Homework Problem Set 1

Eric Addison
Partner: Ari Bruck

Parallel Algorithms
Summer 2017

Problem 1.1

Create an account on TACC stampede using TACC user portal. Include your tacc user id.

Account created. Username ea26799.

Problem 1.2

Suppose that an array does not have all elements that are distinct. Show how you can use any algorithm that assumes distinct elements for computing the maximum to solve the problem when elements are not distinct.

Consider an array `A` of size n that may not have all distinct integer elements, and an algorithm `find_max` that is valid only for arrays with *distinct* elements. The following pseudo code will preprocess the array `A` so that it can be used in `find_max`:

```
1  for all i in parallel do
2    B[i] = n*A[i] + i;
3  mx = find_max(B)/n;
```

The trick here is that the elements in `B` are guaranteed to be unique:

Proof:

Let $a, b \in \mathbb{Z}$ be any two values in the array `A` with indices $i, j \in 0, 1, 2, ..., n-1$ $(i \neq j)$, so $a' = na + i$ and $b' = nb + j$.

Case 1: $a = b$

if $a = b$, then $a' = na + i$ and $b' = na + j$, so because $i \neq j$, $a' \neq b'$.

Case 2: $a \neq b$

Assume without loss of generality $a > b$, then a can be written as $a = b + k$ with $k \geq 1$. Now

$$a' = na + i = n(b + k) + i = nb + nk + i \geq nb + n > nb + j = b',$$

so we have $a' > b' \Rightarrow a' \neq b'$, regardless of the array positions i and j.

□

We know that all of the elements in the modified array `B` are guaranteed to be unique by the preceding proof, and in fact case 2 of the proof also establishes the order-maintaining property we need: for any two elements $a, b \in$ `A`, $a > b \Rightarrow a' > b'$, so there is no chance of corrupting the result with this procedure.

This process can be slightly modified to be used with floating-point values. A small amount of additional pre-processing of the array of floating-point values will transform it into a usable array of integers:

```
1  for all i in parallel do
2    B[i] = (int) (log10(A[i])*10^(N_PRECISION));
3
4  mx = find_max_non_distinct(B);  // algorithm above
5  mx = 10^(((float)mx)/10^(N_PRECISION));
```

Here, `N_PRECISION` is the number of decimal places that the floating point values used can represent (e.g. 7 for 32-bit `float`s, 15 for 64-bit `double`s). There are implementation specific details here that could limit the usefulness of this approach, notably numeric precision.

Problem 1.3

Give a parallel algorithm on a CREW PRAM to determine the largest odd number in a given array of positive integers. Assume that the array has size n and the number of processors available is also n. The size n may not be a power of 2. Your algorithm should not take more than O(log n) time.

Assume 0-based indexing. This algorithm follows the standard binary tree work-depth model, similar to the canonical `reduce`-sum algorithm. The key expression is on line 8. The algorithm returns zero if no odd values are found.

```
1  i = get_my_id();
2  B[i] = A[i];
3  hmax = ceil(log(n))
4  for h = 1 to hmax do
5    if (i < n/(2^h)) then
6      x = (2*i>=n) ? 0 : B[2*i];
7      y = (2*i+1>=n) ? 0 : B[2*i+1];
8      B[i] = max( (x%2)*x, (y%2)*y );
9  print B[0];
```

This algorithm is asymptotically equivalent in complexity to the binary tree `reduce`-sum problem. It has complexities $\boxed{T(n) = O(\log n) \text{ and } W(n) = O(n)}$. It is work-optimal because $W(n) = T_{\text{seq}}(n) = O(n)$. The conditional operators on lines 6 and 7 allow this algorithm to run properly for values of n that are not powers of 2. Another option could be to pad the array with zeros out to the next power of 2, if the additional conditional checks are found to hinder performance. Or, the first iteration of the loop over `h` could be separated and written as the only iteration with the conditional checks, since $h = 1$ is the only value for which it would be possible to index outside of the array.

Problem 1.4

Give a parallel algorithm on a CREW PRAM with time complexity O(log n) and work complexity O(n) to compute the inclusive parallel prefix sum of an array of size n by combining two algorithms: sequential prefix sum that takes O(n) time and O(n) work and a non-optimal parallel prefix algorithm that takes O(log n) time and O(n log n) work.

