We consider (i) the list-suffix problem which is a generalization of the list ranking problem, (ii) the expression-tree evaluation problem which uses list-ranking.

The list suffix problem is similar to the vector suffix problem, except that the input is presented as a list. In particular, given a list of numerical values, we must compute another list such that the the ith element in the new list is the sum of input list elements i through the last element. The list ranking problem is a special case when the values are all 1.

1 Wyllie's Algorithm: n processors, $O(\log n)$ time

The input is a linked list of n elements: two arrays Value(i) and Next(i) of size n, respectively holding the values of the elements, and pointer to the next element of the list. If the last element of the list is stored in Value(j) and Next(j), then we have Next(j)=NIL. The program for processor i consists of repeating the following basic step:

If Next(i) $\neq$ NIL, then
  Value(i) = Value(i) + Value(Next(i))
  Next(i) = Next(Next(i))

After t steps, we can show that either (i) Next(i) points to the 2^tth element in the list from element with index i and Value(i) is equal to the sum of the values in the list from i (inclusive) down to Next(i) (exclusive); or (ii) Next(i) points to NIL and Value(i) is equal to the required suffix. Thus for every element the suffix will be computed in $\log n$ steps on a EREW PRAM. (Can you argue that at each step each processor accesses a distinct memory location, i.e. this is a valid EREW program?)

2 Efficient Randomized Algorithm

The algorithm uses $p = n/\log n$ processors, and works as follows.

1. Select a set of list elements that are non adjacent.
2. *Eliminate* the selected elements from the list.
3. Repeat the first two steps until only one element remains, in which case the prefix for the one element list problem is trivially the value in the element.
4. Fill in values for the elements eliminated in the preceding steps, in the reverse order of their elimination.

The key question is the first step: how to select a (large) set of non-adjacent elements? This is done as follows. Each processor is assigned $\log n$ elements which it is expected to eliminate during the course of the algorithm. This assignment is done very simply: processor i is given elements stored in positions $i \log p$ through $(i+1)\log p - 1$. We will refer to the elements assigned to each processor as being in its queue. The step of selecting elements for elimination itself happens in 3 substeps: (i) Each processor marks the element at the head of it's queue as a *candidate*. (ii) Each processor flips a coin and stores that along with its candidate. (iii) A candidate is rejected if either its own coin is TAIL, or it so happens that the preceding element in the list is also candidate having a coin HEAD. In other words, a candidate is selected for elimination if its coin is a HEAD, and either the previous element in the list was not a candidate or had a TAIL if it was a candidate.

Theorem 1 *After $O(\log n)$ steps each processor finishes all its elements with high probability.*

Proof: Let T_i denote the number of iterations (of steps 1 and 2) needed by processor i to elimate all its elements. Then we need to show that $T = \max_i T_i = O(\log n)$ with high probability. Thus T is a parallel composition of at most a polynomial number of random variables T_i (in fact $n/\log n$ variables), and so it suffices to prove for every i that $T_i = O(\log n)$ w.h.p.

Consider any fixed i. We will estimate $\Pr[T_i > C \log n]$. This is simply the probability that processor i is unsuccessful at least $(C-1)\log n$ times in $C \log n$ attempts. This would be easy to estimate if success/failure in consecutive attempts were independent; however this is not so: if a processor fails to eliminate in a given step, it is somewhat more likely to fail in the next step as well. However, the amount of correlation between successive events is not large, as we will see.

For the purpose of analysis, assume that in each iteration a coin is flipped for each element in the entire list, not just for the elements that are candidates for elimination. Say a processor is *unlucky* in a given attempt if it either gets a TAIL for its element, or the element preceding it in the list gets a HEAD. Else it is said to be lucky. Whenever a processor cannot eliminate its element, it must have been unlucky (though the converse is not true). Since coins are flipped afresh for each iteration, *whether or not a processor is lucky in a given attempt is independent of its fortunes in other attempts.*

