**Lecture on Computing the Maximum Element of an Array**

# 1 Introduction

In this lecture we illustrate some basic ideas in designing parallel algorithms by considering the example of computing the maximum of In this lecture we illustrate some basic ideas in designing parallel algorithms by considering the example of computing the maximum of an array of size $n$. These ideas would also be applicable to other operators such as computing the *sum*, the *product*, the *OR*, and *AND* of a boolean array.

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 1 requires you to extend these algorithms when entries may not be distinct.

# 2 A Binary Tree Based EREW Algorithm

Consider a binary tree such that its leaves correspond to the elements of the array. These are the nodes at level $k$ of the tree. Each of the nodes at level $k-1$ has 2 children. There are $2^{k-1}$ nodes at this level. We require that the label of any node be the maximum value of its children. To that end, we assign each node in the tree to a separate processor. Then, in one parallel step, we can compute labels of all nodes at level $k-1$. By repeating this parallel step we can compute the label of the root in $k = \log n$ steps. 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 $O(n \log n)$. 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
$T(n) = O(\log n)$
$W(n) = O(n)$
$C(n) = O(n)$.

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

# 3 A Fast Parallel Common CRCW Algorithm with suboptimal work

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.

for all $i$ parallel do:
    $isBiggest[i]$ := true;

for all $i, j$: $i \neq j$: parallel do:
    if ($A[j] > A[i]$) then
        $isBiggest[i]$ := false;

for all $i$: parallel do:
    if $isBiggest[i]$ then output (max = $A[i]$);

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

# 4 A Doubly Logarithmic height tree common CRCW Algorithm

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 $k = 2^h$. As a concrete example, let $n = 256 = 2^8$. Here $k = 8 = 2^3$ and $h = 3$. 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. More generally, we divide a group of size $n$ into $\sqrt{n}$ subgroups of size $\sqrt{n}$ as shown in Fig. 1.

How many levels of the tree are there? Assume that the original size $n = 2^{2^h}$. After $i$ splits, each segment contains $n_i = (2^{2^h})^{2^{-i}} = 2^{2^h 2^{-i}} = 2^{2^{h-i}}$. After $h$ levels, there will be $2^{2^0} = 2$ elements per segment, the tree cannot be split any further, so we can say the tree has height $h$. With a work-depth model that has depth $h$, if we can evaluate the max at each level of the tree in $O(1)$ time (we can do this with the all-pairs algorithm), then the time complexity is given by the relation of $h$ to $n$, namely:

$$T(n) = O(h) = O(\log \log n)$$

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:
$T(n) = O(\log \log n)$
$W(n) = O(n \log \log n)$

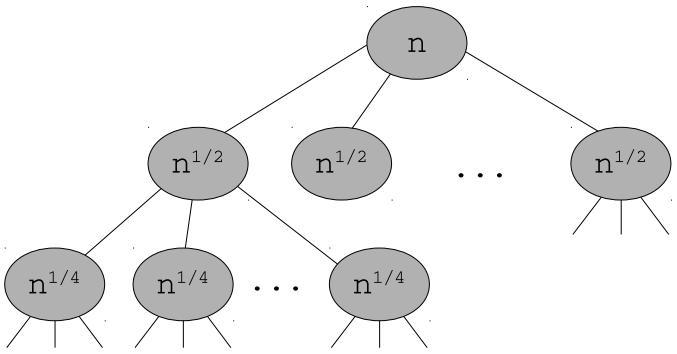


Figure 1: A Doubly Logarithmic Tree

For work complexity, notice that each level of the tree requires $O(n)$ work. Consider tree level $i$, where we will use the all-pairs algorithm locally at each tree node to compute the max of its children. We leave it as an exercise to show that the work done at any level is $O(n)$.

# 5 A Work-Optimal Common CRCW Algorithm with $O(\log \log n)$ time

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.

By applying the cascade technique to the doubly logarithmic tree, we can describe a very efficient work-optimal algorithm for the max problem. Let algorithm $\mathcal{A}$ be the standard $O(n)$ sequential algorithm, and algorithm $\mathcal{B}$ be the doubly logarithmic algorithm. We split the initial array into $N_{\text{seg}} = O(n/(\log\log n))$ segments of size $n_{\text{seg}} = O(\log\log n)$. Now the cascade technique proceeds as:

**step 1:** run $\mathcal{A}$ on each segment (in parallel)

Total time complexity $T(n) = O(\log\log n)$

Total work complexity $W(n) = O(n/(\log\log n)) * O(\log\log n) = O(n)$

**step 2:** run $\mathcal{B}$ on the segment results

Total time complexity $T(n) = O(\log\log(n/(\log\log n))) < O(\log\log n)$

Total work complexity $W(n) = O(n/(\log\log n) * \log\log(n/(\log\log n))) < O(n)$

**Total Complexity:** sum of step 1 and step 2 complexities

Total time complexity $T(n) = O(\log\log n) + O(\log\log n) = O(\log\log n)$

Total work complexity $W(n) = O(n) + O(n) = O(n)$

This is a **work-optimal**, efficient algorithm for the max problem. Since we use the all-pairs algorithm as part of the doubly logarithmic algorithm, it is written for a common-CRCW PRAM model.

Table 1 summarizes all the algorithms discussed in this lecture.

| 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 1: Time Complexity and Work Complexity of Computing Maximum

# 6 Exercises

1. 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.

2. Give work-optimal algorithms to compute *OR* of a boolean array on EREW and common CRCW PRAM.

3. 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.

4. Repeat the previous problem with the topology of a d-dimensional hypercube (i.e. $n = 2^d$ for some integer $d$).

5. Show that the work done at each level in the doubly logarithmic tree algorithm is $O(n)$.