15-750: List Ranking

Scribe: Sriram Somanchi

March 28 2011

1 List Ranking

Recall that the input to the List ranking problem is an array of handles to a collection of weighted nodes that form a single linked list. The output is the same list with each node marked with the sum of weights of all nodes in the list from its successor to the end of the list. For example, if the list is

$$3 \rightarrow 5 \rightarrow 2 \rightarrow 0 \rightarrow 2 \rightarrow 9,$$

then the output is

$$18 \rightarrow 13 \rightarrow 11 \rightarrow 11 \rightarrow 9 \rightarrow 0.$$

In the last class, we have seen Wyllie's algorithm to compute the list ranking which takes $O(n \log n)$ work and $O(\log n)$ depth for a list of length n. We are doing a lot of additional work in Wyllie's algorithm, which can be avoided. One method is to *contract* the list by a constant fraction, recursively compute the list ranking of the contracted list and use it to recompute list ranking for the original list. If we can generate a contracted list and can build the list ranking for the original list from the contracted list using linear work and polylogarithmic time, we would have a linear work and polylogarithmic time algorithm for the list ranking problem. In this class, we will see a randomized algorithm to contract lists in parallel by finding independent sets in parallel. Also we see weighted list ranking where there are weights (including zero) on each node in the list and the rank is sum of weights of all the nodes from the current node to the end of the list. Normal list ranking is a special case where each node is given a weight of 1.

1.1 Independent Set

We generate a smaller list to compute the list ranking, by generating an independent set and skip the nodes in the list.

1.2 Parallel Selection of Independent Set

We employ the following steps to select the nodes which are in the independent set and skipping over the nodes in the independent set.

1. Each node tosses a fair coin

2. For each node

(a) If the node is head and successor is tail, mark the successor for inclusion in the independent set.

(b) If the node is tail do nothing.

With these rules we can apply the method in parallel on each node. Also note that no two adjacent nodes are in the independent set and hence no two adjacent nodes are ever skipped over. This means that this algorithm is race free.

A node is in the independent set if it's predecessor tosses a head and the node tosses a tail. The probability of this happening is 1/4 (except when the node we are interested is the head of the list).

1.3 The algorithm

We now use the above parallel algorithm for computing an independent set to generate a list ranking. Note that in a list, the node at the end of the list points to itself, i.e., $succ(v) = v$ if v is the tail of the list.

- Given a list L with weights on nodes, if length of L is two and the list is $u \to v$, the rank of u is $w_L(v)$ and the rank of v is 0, and we are done. Else, first generate an independent set.
- Generate a new list L' by marking the nodes that are to be included in the shortened list and doing a pack operation. If node $v' = succ(v)$ and v' is in the independent set, v skips over $succ(v)$ to point to $succ(succ(v))$ in L'. Also, assign weight $w_L(v) + w_L(succ(v))$ to v in L' ($w_{L'}(v)$).
- Recursively generate a list ranking R' of the contracted list L'. We will use it to generate list rank R of list L.
- The list rank of a node v_i in L that has been selected in to L' is the same as its rank in R' ($w_R(v_i) = w_{R'}(v_i)$). The rank of an eliminated node v is $w_R(v) = w_L(v) + w_{R'}(succ(v))$.

1.3.1 Work and Depth of the algorithm

Note that in each round of the recursion, a node (which is not the head of the list) survive with probability 3/4, and the algorithm does $O(1)$ per node (say C) in each round. Therefore, expected work for a node added over all rounds is $C(1 + (3/4) + (3/4)^2 + \dots) = 4C$. Adding this up over n nodes, the expected work if $4Cn = O(n)$.

Further, we bound the number of rounds the recursion has to run, which bounds the depth. We will show that with in $k = 2\log_{\frac{4}{3}} n$ rounds, the probability that the list becomes zero is at least $1 - \frac{1}{n}$. And hence with high probability, we can say that we run for $O(\log n)$ rounds. Let X_i^k be the event that node i survives after k rounds of recursion. We have

$$Pr[X_i^k] = \left(\frac{3}{4}\right)^k$$

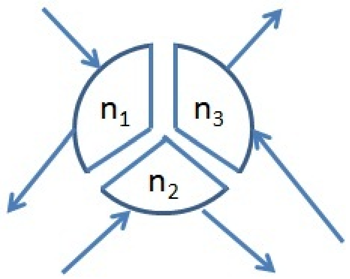

Figure 1: Node splitting to convert a tree to a linked list

Since $\left(\frac{3}{4}\right)^k \leq \frac{1}{n^2}$ for $k \leq 2\log_{\frac{4}{3}} n$, we have

$$\begin{aligned}
Pr[\text{At least one nonhead node surives after } k \text{ rounds}] &= Pr[X_1^k \cup X_2^k \cup \ldots \cup X_n^k] \\
&\leq Pr[X_1^k] + Pr[X_2^k] + \ldots + Pr[X_n^k] \\
&\leq \frac{1}{n^2} + \frac{1}{n^2} + \ldots + \frac{1}{n^2} \\
&= \frac{1}{n}
\end{aligned}$$

Therefore, after running $O(\log_{\frac{4}{3}} n)$ the probability the list would have gone down to size two (and the recursion stops) with probability $\geq 1 - \frac{1}{n}$.

2 Application of List ranking

2.1 Ranking of nodes in the tree

The list ranking explained above is weighted suffix ranking. We can do a similar weighted prefix ranking and apply that to rank the nodes in a tree. In order to rank the nodes of a tree we convert it to a chain. We split each node n into three nodes n_1, n_2, n_3, as shown in Figure 1 and form a chain as follows. The link coming from the parent is linked as incoming to n_1 and then n_1 is linked outgoing to the left child. If there is no left child then it is linked to n_2 of the same node. The link coming from the left child node is linked as in coming to n_2 and outgoing link from n_2 is linked to the right child. If there is no right child it is linked as incoming to node n_3. Otherwise, the link coming from the right child is linked as incoming to node n_3 and then outgoing from n_3 is linked to the parent node. For the root node n_1 does not have incoming node and n_3 outgoing link is marked as end of list. In this way we form a linked list of the nodes of the tree.

2.1.1 Pre-order ranking

In the pre-order ranking the node gets ranked first and then the left child and the right child. Figure 2 shows the pre-order ranking. For each node n as explained earlier, we split into n_1, n_2, n_3.

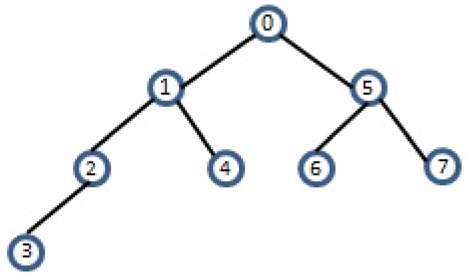

Figure 2: Pre-order ranking of nodes in a tree

In this chain we give a weight of 1 to every node of type n_1 and compute the prefix ranking. The ranking of the nodes in the tree is ranking given to its split of type n_1.

2.1.2 In-order ranking

In order to obtain the in-order ranking, we give for each node of type n_2 a weight of 1, and the rank of the node is the prefix ranking of the node of type n_2 in the list.