Alexandre Simon 6380359
Natanan Nithiittikrai 6380852

Funpar Project

Parallel Sorting Algorithms

For our Funpar project, we wanted to work on one of the most famous, if not the most famous, type of algorithm: sorting algorithms! We decided to test the limits of the sorting algorithms we know and love, as well as discover the hidden parallel potential of some lesser known sorting algorithms. The already popular algorithms we chose to work on are QuickSort, InsertionSort and SelectionSort, and the lesser known algorithms we worked on are TreeSort, RadixSort and SampleSort. For each algorithm we wrote a sequential and a parallel version, which we then timed to see how much faster (or slower) the parallel version is compared to the sequential version.

In the following paragraphs we will explain how the process went and what results we found for each algorithm.

QuickSort

Probably the most popular sorting algorithm out there, Quicksort has also a lot of parallel potential. This shows, since for an input array of 10,000,000 elements for both the sorting and sequential versions, we have a runtime difference of 14 seconds in favor of the parallel version. For the sequential the runtime is 21.24 seconds, and for the parallel it's 7.34 seconds. We can observe that the parallel runtime is 3x faster.

SelectionSort

We had trouble finding parallel potential for SelectionSort, but the algorithm uses the min() and filter() functions so we decided to write parallel versions of both min() and filter() which we called par_min() and par_filter(). Unfortunately, the parallel version we came up with is much slower than the sequential version, even with huge input arrays. The sequential version has a runtime of 1.47 seconds and the parallel version has a runtime of 5.58 seconds. In other words, the sequential version is 4x faster.

InsertionSort

InsertionSort doesn't have a lot of parallel potential either, but with a divide-and-conquer approach and an additional merge() function to merge the output arrays from the divide-and-conquer approach, we managed to parallelize the algorithm. One big hiccup about our parallel version is that using an input array with too many elements will yield a stack overflow error. However, with an input array of 10 elements, the parallel version has a runtime of 1.36 seconds and the sequential version a runtime of 7.73 seconds. Despite the hiccup, we can note a clear speed difference in favor of the parallel version.

TreeSort

TreeSort doesn't have a lot of parallel potential, but when we traverse the tree and encounter a FullNode (a node with a left and right child), the function is recursively called twice for the left and right node. Awesome, we can run both these calls in parallel. Similarly to InsertionSort, one big hiccup about the parallel version of TreeSort that we wrote is that using an input array with too many elements will yield a stack overflow error. However, with an input array of 3,000 elements, the parallel version has a runtime of 0.0099 seconds and the sequential version a runtime of 0.065 seconds. Again, despite the hiccup, we can note a clear speed difference in favor of the parallel version.

RadixSort

RadixSort was complicated to parallelize because it sorts the input array based on each digit of the numbers in the input array, and each sorting is dependent on the previous sorting of the input array. We can't parallelize the sortings, but we managed to make the processes inside each sorting (i.e. counting and placing) parallel. In the end, for an input array of one million elements sorted in descending order, the sequential version has a running time of 3.32 seconds while the parallel version has a running time of 3.25 seconds.

SampleSort

SampleSort is described as a more general version of QuickSort with a lot of parallel potential. What sample sort does is similar to a divide-and-conquer approach of QuickSort, where instead of simply calling quicksort on the input array, it will first divide the input array into buckets and call quicksort on each of these buckets and concatenate the results. Indeed, after writing the sequential version of the algorithm, it was a simple parallelization procedure, since all we had to do was sort each bucket in parallel. The results are 21.55 seconds for the sequential version and 15.1 seconds for the parallel version.