Mold Florian, Karakaya Aytac

Parallel Programming Sorting

Students: Mold Florian, Karakaya Aytac

Approach

We generated between 5.000, 50.000 and 500.00 random numbers for each test. We ran the sorting algorithms 100 times and computed the average runtime in seconds.

We implemented QuickSort and MergeSort in three different variations

- Sequential
- Naive parallel version
- Parallel version with threshold

For the parallelization we used the **pragma omp task** from the OpenMP library. The tests were run with a machine with 8 logical CPU cores.

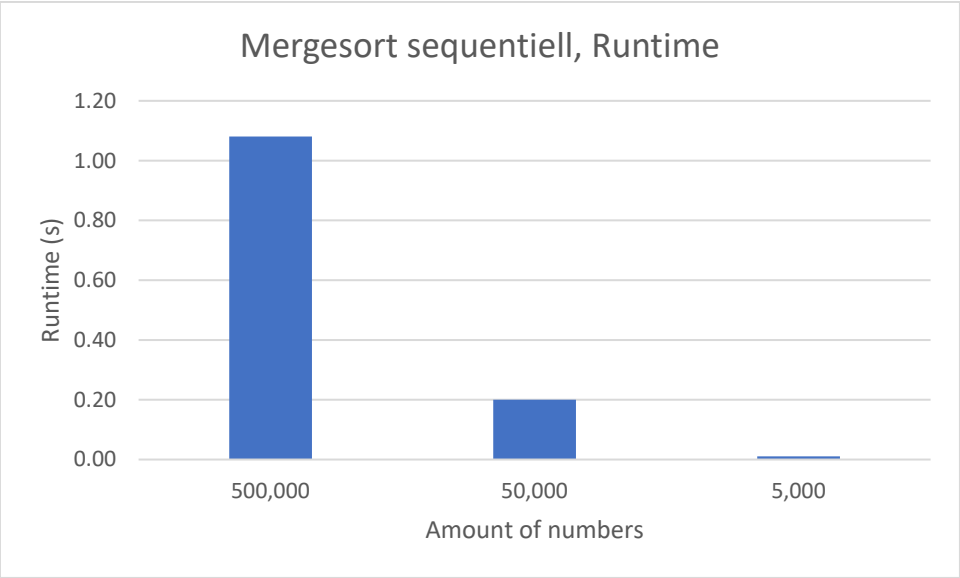
Results

MergeSort

This section describes the sequential, naïve parallel and parallel with threshold implementation of the mergesort algorithm.

