Compsci 330 Design and Analysis of Algorithms
Assignment 3, Spring 2024 Duke University

TODO: Add your name(s) here

Due Date: Thursday, February 8, 2024

How to Do Homework. We recommend the following three step process for homework to help you learn and prepare for exams.

1. Give yourself 15-20 minutes per problem to try to solve on your own, without help or external materials, as if you were taking an exam. Try to brainstorm and sketch the algorithm for applied problems. Don't try to type anything yet.
2. After a break, review your answers. Lookup resources or get help (from peers, office hours, Ed discussion, etc.) about problems you weren't sure about.
3. Rework the problems, fill in the details, and typeset your final solutions.

Typesetting and Submission. Your solutions should be typed and submitted as a single pdf on Gradescope. Handwritten solutions or pdf files that cannot be opened will not be graded. LaTeX[1] is preferred but not required. You must mark the locations of your solutions to individual problems on Gradescope as explained in the documentation. Any applied problems will request that you submit code separately on Gradescope to be autograded.

Writing Expectations. If you are asked to provide an algorithm, you should clearly and unambiguously define every step of the procedure as a combination of precise sentences in plain English or pseudocode. If you are asked to explain your algorithm, its runtime complexity, or argue for its correctness, your written answers should be clear, concise, and should show your work. Do not skip details but do not write paragraphs where a sentence suffices.

Collaboration and Internet. If you wish, you can work with a single partner (that is, in groups of 2), in which case you should submit a single solution as a group on gradescope. You can use the internet, but looking up solutions or using large language models is unlikely to help you prepare for exams. See the course policies webpage for more details.

Grading. Theory problems will be graded by TAs on an S/U scale (for each sub-problem). Applied problems typically have a separate autograder where you can see your score. The lowest scoring problem is dropped. See the course assignments webpage for more details.

[1] If you are new to LaTeX, you can download it for free at latex-project.org or you can use the popular and free (for a personal account) cloud-editor overleaf.com. We also recommend overleaf.com/learn for tutorials and reference.

Problem 1 (Parallel Prefix Sums). Given an array A of n integers, we want to compute an array B of the prefix sums of A. That is, for $0 \leq i < n$, $B[i] = \sum_{j=0}^{i} A[j]$. For example, if $A = [1, 2, 5]$ then the prefix sums would be $B = [1, 3, 8]$.

Describe a parallel algorithm for computing B with $O((\log(n))^2)$ span and $O(n \log(n))$ work. Analyze the total work and span of the algorithm. Prove the correctness of the serialized (non-parallel) algorithm.

Hint. Be careful to note both the work and span requirement; if you try to compute each prefix sum independently the result will likely have quadratic work. Instead consider a divide and conquer algorithm and try to parallelize it.

Solution 1. The recursive procedure PrefixSum(A, B, ℓ, r) computes the prefix-sum of the subarray A from ℓ to r and stores them in B from ℓ to r. We use a parallelized divide-and-conquer approach, with a parallel for loop for the merge step. We first divide the array A into two halves, and spawn a child process so each call is handled in parallel. At each step, we use a parallel for loop to add the value at the current middle index of B to each index on its right. Initially we create an empty array B and call PrefixSum$(A, B, 0, n-1)$.

```
1:  procedure PrefixSum(A, B, ℓ, r)
2:      if r = ℓ then
3:          B[ℓ] = A[ℓ]
4:      m ← ⌊(r + ℓ)/2⌋
5:      spawn PrefixSum(A, B, ℓ, m)
6:      PrefixSum(A, B, m + 1, r)
7:      sync
8:      parallel for i = m + 1 to r do
9:          B[i] = B[i] + B[m]
10:     end parallel for
```

Span and Work. Let $T_\infty(n)$ be the span of the algorithm for input length n. We write the recurrence relation as $T_\infty(n) = T_\infty(n/2) + c \cdot \log(n)$, and $T(1) = 1$. The $\log(n)$ term is the span from the parallel for loop; the factor $n/2$ comes from dividing the input into half, and only having the single $T_\infty(n)$ term is because a single processor only needs to handle one of the child processes in the divide and conquer. The solution to the recurrence is $T_\infty(n) = O((\log(n))^2)$.

