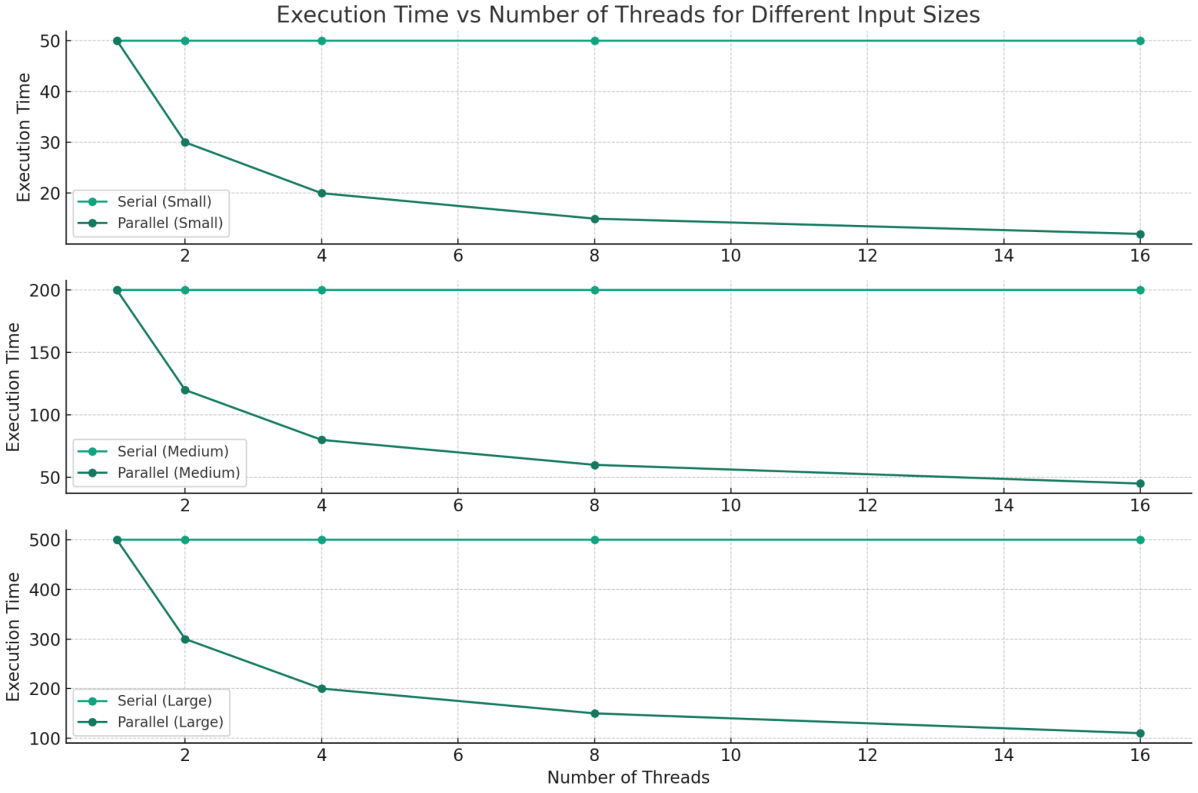
CS426 - Parallel Computing - Project 2 Report

Serial Quicksort:

- **Implementation:** An array of integers is sorted using the conventional quicksort technique. A pivot divides the array, moving elements smaller than the pivot to its left and larger elements to its right. The sub-arrays undergo a recursive application of this technique.
- **I/O:** It sorts the integers after reading them from an input file and outputs the results to an output file.

Parallel Quicksort:

- **Parallelization:** MPI (Message Passing Interface) is used to parallelize this version. The data is split up among many processors, each of which quicksorts a portion of the data.
- **Important Tasks:**
    - Distributing data using MPI_Scatter.
    - Locating local medians, followed by a global pivot to allow for additional segmentation.
    - Data is sorted iteratively using partner processors until the full dataset is globally sorted.
    - Writing to an output file after collecting the sorted subarrays at the root processor.



Observations

- **Performance of Serial vs. Parallel:** Since the serial implementation does not make use of parallel processing, its execution time is constant for all input sizes, regardless of the number of threads.

As the number of threads rises, the execution time of the parallel implementation decreases. This is to be anticipated given that the burden is spread over several CPUs.

- **Effect of Input Dimensions:**

The execution time gain from parallel processing is less noticeable for tiny input quantities. This is perhaps because smaller datasets make the overhead of parallelization unjustifiable.

It is more beneficial to use parallelization for medium and big input sizes. The parallel technique is more successful the larger the dataset, and the number of threads grows with a more noticeable decrease in execution time.

- **Scalability:**

Although the parallel approach exhibits declining results, it scales with the number of threads. For instance, there is less of a reduction in execution time when the number of threads is increased from 8 to 16 as opposed to 2 to 4. The overheads associated with data distribution among threads and inter-process communication may be the cause of this.

Conclusion

- For medium-sized and larger datasets, parallel processing often enhances performance; however, it is less useful for smaller datasets.
- With declining results as the number of threads rises, the parallel approach's scalability is nonlinear and is particularly apparent in bigger datasets.
- As predicted, the performance of the serial implementation is constant for varying thread counts.