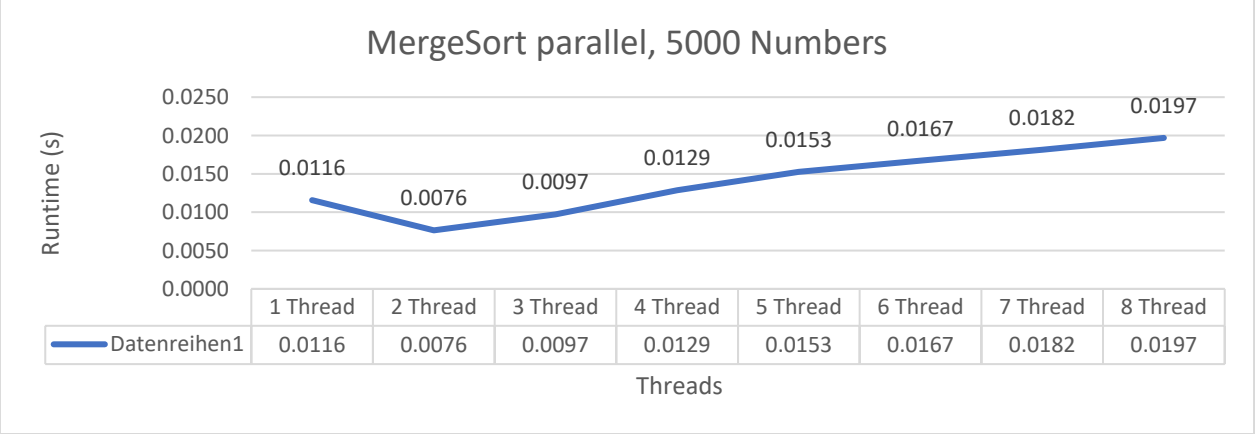
Sequential

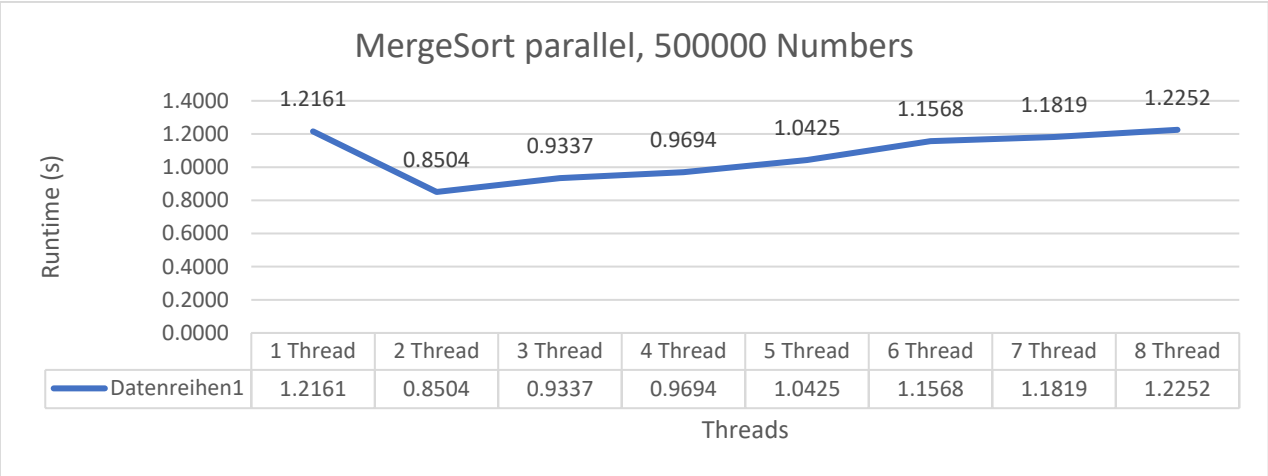
We sorted 500.000, 50.000 and 5.000 numbers with the sequential MergeSort. The time to sort the 5000 numbers was 0.01s, the 50.000 numbers was 0.20s and the 500.000 was 1.08s. Overall, the performance of the sequential MergeSort was very good.



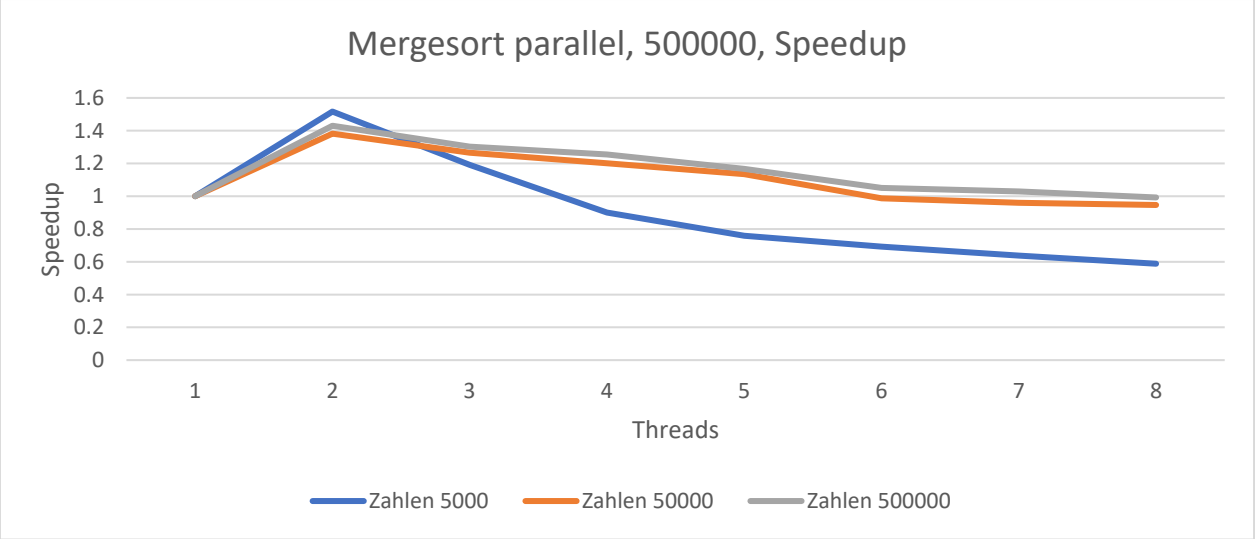
Parallel

The following graphs show the runtime of the naive implemented parallel MergeSort. As we see multithreading does not have a positive effect on the overall runtime. The execution is split in half with 2 Threads, but with more threads the execution time is rising to an even higher value than the execution time of the sequential algorithm. This is a good example of oversubscription, which leads to higher parallel runtime.