For any iteration the probability of processor i being unlucky is $3/4$. Thus the probability of being unlucky at least $(C-1)\log n$ times is atmost

$$\binom{C\log n}{(C-1)\log n}\left(\frac{3}{4}\right)^{(C-1)\log n} = \binom{C\log n}{\log n}\left(\frac{3}{4}\right)^{(C-1)\log n} \leq \left(Ce\left(\frac{3}{4}\right)^{C-1}\right)^{\log n}$$

using $\binom{n}{r} \leq \left(\frac{ne}{r}\right)^r$. But by choosing C large enough $Ce\left(\frac{3}{4}\right)^{C-1}$ can be made arbitrarily small, and the whole expression can be made smaller than n^{-k} for any k. ■

3 Deterministic Algorithm Using Coloring

We can decide which nodes of the list to eliminate at each step by making use of a coloring on the nodes. A valid coloring has the property that no two adjacent nodes are colored the same. Therefore, we can eliminate all nodes of a given color in one step.

Theorem 2 *An n element list can be six colored in $\log^* n$ steps using n processors.*

Initially, set Color$(i) = i$. In each phase, we will recolor each node. Suppose c_1 is the color of node i and c_2 is the color of node Next(i). Scan the binary representations of c_1 and c_2 from the left. Let k be the index where they first differ. The indices are numbered from the right, with the least significant bit having index 0. Let b be that bit in c_1. Recolor node i so that Color$(i) = 2k + b$.

Example: $\begin{array}{ccccc} c_1 & = & 41 & = & 0101001 \\ c_2 & = & 105 & = & 1101001 \end{array}$ In this case, $k = 6$, $b = 0$, and the new value of Color(i) is 12.

We must show that this procedure gives a valid coloring. That is, that no two adjacent nodes have the same color. Suppose that the colors, c_1 and c_2, for two adjacent nodes are not equal at the beginning of a phase. We will show that there is a contradiction if we assume that the colors, c_1' and c_2', for the two adjacent nodes are equal at the end of the phase. $c_1' = 2k + b$ for some unique k and b. The kth bit of c_1 is b. Similarly, the kth bit of c_2 must be b. By definition of k however, these bits must be different. This is a contradiction.

If n is the number of colors initially then $2k + b$ is always $\leq 2\lceil \log n \rceil$. Repeating this step brings us down to constant size in O$(\log^* n)$ steps. For $n = 6$, $2\lceil \log n \rceil = 6$ so we can't get any smaller.

By a more complex procedure the coloring can be done in $O(\log n)$ time using $n/\log n$ processors.

Theorem 3 *List prefix can be done in $\frac{n}{p}\log^* n$ steps with p processors where $p \leq \frac{n}{\log n \log\log n}$.*

Proof: Assign elements $1, \ldots, \frac{n}{p}$ to processor 1. Assign elements $\frac{n}{p}+1, \ldots, 2\frac{n}{p}$ to processor 2, etc.

Algorithm:

1. Six color the list. Takes $O(\frac{n}{p}\log^* n)$ time.

2. Determine most frequently used color. Takes $O(\frac{n}{p} + \log p)$ time.

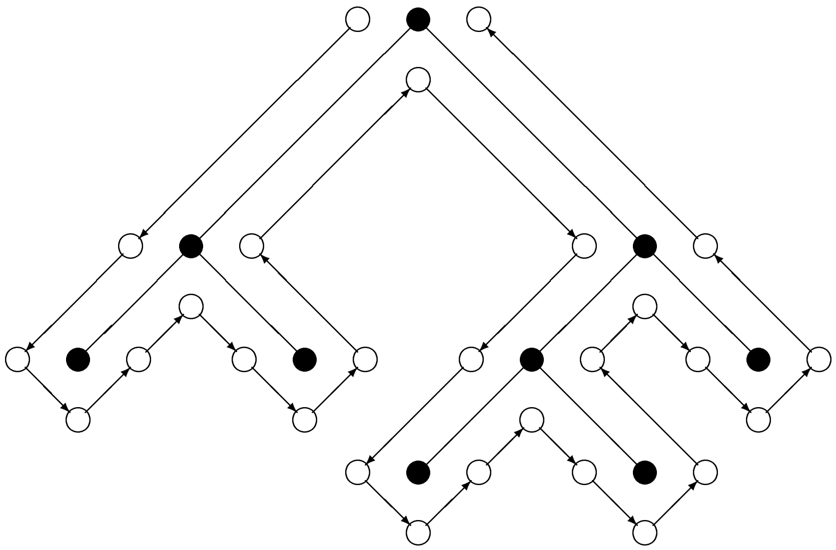

Figure 1: Euler Tour of a Tree

3. Eliminate nodes of that color. Takes $O(\frac{n}{p})$ time.
4. Redistribution step. Takes $O(\frac{n}{p} + \log p)$ time.
5. Recurse if the number of remaining elements exceeds p. Otherwise use Wyllies Algorithm.
6. Fill in the values for the elements we eliminated in step 3. Takes $O(\frac{n}{p})$ time.

