

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

Lecture – 07
Basic Techniques 3

Welcome to the 7th lecture of the NPTEL MOOC on Parallel Algorithms. In the previous couple of lectures, we have been seeing various algorithm design techniques. Today, we shall discuss an algorithm design technique that we shall call Accelerated Crowding. In this design technique, we have a problem instance, the size of which shrinks through steps, but the number of processors remain the same.

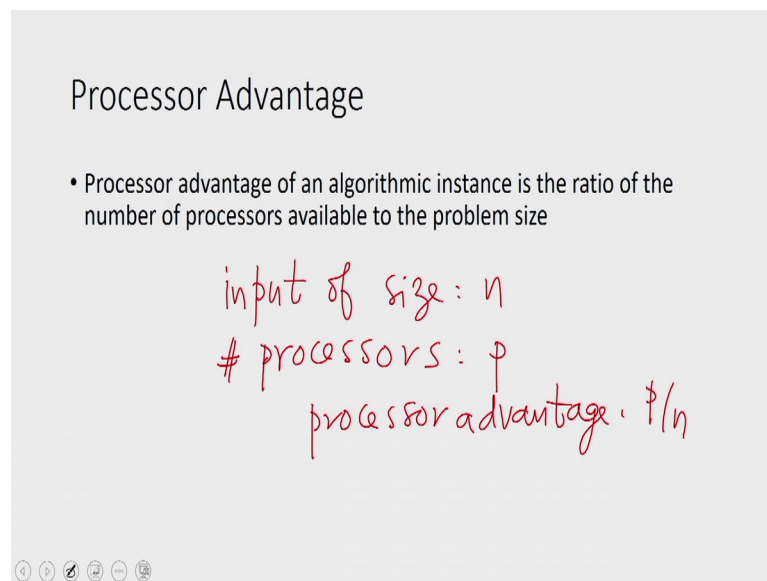
So, as the problem proceeds through the steps, as the algorithm proceeds through its steps, we find that the same number of processors are crowding over smaller and smaller problem instances and therefore, these smaller problem instances can be solved faster.

(Refer Slide Time: 01:14)

Processor Advantage

- Processor advantage of an algorithmic instance is the ratio of the number of processors available to the problem size

input of size: n
processors: p
processor advantage: p/n



A crucial notion here is what is called the processor advantage. The processor advantage of an algorithmic instance is the ratio of the number of processors available to the problem size. Let us say we have an input of size n and we try to solve this input of size n using p processors. In that case, we say that the processor advantage is p by n . You have p processors to solve an instance of size n , therefore the processor advantage is the ratio p by n , the number of processors available to the problem size.

(Refer Slide Time: 02:13)

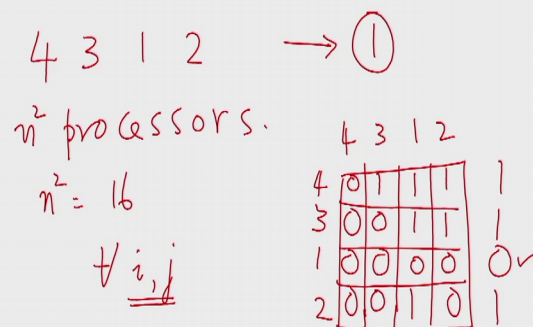
Accelerated Crowding

- Suppose for a problem P we have a super-fast but wasteful-in-processors algorithm
- Say, we want to design a moderately fast, economic-in-processors algorithm for P
- Sometimes a solution can be found by iteratively reducing the problem size so that the processor advantage of the new instances increases exponentially through the iterations

In the algorithm design technique that we call accelerated crowding, suppose for a problem p, we have a super fast algorithm. But, let us say those algorithm is rather wasteful in processors let us say we want to design a moderately fast, but economic in processors algorithm. A solution can be found usually by iteratively reducing the problem size so that the processor advantage of the new instances increases exponentially through the iterations.

(Refer Slide Time: 02:43)

Minimum on COMMON CRCW PRAM



We will consider an example. Let us say we want to find the minimum of n elements; for example, given an array of size let us say 4, we want to find the minimum of the array which is in this case 1.

So, we assume, we are given an array of size n , but let us assume that we have n squared processors, for a problem size of n . So, in this case, n squared equal to 16, we have 16 processors let us say. Using these 16 processors, what we do is this we take a 2 dimensional array a 4 by 4 array along the columns, we have along the rows, we have these values 4, 3, 1, 2 and along the columns also we have these values 4, 3, 1, 2. At each cell of the 2 dimensional array, we compare the row value with the column value. If the row value is greater than the column value, then we write a 1 in the cell; otherwise, we write a 0 in the cell.

At the diagonal positions, we will have 0's because when 4 is compared with 4 we have equality not a greater than sign. So, at all the diagonal positions we have a 0. For a row value of 3 and a column value of 4, we have a result of 0 because a 1 will happen only when the row value is greater than the column value. For the same reason, the value here will be 1; for a 1, 4, we have 0; for a 4, 1, we have a 1; for a 2, 4, we have a 0; for a 4, 2, we have a 1; for a 3, 1, we have a 1 but for 1, 3, we have a 0. For 3, 2, we have a 1 but for 2, 3, we have a 0; for 2, 1, we have a 1 but for 1, 2, we have a 0.

So, the array is filled in this fashion. For each location of the array for every ordered pair i, j , the array location will be filled. If the row value happens to be greater than the column value, then the array location will get a 1; otherwise we will get a 0. So, when the array is filled in this fashion, let us see for each of the rows, we take the minimum, we will take the OR of the bits in that row. So, in the topmost row, we have 0 1 1 1. So, the OR of these 4 bits is 1. In the second row, we have 0 0 1 1, again the OR is 1. In the third row, we have double 0, double 0, the OR of these four is 1, it is 0. In the bottommost row, once again we have 1.

So, we find that 0 happens exactly for the smallest element in the array because this is the element which will be adjudged less than or equal to every single column value including itself. Therefore, for this particular element every single entry in the row is going to be 0 and this will be the only row for which the OR value will be 0. So, when

we look at the OR results, the value for which the answer is 0 will be the smallest element in the array.

