

Homework 2: Parallel Prefix Sum

Stella Greenvoss
11/8/2025 — CS431
All computation done on ix-dev

In this report, three algorithms for computing the prefix sum of an array are compared. The parallel algorithms are both naively implemented using simple OpenMP pragmas and evaluated on ix-dev, which is not optimized for parallel programming.

1 Runtime

The runtime of each method can be seen in Figure 1. In this figure, n is constant at 33554432, with the number of threads varying.

The serial prefix is computed conventionally, with a single for loop computing the value $x[i]$ as $x[i] = x[i] + x[i - 1]$. As expected, the runtime is invariant at different thread counts because the program runs serially.

The first parallel prefix sum runs in $O(n \log n)$. Theoretically, it should exhibit speedup when the thread count increases, since the outer loop only runs $\log n$ times. Given that the inner loop is parallel, the function might be able to run in $O(\log n)$ given n threads. However, clearly this was the slowest function of the three. I think this can be attributed to the high value of n . If I had access to more threads or was writing CUDA code, the parallel section might run near-instantaneously (albeit with the overhead that comes with parallelization). There is also a conditional section in this for loop, which could also account for some of the slowdown.

The second parallel function runs about twice as fast as the serial version. This utilizes the binary tree approach to computing the prefix sum. There are three parallel sections in my code: one to copy the values from the source array to the prefix array, one to compute intermediate sums, and one to walk down the tree and finish the sums. This function was significantly harder to write, because it is difficult to see how the parallel sections are actually independent from one another. It also currently only works for arrays that have a length that is a power of two. This

is due to time limitations (and because the author is writing without the assistance of generative AI); a general implementation would allow for these arrays.

Both of these parallel functions are written without much consideration to data or spatial locality. Data is often accessed in steps that are greater than what any high-level cache could reasonably hold efficiently. I think that a more clever implementation should better account for locality, maybe working in chunks, leading to improved speedup.

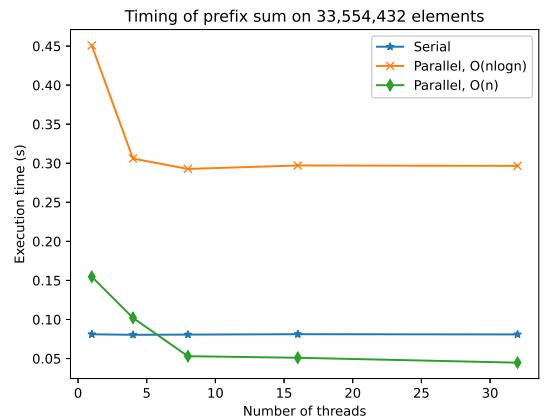


Figure 1: Runtime for each prefix sum computation at each thread count, which varied from 1 to 32.

2 Discussion

I am disappointed that I was not able to get better results during this investigation. Additional analysis of my current code should include running it at varying numbers of n and on Talapas. After more thorough analysis, I would try to identify the biggest bottlenecks (*e.g.*, the conditional section in the first parallel function, perhaps) and eliminate them. Reimplementation in CUDA could also yield interesting results.