Chapter 4

Parallel Reduce Algorithms

4.1 Introduction

In this chapter we illustrate some basic ideas in designing parallel algorithms by computing the maximum of an array of size n. These ideas would also be applicable to other operators such as computing the *sum*, or the *product* of an array. Similarly, we can compute the *OR*, and the *AND* of a boolean array using these ideas.

A sequential algorithm can solve this problem in $O(n)$ time and $O(n)$ work. We now show various parallel algorithms to solve the same problem. For simplicity, we assume that n is a power of 2, i.e., $n = 2^k$. We also assume that all elements in the array are distinct. Exercise 2 requires you to extend these algorithms when entries may not be distinct.

In Section 4.2, we first present a simple algorithm called ParReduce that takes $O(\log n)$ time and $O(n)$ work on an EREW PRAM. Section 4.3 describes a technique to combine the sequential algorithm with a parallel algorithm to reduce the total number of processors. Section 4.4 gives the LLP algorithm for the reduce operation. The LLP algorithm does not have the synchronization requirements of the Algorithm ParReduce. In Section 4.5, we consider a CRCW PRAM and present an algorithm that takes $O(\log \log n)$ time and $O(n)$ work.

4.2 Reduce

Suppose that we have an array A of size n and we want to compute the sum of all numbers. A simple *for* loop achieves this in $O(n)$ time. We now show a parallel algorithm that computes the sum in $O(\log n)$ time assuming that we have n cores. The algorithm is best viewed as constructing a binary tree such that leaves of the tree consists of the elements of the array and the internal nodes correspond to the sum of its children. Fig. 4.1 shows such a tree. Clearly, the root of the tree has the sum of the entire array. Since the tree is binary, the height of the tree is $O(\log n)$. Therefore, if the parallel algorithm can compute a level of the tree in parallel in $O(1)$ time, then the entire algorithm also has the time complexity of $O(\log n)$.

Algorithm ParReduce uses h to indicate the level of the tree. In the first iteration h equals 1 and the partial sums are stored in the first half of the array A. The entry $A[i]$ stores the sum of entries $A[2i-1]$ and $A[2i]$. In the second iteration, we reduce the size of the array that stores partial sum by an additional factor of 2. After $\log n$ iterations, the total sum is stored in $A[1]$. We note here that in the description of the algorithm we have the step $A[i] := A[2i-1] + A[2i]$ which is executed by all eligible i in parallel. This step may entail both reads and writes to the same memory location. For all such steps in the parallel algorithms, we follow the interpretation that all processes first do all the **reads** of the shared memory (into the private registers of the processors). When all reads are finished, then all writes are performed.

Table 4.1 shows values of A array after various values of h.

Algorithm ParReduce: A Parallel Algorithm for Reduce

```
1 input: A: array[1..n] of int // Assume n is a power of 2 for simplicity
2 for h := 1 to log n do
3       forall i in parallel do
4             if (i ≤ n/2^h) then
5                   A[i] := A[2i − 1] + A[2i]
6 return A[1]
```

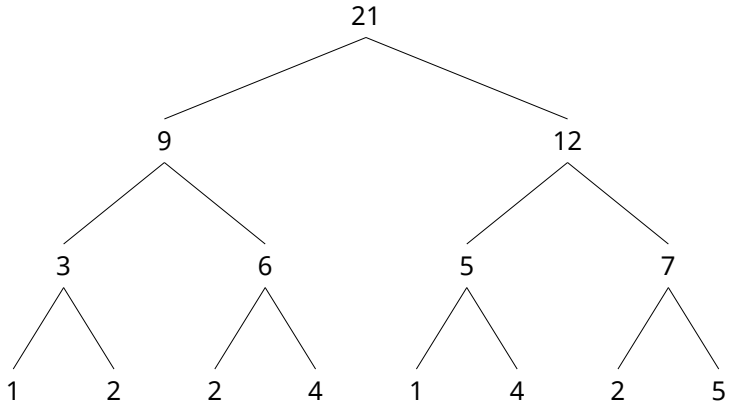

Figure 4.1: A Summation Tree

i	1	2	3	4	5	6	7	8
A	1	2	2	4	1	4	2	5
$h=1$	**3**	**6**	**5**	**7**	1	4	2	5
$h=2$	**9**	**12**	5	7	1	4	2	5
$h=3$	**21**	12	5	7	1	4	2	5

Table 4.1: Entries during Execution of the Parallel Reduce Algorithm

The total time taken by this algorithm is $O(\log n)$ and the total number of comparisons made by this algorithm is $O(n)$. Ignoring for now the question of how various threads determine which comparison to make at what time step, we see that the total amount of work done by all threads is $O(n)$.

Whenever we design parallel algorithms, we should check if we require concurrent reads or concurrent writes by multiple processors in any step. Assuming that each of the processor knows when to read the labels of its children and when to write its own label, there is no conflict in computation and the algorithm can be implemented on a EREW PRAM.

