INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

	Stamp / Signature of the Invigilator

EXAMINATION (End Semester)	SEMESTER (Autumn)

Roll Number									Section		Name	
Subject Number	C	S	6	0	0	2	7	Subject Name	*Parallel Algorithms*			

Department / Center of the Student		Additional sheets	

Important Instructions and Guidelines for Students

1. You must occupy your seat as per the Examination Schedule/Sitting Plan.
2. Do not keep mobile phones or any similar electronic gadgets with you even in the switched off mode.
3. Loose papers, class notes, books or any such materials must not be in your possession, even if they are irrelevant to the subject you are taking examination.
4. Data book, codes, graph papers, relevant standard tables/charts or any other materials are allowed only when instructed by the paper-setter.
5. Use of instrument box, pencil box and non-programmable calculator is allowed during the examination. However, exchange of these items or any other papers (including question papers) is not permitted.
6. Write on both sides of the answer script and do not tear off any page. **Use last page(s) of the answer script for rough work.** Report to the invigilator if the answer script has torn or distorted page(s).
7. It is your responsibility to ensure that you have signed the Attendance Sheet. Keep your Admit Card/Identity Card on the desk for checking by the invigilator.
8. You may leave the examination hall for wash room or for drinking water for a very short period. Record your absence from the Examination Hall in the register provided. Smoking and the consumption of any kind of beverages are strictly prohibited inside the Examination Hall.
9. Do not leave the Examination Hall without submitting your answer script to the invigilator. **In any case, you are not allowed to take away the answer script with you.** After the completion of the examination, do not leave the seat until the invigilators collect all the answer scripts.
10. During the examination, either inside or outside the Examination Hall, gathering information from any kind of sources or exchanging information with others or any such attempt will be treated as **'unfair means'**. Do not adopt unfair means and do not indulge in unseemly behavior.

Violation of any of the above instructions may lead to severe punishment.

Signature of the Student

To be filled in by the examiner

Question Number	1	2	3	4	5	6	7	8	9	10	Total
Marks Obtained											

Marks obtained (in words)	Signature of the Examiner	Signature of the Scrutineer

CS60027 Parallel Algorithms, Autumn 2023–2024

End-Semester Test

24–November–2023 09:00am–12:00pm Maximum marks: 80

[*Write your answers in the question paper itself. Be brief and precise. Answer all questions.*]

Do not write anything on this page.

Questions start from the next page.

1. You are given an unsorted array $A = (a_1, a_2, \ldots, a_n)$ of n distinct positive integers, and m colors numbered $1, 2, \ldots, m$, where $m = O(\log n)$. Each element a_i is given a unique color $c_i \in \{1, 2, \ldots, m\}$. The element colors are supplied in another array $C = (c_1, c_2, \ldots, c_n)$. Your task is to find for each color j, the smallest element b_j of A having that color. If no element of A has the color j, take $b_j = \infty$. Propose an optimal EREW PRAM algorithm to compute the array $B = (b_1, b_2, \ldots, b_m)$ from A and C. Your algorithm should run in $O(\log n)$ time using a total of $O(n)$ operations (justify these performance bounds). You are not allowed to use any sorting algorithm. **(10)**

Solution We use the idea of partitioning. Let $r = \lceil n/m \rceil$. Partition A into r contiguous chunks, each of size m (the last chunk may be smaller). An $m \times r$ array D is used to store the chunk minimums for each color. The size of D is $mr = m\lceil n/m \rceil = O(n)$. The steps of the algorithm are given below.

```
1. For $j = 1, 2, \ldots, m$ and $k = 1, 2, \ldots, r$, pardo:
       Set $D[j,k] := \infty$.
2. For $k = 1, 2, \ldots, r$, pardo:
       For $i = (k-1)m+1, (k-1)m+2, \ldots, (k-1)m+m$, do:
           If $(i \leqslant n)$, then:
               Set $D[C[i],k] := \min\Big(D[C[i],k], A[i]\Big)$
3. For $j = 1, 2, \ldots, m$, pardo:
       Compute $B[j] = \min\Big(D[j,1], D[j,2], \ldots, D[j,r]\Big)$ in parallel by BBT technique.
```

First note that none of the steps in this algorithm requires concurrent read or concurrent write. Initially, n, m, r may be copied to all the processors within the given WT bounds.

