<u>Parallel Assignment</u>

Sravanthi Malepati
002-53-8438

1. Write pseudocode for a non-recursive prefix-sums algorithm that is similar to the one studied in class but that does not use the auxiliary variables B and C. The input array A should hold the prefix sums when the algorithm terminates.

Solution:

Definition: The parallel prefix problem takes a binary associative operator ⊕, and is used to compute the function that maps any given power list p = $\langle x_0 \ldots x_{n¡1} \rangle$ to the power list $\langle x_0\ (x_0 \oplus x_1)\ (x_0 \oplus x_1 \oplus x_2) \ldots (x_0 \oplus \ldots \oplus x_{n¡1}) \rangle$

If n > 1, apply ⊕ to successive pairs of elements to obtain the length-n/2 power list $p' = \langle (x_0 \oplus x_1)(x_2 \oplus x_3) \ldots (x_{n¡2} \oplus x_{n¡1}) \rangle$
Recursively compute the prefix sum of p to obtain the length-n/2 power list $p'' = \langle (x_0 \oplus x_1)(x_0 \oplus x_1 \oplus x_2 \oplus x_3) \ldots (x_0 \oplus \ldots \oplus x_{n¡1}) \rangle$

The powerlist p'' contains the odd-indexed elements of f(p). To get the even-indexed elements of f(p), take the ⊕ of the powerlist obtained by shifting p'' to the right one position (and introducing a 0 in the first position) with $\langle x_0\ x_2\ x_4 \ldots x_{n¡2} \rangle$

For example, take an array x = (1, 3, 6, 10, 15, 21, 28, 36) where n=8 i.e.,2^3

Input: Array of size n = 2^k **Output**: Array A has prefix sum

Algorithm:

```
Begin
        for i = 1 to n pardo
                set Array_A(0,i):= A(i)

        for h = 1 to log n do
                for 1 <= j <= n/2^h pardo
                        set Array_A(h,i) = Array_A(h-1,2i-1) * Array_A(h-
        1,2i)

        for h = log n to 0 do
                for i = 1 to n/2^h pardo
                        i even  : Set  Array_A( (log n +h , i): =
        Array_A((log n +h)+1, i/2)
                        i = 1      : set Array_A( (log n,1): = Array_A( (h,1)
                        i odd  > 1 : set Array_A( (log n +h,i): =
        Array_A(log n +h)+1,(i-1)/2) * Array_A(h,i)
```

end

2. We are given an array of colors A = [a_1, a_2, ..., a_n] drawn from k colors {c_1, c_2, ..., c_k}, where k is a constant. We wish to compute k indices i_1, i_2, ..., i_k, for each element a_i, such that i_j is the index of the closest element to the right of a_i whose color is c_j. If no such element exists, then set i_j = 0. Write pseudocode for solving this problem in O (log (n)) using a total of O(n) operations.

Solution:

Input: Given an Array of colors A = [a_1, a_2, ..., a_n] drawn from K colors c_1, c_2, ..., c_k, where k is a constant and n<k.
i.e., A is subset of k, so n<k.
Output: i_1, i_2, ..., i_k which is the index of the closest element to the right of a_i.

Explanation:

1. Initially the pseudocode for Color Hash Table creates a hash table with colors as key and the indices where the color is present in the array A as values. This hash table is returned to the nearest index pseudocode which computes i_j.
2. Next, traversing the array A can be done in parallelly. If each processor is given a sub-array of A (Ex: The 1st processor gets the whole array, the 2nd processor gets array minus the 1st element, similarly i^{th} processor gets array minus $(i-1)^{th}$ elements).
3. For each sub-array, we determine whether color c_k is in that sub-array for each color and check whether it can be done in parallel way.
4. Finally, we only check for those indices in hash table generated by Color Hash Table pseudocode.
5. By dividing the array into sub-arrays, the algorithm can run in O(log(n)) time and in

O(n) operations.

Pseudo Code for Coloring Hash Table:

For a_i in A do
Add a_i to hashing index table and concat the index value of a_i to the list concated with a_i
end for;

return hash_table;

Pseudo Code for nearby Index Value:

Let Solution_1 be the output of hash_table output

```
for a_i in A pardo
        let current_position = current index position of A
                for c_j in k-color set pardo
                    if c_k is in current list then
                            length = Solution_1.len()
                            temp = array indices for color k
                while (max (temp) < current_position) or max (temp) == current_position
                do
                set length = length + 1
                temp = temp[: length]
        end while;
                Sum the current_position +max (temp) = Result for the color k
                    else
                    current_position + 0 = Result for the color k
                    end if;
                end for;
            end for;
end;
```

3. Suppose that we have an algorithm A to solve a given problem P of size n in O (log (n)) time on the PRAM model using O(n log (n)) operations. On the other hand, an algorithm B exists that reduces the size of P by a constant fraction in O(log(n)/log log(n)) time using O(n) operations without altering the solution. Derive an O (log (n)) time algorithm to solve P using O(n) operations.

Ans:

Given that Algorithm-A solves the problem P of size ‘n’ in O (n log n) operations which are involved in solving the problem and O (log n) time using PRAM model.

And also, Algorithm-B deploys O (n) operations and lowers the size of problem P by a constant fraction in O (log (n)/log log (n)) without solving the problem P but reduces it by a constant fraction (Z) which doesn't affect the solution of the algorithm, but it can help to enhance it.

For obtaining an O (log (n)) time algorithm to solve the problem size of n. We can apply the Algorithm-A and Algorithm-B so that we can achieve the required solution. We will start our solution by applying the Algorithm-B, so now let us assume that we are applying Algorithm-B once the size of the problem P is reduced to half that us n/2. (Because Algorithm-B reduces the size of P by a constant fraction of 2, assume n as 2^{2^k} where k is some constant and constant fraction Z)

First, we apply algorithm-B on P for k-times which then reduces the size of P to $P/2^k$ i.e. n/log n. The time take and work done are as follows:

Time taken to apply algorithm-B on P for k-times = O (k * log n / log (log n))

= O (log n/log (log n))

Work done to apply algorithm-B on P for k-times = O (k * n) = O(n)

Since the size of the problem is reduced, by utilizing the Algorithm-A to do actual computation. Because the second algorithm just reduces the size of the problem.
So, by applying algorithm-A on P of size n/log n we get the solution which can be solved in O ((n/log n) * log (n/log n)) operations and time O (log (n/log n)).

The above equations can be calculated as:
O ((n/log n) * log (n/log n)) => O ((n/log n) * (log n - log (log n)))
=> O (n - n log (log n/log n)) = O(n)

O (log (n/log n)) => O (log n – log (log n)) => O (log n)

Therefore, problem P can be solved in O(n) operations and O (log n) time using both algorithms A and B.