IN3030 mandatory 1

Erlend Kristensen

07.02.2022

K largest elements

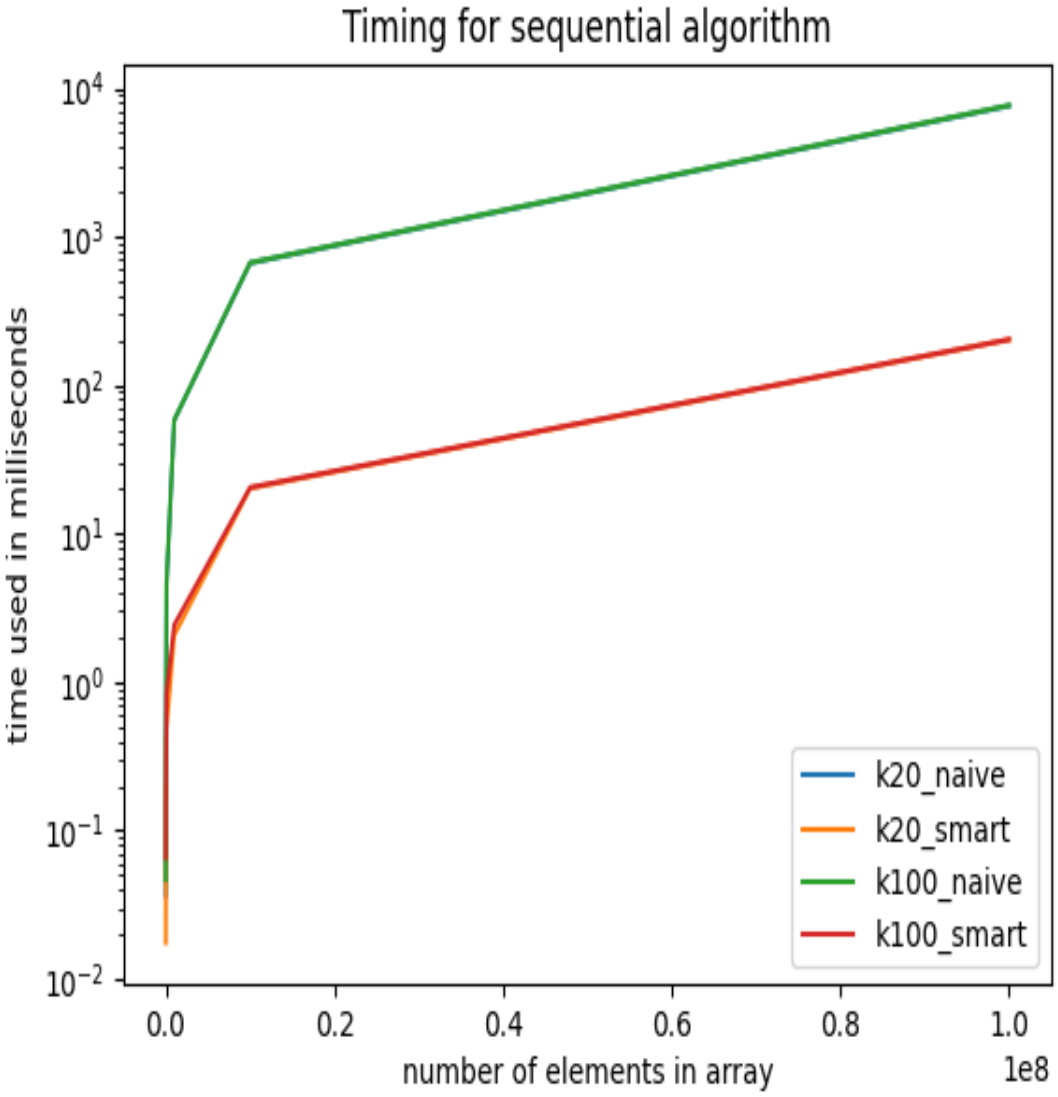
The code for the sequential algorithm can be found in task1.java, the code for the parallell can be found in task2.java, and the code i used for plotting can be found in plot.py.

We first start of by implementing a smart sequential algorithm for finding the k-largest elements in an array, and test it against a naive implementation where we sort the entire array first.

The idea behind the smarter algorithm is that we start of by picking out the first k elements in our array of length n, and sort them. Then we go through each remaining element in our array, and for each element we do a check to see if it is bigger than the smallest element in our k-array (which is at the back of the array, since it is sorted descending). If the new element is smaller, we carry on to the next element, and if it is bigger, we assign the value of the element to the last element of the k-array, and then do an insertion sort on the k-array.

The time complexity for the naive implementation where we just sort first is $O(n * \log n)$, and the time complexity for our smarter algorithm is $O((n - k + 1) * k \log k)$ at worst, seeing as we usually don't have to swap and sort for every new element. We see that this is much smaller, and will only be as slow if $n = k$, because then we simply sort our entire array when we sort the first k elements.

