CS273 Foundations of Parallel and & Network Computing	Spring 2003

Lecture 9 : 2.24.03

Lecturer: Christos H. Papadimitriou *Scribe: Krishnendu Chatterjee*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

9.1 Outline

In this lecture we will introduce a new model of computation called the **parallel random access machine (PRAM)** model. We discuss a few basic algorithms and learn some basic techniques of solving problems in the PRAM model.

9.2 PRAM Model

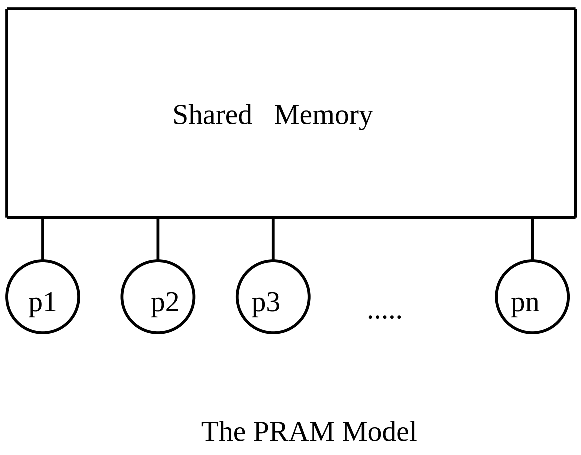

Figure 9.1: PRAM Model

In the RAM model of computation we have a memory block and a single processor with access to the memory and it does some computation. In the PRAM model of computation we have a single shared memory and there are number of processors with access to the memory and the processors work in parallel. The processors can read from and write to the shared memory. The Figure 9.1 shows a picture of a PRAM model. Each processor execute a step of the algorithm at the same time step. There can be several restrictions on the processors access to memory and depending on restrictions on memory access for data contention we have 3 different PRAM model.

- **exclusive read,exclusive write (EREW) :** At each time step of the algorithm two processors cannot access the same memory location to read or write.

- **concurrent read,exclusive write (CREW) :** At each time step of the algorithm two processors cannot write to the same memory location but several processors can read from the same memory location.
- **concurrent read,concurrent write (CRCW) :** At each time step of the algorithm several processors can read from and write to the same memory location. In this model we also need to specify what happens when several processors write to the same location but they write different data. A useful and reasonable restriction can be to make all processors write the same data to update a memory location. This is called common CRCW model.

9.3 Important complexity measures of PRAM Algorithms

In case of sequential algorithm we have time as an important measure of complexity. In this section we introduce the important complexity measures for PRAM algorithms.

- **The time complexity measure:** Time is the most important complexity measure for PRAM algorithms. We denote $T_1(n)$ as the sequential time complexity of a problem and $T(n)$ as the time complexity of a PRAM algorithm with as many processors as needed. Also we denote $T_p(n)$ as the time complexity of a PRAM algorithm when it can use at most p processors. We want $T(n) \ll T_1(n)$. In general we look for PRAM algorithm with *poly-logarithmic* time complexity with *polynomial* many processors.
- **The Work Complexity:** The next important measure of complexity in PRAM algorithm is the work complexity which is defined as follows:

  $$W(n) = \Sigma_{i=1}^{T(n)} P_i \text{ where } P_i \text{ is the number of processors working in the } i-th \text{ iteration}$$

  It is an important parameter as it is unrealistic to have infinitely many processors. We will soon see how the work complexity is related to the time complexity when we have a bounded number of processors.

9.3.1 The Basic Techniques

In case of sequential algorithms we have several techniques like Divide and Conquer approach, Dynamic Programming, Greedy approach to solve problems. In PRAM algorithms we will learn several basic techniques to solve problems which are as follows:

1. Brent's Principle
2. Binary Technique
3. Saturation
4. Optimalization
5. Pointer Jumping
6. Randomized Symmetry Breaking

In this lecture we shall see the application a few of the above techniques to solve a few problems in PRAM model.

9.4 PRAM Algorithms

We will find in this course that several sequential algorithms (e.g Matrix Multiplication) can be easily parallelized whereas for some problems we need an algorithm different from the sequential algorithm to parallelize (e.g. finding connected components in a graph) and there are some problems which are inherently sequential and cannot be parallelized (e.g. DFS of a graph or maximum flow). We now give as an example how the sequential algorithm of *Matrix Multiplication* can be parallelized.