(Refer Slide Time: 06:31)

MINIMUM 1

MIN1
Input: Array $A[1 \dots n]$ of integers.
Output: The minimum integer R in A . Model: COMMON CRCW PRAM

Step 1: pardo for $1 \leq i \leq n, 1 \leq j \leq n$ $B[i][j] = (A[i] > A[j])$;

Step 2: pardo for $1 \leq i \leq n$ $C[i] = \text{OR-COMMON}(B[i])$;

Step 3: pardo for $1 \leq i \leq n$ if $(C[i] == 0)$ $R = A[i]$;

Step 4: return R ;

$O(1)$ time
 using n^2 processor
 processor adv: $\frac{n^2}{n} = n$

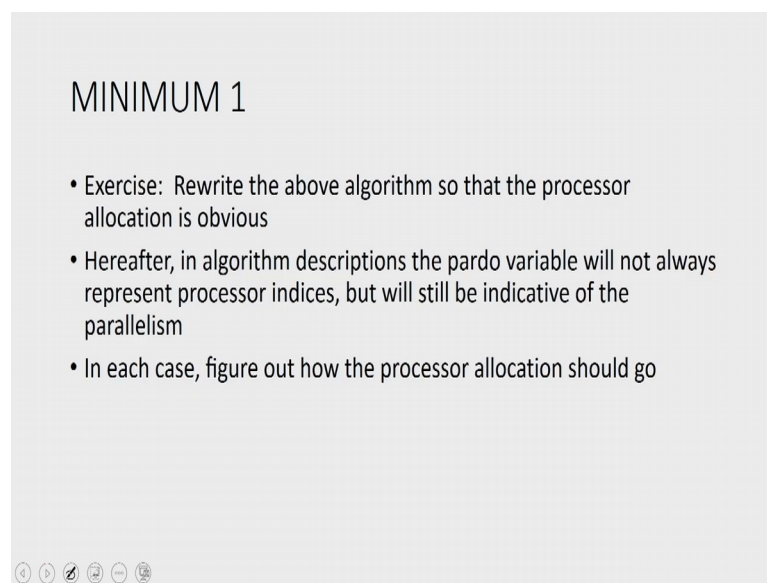
So, the algorithm that we have just seen works like this. We are given an array of size n , we want to find the minimum of this array. First what we do is this, we take a 2 dimensional array and then for each location i, j we fill the location with the Boolean value a_i greater than a_j that is if a_i is greater than a_j , the location will get 1; otherwise the location will get a 0. So, this is what we have done here, in the 2 dimensional array, we have filled the logical values after every comparison. Since, we are doing this in parallel for every ordered pair i, j , we will require n squared processors because the 2 dimensional array which is an n by n array has n squared locations and for each location we will have to dedicate a processor.

With n squared processors, since what we have is a common CRCW PRAM, concurrent reads are permitted. Therefore, multiple processors can read the same input value simultaneously that is a_i can be read by several processes simultaneously for each i and therefore, b_{ij} can be filled in order 1 time. So, the array is filled in order 1 time. There in the second step, we have several parallel executions; one parallel execution for each row, what we do is to invoke the OR algorithm for common CRCW PRAM that we have studied in the earlier lectures on each row independently and in parallel. The result of the OR will be kept in C_i that is for the result of row i will be kept in C_i .

If C_i happens to be 0, we know that exactly one row will have a 0 value. If the minimum is unique, then the corresponding row value A_i will be returned as the smallest element. What if there are multiple minima? Then corresponding to all those elements, we will have a 0 value and all of them will be attempting to return the corresponding row values. But all these row values are identical therefore, on a common CRCW PRAM, we will have a concurrent right where every process are attempting to write will be writing the same value which is a valid right therefore, the right we will go through. The algorithm works correctly.

So, we find that there are only three steps. Each of the steps of the algorithm will work in unit time. So, we have an order 1 time we have an, we have now order 1 time algorithm using n^2 processes, to solve an instance of size n . Therefore, the processor advantages n^2 by n , the number of processors divided by the problem size which is n . So, here we find that the problem size as well as the processor advantage are identical which is n .

(Refer Slide Time: 09:44)



MINIMUM 1

- Exercise: Rewrite the above algorithm so that the processor allocation is obvious
- Hereafter, in algorithm descriptions the pardo variable will not always represent processor indices, but will still be indicative of the parallelism
- In each case, figure out how the processor allocation should go

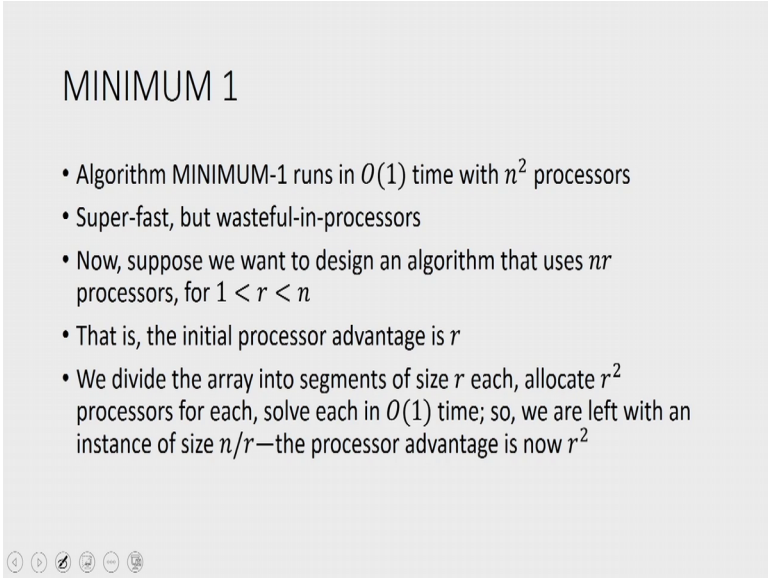
Navigation icons: back, forward, search, etc.

So, what we know now is that the problem of finding the minimum of n elements can be solved in order 1 time on common CRCW PRAM using a processor advantage which is equal to the problem size. Please take a look at the algorithm specification here the processor allocation has not been made explicit here. In the first step, all I say is that for

all possible ordered pairs i, j , $b[i, j]$ has to be filled according to the specification, but then which processor should do the copying of which i, j is not specified.

So, that allocation is unspecified here. So, hereafter in algorithm descriptions the pardo variable will not always represent processor indices, but will still be indicative of the parallelism. The allocation will not always be explicitly stated but in each case, you would be able to figure out the processor allocation easily now.

