CS 39000-ATA: Homework 1

Due on 2025-01-30 23:59

Prof. Kent Quanrud, Spring 2025

Mukul Agarwal and Kovid Tandon

Additional Collaborators: Aaryan Wadhwani and Jayden Lee.

Problem 3.6

Given a set of numbers $x_1, ..., x_n$, the prefix sums are the n numbers,

$$x_1, x_1 + x_2, x_1 + x_2 + x_3, ..., x_1 + x_2 + \cdots + x_n$$

Of course the prefix sums can be computed in $\mathcal{O}(n)$ time, one prefix sum at a time. What about in *parallel*; e.g., on a GPU?

Here we consider a *parallel* computation model, called *PRAM*. To motivate this model, imagine one had arbitrarily many processors, with shared memory, trying to solve a common problem. Even with tons of computing, there may still be sequential bottlenecks — some steps have to be executed before others — and issues of synchronization.

In the PRAM model, the processors have access to shared memory. To simplify the synchronization, we restrict our model to work in *rounds*. In each round, each processor reads a constant amount of information from memory, does a constant amount of work, and writes to a constant number of locations in memory. We will restrict ourselves to algorithms where in each round, different processors always write to different locations, so we can avoid issues of contention resolution.

The number of rounds needed to complete the task is the *parallel running time*. The total number of operations, over all processes, is called the *total amount of work*. The ideal parallel algorithm has total work similar to the best known sequential algorithm, but has a much faster parallel running time.

For example, here is a first attempt at a parallel divide-and-conquer algorithm for prefix sums. We first define our recursive spec:

> prefix-sum($A[1..n]$): Given an array of numbers $A[1..n]$, returns an array of the prefix sums of A.

Our first implementation divides A into equal-size subarrays and recursively computes the prefix sums on each. Observe that the prefix sums of the second half are missing the sum from the first half. We fix this by taking the last prefix sum of the first half and adding it to each prefix sum in the second half. See fig. 3.8 for pseudocode.

Let $T(n)$ denote the parallel running time on an input of size n. We have

$$T(0) = T(1) = 1$$

$$T(n) = T\left(\frac{n}{2}\right) + \mathcal{O}(1) \qquad \text{for } n > 1$$

```
prefix-sum($A[1..n]$)
    if $n \leq 1$ then return $A$
    let $k \leftarrow \lceil n/2 \rceil$
    let $B[1..k]$       $\leftarrow$ prefix-sum($A[1..k]$)
    let $C[1..n-k] \leftarrow$ prefix-sum($A[(k+1)..n]$)
    $C[i] \leftarrow B[k]$ for $i$ in $\{1, ..., n-k\}$          // $\mathcal{O}(n)$ work in $\mathcal{O}(1)$ rounds
    return the concatenation of $B, C$              // $\mathcal{O}(n)$ work in $\mathcal{O}(1)$ rounds
end algorithm
```

Here we observed that although parallel prefix sum has two recursive calls, they are made in parallel. This is the same recurrence as (sequential) binary search, and implies a $\mathcal{O}(\log n)$ parallel running time.

Now we compute the total amount of work. This is the same as the running time if the parallel-prefix-sum algorithm was simulated sequentially. Letting $W(n)$ be the total amount of work on an input of size n, we have

$$\begin{aligned} W(0) &= W(1) = 1 \\ W(n) &= 2W(n/2) + n/2 && \text{for } n > 1. \end{aligned}$$

This parallel algorithm is very fast, but has a $\mathcal{O}(\log n)$-factor more work than the obvious sequential algorithm. The goal of this exercise is to develop a (recursive) parallel divide-and-conquer algorithm that is just as fast — $\mathcal{O}(\log n)$ parallel time — but has $\mathcal{O}(n)$ total work. In addition to designing the algorithm, you should analyze the running time and the work similar to how we did above. (You are welcome to use the same recursive spec, but don't forget to write it out.)

It may be fun to try to figure this out without any hints. (Maybe you'll come up with a new approach!) But if you want a little help, then there are some guiding questions in the following footnote.[1]

---

[1]Hint: Suppose you had the prefix sums $y_i = x_1 + x_2 + \cdots + x_i$ for all even i. Can you get the prefix sums for all odd i in $\mathcal{O}(1)$ rounds and $\mathcal{O}(n)$ work?

