Advanced Algorithms Homework 4

Vijayraj Shanmugaraj, 20171026

March 21, 2021

Exercise 1

Yes this approach can run in the EREW model, with additional constraints for exclusive reads. Now, in the upward traversal, there are no concurrent reads i.e. every memory block is accessed only by one processor.

However, in the downward traversal, in the worst case, one element can be accessed twice, by i) its immediate right child and ii) its right sibling's left child i.e. by an odd-indexed processor or an even-indexed processor. In order to avoid these concurrent reads, we change the algorithm as follows.

Algorithm 1: Modified prefix sum algorithm

Step 1: Upward traversal (Remains same)
for *i = 1 to n/2, in parallel* **do**
  $b_i \leftarrow a_{2i-1} + a_{2i}$
end

Step 2: Recursively compute the prefix sums of $B(b_1, b_2, ...b_n)$ and store them in $C(c_1, c_2, ...c_{n/2})$

Step 3: Modified downward traversal

Prefix sum of "left node" elements
for *i = 1; i ≤ n/2; i = i+2, in parallel* **do**
  if *i = 1* **then**
    $s_1 \leftarrow c_1$
  else
    $s_i \leftarrow c_{(i-1)/2} + a_i$
  end
end

Prefix sum of "right node" elements
for *i = 2; i ≤ n/2; i = i+2, in parallel* **do**
  $s_i \leftarrow c_{i/2}$
end

By converting the downward traversal into two separate for loops, we have effectively separated the above two memory accesses to happen separately, leading to exclusive reads. In any step of the algorithm, only one processor writes into a certain memory location at a given time, and hence the exclusive write condition is taken care of.

Time complexity: Now, Step 1 is O(1) and Step 3 is 2*O(1) ≡ O(1) (Since we have two for loops that need just n/4 processors). And Step 2 is a recursive call on n/2 elements, and hence the time complexity of Step 2 would be T(n/2). So our recursion would still remain

$$\begin{aligned} T(n) &= T(n/2) + O(1) \\ T(n) &= O(\log n) \end{aligned}$$

Work complexity: Step 1 and Step 3 have 2*O(1) (for the two for loops) operations done by n/4 processors. So total work done here would be

$$\begin{aligned} &= \frac{n}{4} \times 2 \cdot O(1) \\ &= O(n) \end{aligned}$$

And Step 2 is a recursive call on n/2 elements, and hence the complexity of work done would be W(n/2). So our recursion would still remain

$$\begin{aligned} W(n) &= W(n/2) + O(n) \\ W(n) &= O(n) \end{aligned}$$

Exercise 2

Now, for the new fast merging algorithm, we first partition A and B into sizes $\log\log n$, say A = $\{A_1, A_2, ...\}$ and B = $\{B_1, B_2, ...\}$. Say A' = $[p_1, p_2, ...]$, B' = $[q_1, q_2, ...]$, where p_i is the first element of block A_i, and q_i is the first element of block B_i. So,

$$|A'| = |B'| = \frac{n}{\log\log n}$$

Step 1: Merge the elements of A' and B'.

In this process, we get the two arrays $rank(A', B')$, which is a list of ranks of all elements of A' in B', and $rank(B', A')$, which is a list of ranks of all elements of B' in A'. This can be done using the non-optimal algorithm done previously. The time and work complexity of this step would be

$$\begin{aligned} T(n) &= O(\log\log(\log\log n)) \equiv O(\log\log n) \\ W(n) &= O\left(\frac{n}{\log\log n} \cdot (\log(\log n - \log\log\log n))\right) \\ &= O\left(\frac{n}{\log\log n} \cdot (\log\log n)\right) \\ &\equiv O(n) \end{aligned}$$

Step 2: Proceed to create the arrays $rank(B', A)$ and $rank(A', B)$.

Now, say $rank(p_i, B') = r_i$. This means p_i will have its rank somewhere in the block B_{r_i}, as $q_{r_i} \le p_i \le q_{r_{i+1}}$. For each p_i, we can get its exact location by sequential/binary search, which