Since the size of D is $O(n)$, Step 1 takes $O(1)$ time and uses $O(mr) = O(n)$ operations.

The inner for loop of Step 2 is sequential. Therefore the parallel running time of this step is $O(m) = O(\log n)$, and the work done is $O(rm)$, that is, $O(n)$.

Finally, each minimum in Step 3 can be computed in $O(\log r)$ time using $O(r)$ operations. Since $r = n/m \leqslant n$, the running time is $O(\log n)$. Moreover, the total work done in Step 3 is $O(mr) = O(n)$.

2. Let $G = (V, E)$ be a connected undirected graph with $V = \{1, 2, \ldots, n\}$, and with E presented as an $n \times n$ adjacency matrix A. Assume that $A[i, j] = 1$ if the vertices i and j share an edge, and $A[i, j] = 0$ otherwise. You are also given another $n \times n$ matrix D such that $D[i, j]$ stores the shortest distance between i and j (where this distance is measured by the number of edges on a shortest i, j path). Propose $O(1)$-time parallel algorithms for solving the problems in the following two parts (write a WT-level pseudocode for each algorithm). In each case, specify the type of the PRAM that you use. Mention the most restrictive type that is needed by your algorithm (in terms of restriction, take EREW $>$ CREW $>$ common CRCW $>$ priority/arbitrary CRCW). For each algorithm, mention the work done, and justify the correctness. The two algorithms should be independent in the sense that no one should require the other as a subroutine.

(a) Given a vertex $s \in V$, output a BFS tree of G rooted at s. The tree should be written to the shared memory in the parent-pointer representation. **(10)**

Solution The pseudocode is given below.

```
Set $parent(s) := 0$.
For $i = 1, 2, \ldots, n$ and for $j = 1, 2, \ldots, n$, pardo:
    If $(A[i, j] = 1)$ and $(D[s, i] = D[s, j] + 1)$, then:
        Set $parent(i) := j$.
```

Type of PRAM: priority or arbitrary CRCW.

Work done is $O(n^2)$.

Correctness: Every node at level $l \geqslant 1$ is connected to a (unique) node at level $l - 1$. By induction, this gives a spanning tree of G. Since the levels of the nodes in the tree are the respective distances of the nodes from the root, the tree is a BFS tree.

Note that if $A[i, j] = 1$, then $|D[s, i] - D[s, j]| \in \{0, 1, -1\}$. Therefore the condition $(D[s, i] = D[s, j] + 1)$ can be replaced by $(D[s, i] > D[s, j])$.

(b) Decide whether G is a bipartite graph. The decision should be written to a global variable in the shared memory. **(10)**

Solution We use a global variable *IS_BIP* to store the decision.

```
Initialize $IS\_BIP := 1$.
Pick any arbitrary vertex $s$.
For $i = 1, 2, \ldots, n$ and for $j = 1, 2, \ldots, n$, pardo:
    If $(A[i, j] = 1)$, and $D[s, i]$ and $D[s, j]$ are of the same parity, then:
        Set $IS\_BIP := 0$.
```

Type of PRAM: common CRCW.

Work done is $O(n^2)$.