The total work done can be found by acting as if we only had a single processor. With only one processor, we need to account for both steps of the divide and conquer, and the looping in the merge step is always $O(n)$ rather than $O(1)$. Thus, we can write the recurrence as $W(n) = 2W(n/2) + O(n)$. Its solution is $W(n) = O(n \log n)$.

Correctness. We proceed by induction on $n = r - l + 1$, the number of elements in the range. In the base case of $n = 1$, the prefix-sum of a single element is just that element.

For the inductive hypothesis (IH), suppose PrefixSum(A, B, ℓ, r) correctly computes the prefix sum for ranges of size less than n. Consider a range of size n. By the IH, after the recursive call PrefixSum(A, B, ℓ, m), we have

$$B[i] = \sum_{j=\ell}^{i} A[j] \quad \text{for all } \ell \leq j \leq m.$$

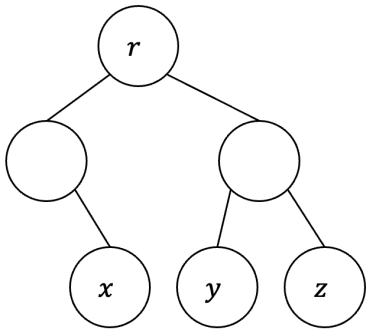
These prefix sums are thus already correctly computed according to the definition. Also, Note in particular that $B[m] = \sum_{j=\ell}^{m} A[j]$. Similarly by the IH, after the recursive call PREFIX-SUM$(A, B, m+1, r)$, we have

$$B[i] = \sum_{j=m+1}^{i} A[j] \quad \text{for all } m+1 \leq j \leq r.$$

Then after adding $B[m] = \sum_{j=\ell}^{m} A[j]$ to each such entry in the right half in the loop on line 8, we have

$$B[i] + B[m] = \left(\sum_{j=m+1}^{i} A[j] \right) + \left(\sum_{j=\ell}^{m} A[j] \right) = \sum_{j=\ell}^{i} A[j] \quad \text{for all } m+1 \leq j \leq r.$$

Problem 2 (Parallel Binary Tree Paths). You are given a binary tree rooted at at node r. Each node in the tree has a left and right child node reference; a NULL reference indicates there is no child. Say that the *distance* between two distinct nodes is one more than the number of nodes on the simple (not repeating nodes) path between them. In the example drawn below, the distance between x and y is 4 (via the path going through the root) whereas the distance between y and z is 2 (via the path through their mutual parent).



Describe a parallel divide and conquer algorithm to compute the maximum distance between any two nodes in the binary tree rooted at r with $O(h)$ span (where h is the height of the tree) and $O(n)$ work (where n is the number of nodes in the tree). Analyze the total work and span of the algorithm. Prove the correctness of the serialized (non-parallel) algorithm.

Hint. Does the greatest distance path necessarily go through the root?

Solution 2. We design a recursive divide and conquer algorithm based on the hint: The maximum distance path might go through the root, otherwise it must be entirely contained in the left or the right subtree (which we can get through recursive calls). If it does pass through the root, then the distance is one more than the greatest distance between the left child and any node in the left subtree plus the greatest distance between the right child and any node in the right subtree.

To keep track of all this information, the algorithm returns both (i) r_d, the maximum distance between any pair of nodes in the subtree and (ii) r_p, the maximum distance between the root and any node in the subtree.

```
1:  procedure MAXDIST(r)
2:      if r is NULL then
3:          return 0, 0
4:      else
5:          spawn x_d, x_p = MAXDIST(r.left)
6:          y_d, y_p = MAXDIST(r.right)
7:          sync
8:      r_p = 1 + max(x_p, y_p)
9:      r_d = max(x_d, y_d, x_p + y_p)
10:     return r_d, r_p
```

Span and Work. The algorithm makes exactly one recursive call on every node in the tree with all other operations constant time for a total of $O(n)$ work. The recursive calls on lines 5 and 6 are computed in parallel. As the calls start from the root, the span is dominated by the longest root to leaf path in the tree, which is $O(h)$ where h is the height.

