Aiden Pratt
6013 A6 Threading

see the next page for the graphs of the strong scaling tests for each of my sum implementations.

Do the curves match the ideal case? If not, why? If the timing doesn't appear to be scaling, try this stage again but start with 100,000 or 1,000,000 elements for your array. Does the timing change if you sum integers vs longs vs floats vs doubles?

I believe my curves do match the ideal cases. This is because as the number of threads are increased, the time is decreased, meaning the threads are helping compute the correct summation of the numbers in the array at a quicker rate than if just one thread were to do so.

When considering the timing and scaling factors, starting with 1,000 elements might have been too short to capture the full performance potential due to the overhead involved in allocating cores and the rapid execution of addition operations.

For instance, when you start with a larger array size, such as 100,000 or 1,000,000 elements, you allow more time for the system to allocate resources efficiently, which can lead to better scaling results. The timing differences observed when summing integers, longs, floats, and doubles may not vary significantly because modern processors often optimize these operations similarly, especially when dealing with large datasets.

Overall, the scaling and timing optimizations depend on various factors such as the hardware architecture, compiler optimizations, and the nature of the operations being performed. By experimenting with different array sizes and data types, you can gain insights into how these factors impact the performance of your implementations.

Produce a single graph containing results from all 3 functions. Does the curve match the ideal case? If not why?

The curve for this graph does match the ideal case. It is clear that as the thread count increases and the number of elements within the test array increases, the time remains consistent.

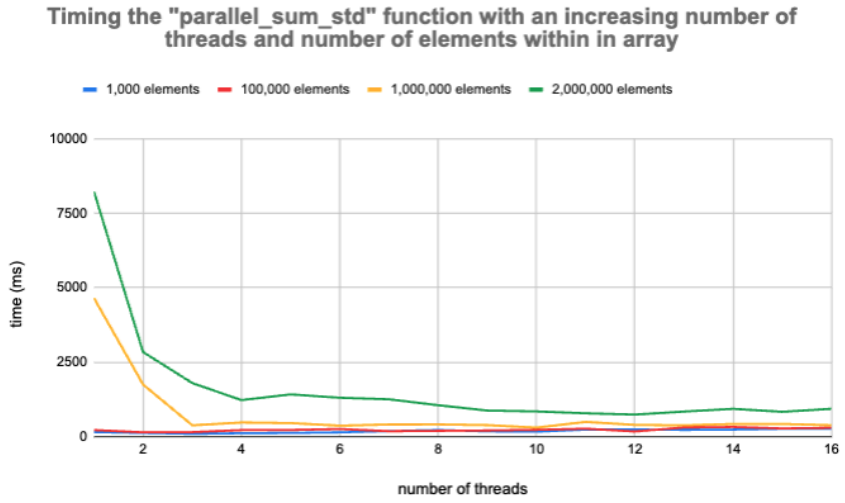
3. Comparison: Which of the three methods above performs best on each test? Explain why.