Parallelizing Matrix Multiplication: It is easy to see that given two matrices every entry of the product matrix can be computed in parallel. Each entry can be computed with $O(n)$ work and in $O(log(n))$ parallel time using a binary tree to add n numbers with $O(n)$ work and in $O(log(n))$ time. Since there are n^2 entries and all can be computed in parallel, we have $T(n) = O(log(n))$ and $W(n) = O(n^3)$.

Brent's Principle

Lemma 9.1 *Given p processors we have $T_p(n) \leq T(n) + \frac{W(n)}{p}$*

Proof: Let at i-th iteration P_i processors be required for the PRAM algorithm with unbounded processor resource. The p processors can simulate the work of the P_i processors in $\lceil \frac{P_i}{P} \rceil$ time. Hence the total time required is

$$T_p(n) = \Sigma_{i=1}^{T(n)} \lceil \frac{P_i}{p} \rceil \;\leq\; \Sigma_{i=1}^{T(n)} \left(\frac{P_i}{p} + 1 \right) = T(n) + \frac{W(n)}{p}$$

Hence proved that $T_p(n) \leq T(n) + \frac{W(n)}{p}$. ■

Remark: In the above proof when we write $T_p(n) = \Sigma_{i=1}^{T(n)} \lceil \frac{P_i}{p} \rceil$ we assume there is a scheduling algorithm which schedules processors. In general it may require non-trivial amount of time and work to schedule the work of P_i processors among p processors but presently we are not concerned about it and ignore this issue.

9.5 Prefix Sum

We now describe an prefix sum algorithm with the following complexity measures:

$$T(n) = O(log(n)), \quad W(n) = O(n)$$

Before describing the algorithm we define what is meant by an work optimal algorithm.

Definition 9.2 [**Optimal Algorithm**]
A PRAM algorithm is called optimal algorithm *if*

$$T(n) = O(log^k(n)) \textit{ for some constant } k, W(n) = O(T_1(n))$$

In other words the work of the PRAM algorithm is of the order of the work (time) of the best known sequential algorithm and works in poly-logarithmic time.

It follows that the algorithm we will describe to compute prefix sum of a set of numbers given in an array is an optimal algorithm. We describe the algorithm below.

Algorithm 9.3 Prefix Sum (X)

Input : $X[1 \cdots n]$
Output: $S[j] = \Sigma_i^j X[i]$ *for* $j = 1, 2, \cdots n$
1. For $i = 1 \ldots \frac{n}{2}$ *pardo*
$\{\ y[i] = x_{2*i} + x_{2*i-1}\ \}$
2. Prefix Sum (Y) $\Rightarrow S'$
3. For $i = 1 \ldots n$ *pardo*
$\{$
odd $i : S[i] = S'[\frac{i-1}{2}] + x_i$
even $i : S[i] = S'[\frac{i}{2}]$
$\}$

The working of the algorithm is illustrated in Figure 9.2. In this algorithm essentially we build a binary tree (binary technique). Let us analyze the work and time complexity. Since in Step 1 and Step 3 the work is done parallely the time taken in both these steps in $O(1)$. Also the total work in Step 1 and Step 3 is $O(n)$. Hence we have the following recurrence

$$T(n) = T(\frac{n}{2}) + O(1) \ = O(log(n))$$

$$W(n) = W(\frac{n}{2}) + O(n) \ = O(n)$$

The work is of the order of the number of nodes in the binary tree. It follows from the above analysis that we have an optimal algorithm. It is also to be noted that the above algorithm is a EREW PRAM algorithm.

