TASK 2

As a group, we tested each algorithm's performance by implementing both serial and parallel versions. The parallelization aimed to reduce the time taken by using multiple threads to process different parts of the data simultaneously.

1. **Quick Sort**: We implemented Quick Sort as a divide-and-conquer algorithm that splits the array into smaller sections and recursively sorts them. For the parallel approach, we used multiple threads to handle different sections of the array at the same time. This parallel version showed significant improvements in speed for large datasets.
2. **Bubble Sort**: As a group, we also tested Bubble Sort, which is a straightforward algorithm that repeatedly compares and swaps adjacent elements. Even though Bubble Sort is not efficient for large arrays, we attempted to parallelize it by assigning threads to different sections. However, it didn't show much improvement, and the performance remained poor compared to other algorithms.
3. **Merge Sort**: For Merge Sort, we used the same divide-and-conquer strategy, splitting the array into two halves, sorting them, and then merging the sorted halves. We parallelized the sorting step but kept the merging process serial. Like Quick Sort, Merge Sort benefited from parallelization for large arrays, though not as much as Quick Sort.

Key Observations

- **Speedup**: As a team, we carefully compared the performance of the parallel algorithms against their serial counterparts. Quick Sort showed the best speedup, reducing its runtime from 0.86 seconds (serial) to 0.43 seconds with just 2 threads. As we added more threads, Quick Sort's time continued to decrease, especially for large arrays. Merge Sort also improved with more threads, but the improvement wasn't as dramatic.
- **Scalability**: We evaluated how each algorithm handled an increasing number of threads. Quick Sort showed the best scalability, with its performance improving significantly as more threads were added. Merge Sort also scaled well but to a lesser extent. Bubble Sort, on the other hand, showed poor scalability, with little improvement even as we increased the number of threads.
- **Memory Efficiency**: During testing, we found that Quick Sort was the most memory-efficient, requiring less additional space than the other algorithms. Merge Sort, which requires extra memory for temporary arrays during the merging process, became slower as the array size increased. Bubble Sort, while not memory-intensive, was extremely slow and inefficient for large datasets.
- **Overhead**: We also noted that adding threads introduces some overhead in managing them, which can reduce the performance benefits of parallelization. While Quick Sort and Merge Sort had minimal overhead, Bubble Sort's overhead was significant, making it even slower when parallelized.

Results

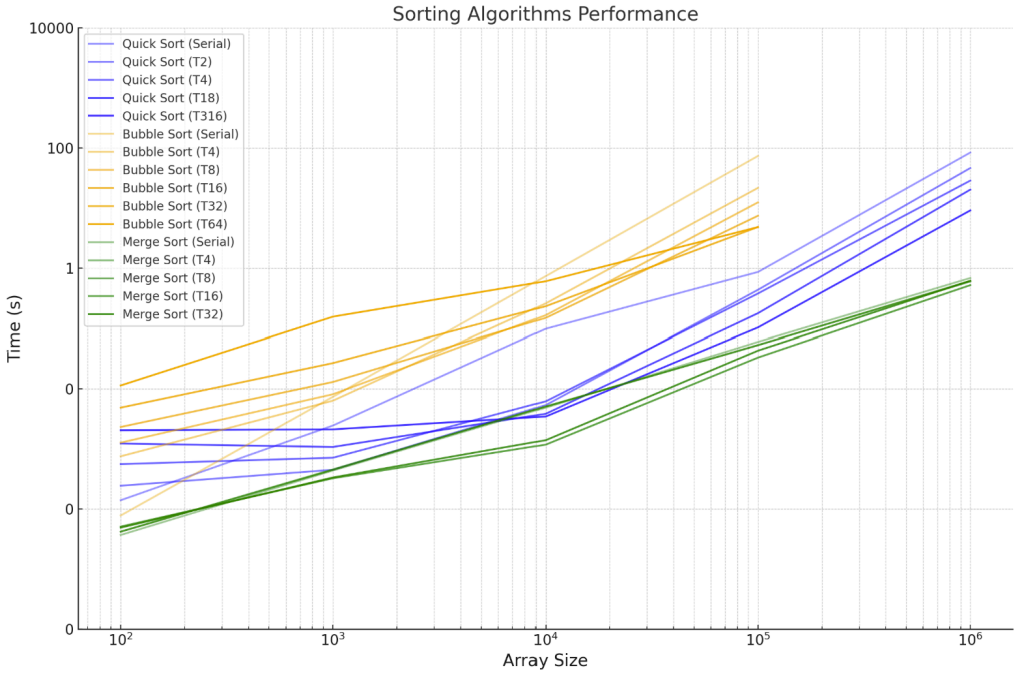
- **Quick Sort**: In our tests, Quick Sort with 1 thread took 83.5 seconds to sort an array of 1,000,000 elements. However, using 18 threads reduced this time to just 9.09 seconds. This showed that Quick Sort scales very well with more threads, making it the fastest algorithm for large datasets.
- **Bubble Sort**: For 1,000,000 elements, Bubble Sort took about 7.65 seconds with 64 threads. Despite the parallelization, Quick Sort still finished the task much faster, in just 0.68 seconds. This highlighted that Bubble Sort is not suitable for large arrays, even with parallelism.
- **Merge Sort**: With 64 threads, Merge Sort took 5.92 seconds to sort an array of 1,000,000 elements. Although this was faster than Bubble Sort, it was still slower than Quick Sort. We concluded that Merge Sort can benefit from parallelization but doesn't perform as well as Quick Sort, especially for large datasets.

Conclusion

After extensive testing and analysis, we concluded that:

- **Quick Sort** is the most efficient choice for sorting large arrays, both in terms of speed and scalability. It performs best when parallelized and significantly outperforms the other algorithms, especially as the array size increases.
- **Merge Sort** is a good alternative but requires more memory and doesn't scale as well as Quick Sort. It's still a solid choice for parallelization but is slower in comparison.
- **Bubble Sort** should be avoided for large datasets. Even with parallelization, it remains inefficient and far slower than both Quick Sort and Merge Sort.

In summary, Quick Sort proved to be the optimal choice for large-scale sorting tasks. Our findings suggest that parallelizing Quick Sort provides the most significant performance improvements, making it the best option for handling big data efficiently



References: Used chatgpt and perplexity ai to generate graph and to check code performance and syntaxes.