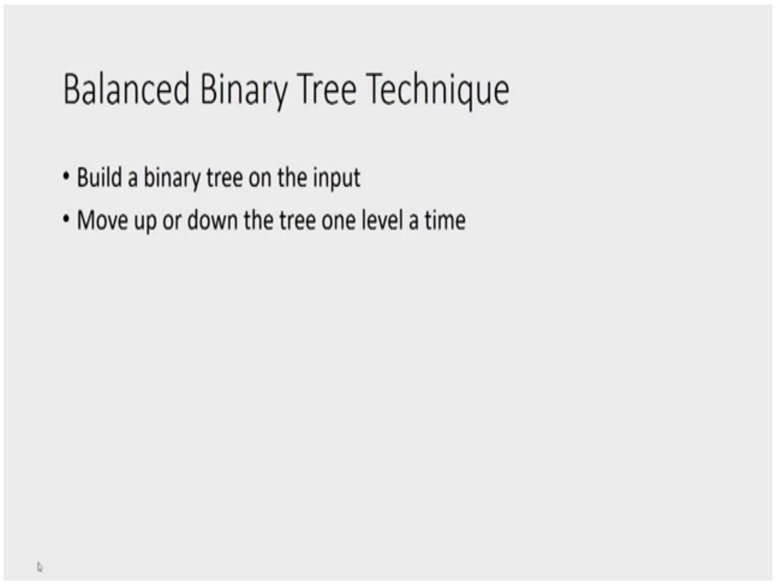
Parallel Algorithms

Prof. Sajith Gopalan
Department of Computer Science & Engineering
Indian Institute of Technology, Guwahati

Lecture – 05
Basic Techniques 1

Welcome to the 5th lecture of the NPTEL MOOC on Parallel Algorithms. Today we begin the study of some parallel algorithm design techniques.

(Refer Slide Time: 00:38)

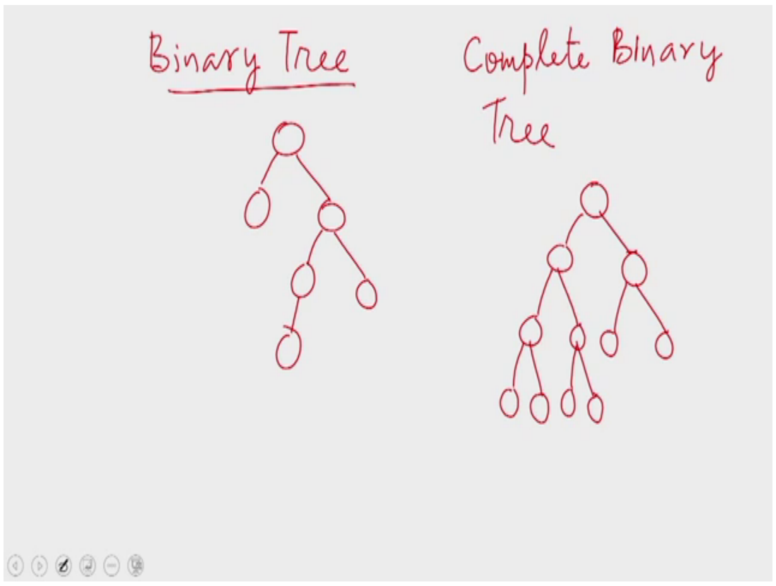


Balanced Binary Tree Technique

- Build a binary tree on the input
- Move up or down the tree one level a time

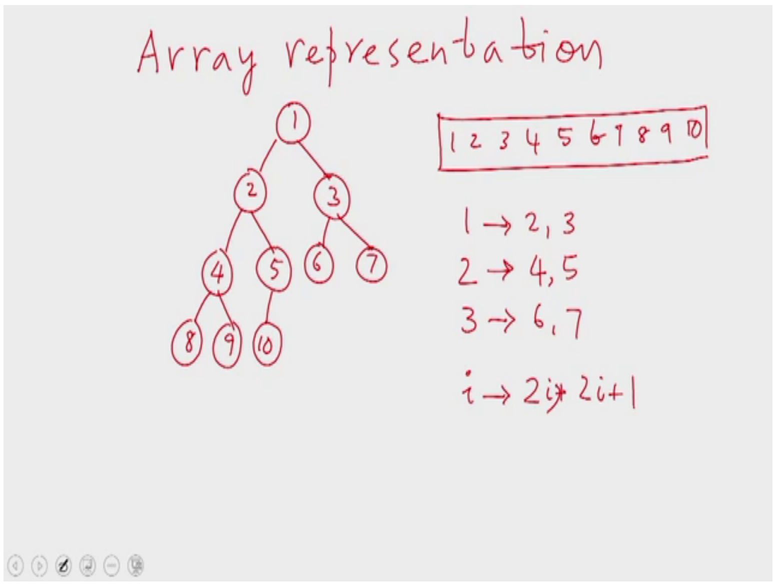
The first design technique that we shall see is the so called balanced binary tree technique. In this we build a binary tree on the given input and move up or down the tree one level at a time to achieve, what we want to achieve.

(Refer Slide Time: 00:56)



A binary tree is a tree in which every node has at most 2 children. In particular a complete binary tree it is like this, in this every level of the tree is full except possibly the last one. So, this is a complete binary tree. This tree has 4 levels: the first, second and third levels are full. A level is full if every possible node is present in present at that level and in the bottom most level all the percent nodes occupy the leftmost positions. These 4 nodes occupy the leftmost possible, 4 positions such a tree is called a complete binary tree.

