1 Overview

In the last lecture, we wrote a non-recursive summation program and showed that analysis of the runtime was more intuitive than with the recursive summation program written previously. We proved a theorem stating that a \( p \)-processor CRCW PRAM algorithm that runs in time \( t_p(n) \), can be implemented on a \( p \)-processor EREW PRAM in time \( t'_p(n) = O(t_p(n) \cdot \log p) \).

In this lecture, we introduce the parallel prefix-sums algorithm. We start with an overview of an iterative version of prefix-sums. We then present a parallel prefix-sum algorithm, followed by an inductive proof to show that this algorithm correctly computes prefix-sums.

2 Prefix-Sums Algorithm

Given an array \( A \), with \( n \) elements, the prefix-sums algorithm returns an array \( B \), such that:

\[
B[i] = \sum_{k=1}^{i} A[k] \tag{1}
\]

In other words, each element of \( B \) in position \( i \), is equal to the sum of all elements from 0 to \( i \) in \( A \). So, if we have the following array as input, \( A = [1, 5, 3] \), then,

\[
\begin{align*}
B[0] &= A[0] = 1, \\
\end{align*}
\]

Our resulting array is \( B = [1, 6, 9] \). We can see then, that the following algorithm iteratively computes prefix-sums.

**Algorithm 1** \textsc{PrefixSums}(\( A[0 \ldots (n-1)] \))

\[
\begin{align*}
B[0] &= A[0] \\
\text{for } i = 1 \text{ to } n-1 \text{ do} \\
\text{return } B[0 \ldots n]
\end{align*}
\]

This algorithm iterates through the arrays, summing each element and storing the new sum in its proper position in \( B \) along the way. We can see that this algorithm runs in linear time. Although our iterative algorithm takes input from array \( A \) and returns the prefix-sums in array \( B \), it is also possible to write the prefix-sums directly into \( A \), creating an in-place prefix-sum algorithm without the use of the additional array \( B \) (see Algorithm 2).
Algorithm 2 InPlacePrefixSums(A[0 . . . (n − 1)])

for i = 1 to n − 1 do
return

2.1 Parallel Prefix-Sums

Prefix-sums inherently seems like an iterative problem since each entry depends upon those that come before it. However, we want to be able to perform this task in parallel. We’ll claim that the following in-place algorithm computes prefix-sums in parallel. For the purpose of analysis, we assume that the input array has a size of 2n + 1 and has index positions i, for −n ≤ i < n, where elements in positions i < 0 are equal to zero.

Algorithm 3 ParallelPrefixSums(A[0 . . . (n − 1)])

for j = 0 to ⌈log n − 1⌉ do
    for i = 2^j to n − 1 in parallel do

We can see that the runtime for the inner for loop will be constant time since the operation is done by all processors in parallel. The outer for loop runs from zero to log n − 1, and so, requires log n time to execute. This tells us that our runtime is T(n) = O(log n).

To calculate work, we need to include the steps done in the inner for loop. We know that i goes from 2^j to n − 1 while the outer for loop goes from j = 0 to log n. So we can describe the work as follows:

\[
W(n) = \sum_{j=0}^{\log n - 1} \sum_{i=2^j}^{n-1} 1
= \sum_{j=0}^{\log n - 1} (n - 2^j)
= \sum_{j=0}^{\log n - 1} n - \sum_{j=0}^{\log n - 1} 2^j
\]

First, let us prove the following:

Claim 1. \(\sum_{j=0}^{k-1} 2^j = 2^k - 1\)

Proof. We’ll prove this using induction.

Base case: k = 1.

\[
\sum_{j=0}^{1-1} 2^j = \sum_{j=0}^{0} 2^j = 1
\]
Inductive step: Assume the claim holds for \( k > 1 \) and let us prove it for \( k' = k + 1 \).

\[
\sum_{j=0}^{k'-1} 2^j = \sum_{j=0}^{k} 2^j = \left( \sum_{j=0}^{k-1} 2^j \right) + 2^k = 2^k - 1 + 2^k = 2 \cdot 2^k - 1 = 2^{k+1} - 1 = 2^{k'} - 1
\]

Thus, we can say that \( \sum_{j=0}^{\log n - 1} 2^j = 2^{\log n} - 1 = n - 1 \). Another way to understand this is by picturing a full binary tree. If a binary tree has \( n \) leaves, then the numbers we are summing are the amount of internal nodes of the tree. The number of internal nodes of a binary tree is equal to one less than the number of leaves. So in our example, the number of internal nodes is \( n - 1 \). In Figure 1 below, the subtree of internal nodes is colored in red. Notice that the level right above the leaves has \( \frac{n}{2} \) nodes. This is similar to our sum equation ending at \( \frac{n}{2} \).

![Figure 1: Binary Tree and subtree](image)

Now that we’ve figured out what the term \( \sum_{j=0}^{\log n - 1} 2^j \) equals, we can continue our analysis of work done for the parallel prefix-sum algorithm, starting where we left off.

\[
W(n) = n \log n - \sum_{j=0}^{\log n - 1} 2^j = n \log n - (n - 1) = \Theta(n \log n)
\]

So work is equal to \( \Theta(n \log n) \).
2.2 Proof of correctness of Parallel Prefix-Sum

We want to prove that the Parallel Prefix-Sums algorithm actually does what it’s supposed to do, which is to calculate the correct prefix-sums and store them back into the array in the appropriate elements. First, let’s begin by understanding what is happening in the algorithm.

