Karen Liu

Rebecca Tang

COSC-273

May 9, 2023

Parallelizing Sorting of Float Arrays

We optimized the problem of sorting an array of floats through utilizing the quick sort algorithm, through parallelizing the sorting using fork join pools, through testing different partition procedures for quicksort, and through adjusting the point at which the program should shift from parallelized sorting to sequential sorting. An attempt was also made at vectorizing the partition procedure, the most expensive part of the sorting.

Detailed table of optimizations and their runtimes:

Optimization	**Runtime (ms)**
Baseline	8194
Parallelizing with fork-join pools using quick sort; pivot is the median value	5443
Partitioning by taking the value in the highest index as the pivot (Lomuto partition)	3646
Adjusting PARALLEL_LIMIT (the maximum size for which the method doesn't fork)	1796
Partition with lowest index as pivot (Hoare partition)	1806

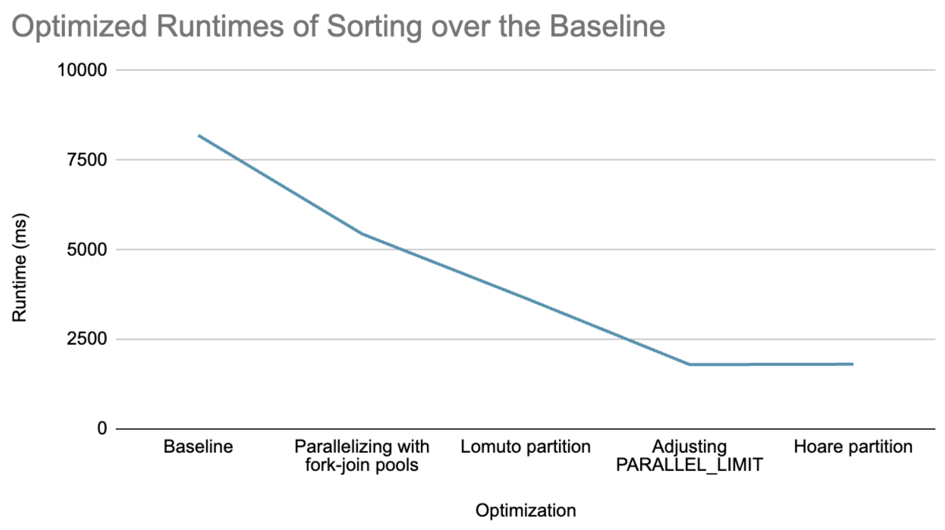
The biggest challenge to starting this project was understanding how fork-join pools work to parallelize recursive programs because the logic is not immediately comfortable, but after some experiments with forking and computing, the process became very intuitive. Specifically, we parallelize the recursive “halves” of each side of a pivot after calling quick sort on the halves.

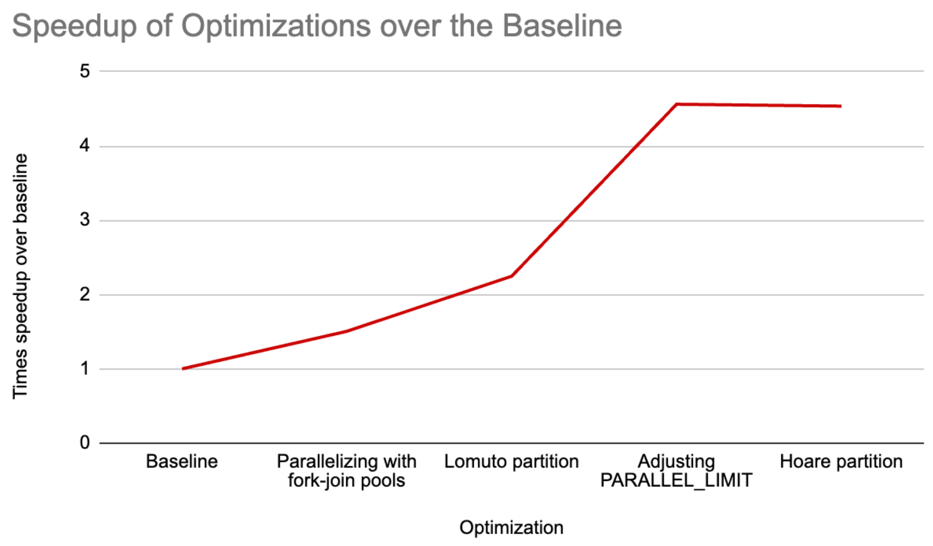
However, we started by doing a naive partition procedure where we searched for the median each time to use as the pivot, but we eventually realized that this was not a good way to pick the pivot and was using O(n) time consistently per subarray. So, we tested Lomuto partition, which picks the highest index’s value as the pivot each time, which ensures that the pivot enters its final position upon partition even though it requires more comparisons. This made a big difference. We try another partition procedure later, but first, we decided to make the most basic and effective optimization: changing PARALLEL_LIMIT.

The default value was set to 10, meaning that the program would switch to Java’s built in Arrays.sort() when the subarray size reaches 10. We experimented to find out when the arrays become just small enough where it would be faster to just use Java’s implementation of sorting, Arrays.sort(), which uses quicksort with a dual-pivot partition procedure, but not too small that we fail to achieve parallelization’s maximum benefits. We moved from 1,000 and 1,000,000 in steps of 10,000 and then 2,500, and we eventually converged on 55,000 as the optimal size limit on the cluster.

After, we tested another partition procedure, which uses Hoare’s partition, where the first index is chosen as the pivot. This procedure was saved for later because it’s logic was difficult to reason through since the pivot is not fixed from the beginning, but we wanted to try it because Hoare's uses fewer swaps than Lomuto's. However, it provided no noticeable benefit, and it even seemed to perform worse. We're unsure of why, but it seems that any changes it has made to our program compared to Lomuto's is insubstantial compared to the rest of the program.

Below are our charts depicting the optimizations tested successfully, their raw numbers and their speedup comparisons:





The final optimization that we attempted, though unsuccessfully, was vectorization of the partition procedure, which is the most expensive part of quicksort. However, we could not successfully swap vectors with each other by comparing them to each other using masks and .allTrue() because we couldn’t successfully compare the vectors that got left behind — specifically, the vectors that did not include all elements either less than or greater than the pivot. We weren’t sure how to successfully utilize sorting networks for the process, which is regretful because we feel that a successfully vectorized partition procedure would provide substantial speedup.