INTRODUCTION

The aim of the project is to smooth one dimensional time series data using a median filter both in parallel and sequential programming to check which program execution time is faster. The median filter slides over the data item by item, generating new data set, where each item is replaced by the median of the neighbouring entries. Parallel programing is a type of programming in which many calculations or the execution of processes are carried out simultaneously. In contrast, sequential or serial programming is where the execution of a process are carries out one item at a time. The parallel and sequential programs will be applied with different array sizes, median filter size and threads (sequential cut-off) and their executions timed. The parallel program will use Java Fork/Join framework to parallelize the median filter operation using a divide-and-conquer algorithm. The aim of the divide-and-conquer algorithms that use threads to produce results more quickly.

METHOD

I applied a range of data of different array size (25 - 1 million), median filter size and numbers of threads (500000-650000) to the parallel and serial program. The execution was timed and reported the speedup of the parallel relative to the serial implementation. The timing was done 5 times and the timing was restricted to the median filter code. A call to System.gc() to minimize the likelihood that the garbage collector will run during the execution , this was outside the time measuring method.

The code was tested on two different dual-cored machine architectures.

Sequential median filter:

```
 tick();
 // adding the start boundary
 for(int i = 0; i<width;i++){
     outputArray.add(inputArray.get(i));
 }

  //getting the elements in the array of size filter by visiting each element
   for(int i = 0; i< inputArray.size()-filter+1;i++)
  {
      for (int z = i;z<i+filter;z++){
        fElements.add(inputArray.get(z));
      }
     // System.out.println(fElements);                    // print out the filtered elements
      Collections.sort(fElements);                       // sorting the filtered elements to be able to get the

      outputArray.add(fElements.get(mid-1));             // adding median elements to output array
      fElements.clear();                                 // clear the f elements list for the next elements

  }

// adding the end boundary
  for(int i = inputArray.size()-width; i<inputArray.size();i++){
     outputArray.add(inputArray.get(i));
 }
float time = toc();
```

The code above is the sequential median filter code. After the array with data items is populated the outputArray of size of the input array, is created to store the smoothed input array. The timing is started (tick()), the first for loop adds the lower boundary/boundaries of the outputArray by taking the first item in the inputArray up to but excluding the size of the median filter.

The second for loop, then does the smoothing of inputArray by taking elements from it of the size size of the median filter and stores in an array(fElemennts) and then sorts the array and add the median of the array to outputArray.

The third for loop does the same as the first for loop but it adds the higher boundaries to the outputArray, the filtered array. The timing of the execution is then stopped.

Parallel median filter:

```
protected ArrayList<Double> compute(){// return answer - instead of run
      if((hi-lo) < SEQUENTIAL_CUTOFF) {
      ArrayList<Double> outputArray= new ArrayList<Double>();
      ArrayList<Double> fElements= new ArrayList<Double>(filter);                   //stores the elements of si
      int mid = (filter+1)/2;

     // adding the start boundary
      for(int i = 0; i<width;i++){
          outputArray.add(inputArr.get(i));
      }

       //getting the elements in the array of size filter by visiting each element
        for(int i = 0; i< inputArr.size()-filter+1;i++)
       {
           for (int z = i;z<i+filter;z++){
             fElements.add(inputArr.get(z));
           }
          // System.out.println(fElements);                  // print out the filtered elements
           Collections.sort(fElements);                      // sorting the filtered elements to be able to ge

           outputArray.add(fElements.get(mid-1));            // adding median elements to output array
           fElements.clear();                                // clear the f elements list for the next element

       }

     // adding the end boundary
       for(int i = inputArr.size()-width; i<inputArr.size();i++){
          outputArray.add(inputArr.get(i));
      }
       return outputArray;


  else {
         Assign3 left = new Assign3(inputArr,lo,(hi+lo)/2,filter);
         Assign3 right= new Assign3(inputArr,(hi+lo)/2,hi,filter);
         // order of next 4 lines
        // essential ï¿½ why?
        left.fork();
        ArrayList<Double> rightAns = right.compute();
        ArrayList<Double> leftAns  = left.join();
        leftAns.addAll(rightAns);
        return leftAns;
     }

   }

}
```

The above code is the parallel median filter. The if part is executed if the result, (hi-lo), the higher boundary less lower boundary is less than the sequential cutoff(number of threads). This part of the code does the same thing as the sequential median filter smooth the array.

