Lecture 11

1 Overview

Today, we will discuss developing algorithms for a basic model for parallel computing the Parallel Random Access Machine (PRAM) model.

2 Definition.

The PRAM is a generalization of the RAM model where we have a single processor and a memory block with unit time access to any memory location.

A PRAM consists of an arbitrary number of processors and memory, where each processor has unit time access to any memory location. That is, computation proceeds according to a single clock where each processor executes a single instruction in this clock cycle including memory read/writes. This can be contrasted with, for example, the BSP model where access to local memory is faster than communicating with another processor. Here, one can essentially communicate for "free", i.e., one processor writes and another reads the memory location. Of course, this immediately creates complications. For example, what happens when two processors attempt to write to the same location.

There are various versions of the PRAM model which addresses this question. For example, some variants are

1. **Exclusive Read/ Exclusive Write (EREW)**. Only one processor can write to or read from a memory location in any one step. (The program is simply illegal if the programmer makes the mistake of attempting to do so.) This is the easiest to implement (i.e., the "routing" that needs to be done in each step is a permutation routing, in some sense.)

2. **Concurrent Read/Exclusive Write (CREW)**. Reads are concurrent Here, more than one processor can read (concurrent read), but only one write in any step. This is harder to implement, many messages are routed to the same memory location.

3. **Concurrent Read/Exclusive Write (CRCW)**. Reads and writes are concurrent. Here, one may refine the model further to say what happens when more than one processor writes to a memory location. For example, it could result in a single value, the sum of the values, a garbage value, an arbitrary value. I believe all of these have been studied. This is not harder to implement than CREW since the basic problem is many messages being routing to the same memory location.

3 Complexity Measures.

For sequential algorithms time is the basic measure of complexity (though space is also sometimes studied.) Here, there are two that come to mind immediately.

1. Time. Here, we denote the time for a paralell algorithm on an arbitrary number of processors by $T(n)$ and of one p processors by $T_p(n)$ and the sequential complexity by $T(n)$ for a problem of "size" n. Of course, this may be a bit loose, since the bounds may be acheived by different algorithms. A notion that a problem can be parallelized well is that the running time is polylogarithmic in the size of the problem given a polynomial number of processors.

2. Work. The next most important measure of complexity in PRAM algorithms is the work measure which is defined as follows.

$$\sum_{i=1} T(n)P(i),$$

   where $P(i)$ processors are working in the i-th iteration. This is important since assuming an arbitrary number of processors is a bit suspect. We will use this measure to relate $T_p(n)$ and $T(n)$.

4 The basic techniques.

In sequential algorithms we have several basic techniques: Divide and Conquer, Greedy, Dynamic Programming. The following techniques used for PRAM algorithm design.

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

In this lecture we apply some of these techniques to produce PRAM algorithms.

5 PRAM algorithms

For some problems, one can easily parallelize the sequential algorithms (e.g. Matrix Multiplicaiton), while some need new algorithms (e.g. connected components), and some are inherently difficulat to parallelize (DFS of a graph or maximum flow.)

We begin with matrix multiplication. Here one can compute $C = A \times B$, by computing each entry $c_{i,j}$ with $O(n)$ work and $O(\log n)$ parallel time by computing the dot produce

$a_i \cdot b_j$ using a binary tree. Since there are n^2 entries the total work is $O(n^3)$ or the same as the algorithm that we parallized. That is, $T(n) = O(\log n)$ and $W(n) = O(n^3)$.

What if we have fewer than n^2 processors, say p. Do we need to design an algorithm that specifically runs on p processors? Perhaps not.

LEMMA 1
Given p processors, $T_p(n) \leq T(n) + W(n)/p$.

PROOF:
If $P(i)$ processors are used in step i, then p processors can simulate the step in time ⌈P(i)/i⌉. Hence the total time of this simulation is bounded by

$$\sum_{i=1}^{T(n)} \lceil \frac{P(i)}{p} \rceil \leq \sum_{i=1}^{T(n)} (\frac{P_i}{p} + 1) = T(n) + \frac{W(n)}{p}.$$

That is, $T_p(n) \leq T(n) + \frac{W(n)}{p}$.
□

6 Prefix Sum

Next, we consider the very useful (for the PRAM) subroutine of prefix sum. This is the problem of for an vector a, computing b, where $b_i = \sum j < ia_i$. We develop an optimal algorithm for this problem. We consider the following program.

