PAP Report OpenMP

Morand Lucas, Renel Elliot

March 2022

1 Bubble Sort

We've implemented both the sequential and parallel version of bubble sort without much trouble. We just had to be careful to have the parallel parts take care of different chunks of the array, and take care of the in-between element after. We've noticed during our testing that the supposed worst case scenario (when the given array is sorted in descending order) is actually faster than the average case (when the given array is randomized) as seen in the Graph 2.

Furthermore, for smaller sizes of data, the sequential version of bubble sort is faster, this is due to the creation of threads taking more time being created than they gain by running the algorithm in parallel (cf Graph 1).

When varying the number of threads, we can see that adding more threads increases speedup, but too much threads worsen this speedup value (cf Graph 3).

2 Merge Sort

We ran into a bit of difficulties with merge sort. We tried to implement an iterative version of merge sort using tasks but didn't manage to make it sort correctly. Therefore we stuck with a recursive implementation of merge sort using tasks. Even on big data inputs (more than 2^{20}), our parallel version is slower than the sequential version (cf Graph 4). This is due in part by the algorithm itself which is already fast, and the creation of a large number of tasks due to recursion also slows down the execution. A way to make this better would be to not create task for small sections of the array (we already implemented this with a threshold of size 64), and also to make the algorithm iterative instead of recursive. However when we implement it the gain is almost non-existent, just a few milliseconds at best, the biggest change was the first one mentioned (adding a threshold).

Similarly to bubble sort, more threads meant more speedup, up to a certain point, even though the difference is barely noticeable (cf Graph 5).

3 Odd-Even Sort

Odd-Even sort wasn't much harder than bubble sort, in fact it was easier to implement the parallel version as this one didn't have the problem bubble sort had, which is to not sort between the chunks. Here, we can have two simple parallel for loops, one on evens then one on odds, without anything more to add. The comparison of the algorithms (sequential and parallel) are similar to the bubble sort algorithms, that is, the parallel version preforms better after a certain size of input (cf Graph 6).

Where this sorting algorithm really shines is comparing it to bubble sort itself, where the odd-even has a clear advantage (cf Graph 7).

In terms of the variation of threads, we can see that the less thread, the better the performance is for higher problem size (cf Graph 8).