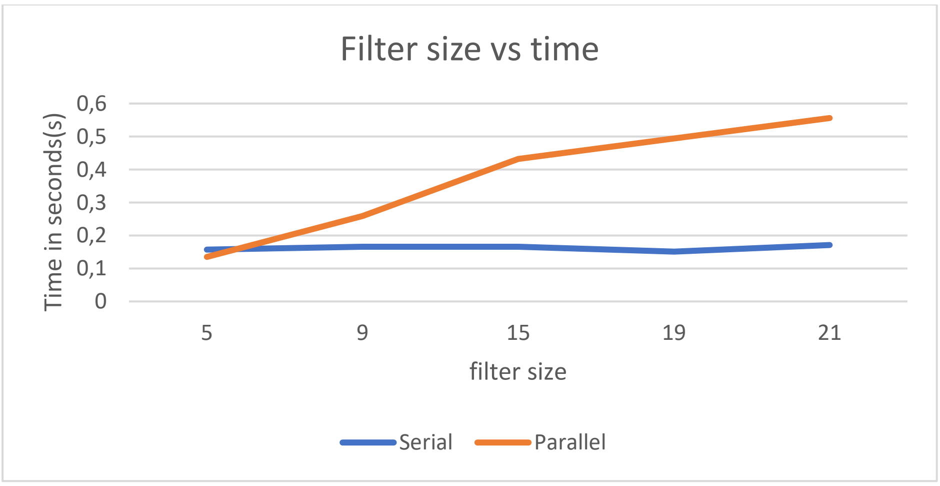
The else part is executed if the, `if((hi-lo) < SEQUENTIAL_CUTOFF`, is false. This section of code splits the inputArray (unfiltered array) into left and right side. The left side splits into threads and calculates the median filter its boundaries being the lower boundary of the unfiltered array and its higher boundary the sum of the higher and lower boundary of the unfiltered array divided by 2, so its splits in half. The left side executes and waits for the right side. The right side splits into threads and calculates the median filter its boundaries being the sum of the higher and lower boundary of the unfiltered array divided by 2 and the higher being the higher boundary of the unfiltered array. The right side executes and goes back to check the sequential part. If it is false it will continue executing, however if it's true the sequential part will be executed, this will be done until the right side is done threading. The result of the right side is the added to the left side. The execution is timed in the main java class.

RESULTS AND DISCUSSION

1. Changing filter size, array size and keeping the Cut-off constant at 600 000.

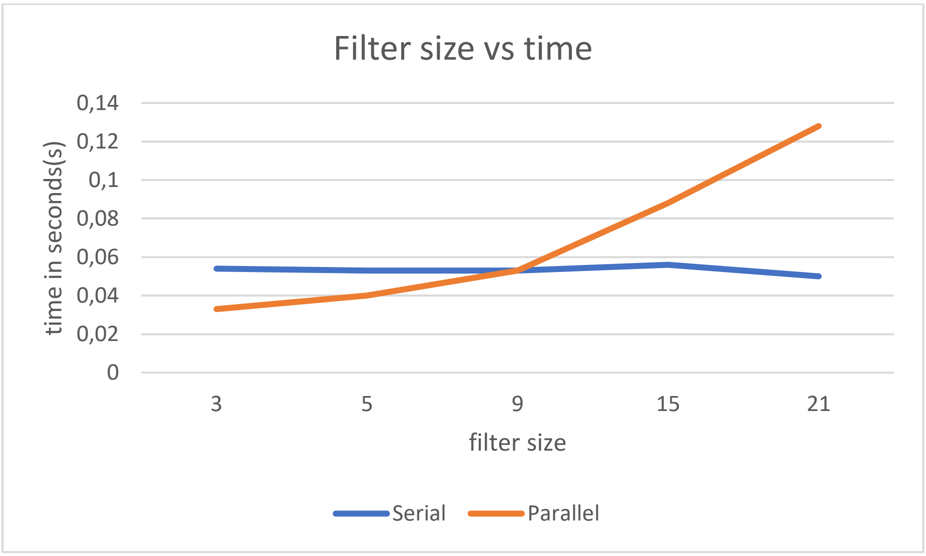
Filter size vs time (Arraysize 1 000 000)

