CSC2002S - Assignment 1 2015

Anna Borysova
BRYANN004

Introduction

Median filtering is the process of replacing every data point in a set of data with the median of the neighbouring data points. It is often used to reduce noise in audio files and images. In this investigation, we have used several one dimensional data sets to investigate the feasibility of using a parallel solution to this task. The range of data considered "neighbouring" will be experimented with, ranging from 3 to 21 (odd integers only). The motivation behind investigating parallelism is that processing speed is going to increase by adding more processors rather than by improving the efficiency of one. We are in the twilight between sequential programming and parallelism and this report aims to determine where this median filtering algorithm should fall to increase efficiency as the problem size tends to infinity.

Methods

The sequential solution was coded as simply and clearly as possible to allow for easiest interpretation. It consists of two classes: the `SerialMedianFilterUI` class and the `SerialMedianFilter`. `SerialMedianFilterUI` contains the `main` method which deals with the reading and writing to file, while `SerialMedianFilter` stores the array to be filtered, and filters it using the public method `filter()` and two private helper methods: `cutArray(int index)` and `median(float[] array)`.

The parallel solution adapts the serial solution to use `RecursiveAction`. The main class, now called `ParallelMedianFilterUI` now has a static `ForkJoinPool` variable to allow the main method to call `invoke` on a thread of type `MedianFilterThread` which is an adaptation of `SerialMedianFilter`. Given below is the implementation of the `compute()` method:

```
protected void compute(){
    if(hi - lo < SEQUENTIAL_CUTOFF) {
        for (int i=lo; i<hi; i++){
            if (i>=side && i<size-side){
                retArr[i] = this.median(this.cutArray(i));
            }
            else{
                retArr[i] = array[i];
            }
        }
    }
    else {
        MedianFilterThread left = new MedianFilterThread(array, retArr,
          filterSize, lo,(hi+lo)/2);
        MedianFilterThread right= new MedianFilterThread(array, retArr,
          filterSize, (hi+lo)/2,hi);
        left.fork();
        right.compute();
        left.join();
    }
```

}

This implementation is a classic example of the divide-and-conquer strategy for the Java ForkJoin framework. The output and input arrays are stored once and only the references get passed down the threads, thus saving space and time.[1]

There is no dependency between the sequential and parallel solutions so as to make testing easier.

To test the validity of the parallel algorithm the output file of `ParallelMedianFilterUI` was compared to the output file of `SerialMedianFilterUI` using an online difference checker[2], for every filter size for file inp1.txt. The basic validity of the sequential algorithm is assumed after checking it against the example array [280631].

A separate testing class was used, `SerialVParallel` which runs the main methods of `SerialMedianFilterUI` and `ParallelMedianFilterUI` with different parameters, times each iteration, averages them, and outputs the average. The basic structure is as follows:

```
public static void main(String[] args) throws FileNotFoundException {
    for (String file: files){
        float totFileSpeedup = 0;
        for (String filter: filters){
            long serialTotal = 0;
            long parallelTotal = 0;
            for (int i=0; i<ITERATIONS; i++){
                long start = System.currentTimeMillis();
                String[] param = {file, filter, "out.txt"};
                SerialMedianFilterUI.main(param);
                long serialTime = System.currentTimeMillis()-start;
                serialTotal += serialTime;

                start = System.currentTimeMillis();
                ParallelMedianFilterUI.main(param);
                long parallelTime = System.currentTimeMillis()-start;
                parallelTotal += parallelTime;
            }
            float avFilterSpeedup = serialTotal/(float)parallelTotal;
            System.out.println("* Speedup for filter: " + avFilterSpeedup);
            totFileSpeedup += avFilterSpeedup;
        }
        System.out.println("* Speedup for File: " + totFileSpeedup/filters.length);
    }
}
```

The cutoff points were tested manually so as to adhere to input specifications. The parallel and sequential solutions were run in the same loop so as to keep them alternating and reduce the effect any anomalies in processing speed. The `ITERATIONS` parameter was set to 100 for all tests so as to decrease variation, and the results (the speedup averages for each filter and file) were recorded manually into a .ods document. The relevant specifications for the machine on which these tests were run are as follows:

```
CPU(s):                 4
Thread(s) per core:     2
CPU MHz:                800.000
```