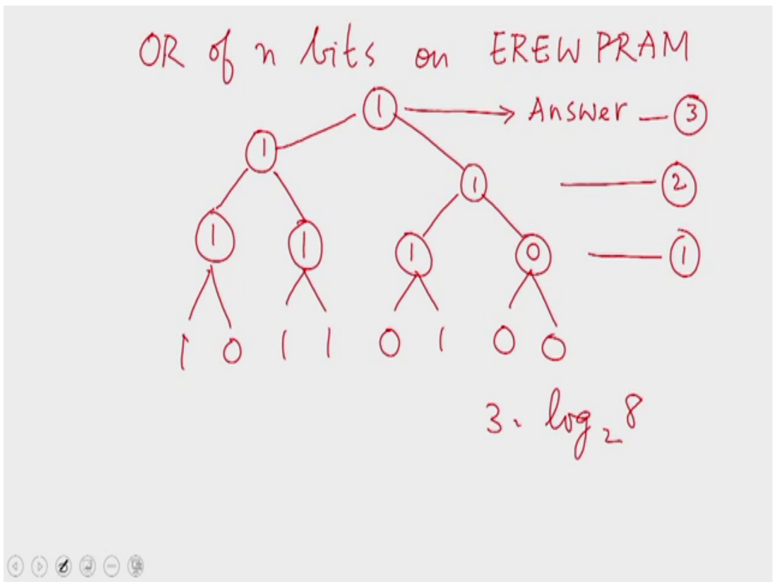
(Refer Slide Time: 02:17)



A complete binary tree can be represented using an array; you must be familiar with this array representation of binary trees from heapsort. Example, if you have a complete binary tree with nodes numbered in this fashion with 10 nodes. These 10 nodes we can assume occupy consecutive positions, in an array of size 10. Here we find that the children of node 1 are nodes 2 and 3, the children of node 2 are 4 and 5, the children of node 3 are 6 and 7.

In general, for node i the children are 2 i plus 2 i and 2 i plus 1, the children of node i are 2 i and 2 i plus 1, the left child of node i is 2 i and the right child of node i is 2 i plus 1. So, this is the array representation of a binary tree. So, given an input we can assume that there is an array representation on top of that input.

(Refer Slide Time: 04:03)

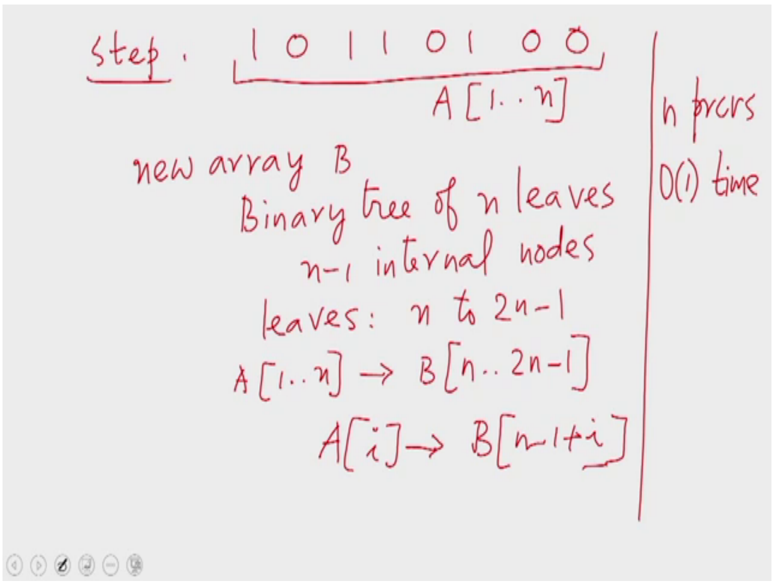


So, using this array representation we can find the OR of n bits on an EREW PRAM. We have already seen how to find the OR of n bits on the common CRCW PRAM as well as the tolerance CRCW PRAM. Both of those algorithms ran in order one time, but on an EREW PRAM it is not possible to find the OR of n bits in order of one time. The algorithm proceeds in this fashion, given an array of size n here we have taken an array of size 8 for example. So, we assume that n is a power of 2 for convenience. We construct a binary tree on top of this array; the OR of the given array is calculated in this fashion.

The OR of the first 2 bits is 1, the OR of the next 2 bits is 1, the OR of 0 and 1 is 1 again and the OR of 0 and 0 is 0. So, we have now reduced the array of size 8 to an array of size 4, the OR of these 4 bits will form the OR of the given 8 bits. So, the problem size has been reduced by a factor of 2 and then we recurs; the OR of these 2 bits is 1 and the OR of these 2 bits is 1 again.

Now, we have reduced the array size to 2 and the OR of these 2 bits is 1 again which happens to be the answer. So, you see that given an array of size 8 the OR of these 8 bits can be found in 3 steps. This is the 1st step, this is the 2nd step and the answer is formed in the 3rd step. And, 3 happens to be the logarithm of 8 to the base 2. So, the algorithm involves log n steps.

(Refer Slide Time: 06:34)



Step.

$\underbrace{1\ 0\ 1\ 1\ 0\ 1\ 0\ 0}_{A[1..n]}$

new array B

Binary tree of n leaves

n-1 internal nodes

leaves: n to 2n-1

$A[1..n] \rightarrow B[n..2n-1]$

$A[i] \rightarrow B[n-1+i]$

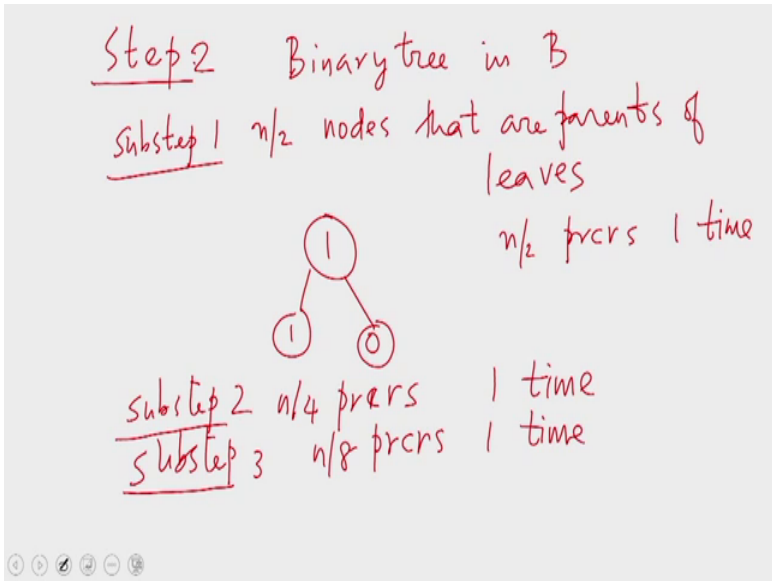
n procrs

O(1) time

So, now let us look at the details of the algorithm. In the 1st step given the array, what we do is to copy this array. Suppose, this array is in locations 1 to n of A; we copy these elements into a new array B. B is going to contain a binary tree of n leaves. A binary tree of n leaves contains 2 n minus 1 nodes, there are n leaves and n internal n minus 1 internal nodes. So, there is a total of 2 n minus 1 nodes and in the array representation the n leaves will occupy positions n to 2 n minus 1 both inclusive.

