CSC2002S Aidan de Nobrega DNBAID001

Assignment 1: Parallelism 05/08/2015

Introduction

The aim of this assignment is to gain a fundamental understanding of multithreaded parallelism by implementing sequential and parallel solutions to the median filtering problem. Median filtering is a nonlinear digital preprocessing technique used in sound mastering and statistics (among other disciplines) to "smooth" time series data or remove random noise from a data set (Blacklemon67, 2015). The employed method involves a filter window[i] sliding over the data and replacing all elements with their respective medians (calculated using neighbouring elements). With larger windows, computation becomes more and more expensive.

This assignment aims to implement a parallel algorithm that will decrease computation time relative to the sequential algorithm. We will first implement the serial method, then a parallel method that utilises divide-and-conquer with Java's Fork/Join Framework, and lastly we will implement a method using standard threads. Although Amdahl's Law defines a theoretical upper bound for the speedup of a program due to limitations inherent in the sequential part of the program[ii], we still expect both parallel algorithms to run significantly faster than the serial one. And bearing in mind the fact that Java's standard threads are considered "heavyweight" (Farnham, 2011), we expect our second implementation to run faster than the last.

Lastly, we will experiment with different parameters (sequential cutoffs, window size, filtering type, etc.) in order to optimise our program and gain greater understanding of the various multithreading techniques. Filter(ing) type refers to whether we will be filtering according to the median or the mean. The mean filtering system has been implemented as a peripheral component and holds no bearing on the aims of this assignment, so it will not be discussed in any great detail.

Methods

Initially, the data is read from the input file and an array is instantiated with those data as elements. The median filtering is such that the first few elements in the array and the last few that fall outside any filter window are unprocessed. For example, if the filter size is set at 5, the first 2 (and last 2) elements in the array are transferred unchanged to the final array at the same indices.

All elements within the borders outlined above will be processed by the filter – consider this set of elements as the initial subarray (see *Figure 1*).

[i] The size of the window is user-defined and particular to the data in question.

[ii] Amdahl's Law is applicable to parallelism using multiple processors.

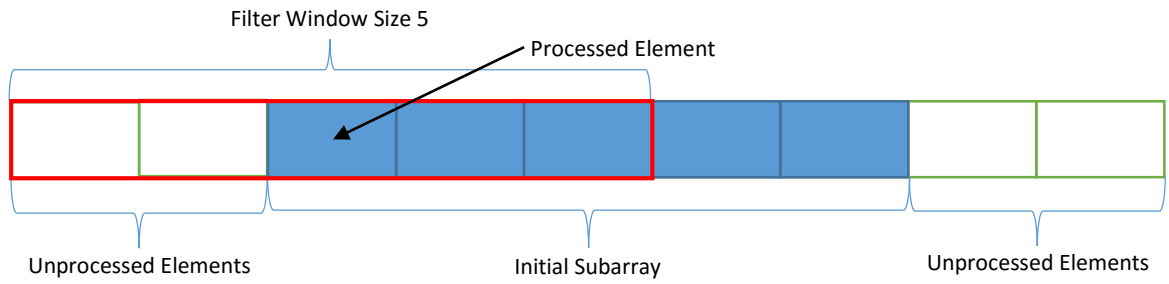


Figure 1: The initial array of data

The timer is started immediately before the predetermined algorithm starts and stops immediately afterwards, before the values are printed to file. This allows us to focus our attention on the differences in the algorithms and minimise error variance in our tests.

To sequentially filter the data, a `FilterObject` object is instantiated with the starting and ending indices of the initial subarray as fields – `lo` and `hi` respectively. The `FilterObject` invokes its `seqFilter()` method (see *Code Example 1*) which traverses the subarray from `lo` to `hi` (inclusive) and filters each element (within a window) according to the determined filter size and type.

```
void seqFilter()
{
    for (int i = lo; i < hi; i++)
    {
        double[] window = new double[inputSize];
        for k in range(i-(filterSize/2), i+(filterSize/2)+1)
        {
            window.add(startArray[k]); //add to first empty slot
        }
        finalArray[i] = getMedian(window));
    }
}

static double getMedian(double[] array) //getMean() is similar
{
    sort(array);
    return array[filterSize/2];
}
```

Code Example 1: Pseudocode for `seqFilter()` and static helper method `getMedian()`

Note that the subarray being filtered is not actually an array object. Instead, it is merely an index reference to the middle part of the original array excluding the indices at the extremes that are not processed. This is an important distinction, as it allows us to parallelise the algorithm without having to worry about `NullPointerException`s being thrown.