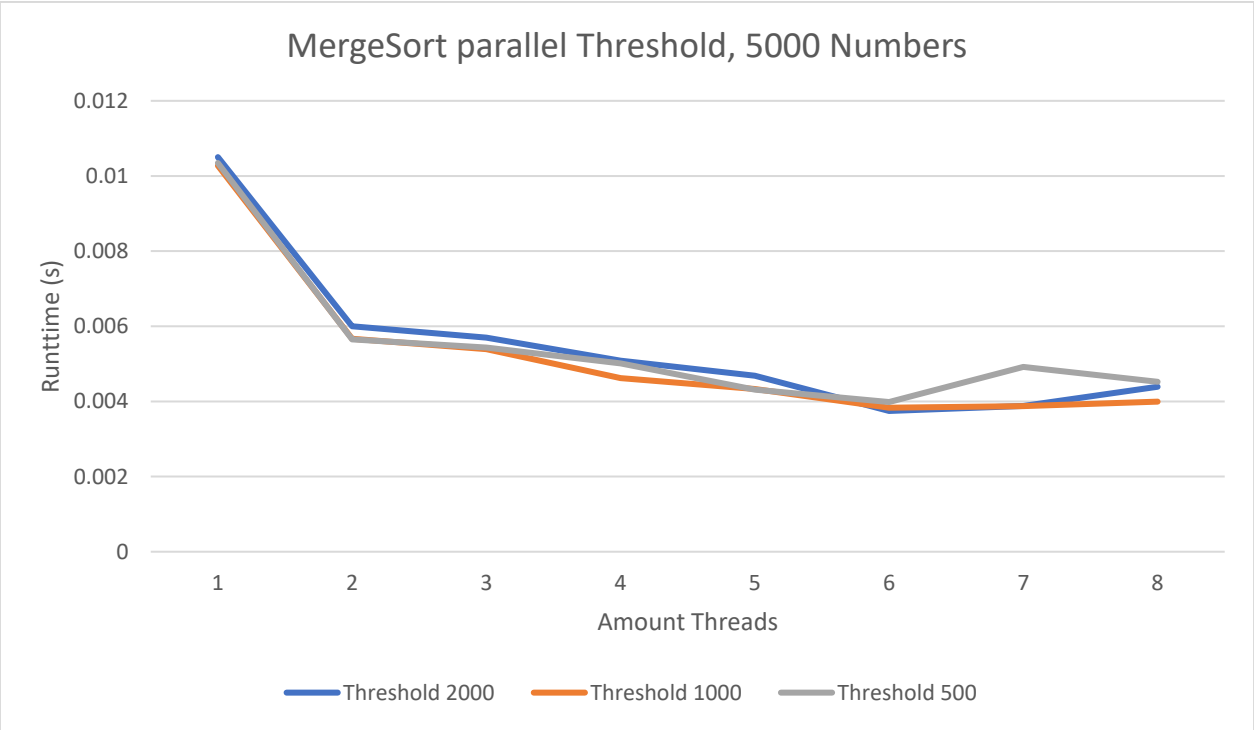


This graph shows the speedup of the parallel MergeSort with 500.000 numbers. Just like visualized in the graph above, the maximum speedup can be achieved with 2 threads. Afterwards the speedup constantly sinks. The least speedup is achieved for the smallest dataset and the most threads. This is an example for oversubscription.

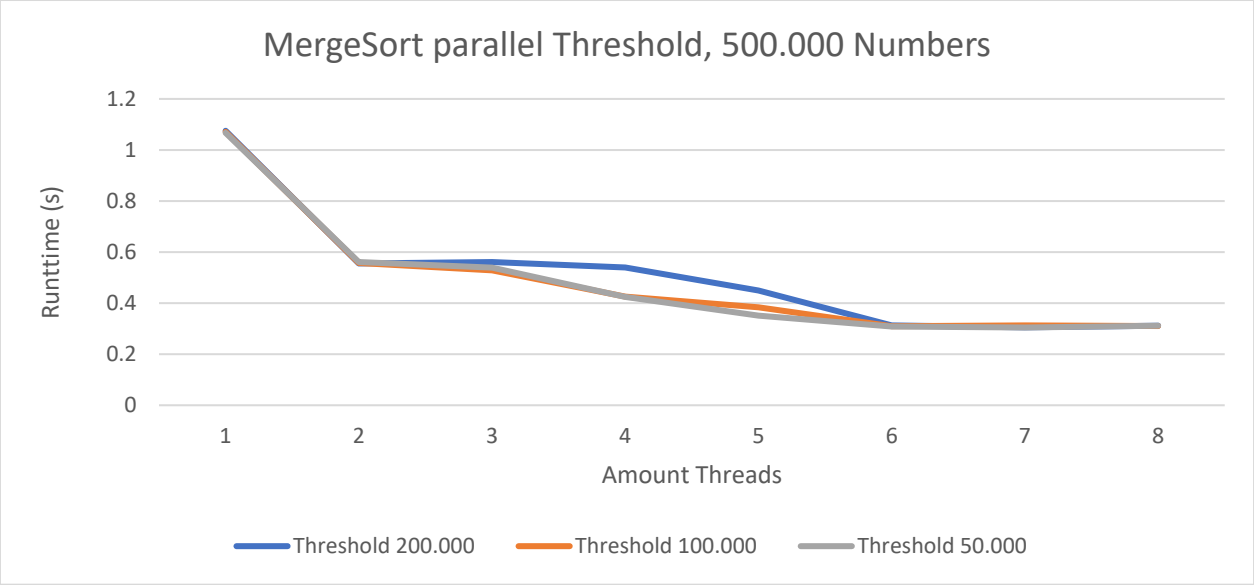


Parallel with threshold

This graph shows the runtime, when sorting 5000 numbers with the parallel MergeSort with a threshold. The program was executed with a threshold of 500, 1000 and 2000. The runtimes with the different thresholds are almost the same, so the different thresholds do little to nothing compared to each other.