(Refer Slide Time: 10:43)



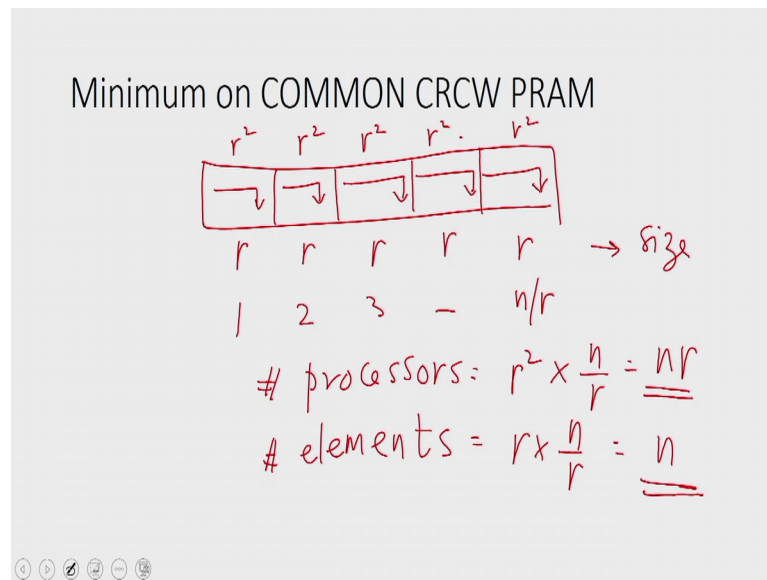
MINIMUM 1

- Algorithm MINIMUM-1 runs in $O(1)$ time with n^2 processors
- Super-fast, but wasteful-in-processors
- Now, suppose we want to design an algorithm that uses nr processors, for $1 < r < n$
- That is, the initial processor advantage is r
- We divide the array into segments of size r each, allocate r^2 processors for each, solve each in $O(1)$ time; so, we are left with an instance of size n/r —the processor advantage is now r^2

So, this algorithm runs in order 1 time with n squared processes. This is a bit super fast algorithm, but it is very wasteful in processes. It is a non optimal algorithm to find the minimum of n integers, all you require is order n time sequentially therefore, we would expect an algorithm to run in order n cost, but this is very wasteful this algorithm runs in order one time using n squared processors.

So, for the cost of the algorithm is order of n squared. Now, suppose we want to design an algorithm that uses $n r$ processors where r is less than n , that is the initial process advantages are instead of n that we have just seen. So, what we do is this.

(Refer Slide Time: 11:26)

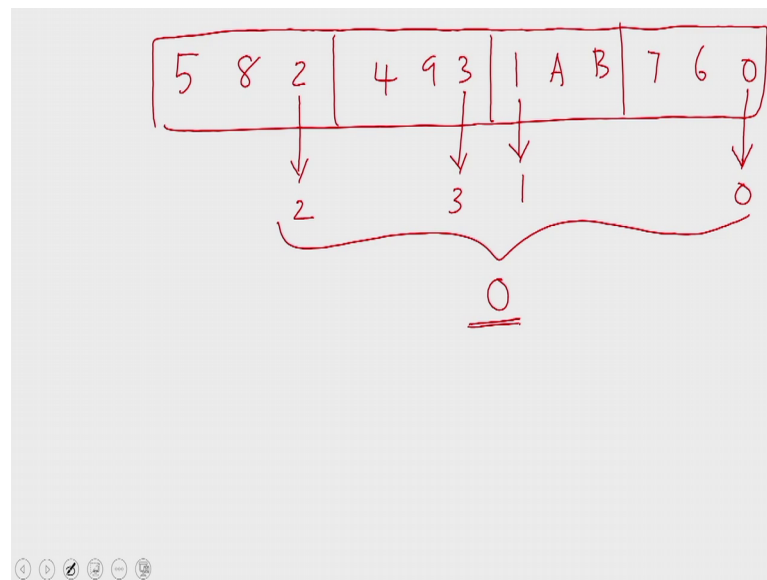


We divide the given array into a number of segments. Each segment is of size r . So, the segments can be numbered like this. The leftmost segment is called the first segment and the rightmost segment is called the n by r th segment. To each segment we allocate r squared processors. So, the total number of processors allocated is r squared is the number of processors are located per segment and n by r is the number of segments. So, the product is nr which is indeed the number of processors that we have and the number of elements in all the segments put together is r into n by r which is n .

So, as you can see the processor advantages are initially and the problem has been divided in this fashion. Now, each of the segments can be solved in order 1 time because the segment has the size of r and there are r squared processors allocated to the segment. Using the algorithm that we have just seen where the processor advantage is equal to the size of the problem, the problem can be solved in order 1 time. So, each of these segments can be solved in order 1 time; that means, we find the minimum within each segment. How will we find the global minimum now?

We have obtained the local minima of the segments. If you take the minimum of all these minima, then you would get the global minimum. Let me take an example.

(Refer Slide Time: 13:45)



Let us say we have an array of size 12 A and B are hexa decimal 10 and 11 respectively. So, let us say this is the given array. Suppose, we divide this array into segments of size 3 each, our goal is to find the minimum in the entire array. We divide this into several local problems and solve each local problem.

The smallest element within the first segment which is 5, 8 and 2 is 2. The smallest element in the second segment which contains 4, 9 and 3 is 3. The smallest element in the third segment which contains 1, A and B is 1 and the smallest element in the fourth segment is 0.

So, the original instance of size 12 has now reduced to an instance of size 4. If you find the smallest of these four elements, you would have the answer for the global problem. The smallest element in the array in any case would have been a local minimum as well. Therefore, when we find the local minima this certainly would be selected. Therefore, in the reduced problem instance, the smallest element will be present and this indeed is the global minimum therefore, it will be smaller than any other selected elements selected from the other segments.

Therefore, when we take the minimum of these elements, you will indeed be returning the global minimum as the value. Therefore, the idea of algorithm is correct.

(Refer Slide Time: 15:36)

MINIMUM 2

Input: Array $A[1 \dots n]$ of integers; r the processor advantage.

