Casey Key

Lab 2: OpenMP Tasks

1. Algorithm Description

The algorithm used in this experiment is merge sort. Merge sort uses two recursive calls to split an array of numbers into halves, creating a binary tree. Once the tree has been built, leaves are merged into temporary arrays of size 2^x, where x is number of levels from the leaf nodes. The merging step must examine every element of the array, which is at most n. The binary tree has a maximum height of log(n). Since we merge for every level of the binary tree, our worse-case time complexity is O(nlogn). The space complexity for this algorithm is n, as that is the maximum size of the temporary array for the sorted elements.

2. Parallelization Approach.

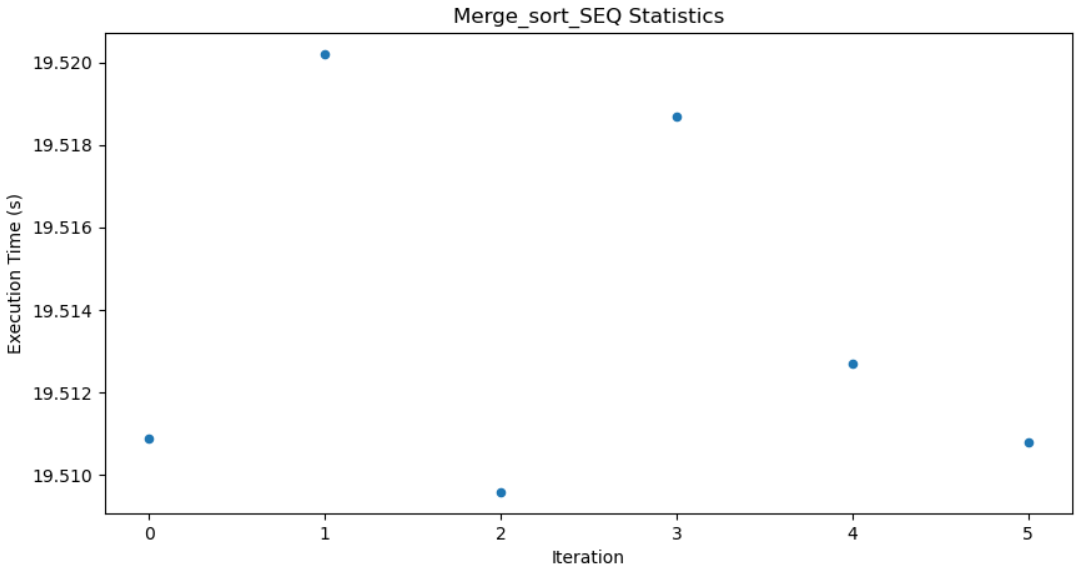
To parallelize this algorithm, we used many incremental steps to achieve the best performance. First, we used tasks to execute recursive calls in parallel. Then, we noticed overhead associated with creating new tasks outweighed the benefits of parallelization. So, we moved on to conditionally creating tasks, also known as pruning. After this, we noticed the master thread was not executing any tasks, so we restructured the code to keep the master thread busy.

3. Experimental Setup

I used the Cheyenne state capital machine for this experiment. This machine has 8 cores. The L1d and L1i cache are 256 KiB, the L2 cache is 2 MiB, and the L3 cache is 20 MiB. I am the only user on this machine.