filter size	Parallel (time seconds)	Serial (time seconds)
5	0.135	0.1571
9	0.259	0.166
15	0.432	0.166
19	0.494	0.151
21	0.556	0.171



(Array size 100 000)

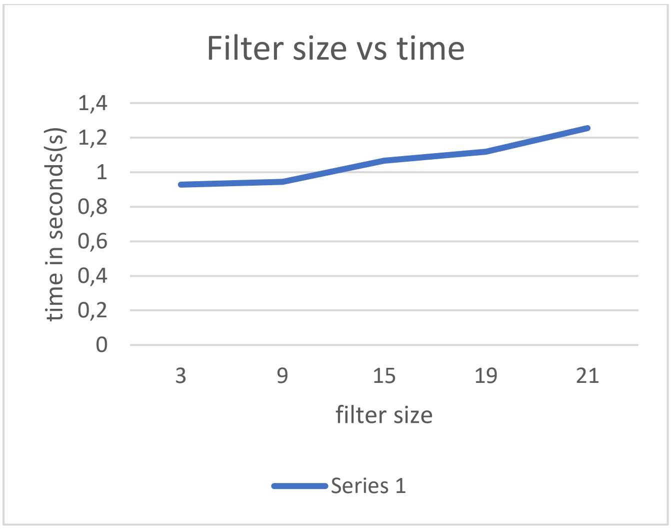
filter size	Parallel (time seconds)	Serial (time seconds)
3	0.033	0.054
5	0.04	0.053
9	0.053	0.053
15	0.088	0.056
21	0.128	0.05



2. Changing the cut-off and keeping the arraysize constant at 1 000 0000 elements.

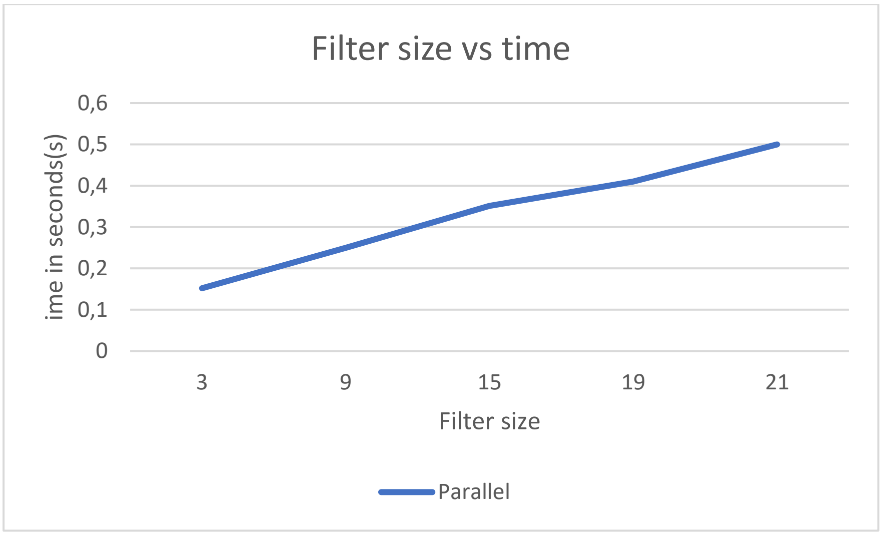
Filter width vs time (cutoff 50 000)

