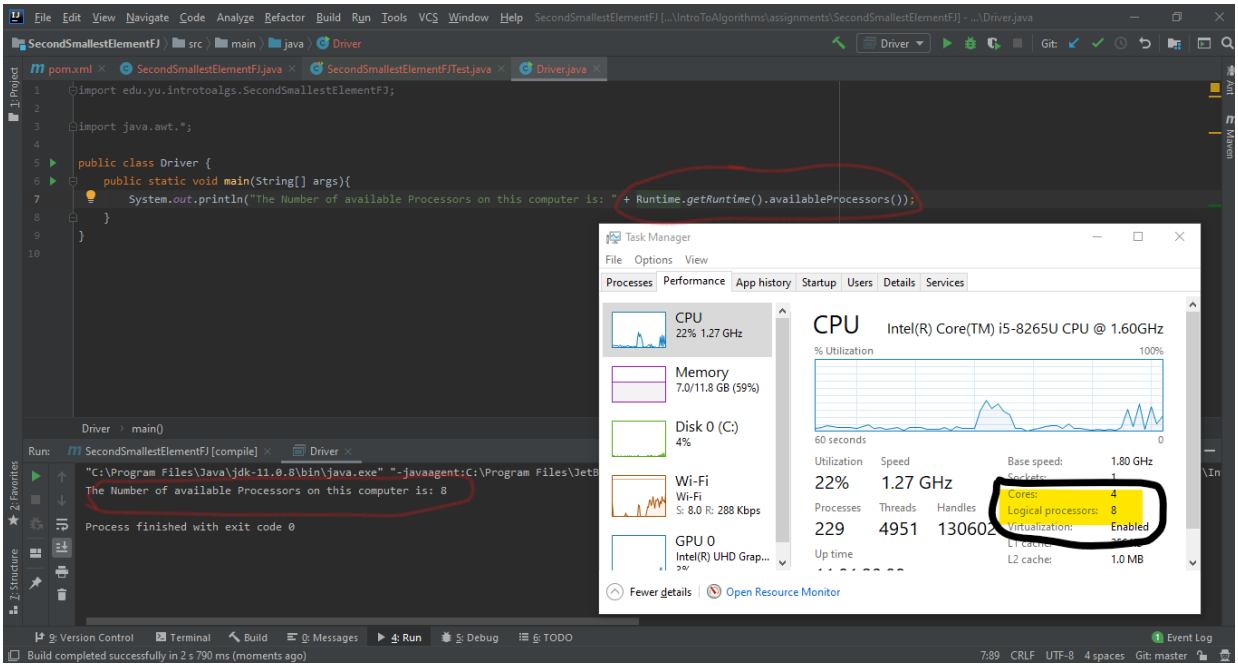
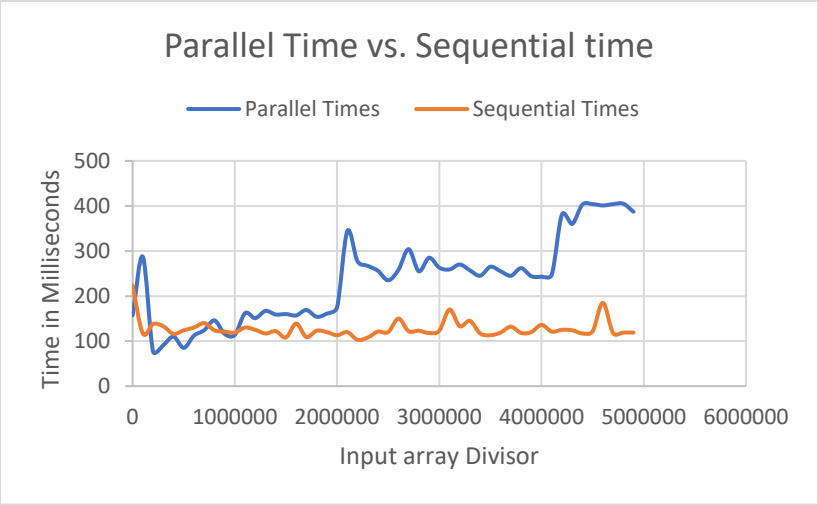
1. Michael Edelman - SecondSmallestElementFJ