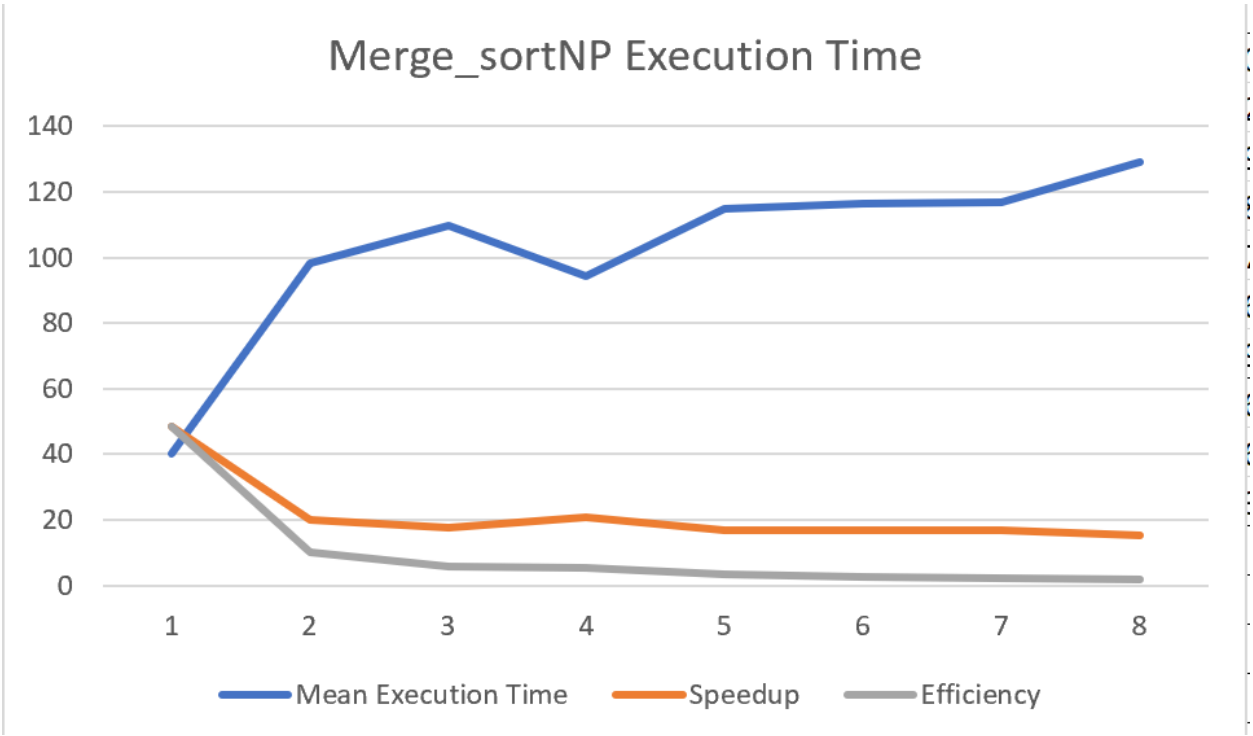
4. Experimental Results

To start the experiment, I began with finding the average execution time for the sequential merge sort algorithm. Here is a plot of the results.