filter size	Parallel(time seconds)
3	0.928
9	0.944
15	1.066
19	1.119
21	1.255

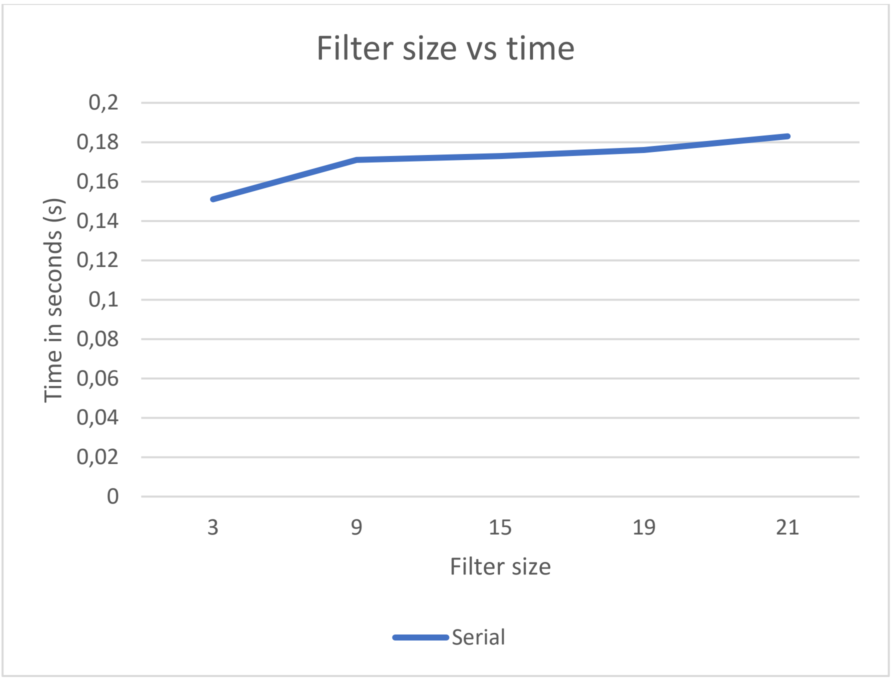


Filter width vs time (cutoff 500 000)

filter size	Parallel(time seconds)
3	0.152
9	0.25
15	0.351
19	0.41
21	0.5



filter size	Serial(time seconds)
3	0.151
9	0.171
15	0.173
19	0.176
21	0.183



- Looking at results between parallel and serial time. The parallel time is less when its starts with a small filter size relative to the serial time but as the size of the filter increases the serial time is less or faster relative to the parallel time. the serial time is less relative to the parallel time overall, when the cutoff(600 000) is the same for parallel with different array size (of 1000 000 and 100 000) and changing the median filter size.
- Looking at the results for the parallel program when cutoff (50 000 and 500 000) is changed with the array size constant at 1000000 elements and changing filter size. Serial program with array of 1000000 elements and also changing filter size.
  - The parallel program with cutoff of 500000 works faster than with cutoff of 50 000.
  - The parallel program time (cutoff 500 000) is faster than the serial program time of the same array elements of 1000 000.
  - Thus, it is worth using parallelization in java to tackle this problem but ensuring the cutoff time is large for an array with a large size.
  - For this problem, the optimal cutoff for this problem is 500 000 (half the size of the array size).

CONCLUSIONS

- it is efficient using parallelization in java to do many calculations at the same time or simultaneously but ensuring the cutoff time is large for an array with a large size.
- the cutoff size will determine the number of threads the parallel programs is going to create.
- Many threads can increase the time of execution.
- parallel programming execution is faster than serial programming.
- It is not ideal to use serial programming to do many calculations as the serial process executes item by item.
- Applying the code in different machines with different number of cores, shows different results. The results applied to a machine with 2 cores give execution time which is less relative to execution time given by a machine with more than 2 cores.