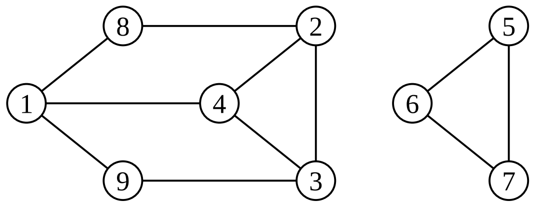
Correctness: If the condition in the body of the for loop is never satisfied, then put all the vertices at even distances from s in one part, and all the vertices at odd distances from s in another part. This gives a bipartitioning of V. Conversely, suppose that the condition in the body of the for loop is satisfied by some i and j. Consider a BFS tree T of G rooted at s. Let k be the LCA of i and j in the tree. Then the k, i path in T, the k, j path in T, and the edge (i, j) constitute an odd-length cycle in G.

Note again that if $A[i, j] = 1$, then $|D[s, i] - D[s, j]| \in \{0, 1, -1\}$. The cases ± 1 are not problematic. We only need to avoid the case $D[s.i] = D[s, j]$, so the parity check can be replaced by a check of this condition.

3. Let $G = (V, E)$ be an undirected graph with $V = \{1, 2, \ldots, n\}$. An *independent set* I of G is a subset $I \subseteq V$ such that no two vertices in I share an edge. An independent set I is called *maximal* if $I \cup \{v\}$ is not an independent set for all $v \in V - I$.

Let us represent an independent set I as a sequence $v_1, v_2, \ldots, v_s$ of vertices in the sorted order $v_1 < v_2 < \cdots < v_s$. Such a sequence can be identified with the string $v_1 v_2 \ldots v_s$ over the alphabet $\{1, 2, \ldots, n\}$. The **lexicographically first maximal independent set** (**LFMIS**) is a maximal independent set that appears before all other maximal independent sets, as strings. By definition, the LFMIS of G is unique.

For example, the LFMIS for the following graph is $\{1, 2, 5\}$. Some other maximal independent sets in the graph are $\{1, 2, 6\}$, $\{2, 5, 9\}$, and $\{4, 7, 8, 9\}$.



The decision problem **LFMISP** takes as input the undirected graph $G = (V, E)$ and a vertex $v \in V$, and outputs the decision whether v belongs to the LFMIS of G.

(a) Propose a **sequential** greedy algorithm to compute the LFMIS of G in polynomial time. Establish the correctness and the running time of your algorithm. **(10)**

Solution The algorithm has the following pseudocode.

```
Initialize $I := \emptyset$.
For $v = 1, 2, \ldots, n$ (in that order), do:
    If there is no edge between $v$ and any vertex of $I$, then:
        Set $I := I \cup \{v\}$.
Return $I$.
```

Running time: $O(n^2)$ (can be improved by suitable data structures).

Correctness: First, note that the algorithm produces a maximal independent set (why?).

Let $I = \{v_1, v_2, \ldots, v_k, v_{k+1}, v_{k+2}, \ldots, v_s\}$ be the output of the algorithm. Suppose that I is not the LFMIS. Instead the LFMIS is $J = \{v_1, v_2, \ldots, v_k, w_{k+1}, w_{k+2}, \ldots, w_t\}$. Here, I and J agree on the first k vertices (for some $k \geqslant 0$). After that, either one of I and J ends (both cannot end together because $I \neq J$), or I and J have different vertices at the $(k+1)$-st position (that is, $v_{k+1} \neq w_{k+1}$).

If $k = t$ (that is, J ends after the common prefix), then J is not maximal (in this case, I contains all the vertices of J plus some more because $I \neq J$). So, $k < t$.

Analogously, if $k = s$ (that is, I ends after the common prefix), then I is not maximal, but the algorithm outputs a maximal independent set. So $k < s$.

But then, $w_{k+1} < v_{k+1}$ (because J is lexicographically smaller than I), so w_{k+1} is considered before v_{k+1} in the for loop of the algorithm. Moreover, w_{k+1} is independent of $v_1, v_2, \ldots, v_k$ (they are all in J). Therefore there is no reason why the algorithm skipped w_{k+1} and included v_{k+1}.

(b) Prove that LFMISP is P-complete. (**Hint:** NORCVP) **(10)**

Solution Computing the LFMIS solves LFMISP. By Part (a), LFMISP is in P.

