IN3030 – Oblig1.

Marcusti@student.matnat.uio.no

Benchmarking was done on my laptop which consists of a:
i7-7500U CPU @ 2.70GHz - Dual-core / 2 threads per core.
For a total of 4 Threads.

Program

I've decided to make one class (Sorter) which includes all the algorithmic methods. Thereafter I've made two main programs, Opg1 and Opg2.

Parallel Solution

While the sequential and the Arrays.sort (called Java sort from now on) solutions are quite straight forward, especially as the sequential solution is described in the oblig, the parallel solution needs some explaining.

So how does my parallel solution work? I've decided to split the main array consisting the N-elements, into T parts. Where T is the number of Threads the current computer has available. For each part T of the main array, I do the exact same sequential calculations. I assume the K-first elements in each array T are the largest, sort them descending, and then go on from element T[k+1]..T[length] to see if any other elements in T are larger than T[k]. If any larger value is found, swap elements and sort the top K- elements again.

After each thread has finished sorting their part of the array, I need to find which of the T*K items are largest. I do this by assuming T_0 has the largest elements, and then compare all elements in $T_1...T_{n\_threads}$. If I find any element larger than the smallest one found in T_0, I commit a swap followed by re-sorting the top K- elements in T_0. After all comparisons are done, we are left with the top K- elements of the entire array placed in the top K indices. All elements of the original array are still intact, with no overwrites or deletions.

Results

Below are the results I've gathered running the programs from my laptop.

K = 20

Algorithm	**N = 1 000 (ms)**	**N = 10 000 (ms)**	**N = 100 000 (ms)**	**N = 1 000 000 (ms)**	**N = 10 000 000 (ms)**	**N = 100 000 000 (ms)**
Java Sort	0.422	1.186	6.024	74.66	839.2	9636
Sequential	0.063	0.274	0.421	0.680	4.050	39.73
Parallel	0.303	0.467	0.710	0.700	2.373	20.65
Speedup S	6.69	4.32	14.30	109.79	207.20	242.53
Speedup P	1.39	2.53	8.48	106.65	353.64	466.43
$Speedup\,\frac{S}{p}$	0.20	0.58	0.59	0.97	1.70	1.92