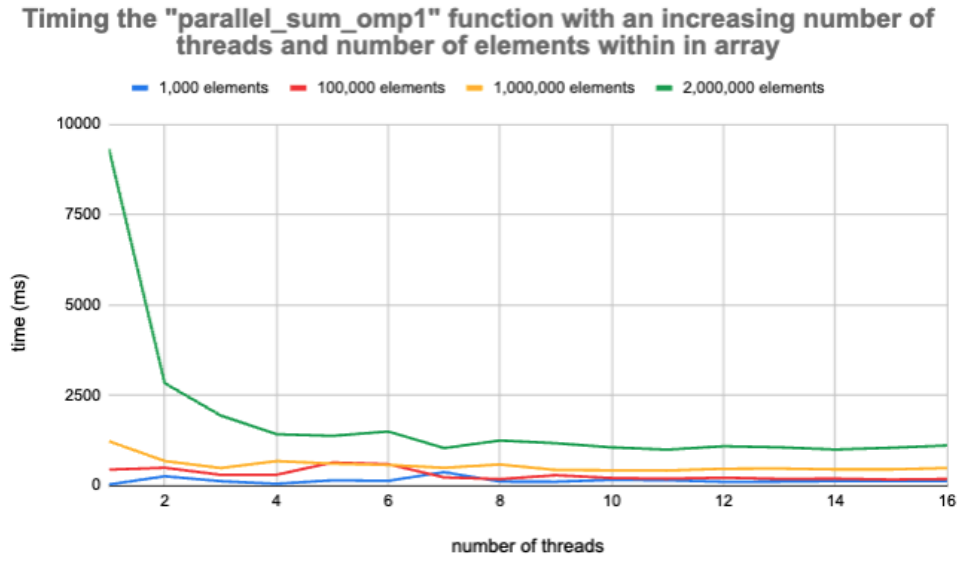
All of the tests appear to perform very similarly for both of the tests. This suggests that some sort of performance bottleneck may lie elsewhere. For example, in memory access patterns, cache utilization, or other system level factors. If I had to pick which performed the best based on my experiments and the charts they created, I would choose the parallel

sum omp1 function I wrote. This method specifically uses #pragma verbiage within the method to ensure proper thread handling of the summation of the numbers. When the threads are operating as efficiently as possible, it will result in faster timing graphs with the accurate answer.

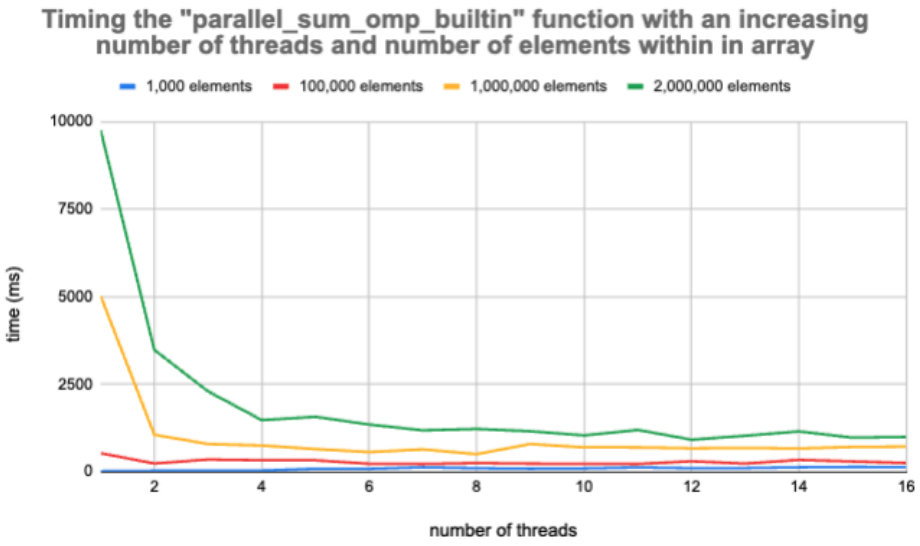
strong scaling graphs:

In the strong scaling tests, it is apparent that all functions for the threading summation performed very similarly. First, it is important to note that the addition of 1,000 and even 100,000 elements result in a near negligible time increase or decrease simply because that is too small of an array. When I tested these functions with 1,00,000 and 2,000,000 elements, it is apparent that there is a greater initial time spent with fewer threads. This time is dramatically quicked as more threads are introduced and used by the function to correctly sum the elements in the array.





Aiden Pratt
6013 A6 Threading



Aiden Pratt
6013 A6 Threading

weak scaling graph:

In this graph I portray threading parallel sums across the three different functions I wrote. The weak scaling test assumes the work is evenly distributed across threads, and increases the size of the array of numbers to be summed alongside the number of threads being utilized by the algorithm. this results in a very consistent looking line chart across each function. In this chart, the increase in size of array is seen to gradually take slightly more time, but is consistent across each function tested.