So, the first thing that we do is to copy A 1 to n into B n to 2 n minus 1. This copying can be done in order 1 time, if you have as many processes as there are elements. The ith processor will read the ith bit which is the ith location of A and that will be copied into the n minus 1 plus ith location of B. The ith element of A has to be copied into the n minus 1 plus ith location of B. Thus, copying can be done in order 1 time, if you have n processors. This is the 1st step.

(Refer Slide Time: 09:11)



Step 2 Binary tree in B

Substep 1 n/2 nodes that are parents of leaves

n/2 prcrs 1 time

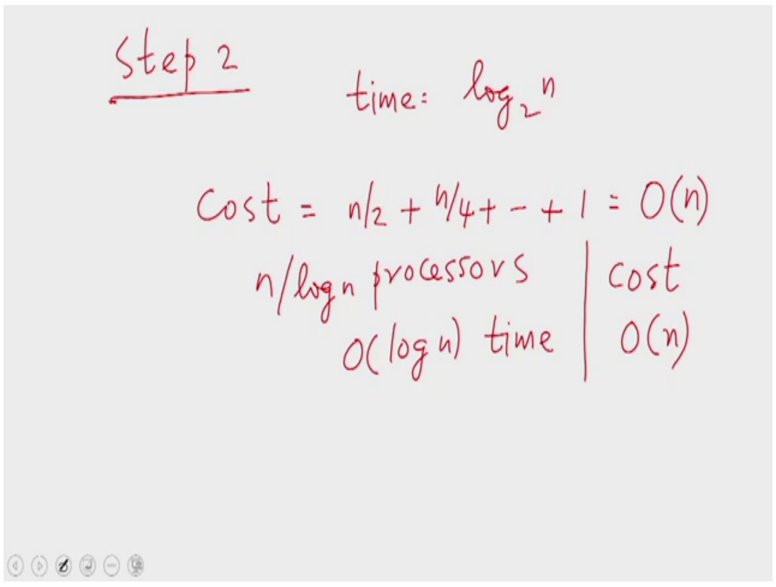
Substep 2 n/4 prcrs 1 time

Substep 3 n/8 prcrs 1 time

Then in the 2nd step; the 2nd step has several sub steps. In the second step we go up the tree, now we have a binary tree in B and we go up the tree performing OR of 2 elements at a time thereby reducing the size of the array by a factor of 2 in every single sub step. So, initially we have n by 2 nodes that are parents of the leaves. These nodes are first filled; assume you have one processor sitting at each of these nodes. These processor will look at the children of these nodes and they will find the OR of the children and keep them at these nodes, that is the processor sitting here will look at the left child as well as the right child, take the OR of the bits at the children and store it within the node.

So, this will require n by 2 processors unit time. This is the 1st sub step; in the 2nd sub step we use n by 4 processors. These n by 4 processors will occupy the parents of the nodes which were computed in the previous sub step and these nodes can again be filled in unit time. So, the second step can be executed using n by 4 processors in 1 unit of time. This is the 2nd sub step, by extending the argument we find that in the third step 3rd sub step using n by 8 processors in 1 time we can fill, the nodes that are at level 3 from the bottom.

(Refer Slide Time: 11:44)



Step 2

time: $\log_2 n$

$$\text{Cost} = n/2 + n/4 + \cdots + 1 = O(n)$$

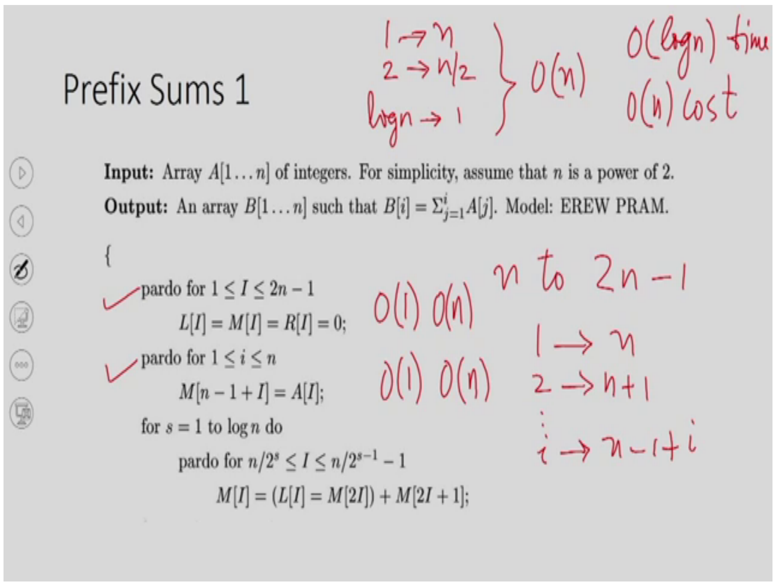
$n/\log n$ processors | cost

$O(\log n)$ time | $O(n)$

Continuing like this we find that in 2nd step, we compute the OR of n bits by performing n by 2 operations in the 1st step, n by 4 operations in the 1st 2nd sub step and so on. Finally, we have a step containing just ones operation. So, this is order of n. So, the cost of the entire second step is order of n whereas, the time taken by the 2nd step is log n to the base 2. So, here we have a perfect situation where Brent scheduling principle can be applied.

This is a step that runs in log n steps and the total cost of all these sub steps put together as order n. So, by Brent scheduling principle if we have n by log n processors, we should be able to solve this step in order of log n time for a total cost of order of n. So, by Brent scheduling principle step 2 can be scheduled in order of n cost and order of log n time finally, in the 3rd step we return the roots value.

