Chapter 5

Parallel Prefix Algorithms

5.1 Introduction

In this chapter we discuss the *parallel-prefix* operation. The parallel-prefix operation is also called *scan* operation and we will use these terms interchangeably. The reduce operation applies an *associative* operation, such as the sum, the product, the min or the max, to an array of size n to get an answer that corresponds to the sum, the product, the min or the max of the entire array. The parallel-prefix operation computes the answer for all prefixes of the array.

This chapter is organized as follows. Section 5.2 gives an algorithm for the *parallel-prefix* operation that requires $O(n \log n)$ work. Section 5.3 presents work-optimal algorithms for the parallel-prefix operation. It first gives a simple recursive parallel algorithm for the scan operation. It then presents an iterative version due to Blelloch [Ble90]. Section 5.4 gives an LLP version of the parallel-prefix operation. Finally, Section 5.5 presents some applications of the parallel-prefix operation.

5.2 A Parallel Prefix Sum algorithm with $O(n \log n)$ work

Suppose that we have an array A of size n. We want to compute C, the sum of all possible prefixes of A, formally defined as follows:

$$C[k] = \sum_{i=0}^{i \leq k} A[i]$$

For example, suppose $A = [1, 4, 9, 16]$. Then, $C = [1, 5, 14, 30]$. This operation called **parallel prefix** or **scan** operation is a fundamental building block for many parallel algorithms.

There is an easy linear time sequential algorithm for this problem.

Algorithm SeqPrefix: A Sequential Algorithm for Prefix-Sum

```
1 for (int $i = 0; i < n; i++$)
2     $C[i] := A[i]$
3 for (int $i = 1; i < n; i++$)
4     $C[i] := C[i-1] + A[i]$
```

This algorithm taken $O(n)$ time and requires $O(n)$ work.

The following parallel algorithm reduces the overall time by computing the sum in a binary tree like fashion rather than one at a time as done by the sequential algorithm.

Algorithm ParPrefix: A Parallel Algorithm for Prefix-Sum with $O(n \log n)$ work

```
forall $i \in [0, n-1]$: in parallel do
    $C[i] := A[i]$;
for (int $d = 1; d < n; d = 2d$)
    forall $i \in [1, n-1]$: in parallel do
        if $(i - d \geq 0)$
            $C[i] := C[i] + C[i-d]$;
```

After every iteration of the second parallel for loop, $C[i]$ holds sum of all $A[k]$ such that $i - 2d + 1 \leq k \leq i$ for any d. Table 5.1 shows values of C array after various values of d.

A	3	2	4	2	1	3	5	2
C, $d = 1$	3	**5**	**6**	**6**	**3**	**4**	**8**	**7**
C, $d = 2$	3	5	**9**	**11**	**9**	**10**	**11**	**11**
C, $d = 4$	3	5	9	11	**12**	**15**	**20**	**22**

Table 5.1: Entries during Execution of the Parallel Prefix Algorithm

One needs to be careful in implementing the above high-level algorithm. There needs to be proper synchronization of threads for reading and writing in-place. For example, the pseudo-code for implementing parallel-prefix on a GPU is as follows. Observe that the *for* loop with index i is not in the pseudo-code. This is because the code is executed by all processes P_i.

Algorithm ParPrefixGPU: A Parallel Prefix Algorithm on GPU with $O(n \log n)$ work

```
$P_i$::
$C[i] := A[i]$;
for (int $d = 1; d < n; d = d * 2$) // in sequence
    if $(i \geq d)$ $val = C[i-d]$;
    barrier();
    if $(i \geq d)$ $C[i] = C[i] + val$;
    barrier();
endfor;
```

This algorithm requires $O(\log n)$ time but is not work-optimal. It requires $O(n \log n)$ work whereas the sequential algorithm requires only $O(n)$ work. We now present a work-optimal algorithm for this problem.

5.3 A Work-Optimal Parallel Prefix Sum Algorithm

We now show an algorithm that computes parallel prefix with $O(n)$ work. We give two versions of this algorithm. First, we give a recursive version. We will use arrays indexed starting from 1 and assume that the input array has size n that is a power of 2, i.e., let $n = 2^k$ for some integral k. An example of the recursive algorithm Recursive-ParPrefix is shown in Table 5.2. We first take care of the base case when n equals 1. Otherwise, we copy the array A into B such that the array B has the size that is half of the size of A and every two elements of A are added to one element of B. We make the recursive call to parallel prefix of B, getting the result in array D. Once we have recursive solution in D, we can compute the solution in C by doing a case analysis on index i.