Output: The minimum integer R in A . Model: COMMON CRCW PRAM

Step 0: if $(n == r)$ return $\text{MIN1}(A[1 \dots n])$;

Step 1: pardo for $1 \leq i \leq n/r$ $B[i] = \text{MIN1}(A[(i-1)r + 1 \dots ir])$;

Step 2: return $\text{MIN2}(B[1 \dots n/r], r^2)$;

So, now, let us see how the algorithm proceeds. If n equal to r , that is the processor advantages identical to the size of the problem, we invoke the previous algorithm which runs in order 1 time. Otherwise, we divide the array into r segments, the segments of size r . So, there are n by r segments for each segment in parallel, we invoke the previous algorithm using r squared processors, then we invoke minimum two recursively using this reduced array. The new array is given in b and it has n by r elements. On these n by r elements, we would invoke min 2 recursively using a processor advantage of r squared. Why is the processor advantage r squared now?

(Refer Slide Time: 16:24)

$$\begin{array}{l} \text{problem size} = n \\ \# \text{ processors} = nr \\ \text{pr. advantage} = r \\ \hline \text{new problem size} = \frac{n}{r} \\ \# \text{ processors} = nr \\ \text{pr. advantage} = \frac{nr}{\frac{n}{r}} = r^2 \end{array}$$

Initially, we had a problem size of n , the number of processes n/r which means the processor advantage is r . Now, after one invocation, the new problem instance has a size of n/r ; that is because we have one element coming out of every single segment and there are n/r segments. So, the new problem size is n/r , the number of processors is the same, we still have n/r processors.

Therefore, the processor advantage is n/r divided by n/r^2 , the number of process divided by the new problem size which is r^2 , that is the new problem sizes n/r and the new processor advantages r^2 . This allows us to write a recurrence relation.

(Refer Slide Time: 18:05)

MINIMUM 2

$$T(n, r) = T\left(\frac{n}{r}, r^2\right) + c$$

$\frac{n/r}{r^2} : n/r^3$ segment
 $\text{pr. adv} = \frac{nr}{n/r^3} = r^4$

If $T(n, r)$ is the time taken by a call $\text{MIN2}(A[1 \dots n], r)$, then for some constant c ,

$$T(n, r) = T(n/r, r^2) + c = T(n/r^3, r^4) + 2c = T(n/r^7, r^8) + 3c = \dots = T(n/r^{2^s-1}, r^{2^s}) + sc$$

With $s = O(\log(\frac{\log n}{\log r}))$, thus $T(n, r) = O(s) = O(\log(\frac{\log n}{\log r}))$.

T of n/r that is when we invoke the algorithm with a problem size of n and the processor advantage of r , we find that after spending order one time we managed to reduce the problem size to n/r and increase the processor advantage to r^2 .

Therefore, the recurrence relation will be like this. If we continue with this now, the new problem instance is n/r and the processor advantages r^2 . Therefore, now we will be dividing the array the remnant array of size n/r into segments of size r^2 each. Therefore, we will have n/r divided by r^2 which is n/r^3 ; these many segments and we will have one element coming out of each segment. So, the new problem size will be n/r^3 and the number of processors remain identical. Therefore, the processor advantage will be n/r divided by n/r^3 which is r^4 .

4. Therefore, the recurrence relation that we have is this $T(n, r)$ is $T(n/r, r)$ squared plus c .

So, when we unroll this once more, what we get is now the new problem instance is n/r^2 and there are n/r processes still for a processor advantage of r^2 . Therefore, at the next level of recursion, we would be dividing the remnant array into segments r segments of size n/r^4 . So, there will be n/r^3 segments now. Therefore, the next iteration will have a problem instance of size n/r^7 and therefore, the processor advantage would be r^8 . So, if you continue like this, now the pattern is clear. If you continue like this for s steps, you would have 1, 3 and 7 are all 1 less than a power of 2.

So, you can write this as $2^s - 1$. The processor advantage now would be r^{2^s} . This will be the situation after s levels of recursion and you would have spent a total of $s \cdot c$ time until then. When would the recursion stop?

(Refer Slide Time: 20:35)

recursion stops
 problem size = pr. adv.

$$\frac{n}{r^{2^s} - 1} = r^{2^s}$$

$$\Rightarrow n = r^{2^s} \cdot (r^{2^s} - 1) = r^{2^{s+1}} - r^{2^s}$$

$$\Rightarrow \log_r n = 2^{s+1} - 1$$

The recursion stops when the problem size is equal to the processor advantage, that is because when the problem size is equal to the processor advantage, we can invoke the first minimum algorithm that we saw that runs in order of 1 time.

So, after s steps, if we were to invoke that algorithm that would mean the number of elements is equal to the processor advantage, which means this implies that n is equal to

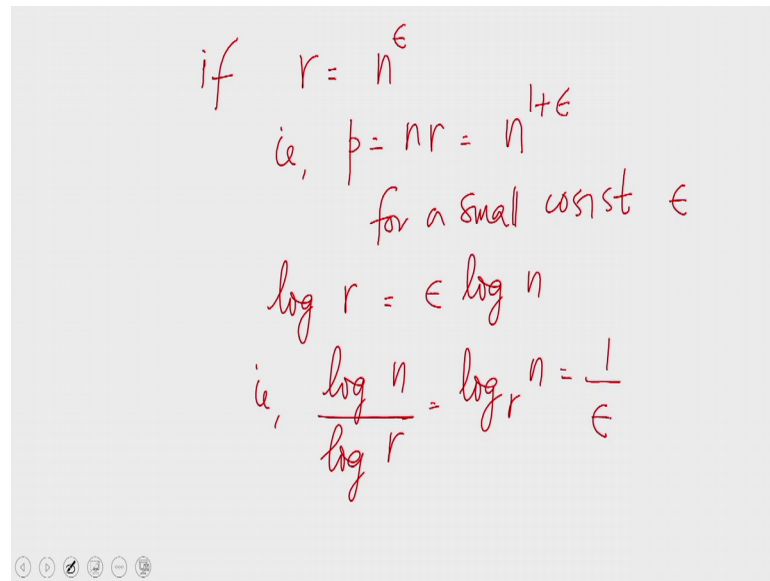
r^{2^s} into $r^{2^s - 1}$ or $r^{2^s + 1}$; n is equal to this. If you take logarithm on both sides logarithm to the base of r , you would get $2^s + 1$ on the right hand side, which means \log of n to the base r plus 1.