(Refer Slide Time: 13:26)



Prefix Sums 1

$$\left.\begin{array}{l} 1 \to n \\ 2 \to n/2 \\ \log n \to 1 \end{array}\right\} O(n) \qquad \begin{array}{l} O(\log n) \text{ time} \\ O(n) \text{ cost} \end{array}$$

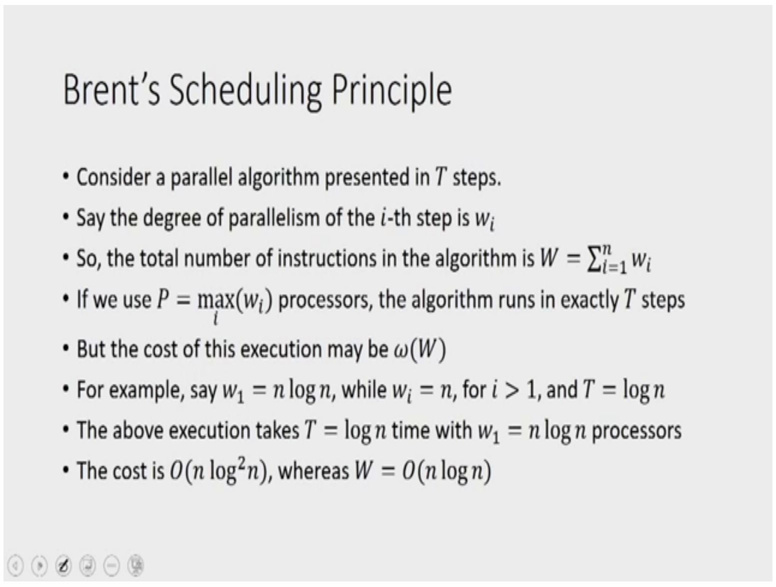
Input: Array $A[1 \ldots n]$ of integers. For simplicity, assume that n is a power of 2.
Output: An array $B[1 \ldots n]$ such that $B[i] = \Sigma_{j=1}^{i} A[j]$. Model: EREW PRAM.

```
{
✓ pardo for $1 \leq I \leq 2n-1$                    $O(1)$ $O(n)$    $n$ to $2n-1$
      $L[I] = M[I] = R[I] = 0$;
✓ pardo for $1 \leq i \leq n$                       $O(1)$ $O(n)$    $1 \to n$
      $M[n-1+I] = A[I]$;                                               $2 \to n+1$
  for $s = 1$ to $\log n$ do                                            $\vdots$
      pardo for $n/2^s \leq I \leq n/2^{s-1} - 1$                       $i \to n-1+i$
          $M[I] = (L[I] = M[2I]) + M[2I+1]$;
```

So, this is the algorithm summarized to find the OR of n bits on EREW PRAM where, the input is an array A of n bits and the output is to be the OR of these bits. In the 1st step in parallel for every single element of the array, we copy the element into the leaf of the binary tree that we are going to construct in array B. Now, the elements occupy the leaf positions in array B, then we have log n sub steps we have a loop with log n iterations. In each iteration, we fill one level of the tree going from bottom to top.

So, in the 1st sub step we fill the penultimate level of the tree, the parents of the leaves then in the 2nd step we fill the grandparents of the leaves and so on. Finally, at the end of the loop in the log n step the root will contain the result of the OR of these n bits. And, the value which is contained in the root, the root of the array B is B B 1. This value is returned by the first processor nowhere, in this algorithm have we used an exclusive read or have we used a concurrent read or a concurrent write. Therefore, this is an exclusive read exclusive write algorithm.

(Refer Slide Time: 15:00)

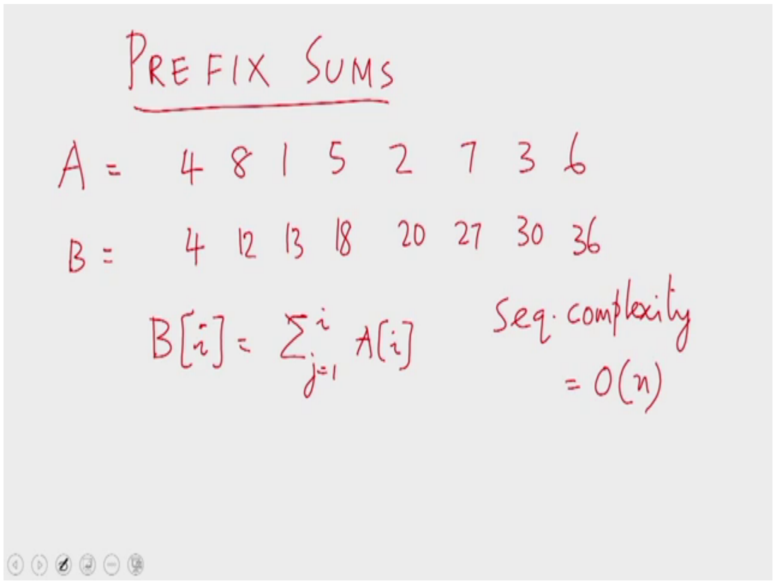


Brent's Scheduling Principle

- Consider a parallel algorithm presented in T steps.
- Say the degree of parallelism of the i-th step is w_i
- So, the total number of instructions in the algorithm is $W = \sum_{i=1}^{n} w_i$
- If we use $P = \max_i(w_i)$ processors, the algorithm runs in exactly T steps
- But the cost of this execution may be $\omega(W)$
- For example, say $w_1 = n \log n$, while $w_i = n$, for $i > 1$, and $T = \log n$
- The above execution takes $T = \log n$ time with $w_1 = n \log n$ processors
- The cost is $O(n \log^2 n)$, whereas $W = O(n \log n)$

So, the algorithm runs in order of log n time for a cost of n order n. Therefore, this is an optimal algorithm.