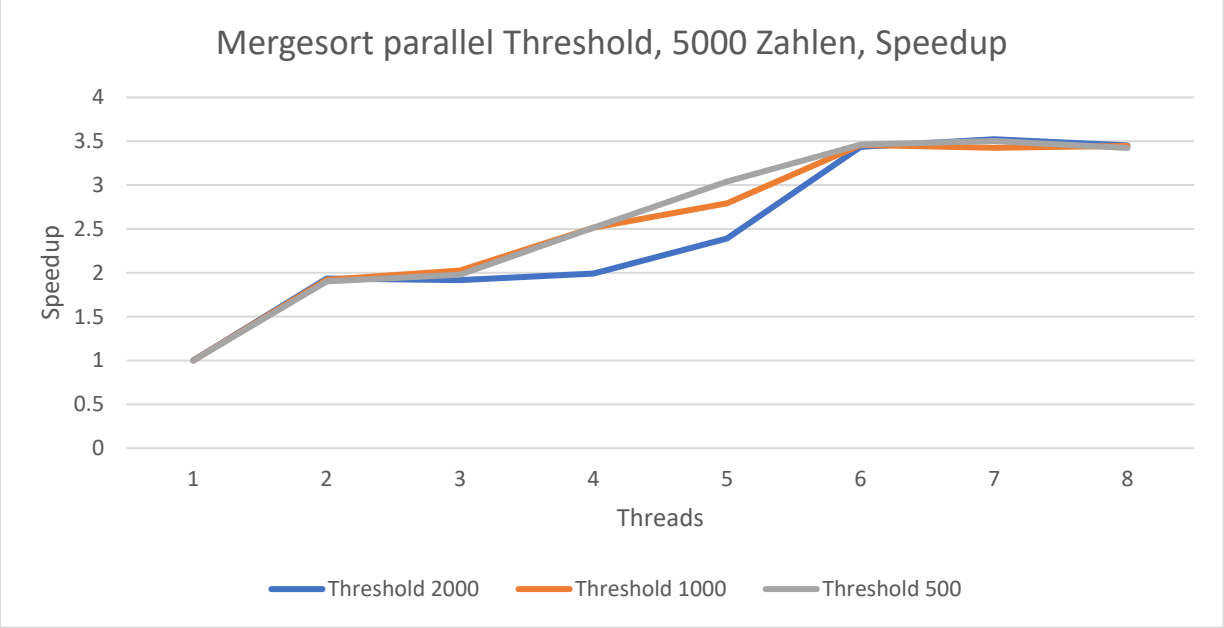


This graph shows the runtime, when sorting 500.000 numbers with the parallel MergeSort with a threshold. The program was executed with a threshold of 50.000, 100.000 and 200.000. The runtimes with the different thresholds are almost the same, so the different thresholds do little to nothing compared to each other.



The next graph shows the speedup of the parallel MergeSort with the threshold 20000, when sorting 500.000 numbers. The speedup stands for all thresholds that we chose, because it achieves almost the same values. The speedup rises with the amount of threads that were added to the computation and

stagnates with 8 threads. The speedup we achieved with 8 Threads was about 3,45 and it was 3,52 with 7 threads. The speedup is very good compared with the naïve implementation of the parallel MergeSort.



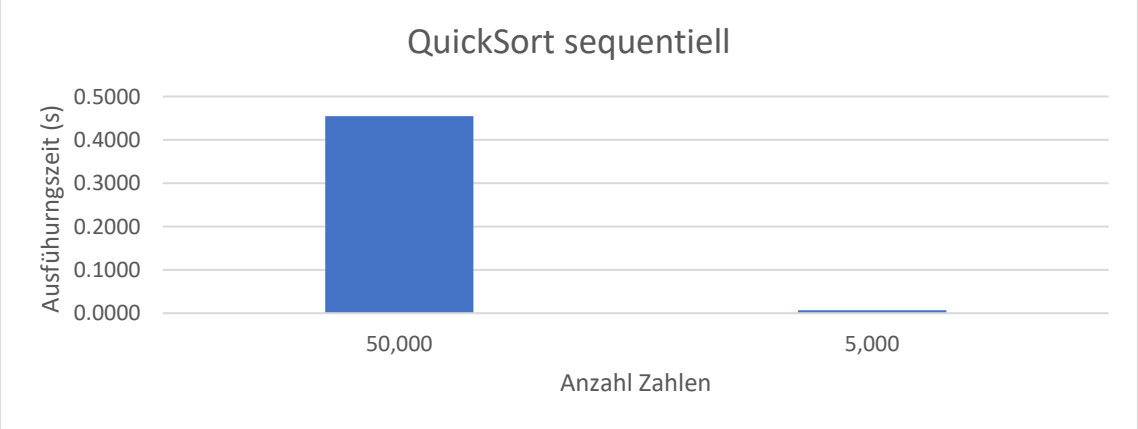
In conclusion we can observe that our chosen thresholds do not make a great difference for the MergeSort in the small and the large dataset. The thresholds almost performed the same.