Algorithm Recursive-ParPrefix: A Recursive Parallel Prefix Algorithm with $O(n)$ work

```
1 if $(n = 1)$ then $C[1] = A[1]$ else
2     for $i := 1$ to $n/2$ in parallel do
3         $B[i] := A[2i - 1] + A[2i]$;
4     $D :=$ recursive-ParPrefix$(B[1], B[2], ..., B[n/2])$;
5     for $i := 1$ to $n$ in parallel do
6         if $(i = 1)$ then $C[1] := A[1]$;
7         else if $even(i)$ then $C[i] := D[i/2]$
8         else if $odd(i)$ and $(i > 1)$ then $C[i] := D[i/2] + A[i]$
```

A	1	2	3	4	5	6	7	8
B	3	7	11	15				
D	**3**	**10**	**21**	**36**				
C	1	**3**	6	**10**	15	**21**	28	**36**

Table 5.2: Example of a Recursive Scan

.

Let us compute the time required by recursive parallel-prefix. In one parallel time step, the input size is reduced by a factor of 2. Hence, the algorithm takes $O(\log n)$ time. To compute the total work required by the above algorithm, we use the following recurrence relations.

$$W(n) = W(n/2) + O(n).$$

$$W(1) = O(1).$$

Therefore, $W(n) = O(n)$.

We now show a non-recursive version of the algorithm which computes the *exclusive* scan of an array. The inclusive scan of an array A returns for every index i, the sum of all entries before i including i, whereas the exclusive scan returns the sum of all entries in prefix excluding itself. For example, Table 5.3 shows a simple inclusive and exclusive scan of an array A.

Going from exclusive scan to inclusive parallel prefix sum can be performed in one parallel step.

The work-optimal algorithm can be viewed in a tree like fashion except that we require two sweeps of the tree — one in the upward direction and one in the downward direction.

An example is shown below. Suppose that the array A equals $[01, 02, 03, 04, 05, 06, 07, 08]$. For simplicity, we have used $A[i] = i$. Now, the upward sweep is shown in Fig. 5.1.

The upward sweep can be understood in terms of the tree as follows. For any vertex v, we compute:

$$sum[v] = sum[L[v]] + sum[R[v]]$$

Input Array A	01 02 03 04 05 06 07 08
Inclusive Scan of A	01 03 06 10 15 21 28 36
Exclusive Scan of A	00 01 03 06 10 15 21 28

Table 5.3: Example of an Inclusive and Exclusive Scan

.

Algorithm Blelloch-Scan: Blelloch's Parallel Prefix Algorithm with $O(n)$ work

```
forall $i$ in $[0, n-1]$: in parallel do
    $B[i] := A[i]$;
for $h := 0$ to $\log n - 1$ do // upward sweep: reduce operation
    forall $i$ from 0 to $n-1$ in steps of $2^{h+1}$ in parallel do
        $B[i+2^{h+1}-1] := B[i+2^h-1] + B[i+2^{h+1}-1]$;
$B[n-1] := 0$;
for $h := \log n - 1$ down to 0 do // downward sweep
    forall $i$ from 0 to $n-1$ in steps of $2^{h+1}$ in parallel do
        $LeftValue = B[i+2^h-1]$; // store the value of the left child
        $B[i+2^h-1] = B[i+2^{h+1}-1]$; // Left child gets the value of this node
        $B[i+2^{h+1}-1] = LeftValue + B[i+2^{h+1}-1]$; // Right child gets the sum
```

```
01  02  03  04  05  06  07  08
    03      07      11      15
            10              26
                            36
```

Figure 5.1: Upward Sweep of Blelloch Scan

where $L[v]$ and $R[v]$ are the left and right children of v.

We now replace the root by 0 and start the downward sweep. The downward sweep can be understood in terms of the tree as follows.

$$scan[L[v]] = scan[v]$$

$$scan[R[v]] = sum[L[v]] + scan[v]$$

The upward sweep followed by the downward sweep is presented in Fig. 5.2.

Figure 5.3 shows the values computed using upward scan and the downward scan in a binary tree. The upward scan values are shown inside circles and the downward scan values are shown outside circles.

