CS260 Homework 3

March 12, 2020

1 Parallel Construction of BSTs (15pts)

(a) We will split the array into two halves, construct a tree for the left half and the right half respectively in parallel. Then we union the two trees and get the output. Pseudocode is shown below.

```
build(A, n) {
  if n = 0 return empty tree;
  if n = 1 return singleton(A[0]);
  n' = n/2;
  in parallel:
    TL = build(A, n');
    TR = build(A+n', n-n');
  return union(TL, k, TR);
}
```

(b) Considering the two sub-problems each have input size $n/2$, the work of the final union function is $O(n)$, and depth is $O(\log^2 n)$. We have:

$$W(n) = W(n/2) + O(n)$$

$$D(n) = D(n/2) + O(\log^2 n)$$

(c) Solve the recurrence above we have $W(n) = O(n \log n)$, $D(n) = O(\log^3 n)$.

(d) Yes. We first construct a tree using the algorithm above, then use the `to_array` algorithm in the lecture to output it into an array in parallel.

(e) B. merge sort.

2 Parallel Treaps and Programming (60 pts)

2.1 Treaps (15pts)

Answer the following questions:

(a) The one with the highest priority.

(b) For a set of keys and their priorities fixed, the root is uniquely fixed because it is the one with the highest priority. The set of keys to the left and right subtrees are decided solely based on if it is larger than the root key or not, so the set of keys in the left and right subtrees are also fixed. As a result, the two children of the root are fixed. By induction we can show that the final state of the treap is unique.

(c) For any treap, let's consider a quicksort algorithm, where each key will be associated with a priority as in a treap. Then in each round of recursive call, we always use the element with the highest priority as the pivot. In this case, all the elements still have equal probability to be the pivot, thus this algorithm behaves exactly the same as a standard quicksort algorithm. Each round in this algorithm corresponds to a level in the treap. As we've seen in class, the number of rounds for a quicksort algorithm is $O(\log n)$ with high probability. Therefore the height of a treap is also $O(\log n)$ with high probability.

(d) It is similar to the quicksort algorithm: for a set of keys with priorities, we first find out the element e_r with the highest priority. This can be done by a parallel reduce algorithm. Then we use it as the root, partition the input array based on the key of e_r into L and R. Finally we recursively construct the left and right subtrees with input L and R respectively, in parallel.

2.2 Joining and Spliting Treaps (25pts)

(a) We can just compare k's priority with the priority of the roots of the two trees. If k has a higher priority than both roots, k has a priority higher than all keys in the input trees.

(b) In every `join` calls in the `split` algorithm, the pivot is at a higher level in the input tree than the two input trees. Thus the pivot must have a higher priority than the other input elements.

(c) We consider the operations related to returning T_L in the `split` algorithm, the cost to constructing T_R is symmetric and is asymptotically the same. For input tree T, the total number of `join` functions called is at most the height of T (one pivot for each level). Since all `join` calls are direct `join`, the cost are all constants. In total, the cost of `split` on treaps is proportional to the height of T, which is $O(\log n)$ with high priority.

(d) -

2.3 Union/intersection/difference on two treaps (20pts)

Answer the following questions:

(a) We always use the root with a higher priority to do `split`. So if $T1$'s root has a higher priority, we swap the two trees. Pseudocode is shown below.

```
union(T1,T2) {
  if T1 is empty return T2;
  if T2 is empty return T1;
  if T1's root has a higher priority than T2's root, return union(T2, T2);
  k = key at the root of T2;
  L2 = the left subtree of T2;
  R2 = the right subtree of T2;
  (L1, b, R1) = split(T1, k);
  in parallel:
    TL = union(L1, L2);
    TR = union(R1, R2);
  return join(TL, k, TR);
}
```

(b) No. Both of them used `join2`. To make `join2` work, we still need special design or make the general `join` work.

(c) -