(Refer Slide Time: 21:57)

$$\begin{aligned} \log_r n + 1 &= 2^{s+1} \\ \log_2 (\log_r n + 1) &= s+1 \\ s &= \log_2 (\log_r n + 1) - 1 \\ &= O(\log_2 \log_r n) \text{ time} \\ &\quad \underline{\hspace{1.5cm}} \\ &\quad nr \text{ processors} \end{aligned}$$

It is the same as 2^{s+1} . If you take logarithm once again, you have logarithm to the base 2 of $\log n$ to the base r plus 1 is equal to $s+1$. That tells you after how many steps the recursion would stop; that is s is equal to logarithm to the base 2 of logarithm to the base r of n plus 1 the whole minus 1 or which is order of $\log_2 \log_r n$. So, the algorithm would run in order of logarithm to the base 2 of logarithm to the base r of n using nr processors. So, let us see what this entails for different values of r .

(Refer Slide Time: 23:22)

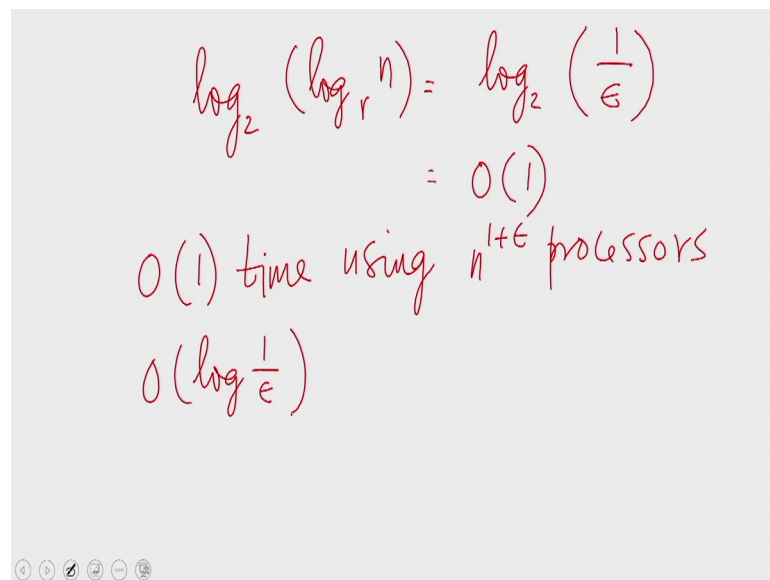


Handwritten mathematical derivation on a whiteboard:

$$\begin{aligned} \text{if } r &= n^\epsilon \\ \text{ie, } p &= nr = n^{1+\epsilon} \\ &\text{for a small const } \epsilon \\ \log r &= \epsilon \log n \\ \text{ie, } \frac{\log n}{\log r} &= \log_r n = \frac{1}{\epsilon} \end{aligned}$$

If you take r as n power epsilon, that is the number of processes p is nr which is n power 1 plus epsilon for a small constant epsilon, we find that $\log r$ is epsilon log n , that is log n by log r which is the same as log n to the base r . This is 1 by epsilon.

(Refer Slide Time: 24:18)



Handwritten mathematical derivation on a whiteboard:

$$\begin{aligned} \log_2 (\log_r n) &= \log_2 \left(\frac{1}{\epsilon} \right) \\ &= O(1) \\ O(1) \text{ time using } n^{1+\epsilon} \text{ processors} \\ O\left(\log \frac{1}{\epsilon}\right) \end{aligned}$$

Therefore, the running time of the algorithm would be which is order of log to the base 2 of log n to the base r . This would be the logarithm of 1 by epsilon, which is order of 1. So, the algorithm would run in order 1 time using n power 1 plus epsilon processors on the same model.

So, this is an improvement. The first algorithm that we had run in order 1 time using in squad processors, this new algorithm also runs in order 1 time but uses $n^{1+\epsilon}$ processors. The only difference is that the constant factor involved is larger here because here the actual running time is order of $1/\epsilon$. The smaller epsilon is the larger $1/\epsilon$ is therefore, $1/\epsilon$ is an increasing function in epsilon in $1/\epsilon$.

Therefore, the constant factor here is larger than the constant factor in the earlier algorithm that we saw, but the number of processors used is substantially less from n^2 we have reduced the number of processors to $n^{1+\epsilon}$ that is if you have r equal to n^ϵ .

(Refer Slide Time: 25:59)

$\text{if } r = 2$
 $O(\log_2 \log_2 n)$ time
 $\# \text{ processors} = 2n$
 $\log \log (2^{65536}) = \log 65536 = 16$
 $O(\log \log n)$ time n processors

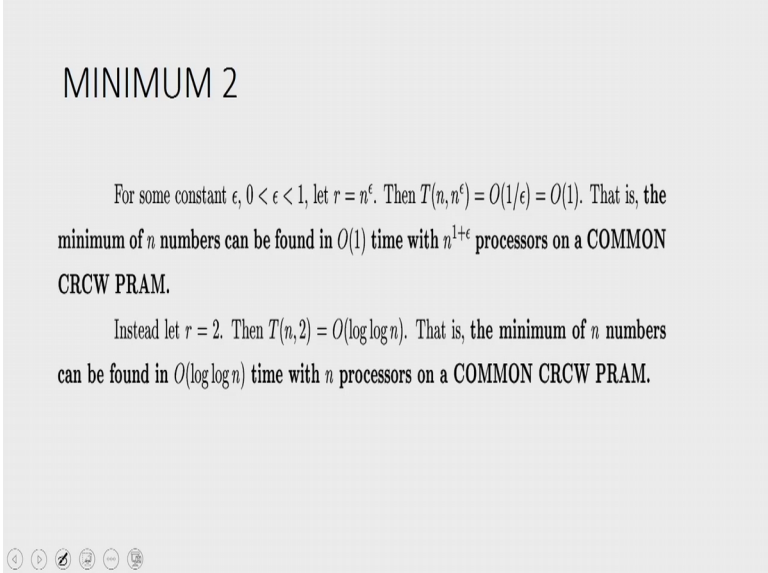
Let us try another value of r . If n is equal to 2, then the running time is order of log to the base 2 of log to the base 2 of n and the number of processors $2n$ that is if the processor advantage is to then the number of processes $2n$ and the running time is log of log n ; log of log n is an extremely slow growing function it is a logarithm of the logarithm. Therefore, even for a number of the size of 2^{65536} , log of log this would be log of 65536.