Redistribution can be accomplished by compacting the *array* of elements. Assign 1 to elements not yet eliminated. Assign 0 to the others. Run an *array* prefix calculation to find where to move a not yet eliminated element.

Overall, the time, $T(n)$ can be calculated from the above timing information. Let $k = \frac{6}{5}$ and note that $T(p) = \log p$.

$$\begin{aligned}
T(n) &\leq \frac{n}{p}\log^* n + \log p + T(\frac{n}{k}) \\
&\leq \frac{n}{p}\log^* n + \log p + \frac{n}{kp}\log^*(\frac{n}{k}) + \log p + T(\frac{n}{k^2}) \\
&\leq (1 + \frac{1}{k} + \frac{1}{k^2} + \ldots)\left[\frac{n}{p}\log^* n\right] + (\log_k \frac{n}{p})\left[\log p\right] \\
T(n) &= O(\tfrac{n}{p}\log^* n + (\log p)\log \tfrac{n}{p})
\end{aligned}$$

If $p = n/(\log n \log\log n)$ then $T(n) = O(\frac{n}{p}\log^* n)$. Anderson and Miller [1] give a deterministic $O(n/p)$ time algorithm using $p \leq n/\log n$ processors (optimal) using more sophisticated version of this idea.

4 Leaf Numbering on a Tree

List prefix has many applications, e.g. generating different numberings of a tree. Consider the problem of numbering the leaves of a tree from left to right. We assume that the input consists of a tree with n vertices, numbered 1 through n in some order. The edges are represented using two n element arrays Left and Right, with Left(i) and Right(i) respectively denoting the left and right children of vertex i.

We first build a list (called the *Euler tour*) which runs up and down the tree hitting the leaves in left to right order. Figure 1 shows the Euler tour (arrows connecting circles) of a tree (nodes are indicated by solid discs). A prefix calculation on such a list where leaves have value 1 and internal vertices have value 0 then gives us the numbering for the leaves.

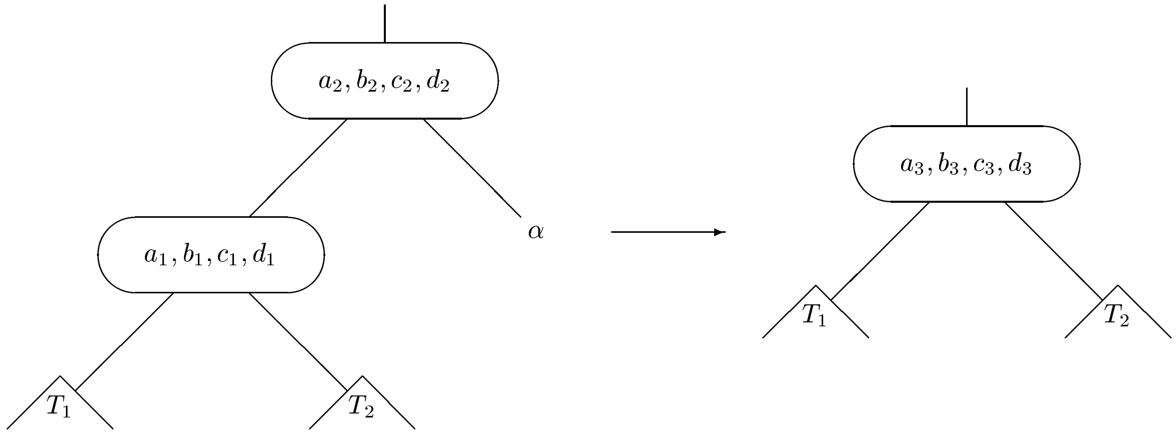

Figure 2: Raking a Leaf

To construct such a list, we associate each tree vertex with three nodes, left, right, and bottom. The left node of a vertex v points to the left node of the left child if any of v, else it points to the bottom node of v. The bottom node of v points to left node of the right child if any of v, else it points to the right node of v. The right node is null if v is the root, else it points to the right node of the parent of v. Figure 1 shows an example.

