Parallelizing Prefix Sum

Carmen Park

November 9, 2025

1 Prefix Sum Algorithm

While writing a serial prefix algorithm is relatively easy, it seems there have been many parallel implementations. For our second homework assignment, we followed two popular existing algorithms, the Hillis steel and Blelloch scan.

1.1 Hillis Steele Scan

The Hillis Steele Scan [3] was the first parallel implementation we were tasked with. It is an inclusive scan that was popularized in 1986 paper by Danny Hillis and Guy Steele. Given an array of length n, it builds prefix sums in $\log n$ iterations. It has a computational complexity of $\mathcal{O}(n \log n)$ since it performs $\log(n)$ operations n times.

In both prefix sum functions, I started by copying the source array to the destination prefix array. While this introduces an initial $\mathcal{O}(n)$ memory copy operation, it was important to me to keep the source data untouched throughout the computation. Looking back, I could have probably could have better optimized this with different strategies, especially with the second function.

From there, my first approach of the Hillis Steele Scan was to iterate through the strides, and parallelize the summation of the array.

```
for (int s = 1; s < n; s *= 2){
    #pragma omp parallel for
    for (int i = s; i < n; i++){
        prefix[i] += prefix[i - s];
    }
}
```

This immediately led to race conditions, where multiple threads were accessing and rewriting to memory, and giving an inaccurate prefix sum. I tried inverting my loop for a reverse iteration (it looked very similar to the code above but the indices were essentially flipped), and another solution could have been to use atomic operations, but I finally decided to create another temporary array outside of the loop to help buffer this access issue.

```
#pragma omp parallel for
for (int i = 0; i < n; i++){
    if (i < stride){
        temp[i] = prefix[i];
    }else{
        prefix[i] = temp[i] + temp[i - s];
    }
}
```

While this did introduce $\mathcal{O}(n)$ memory overhead and additional copy operations, this strategy helped correct how the parallel execution accessed any sample array size.

One thing I did notice was that unless there were many cores and the problem was quite large, the sequential prefix sum ran often twice as fast as the Hillis Steele prefix sum. This would be expected given their time complexity, but it still felt surprising.

1.2 Blelloch Scan

Guy Blelloch popularized another scan in 1990 that also acts as a prefix sum [2] when using addition. It has a $\mathcal{O}(n)$ work efficiency with and has a bit more complex of an algorithm. I found this system more intuitive and symmetric, and while there are most likely steps to better optimize my code, I feel that I have a good foundation of understanding to now build off of.

I went about this by testing a very small sample size, and testing first the reduction, followed by the down sweep. My biggest issue was potentially with how I did not see any race conditions in both of these implementation. For example,

```
for (int s = n/2; s >= 1; s /= 2){

#pragma omp parallel for
    for (int i = (s*2-1); i < n; i+= s*2){
        prefix[i] = temp[i] + temp[i - s];
        prefix[i-s] = temp[i];
        temp[i] = prefix[i];
        temp[i-s] = prefix[i-s];
    }
}
```

this was how I implemented the down sweep. I ran a variety of array sizes from the terminal and all passed; they were just as correct as the serial prefix sum. My issues with this though are that there could be access errors, and I was under the assumption that this pattern of reading and writing to the same arrays in parallel would never produce correct prefix sums due to data dependencies between threads.

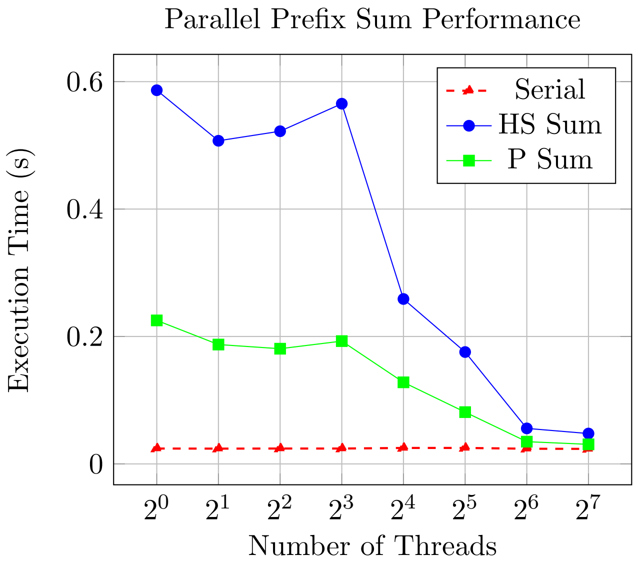
2 Reflection

Reflecting on my work for this assignment, I feel a bit unresolved and curious as to how I was able to not see issues with my second implementation of prefix sum.

Some solutions could be reading and writing in separate parallel sections, or using temporary variables, and possibly just using a single array rather than using two as I had in the first prefix sum implementation.

I have kept it as it is for now, but it makes me want to know more about how race conditions happen, since it might be more complex than my current understanding of what leads to them.

Another article that helped explain some of these methodologies was the NVIDIA "Parallel prefix sum (scan) with CUDA" [1]. Working with openMP to produce these results was very rewarding. An easy step from here could be to run the same algorithms on a GPU with CUDA.



The overall performance times with only 33,554,432 elements was not very optimized for the parallel implementations. There was improvement consistently as thread count increased from 1 to 128 threads which does suggest that with larger problem sizes the algorithms would have more favorable speedup ratios.

This "slowdown" definitely leads me to believe that successful parallelization requires balancing well designed algorithms with adequate architectural constraints. While I am still learning how to orchestrate these aspects, I hope to better understand algorithmic constants and memory hierarchy effects in future assignments.

References

[1] Harris, M., Sengupta, S., & Owens, J. D. (2007). Parallel prefix sum (scan) with CUDA. In H. Nguyen (Ed.), *GPU Gems 3* (Chapter 39). Addison-Wesley Professional. https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda

[2] Udacity. (2015, February 23). *Blelloch scan - intro to parallel programming* [Video]. YouTube. `https://www.youtube.com/watch?v=mmYv3Haj6uc`

[3] Udacity. (2015, February 23). *Hillis Steele scan - intro to parallel programming* [Video]. YouTube. `https://www.youtube.com/watch?v=RdfmxfZBHpo`