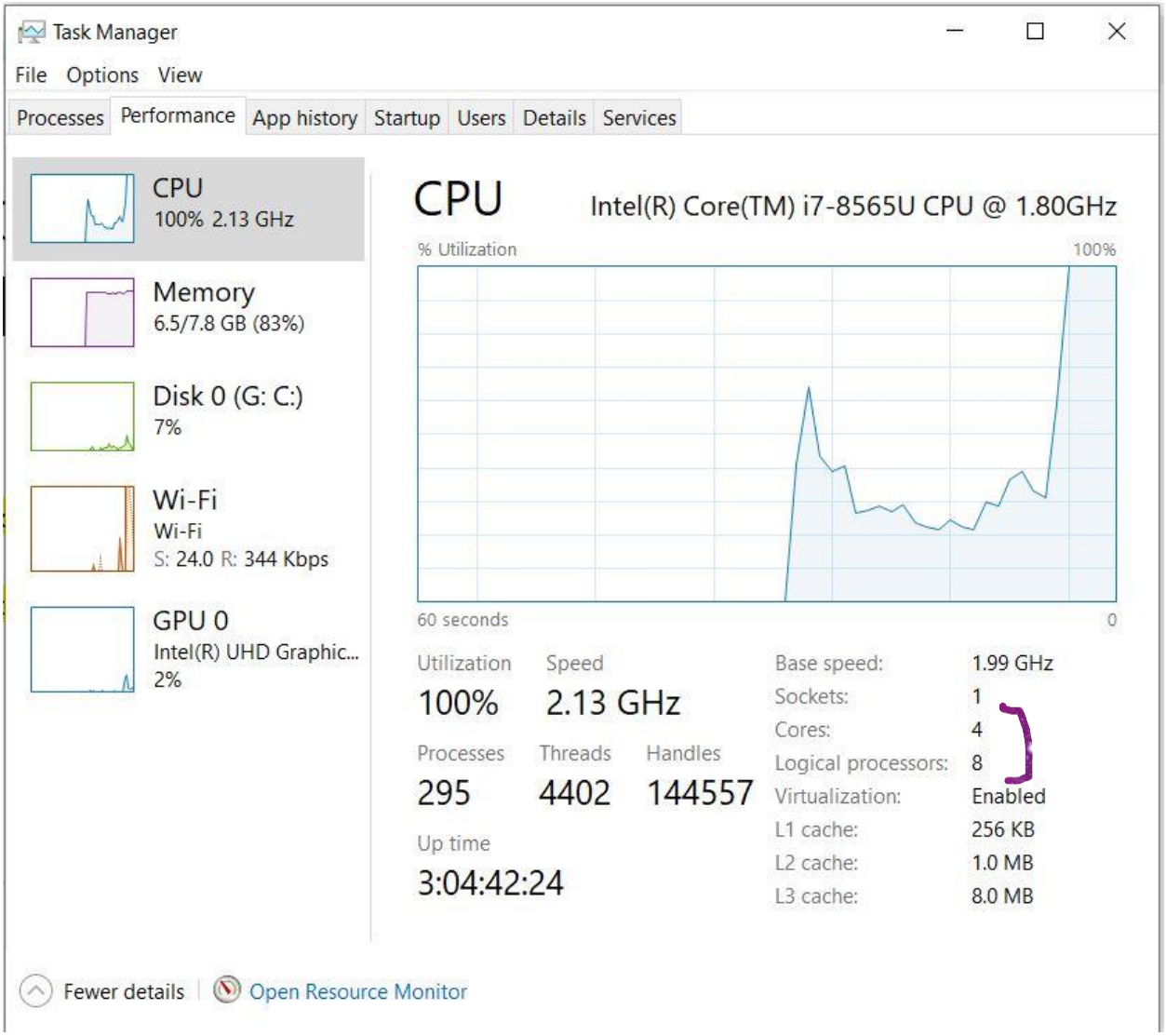
Jason Caplan

Intro to Algorithms

Prof. Leff

SecondSmallestElementFJ Assignment

Screenshot of number of cores/processors:



Experimental data showing sequential version is O(n)

Input size	Running time in ms (avg of 3 runs)	Ratio to previous
2^22	5	n/a
2^23	13	2.6
2^24	18	1.38
2^25	31	1.72
2^26	64	2.06
2^27	97	1.52
2^28	203	2.09

(obtained by running SequentialSecondSmallestElement.main())

I was only able to get more or less consistent ratios for very large input size, and was limited to 2^28 before running into heap space errors. Nonetheless, we can see that as we double the input, the time approximately doubles (average ratio is 1.895), so it is indeed O(n).

Sequential vs. FJ Table and Analysis

The table is on the following page (all time is in ms).

Again, for small inputs I did not get meaningful results, (in small inputs, FJ would often take non-zero ms, and Sequential would read as 0, so ratios could not be calculated, but we do see here that the overhead of FJ was not worth it in these cases). So I only include here experiments for 3 input sizes, and within each one I include the running time with different values for threshold.

A consisten pattern emerges: For larger values of threshold, the FJ does not perform much better than the Sequential (in fact, in some smaller inputs I constantly saw almost exactly 1 for speedup in these large threshold cases, which makes sense). As threshold gets closer to 0.0, the FJ speedup is more pronounced. This is expected, as the more granular we go, the better we split up the work amongst the cores and the better our overall performance. However, when we got too granular, effectively needing a call to the sequential algoruithm for each and every element in the array as its own subarray, our performance plummeted, since running the sequential algorithm on that level doesn't actually gain anything, and all the work is performed in the merge back up the recursive tree of the smallest two found thusfar in the recursive sub-tree.

Additionally, the highest speedups appeared as input size grew as large as possible. This also makes sense, as the effects of parallelization should be felt stronger the larger the input, just like the difference between preparing a wedding (large input) with a team vs. alone is much larger than the difference between planning a picnic (small input) with a team vs. alone.

Regarding the optimal performance boost we achieved: using 8 processors, we might have expected an 800% speedup, but the most I saw during testing at any point was in the high 400's. I can think of 2 explanations for this: FJ doesn't really

speedup linearly with respect to the nnumber of processors, but rather only to half the number of processors. Alternatively, it's because my machine has only `4` "real" cores, and the number `8` represents "logical processors," which may not perform as well as `8` real cores.

Input Size 2^22, Sequential Time = 11		
Threshold	FJ Time	Speedup (Seq/FJ)
0.9	6	1.83
0.5	5	2.2
0.25	4	2.75
0.1	5	2.2
0.01	5	2.2
0.001	3	3.66
0.0	111	0.1
Input Size 2^26, Sequential Time = 60		
Threshold	FJ Time	Speedup (Seq/FJ)
0.9	54	1.11
0.5	58	1.03
0.25	33	1.82
0.1	27	2.22
0.01	20	3.0
0.001	18	3.33
0.0	1351	0.04
Input Size 2^28, Sequential Time = 281		
Threshold	FJ Time	Speedup (Seq/FJ)
0.9	185	1.52
0.5	199	1.41
0.25	101	2.78
0.1	83	3.39
0.01	76	3.70
0.001	69	4.07
0.0	6017	0.04

(obtained by running SecondSmallestElementFJ.main())