(Refer Slide Time: 15:14)



PREFIX SUMS

A = 4 8 1 5 2 7 3 6

B = 4 12 13 18 20 27 30 36

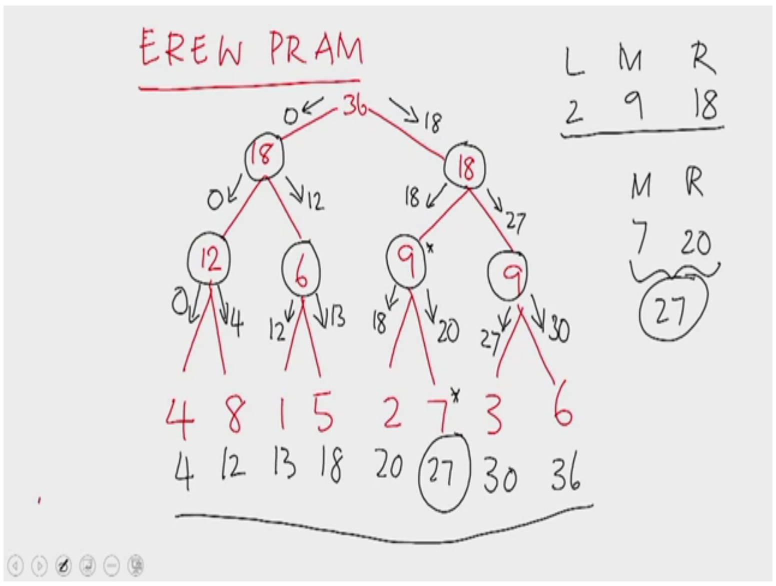
$B[i] = \sum_{j=1}^{i} A[i]$

Seq. complexity $= O(n)$

So, this is also an example of an application of Brent's scheduling principle. The next problem we shall consider is that of prefix sums. The input to the prefix sum problem is an array, an array of integers. Let us say this is the array given to us, this is array A; what is needed is to compute an array B. The array B has to be computed so, that the ith location of B will contain the sum of the first i locations of A. That is in B the first location should contain 4 which is the sum of which is the first location of which is the content of the first location of A.

In the second location of B we should store 12 which is 4 plus 8, in the third location we should store 13 which is 4 plus 8 plus 1. Then, 18 in the fourth location which is the sum of the first four locations of A. Then, 20 27 30 36. So, when the sums are computed in this fashion we say B form the prefix sums of A. So, you can see that the sequential complexity of the problem is order of n. The sequential algorithm is an easy one, you scan the array from the left to right from the left to the right and as you go along you form the sums.

(Refer Slide Time: 17:13)



Now, let us see how to design a parallel algorithm for this problem on EREW PRAM. First let me explain how the algorithm will work. We take the array that is given to us and construct a binary tree on top of this. And, as we go up the binary tree we sum the values. The parent is going to contain the sum of the values of the children. So, the parent of 4 and 8 will contain 12, the parent of 1 and 5 will contain 6. The parent of 2 and 7 will contain 9, the parent of 3 and 6 will contain 9 again. The parent of 12 and 6 will contain 18 and the parent of 9 and 9 also will contain 18 and then the rope will contain 36. So, by moving up the binary tree starting from the leaf nodes, we compute this value at every single internal node of the tree. At the root of the tree we will have the sum of all the given elements.

Now, if you look at the root what we find is that the sum on the left side is 18 and the sum on the right side is 18. Our goal is to find the prefix sums of the values that exist at the leaves. Now, you find that the left sum namely the 18 that is coming from the left side here has to be added to every single element on the right side. Therefore, what the root does is to send the value it obtained from the left side namely 18 onto the right side. The root also sees that a value of 0 in comparison to the 18 that must be added to the nodes on the left right side; a value of 0 should be added to the leaves in the left subtree of the root.

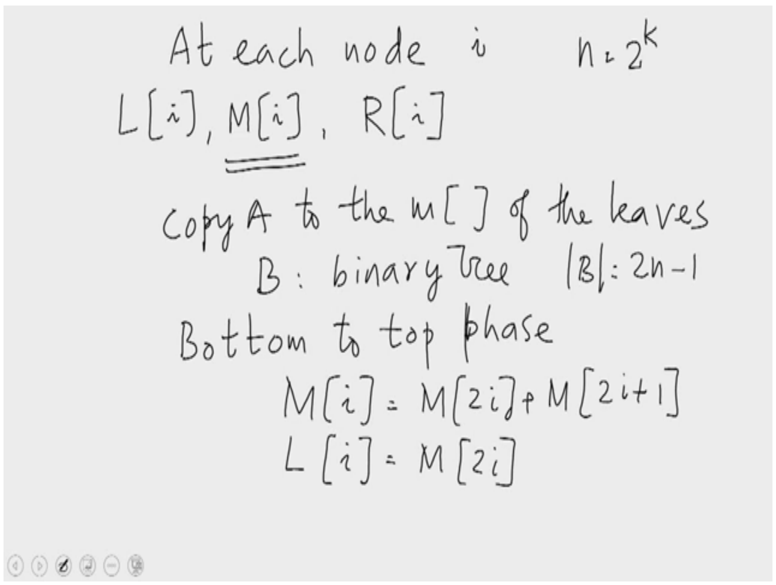
Therefore, the root will forward a value of 0 towards its left side and a value of 18 towards its right side. Now, let us consider the node with a value of 12. This is a child of the node with a value of 18 here. This node has obtained a value of 12 from the left side and a value of 0 from above. Therefore, this node realizes that a value of 12 has to be added to every single node on its right subtree. So, accordingly it will send a value of 12 to its right child and similarly a value of 0 is to be added to every single node in the left subtree of this node. Therefore, it will forward 0 towards the left side.

When we come to this node instead, what we find is that this node has received an 18 from above which must be added to every single leaf of this node. Therefore, this 18 has to be forwarded on all downward paths, but then in addition to that it also received a 9 from the left side. So, this 9 and this 18 together have to be added to every single leaf of the right subtree. Therefore, what it forwards to the right side is 18 plus 9 27 and it forwards the 18 which it received from the parent without any change to the left side. This is to be added to every single leaf of this subtree. Now, when we come to the node that contains 12; it has received a value of 0 from above and it receives a value of 4 from below on the left side.