[1]The Doctor approves.
[2]https://www.diffchecker.com/diff

Results and Discussion

The first set of tests were aimed to determine at which sequential cutoff point the rest of the tests should run. While it is possible that different combinations of filter sizes and file sizes may give different answers, an approximate value was all that was searched for in this case. The inner loop (for every filter size) ran for 50 iterations. The average for the file (inp.txt in this case) over all the filter sizes was taken and plotted against the sequential cutoff values tested to yield the graph shown in Figure 1.

These results are rather suspect, because the seemingly optimal cutoff value is so low. Lower values were not tested because it may have caused issues with the larger filter sizes. Regardless of the suspicion, the rest of the tests were carried out with the cutoff value 25.

The rest of the tests resulted in bad news for parallelism, or at least for this problem. While the speedup did, on average, increase as filter size increased for every file, as expected, the only visible difference between the different test files, and hence test sizes was the variation of the data. The very erratic results for the first two files (inp1.txt and inp1B.txt) are odd because the data points were all an average of a hundred runs. This means that the variation in speedup is so high, that a hundred runs can only flatten it for the larger file sizes. As for the overall increase in speed, even the largest speedup was not half of what the optimal speedup is for the architecture these tests were being run on (four cores should mean four times speedup). In these tests, the parallel solution did not reach two times speedup.[3] This means that the parallel portion of the code (actual median filtering) does not grow significantly faster than the sequential portion (reading in of the file). This is a necessity for parallelism in a given program to work, otherwise Amdahls's Law dominates. It is not worth using parallelisation to tackle this problem in Java, both for the ridiculous (relative to the tiny speedup) variation and the poor/nonexistent increase in speedup as the file size increases.

There are some other curiosities in the data. It is tempting to dismiss the increased speedup for filter size 3 as just a result of random variation, but this effect held true even for inp4.txt, which has the smallest random variation. This is worth further investigation. Another interesting quirk is that the speedup for file inp4.txt was significantly lower than for the other files. This is possibly because the file is larger than the others, and so the sequential cutoff of 25 may have been inappropriate. However, each file size is approximately 4.5 times larger than the previous file size, which would imply similar differences in speedup between the other input file sizes should be found due to an increasingly inappropriate sequential cutoff value. Further investigation is needed.

Mean Filtering Extension

The setup for mean filtering was very similar to median filtering, with the only difference being the helper method which calculated the median of an array. It was changed to calculate the mean.

The same tests were run, for all the filter sizes and files, and with a sequential cutoff of 25. The number of iterations in this case however were only 50, so a greater variation is to be expected.

As the results in Figure 3 show, the speedup is even worse for mean filtering, and does not increase with filter size. The same pattern of more stable results with greater file sizes seems to hold with mean filtering, however, with mean filtering, the speedup for inp4.txt is not significantly lower as it was in median filtering. This may imply something particular to median filtering, but more likely some defect in the data.

Conclusions

An argument can be made for using the parallel approach for the median filtering problem in java, because on average, there is an speedup, so it does not worsen the speed. It is not a particularly difficult improvement to make, so perhaps it is worth it. However, I would not recommend it. The increase is tiny, and the payoff

[3] However, the variation in speedup for smaller file sizes is huge, and if one forgets about scientific integrity for a moment, these tests could be run until a two or maybe even a three times speedup pops up.

for a greater number of processors as the problem size tends to infinity would be negligible. The sequential part of this problem does not outweigh the parallel part, but it is uncomfortably close.