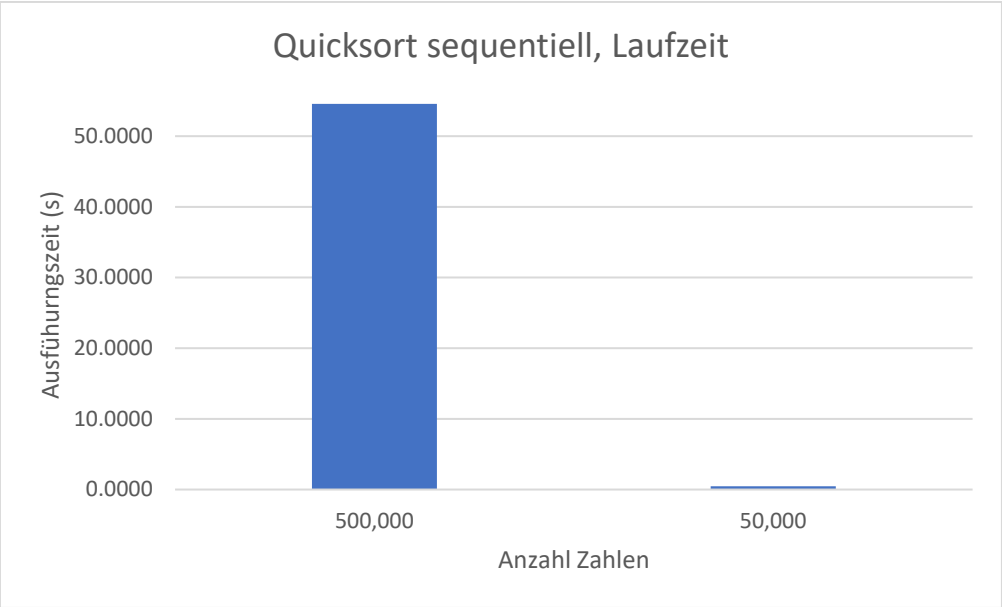
QuickSort

This section describes the sequential, parallel and parallel with threshold implementation of the QuickSort algorithm.

Sequential

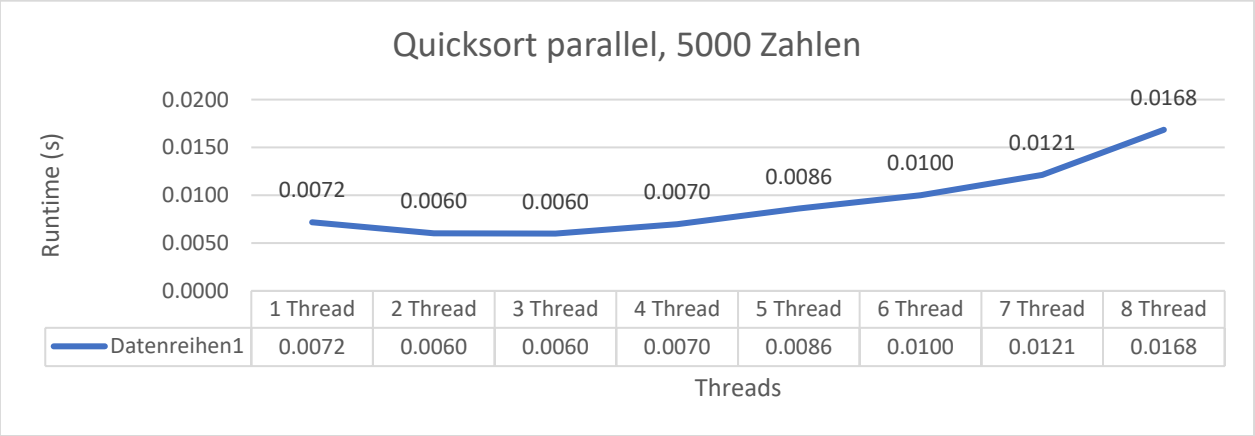
We sorted 500.000, 50.000 and 5.000 numbers with the sequential QuickSort. The time to sort the 5000 numbers was 0.0068s, the 50.000 numbers was 0,4546s and the 500.000 was 54,5657s. Overall, the performance of the sequential QuickSort was good was smaller amount of numbers. The execution time for the large amount of numbers was very long in comparison.