PrefixSum(x) Input: x
Output: S prefix sum array of x

1. If $n = 1$, return x.
2. For $i = 1, \ldots n/2$ pardo
   $\{y[i] = x[i] + x[2 * i + 1]\}$
3. $S' = PrefixSum(y)$
4. For $i = 1, \ldots n$ pardo {
   odd $i : S[i] = S'[i - 1/2] + x_i$
   $even$i: S[i] = S'[i-1/2]
   }

A recurrence (omitting the base case) for the running time is $T(n) \leq T(n/2) + O(1)$, which implies that $T(n) = O(\log n)$. For the work, the recurrence is $W(n) \leq W(n/2) + O(n)$ (again ommitting the base) which solves to $W(n) = O(n)$.

Notice that this algorithm can be implemented on a EREW PRAM.

7 Finding the maximum of a set of numbers

Here, we will first describe an algorithm for finding the maximum of a set of n numbers in $O(\log n)$ for the EREW PRAM. Then we will illustrate some algorithmic techniques for the CRCW PRAM which will give us an $O(\log \log n)$ algorithm for this problem.

An EREW algorithm can be obtained by replacing the $+$ in the prefix sum algorithm with the **max** operation, and checking the last element of the array that is output. This runs in $O(\log n)$ time does $O(n)$ work. (We may refer to this algorithm as Max1.)

We will produce an algorithm for the COMMON CRCW model, where a write only succeeds if all the processors write the same value to a location. (We assume that the numbers are all distinct. This can be fixed by changing the comparison function slightly.)

Max2(x) Input: X
Output: max element of non-negative numbers in X

1. For $i = 1, \ldots n$ pardo
   $\{A[i] = 0\}$
2. For a processor $P(i, j)$ pardo
   {
   If $x[i] > x[j]$
   bigger$[i, j] = 1$
   Else
   bigger$[i, j] = 0$
   }
3. For a processor $P(i, j)$ pardo
   {
   $A[i] =$ bigger(i, j).
   }
4. For a processor $P(i)$ pardo
   {
   If $A[i] == 1$ output $x[i]$.
   }

The algorithm compares all elements in step 2, setting *bigger*$[i, j]$ to one if i is larger than j. In step 3, $A[i]$ is set to one only if it is found to be bigger than every other number in step 2 (this is since common only writes 1 if all the writes have the same value.) Only the maximum could have been bigger every time, so in the final step we only output the maximum element.

Here, the time is $O(1)$, but the work is $O(n^2)$. That is, the algorithm does far mor work than $O(n)$, and is not work optimal. We now describe a slower algorithm that has better work complexity.

Here is a more efficient algorithm.

Max3(x) Input: X
Output: max element of non-negative numbers in X

1. For $i = 1, \ldots \sqrt{n}$ pardo
   $\{m[i] = \text{Max3 } (X[i\sqrt{n}+1, \ldots, i+1\sqrt{n}\}$

2. maximum = Max2 (m)

This is a recursive algorithm where each subproblem has size $\sqrt{n}$, and thus, we can bound the time and work with the following recurrence.

$$T(n) = T(\sqrt{n}) + O(1) = O(\log\log n).$$
$$W(n) = \sqrt{n}W(\sqrt{n}) + O(n) = O(n\log\log n).$$

An improvement, but not yet optimal. Now we introduce a technique called *Optimalization.*

Max4(x) Input: X
Output: max element of non-negative numbers in X

1. Run algorithm Max1 for $\log\log\log n$ steps. (Now we will have an array one of whose element is the maximum that has size $\frac{n}{\log\log n}$.
2. Run Max3 on the resulting array.

Voila. Now step 2 runs on an array of size $n/\log\log n$, it does $O(n)$ total work. The total time is $O(\log\log n)$.

8 List Ranking.

In this section, we wish to rank every element of a list. That is, compute the order starting from the end of the list. The data structure is an array of pointers to the next element. The end of the list contains a nil pointer.

ListRank(x) Input: list arrays: *pointer*
Output: rank array with distance from end: *count*

1. For $i = 1, \ldots n$ pardo
   $\{count[i] = 1$

2. Repeat for $\log n$ steps. $\{$
   $count[i] = count[i] + count[pointer[i]]$
   If $pointer[i]! = nil$ then $pointer[i] = pointer[pointer[i]]$
   $\}$

We see this algorithm runs in time $O(\log n)$ and does $O(n \log n)$ work by inspecting the loops indices. The algorithm maintains the invariant that $count[i]$ contains the number of elements in the list from i to the element specified by your current pointer (including element i.) This holds at the beginning, and in each step the loop enforces this condition since we assign a pointer that jumps over two pointers (the current and the next pointer) and assign the count value to the sum of the count value of those jumps. At termination, every pointer points to *nil*, since in each iteration (other than the one that assigns it to nil) a pointer doubles the distance down the list that it points to.