Correctness. We proceed by induction on the size of the tree. In the base case of $n = 0$, the distance and height are both 0.

For the inductive hypothesis (IH), suppose that for any tree of size less than n, MAXDIST(r) returns (i) r_d, the maximum distance between any pair of nodes in the subtree rooted at r and (ii) r_p, the maximum distance between the root and any node in the subtree rooted at r.

Consider an arbitrary tree T_r rooted at r with n nodes. Note that the IH implies that x_d and x_p correctly characterize quantities (i) and (ii) respectively on the subtree rooted at r.left and similarly y_d and y_p on the subtree rooted at r.right. Suppose the maximum distance is between nodes a and b. Consider two cases.

1. a and b are both in the same subtree from r; without loss of generality, suppose both are in the subtree rooted at r.left. Then the maximum distance is x_d, which is greater than y_d and $x_p + y_p$ and is returned.
2. Else, one of a or b (without loss of generality, suppose a) is in a subtree from r (without loss of generality, suppose the subtree rooted at r.left.) and the other (b) is either equal to r or is in the other subtree (rooted at r.right). Then the maximum distance is $x_p + y_p$, which is greater than x_d and y_d and is returned.

Problem 3 (Parallel Rearrange). Consider an array A of length n in which $m \leq n$ elements are positive numbers, and the remaining elements are 0.

Describe a parallel algorithm with $O((\log(n))^2)$ span and $O(n \log(n))$ work to rearrange the elements in A such that all the m positive numbers are in the first m positions of the array (in any order) with zeros following. For example, if the input is $A = [0, 1, 0, 3, 2, 0, 0]$ then the rearranged array could be $[1, 3, 2, 0, 0, 0, 0]$.

Analyze the total work and span of the algorithm. You do not need to prove the correctness of the algorithm, but be sure to explain it clearly.

Hint. Consider first devising a divide-and-conquer strategy for the problem, and then think about how you might parallelize this approach.

Solution 3. We use a recursive divide and conquer algorithm REARRANGE(A, l, r) that divides the array A into two halves, rearranges each half so that the positive items are in front and then merges two halves. The procedure returns the number of positive items in A between l and r. The merge step copies the positive items in the second half of the range, from the second recursive call, into a temporary array B in parallel, then copies those elelements back onto the next open positions in A in parallel. Finally, 0s are filled in any remainder of the range.

```
1:  procedure REARRANGE(A, l, r)
2:      if l = r then
3:          if A[l] = 0 then
4:              return 0
5:          else
6:              return 1
7:      else
8:          m = ⌊(l+r)/2⌋
9:          spawn n1 = REARRANGE(A, l, m)
10:         n2 = REARRANGE(A, m + 1, r)
11:         sync
12:         B = new array of length n2
13:         parallel for i = 0 to n2 − 1 do
14:             B[i] = A[m + 1 + i]
15:         end parallel for
16:         parallel for i = 0 to n2 − 1 do
17:             A[l + n1 + i] = B[i]
18:         end parallel for
19:         parallel for i = l + n1 + n2 to r do
20:             A[i] = 0
21:         end parallel for
22:         return n1 + n2
```

Correctness. The proof of correctness is by induction on the size of the subproblem. If $r = l$, i.e., the size is 1 then there is nothing to rearrange and the procedure correctly returns the number of positive elements, 0 or 1 depending on whether $A[l]$ is positive.