We use `ArrayList`s rather than standard arrays in this implementation for two reasons:

1. We can use one fewer for-loop by not having to add items to window by index.
2. We can more efficiently get and set items to/from the initial array and the final array.

After the subarray has been filtered and the elements added to the final array, the timer is stopped for that particular run[i]. If it is the last run, the program outputs the elements from the final array to the output file in the same format as the input file.

Parallelising the data is not an overtly difficult task. In fact, it is near-embarrassing as there is no dependency among the threads and each thread works independently on its allocated task. This is because we refrain from splitting the array into two subarrays to be processed in parallel. Instead, we split indices, i.e., we make thread 1 process the first half of the indices and we make thread 2 process the second half from a single array and then both threads independently set the new values to the same indices in the final array.

To begin the parallel algorithm, our `ForkJoinPool` calls `invoke()` with our `FilterObject` object as its parameter. Our object's `compute()` method (an implementation that overrides the same method in RecursiveAction) is called and we observe the following:

- If the subarray is "small enough", i.e., if the difference between our upper bound and our lower bound for the subarray is below the sequential threshold, the subarray is filtered sequentially using the algorithm in *Code Example 1*.
- Otherwise our `FilterObject` object is split in two as described above. Both objects make a recursive call, but the object referencing the smaller indices creates a new thread while the other object carries on along its current thread.

```
void compute()
{
    if (hi - lo < SEQUENTIAL_THRESHOLD)
    {
        seqFilter(); //see Code Example 1
    }
    else
    {
        FilterObject left = new FilterObject(lo,(hi+lo)/2);
        FilterObject right = new FilterObject((hi+lo)/2,hi);
        left.fork(); //recursive call with thread creation
        right.compute();
        left.join();
    }
}
```

Code Example 2: Pseudocode for `compute()` in `FilterObject`

In order to validate the correctness of the parallel approach, one can make use of the `FileChecker` class. A user can pass two files as input and the program will compare the two to make sure that all elements are accounted for and in the correct positions. If the program encounters different file sizes, it will conclude that the files are not equal and exit without processing all the elements. Otherwise, it will create an array for each file and

[i] Each implementation is run 20 times, the slowest time is discarded (it is usually an outlier), and an average is outputted at the end.

compare every pair of elements against each other. If it finds an inconsistency, it will conclude that the files are not equal, state the line at which the inconsistency occurred, and then exit. If the program parses without any issues, it will conclude that the files are equal. We can check that the parallel algorithm is correct by first running the sequential[i] filter on a file, then the parallel filter on the same file, and finally comparing the two using the `FileChecker` class.

The sequential threshold for the divide-and-conquer algorithm is not determined by the user. It is fixed at 100, which is the value that allows the algorithm to run at optimal efficiency[ii]. Similarly the number of threads in the standard threads implementation is determined independently for each run using `Runtime.getRuntime().availableProcessors()`. This method "returns the maximum number of processors available to the Java Virtual Machine at execution time".[iii] The factors determined by the user are the following: the input data file, the output file, the size of the filter window, the algorithm implementation to use, and whether to filter according to median or mean.

The tests were performed on 2 computer architectures to establish how an increase in the number of logical cores can enhance parallel speedup.

Dell Inspiron 3543 i7-5500u (2 cores with hyperthreading → 4 logical cores)

UCT Nightmare Server via SSH (8 logical cores)

The individual cores in the 4-core machine are faster than the ones in the 8-core machine, so we would expect the sequential portion of the program run faster on the 4-core machine. This is because there is only one thread running in serial execution. Taking advantage of multiple cores is the essence of parallelism that is lost on sequential implementations. In contrast, the parallel implementations should run faster on Nightmare because of the additional cores being utilised.

Focusing on median filtering, and setting `FilterObject.SEQUENTIAL_THRESHOLD` to 100[iv], we will first run each implementation on input datasets of varying sizes. For consistency, we will fix the filter size across the different inputs and implementations in order to isolate the input size as the independent factor of concern. Then we will apply all the implementations to the same file but change the size of the filtering window. These tests will give us a good indication of the effect that different factors have on parallel speedup.