**Algorithm 4 ParallelPrefixSums(A[0 . . . (n − 1)])**

```plaintext
for j = 0 to ⌈log n − 1⌉ do
    for i = 2^j to n − 1 in parallel do
```

Let’s say that we have the input array A, with size, n = 14, and elements initialized to one (See Figure 2). In the first iteration, j = 0, and the elements with indices i, from 2^j = 2^0 = 1 to n − 1, get the sum of its own value and that of the element to its left. On the next iteration, j = 1, and now the elements from 2 ≤ i ≤ n − 1, get the sum of its own value with that of the element 2 indices away on the left. We continue this until j = ⌈log n⌉ − 1. In our case, this is when j = 3. See Figure 2 for a visualization of the what is happening in this algorithm.

![Figure 2: Parallel Prefix Sum](image)

We can now look at the algorithm in steps, as j is incremented. Notice that after step j completes, the first 2^j+1 elements of array A are set to correct prefix-sums values. In other words, before step j occurs, the first 2^j elements of array A are correct prefix-sums values. After step 0, when j = 0, has occurred, the first two (2^0+1 = 2) elements are correct prefix-sums values. We see this in Figure 2, where elements A[0] = 1 and A[1] = 2. Notice also, that each of the rest of the elements
is set to the sum of $2^{j+1}$ preceding elements, including itself, after step $j$ has completed. In our example for step 0, we see that the rest of the elements in $A[2 \ldots 12]$ are all set to the value 2, which is the sum of the two elements: the preceding one and itself. We will now use these observations to prove the correctness of the parallel prefix-sum algorithm using induction.

**Claim 2.** Before each round $j$ in the algorithm $A[i] = \sum_{k=i-(2^j-1)}^i A[k]$

We can run through the summation to get an understanding of what’s going on. Let’s use our example from Figure 2, but we’ll only concern ourselves with the first four elements. Remember, we stated earlier that the input array $A$, is actually of size $2n - 1$ with elements at indices less than zero, initialized to zero. We begin with $j = 0$. Then, $b = 2^0 - 1 = 0$. We can then see that for $i = 0$, $k = 0 - 0 = 0$. So the summation goes from zero to zero, and before round zero, $A[0] = A[0]$. We see that before round zero, this is the case for all $i$, so that $A[i] = A[0]$. This makes sense since no work has been done before round zero. The same can be done for additional levels of $j$ to verify that this summation makes sense. Table 1 below, shows this for $0 \leq i < 4$ and $j \leq 2$. Notice the general pattern of iterations emerging as $j$ increases, and see how it matches the steps in Figure 2.

<table>
<thead>
<tr>
<th>$j$</th>
<th>$2^j - 1$</th>
<th>$i$</th>
<th>$k$</th>
<th>$A[k]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$A[0] = A[0] = 1$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>$A[0] = A[-1] + A[0] = 1$</td>
</tr>
</tbody>
</table>

Now that we understand the claim that’s being made, we can begin the inductive proof for the parallel prefix-sum algorithm.

**Proof.** **Base case:** $j = 0$
Thus, we see that the base case holds for \( j = 0 \). Now, we do the inductive step. First, we assume that for some \( j \), the claim holds true so that before the \( j^{th} \) round, \( A[i] = \sum_{k=i-(2^j-1)}^{i} A[k] \). Then, by looking back at the algorithm, we look at the state of the array elements before round \( j' = j + 1 \). From the algorithm, we know that before round \( j' \),

\[
\]

Let \( i' = i - 2^{j'} \), then

\[
= \sum_{k=i-(2^{j'}-1)}^{i} A[k] + \sum_{k=i'-(2^{j}-1)}^{i'} A[k], \text{ by our inductive hypothesis}
\]

To prove our claim, we need to show that \( A[i] = \sum_{k=i-(2^{j'}-1)}^{i} A[k] \). We can distribute the negative in the first summation and replace \( i' \) with \( i - 2^{j'} \) in the second summation.

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

Now let’s take a look at these summations separately. The first summation \( A[i] = \sum_{k=i-2^{j'}+1}^{i} A[k] \) is equal to \( A[i - 2^{j'} + 1] + \ldots + A[i - 2] + A[i - 1] + A[i] \). What this means is that we’re summing consecutive elements in an array from positions \((i - 2^{j'} + 1)\) to \( i \). Now if we look at the second summation from the equation above, we see that \( \sum_{k=i-2^{j'}}^{i-2^{j}-1} A[k] \) is equal to \( A[i - 2^{j} - (2^{j} - 1)] + \ldots + A[i - 2^{j} - 1] + A[i - 2^{j}] \). This is also the sum of consecutive elements in the array. Notice that these two sequences are actually next to one another and form one long consecutive sequence (see figure 3).
The first summation term is the latter half of the consecutive sequence, while the second summation term is the former half. We see that each sequence is the sum of $2^j$ elements. We can easily check this by subtracting the last index from the first and adding one as we did earlier,

$$i - (i - 2^j + 1) + 1 = i - i + 2^j - 1 + 1 = 2^j$$

$$i - 2^j - (i - 2^j - (2^j - 1)) + 1 = i - 2^j - i + 2^j + 2^j - 1 + 1 = 2^j$$

Then, we can see that this sequence is summing $2 \cdot 2^j = 2^{j+1} = 2^{j'}$ elements. Now that we know the sequences are consecutive, we can say that

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

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

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

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

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

□
Thus, after round $j = \lceil \log n - 1 \rceil$, $A[i] = \sum_{k=i-(2^j-1)}^{i} A[k]$ for every index $0 \leq i \leq n - 1$, which is the definition of prefix sums.