Therefore, it will forward this value 4 onto the right side onto the left side it forwards a 0. The node which is labeled 6 receives 12 from above and it receives 1 from the left side. Therefore, it will send 13 to the right side; what it sends to the left side will be the value it receive from the top which is 12. This 12 has to be added to all its children. Now, the left node with a label of 9 receives a value of 18 from above and it receives 2 from its left child. The sum of these two namely 20 will be sent to its right child and the value it receives from the top which is 18, will be sent to the left side. The node that contains, the second node that contains 9 has received 27 from above and it has received 3 from its left child.

The sum of these two, it is 30 will be sent to its right side. What it sends to its left side is the value which received from the top which is 27. So, now the leaves are receiving some values from above, these values have to be added to the contents of the leaves. So, 4 plus 0 is 4 8 plus 4 is 12 1 plus 12 is 13 13 plus 5 is 18 2 plus 18 is 20 20 plus 7 is 27 27 plus 3 is 30 and 30 plus 6 is 36, we find that these are indeed the prefix sums of the given array. So, the algorithm works correctly. So, now, let us see a formal specification of the algorithm and do an analysis of the algorithm.

(Refer Slide Time: 23:18)



At each node i $n = 2^k$

$L[i], \underline{\underline{M[i]}}, R[i]$

Copy A to the $m[\ ]$ of the leaves

B : binary tree $|B| : 2n-1$

Bottom to top phase

$M[i] = M[2i] + M[2i+1]$

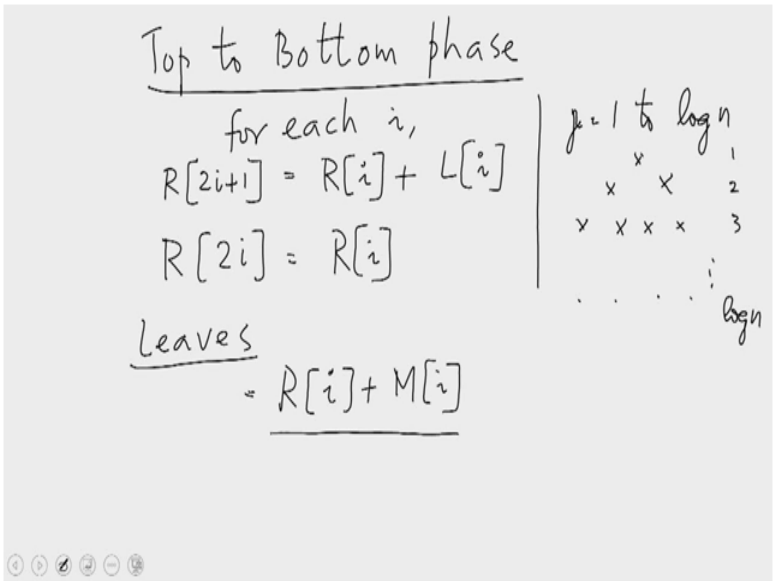
$L[i] = M[2i]$

So, how do we achieve this effect in a formal specification of the algorithm. For this what we do is this at each node, we store 3 values. M i which we call the middle value, L i which we call the left value and R i which we call the right value. So, initially we want the given array to occupy the middle values of the leaves. So, what we do first is to copy the given array to the m values of the leaves. So, we have an array B in which we will hold the binary tree as in the previous algorithm. This array B has a size of 2 n minus 1, a binary tree a full binary tree of n leaves will have 2 n minus 1 nodes.

Here again we assume that n is a power of 2 for simplicity, if n is not a power of 2 you can always pad n up to the nearest power of 2. So, we have copied A into the m values of the leaves and then as in the previous algorithm we have a bottom to top face. In the bottom to top face the values are received from the bottom by each node and the values that are received are summed up and kept in the middle value, that is we repeatedly do this.

M of i is M of 2 i plus M of 2 i plus 1. Note 2 i is the left child of node i and node 2 i plus 1 is the right child of node i. So, what node i does is to sum the values that it receives from the left side as well as the right side. And, this sum will be stored in the middle value of i and the left value of i will contain the value it receives from the left child. So, in the bottom to top pass we repeatedly perform this.

(Refer Slide Time: 26:12)



Top to Bottom phase

for each i,

$R[2i+1] = R[i] + L[i]$

$R[2i] = R[i]$

Leaves

$= R[i] + M[i]$

$j = 1$ to $\log n$

1

2

3

$\log n$

Once we reach the top the root will contain the sum of all the elements. Now, the second part of the algorithm which is the top to bottom phase begins. In the top to bottom phase for each node i, when it comes alive what we have to do is this; the value which it received from the top from the parent which is let us say in its right value. So, we assume the right value of a node is the value which it receives from the top. This plus the value it receives from the left side should be sent to the right child. So, this is what is to be the right value of its right, 2 i plus 1 is its right child and the R value is supposed to be the value which it receives from above receives from the parent.

So, 2 i plus 1 node 2 i plus 1 is going to receive this value R i plus L i from node i which is its parent. Analogously, node 2 i which is the left child of node i will receive a value which is the same as R i; that is what node i receives from the parent will be passed on to node 2 i without any change. Now, this is done repeatedly for each i as node i comes alive. So, node i comes alive when the entire level of the nodes in which i belongs is brought up alive. So, there are log n phases as you go down when j varying from 1 to log n; we bring alive the levels of the tree.