For proving the P-hardness of LFMISP, we make a reduction from NORCVP. Let $\phi = (g_1, g_2, \ldots, g_n)$ be an instance for NORCVP. We make the standard assumptions about ϕ, that is, all inputs are 1, $g_i = \neg(g_j \vee g_k)$ means $j, k < i$, and g_n is the output. We prepare a graph G which is the same as the circuit for ϕ with the vertex ordering $g_1 < g_2 < \cdots < g_n$, and take $v = g_n$. It is easy to see that $\{g_i \mid g_i = 1\}$ is the LFMIS of the graph. Indeed this LFMIS is precisely what will be returned by the algorithm of Part (a) (why?). Therefore g_n is in the LFMIS of the graph if and only if $g_n = 1$ in the circuit.

4. Let $S = \{1, 2, \ldots, m\}$ be a finite set of **constant** size m, and $F = \{\boxdot : S \times S \to S\}$ the set of all binary operations on S. Each such operation $\boxdot$ is specified by an $m \times m$ look-up table $\mathrm{LUT}_{\boxdot}$ such that $\mathrm{LUT}_{\boxdot}(i, j) = i \boxdot j$ for $1 \leqslant i, j \leqslant m$.

Let T be a rooted full binary tree such that the leaves are labeled by elements of S, and each internal node u is labeled by an operation $\boxdot_u \in F$. Assume that T is given in the adjacency-list representation with additional pointers to enable fast Euler-tour computations.

Develop a parallel algorithm to evaluate the expression given by T. State the running time of and the work done by your algorithm. Note that the functions $\boxdot \in F$ need not be commutative or associative. **(10)**

Solution We will perform tree contraction by repeatedly applying the rake operation concurrently to sets of leaves, as done in the class. For each node u of T, let $val(u)$ denote the value of the function represented by the subtree rooted at u.

Let $H = \{\mu : S \to S\}$ be the set of all unary operations on S. For each node u, we maintain a unary operation $\mu_u \in H$ with the following invariant: for each internal node u, with children u_ℓ and u_r, $val(u) = \mu_{u_\ell}(val(u_\ell)) \boxdot_u \mu_{u_r}(val(u_r))$. For each node u, we initially take μ_u to be the identity operation that maps each $s \in S$ to itself. Clearly, the invariant is satisfied initially.

In order to explain how to update the expressions after a rake operation, we let ℓ be a leaf with parent p, grandparent g, and sibling x. Assume that ℓ is the left child of p, and p is the left child of g (other cases can be handled analogously). Let y be the sibling of p. Consider applying rake on ℓ. Before the rake operation, our invariant gives $val(g) = \mu_p(val(p)) \boxdot_g \mu_y(val(y))$, and $val(p) = \mu_\ell(val(\ell)) \boxdot_p \mu_x(val(x))$. We thus have

$$val(g) = \mu_p\Big(\mu_\ell(val(\ell)) \boxdot_p \mu_x(val(x))\Big) \boxdot_g \mu_y(val(y)) = \mu'_x(val(x)) \boxdot_g \mu_y(val(y)),$$

where

$$\mu'_x(z) = \mu_p\Big((\mu_\ell(val(\ell)) \boxdot_p \mu_x(z)\Big) \text{ for all } z \in S.$$

This is well-defined, since $\mu_\ell(val(\ell))$ is a constant in S. After the rake is performed, g becomes the parent of x and y, so the invariant is maintained if we replace μ_x by μ'_x.

The sets F and H have constant sizes, and so too is the description of each $\boxdot \in F$ and each $\mu \in H$. Thus, the above update can be carried out in constant time. By the analysis of tree contraction done in the class, the algorithm runs in $O(\log n)$ time, and uses $O(n)$ number of operations.