4 Performance Graphs

4.1 Bubble Sort

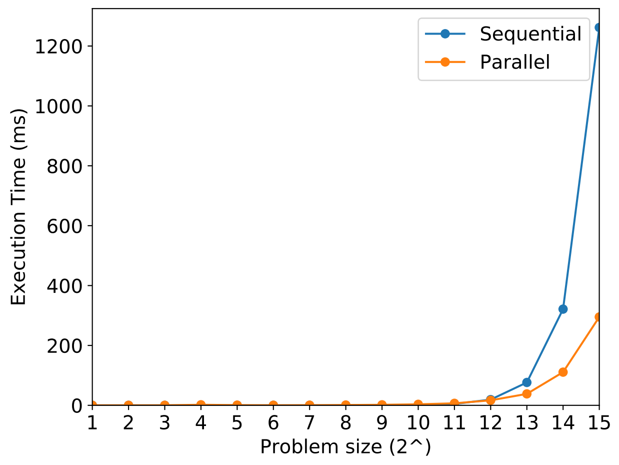

Figure 1: Bubble Sort - Sequential vs Parallel

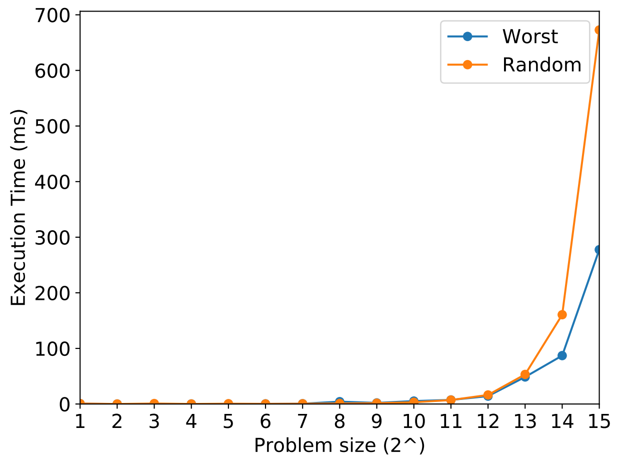

Figure 2: Bubble Sort - 'Worst Case' vs Random Array

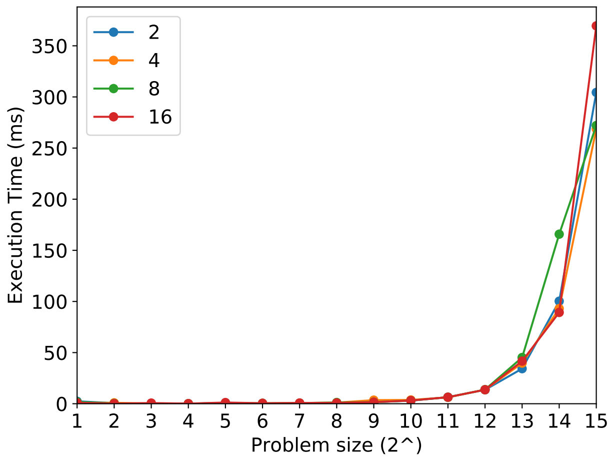

Figure 3: Bubble Sort - Thread Variation

4.2 Merge Sort

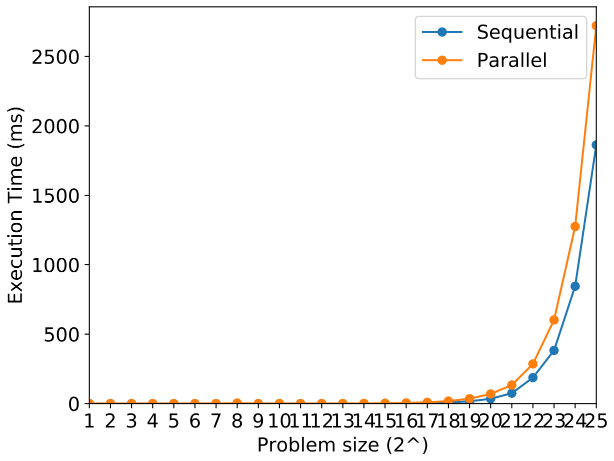

Figure 4: Merge Sort - Sequential vs Parallel

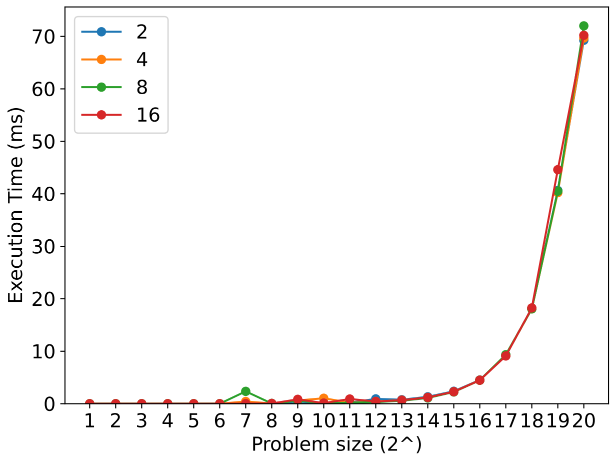

Figure 5: Merge Sort - Thread Variation

4.3 Odd-Even Sort

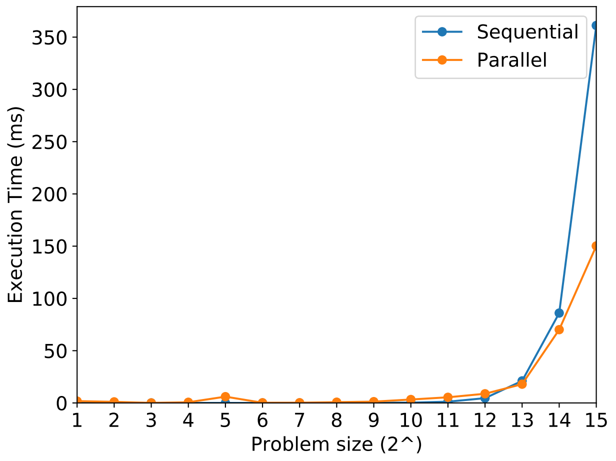

Figure 6: Odd-Even Sort - Sequential vs Parallel

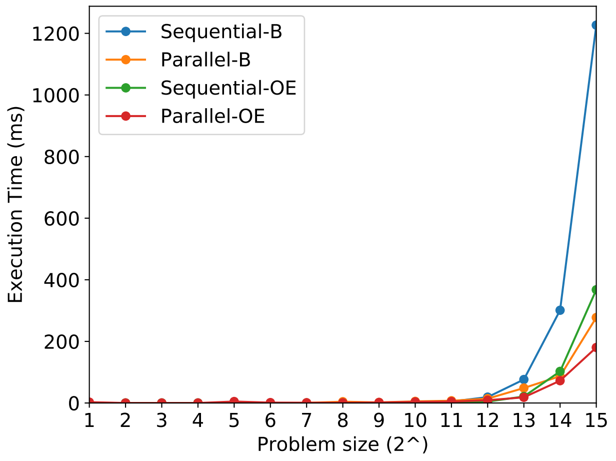

Figure 7: Odd-Even Sort - Sequential vs Parallel

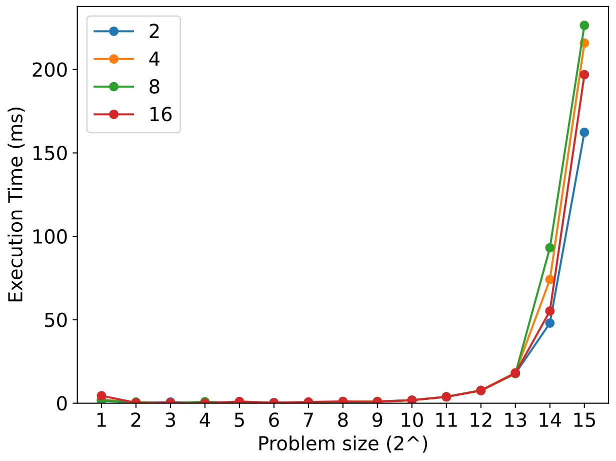

Figure 8: Odd-Even Sort - Thread Variation