So, 16 therefore, for all practical values of n , you could say log of log n is less than or equal to 16. So, this algorithm is indeed very fast. If you have an algorithm that runs in order of log of log n time using $2n$ processors, then since the model is self simulating,

we can simulate the same algorithm using n processors for the same time complexity. You have the number of processors the running time doubles but even that with a doubling of the running time, we still have order of \log of $\log n$ running time and we have now used n processors.

The algorithm is still not optimal the cost of the algorithm is order of n times \log of $\log n$ whereas, finding the minimum of n elements sequentially takes only order n cost. Therefore, the algorithm is still not optimal. This is a suboptimal algorithm, but it is a very fast algorithm indeed it runs in order of \log of $\log n$ time.

(Refer Slide Time: 28:17)



MINIMUM 2

For some constant ϵ , $0 < \epsilon < 1$, let $r = n^\epsilon$. Then $T(n, n^\epsilon) = O(1/\epsilon) = O(1)$. That is, the minimum of n numbers can be found in $O(1)$ time with $n^{1+\epsilon}$ processors on a COMMON CRCW PRAM.

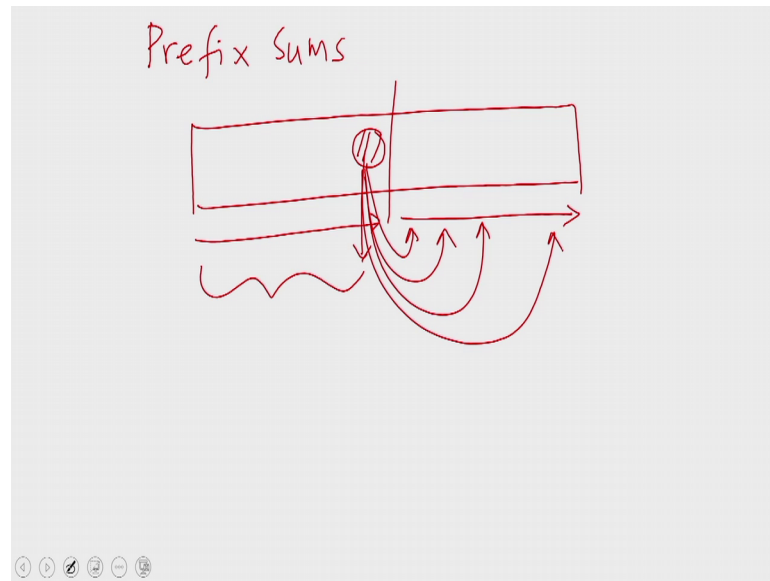
Instead let $r = 2$. Then $T(n, 2) = O(\log \log n)$. That is, the minimum of n numbers can be found in $O(\log \log n)$ time with n processors on a COMMON CRCW PRAM.

Navigation icons: back, forward, search, etc.

So, to summarize the algorithm for a constant epsilon where epsilon is between 0 and 1 runs in order of 1 time with n power 1 plus epsilon processors, instead if we let r equal to 2, then the running time of the algorithm is order of double log in, that is the minimum of n numbers can be found in order of double log n time with n processors. This is an example of the algorithm design technique called accelerated crowding. The next algorithm design technique that we are going to study is one that is familiar to you from the sequential algorithm setting, it is called Divide and Conquer.

So, in this case, given instance of the problem is divided into multiple problem instances of smaller size. We solve each of the smaller instances independently and then combine the results to form a solution for the global instance. We would consider a familiar problem.

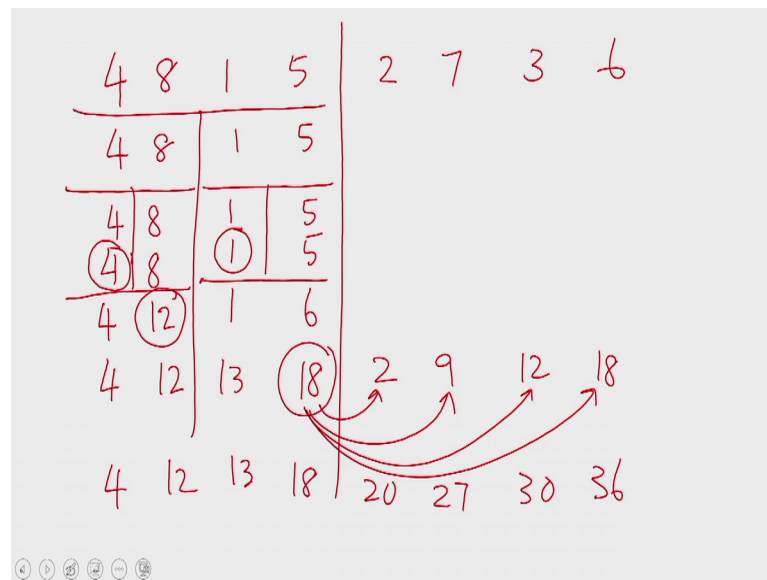
(Refer Slide Time: 29:10)



For example that of prefix sums. What we do is this. Suppose we are given an array of size n of which we want to find the prefix. Let us say we divide the array into two halves. We solve the prefix sum in each half independently, then the rightmost element in the left instance. This will be the sum of all the values in the left half, but when the right side of the division was solved, the elements to the left were not considered.

The prefix sums were computed starting from the point of division. To get the correct prefix sums in the right half what is needed is to add the sum on the left side to every single element on the right side; which means this sum value must be added to every single element on the right side. So, this is the recursive algorithm for finding prefix sums. Let us work out an example.

(Refer Slide Time: 30:30)



You are given an array of size 8, this is the same example that we saw before, we divide this in the middle. So, we have now two problem instances of size four each. To solve the left instance which is 4 8 1 5, we again divide this into two problem instances of size two each. To solve this instance of size two, we again divide it into two instances of size one each. The solution of an instance of size one is clear enough. The values are identical to the given values, when we have an instance of size one the problem has a trivial solution.

Then, the rightmost value of the left solution which in this case happens to be 4, this is to be added to every value on the right. The left value will remain as it is, but the value on the right will have to get this 4 added to it. Therefore, the right value will now be 12.

So, the solution for the array of size two, namely 2 8, 4 8 this is 4 12. Now, let us consider the instance 1, 5. This was divided into instances of size 1 each; their solutions would be 1 and 5. You take the solution of the sum of the left side which is the last value on the left side which in this case happens to be 1. This 1 has to be added to every value on the right, there is only 1 value on the right which is 5. So, 5 plus 1 is 6. On the left side, we have 1.