And then: how do you get the prefix sums for even i efficiently? Of course you could compute all the prefix sums, and then pick out the even indices i. But that's not going to be help us speed up computing all the prefix sums. Can you get the prefix sums for even i by creating another instance of the prefix sums problem with significantly fewer numbers?

Our algorithm is given as follows.

prefix-sum($A[1..n]$): Given an array of numbers $A[1..n]$, returns an array of the prefix sums of A.

```
prefix-sum($A[1..n]$)
    **if** $n \leq 1$ **then return** $A$
    **let** $k \leftarrow \lfloor n/2 \rfloor$
    **let** $B[1..k] \leftarrow 0$
    $B[i] \leftarrow A[2i-1] + A[2i]$ **for** $i$ **in** $\{1, ..., k\}$ // $\mathcal{O}(n)$ work in $\mathcal{O}(1)$ rounds
    **let** $C[1..k] \leftarrow$ prefix-sum($B[1..k]$)
    **let** $D[1..n] \leftarrow 0$
    $D[1] \leftarrow A[1]$
    $D[2i] \leftarrow C[i]$ **for** $i$ **in** $\{1, ..., k\}$ // $\mathcal{O}(n)$ work in $\mathcal{O}(1)$ rounds
    $D[2i+1] \leftarrow C[i] + A[2i+1]$ **for** i **in** $\{1, ..., k\}$ // $\mathcal{O}(n)$ work in $\mathcal{O}(1)$ rounds
    **return** $D$
**end algorithm**
```

Proposition 2.1: The algorithm prefix-sum is correct (i.e., it adheres to its specification).

Proof: We will prove that the algorithm prefix-sum correctly implements its specification by induction on n. In the base case where $n \leq 1$, the list of prefix sums is exactly A, which is correctly returned. Consider the general case where $n \geq 2$. We assume by induction that the implementation is correct for all strictly smaller arrays.

We collapse A into a smaller array B, where $B[i] = A[2i-1] + A[2i]$. This ensures that the ith element in B is the sum of the corresponding values in the ith even index of A and the $i-1$th odd index of A. By induction, $C \leftarrow$prefix-sum(B) correctly gives us an array of all the prefix sums of B. Notice that

$$C[i] = \sum_{j=1}^{i} B[j] = \sum_{k=1}^{i} (A[2k-1] + A[2k])$$

Thus $C[i] = A[1] + \cdots + A[2i]$. Therefore, C is an array of all the prefix sums of all even indices of A. Furthermore, notice that $(C[i] + A[2i+1]) = A[1] + \cdots + A[2i] + A[2i+1]$, which is the correct prefix sum for the $2i + 1$(odd) index of A.

We can then correctly compute all the prefix sums of A since $A[1]$ is the correct prefix sum for $A[1]$, $C[i]$ is the correct prefix sum for all even indices ($2i$) of A, and $(C[i] + A[2i+1])$ is the correct prefix sum for the off indices ($2i + 1$) of A. Thus, the algorithm correctly calculates all the prefix sums and the proof is complete. $\square$

Proposition 2.2: The algorithm prefix-sum has running time $\mathcal{O}(\log n)$ under a parallel model of computation.

Proof: Notice that the runtime of the algorithm is similar to the naïve parallelized algorithm. Letting $T(n)$ denote the parallel running time on an input of size n, we have

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ T(\lfloor n/2 \rfloor) + \mathcal{O}(1) & \text{otherwise} \end{cases}$$

and so we have that $T(n) \in \mathcal{O}(\log n)$. □

Proposition 2.3: The algorithm prefix-sum has total amount of work $\mathcal{O}(n)$.

Proof: For total amount of work, letting $W(n)$ represent the total amount of work for input size n, notice that we have exactly one recursive call which induces work of $W(\lfloor n/2 \rfloor)$ and some fixed linear work of $\mathcal{O}(n)$. Hence,

$$W(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ W(\lfloor n/2 \rfloor) + \mathcal{O}(n) & \text{otherwise} \end{cases}$$

and consequently $W(n) \in \mathcal{O}(n)$. □