states that $\frac{T_1}{T_p} = \frac{1}{(s+\frac{(1-s)}{p}}$ to find what the sequential portion is. We will use the 4 Core speed-up

$$3.662176651 = \frac{1}{(s+\frac{(1-s)}{p})}$$

Therefore s = 0.0307

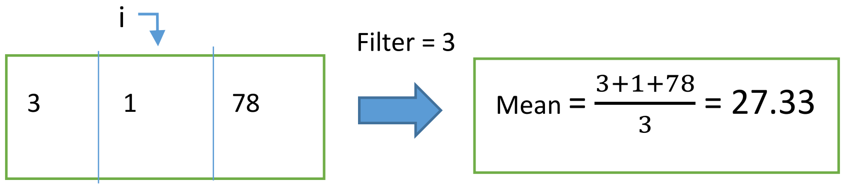
Amdahl's law also states $Parellelism = \frac{T_1}{T_\infty} = \frac{1}{s} = \frac{1}{0.0307}$

Hence Parallelism = 32.52156

So the maximum speed-up possible for this algorithm is a speed-up of 32.5 which could only be attained with more than 32 processors. After 32 processors, a higher speed-up cannot be attained and parallelisation is not as worth it.

Extension to Assignment – Mean Filtering

A mean filter functionality was added to the algorithm so as to be able to compare the performance of median filtering to mean filtering. Mean filtering is replacing an elements value with the mean of its immediate neighbours at a specified filter.



In theory, the mean filter is a lot less computationally heavy as opposed to the median filtering which would be sorting an array more than once in every thread. Hence the mean filter should not speed up by much in a parallel algorithm since there is less work that threads will end up doing.

The mean filtering was run at the same optimal performance variables as the above so as to keep fairness when making comparisons. The average speed up over 1000 trials ends up being 1.055 which shows that the parallel algorithm runs at a similar speed as the sequential. This is possibly due to the fact that mean filtering computations can be completed quickly, so much so that the parallel algorithm can't really produce a significant speed-up (i.e the parallel algorithm can't possible get any faster). If this were true then the overheads of creating threads overpowers the work being done in the thread itself hence for a larger data size we would expect a decrease in speed-up as more threads are being created. A larger data size of 2 million elements is tested and the average speed-up is found to be 0.813 which is lower than the smaller data size array. Mean filtering computations therefore are overpowered by the parallel overheads to do work on the noise leading to poorer results.

It can be seen from the above results that it is not beneficial to parallelise a mean filtering algorithm as there is little to no speed-up when processing the algorithm. This means a parallel approach should be emphasized more with median filtering over mean filtering (Only useful with mean filtering at extremely large data sizes).

Conclusion

In conclusion, a parallel algorithm solution to the median filtering problem is an efficient one and can prove useful if used in the right scenarios. The optimal variables for filter size, data size and sequential cut-off where found to be 19, medium sized (100 000 – 400 000 elements) and 1000 respectively. If a parallel algorithm is used with similar set variables with a high core machine, an impressive speed-up can be achieved to assist in large median filtering computations.

It was also observed that with an increase in number of cores came an increase in speed-up such that a 2-core architecture achieved a max speed-up of 2 while a 4-core architecture could achieved a max speed-up of 4 hence the number of cores is proportional to the speed-up at least till up to around 32 cores.

Parallel algorithms are not always the best way to go either as the speed-up with smaller files at low filters becomes insignificant. Also, an algorithm that isn’t computation heavy might not experience a significant speed-up when parallelised. This can be seen from the mean filtering algorithm results where there is barely speed-ups with the parallel algorithm.