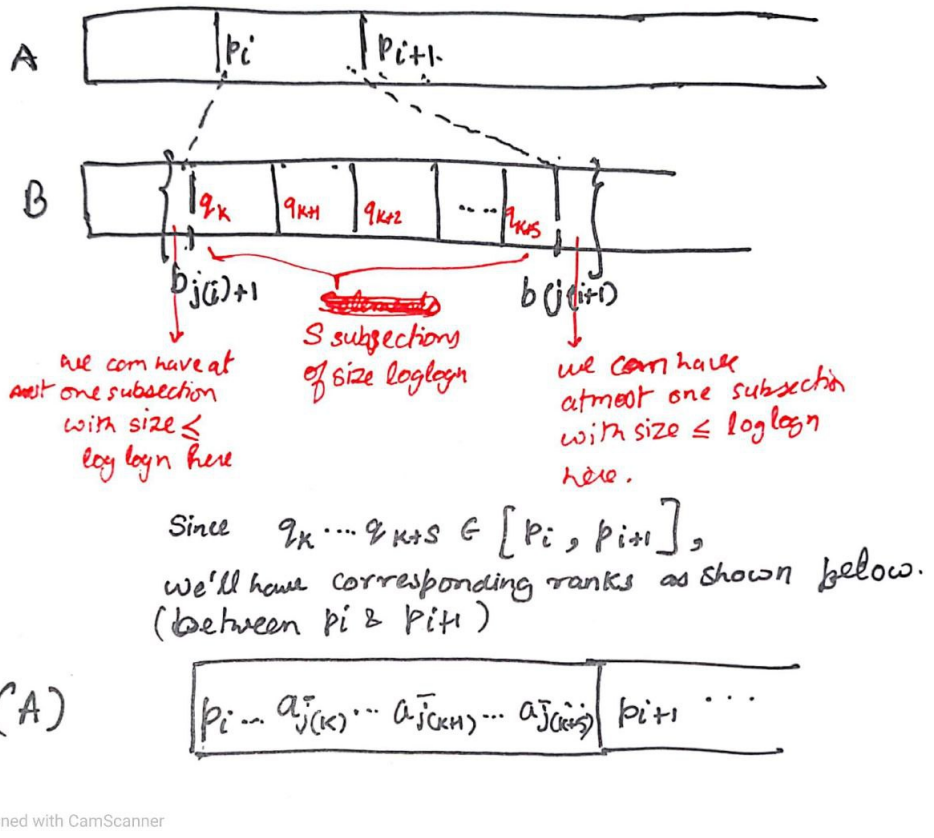
will take O(log log n) for each element. Since we have around $n/\log\log n$ elements, this can be found in the parallel time and work complexity

$$\begin{aligned} T(n) &= O(\log\log n) \\ W(n) &= \log\log n \text{ operations } \times \frac{n}{\log\log n} \text{ elements,} \\ &= O(n) \end{aligned}$$

considering $n/\log\log n$ processors at our disposal.

Step 3: Now $\forall i$, get the ranks of elements $A_i \setminus \{p_i\}$ in B, and $\forall k$, get the ranks of elements $B_k \setminus \{q_k\}$ in A.

We have $rank(B', A)$ and $rank(A', B)$ from step 2. Say $rank(p_i, B) = j(i)$ and $rank(q_k, A) = \bar{j}(k)$. Given that $rank(p_i, B) = j(i)$, and p_i is the first element of A_i, we know that all elements in A_i will have their ranks in B in between $b_{j(i)}$ and $b_{j(i+1)}$ since all elements in A_i lie between p_i and p_{i+1}. Now, if j(i+1) - j(i) $\leq$ log log n, we can merge A_i with the corresponding subarray of B ($b_{j(i)+1}...b_{j(i+1)}$) in O(log log n) time sequentially. Otherwise, if j(i+1) - j(i) > log log n, there exists elements of B', say $q_k, q_{k+1}, ...q_{k+s}$ that lie between $b_{j(i)+1}$ and $b_{j(i+1)}$. Since we know the ranks of $q_k, q_{k+1}, ...q_{k+s}$ in A i.e. $\bar{j}(k), ...\bar{j}(k+s)$, our problem comes down to merging s+2 disjoint sequence pairs as shown in the diagram below:



Since the subsequences are completely disjoint, we can sequentially merge them in O(log log n) time using O(s) operations. This process can be performed concurrently for all blocks A_i. So,

the time and work complexity

$$\begin{aligned} T(n) &= O(\log\log n) \\ W(n) &= O(s) \text{ operations } \times \frac{n}{\log\log n} \text{ subsequences,} \\ &= O(\log\log n) \times \frac{n}{\log\log n} \text{ subsequences (Since s cannot be larger } |A_i| \text{ or } |B_i|) \\ &= O(n) \end{aligned}$$

Clearly, for each step,

$$\begin{aligned} T(n) &= O(\log\log n) \\ W(n) &= O(n) \end{aligned}$$

Hence, our algorithm is time and work optimal.