There was only one issue I had with my implementation of the parallel algorithm: my run times were absurdly fast relative to the times I was getting for my sequential program – I was getting a speedup close to a factor of 10^4. I knew that the issue lay somewhere along the lines of a data race, but because of how my program was designed, there was no appropriate way to introduce a `join()` command. A few hours of internet research led me to the `Future` interface where I learnt about asynchronous tasks and of ways to verify thread completion. I

[i] The sequential algorithm is trivial enough to not need a validation of its own…I hope.

[ii] In general, testing found the differences between various reasonable sequential thresholds to be negligible. Refer to the results section for relevant data.

[iii] According to the Javadoc for the Runtime class

[iv] Please see results section to see why this value was chosen

learnt that `RecursiveAction` implements `Future`, and consequently, a second query of a `RecursiveAction` object bypasses its computation and returns immediately. This means I had to pass an anonymous `RecursiveAction` object as a parameter of `pool.invoke()`. I had a similar issue with my standard threads implementation. I learnt that I could insert a while-loop that loops until the number of threads has reduced to a single thread, thus ensuring that all branches of the master thread are finished processing.

Results & Discussion

From the get-go, it was immediately obvious that the parallel algorithms were significantly faster than the sequential algorithm given a large enough dataset. This makes sense as one would only bother parallelizing a program if they anticipated very large data sets as inputs. A problem with comparing the algorithms for small inputs is the significant error variance that can occur from things such as background tasks and operating temperature affecting the performance/availability of logical cores. Testing was done using four input files of various sizes[i], all "large enough" to generate consistent behavior from the program.

The first task was to find the sequential cutoff that worked best for the divide-and-conquer algorithm across all datasets and filter sizes. Two files of very different sizes were tested across a range of sequential cutoffs. The two files were processed with different filter sizes.

	10	50	100	500	1000	2000	5000	10000
1999886[5]	225	226	216	219	224	229	239	249
35999[21]	212	211	193	194	196	196	198	200

Figure 2: A comparison of the different sequential thresholds using two files of different sizes and different filter widths. The notation for the data labels is file size[filter size].

[i] Sizes in ascending order: 35999, 100001, 440001, 1999886

From *Figure 2* we can clearly see that, for both files, a sequential cutoff of anything in the range 100 to 500 proved to be the most efficient. In parallel programs, performance tends to be quite similar within a particular range of sequential thresholds. Values lower than this tend to produce a performance deficiency due to thrashing, whereas values higher than this range tend to produce a performance deficiency due to load imbalance (Kumar, et al., 1990). Based on this, we will be using a fixed sequential cutoff of 100 for all the following tests.

We shall now see how the three algorithms compare when executed on input files of various sizes and filter windows of various widths. When running the program, the users declare the input file and the size of the filter they want to apply. For the sake of consistency, when comparing performance across input sizes, the filter size will be kept constant at 11. Similarly, when comparing performance across filter sizes, we will use the same file as the input dataset.

From *Figure 3* it is obvious that both parallel implementations process the dataset faster than the sequential one – with the Fork/Join Framework outperforming standard threads – as expected. An important feature of the results is that the sequential graph and the parallel graphs diverge as the input size increases. This reinforces our supposition that parallelization becomes more viable as the anticipated input size increases.