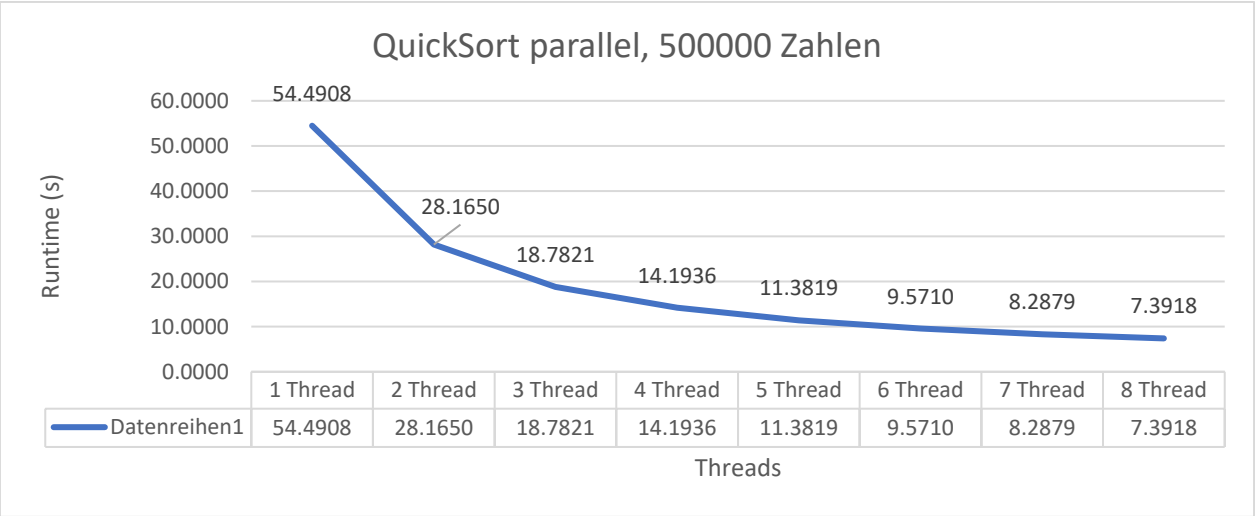


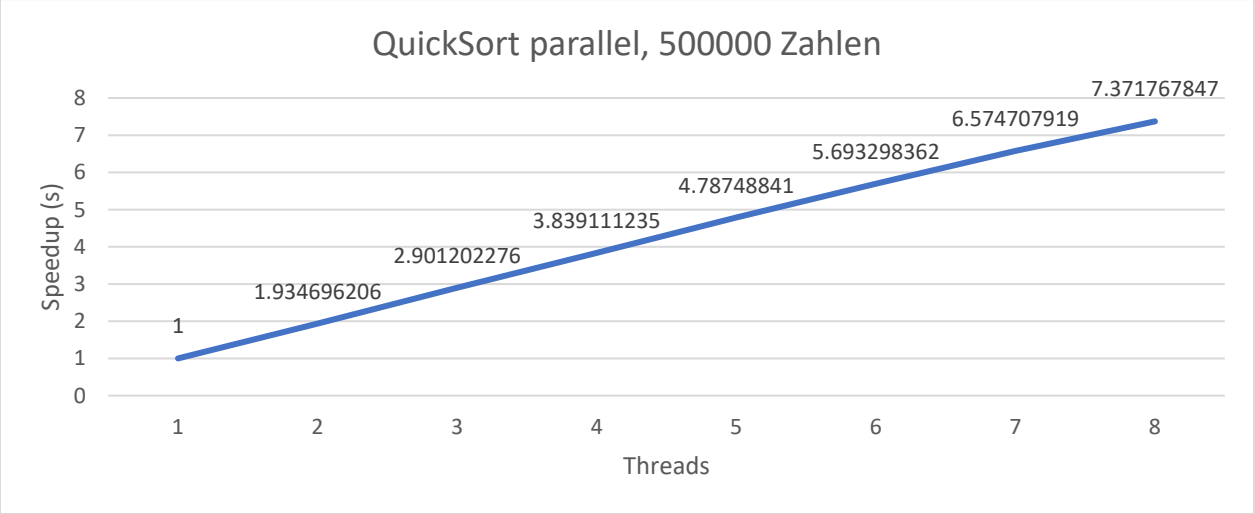
Parallel

The following graph shows the runtime of the naive implemented parallel QuickSort. For the 5000 numbers the runtime increases rapidly with more threads being used. This is a good example of oversubscription, which leads to higher parallel runtime.



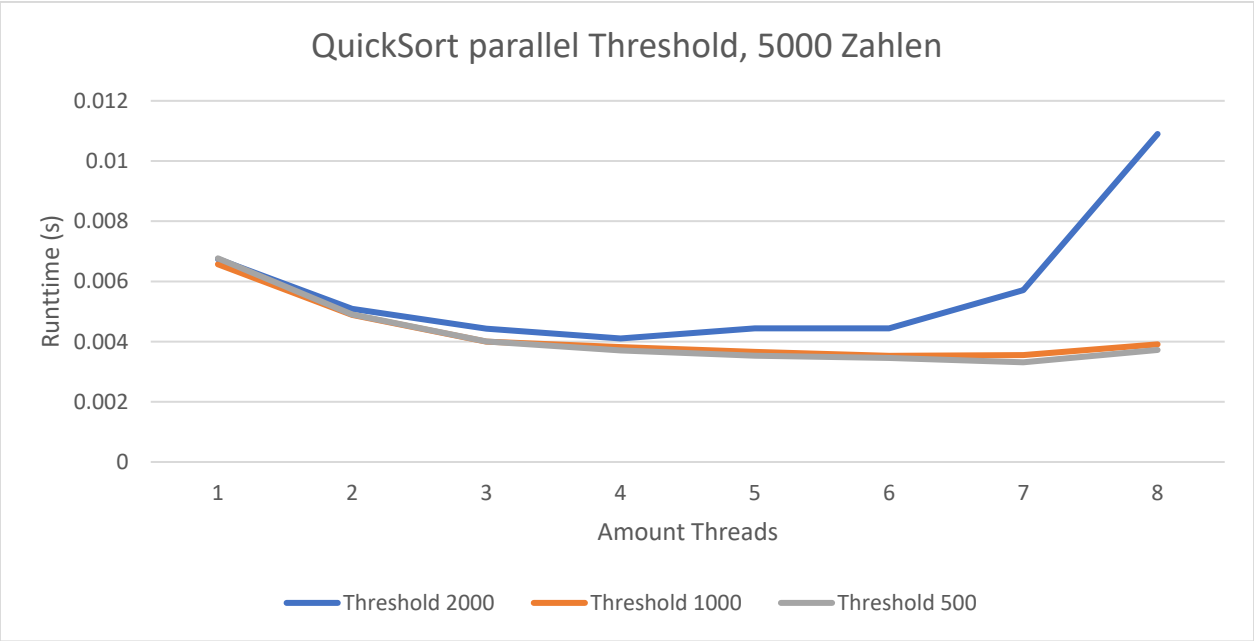
For 500.000 numbers the runtime becomes better with more threads. This is quite surprising, because we thought that it would be slower with more threads. Just like in the example with 5000 numbers. I can't explain the sudden speedup with that data. Maybe the Oversubscription does not happen with that large amount of data, because the threads share the tasks equally. For 8 threads there is a speedup of 7.3, which is the best speedup we got with all quicksort variants.