It is also important to consider the *cost* of a PRAM algorithm. The cost of an algorithm is simply the product of the number of processors used by the algorithm and its time complexity. In this case, we are using n processors; therefore, the cost of this algorithm is $O(n \log n)$.

4.3 Accelerated Cascading Technique

We now show a technique that combines two different algorithms to reduce the cost (and sometimes the work) of many PRAM algorithms. Instead of applying the binary tree algorithm on the entire array of size n, we first divide the array into $O(n/\log n)$ blocks of $O(\log n)$. Also, we use only $O(n/\log n)$ processors so that each block can be assigned to a single processor. Now, each processor can compute the maximum of its block using a sequential algorithm. Since this step can be performed in parallel for different blocks, this step takes $O(\log n)$ time with the total cost of $O(n/\log n * n) = O(n)$. Now we have $O(n/\log n)$ numbers and we need to compute the maximum of these numbers. At this point, we apply the binary tree based algorithm. It will take us $O(\log n)$ time with $O(n/\log n)$ processors. Hence, the total cost of the second step would only be $O(n)$. Combining the time, the work and the cost of each of the steps, we get $T(n) = O(\log n)$, $W(n) = O(n)$, and $C(n) = O(n)$.

It can be shown that $O(\log n)$ time is required on EREW PRAM. Also, since the work matches that of a sequential algorithm, we have an optimal work-time algorithm on EREW PRAM.

4.4 LLP-Reduce Algorithm

The parallel reduce algorithm has two disadvantages: it destroys the original input array and it requires that all threads synchronize after each iteration of the outer for loop. We can take care of the first problem by making a copy of the original array. This step can be carried out in $O(1)$ time with the number of cores equal to the size of the array. We now show a LLP based algorithm to compute the sum that takes care of both of these problems. We again assume that the array size of A is a power of 2 for simplicity. We would like to compute an array G of size $n-1$ such that $G[1]$ contains the sum of the entire array, $G[2]$ contains the sum of the first half of the array, $G[3]$ contains the sum of the second half of the array, and so on.

Fig. 4.2 shows the arrays G and A for n equal to 8. Note that for $1 \leq j < n/2$, the children of node $G[j]$ are the nodes $G[2j]$ and $G[2j+1]$. For $n/2 \leq j < n$, the children of nodes $G[j]$ are the nodes $A[2j-n+1]$ and $A[2j-n+2]$. Then, the least vector G that satisfies

$$B_{sum}(G) \equiv (\forall j : 1 \leq j < n/2 : G[j] \geq G[2j] + G[2j+1]) \wedge (\forall j : n/2 \leq j \leq n-1 : G[j] \geq A[2j-n+1] + A[2j-n+2])$$

is equal to the recursive sum. The array $G[1..n/2-1]$ contains the sum of the left and the right children which are in G itself. The array range $G[n/2..n-1]$ contains the sum of elements of A viewed as the leaves of the tree.

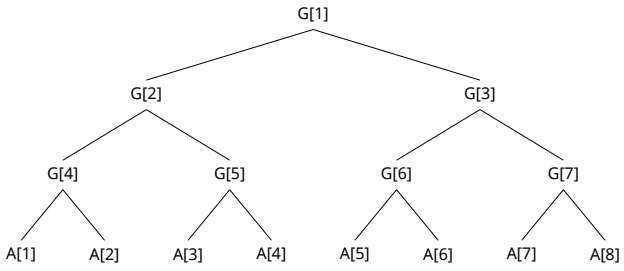

Figure 4.2: A Summation Tree for LLP Reduce

The parallel algorithm LLP computes G in $O(\log n)$ time. Instead of using $+$, we can use any other associative operator such as *min* or *max*.

Algorithm LLP-Reduce: A Parallel LLP Algorithm for Reduce

```
1 P_j:
2 input: A: array[1..n] of int // Assume n ≥ 2 is a power of 2 for simplicity
3 shared var: G: array[1..n − 1] of int
4 init: G[j] := −∞
5 ensure: G[j] ≥ G[2j] + G[2j + 1] if 1 ≤ j < n/2
6 ensure: G[j] ≥ A[2j − n + 1] + A[2j − n + 2] if n/2 ≤ j < n
```

Table 4.2 shows values of the array G after various values of t, the iteration number. For simplicity, we consider a maximally parallel execution in which any entry that can change does so at any iteration. However, the final answer remains the same even if the execution is not maximally parallel. There is no synchronization required for this algorithm.

i	1	2	3	4	5	6	7
G	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$t = 1$	$-\infty$	$-\infty$	$-\infty$	3	6	5	7
$t = 2$	$-\infty$	9	12	3	6	5	7
$t = 3$	21	9	12	3	6	5	7

Table 4.2: Entries during a maximally Parallel Execution of the LLP-Reduce Algorithm

Although our discussion was for the sum, it is clear that any associative operation such as the product, the min, the max, or the boolean operator such as the AND, the OR, or the XOR can be used.