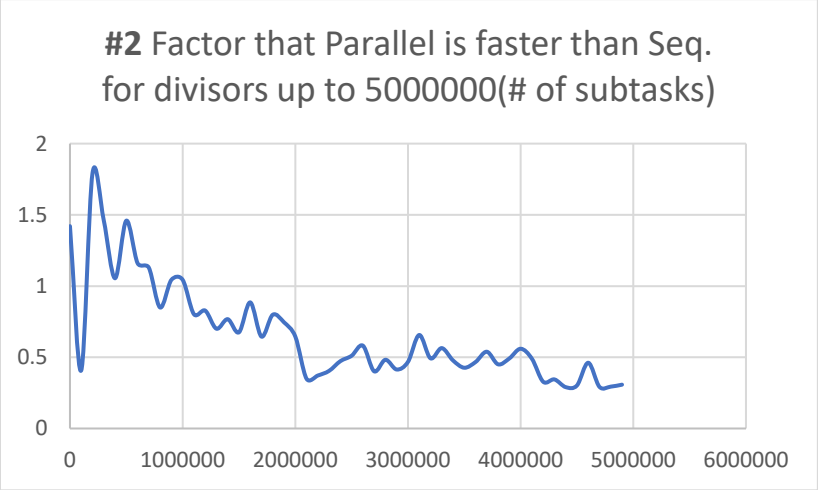


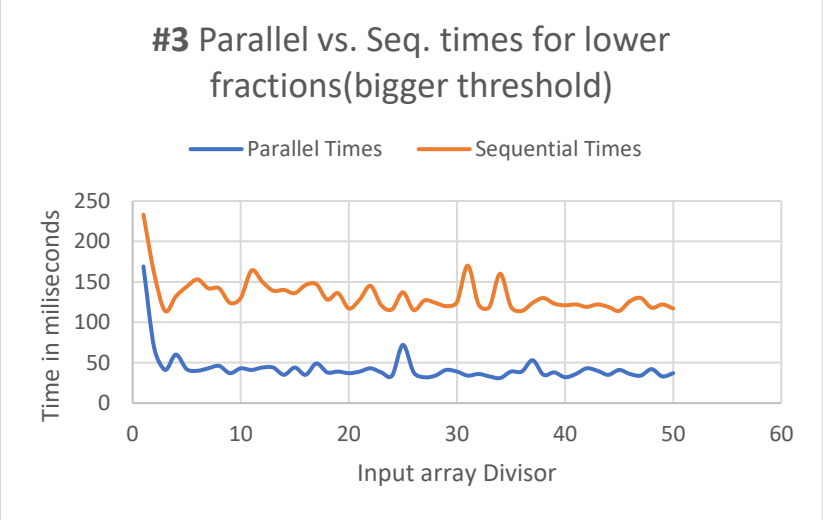
2.

