Corinne Jones
CS6013-Operating Systems

Assignment 6 Analysis Doc

Strong Scaling

The N (array size) was fixed for the “Strong Scaling” tests, and num_threads (number of threads) was incremented by one from 1-16. The input array was then increased and run again with the number of threads from 1-16. This test was run for the three functions: std, omp1, and builtin. The results are shown in **Figures 1.1**, **1.2**, and **1.3**. The numbers next to the solid colored line on the X-axis demonstrate the array size.

As all three figures demonstrate, the timings dramatically decrease until an optimal threshold is reached. This drastic decrease demonstrates the improved efficiency of thread creation. From here, the graphs flat-line, suggesting that the extra work of creating a thread does not increase efficiency as it did previously. For example, the creation of threads takes significant overhead. If each thread has a small enough portion to add, it could complete the addition while other threads continue to be created. This would prove less efficient since threads would be waiting for other threads to be created while they could be performing work.

The timings here are demonstrated with an array of integers. The question was posed, “Does the timing change if you sum integers vs longs vs floats vs doubles? When the same test was run with arrays of integers, it took significantly longer to perform the additions than when compared with integers. This suggests that the variable type affects speed and performance during parallelization. Due to the larger data type, more memory bandwidth is consumed. Floats and doubles would be expected to take longer than integers.

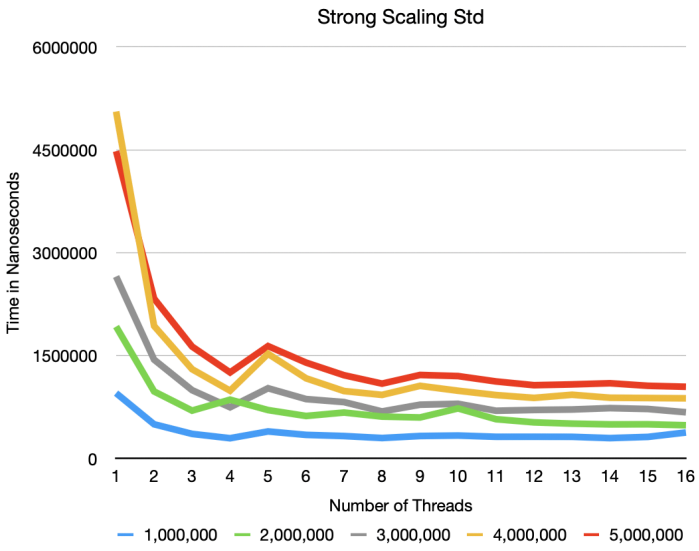

Figure 1.1

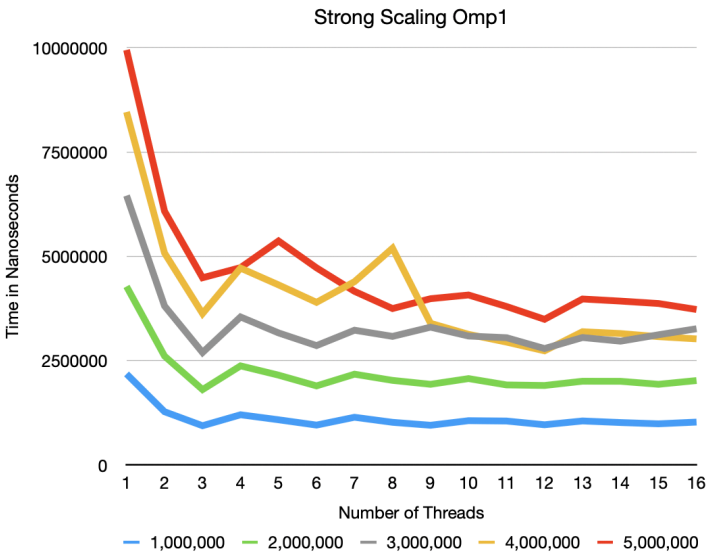

Figure 1.2

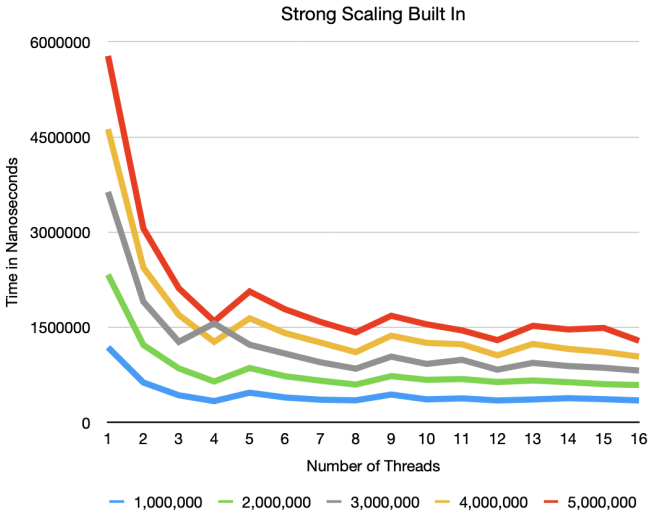

Figure 1.3

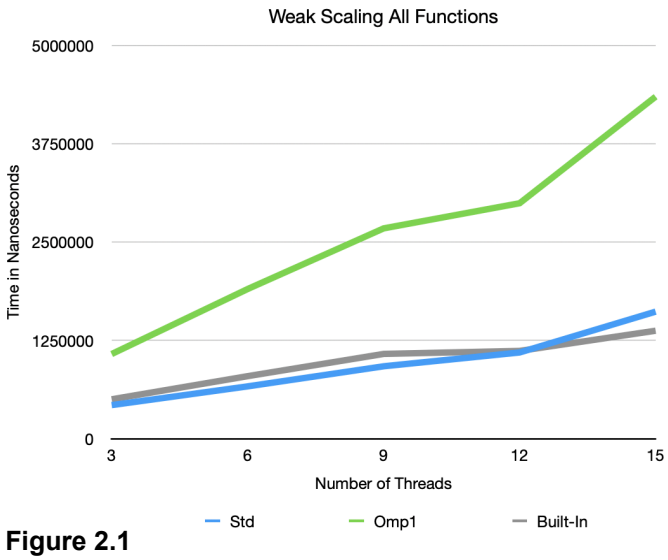

Figure 2.1

Weak Scaling

N (array size) and num_threads (the number of threads) were increased together for this test. The number of threads was increased by increments of three, and N was increased in increments of 1,000,000. All three methods performed this test. The results are demonstrated in **Figure 2.1**.

All three of the graphs show weak scaling. This graph demonstrates that the std and built-in function had weaker scaling than Omp1. This is due to how parallelization was managed in each function. Omp1 could have more overhead in the thread creation than in the other two functions. There are small increases in timing, which would be expected based on how much N was increased in correlation to num_threads. Std and Built-in roughly stay constant, as would be expected. As the workload increases, the threads increase, making the timing relatively consistent.

Comparison

Between the three different functions, the OpenMP built-in reduction and the standard C++ threads (std) function exhibited similar performance levels. However, the standard function required manual creation and synchronization of threads, which was more time-consuming and prone to errors due to the complexity of managing threads manually. Interestingly, the OpenMP built-in reduction performed slightly slower than expected. Although the distribution and synchronization of the threads were handled automatically—making it straightforward to implement—I had anticipated the built-in reduction to outperform the standard implementation due to its presumed efficiency and reduced overhead. This unexpected result suggests other factors influencing the performance, such as the overhead of managing parallel regions or compiler optimizations.