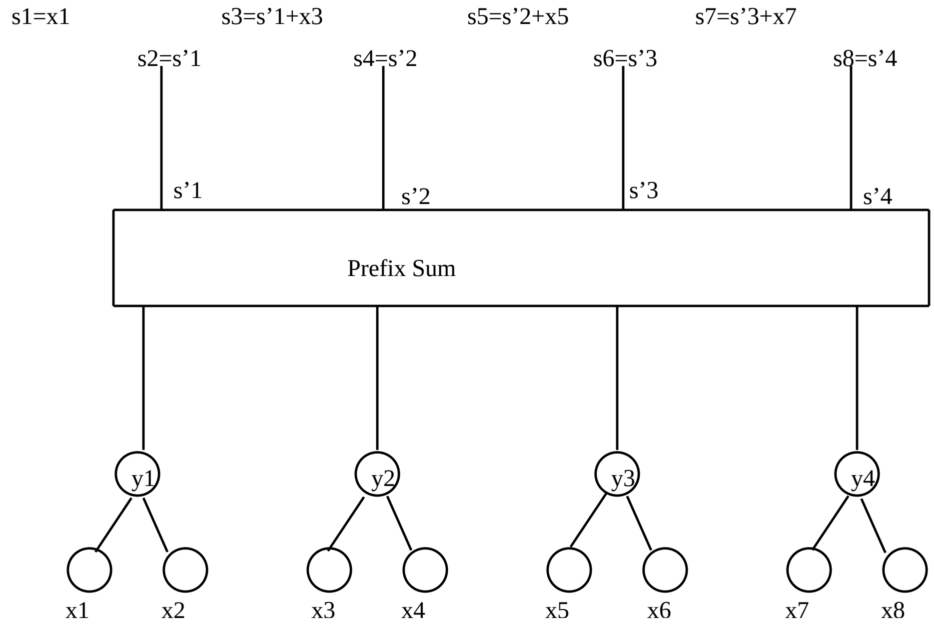

Figure 9.2: The Prefix Sum Algorithm Steps

9.6 Finding Maximum of a Set of numbers

In this section we define several algorithms to find maximum of a set of numbers. We first describe an optimal algorithm for the EREW model which works in $O(log(n))$ time. Then we will describe an optimal algorithm which works in $O(log\ log(n))$ time for the common CRCW model.

Algo1 : max1 Change the + operator of the Prefix Sum Algorithm with max operator. It follows from the previous section that *max1* is a EREW PRAM algorithm with

$$T(n) = O(log(n)), \quad W(n) = O(n)$$

We now describe an algorithm *max2* to find the maximum of a set of numbers given in an array in constant time. The algorithm will be a common CRCW algorithm. We assume without loss of generality that all the input numbers are distinct.

Algorithm 9.4 max2 (X)

Input : $X[1 \cdots n]$
Output: *maximum element of all* X

1. *For* $i = 1 \ldots n$ *pardo //[Initialize array A to all* 0*'s]*
   $A[i] = 0$
2. *For a Processor* $P(i, j)$ *pardo //[Fill up the* max *matrix]*
   {
   If $(x[i] > \; x[j])$
   $\quad max[i, j] = 1$
   Else
   $\quad max[i, j] = 0$
   }
3. *For a processor* $P(i, j)$ *pardo //[Using common CRCW model write* 1 *in* $A[i]$ *when* $X[i]$ *is maximum]*
   $A[i] = max[i, j]$
4. *For a processor* $P(i)$ *pardo //[output maximum]*
   If $(A[i] == 1)$ *output maximum* $= X[i]$.

The algorithm works as follows: There is a processor for every i, j and it compares $x[i]$ and $x[j]$ and sets the matrix max entry $max[i, j] = 1$ if $x[i]$ is greater than $x[j]$. After Step 2 only one row, say i-th row, of the matrix max will have all 1 where $x[i]$ is the maximum. The array A is initialized to all 0's in Step 1. In Step 3 all the processors in row i tries to write in the i-th entry of A. The Figure 9.3 shows that all the processors in i-th row of the max matrix writes in the i-th entry of array A. All elements of A was initialized to 0 and only one row of max has all 1's. Hence only one $A[i]$ will be 1 after Step 3 since we have the common CRCW model and the corresponding $x[i]$ is the maximum. Hence the output $maximum$ is the maximum element of X. For the above algorithm we have

$$T(n) = O(1) \quad W(n) = O(n^2)$$

This not a work optimal algorithm. It works in constant time in common CRCW PRAM model. We now describe an algorithm which has a better work complexity and worse time complexity.

Algorithm 9.5 max3 (X)

Input : $X[1 \cdots n]$
Output: *maximum element of all* X

1. *For* $i = 1 \ldots \sqrt{n}$ *pardo*
   $m[i] = max3(X[(i-1) \times \sqrt{n} + 1 \; \ldots i \times \sqrt{n}])$
2. $maximum = max2\{m_1, m_2, \ldots m_{\sqrt{n}}\}$.

