Lecture 12

1 Overview

Today we will discuss a randomized algorithm for list ranking that is more efficient than pointer jumping, deterministic coloring, and an algorithm for tree evaluation.

2 Randomized list ranking.

Recall, the parallel prefix algorithm. It paired elements, combined them, and recursed on the results of pairing those elements (then it could compute the final prefix by using the result of the recursive call.) In that case, the pairing was easy since one could group successive pairs, where the first was even indexed and the second was odd indexed.

The idea, in some sense is that the even elements took control of the odd elements. This worked since there was no contention. No "taken" over element would take over something else.

Here, we obtain this condition by producing an (large) independent set in the list at random. That is, a set of list elements such that for each element any element connected to them is not in the set. A recursive algorithm can them have these elements bow out, that their predecessors can take over for them, i.e., include their value and then point to the next element. The total number of iterations, will be the time for all the nodes to drop out.

Consider the following, "independent set" algorithm. For each list element, we flip a coin. If it is heads and the predecessor is tails, include it in the set. Only nonadjacent elements will be included since otherwise consider the second element in a pair, it must have flipped a heads whereas its predecessor must have flipped a tail, which contradicts the assumption that its predecessor was chosen.

Now, what is the probability that a particular element survives t steps. In each step, it is eliminate with probability $1/4$. Thus, it survives with probability at most $3/4$ in one iteration or $(3/4)^t$ for t iterations. Thus when $t \geq 2\log_{3/4} n$ it survived with probability at most $1/n^2$ after this many iterations. Thus, none will survive with probability at least $1 - 1/n$.

Thus, the running time is $O(\log n)$ with high probability. Just like pointer jumping!

Well, perhaps we do less work on average. The work on a particular element is proportional to the number of iterations it survives. The expected value of this quantity is

$$\sum_{i \geq 0} i(3/4)^i,$$

which is $O(1)$.

Thus, the total expected work is $O(n)$. This is better than pointer jumping.

3 A deterministic algorithm for coloring.

Notice that in the previous section, the only place one used randomness is to select a large set of non-adjacent (independent) elements. Here, we solve a generalized version of finding a large independent set. In particular, we show that we can color the list using only six colors so that no two elements with the same color are adjacent. That is, we prove the following lemma.

LEMMA 1
There is a deterministic algorithm for six coloring a list in $O(\log^* n)$ time.

We start with the assumption that each list element has a unique name. For example, the memory location that holds it. Actually, we assume that it is the array location that holds it, since the list is held in an array. (For applications, this is easy to enforce as we will see.)

Now, this is an n-coloring of the list. We now produce a $\log n + 1$-coloring of a list in one step as follows.

For an element, recolor it with $2k + b$ where k is the least significant bit position where the element differs from its next element, and b is the value of that bit for this element.

This is a legal coloring, since for two adjacent nodes. Either the number k differs, or if it is the same, the bit b must differ by construction.

The number of colors here is $2 \log n$ where n is the original number of colors. This doesn't reduce when $n \leq 6$, so at this point, we stop.

The number of iteration is $O(\log^* n)$ since in each iteration the number of colors drops by essentially taking the log of the old number of colors.

Now, we can use this coloring to do list ranking. Instead of using the randomized method of the previous section to find a set of nodes to eliminate, we pick the largest color class.

This running time for this method is $O(\log n \log^* n)$ and the total work is $O(n \log^* n)$. So, it is a bit less efficient. One can make an efficient deterministic algorithm using versions of this idea.

4 Applications of List Ranking

List ranking can be used as a fast implementation of walks over data structures that we use naturally in sequential computing. For example, considering numbering the leaves of a binary tree, with respect to some ordering of the children; say each node has a left and right child.

We can form a list from the tree data structure, by making three list elements for each node; left, right, and bottom. We can connect them in the natural order, e.g., the left node points to the left node of the left child, the right node points to the bottom node of the parent if it is a left child and to the right node of the parent if it is a right child of the parent. A leaf consists of a single node.

One can then number the leaves by associating a value of 0 with the non leaf list elements and 1 with the leaf elements and computing a list prefix ranking.

Question: Show how to compute the height of a node (max distance to a leaf) in the tree using list ranking.

5 Evaluating Expression Trees.

Say, we have an expression,

$$(a_1 + a_2) * (a_3 * a_4 + 1) + (a_7 * a_6 + a_8).$$

This can be viewed as a tree, where each internal node is associated with an operator; $+, \times, -$. We would like to evaluate these quickly in parallel.

One thing, one can do is to just evaluate all the "cherries." That is, nodes both of whose children are leaves. But, there are easy examples, where the total time is $\Omega(n)$ if we do this; consider a path with leaves hanging off the path.

Thus, we could do something more clever. We develop an operation called "rake" which can eliminate any leaf. We will then show that we can eliminate all the odd numbered leaves in a constant number of steps. Then, by recursing, we can rake the entire tree in $O(\log n)$ steps.

To define the rake operation, we generalize the operation at the node a bit. (Well, perhaps more than a bit.) Each tree node is now represented by a quadruple (a, b, c, d); and the associated operation on the incoming values x and y is defined to be $axy + bx + cy + d$. We can represent a multiply tree node with the quadruple $(1, 0, 0, 0)$ or an addition node with $(0, 1, 1, 0)$.

Now, we consider a tree node with tuple (a', b', c', d') and children that consist of right child leaf with value α and a left child tree node (a, b, c, d) with children T_1 and T_2.

The rake operation produces a new tree node with children T_1 and T_2, with tuple

$$\begin{aligned}
a'' &= aa'\alpha + ab' \\
b'' &= ba'\alpha + bb' \\
c'' &= ca'\alpha + cb' \\
d'' &= da'\alpha + db' + c'\alpha + d'
\end{aligned}$$

The cases change a bit when T_1 or T_2 are null.

We wish to do the rake operations in parallel. The key is that the operations should not overlap. One can see that raking the odd numbered children that are left children and then raking the odd numbered children that are right children will work.

Raking the entire tree in $O(\log n)$ time is slightly complicated since one cannot compute a prefix in every iteration. But, after raking all the odd leaves one then is left with only even leaves. Thus, one can renumber by dividing by two and raking the odd leaves. (One thing, though, if an odd leaf has a common parent with an even leaf, the parent sucks in both leaves, and "becomes" the even leaf.)

Question 2: Show that given a 6 colored list of n elements, how it can be 3 colored in time $O(1)$ in parallel using $O(n)$ processors.

Question 3: Suppose you are given a list of n elements that are k colored. Show that it can be 3 colored in time $O(k + n/p + \log p)$ with p processors.

Question 4: Show that a list can be deteriministically ranked using p processors in time $O(\frac{n}{p} + \log n \log\log n)$. (You have to do a couple of phases.