5. Let $X = (x_1, x_2, \ldots, x_n)$ be an unsorted array of distinct integers. For $1 \leqslant i \leqslant n$, the nearest smaller value (NSV) of x_i is the element x_j of X such that (i) $j < i$, (ii) $x_j < x_i$, and (iii) $x_k > x_i$ for all k in the range $j+1 \leqslant k \leqslant i-1$. If no such j exists, then the NSV of a_i is undefined. The all nearest smaller value (ANSV) problem for X is the determination of j for each $i \in \{1, 2, \ldots, n\}$ such that $\mathrm{NSV}(x_i) = x_j$ provided that the NSV of x_i is defined. If the NSV of x_i is not defined, the ANSV problem will return the index 0 for that i. For example, the ANSV for $X = (4, 3, 8, 1, 6, 7, 2, 5)$ is $(0, 0, 2, 0, 4, 5, 4, 7)$.

(a) Let ALG be a parallel algorithm that solves the ANSV problem for X in $T(n)$ time using $W(n)$ work. You are given two sorted arrays $A = (a_1, a_2, \ldots, a_m)$ and $B = (b_1, b_2, \ldots, b_n)$ of integers. Assume that these $m+n$ elements are distinct from one another. Use ALG to merge A and B in $T(m+n) + O(1)$ time using $W(m+n) + O(m+n)$ work. Do not use any other merging algorithm covered or not covered in the class. **(10)**

Solution It suffices to compute $\mathrm{rank}(a_i : B)$ for all $1 \leqslant i \leqslant m$, and $\mathrm{rank}(b_j : A)$ for all $1 \leqslant j \leqslant n$. We explain only the computation of all $\mathrm{rank}(a_i : B)$ (the other computation being analogous). Prepare the array $X = (b_1, b_2, \ldots, b_n, a_m, a_{m-1}, \ldots, a_1)$. This takes $O(1)$ time and $O(m+n)$ work. Invoke ALG on X to get the ANSV array U for X. This step takes $T(m+n)$ time and $W(m+n)$ work. Finally, note that $\mathrm{rank}(a_i : B) = U[n + (m+1-i)]$, so all $\mathrm{rank}(a_i : B)$ can be computed from U in $O(1)$ time using $O(m)$ work.

(b) Design the algorithm ALG with $T(n) = \mathrm{O}(\log n)$ and $W(n) = \mathrm{O}(n \log n)$. ALG should meet these bounds on a CREW PRAM. **(10)**

Solution Fix an i. Consider the array $Y_i = (y_{1,i}, y_{2,i}, \ldots, y_{i,i})$, where $y_{k,i} = \min(x_k, x_{k+1}, \ldots, x_i)$ for $1 \leqslant k \leqslant i$. Here, $y_{k,i}$ are the suffix minima of the subarray $\{a_1, a_2, \ldots, a_i\}$. The array Y_i is sorted in the non-decreasing order. NSV of x_i is the last $y_{k,i}$ that is smaller than x_i. However, if such a k does not exist, that is, if $y_{1,i} = x_i$, then the NSV of x_i is not defined. Since Y_i is sorted, the desired k can be found by performing binary search in Y_i. Preparing the entire suffix minima array Y_i and the binary search can be done in $\mathrm{O}(\log i) = \mathrm{O}(\log n)$ time as desired (the procedure for different i can run in parallel). But the work done will be $\mathrm{O}(\sum_i i) = \mathrm{O}(n^2)$.

For performing the binary search, we do not need the entire array Y_i. Only the elements that are queried should be known or computed whenever they are needed. This can be achieved by range minima calls, each of which takes $\mathrm{O}(1)$ sequential time. A data structure S for supporting these queries can be prepared only once for the entire array X in $\mathrm{O}(\log n)$ time and using $\mathrm{O}(n)$ work. After this precomputation, the ANSV computations (for all i) based upon query-driven binary search can be done in $\mathrm{O}(\log n)$ time using $\mathrm{O}(n \log n)$ operations.