Parallel Algorithms	Spring 2022

Problem Set 4

Prof. Nodari Sitchinava — *Due: Wednesday, March 2, 2022 at 10:30am*

You may discuss the problems with your classmates, however **you must write up the solutions on your own** and **list the names** of every person with whom you discussed each problem.

Start **every** problem on a separate page. ***Any problem submitted by 11:59pm Friday February 25, 2022 will receive an additional 10% of the score you receive on that problem.***

1 Parallel Search (50 pts)

In lecture we have learned how to search for some value x in a multiway $(p+1)$-way tree in $O(\log_{p+1} n)$ time using p processors. We also saw a relationship between binary search in a sorted array and a search in a balanced binary search tree (BST). There is a similar relationship between parallel $(p+1)$-way searching in a sorted array and searching in a balanced multiway $(p+1)$-way search tree. In this problem you will design a parallel algorithms to search for a value x in a sorted array in $O(\log_{p+1} n)$ time and $O(p \log_{p+1} n)$ work without constructing the search tree.

Recall the definition of a $(p+1)$-way search tree: Each node v of the search tree contain p keys, $key[1..p]$, and $p+1$ pointers to $p+1$ children, $child[0..p]$. Then in a valid $(p+1)$-way search tree, every node v stores in the subtrees rooted at the i-th child, the following elements:

- All keys stored in the subtree rooted at $child[0]$ are smaller or equal to $key[1]$.
- If $1 \le i \le p-1$, all keys stored in the subtree rooted at $child[i]$ are greater than $key[i]$ and smaller than or equal to $key[i+1]$
- All keys stored in $child[p]$ are greater than $key[p]$.

(a) **(10 pts)** One way to search in a sorted array of items is to first construct a balanced $(p+1)$-way search tree on the elements of the array. Given a sorted array $A[1..n]$ of n items, which elements of A will be in the root of the tree? Make sure your answer works for any p, not just a multiple of n, including $p > n$.

(b) **(10 pts)** Let $A[l] = key[i]$ and $A[r] = key[i+1]$ in a node v of a search tree. Which elements of array A would be stored in the subtree rooted at $child[i]$ of v?

(c) **(30 pts)** Design an $(p+1)$-way searching algorithm on a sorted array ***without*** explicitly constructing the search tree. Write down the pseudocode and prove its correctness. Analyze the time of your algorithm if it were implemented in a p-processor CREW PRAM model. *Hint: generalize binary search to $(p+1)$-way search using the information from parts (a) and (b).*

MULTIWAYSEARCH$(x, A[l..r], p)$ $\quad\triangleright$ Returns the largest index i, such that $A[i] \le x$

.

.

.

2 Parallel Merging (50 pts)

In lecture we have seen a work-efficient merging algorithm that runs in $T(n) = O(\log n)$ time and $W(n) = O(n)$ work. To achieve these bounds, the algorithm ensured that each subproblem that was solved sequentially at the base case was on subarrays B_i and C_i of size at most $\log n$ elements, each. However, each B_i or C_i could be smaller than $\log n$ elements, resulting in some processors doing less work than others. While asymptotically speaking it results in $T(n) = O(\log n)$ time, constant factors do matter in practice and, ideally, each processor would perform exactly the same amount of work. In this problem you will design a merging algorithm where the work among the processors is perfectly balanced, i.e., each processor will (sequentially) merge the subarrays B_i and C_i, such that $|B_i| + |C_i| = \log n$. That is the output array A_i of each processor will have ***exactly*** $\log n$ elements.

Let B and C be two sorted arrays, each containing n elements.

(a) **(5 pts)** Write down pseudocode for a ***sequential*** algorithm to merge the two arrays into another array A of size $2n$. Your algorithm should run in time $T_1(n) = O(n)$ time.

(b) **(15 pts)** Let $A = B \cup C$ be the merge of B and C and let x be the median of A, i.e., $\textsc{rank}(x, A) = n$. Design an $O(\log n)$-time ***sequential*** algorithm to find x ***without computing*** A. Your algorithm should find the indices $i = \textsc{rank}(x, B)$, and $j = \textsc{rank}(x, C)$. Note that your algorithm doesn't know the value of x, but only its rank in A.

Write down the pseudocode for your algorithm, prove its correctness, and analyze its running time. Don't forget to state your invariant/inductive hypothesis before using/proving it.

*Hint: Use binary search. That is start with $i = \lceil \frac{n}{2} \rceil$ and $j = \lceil \frac{n}{2} \rceil$ and reason about where x can and cannot be in B and C relative to $B[i]$ and $C[j]$. To solve this problem, you should understand **why** binary search works and use a similar argument.*

(c) **(5 pts)** Design a parallel algorithm for 2 processors that merges the arrays A and B into another array C and runs in time $T_2(n) = \frac{T_1(n)}{2} + O(\log n)$. Analyze the running time of your algorithm. *(Hint: use the median computation from part (b) to make sure that each processor compares at most n elements during the merging step.)*

(d) **(10 pts)** Now let x be the k-th smallest element in $A = B \cup C$, instead of the median. Design an $O(\log n)$-time ***sequential*** algorithm to find the indices $i = \textsc{rank}(x, B)$, and $j = \textsc{rank}(x, C)$ without knowing the value of x and without computing A. Write down the pseudocode, prove its correctness and analyze its running time. *Hint: generalize your solution in part (b).*

(e) **(10 pts)** Design a parallel algorithm that computes the merged array $A = B \cup C$ in time $\left\lceil \frac{T_1(n)}{p} \right\rceil + O(\log n)$ using p processors. Analyze the running time of your new algorithm. *Hint: Generalize your solution in part (c) to work with p processors, instead of just 2.*

(f) **(5 pts)** What is the maximum value of p for which your algorithm in part (e) remains work-optimal? What does this mean about the time $T(n)$ and work $W(n)$ of your merging algorithm (without the mention of p)?