4.5 A Fast Parallel Common CRCW Algorithm

We now give a faster algorithm on common CRCW PRAM. This algorithm requires $O(n^2)$ processors. We have a boolean array $isBiggest$ of size n which is initialized to true for all i. We use one processor for every distinct i and

j which allows us to compare every distinct $A[i]$ and $A[j]$ in one time step. The processor assigned to (i, j) writes *false* on $isBiggest[i]$ if the entry for $A[j]$ is bigger than $A[i]$. Note that multiple writers may access $isBiggest[i]$ in one time step. However, all of them would be writing the common value *false* to that memory location. This step assumes common CRCW PRAM.

Algorithm FindingBiggest: A Common CRCW Parallel Algorithm for Finding the Maximum

```
1 var isBiggest: array[1..n] of boolean;
2 forall i in parallel do:
3       isBiggest[i] := true;
4 forall i, j: i ≠ j: in parallel do:
5       if (A[j] > A[i]) then isBiggest[i] := false;
6 forall i: in parallel do:
7       if isBiggest[i] then output (max = A[i]);
```

We have: $T(n) = O(1)$, and $W(n) = O(n^2)$.

How can we reduce the work complexity? We now show a technique that allows us to reduce the work complexity on a common CRCW machine. In binary trees we were comparing two labels using one processor and one time step. From the fast-parallel algorithm, we know that we can find the maximum of $\sqrt{n}$ nodes in $O(1)$ time using n processors. Can we use this fast parallel algorithm to increase the branching but reduce the height of the tree? For this section assume that $n = 2^k$ and $k = 2^m$. As a concrete example, let $n = 256 = 2^8$. Here $k = 8 = 2^3$. At the first level, we will have a branching of 16, i.e., the root would have 16 subtrees each of size 16. If we knew the maximum of each of the subtrees, then in $O(1)$ step we can compute the overall maximum in $O(1)$ steps. This step will require $16^2 = 256$ processors. Let us now determine the way to compute the maximum of the subtree with 16 labels. Each of these subtrees would have 4 subtrees of size 4. Using 16 processors, we can compute maximum of this tree. Since there are 16 subtrees, we need 16 times 16 processors at the second level. Since the height of the tree is $\log \log n$, the total number of time steps is $O(\log \log n)$. At each step, we use n processors. This gives us: $T(n) = O(\log \log n)$, and $W(n) = O(n \log \log n)$.

We now show that the previous algorithm that is not work optimal can be made work-optimal by the technique of "cascading" multiple algorithms. Note that we have an additional factor of $O(\log \log n)$ in the work. To eliminate this factor, we first partition our array of size n into $O(n/\log \log n)$ blocks each of size $O(\log \log n)$. If our array was only of size $O(n/\log \log n)$ and we apply the doubly logarithmic height tree algorithm, we would have the desired work complexity of $O(n)$. So now we consider how to compute the maximum of a block of $\log \log n$ numbers. Since this block is small in size, we can compute the maximum using a single processor in $O(\log \log n)$ time. Hence, the entire algorithm is as follows:

Step 1: run sequential algorithm on each of the blocks of size $O(\log \log n)$. This takes time equal to $O(\log \log n)$, and work equal to $O(n)$. We now have an array of size $n/\log \log n$.

Step 2: We now run the doubly logarithmic height tree algorithm. The total work is $O(n)$, and the total time is $O(\log \log n)$.

When we combine the time complexity of the work complexity of these two steps, we get the desired time complexity of $O(\log \log n)$ and the work complexity of $O(n)$.

4.6 Summary

Table 4.3 summarizes all the algorithms discussed in this chapter.

Algorithm	Work	Time	PRAM
Sequential	$O(n)$	$O(n)$	
Binary Tree Based	$O(n)$	$O(\log(n))$	EREW
All-pair Comparison Algorithm	$O(n^2)$	$O(1)$	CRCW
Doubly Logarithmic Tree Algorithm	$O(n \log(\log(n)))$	$O(\log(\log(n)))$	CRCW
Cascaded Algorithm	$O(n)$	$O(\log(\log(n)))$	CRCW

Table 4.3: Time Complexity and Work Complexity of Computing Maximum

4.7 Problems

1. Generalize the Algorithm LLP-Reduce for all values of n (i.e., n may not be a power of 2).
2. Suppose that an array does not have all elements that are distinct. Show how you can use any algorithm that assumes distinct elements for computing the maximum to solve the problem when elements are not distinct.
3. Give work-optimal algorithms to compute *OR* of a boolean array on EREW and common CRCW PRAM.
4. Suppose that instead of PRAM you have a network of processors connected by a two dimensional square mesh. Assuming that every hop on the network takes one unit of time, give an algorithm with $O(\sqrt{n})$ time complexity to compute the maximum in the network.
5. Repeat the previous problem with the topology of a d-dimensional hypercube (i.e. $n = 2^d$ for some integer d).

4.8 Bibliographic Remarks

Parallel algorithms for computing maximum or minimum of an array are presented in most books on parallel computation (e.g. [JáJ92]).