Brandon Mountan

04/18/2025

CS 6013

Parallel Sum Implementation Analysis

Looking at these parallel sum implementations, it's clear that none of them hit that perfect speedup we'd hope for when adding more threads. When I ran the strong scaling tests (keeping the array size the same while adding threads), the speedup was decent but definitely not linear - doubling threads didn't cut the time in half. This makes sense when you think about it - threads have to communicate, share memory, and eventually combine their results, all of which adds overhead.

I noticed bigger arrays (10 million elements) scaled much better than smaller ones. With smaller arrays, the overhead of creating and managing threads often outweighed the benefits of parallelization. As for data types, integers were consistently faster than floating-point numbers across all implementations. Floats beat doubles too, which isn't surprising since they're half the size and easier for the memory system to handle.

The weak scaling tests were interesting - ideally, if you double both the work and the threads, execution time should stay roughly the same. But in reality, all implementations showed increasing times as thread counts grew. Memory bottlenecks and the increasing cost of combining partial results are likely culprits here.

Among the three approaches, the built-in OpenMP reduction generally performed best, especially for larger arrays. It benefits from OpenMP's optimization magic - the runtime system and compiler know the hardware intimately and can make smart decisions about thread management. The custom OpenMP implementation did well with integers but wasn't as great with floating-point math. The STL threads version offered good control but couldn't match OpenMP's performance as thread counts increased.

Bottom line: if you want the best performance with minimal effort, go with built-in OpenMP reductions. The compiler optimizations and lower thread overhead make it the clear winner for most real-world scenarios.