So, the solution for the instance 1, 5 is 1, 6. Then combining the solutions 4, 12 and 1, 6 we take the rightmost sum on the left side, which is 12 and this 12 is added to every value on the right side. Therefore, we have 13 and 18 here and on the left side we have 4 and 12 remaining as it is. So, the instance 4 8 1 5 has the solution 4 12 13 and 18 as you

can see the solution is correct. Similarly, for the instance 2 7 3 6 we form the solution 2 9 12 18.

So, now the problem on the left side as well as the problem on the right side have been independently solved. To find the global solution, once again we look at the rightmost sum on the left side which in this case happens to be 18. And add this to every value on the right side. Therefore, the final answer would be; on the left side, the values are written as it is.

So, 4, 12, 13 and 18 and on the right side, we have 18 added to these values 18 plus 2, 20, 18 plus 9, 27; 18 plus 12, 30 and 18 plus 18, 36. So, the final prefix values are prefix sum values are 4, 12, 13, 18, 20, 27, 30 and 36 which you can see are correct.

(Refer Slide Time: 34:32)

Prefix Sums 2

PREFIX-SUMS-2

Input: Array $A[1 \dots n]$ of integers. For simplicity, assume that n is a power of 2.

Output: An array $B[1 \dots n]$ such that $B[i] = \sum_{j=1}^i A[j]$. Model: CREW PRAM.

```

{
  if ( $n == 1$ ) return  $A$ ;
  pardo for  $0 \leq i \leq 1$ 
     $B[i \cdot n/2 + 1 \dots (i + 1)n/2] = \text{PREFIX-SUMS-2}(A[i \cdot n/2 + 1 \dots (i + 1)n/2]);$ 
  pardo for  $1 \leq i \leq n/2$ 
     $B[n/2 + i] = B[n/2];$ 
  return  $B$ ;
}
```

So, in general, the algorithm proceeds in this fashion. Given an array of size n where we assume that n is a power of 2, the algorithm proceeds like this. If n equal to 1, return the array as it is, that is if the input is an array of size 1, nothing is to be done. The array is returned as it is.

Otherwise, for i varying from 0 to 1, that is there are two instances; i equal to 0 and i equal to 1; these two instances are to be executed in parallel pardo for 0 less than or equal to i less than or equal to 1. Two instances to be executed in parallel, we solve 2 instances when i equal to 0. We are invoking prefix sums 2 recursively on a $1/2$ a n by 2

when i equal to 1, we are invoking the algorithm recursively on values $a_{n/2+1}$ to a_n . So, the two halves are solved independently and in parallel the corresponding results are stored in the b array. And then, for every element on the right side, there are $n/2$ elements on the left side.

So, for i varying from 1 to $n/2$, we do this in parallel. The content of $a_{n/2+i}$ will be enhanced by the content of $b_{n/2+i}$; $b_{n/2+i}$ is the leftmost element, rightmost element on the left side. This is added to every single element on the right side and then the array b is returned. So, this is the recursive specification of prefix sums 2. So, let us see what is the time complexity of the algorithm.

(Refer Slide Time: 36:04)

Prefix Sums 2

$$\begin{aligned}
 T(n) &= T(n/2) + c_1 \\
 &= T(n/4) + 2c_1 \\
 &= T(n/8) + 3c_1 \\
 &= T(n/2^k) + kc_1
 \end{aligned}$$

Time : $T(n) = T(n/2) + c_1 = O(\log n)$. Cost: $C(n) = 2C(n/2) + c_2n = O(n \log n)$.

$$\begin{aligned}
 &= T(1) + c_1 \log n \\
 &= O(\log n)
 \end{aligned}$$

We find that the time complexity is like this. When we are given an instance of size n , we form two instances of size $n/2$ each and solve these two instances independently and in parallel. And then combining the results, we spent another order 1 time. We take the rightmost element on the left side and add this to every single element on the right side.

If we use a CRAW PRAM, then this value can be read by every single processor on the right side simultaneously and the addition will take only order 1 time. Therefore, c_1 here is a constant. So, the recurrence relation is $T(n) = 2T(n/2) + c_1$. You can see that the solution for this recurrence relation is order $\log n$ that is because $T(n) = 2T(n/2) + c_1$. If you unroll this recurrence, we find that this is $T(n/4) + 2c_1$ plus c_1 which is $3c_1$. If we unroll once again, we have $T(n/8) + 3c_1$ plus c_1 . So, in

general if you unroll k times, we have T of n by 2^k plus $k C$. If you put k equal to $\log n$ in particular, this becomes T of 1 plus $C \log n$. But when there is only 1 element in the array, the array is returned as it is without doing anything.

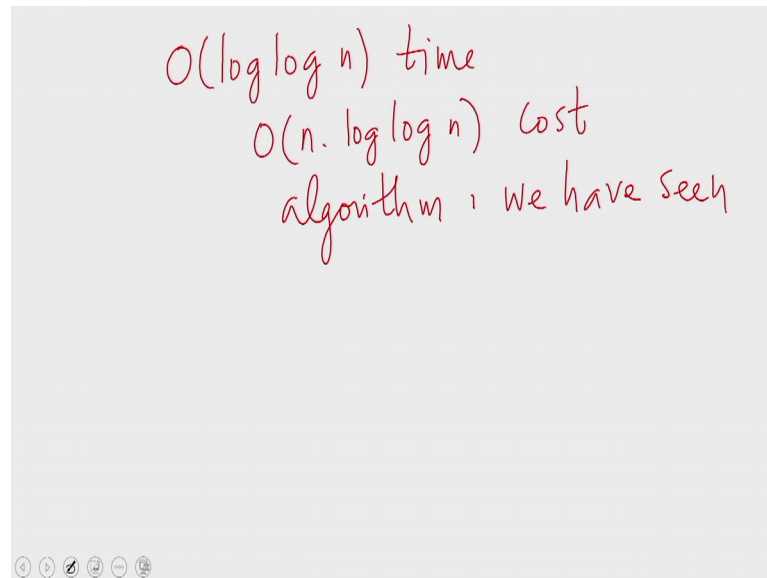
So, the algorithm would take order 1 time. Therefore, this is order of $\log n$ that is why the time complexity is order of $\log n$. But the cost of the algorithm is governed by this recurrence relation. The cost on an instance of size n is twice the cost of the invocations on size n by 2 each. That is we invoke the algorithm on the left side as well as the right side. The cost of both the invocations would count. So, we have a term two times c of n by 2. And then for combining them, we would require n by 2 processes on the right side to be active. All these processes will have to read the rightmost value on the left side and add this value to their own values.