References

- https://www.diffchecker.com/diff

- https://en.wikipedia.org/wiki/Median_filter

Figure 1:

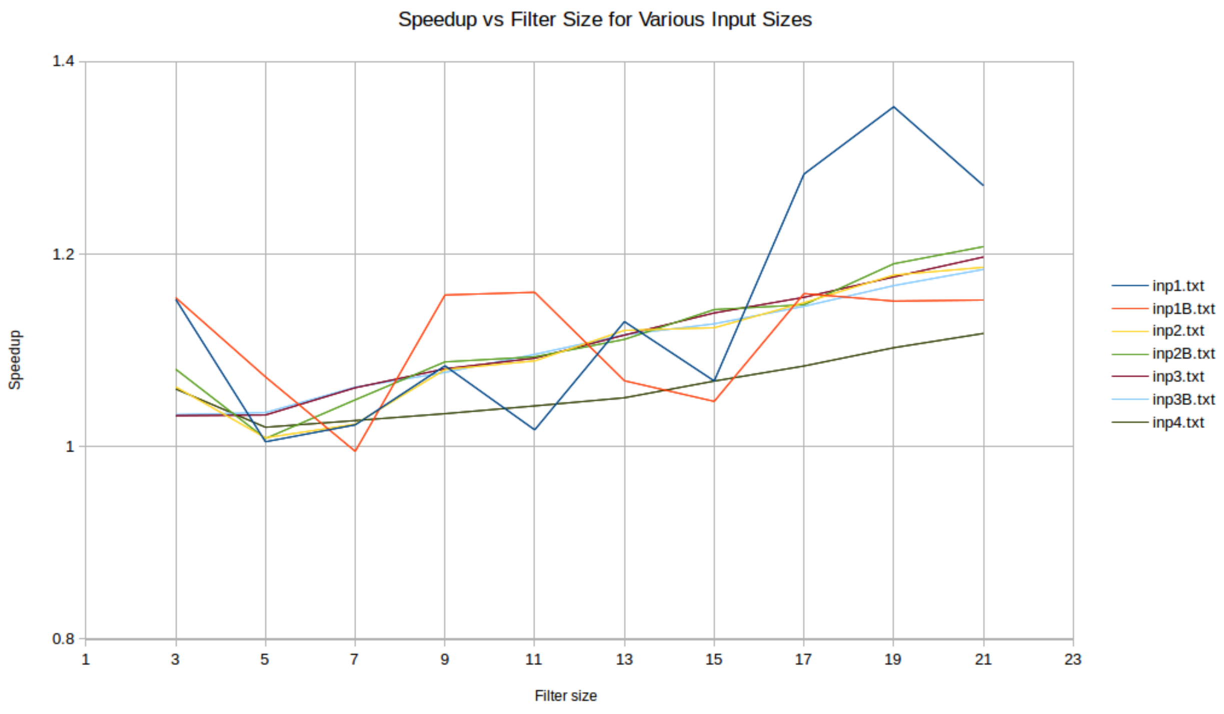


Figure 2:

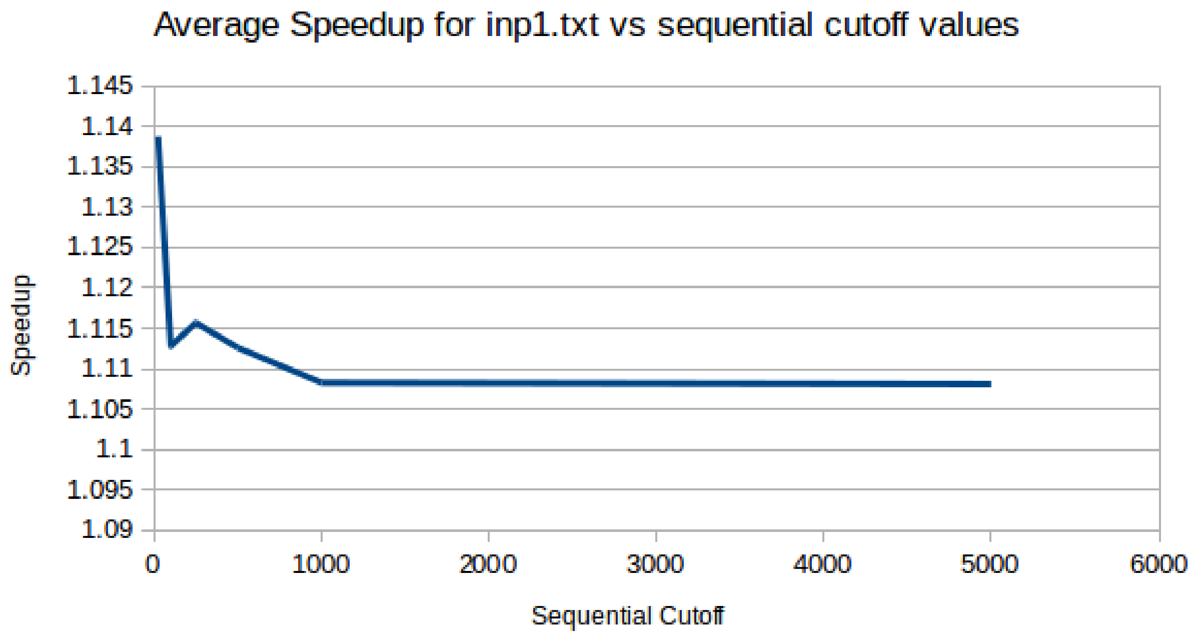
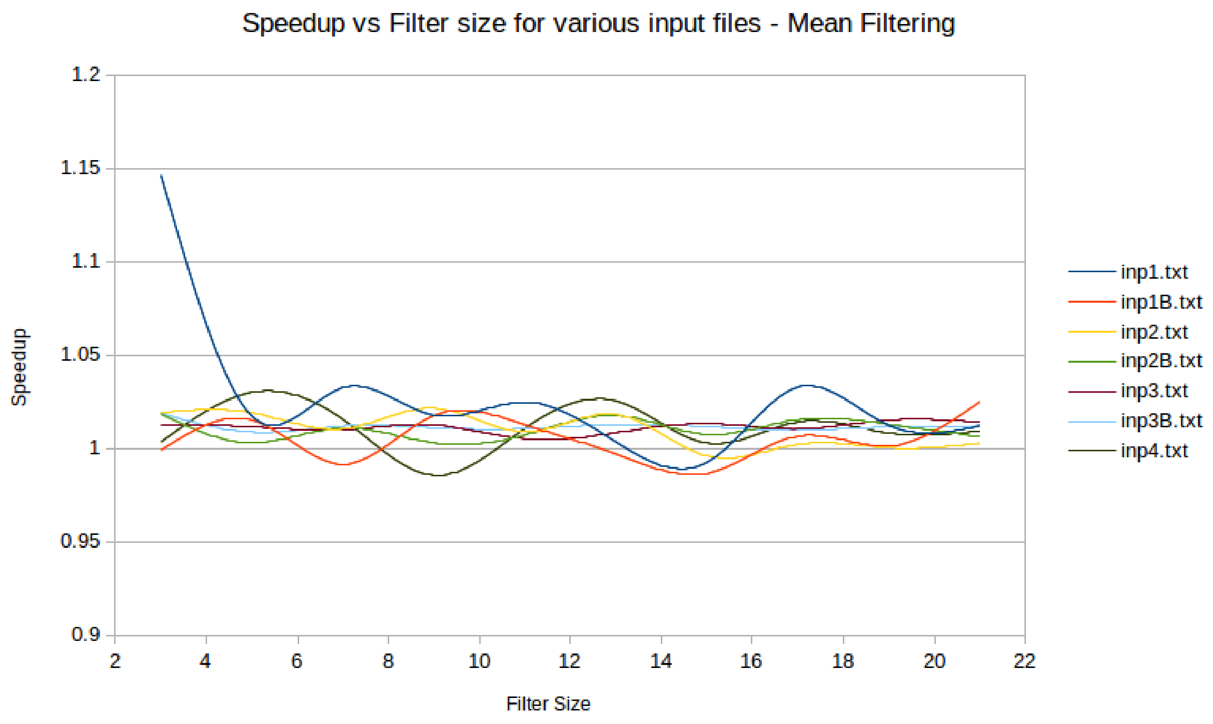


Figure 3:


Speedup vs Filter size for various input files - Mean Filtering
Speedup
Filter Size
1.2
1.15
1.1
1.05
1
0.95
0.9
2
4
6
8
10
12
14
16
18
20
22
inp1.txt
inp1B.txt
inp2.txt
inp2B.txt
inp3.txt
inp3B.txt
inp4.txt