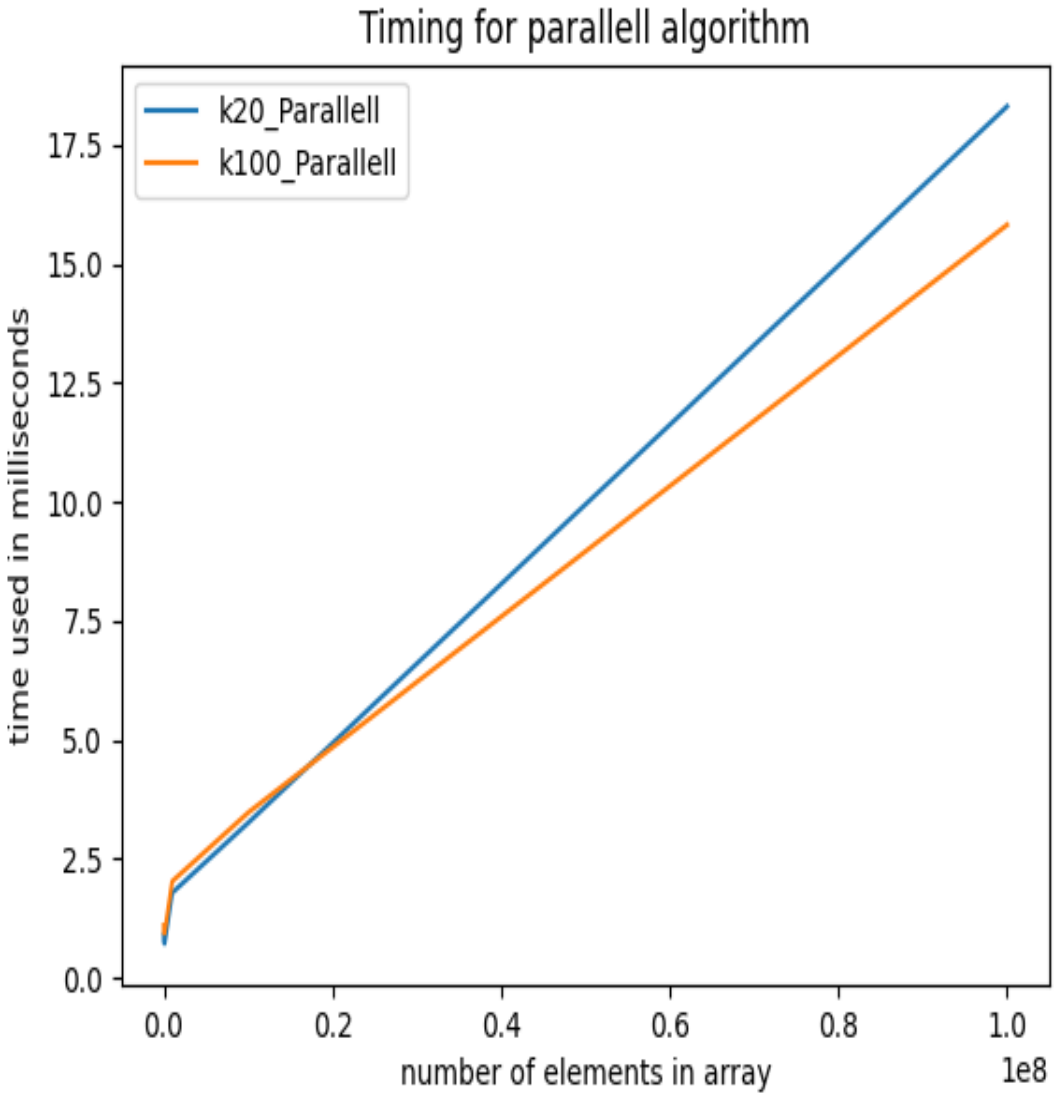
We now test these two against each other by running timed tests. Here we take the median of 7 iterations, for $n = 1000, 10000, ..., 100000000$.