Assume the procedure is correct for subproblems of size $r - l + 1$ less than n. Consider a subproblem of size $r - l + 1 = n$. By assumption, after the recursive calls on line 9 and 10, we have the following

properties:

(i) $A[i] > 0$ for all i in $[l \ldots l + n_1]$ (the first part of the left subarray)

(ii) $A[i] = 0$ for all i in $[l + n_1 \ldots m]$ (the second part of the left subarray)

(iii) $A[i] > 0$ for all i in $[m \ldots l + n_1]$ (the first part of the right subarray)

(iv) $A[i] = 0$ for all i in $[m + 1 + n_2 \ldots r]$ (the second part of the right subarray)

Thereafter, we consider three for loops:

- The first loop on lines 13-15 copies the positive elements from the first part of the right subarray (iii) into an auxiliary array B. An auxiliary memory is used to avoid a possible determinacy race condition that could otherwise arise if we directly copy these elements into the first part of A, as that operation may run over into the second part of A.
- The second loop on lines 16-18 copies the same elements from B back onto A beginning at the first nonzero element in the left subarray (i-ii). At this point, all of the positive elements are in the first $n_1 + n_2$ positions of A.
- The final loop on lines 19-21 simply writes 0s after the first $n_1 + n_2$ positive elements to meet the problem requirement. Note that otherwise, due to the use of the additional array B, it is possible that there are still some positive elements duplicated in A.

The final return value of $n_1 + n_2$ follows straightforwardly from the inductive hypothesis: The number of positive elements overall is simply the number from the left and the right subarrays added together.

Span and Work Analysis. Let $T_\infty(n)$ be the span for an input of size n. A single core only needs to compute one of the two recursive calls on an input of half the size. The parallel for loops in the merge step have $O(\log(n))$ span. So the span is characterized by the recurrence

$$T_\infty(n) \leq T_\infty\left(\frac{n}{2}\right) + c \cdot \log(n).$$

The solution is $T_\infty(n)$ is $O((\log(n))^2)$.

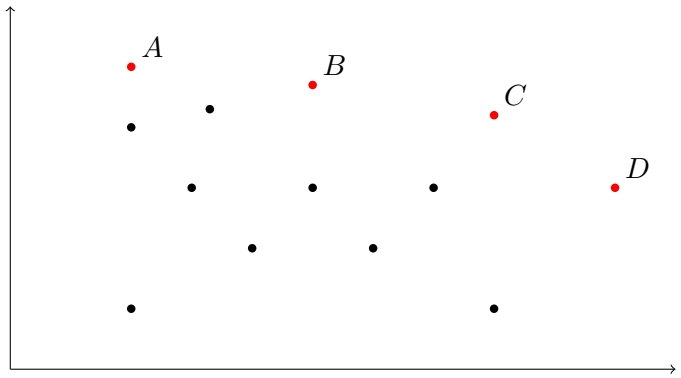
Let $W(n)$ be the total work on an input of size n. There are two recursive calls, each on an input of half the size. The for loops are all $O(n)$. So in total the work is characterized by the recurrence

$$W(n) \leq 2W\left(\frac{n}{2}\right) + c \cdot n.$$

The solution is $W(n)$ is $O(n \log n)$.

Problem 4 (Parallel Maximal Points). You are given an array $P = [(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})]$ of n unique points in the two-dimensional Euclidean plane sorted by x-coordinates with ties broken by y-coordinate (that is, $x_1 \leq x_2 \leq \cdots x_n$, and if $x_i = x_{i+1}$ then $y_i < y_{i+1}$). A point (x_i, y_i) *dominates* another point (x_j, y_j) if $x_i > x_j$ and $y_i \geq y_j$ or $x_i \geq x_j$ and $y_i > y_j$ (that is, if it is up and to the right). A point is *maximal* if it is not dominated by any other point.

See below for an illustration where the red points A, B, C, D are all maximal points in the diagram.