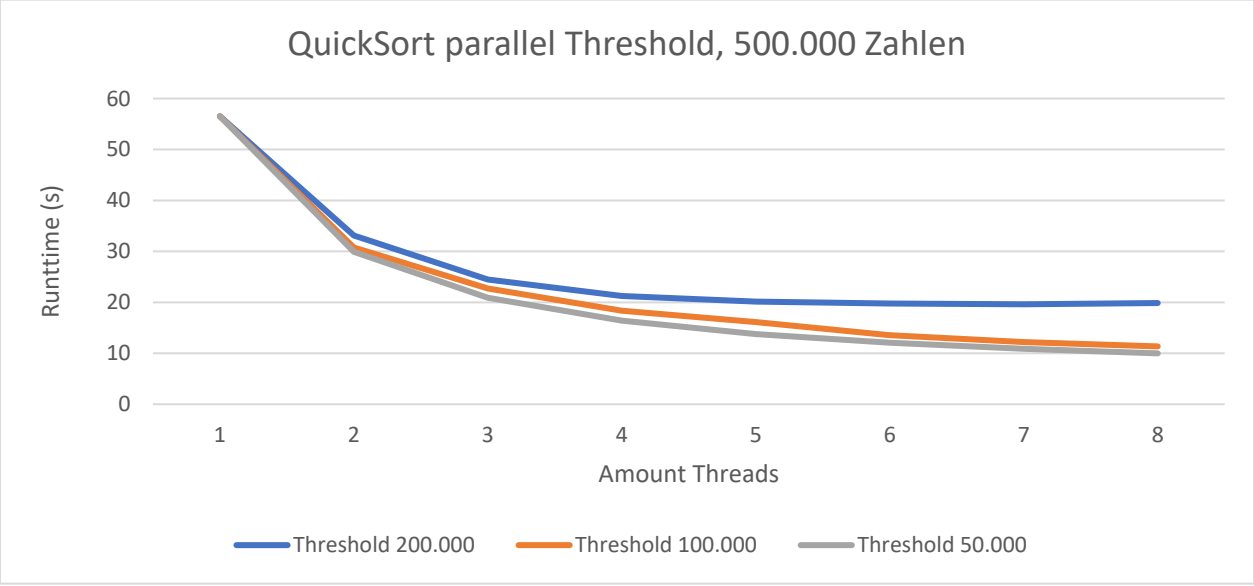


Parallel with threshold

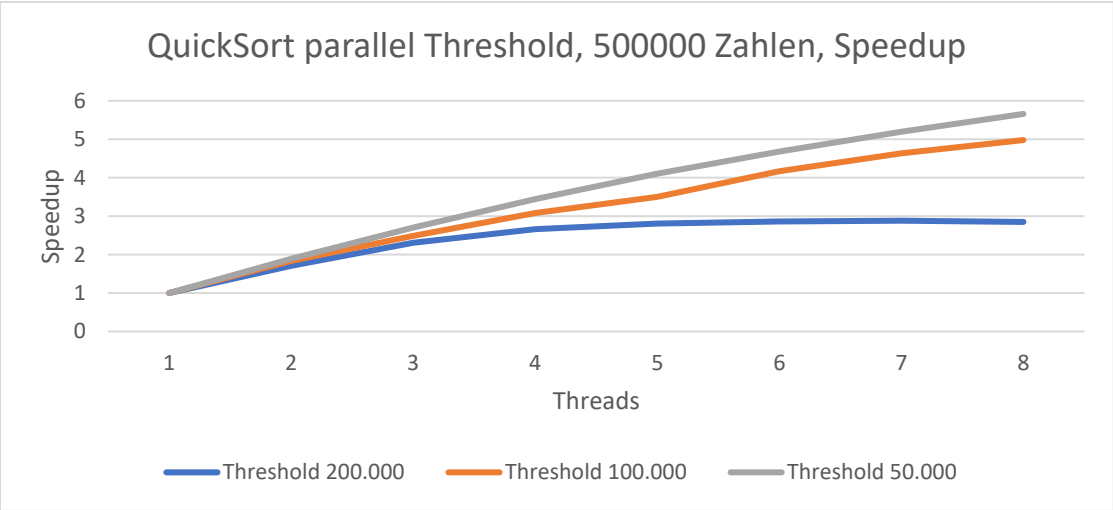
This graph shows the runtime, when sorting 5000 numbers with the parallel QuickSort with a threshold. The program was executed with a threshold of 500, 1000 and 2000. The runtimes with the thresholds 500 and 1000 are almost the same. But in the runtime of the algorithm with the threshold 2000 the runtime increases, with more threads. We think that the threshold is too big here, the runtime is very much like the sequential version plus the cost of creating tasks. It looks a lot like the naive parallel runtime of the QuickSort algorithm, because the runtime increases very drastically starting with 6 threads.