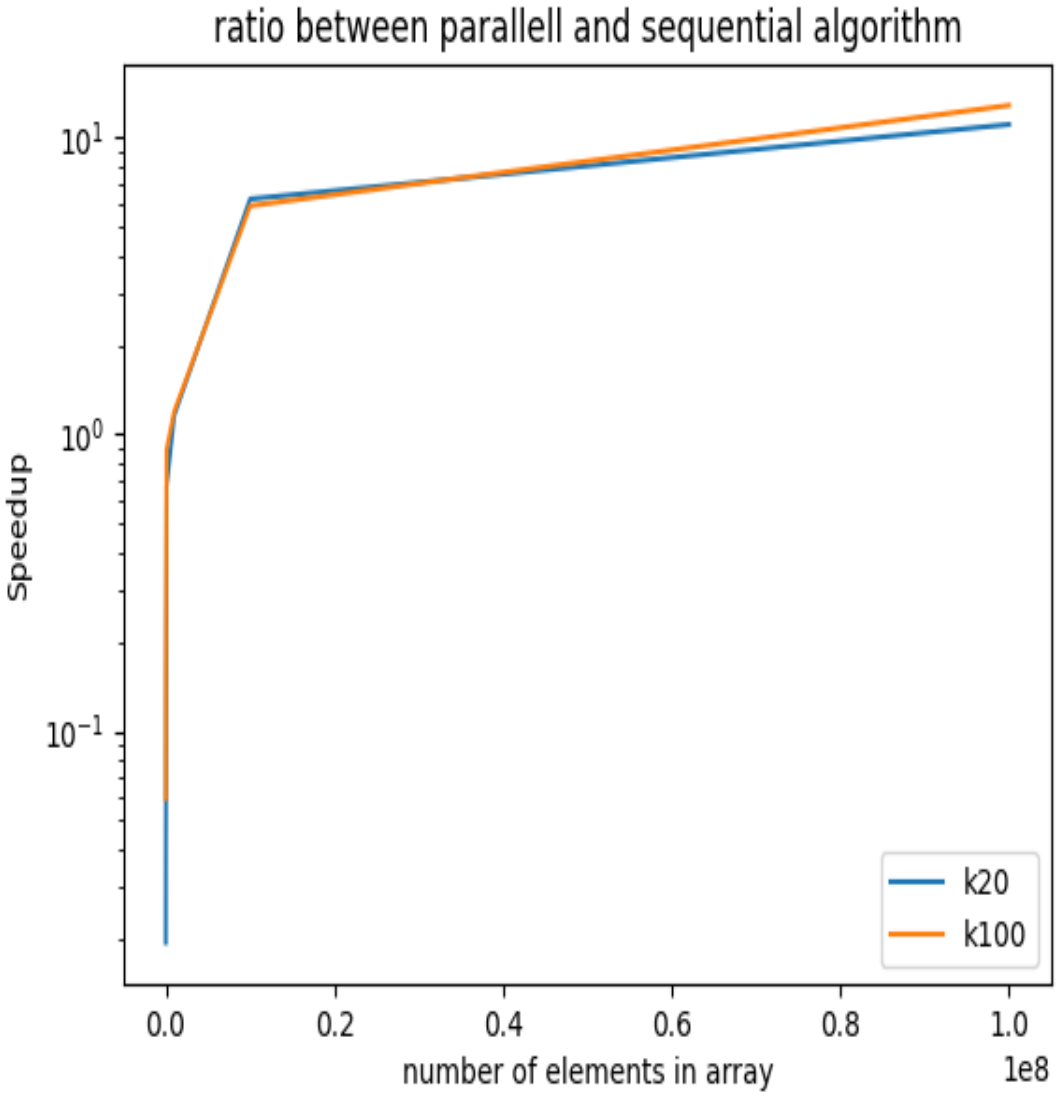


As we see in this figure, our smart implementation is a great amount faster than the naive implementation, being about 1000 times faster for the bigger amounts of n. We also see that they both follow a similar growth on the time curve, which was expected from our calculations on the $O(n)$

notation.

We now do a time test of the parallel algorithm, and compare the time to our sequential algorithm.





For the parallel code I used 6 cores. Here we see the speedup when using the parallel implementation being about $8-10$ times. Usually it should not be over 6, but since we initialize the arrays at random and only go through 7 iterations, the results can vary greatly.

However, it clearly shows a great amount of speedup with our parallel code, meaning that it is implemented well.

The code is run on a AMD Ryzen 5 5500U cpu with 6 cores and a base speed of 2.1 GHz.

time in ms for k=20			
Size of array (n)	Naive	Smart	Parallel
1000	0.0365	0.0176	0.9082
10 000	0.4128	0.0301	0.756
100 000	4.7158	0.4841	0.724
1 000 000	58.0123	2.0736	1.7908
10 000 000	665.1101	20.2342	3.2584
100 000 000	7625.4721	202.8154	18.3112

time in ms for k=100			
Size of array (n)	Naive	Smart	Parallel
1000	0.0468	0.0657	1.1088
10 000	0.4021	0.1483	0.9432
100 000	4.7034	0.84	0.9435
1 000 000	58.3454	2.4266	2.0316
10 000 000	668.425	20.4594	3.4735
100 000 000	7676.6177	203.2082	15.8223

As we see for both k values, it seems that the parallel algorithm gets faster at about $n = 100000$ to $n = 1000000$. We also observe that the difference in speed between $k = 20$ and $k = 100$ is only noticeable up to $n = 10$ million, where the speed seems to be about the same.