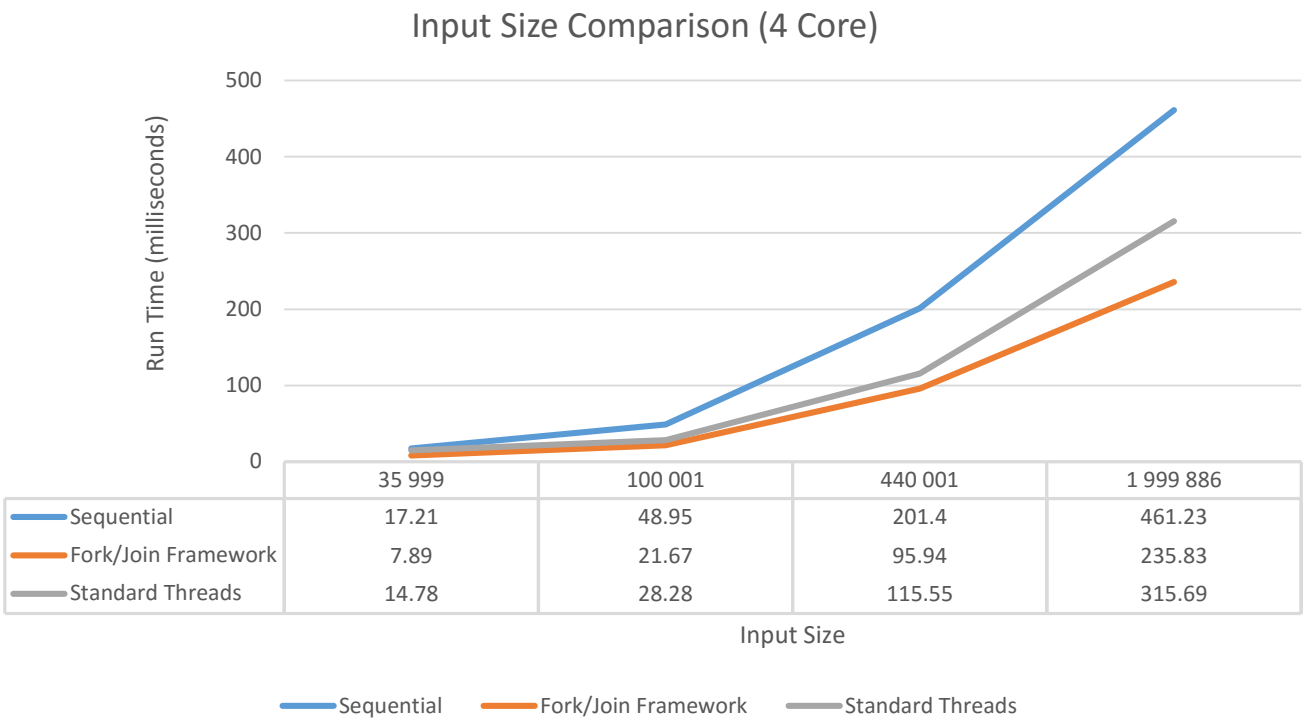
4-Core Machine: The speedup is approximately 2.1x for the Fork/Join Framework and 1.5x for the standard threads[i]. Because the speedup factor is constant, larger input sizes will see a larger "wall time" difference than smaller input sizes. Practically speaking, this is a necessary requirement for parallelised algorithms.

8-Core Machine: The difference on the 8-core machine is considerably more distinct. As stated previously, only one thread executes in a serial run so we would expect Nightmare's sequential run to be slower due to its feebler processors. However, interestingly enough, the parallel implementations run faster on the 8-core machine due to the presence of more available processors. This means that Nightmare can have more threads running concurrently than the 4-core machine. Even though its processors are not as powerful, the benefits of having more cores outweigh its weaknesses. The benefits become even more astounding as we see the performance divergence at the upper end of the chart. The speedup is approximately 4.2x for Fork/Join and 2.7x for standard threads. This tells us that, to some extent, it is better to increase the number of processors than to have more powerful processors. At this point, it is already evidently worth the effort of parallelising the sequential algorithm in order to get improvements of this magnitude.

From *Figure 4* we see a similar pattern in that the parallel implementations are substantially faster than the serial one and the Fork/Join multithreading once again outperforms the standard threads.

4-Core Machine: the performance difference between the sequential and parallel implementations increases significantly as the filter size increases. The speedup increases exponentially as we increase the window size from 3 to 21, initially getting a speedup of 1.8x and finally a speedup of 2.3x between Fork/Join and sequential.