Assume we have a non-optimal algorithm to compute an inclusive parallel prefix sum, (from lecture):

```
1  for all i parallel do
2    C[i] = A[i];
3  for(d=1; d<=n; d*=2)
4    for all i parallel do
5      if(i-d>=0)
6        C[i] = C[i] + C[i-d];
```

The first loop in this algorithm has complexities $T_1(n) = O(1)$ and $W_1(n) = O(n)$, and the second loop has complexities $T_2(n) = O(\log n)$ and $W_2(n) = O(n \log n)$, so $T_{\text{par}} = O(\log n)$ and $W_{\text{par}} = O(n \log n)$.

Consider splitting the input array `A` into $n/\log n$ segments of size $\log n$. Assume we can call the standard $O(n)$ sequential algorithm as `seq_prefix_sum_inplace(A)` (assume this implementation modifies the array in place). A description of the cascaded algorithm is:

Step 1: Copy array `A` into new array `B`	$T(n) = O(1)$	$W(n) = O(n)$
Step 2: Call `seq_prefix_sum()` on segmented sub-arrays	$T(n) = O(\log n)$	$W(n) = O(n)$
Step 3: Form new array from the last element of each segment	$T(n) = O(1)$	$W(n) = O(n)$
Step 4: Call `parallel_prefix_sum()` on new array (size `n/log(n)`)	$T(n) = O(\log n)$	$W(n) = O(n)$
Step 5: Add step 4 results to step 2 results with divided index	$T(n) = O(1)$	$W(n) = O(n)$

Looking down the left column, it is clear that the resulting algorithm will have complexities $\boxed{T(n) = O(\log n) \text{ and } W(n) = O(n)}$. Pseudo code for the cascaded algorithms follows (assume 0-based indexing, and that a subarray can be extracted with syntax `A[x:y]`):

```
1  // step 1
2  for all i in parallel do
3    B[i] = A[i];
4
5  // step 2
6  lgn = log(n);
7  n_div_logn = n/log(n);
8  for i = 1 to n_div_logn parallel do
9    seq_prefix_sum_inplace(B[lgn*(i-1):lgn*i-1]);
10
11 // step 3
12 C = allocate_new_array( n_div_logn );
13 for i = 1 to n_div_logn parallel do
14   C[i] = B[i*lgn-1];
15
16 // step 4
17 D = parallel_prefix_sum(C);
18
19 // step 5
20 for all i parallel do
21   B[i] = B[i] + D[i/n_div_logn];
```

Problem 1.5

Given an integer array A and two numbers x and y, give a parallel algorithm on a CREW PRAM to compute an array D such that D consists only of entries in A that are greater than or equal to x and less than or equal to y. The order of entries in D should be same as that in A.

I use a four step approach here:

Step 1: Compute indicator array `B`	$T(n) = O(1)$	$W(n) = O(n)$
Step 2: Compute index locations using exclusive parallel prefix-sum	$T(n) = O(\log n)$	$W(n) = O(n)$
Step 3: Allocate new array of proper size	$T(n) = O(1)$	$W(n) = O(1)$
Step 4: Populate `D` with appropriate elements	$T(n) = O(1)$	$W(n) = O(n)$

The indicator array contains all 1's and 0's determining whether a given element in the input array meets the criteria. The indicator is prefix-summed to determine the correct indices in the new array that should contain the associated values from the input array. The required size for the new array is taken from the last entry of the prefix sum (i.e. largest required index). This algorithm will have complexities $T(n) = O(\log n)$ and $W(n) = O(n)$. A pseudo code implementation, assuming the result of a logical expression like `a>b` returns an integer value of 1 for `true` and 0 for `false` (like in C):

```
1  // step 1
2  for all i parallel do
3    B[i] = (A[i]>=x) & (A[i]<=y);
4
5  // step 2
6  C = exclusive_parallel_prefix_sum(B);
7
8  // step 3
9  new_size = C[n-1]+1;
10 D = allocate_new_array(new_size);
11
12 // step 4
13 for all i parallel do
14   if(B[i]==1)
15     D[C[i]] = A[i];
16
17 print D
```