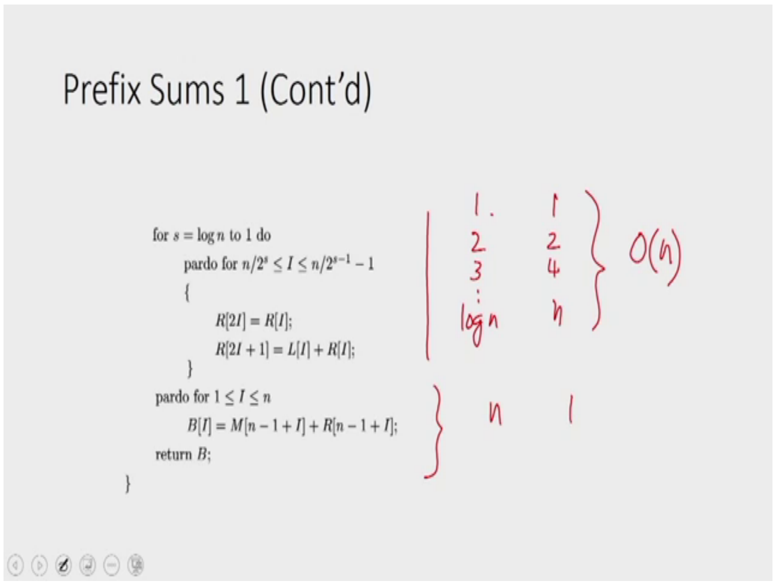
When j equal to 1 the root is activated, when j equal to 2 the children of the root are activated, when j equal to 3 the grandchildren of the root are activated and so on. So, when j equal to log n we will be activating all the leaves. So finally, all the leaf values will be filled, then in the last step of the algorithm we look at the leaves. At the leaves we have to calculate the sum of the value that the leaf received from the parent, which will be present in its write value plus the original value that was present there.

This is to be the final value of node i. So, this is how we calculate the prefix sums. So, here you can see that at every node there are 3 values. For example, at this node there are 3 values: the left value, the middle value and the right value. The left value is what it receives from the left side. So, what it receives from the left side is the sum on the left side which is 2. The middle value is the sum of the values have received from the 2 children which happens to be 9; 2 plus 7 9 and the R value is what it receives from above what this node receives from above is 18.

So, node that is marked here will contain these values when the algorithm is finished. Finally, at the bottom for example, at this node the original value that was presented this node is 7 which was stored and stored as the middle value of this node. And, the value it receives from the parent will be presented in the right value which is 20. The sum of these two will form the prefix sum corresponding to this position. So now, let us see a formal specification of this algorithm. In the first step of the algorithm for every single node of the binary tree we set all the left values, middle values and right values to 0. And, then in the second step the elements of the array A are copied into the leaves of the binary tree that is what is happening here.

The leaves are at locations n to 2 n minus 1. So, leaf 1 will occupy position n, leaf 2 will occupy positions n plus 1 and so on. So, in general leaf i will occupy position n minus 1 plus i. This is exactly as in the previous algorithm. The ith element of the array is copied into the ith leaf of the tree, after that we have log n steps. This is the bottom to top face, in the ith step we fill the ith level of the tree that is ith from the bottom. The filling is done this way the left value of a node will be the same as the value sent by the left side. And, the middle value of the node will be the sum of the values from the 2 children. When we reach the root, the root will contain the sum of all the values.

(Refer Slide Time: 32:07)



Prefix Sums 1 (Cont'd)

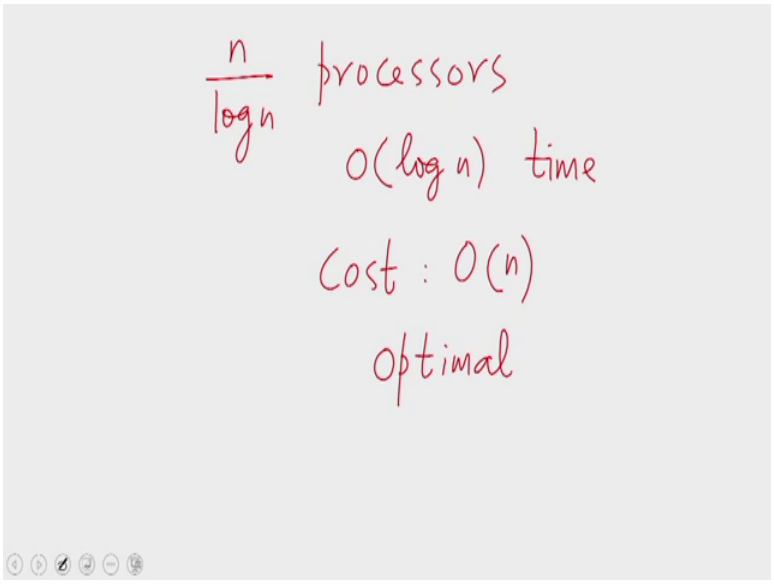
```
for $s = \log n$ to 1 do
    pardo for $n/2^s \le I \le n/2^{s-1} - 1$
    {
        $R[2I] = R[I]$;
        $R[2I + 1] = L[I] + R[I]$;
    }
pardo for $1 \le I \le n$
    $B[I] = M[n - 1 + I] + R[n - 1 + I]$;
return $B$;
}
```

Then start a top to bottom phase, in the top to bottom phase we have again login steps and what each node does is this. The value it received from the top namely R I, this will be sent to the left child without any change. And, the value to see from the top added to its left value, that is the value which it received from the left child will be sent on to the right child. So, R 2 I plus 1 will be set to L I plus R I. So, at this point all the leaves have collected the values that must be added to their original values, that is what is done in the final step. For every leaf in parallel what we do is this, the middle value and the right value are added together.

The middle values, the original value and the right value is the value which had got from the parent and then the array is returned. Now, let us do an analysis of this algorithm. The first two steps, the initialization steps as well as the copying of the elements into the leaves can be done in order of 1 time for a cost of order of n. Order of 1 time order of n cost and here also order of 1 time order of n cost. In the third step which is the bottom to top phase, there are log n sub steps. In these log n sub steps the degree of parallelism varies. The first step has a degree of parallelism of n, the second step has a degree of parallelism of n by 2 and so on.

The last step which is the log n step has a degree of parallelism of 1 therefore, the total cost here is order of n. So, by Brent scheduling principle all these steps together can be executed in order of log n time and order of n cost. Similar, is the case with the top to bottom face. Here again there are log n steps, in the first step the degree of parallelism is 1. In the second step the degree of parallelism is 2, in the third step the degree of parallelism is 4 and so on. So, in the log n step the degree of parallelism is n. So, here again the total cost is order of n; using n by log n processors in order of log n time we can execute this using Brent scheduling principle and the final step is 1 of summing at every single node. So, with n processors you can execute this in 1.