We now show the correctness of the above procedure. We say that vertex x *precedes* vertex y if x appears before y in the preorder traversal of the tree.

```
01  02  03  04  05  06  07  08
    03      07      11      15
            10              26
                            36
                            00
            00              10
     00     03      10      21
 00  01   03 06  10  15  21  28
```

Figure 5.2: Blelloch Scan

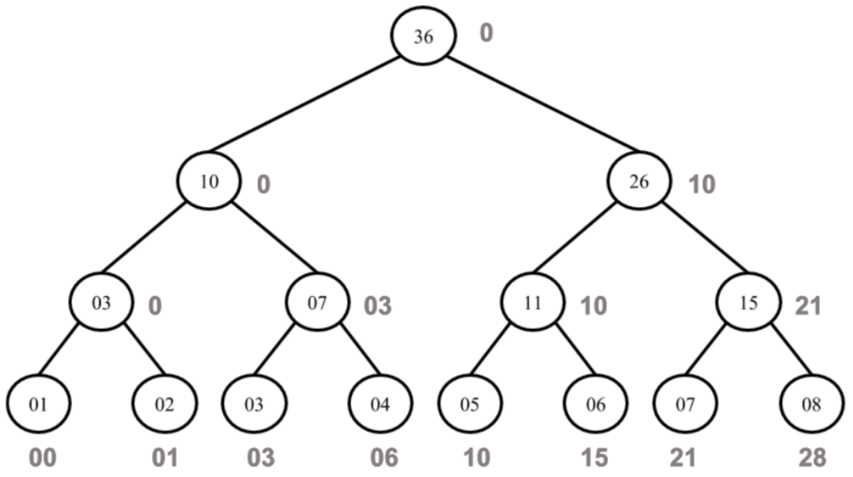

Figure 5.3: Blelloch Scan Algorithm applied to Parallel Prefix Puzzle

Theorem 5.1 *After a complete down-sweep, each vertex of the tree contains the sum of all the leaf values that precede it.*

Proof: The proof is inductive from the root: we show that if a parent has the correct sum, both children must have the correct sum. The root has no elements preceding it, so its value is the identity element which is zero for sum.

The left child of any vertex has exactly the same leaves preceding it as the vertex itself. This is because the preorder traversal always visits the left child of a vertex immediately after the vertex. By the induction hypothesis, the parent has the correct sum, so it need only copy this sum to the left child. The right child of any vertex has two sets of leaves preceding it, the leaves preceding the parent, and the leaves at or below the left child. Therefore, by adding the parent's down-sweep value, which is correct by the induction hypothesis, and the left-child's up-sweep value, the right-child will contain the sum of all the leaves preceding it.

■

5.4 LLP based Parallel Prefix-Sum

Let A be an array of n numbers. Assume that n is a power of two for simplicity. We would like to compute parallel (exclusive) prefix sum of A. We compute the parallel prefix sum in the array G using ideas in Blelloch scan. Our LLP based implementation will use additional space but has less synchronization. Let the input array be A of size n. When n is a power of 2, the summation tree of the reduce operation has $n-1$ nodes. We create an additional array S of size $n-1$ that simulates the upward scan using Algorithm LLP-Reduce. We now use another LLP algorithm to simulate the downward scan. We model the downward scan using an array G of size $2n-1$ as follows. The first $n-1$ elements of the array compute the scan values for the intermediate nodes. The last n entries give us the required values for exclusive scan. Similar to LLP Algorithm for Reduce, we use the indexing scheme of a perfect binary tree such that for any node with index i, its left child is given by $2i$, its right child by $2i+1$, and its parent by $i/2$ (for non-root nodes).

The LLP algorithm initializes every element of G to $-\infty$. Each element $G[j]$ is updated exactly once when the forbidden predicate is true for index j. Since $G[1]$ corresponds to the root of the tree for the downward scan, we set $G[1]$ to 0 as required by the first ensure clause of the LLP. The left child of any node is even and is updated by the second ensure clause by copying the value of the parent. All the left children are characterized by their indices being even and thus setting $G[j]$ to $G[j/2]$ copies the value of the parent. The right child of any node is updated by the third and the fourth ensure clauses as the sum of the parent value and the S value for its left sibling. The fourth clause is used for the last level of the tree when the sum of the subtree at that node is simply equal to A value.

Algorithm LLPScan: A Parallel LLP Algorithm for Scan

```
1  $P_j$: Code for thread $j$
2  // Assume $n$ is a power of 2 for simplicity
3  input: $A$: array[1..$n$] of int
4        $S$: array[1..$n-1$] of int // summation tree
5  output:$G$:array[1..$2n-1$] of int such that $G[i] = \sum_1^{i-n} A[i-n]$ for $i \geq n$.
6  init: $G[j] := -\infty$
7  ensure: $G[j] \geq 0$ if $j = 1$
8  ensure: $G[j] \geq G[j/2]$ if $j$ is even
9  ensure: $G[j] \geq S[j-1] + G[j/2]$ if $j$ is odd and $j < n$
10 ensure: $G[j] \geq A[j-n] + G[j/2]$ if $j$ is odd and $j > n$
```

5.5 Applications

Although our description of the parallel prefix operation was based on the sum operator, it is applicable to any *associative* operation such as the product, min, or max. Many problems on arrays can be solved using the parallel prefix operation one or more times. We give some examples below.