Exercise 3

We know that for the parallel search algorithm,

$$\begin{aligned} T(n) &= O(\log_p n) \\ W(n) &= O(p\log_p n) \end{aligned}$$

So, we require $T(n) = O(\log\log n)$. So,

$$\begin{aligned} \log_p n &= k \cdot \log\log n \quad \text{(k is some constant)} \\ \frac{\log n}{\log p} &= k \cdot \log\log n \\ \frac{\log p}{\log n} &= \frac{1}{k \cdot \log\log n} \\ \log p &= \frac{\log n}{k \cdot \log\log n} \\ p &= e^{\frac{\log n}{k \cdot \log\log n}} \\ &= n^{1/\log\log n} \quad \text{(Taking k = 1 for convenience)} \end{aligned}$$

The work complexity

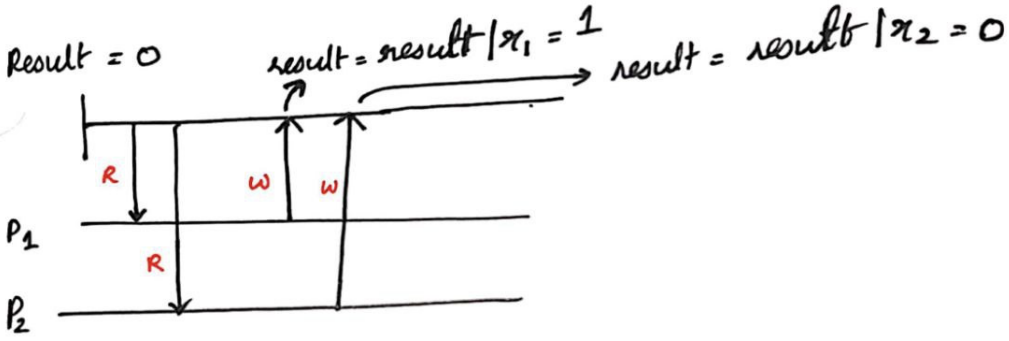
$$\begin{aligned} W(n) &= O(p\log_p n) \\ &= O(p\frac{\log n}{\log p}) \\ &= O(n^{1/\log\log n} \cdot \frac{\log n}{\log p}) \\ &= O(n^{1/\log\log n} \cdot \frac{\log\log n \cdot \log n}{\log n}) \\ &= O(n^{1/\log\log n}\log\log n) \end{aligned}$$

Exercise 4

Assumption: The maximum number of bits for a number is b, a constant.

Now since we have a CRCW model, we cannot simply allow all processors to perform bitwise-OR on a common output. We may have inconsistencies caused due to the concurrent nature of the model. Take a single bit problem, for example. We will assign our result = 0 initially. Consider the following scenario:

Processor P_1 wants to bitwise-OR $x_1 = 1$ to the result, and processor P_2 wants to bitwise-OR $x_2 = 0$ to the result. Consider this particular situation, where P_1 and P_2 read from the result variable, and write back into the variable in the same order.



Now although P_2 reads the correct value of the result variable in the beginning, it does not acknowledge the change that the result variable has undergone due to the write that happened by P_1, where it has successfully performed the operation on the result variable. Instead, P_2 still works with an outdated value of result, and writes back result$|x_2 = 0$ back into the result variable, when it should have actually been 1.

Now, keeping in mind the 1-bit example, we can say that inconsistency arises only when a processor tries to write a 0 as an intermediate bitwise-OR result, and is not timed well. In other cases, if any processor wants to write back a 1, the final bitwise-OR will have a 1 in that bit (since there is no combination in the OR truth table where one of the inputs is a 1 and the result of the bitwise-OR is 0).

Now, while developing an algorithm for this we have to deal with each bit of the number separately, since a given number might have a combination of 0s and 1s, and hence bitwise-OR-ing numbers together might still result in inconsistency in certain bits. Hence, consider

the algorithm below, having n processors at our disposal:

Algorithm 2: Parallel bitwise-or calculation algorithm

Step 1: Initializing variables
A[n]; Result[b]; (A is the input array with n numbers)
for *i = 0 to b-1, in parallel* **do**
  Result[i] = 0
end

Step 2: Working with individual bits of the result
for *i = 0 to n-1, in parallel* **do**
  n = A[i]
  j = 0
  while $n \neq 0$ **do**
    if n mod 2 = 1 **then**
      Result[j] = 1
    end
    $n \longleftarrow n >> 1$
    $j \longleftarrow j + 1$
  end
end
Step 3: Converting the result array into a final numerical value
final_result = 0

(By a single processor)
for *i = 0 to b-1* **do**
  final_result ⟵ final_result + (Result[i] << i)
end

Why this algorithm works:

Now, in this algorithm, the i^{th} element of the Result array represents the i^{th} bit of the final result. Now only if processors find a 1 in their number's i^{th} bit, they get to write into the result array. And since 1|1 = 1, parallel writes of 1 will not modify the result, since the bit effectively will stay as 1, and since both 0|1 = 1|1 = 1 . If none of the numbers have 1 in the i^{th} bit, i.e. the i^{th} bit of all numbers is 0, that bit will be 0 in the final result, and since no processor will get a chance to write into Result[i] in the mentioned case, Result[i] will hold its original value i.e. 0, the value it would have held even after the bitwise-or of all numbers. In the final step, we make one processor to reconstruct the final number from the Result array, since the same problems discussed above would take place if we try to involve b processors here, and there is usually no standard operator that allows the processor to work with exactly one bit of a number without performing calculations on the other bits simultaneously.