This graph shows the runtime, when sorting 500.000 numbers with the parallel QuickSort with a threshold. The program was executed with a threshold of 50.000, 100.000 and 200.000. The runtime of the thresholds is almost the same for 100.000 and 50.000. Threshold 200.000 becomes slower with more threads being added. It doesn't rise like the example with the 5000 numbers, but it slower than the other thresholds.

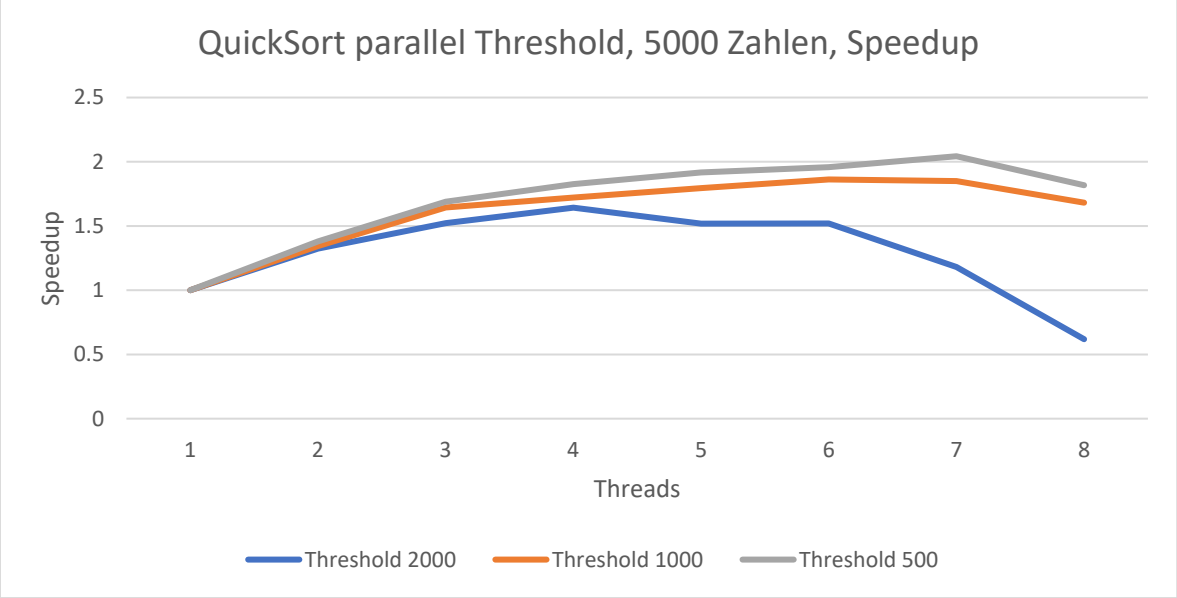


The next graph shows the speedup of the parallel QuickSort with the threshold 200.000, 100.000 and 50.000, when sorting 500.000 numbers. The speedup rises with the amount of threads that were added to the computation for a threshold of 100.000 and 50.000. The best speedup is achieved with a threshold of 50.000. The threshold of 200.000 does not get faster with more threads. It begins to stagnate with five threads and does not speedup with more threads. The speedup is not so good compared with the naïve parallel QuickSort. Our maximum speedup with a threshold of 50.000 is about 5.6 which is worse than the naive parallel. A very weird result.



The next graph shows the speedup of the parallel QuickSort with the threshold 2000, 1000 and 500, when sorting 5000 numbers. At the beginning the speedup rises with more threads and for a threshold of 500 and 1000 it stagnates for a bit and gets lower with 8 threads.

As observed in the runtime, the threshold of 2000 is very bad for the speedup. It rises until 4 threads and then gets constantly worse. For 8 threads it even worse than the single threaded solution. As said before the threshold 2000 is too big here.



In conclusion we can observe that high thresholds for QuickSort are slower than lower ones. The two smaller thresholds performed almost the same, whereas the larger threshold is considerably slower for the large and small dataset.

Mold Florian, Karakaya Aytac

Analysis

All the results and graphs can be found inside the **analysis.xlsx**. In the spreadsheet even more comparisons exist.