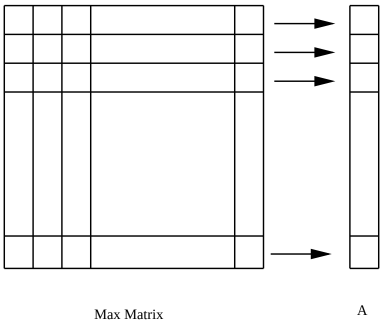

Figure 9.3: Finding maximum in a common CRCW Model

The algorithm breaks the n elements into $\sqrt{n}$ blocks of $\sqrt{n}$ elements each, recursively finds the maximum of this blocks and then find the maximum of the maximum's of $\sqrt{n}$ blocks using the *max2* algorithm. We now analyze the time and work complexity of the algorithm. It follows from the analysis of *max2* that it works in constant time and for $\sqrt{n}$ elements it does $O(n)$ work. Hence Step 2 of *max3* takes $O(1)$ time and does $O(n)$ work. Thus we have the following recurrence

$$T(n) = T(\sqrt{n}) + O(1) = O(loglog(n))$$

$$W(n) = \sqrt{n} \times W(\sqrt{n}) + O(n) = O(n \times loglog(n))$$

Now we will use a technique call *Optimalization* to obtain an optimal algorithm which works in $O(loglog(n))$ time.

Algorithm 9.6 max4 (X)

Input : $X[1 \cdots n]$
Output: *maximum element of all X*

1. *Run* max1 *for log log log*(n) *steps. (Now the array will have size* $\frac{n}{log\ log(n)}$*)*
2. *Run* max3 *on the resulting array*

Since Step 2 runs on a set of elements of size $O(\frac{n}{log\ log(n)})$ it runs in $O(log\ log(n))$ time and it does $O(n)$ work. Hence this algorithm does $O(n)$ work and runs in $O(log\ log(n))$ time.

9.7 List Ranking

Given a list of elements we want to rank every element in the list as shown in Figure 9.4. We now describe the algorithm to rank the elements of the list using the *Pointer Jumping Technique.* Every element in the list initially have a value *count* set to 1 and *pointer* pointing to the next element in the list.

Algorithm 9.7 List Ranking

Input : *A list of elements.*
Output: *A ranking of all the elements*

1. *Repeat for $log(n)$ steps pardo*
   $\{$
   $count[i] = count[i] + count[pointer[i]]$
   $pointer[i] = pointer[pointer[i]]$
   $\}$

In the above algorithm we have a processor for every node in the list. The time complexity is $O(log(n))$ since Step 1 is repeated $O(log(n))$ time and each Step takes constant time. Each processor works for $log(n)$ steps. Hence the work complexity of this algorithm is $O(n \times log(n))$. A working of the algorithm is shown on an example in Figures 9.5 , 9.6 and 9.7. We could do the prefix sum and hence ranking of elements in an array using $O(n)$ work and $O(log(n))$ time using an array representation. The advantage of using an array is we know in advance which elements are even and which are odd. We do not have this information using a list representation. In the next lecture we will see how using the technique of *Randomized Symmetry Breaking* we can perform list ranking in $O(log(n))$ time and $O(n)$ work.

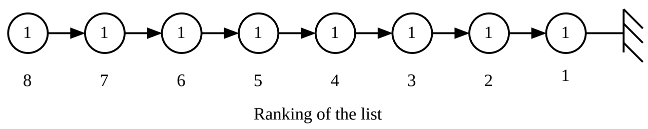

Figure 9.4: List Ranking

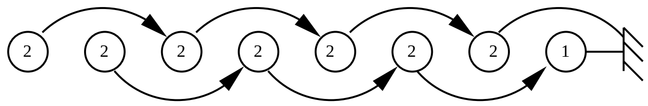

Figure 9.5: Step 1 of the Algorithm

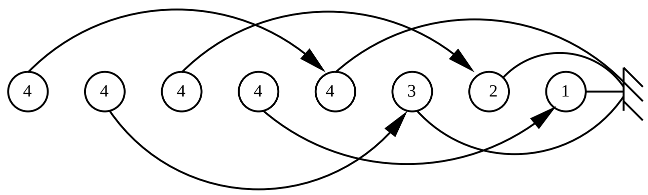

Figure 9.6: Step 2 of the Algorithm

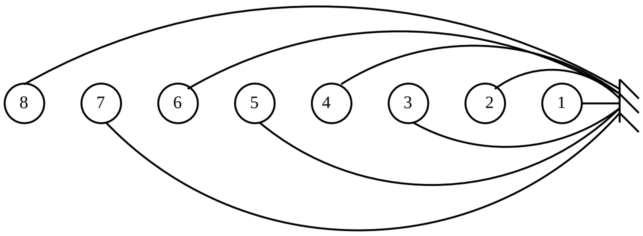

Figure 9.7: Step 3 of the Algorithm