So, there are n by 2 processors operating for one step which is a cost of order of n . Therefore, the recurrence relation is C of n equals 2 times C of n by 2 plus C of n . This is the recurrence relation that is familiar to you. This is identical to the recurrence relation for merge sort. So, you know that the solution for this recurrence relation is order of $n \log n$.

So, the algorithm runs in order of $n \log n$ cost in order of $\log n$ time. As you can see this is worse than the previous prefix sums algorithm that we have seen and the way the algorithm is specified branch scheduling principle will not apply directly. That is because the algorithm is a recursively specified one. But of course, it uses a very different algorithm design technique. It uses divide and conquer whereas, the previous prefix sums algorithm that we have seen was a balanced tree algorithm.

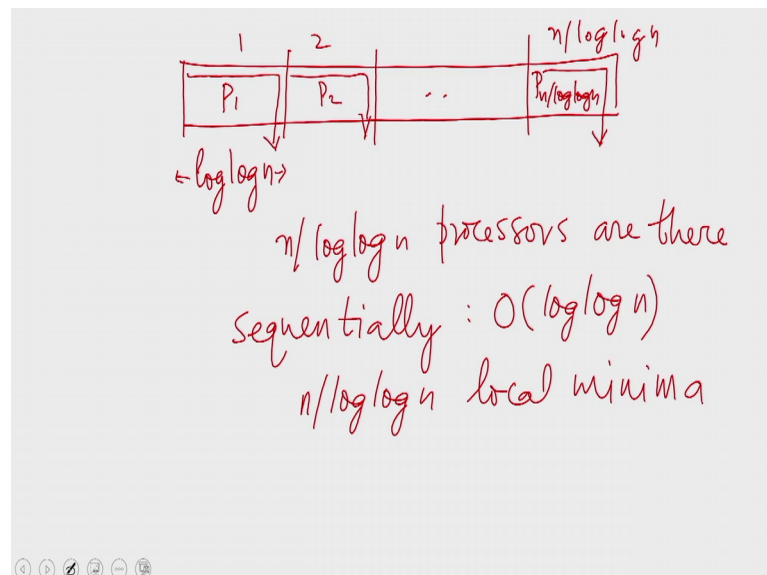
Now, can we apply branch scheduling principle on this algorithm? On the face of it, it cannot be. But every recursive algorithm can be converted into an iterative algorithm and if you convert this algorithm into an iterative one, you would find that branch scheduling principle will be applicable. And therefore, the cost of the algorithm can be reduced from order of $n \log n$ to order of n . That would be possible only on the iterative version of this algorithm. As another example of divide and conquer, we would design an optimal algorithm for finding the minimum of n numbers on common CRPW PRAM.

(Refer Slide Time: 40:17)



While ago we have seen an order double log in time algorithm for a cost of order of n times double log n . This is the second minimum algorithm we saw, using n processors that algorithm ran in order of double log n time. Let us say we want to convert this algorithm into an optimal algorithm. For doing this, what we do is this.

(Refer Slide Time: 41:01)



Given an array of size n where we have only n by double log n processors, we divide this array into segments of size double log n each. The size of each segment is double log n .

So, the number of segments would be n by $\log \log n$. And we assume we have n by $\log \log n$ processors. That is we have 1 processor per segment.

So, we can assign 1 processor per segment. Using these processors, sequentially, we can solve each segment. That is the first processor will scan the elements in the first segment sequentially and find out the minimum and output the minimum. The second processor does the same in parallel and simultaneously and so does every single processor.

Since, the size of the segment is $\log \log n$, this would take order of $\log \log n$ time. So, this is an order of since each segment is of a size $\log \log n$ with one processor per segment, the minimum within the segment can be found in order of $\log \log n$ time. Now, to find the global minimum all that we need to do is to find the minimum among these minimum, among the local minima.

So, we have n by $\log \log n$ local minima. Now, the solution should be obvious to you, there are n by $\log \log n$ local minima the minimum of which is the answer that we seek and we have n by $\log \log n$ processors. So, to solve this problem, we can invoke the previous algorithm that is the second minimum algorithm. So, this is our third minimum algorithm. So, to summarize this algorithm proceeds in this fashion. This algorithm assumes n by $\log \log n$ processors. It takes an array of size n , divides it into several segments of size $\log \log n$ each.

So, there are n by $\log \log n$ segments, we depute one processor to each segment, the processor will solve the segment sequentially. Since the segment has the size of $\log \log n$ the sequential solution will take order of $\log \log n$ time. So, out of each segment, we pick out one minimum.

Since there are n by $\log \log n$ segments we have a total of n by $\log \log n$ local minima. The minimum among these local minimum is the global minimum that we seek. But since, we have n by $\log \log n$ processors, we can solve this in order of $\log \log n$ time, the number of processors identical to the number of elements we have.

(Refer Slide Time: 44:38)

$$\begin{aligned} & \log \log \left(\frac{n}{\log \log n} \right) \\ &= \underline{O(\log \log n)} \\ & \text{cost} = O(n) \\ & \text{COMMON CRCW PRAM} \end{aligned}$$

Therefore, this can be solved now in strictly speaking double log of n by double log n . This is the problem size now. So, when it is solved like this, since n by double log n is less than n you can see this is order of double log n . So, invocation of minimum 2 will take order of double log n time. This is in addition to the time we spent within the segments. Within each segment, we spent order of double log n time. Since all the segments were solved in parallel, the total time taken is still order of double log n and we have used only n by double log and processors.

Therefore, this is an optimal algorithm; the cost of the algorithm is order n and the model that we have used is common CRCW PRAM. So, this is an optimal algorithm. You cannot be expecting to asymptotically improve on this. So, the problem of finding the minimum n numbers on common CRCW PRAM can be solved in order of double log n time for a cost of order n which is optimum ok. That is it from the 7th lecture. Hope to see you in the next lecture.

Thank you.