Describe a *parallel* algorithm to compute all maximal points in P with $O((\log(n))^3)$ span and $O(n(\log(n))^2)$ work. Analyze the work and span of your algorithm. You do not need to prove the correctness of the algorithm, but be sure to explain it clearly.

Hint. Note that the input is assumed to be sorted. See last week's homework and think about how to parallelize it. Be careful not to introduce a determinancy race.

Soltuion 4. The algorithm follows the general divide and conquer algorithm shown previously. Here we emphasize the changes to parallelize the algorithm, focusing largely on the parallel helper procedures that are introduced. As for the main PMaximalPoints algorithm, the recursive calls are simply run in parallel and then the new parallel helper procedures are used. To simplify the discussion, we discuss the work and span of helper procedures as we describe them

PConcat(A, B) simply returns a new array containing all of the elements from A and B. This can be done in parallel by copying all elements from A in a parallel for loop and then all from B. The work is linear in the size of A and B, and the span is logarithmic in the size of A and B.

PMaxY(A, i, j) returns the maximum y value among the points in A between indices i and j. We can do so via parallel divide and conquer by finding, in parallel, the maximum of the left and right half of the range and then returning the greater. The resulting work is linear in the size of the range and the span is logarithmic in the size of the range.

PFiltered(S, i, j, y) returns an array containing all of the points from S between indices i and j with y-value greater than y. It does so by divide and conquer, finding all such points in the left and the right half of the range in parallel, then using the PConcat procedure to combine the results. The work is characterized by the recurrence $W(n) \leq 2W(n/2) + c \cdot n$ is $O(n \log(n))$ where n is the number of points in the range and the linear term comes from the call to PConcat. The span is characterized by the recurrence $T_\infty(n) \leq T_\infty(n/2) + c \cdot \log(n)$ is $O((\log(n))^2)$ where the $\log(n)$ term again comes from the span of PConcat.

```
1: procedure PConcat(A, B)
2:     C = new Array of size A.size + B.size
3:     parallel for i = 0 to A.size − 1 do
4:         C[i] = A[i]
5:     end parallel for
6:     parallel for i = 0 to B.size − 1 do
7:         C[A.size + i] = B[i]
8:     end parallel for
9:     return C
```

```
1: procedure PFiltered(S, i, j, y)
2:     if i = j then
3:         Let (x_i, y_i) = S[i]
4:         if y_i > y then
5:             return new Array with S[i]
6:         else
7:             return empty Array
8:     m = ⌊(i + j)/2⌋
9:     spawn A = PFiltered(S, i, m, y)
10:    B = PFiltered(S, m + 1, j, y)
11:    synch
12:    return PConcat(A, B)
```

```
1: procedure PMaxY(A, i, j)
2:     if i=j then
3:         return y-value of A[i]
4:     m = ⌊(i + j)/2⌋
5:     spawn y_l = PMaxY(A, i, m)
6:     y_r = PMaxY(A, m + 1, j)
7:     synch
8:     return max(y_l, y_r)
```

```
1: procedure PMaximalPoints(P, i, j)
2:     if i=j then
3:         return new Array with P[i]
4:     m = ⌊(i + j)/2⌋
5:     spawn L = PMaximalPoints(P, i, m)
6:     R = PMaximalPoints(P, m + 1, j)
7:     sync
8:     y_max = PMaxY(R, 0, R.size − 1)
9:     L_f = PFiltered(L, 0, L.size − 1, y_max)
10:    return PConcat(L_f, R)
```

The outer call is PMaximalPoints$(P, 0, n-1)$. The total work is characterized by the recurrence $W(n) \leq 2W(n/2) + c \cdot n \log(n)$ is $O(n(\log(n))^2)$ where the $n \log(n)$ term comes from the work of the PFiltered procedure which dominates the non-recursive work of a call. The span is characterized by the recurrence $T_\infty(n) \leq T_\infty(n/2) + c \cdot (\log(n))^2$ is $O((\log(n))^3)$ where the $(\log(n))^2$ term again comes from the PFiltered procedure.