(Refer Slide Time: 35:08)



$\frac{n}{\log n}$ processors

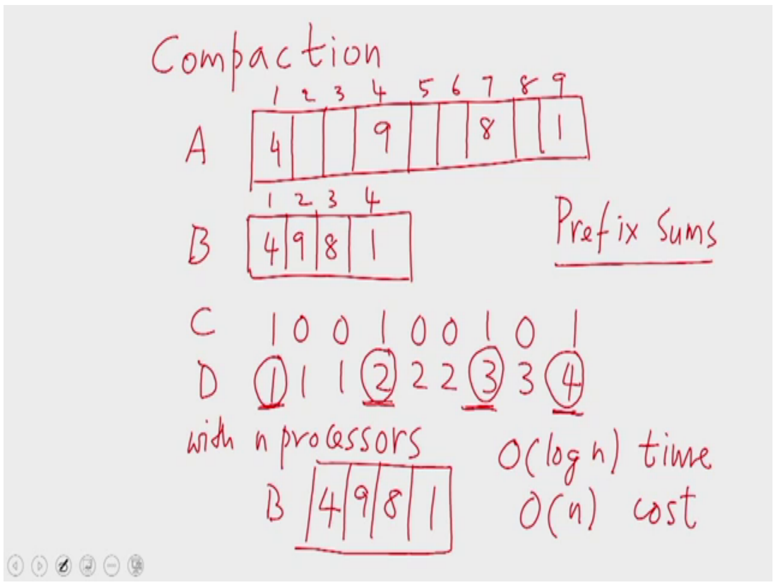
$O(\log n)$ time

Cost : $O(n)$

optimal

Therefore, if you apply Brent scheduling principle to the entire algorithm we find that, with n by log n processors we can find the prefix sum of n integers in order of log in time. Therefore, the cost of the algorithm is order of n which is the same as the sequential complexity of the problem; therefore, we say this algorithm is optimal.

(Refer Slide Time: 35:45)



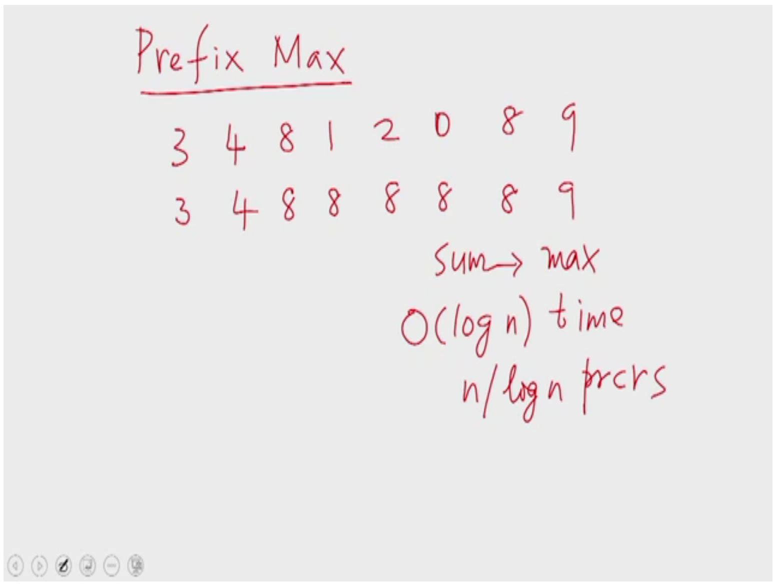
Now, let us see a few applications of the prefix sum problem. The first one is that of compaction. Let us say we have an array A, in which not every element is every location is occupied by useful element. Let us say we have an array like this in which only locations 1 4 7 and 9 have useful elements. So, this is the input to the algorithm. We want to compact A into an array B in which all the useful elements will occupy consecutive positions. How do we achieve this using terrific sums? We can use prefixes as a subroutine. What we do is this; we take an array a bit vector an array of bit vectors and indicate which positions have useful elements.

So, this is array C; array C indicates which positions have useful elements and then we find the prefix sums of array C. The prefix sums would be, D forms the prefix sums of array C. Now, out of D let us consider only those elements at which C has a 1. So, this element has a 1 in C, this element also has a 1 in C, this element also has a 1 in C and this element also has a 1 in C. So, let us say we have n processors. So, let us assume one processor occupies each single position of D, if a processor is qualified the qualified nodes have been circled here. So, if a processor is qualified then the processor will copy the corresponding location of A into the index provided here.

So, the processor that is sitting on the first element of D will copy the corresponding element which is 4 into the first location of B. So, B is going to contain 4 in its first location. The processor which is occupying the fourth position in array D, we will take the corresponding element namely D and copy it to the second position here which is the value that D 4 contains. So, 9 will occupy the second position of B, the processor which is at the seventh position; namely this takes the corresponding element which is 8 and copies it into the third position. And, the processor which occupies the ninth position will take the corresponding element namely 1 and copy it into the fourth position of array B.

Now, you find that array B contains the elements has to be desired. A has been compacted into B and the complexity of the algorithm is dominated by that of prefix sums. Therefore, the design time complexity of compaction is identical to that of terrific sums; the rest are all order 1 steps to be executed with n processors. So, if you have n by log n processors those steps can also be executed in order log n time. Therefore, the complexity is order of log n time and for a cost of order of n.

(Refer Slide Time: 40:24)



Prefix Max

3 4 8 1 2 0 8 9

3 4 8 8 8 8 8 9

$\text{sum} \rightarrow \text{max}$

$O(\log n)$ time

$n/\log n$ prcrs

A problem related to prefix sums is prefix minimum or maximum. We shall come to the prefix maximum, prefix minimum is analogous. Given a set of elements the prefix maxima of the set of elements will be the largest element you have seen 2 at each position. So, at the first position 3 is the largest element you have seen till then, at the second position it is 4 and the third position it is 8. At the fourth position it is the largest of the first four elements which is 8. Here again it is 8, here again it is 8, here again it is 8, at the final position it is 9; 9 happens to be the largest.

So, prefix maximum can be computed using an algorithm that is exactly analogues to prefix sums; all you have to replace or all you have to do is to replace sums with the maximum of 2 elements. So, this problem also can be solved using the binary tree technique in order of log in time using order of n by log n processors, that is it from this lecture. Hope to see you in the next lecture.

Thank you.