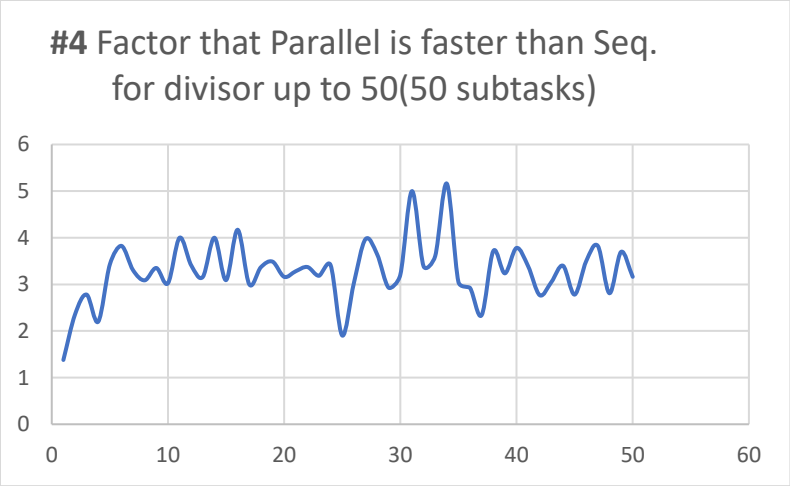
DATA	Argument
doubled array size: 100000 original array size: 50000 Their performance ratio: 1.1307914849518546 doubled array size: 200000 original array size: 100000 Their performance ratio: 1.414187116564417 doubled array size: 400000 original array size: 200000 Their performance ratio: 1.374268708348171 doubled array size: 800000 original array size: 400000 Their performance ratio: 1.7969820359281437 doubled array size: 1600000 original array size: 800000 Their performance ratio: 2.882888549435409 doubled array size: 3200000 original array size: 1600000 Their performance ratio: 1.1610653803806428 doubled array size: 6400000 original array size: 3200000 Their performance ratio: 1.8688200253757983 doubled array size: 12800000 original array size: 6400000 Their performance ratio: 2.318551183710651 doubled array size: 25600000 original array size: 12800000 Their performance ratio: 2.5250975167343532 doubled array size: 51200000 original array size: 25600000 Their performance ratio: 2.9660013946666024 **Average ratio over 10 iterations: 1.9438653396096044**	This data arises from test code that kept doubling the size of the input array (randomized data set each time). The code processes the whole array sequentially by setting the "fraction to apply sequential cut off" to 1.0. This results in a threshold(number of array elements to designate to each processor) the size of the input array which means the whole array is processed sequentially by one core. Big O notation describes the upper bound for the performance of an algorithm based on input size. We need to show that our algorithm is O(n). O(n) means our algorithm's performance is <= constant * n. We can see from our data that as we double the input size the performance is on average doubled. Mathematically this makes sense because for O(2n)/O(n) = 2 which means that when the input size is doubled for an algorithm that has O(n) behavior the increase will be linear, so by a constant of 2. We see this behavior exhibited in our data where the average performance ratio was 1.94 (close to 2). Therefore based on our experimental data we have empirically verified that our algorithm exhibits O(n).

3.


Parallel Time vs. Sequential time
Parallel Times
Sequential Times
Time in Milliseconds
500
400
300
200
100
0
0
1000000
2000000
3000000
4000000
5000000
6000000
Input array Divisor


#2 Factor that Parallel is faster than Seq. for divisors up to 5000000(# of subtasks)
2
1.5
1
0.5
0
0
1000000
2000000
3000000
4000000
5000000
6000000


#3 Parallel vs. Seq. times for lower fractions(bigger threshold)
Parallel Times
Sequential Times
Time in miliseconds
250
200
150
100
50
0
0
10
20
30
40
50
60
Input array Divisor



4. Using the fork/ join framework enabled me to utilize my four cores and as a result saw performance gains of as much as 5 times faster than sequential performance as shown in graph number #4 when the input array was divided into 34 subtasks(a fraction of 1/34). It achieves this by processing the program in parallel with its 4 cores and 8 logical processors and utilizing work-stealing. The average performance increase was 3.28x faster for the parallel vs. the sequential. Observed factors above 4x the speed might be perplexing at first because we only have 4 cores, but might be able to be explained by hyperthreading. According to Intel Hyper-Threading can result in 30% increase in performance of your computer ("How to Determine the Effectiveness of Hyper-Threading Technology with an Application" (intel.com)) This means our theoretical best performance would be 4 + (4 * .3) = 5.2X faster because we have four cores to make it potentially 4X faster and each core is hyper threaded so will have performance boost of (30% of 4). Hp.com said that hyperthreading's benefits are on a case by case basis due to overhead, so this could explain why we only have our average is 3.28X (Hyper-Threading and Everything You Need to Know (hp.com)).

However, I expected a performance degradation at some point due to the propensity for a fine-grained algorithm with a sequential cutoff that's too small to incur a performance penalty. This is due to the overhead of unnecessary thread creation. To see when this occurs, I varied the input array divisor between 1/1 – 1/50000000. This data is contained in graph #1. Its clear that as the input array is divided up more and more finely after about 1/1000000 performance starts to degrade to such an extent that we can see from graph two that the parallelized process is actually almost 4Xs slower than the sequential! (0.25). This observation matches well with the idea that "sequential cutoff" variable plays a big factor in the performance of fine-grained parallelism.