K = 100

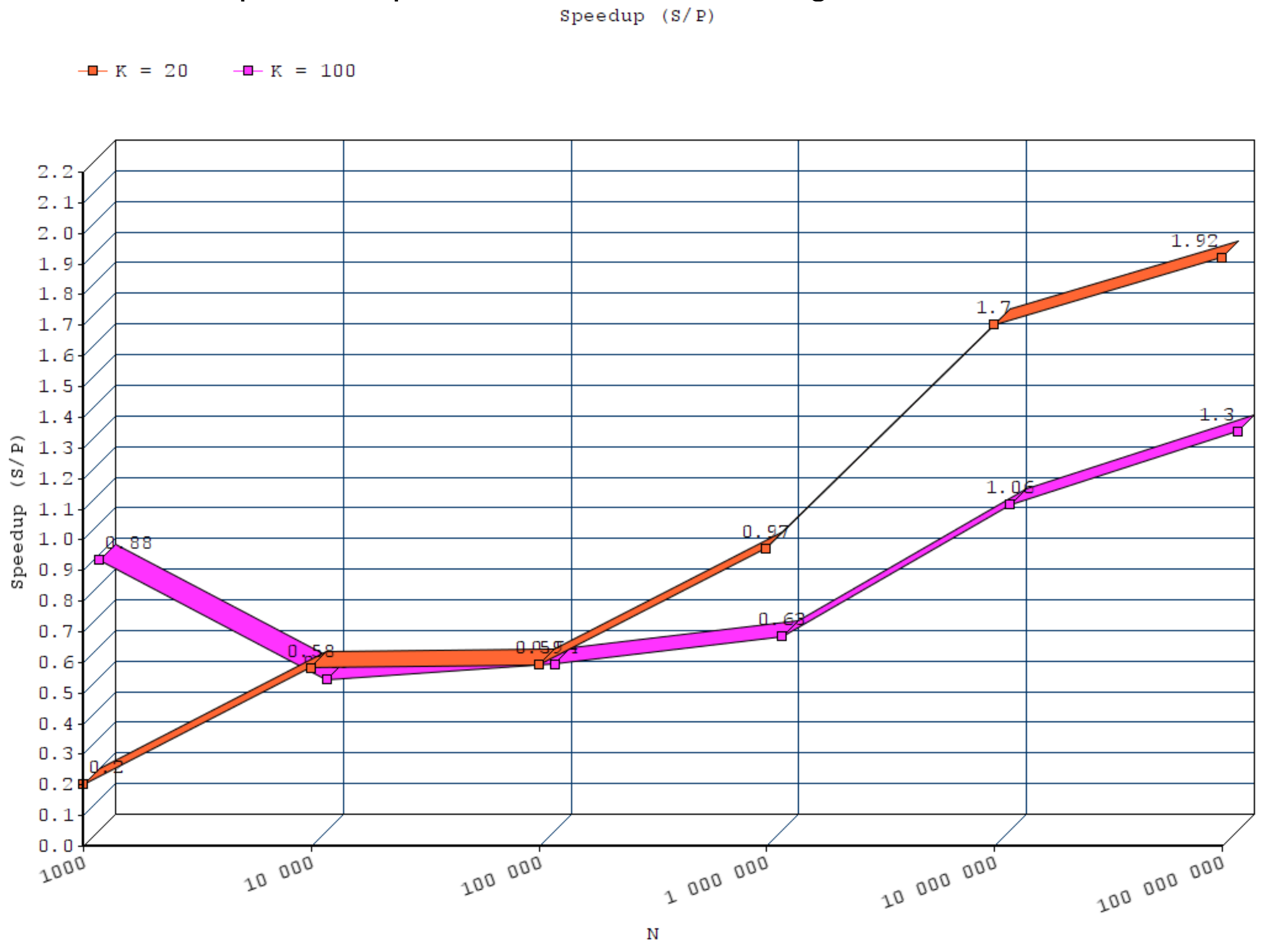
Algorithm	**N = 1 000 (ms)**	**N = 10 000 (ms)**	**N = 100 000 (ms)**	**N = 1 000 000 (ms)**	**N = 10 000 000 (ms)**	**N = 100 000 000 (ms)**
Java Sort	0.363	1.017	6.973	72.33	829.8	9629
Sequential	0.953	0.658	0.962	1.214	4.101	39.98
Parallel	1.077	1.330	1.752	1.926	3.835	30.73
Speedup S	0.38	1.54	7.24	59.57	202.34	240.84
Speedup P	0.33	0.76	3.98	37.55	216.37	313.34
$Speedup\,\frac{S}{p}$	0.88	0.49	0.54	0.63	1.06	1.30

Conclusion

As we can see the actual timings and speedups vary a bit, but we can still see some logic in how the different algorithms performed. For really small values of N, of course, the Java sort will be the quickest way of sorting the array. But somewhere in between 1000-10 000, the custom algorithm catches up and is quite a bit faster than the Java sort. This makes sense as the Java sort algorithm has to sort every single element, while our custom algorithm only has to sort the top K- elements, for then to only compare all other elements.

As for the sequential vs. parallel solutions, we notice how the parallel solution seems to always outperform the sequential one, given a large N number of elements. This also makes sense, as the sequential algorithm only has to sort the top K elements, while the parallel algorithm has to sort K * n_threads. Also the cost in time of creating Threads, as well as synchronization just doesn’t pay off for small values of N.

However, when N approaches really big values, we see how the parallel algorithm will outperform the sequential one in the long run. This can be explained by how the sequential algorithm only has one thread working on comparing all elements, while of course the parallel solution has in my case, 4 threads working on the same number of elements in parallel. The punishment of creating new threads and synchronization is here neglected as the amount of pure computation is the time-limiting factor.



This graph shows the performance of the parallel vs. sequential algorithms. For K = 20 we see that the sequential solution is the better one until somewhere close to N = 1 million. For K = 100, the parallel solution doesn't pay off before N = 10 million. At least not on my laptop.