Time complexity: Step 1 takes O(1) time, where b processors simultaneously write into one memory slot each. If b > n, then the time taken would be O(b/n) ≡ O(b). Step 2 takes O(b) time, since the n processors are iterating through the bits of one number (which is, in the worst case b bits), in parallel. Step 3, clearly takes O(b) time too. Hence the time complexity of this algorithm is

$$T(n) = O(b) \equiv O(1) \text{ (Considering b to be a constant)}$$

Work complexity: Step 1 involves the filling of b memory slots. Hence the work for Step 1 is

O(b). Step 2 involves n processors are iterating through b bits (in the worst case), and hence the work done is O(bn). The third step is just one processor going through b iterations. Hence the work here is O(b)

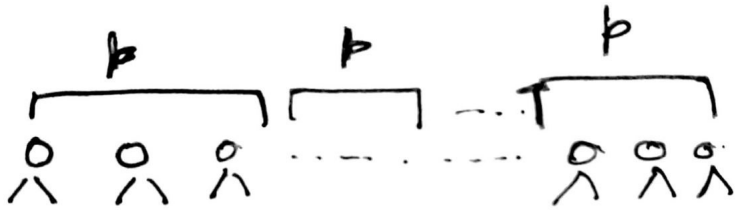
$$W(n) = O(bn) + 2 \cdot O(b) \equiv O(bn) \equiv O(n) \text{ (Considering b to be a constant)}$$

Exercise 5

We are making the obvious assumption that $p < n - 1$, since if $p \geq n - 1$, we don't get any added advantage, because the algorithm on whole uses n-1 processors, and hence, having an excess of processors would not be beneficial in any way. (Time complexity will be O(logn) and work complexity will be O(n)).

In both the upward and downward traversals, we usually use $n/2^i$ processors in one parallel time unit (basically in one "level" in the tree visualization of the processors, and where i is the level of node w.r.t the root). Now, when $n > p$, we will have levels where the number of operations required is $l = n/2^k > p$ in the beginning. In that case we will use l/p units of parallel time by running a batch of p nodes at a time in one level (as show below).

(Note that we will be reusing processors between levels, unlike the original algorithm, since only processors in the same level run together in a given unit of parallel time, making the need for processors being available on other levels other than the level running calculations redundant.)



Time complexity: Previously we said the upward traversal step and downward traversal steps each happen in O(1) unit time, since we had enough number of processors at our disposal to run these computations for all elements in the level at parallel. But now since we run l/p batches for a particular level with l elements, the time taken for a particular process will be l/p time units.

Now, also note that at some level x (measured from the bottom), $n/2^x \leq p$. At that level, the number of processors are ≥ number of elements to be processed. Hence, the overall time for the upward traversal from level x to level n (the root) and the downward traversal from the

root to level x would be O(log(p)) time.

$$\begin{aligned} T(n) &= O\left(\frac{n}{2p} + \frac{n}{4p} + ... + \frac{n}{2^x \cdot p}\right) + O(\log p) \\ &= O\left(\frac{n}{p} \cdot (1/2 + 1/4 + ...1/2^x)\right) + O(\log p) \\ &= O\left(\frac{n}{p} \cdot (1 - 1/2^{x+1})\right) + O(\log p) \end{aligned}$$

Now, to get 2^x, we take $n/2^x = p \implies 1/2^x = p/n$. So,

$$\begin{aligned} T(n) &= O\left(\frac{n}{p} \cdot \left\{1 - \frac{p}{2n}\right\}\right) + O(\log p) \\ &= O\left(\frac{n}{p} - \frac{1}{2}\right) + O(\log p) \\ &= O\left(\frac{n}{p}\right) + O(\log p) \end{aligned}$$

Work complexity: Although we are batching the upward and downward traversals in order to operate on p processors at a time, we do not introduce or remove any steps of the algorithm i.e. the work done in Step 1 and Step 3 would be O(1) operations by p processors in n/p batches $\equiv O(1) \cdot p \cdot (n/p) = O(n)$. Step 2 is a recursive operation, where the work done will still be W(n/2) work. Hence the recursion equation still remains the same from the original algorithm i.e.

$$\begin{aligned} W(n) &= W(n/2) + O(n) \\ W(n) &= O(n) \end{aligned}$$