

15-750 — Graduate Algorithms — Spring 2006

Miller and Derryberry

Assignment 5 Due date: Monday, April 24, 2006.

Some Reminders:

- Read the Policies section on the course website before you start working on this assignment.
- You may work in **groups of size up to 3** for this problem set if you wish. However, **you should write up your solutions separately**. That is, collaboration should be limited to talking about the problems, so that your writeup is written entirely by you and not copied from your partner. In addition, state whether you worked alone and **list all collaborators**.
- Please refrain from consulting external materials when solving these problems. This does not apply to looking up standard inequalities, definitions, and such things (e.g., $(1 + 1/x)^x \leq e$ or Stirling's Formula).
- Please **submit both an electronic version, as well as a hard copy of your solutions** at the beginning of class on the due date.
- In all problems, it is implicit that you should show that your answer is correct, even when this is not explicitly stated.
- If you have questions, contact the course staff.

1 Lowest Common Ancestor

Given a rooted tree T of size n and a set of m non-tree edges.

- (a) (5 points) Show how to pre-process the tree using a randomized algorithm that runs in expected $O(\log n)$ time and expected $O(n)$ work so that ancestor queries can be done in constant work.

SOLUTION: Using the Euler tour technique from class, we can compute the preorder and postorder numberings for all vertices. To determine whether a vertex u is an ancestor of v , we simply test whether $preorder(u) < preorder(v)$ and $postorder(v) < postorder(u)$. Both of these inequalities are true exactly when u is a (proper) ancestor of v .

To see this, note that $preorder(u) < preorder(v)$ implies u is an ancestor of v or $postorder(u) \leq postorder(v)$ (i.e., we do not see v until we finish seeing u 's subtree), so the additional knowledge $postorder(v) < postorder(u)$ implies u is an ancestor of v . Conversely, if u is an ancestor of v , we know we see u before v in any traversal of the tree

so $preorder(u) < preorder(v)$ and we know that after our last visit to v , we must still return to the root, which requires another visit to u , so $postorder(v) < postorder(u)$.

The running time of this preprocessing is $O(\log n)$ with total work $O(n)$ as discussed in class.

- (b) (10 points) Show how to label all the non-tree edges such that each edge has the lowest common ancestor of its end points as its label. Your algorithm should run in expected time $O(\log n)$ and expected work $O((n + m) \log n)$.

SOLUTION: After preprocessing T so that we can perform constant time ancestor queries, we can compute the lowest common ancestor (LCA) of all non-tree edges' end points as follows. For each vertex v in parallel, initialize an array A_v such that $A_v[i]$ contains a pointer to $p^{2^i}(v)$, the 2^i -th vertex encountered following parent pointers or nil if no such vertex exists. Computing this in parallel is accomplished using logic similar to list-ranking via shortcutting and the observation that $A_v[i] = A_{A_v[i-1]}(i-1) = p^{2^{i-1}}(p^{2^{i-1}}(v))$.

Next, for each non-tree edge (u, v) in parallel, we first test whether one end point is an ancestor of the other in constant time, labeling the LCA as appropriate if this is the case. Else, we search the ancestors of u to determine the lowest such ancestor that is also an ancestor of v (it is straightforward to see that this is in fact the LCA of u and v). Hence, it suffices to show how we use the arrays A_x to accomplish this in $O(\log n)$ time. This can be accomplished via the following algorithm.

FINDANCESTOR(u, v)

```

1  for  $i \leftarrow \lg n$  to 0
2  do if  $u \neq \text{nil} \wedge A_u[i]$  is not an ancestor of  $v$ 
3     then  $u = A_u[i]$ 
4  return  $p(u)$ 
```

Notice that an invariant is maintained throughout the for loop in FINDANCESTOR that at the end of the if statement $A_u[i]$ is either nil or an ancestor of v . Also, at all times u is not an ancestor of v (note the precondition on u and v). Therefore, when $i = 0$, u is not an ancestor of v , but $p(u)$ is. The running time of FINDANCESTOR is clearly $O(\log n)$.

2 Maximal Independent Set

Suppose that $G = (V, E)$ is an undirected graph with n vertices and m edges. For both parts of this problem you may use version 2 of Luby's algorithm.

- (a) (5 points) Describe an efficient implementation of Luby's algorithm. You should get $O(\log n)$ time and $O((n + m) \log n)$ work.

SOLUTION: The following pseudocode sketches an implementation of version 2 of Luby's algorithm.

```

MAXIMALINDEPENDENTSET( $G = (V, E)$ )
1  initialize all boolean fields to false
2  repeat
3       $notFinished = \mathbf{false}$ 
4      for  $v \in V$  in parallel
5           $v.random \leftarrow$  a random number in  $[0, 1]$ 
6           $v.smallest \leftarrow \mathbf{true}$ 
7      for  $(u, v) \in E$  in parallel
8          do if  $\neg u.inIndSet \wedge \neg v.inIndSet \wedge \neg u.nextToIndSet \wedge \neg v.nextToIndSet$ 
9              then if  $u.random < v.random$ 
10                 then  $v.smallest \leftarrow \mathbf{false}$ 
11                 else  $u.smallest \leftarrow \mathbf{false}$ 
12      for  $v \in V$  in parallel
13          do if  $v.smallest \wedge \neg v.inIndSet \wedge \neg v.nextToIndSet$ 
14              then  $v.inIndSet \leftarrow \mathbf{true}$ 
15                   $notFinished \leftarrow \mathbf{true}$ 
16      for  $(u, v) \in E$  in parallel
17          do if  $u.inIndSet$ 
18              then  $v.nextToIndSet \leftarrow \mathbf{true}$ 
19          if  $v.inIndSet$ 
20              then  $u.nextToIndSet \leftarrow \mathbf{true}$ 
21  until  $notFinished = \mathbf{false}$ 

```

By inspection, this implements version 2 of Luby's algorithm and each iteration of the repeat loop consumes $O(1)$ time and $O(n+m)$ work. By a similar argument to what was presented in class for version 1 of Luby's algorithm, the expected number of iterations is $O(\log n)$. Note that this algorithm requires both concurrent read and concurrent write, where write conflicts are resolved by allowing an arbitrary one of the writers to write its value. Note that we picked random real numbers and assumed they would never conflict, but we could relax this and choose random integers from $\{1, \dots, n^3\}$ and have no pair of vertices conflict with constant probability (i.e., we would have to redo an iteration with probability at most a constant).