[i] $Speedup = Time_{OLD}/Time_{New}$



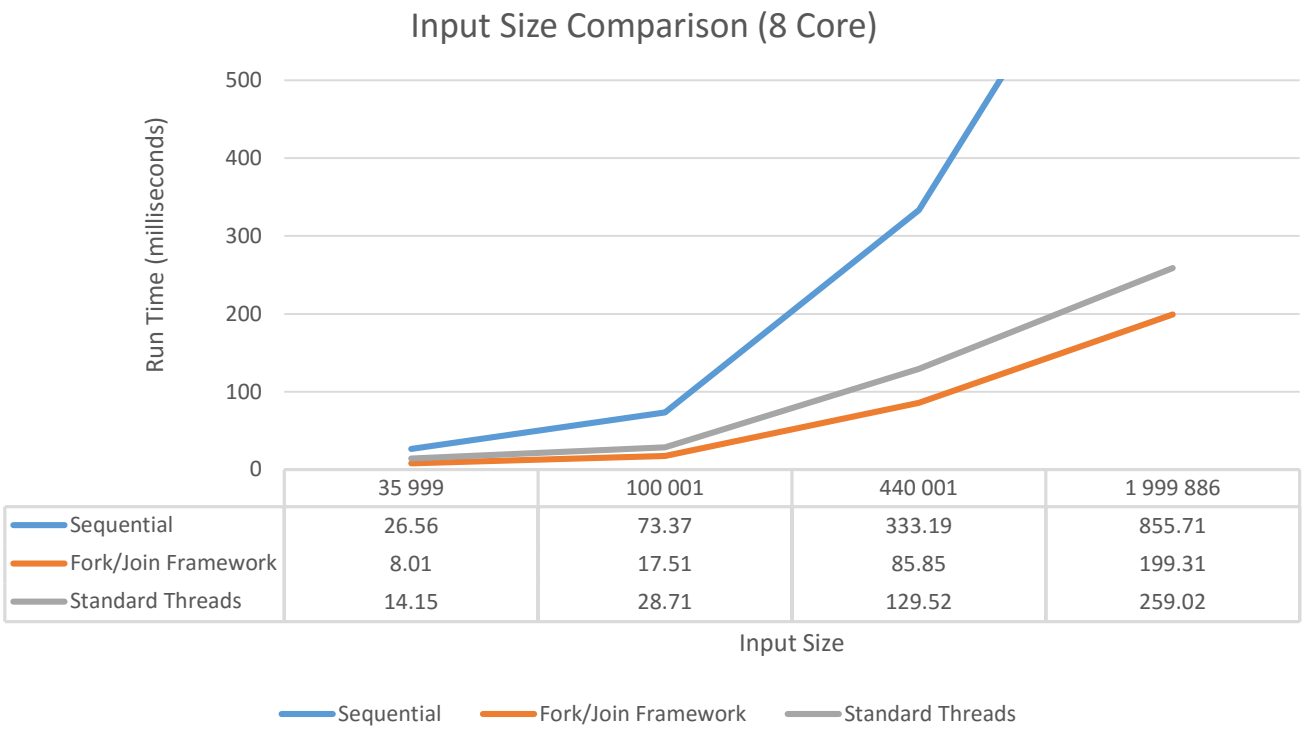


Figure 3: A comparison of the three algorithms across input datasets of various sizes. The first chart is a comparison run on the quad-core machine, whereas the other is a comparison run on Nightmare.

8-Core Machine: Again, because the individual processor is not as powerful on Nightmare as it is on the 4-core, the sequential implementation is almost twice as slow as it is on the 4-core, but the parallel versions are slightly faster. This results in an even wider divergent berth in performance between the sequential and the parallel implementations starting at 3.6x speedup for a window size of 3 and ending with a 4.6x speedup for a window size of 21.

Assuming the entire portion of code within the run timer was parallelised, in an ideal world we would expect to see a theoretical parallel speedup of 4x for the 4-core machine (considering it has 4 times the processing ability once we introduce multithreading) and a speedup of 8x for the 8-core machine. Both discrepancies are primarily attributable to the fact that at no point in the execution of the algorithm do we get to make use of 100% of the machines' potential resources. Nonetheless, the speed up we obtained has proven to be more than satisfactory enough.