Using p processors, the nodes together with their connections can be constructed in n/p steps. The prefix takes an additional n/p steps, provided $p \leq n/\log n$.

5 Evaluating Expression Trees

Suppose we are given an n-node binary tree in which each internal node is an operator $+$ or $\times$, and leaves are integers. We can evaluate the expression represented by the tree in $O(\log n)$ time using $n/\log n$ processors[2].

We only describe the main idea, slightly adapted from [2]. The basic operation is called *raking*, and it eliminates a single leaf of the tree in $O(1)$ time using one processor. As will be seen, given a left-to-right numbering of the leaves, we can rake all the odd numbered leaves in a constant number of steps. Subsequently we can rake all leaves having numbers $4i + 2$ for all i; in the step following all leaves numbered $8i + 4$ for all i and so on to get the required time bound.

For the raking operation we use a generalized representation for each tree node as follows. Each tree node is represented by a quadruple (a, b, c, d). Given values x and y of its children, the value that is computed by applying the general operator is $axy + bx + cy + d$. In this framework, we can represent a $\times$ node by setting $a = 1$ and $b = c = d = 0$. Likewise a $+$ node is obtained for $a = d = 0$ and $b = c = 1$. The operation of raking a leaf whose value is α is shown pictorially in Figure 2. The tree is locally restructured as shown, and we set $a_3 = a_1a_2\alpha + a_1b_2$, $b_3 = b_1a_2\alpha + b_1b_2$, $c_3 = c_1a_2\alpha + c_1b_2$, and $d_3 = d_1a_2\alpha + d_1b_2 + c_2\alpha + d_2$. The case when T_1 and T_2 are null is handled with a minor change. The next idea is to do parallel rake operations. For this, we should ensure that the nodes involved in the rake operations are disjoint. It is easily seen that all odd numbered leaves can be raked in two parallel steps as follows: (i) in one step rake all the odd numbered leaves that are the left children (ii) in another rake all the odd numbered leaves that are right children. The same idea works for leaves $4i + 2$, leaves $8i + 4$ and so on.

Problems

1. Suppose you have a 6 coloured list of n elements. Show that it can be 3 coloured in time $O(\frac{n}{p})$ using p processors.

2. Suppose you have a k coloured list of n elements. Show that it can be 3 coloured in time $O(k + \frac{n}{p})$ using p processors.

   (Hint: First rearrange the list in memory so that the nodes are stored in memory in sorted order of their colours. Then extend the idea of the previous exercise. The following result will be useful to perform the rearrangment: n keys each w bits long can be ranked in time $O(2^w + \frac{n}{p} + \log p)$ using a p leaf completely binary tree.)

3. Show that an n node list can be deterministically ranked using p processors in time $O(\frac{n}{p} + \log n \log\log n)$. (Hint: Use deterministic coin tossing to colour the list in $\log n$ colours, and then use the result of the previous problem.)

4. Give an algorithm to colour the vertices of a tree such that adjacent vertices have different colours. Give an algorithm that runs in polylog time using a linear number of processors. Assume that the tree is rooted, i.e. is specified as an array PARENT, where PARENT[i] gives the parent of vertex i.

5. Suppose you are three arrays PARENT, LEFT, and RIGHT that describe a binary tree. Show how to construct an array DIST where DIST[i] gives the distance of vertex i from the root.

References

[1] R. Anderson and G. Miller. Deterministic parallel list ranking. In J. Reif, editor, *Aegean Workshop on Computing: VLSI Algorithms and Architectures.* Volume 319 of *Lecture Notes in Computer Science*, pages 81–90, New York, NY, June 1988. Springer Verlag.

[2] S. Kosaraju and A. Delcher. Optimal parallel evaluation of tree-structured computations by raking. In J. Reif, editor, *Aegean Workshop on Computing: VLSI Algorithms and Architectures.* Volume 319 of *Lecture Notes in Computer Science*, pages 101–110, New York, NY, June 1988. Springer Verlag.