- *Array Compaction*: Suppose we are given an array A of integers. Our problem is to create another array B consisting of only those elements that specify certain given *filter* predicate. As a simple example, we may want the array B to contain only those entries in A that are divisible by 5. If for every entry in A that is divisible by 5, we could determine the number of entries that appear in A before this entry that are also divisible by 5, we are done. We can store that entry at the right index in B. To solve this problem, In $O(1)$ time we create an additional array C such that $C[j]$ equals 1 if $A[j]$ satisfies the filter predicate and 0 otherwise. The parallel prefix of the array C gives us the index where the entries of A should go in the array B.

- *Sparse Dot Product*: Suppose that we have a vector A of size n such that most entries are zero. Instead of storing the entire vector, we can store a *sparse* vector B such that $B[i]$ corresponds to i^{th} nonzero entry and it stores $(j, A[j])$ where j is the index of i^{th} nonzero entry. Suppose that we want to compute the dot product of A with a dense vector v. With n processors, we can easily compute the dot product in $O(\log n)$ time. If we had a sparse matrix A, and we wanted to computes Av, then by computing dot product of different rows of A with v in parallel, we can compute Av also in $O(\log n)$ time.

- *Maximum Subsequence Problem*: Suppose that you are given a nonempty array A of integers of size n. The array consists of both positive and negative integers. We give an algorithm that gives a nonempty subsequence in the array with the largest value. Formally, we are required to find

  $$max_{i,j \geq i} \sum_{k=i}^{j} A[k]$$

  The algorithm is:

  - First reduce-max and return this if the output is negative (this handles the edge case when all elements are negative).

– Compute parallel prefix sum, and call the resulting array B.

– Compute parallel postfix max on B (this can be done with Blelloch scan on a reversed B and call the resulting array C.

– Create a new array D, and for each index i in parallel write $D[i] := C[i] - B[i] + A[i]$.

– Finally output the result from reduce-max on D.

This procedure has time complexity $O(\log n)$ and work complexity $O(n)$ in the CREW PRAM model.

Below is the psuedo-code:

Algorithm Maximum-Subsequence: Parallel Maximum Subsequence

```
1 $m :=$ reduce_max($A$)
2 if $m < 0$ then output m
3 $B :=$ prefix_sum($A$)
4 $C :=$ postfix_max($B$)
5 $D :=$ allocate_new_array($n$)
6 foreach $i \in \{1, 2, ..., n\}$ in parallel do
7     $D[i] := C[i] - B[i] + A[i]$
8 end
9 output reduce_max($D$)
```

5.6 Summary

The following table lists all the algorithms discussed in this chapter.

Problem	Algorithm	Parallel Time	Work
Parallel Prefix	Hillis-Steele	$O(\log n)$	$O(n \log n)$
Parallel Prefix	Recursive	$O(\log n)$	$O(n)$
Parallel Prefix	Blelloch	$O(\log n)$	$O(n)$
Parallel Prefix	LLP	$O(\log n)$	$O(n)$

5.7 Problems

1. Suppose that we have a mesh of size $\sqrt{n}$ rows and $\sqrt{n}$ columns. An array of size n is distributed on this mesh in a row-major form. Design an algorithm such that every node has the parallel prefix at the end of the algorithm. Your algorithm should not take more than $O(\sqrt{n})$ time and $O(n^{3/2})$ cost.

2. Consider a mesh of size $n^{1/3}$ rows and $n^{1/3}$ columns. An array of size n is distributed on this mesh in a row-major form such that each node has $n^{1/3}$ elements. Give a work optimal algorithm that takes $O(n^{1/3})$ time.

3. Give an algorithm to compute prefix-sum on a hypercube of n nodes with $O(\log n)$ time complexity. What is the work complexity of your algorithm?

4. Give a parallel-prefix algorithm to divide the array into two parts: the part with odd entries followed by the part with even entries.

5.8 Bibliographic Remarks

Parallel Prefix algorithms are presented in most books on parallel computation (e.g. [JáJ92]). The reader will find many applications of the parallel prefix computation in [Ble90].