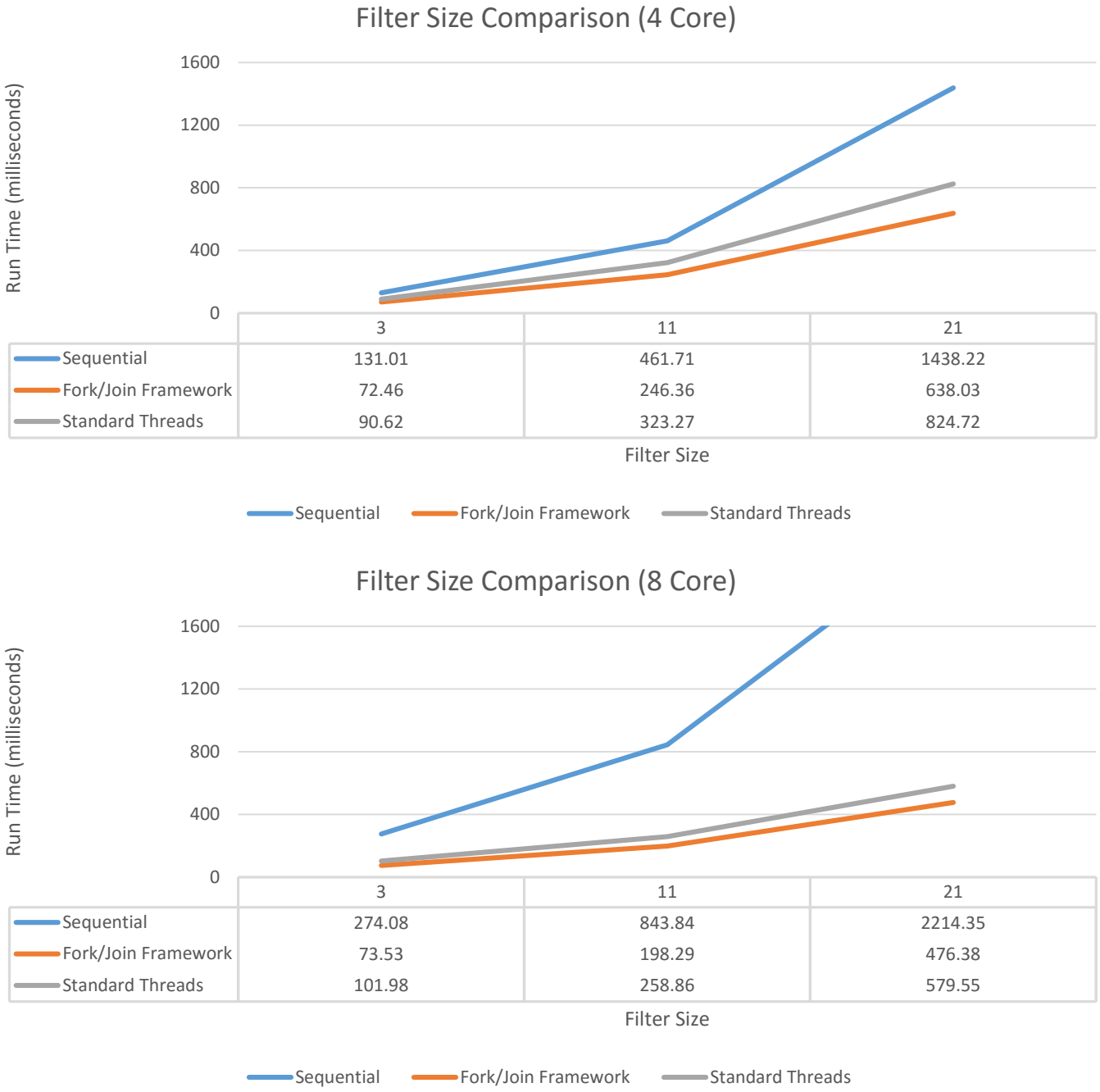


Figure 4: A comparison of the three algorithms across filter windows of varying sizes. The first chart is a comparison run on the quad-core machine, whereas the other is a comparison run on Nightmare.

Conclusions

It is evident from the results that our naïve median filtering algorithm has proven to be worth the effort of parallelising as the performance differences are truly significant. We obtained impressive speedups, with the lightweight Fork/Join threads outpacing the standard threads at every benchmark.

The parallel implementations show the most improvement over sequential when we have larger datasets and when we filter over wider windows. A combination of the two factors would see an unprecedented speedup from the parallel algorithms and would do yet more to cement the case in favour of parallelisation. If the anticipated size of the datasets was not very large and we were filtering over small windows, it might not be worthwhile to parallelise an optimised sequential program. It takes time and the added complexity makes it harder to develop efficient, correct, bug-free code.

Testing on two machines of different architectures showed that multithreaded programs become more and more viable the more processors we have, but we must keep Amdahl's Law in mind regarding the inevitable diminishing returns we will get by introducing more and more processors.

Our first test showed that a sequential threshold of anything between 100 and 500 performs best for our program and anything outside the range will start to see less optimal results, although the differences are negligible within a wider range.

References

Blacklemon67, 2015. *Wikipedia: Median Filter.* [Online]
Available at: https://en.wikipedia.org/wiki/Median_filter
[Accessed 08 August 2015].

Farnham, K., 2011. *Java Weblogs: Editor's Blog.* [Online]
Available at: https://weblogs.java.net/blog/editor/archive/2012/02/29/look-java-thread-overhead
[Accessed 05 August 2015].

Kumar, V., Kanal, L. N. & Gopalakrishnan, P. S., 1990. *Parallel Algorithms for Machine Intelligence and Vision.* 1st ed. New York: Springer-Verlag.