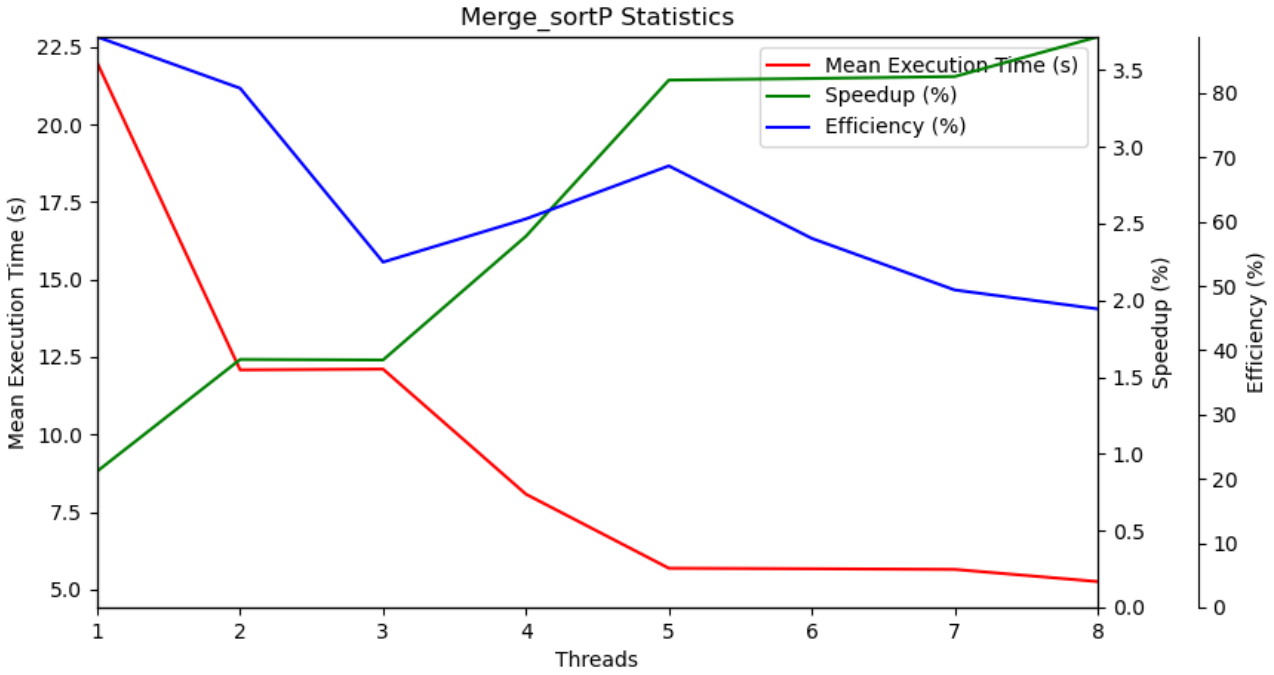


The average execution time was 19.5138 seconds.

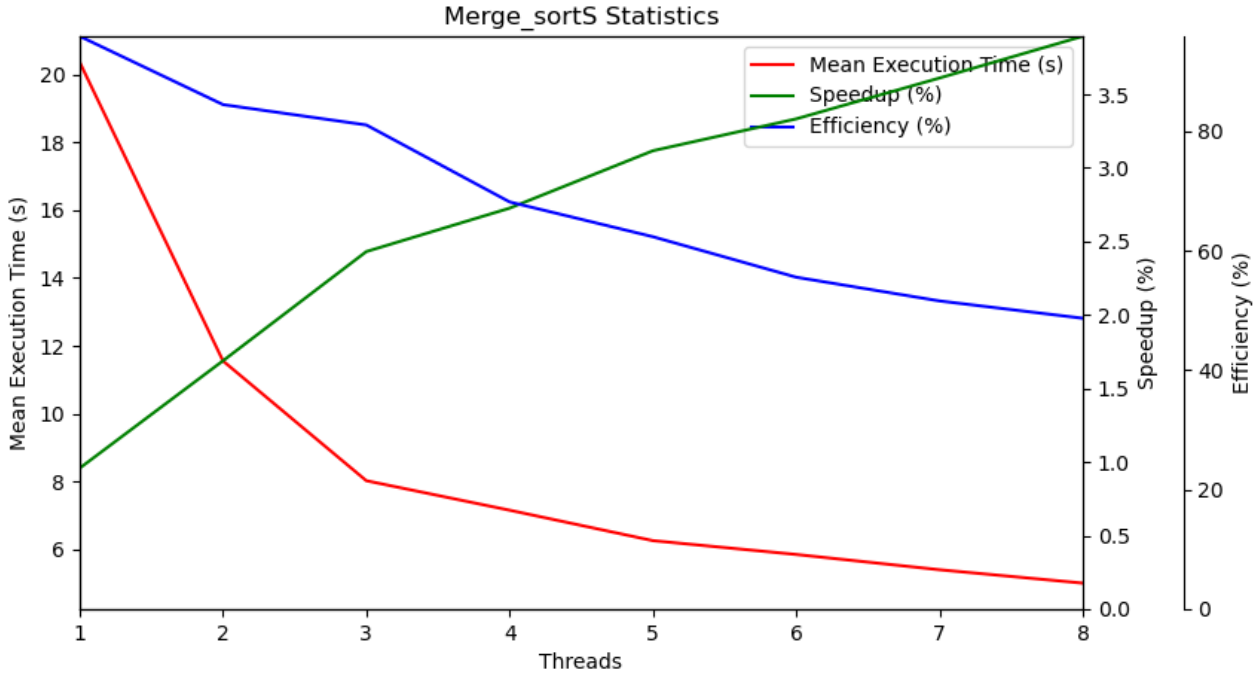
Next, I tested the Merge_sortNP program from 1 to 8 threads. This program spawned a task for every recursive merge sort call.



The next test was conducted with Merge_sortP from 1 to 8 threads. This program spawns tasks for merge sorts when the array to sort is larger than 1000 elements.



Finally, I ran an experiment using Merge_sortS which spawns tasks for one of the recursive calls, allowing the master thread to work on the other recursive call.



5. Conclusion

In conclusion, we see that it's important to only spawn as many tasks as we need. IF we spawn too many, the time it takes to manage the threads outweighs the speedup gained from parallelization. Once we have found an optimal number of tasks, we can further optimize by making sure all threads have work. This is evidenced by the final speedup of 3.895 when using Merge_sortS versus the final speedup of 3.715 when using Merge_sortP, both using 8 threads.