- (b) (10 points) Suppose the graph is only given to you implicitly. In particular, the input is a set S of size n and a collection of subsets of S , A_1, \dots, A_k . We say two subsets are independent if they are disjoint. Design and analyze a work efficient parallel algorithm to find a maximal independent subset of A_1, \dots, A_k . You should get $O(\log k)$ time and

$O((n + m) \log k)$ work where m is the sum of the sizes of the A_i 's, that is,

$$m = \sum_{i=1}^k |A_i|$$

SOLUTION: The following pseudocode expresses how this alternatively-phrased maximal independent set problem can be solved using version 2 of Luby's algorithm.

```

MAXIMALINDEPENDENTSET( $S, A_1, \dots, A_k$ )
1  initialize all boolean fields to false
2  repeat
3       $notFinished = \mathbf{false}$ 
4      for  $i \in \{1, \dots, k\}$  in parallel
5           $A_i.random \leftarrow$  a random number in  $[0, 1]$ 
6           $A_i.smallest \leftarrow \mathbf{true}$ 
7          for  $(i', x') \in \{(i', x') \mid i' \in \{1, \dots, k\} \wedge x' \in A_{i'}\}$  in parallel
8          do if  $\neg A_i.inIndSet \wedge \neg A_i.nextToIndSet$ 
9              then  $x.random \leftarrow A_i.random$  // minimum concurrent write
10         for  $(i', x') \in \{(i', x') \mid i' \in \{1, \dots, k\} \wedge x' \in A_{i'}\}$  in parallel
11         do if  $A_i.random > x.random$ 
12             then  $A_i.smallest \leftarrow \mathbf{false}$ 
13         for  $i \in \{1, \dots, k\}$  in parallel
14         do if  $A_i.smallest \wedge \neg A_i.inIndSet \wedge \neg A_i.nextToIndSet$ 
15             then  $A_i.inIndSet \leftarrow \mathbf{true}$ 
16              $notFinished \leftarrow \mathbf{true}$ 
17         for  $(i', x') \in \{(i', x') \mid i' \in \{1, \dots, k\} \wedge x' \in A_{i'}\}$  in parallel
18         do if  $A_i.inIndSet$ 
19             then  $x.inIndSet \leftarrow \mathbf{true}$ 
20         for  $(i', x') \in \{(i', x') \mid i' \in \{1, \dots, k\} \wedge x' \in A_{i'}\}$  in parallel
21         do if  $x.inIndSet$ 
22             then  $A_i.nextToIndSet \leftarrow \mathbf{true}$ 
23
24     until  $notFinished = \mathbf{false}$ 

```

By inspection, this algorithm implements version 2 of Luby's algorithm and each iteration of the repeat loop consumes $O(1)$ time and $O(m + n)$ work. By a proof similar to what was presented in class for version 1 of Luby's algorithm, the total number of iterations is $O(\log k)$ because this algorithm is working on a graph with k nodes and $O(k^2)$ implicit edges. Also, note that we are assuming in line 9 that concurrent writes of numbers are resolved by taking the minimum value that is being written to the location. Other concurrent writes need only to be resolved by allowing an arbitrary writer to write its value.

3 Merging Two lists

Suppose we have two sorted lists $A = a_1, \dots, a_n$ and $B = b_1, \dots, b_n$ all of whose elements are distinct. The goal is find a parallel algorithm to merge these two lists onto one sorted list of length $2n$. Observe that if we can determine the index of every element in the final sorted list we can actually “sort” the list in constant time and linear work.

- (a) (5 points) Construct a simple parallel algorithm that works in $O(\log n)$ time and $O(n \log n)$ work.

SOLUTION: (courtesy of Jim McCann)

To merge the lists in $\log(n)$ time with $2n$ processors, assign one processor to each element and use a binary search to find how many elements in the other list are before it. Summing the number of elements before it in its list and the other list gives the number of elements before it in the final list, so it can be copied to its final address.

Correctness:

Not much to say.

Timing:

We use n processors, and each does a binary search (that is, spends $O(\log n)$ time).

- (b) (10 points) Show how to use the algorithm from the first part to find an $O(\log n)$ time algorithm that uses $O(n)$ total work.

Hint: Divide your lists up into pieces of approximate size $\log n$.

SOLUTION: (courtesy of Jim McCann)

Place a processor on every $\log(n)$ th element in each list. Each processor figures out where the element it is sitting on is in the final list (using the method from (a)), then uses a list-merging algorithm to figure out the next $2 \log n$ elements in the final list.

Correctness:

Whenever an element is assigned to a final index, that assignment is correct. So it suffices to show that every element is assigned to.

This, in turn, boils down to showing that every element must be within $2 \log n$ of the element a processor is sitting on in the final list. But this is easy to see, since the maximum number of elements from list a that can be between an output position and a processor is $\log n$, and similarly for the maximum number of elements from list b .

Timing:

We use $2n/\log n$ processors, and each does $O(\log n)$ to find its initial position, then merges $2 \log n$ elements ($O(\log n)$ time with the standard algorithm).