An efficient parallel algorithm for the single function coarsest partition problem*

Joseph F. Jájá
Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA, and Department of Electrical Engineering, Institute for Systems Research, University of Maryland, College Park, MD 20742, USA

Kwan Woo Ryu**
Department of Computer Engineering, Kyungpook National University, Daegu, South Korea

Communicated by O.H. Ibarra
Received December 1992
Revised March 1993

Abstract

We describe an efficient parallel algorithm to solve the single function coarsest partition problem. The algorithm runs in $O(\log n)$ time using $O(n \log \log n)$ operations on the arbitrary CRCW PRAM.

The previous best-known algorithms run in $O(\log^2 n)$ time using $O(n \log^2 n)$ operations on the CREW PRAM, and $O(\log n)$ time using $O(n \log n)$ operations on the arbitrary CRCW PRAM. Our solution is based on efficient algorithms for solving several subproblems that are of independent interest. In particular, we present efficient parallel algorithms to find a minimal starting point of a circular string with respect to lexicographic ordering and to sort lexicographically a list of strings of different lengths.

1. Introduction

The single function coarsest partition problem can be described as follows. Given a set $S$ of $n$ elements, an initial partition $B = \{B_1, B_2, \ldots, B_k\}$ of $S$, and a function $f$ on

* Correspondence to: J.F. Jájá, Inst. for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA.

** Supported in part by Nondirected research fund, Korea Research Foundation, 1992.

0304-3975/94/$07.00 \textcopyright 1994$ Elsevier Science B.V. All rights reserved.

SSDI 0304-3975(93)E0135-Q
S, the problem is to form a new partition \( Q = \{ Q_1, Q_2, \ldots, Q_n \} \) in which each set \( Q_i \in Q \) is a subset of some set \( B_j \in B \), and each image set \( f(Q_i) \) is a subset of some set \( Q_j \in Q \); in addition, \( Q \) is the coarsest such partition (i.e. \( Q \) has the fewest number of sets that satisfy the above constraints). This problem has been considered by several authors \([1, 7, 10, 14, 16, 18]\), who mention a number of related applications.

There are two well-known sequential algorithms to solve this problem. An \( O(n \log n) \) time algorithm is given in \([1]\), and a linear-time algorithm appeared later in \([16]\). Several parallel algorithms have also appeared in the literature. In \([14]\), JáJá and Kosaraju provide an \( O(\sqrt{n}) \) time algorithm on a \( \sqrt{n} \times \sqrt{n} \) mesh of processors, and three different PRAM algorithms have appeared in \([7, 10, 18]\). Srikanth \([18]\) describes an \( O(\log^2 n) \) time algorithm that uses \( (n \log^2 n) \) operations on the CREW PRAM; Galley and Iliopoulos \([10]\) describe an \( O(\log n) \) time algorithm that uses \( O(n \log n) \) operations on the arbitrary CRCW PRAM; Cho and Huynh \([7]\) provide an \( O(\log n) \) time algorithm that requires \( O(n^3) \) operations on the EREW PRAM and \( O(n^2) \) operations on the CREW PRAM.

In this paper, we present a parallel algorithm that solves the single function coarsest partition problem in \( O(\log n) \) time using \( O(n \log \log n) \) operations. The corresponding sequential algorithm runs in linear time. The nonlinear number of operations required by our parallel algorithm is due to the use of an integer sorting routine over the range \([1, 2, \ldots, n^{O(1)}]\). All the remaining steps of the parallel algorithm can be executed in \( O(\log n) \) time using a linear number of operations. We use the integer sorting algorithm of \([4]\) which runs in \( O(\log n / \log \log n) \) time using a total of \( O(n \log \log n) \) operations.

We make use of the Arbitrary CRCW PRAM which allows all the processors to read and write from a shared memory location in one step. However, when more than one processor attempt to write into the same location, only one of the processors succeeds and we do not care which one. We sometimes refer to the common CRCW PRAM, a weaker model in which processors must write the same value whenever they attempt a simultaneous write into the same location.

Our solution for the coarsest partition problem is based on efficient algorithms for solving several subproblems that are of independent interest. One such problem is to find a minimal starting point (m.s.p.) of a circular string with respect to lexicographic ordering. This problem is known to admit a sequential linear-time algorithm \([5, 17]\). In a private communication, Vishkin observed that this problem could be solved in \( O(\log n) \) parallel time using \( O(n \log n) \) operations by constructing an appropriate suffix tree, and in \([12]\), Iliopoulos and Smyth developed an \( O(\log n \log \log n) \) time algorithm using \( O(n \log \log n) \) operations. Both algorithms run on the arbitrary CRCW PRAM. We present a more efficient (and simpler) algorithm that uses a completely different strategy and that runs in \( O(\log n) \) time using \( O(n \log \log n) \) operations on the same model.

A second subproblem of independent interest consists of sorting lexicographically a list of strings of different lengths containing altogether \( n \) characters drawn from an alphabet of size polynomial in \( n \). Our algorithm can be used to solve this problem in
O\log n\) time using \O(n \log \log n)\ operations. The best previous known algorithm to solve this problem runs in \Ologn) time using the same number of operations and the same model as ours \[11\].

A third subproblem of independent interest can be stated as follows. Given a set of \(k\) strings whose total length is \(n\), and an equivalence relation on this set that can be checked, for any pair of strings, in \(O(1)\) time with a linear number of operations, partition the set into the corresponding equivalence classes. This problem can be solved in \(O(1)\) time using \(O(nk)\) operations by comparing every pair of strings in parallel, and then deducing the equivalence classes. We present a parallel algorithm that runs in \(O(\log n)\) time using \(O(n)\) operations.

The rest of the paper is organized as follows. In Section 2, the strategy of the overall algorithm is presented, and, the special case when the graph induced by function \(f\) consists of a set of cycles is handled in Section 3. The three subproblems introduced above are solved in Sections 3.1 and 3.2. The tree nodes and some remaining details are covered in Sections 4 and 5.

### 2. The overall strategy

Given a set \(S\) of \(n\) elements, an initial partition \(B = \{B_1, B_2, \ldots, B_k\}\) of \(S\), and a function \(f\) on \(S\), we seek a new partition \(Q = \{Q_1, Q_2, \ldots, Q_m\}\) of \(S\) that satisfies the following conditions:

1. Each set \(Q_i \in Q\) is a subset of some set \(B_j \in B\).
2. Each image set \(f[Q_i] = \{f(x) | x \in Q_i\}\) is a subset of some set \(Q_j \in Q\).
3. \(Q\) is the coarsest partition, i.e. \(Q\) has the fewest number of sets that satisfy the above two conditions.

Without loss of generality, we assume that \(S = \{1, 2, \ldots, n\}\). Hence, the input can be specified by two arrays \(A_f[1..n]\) and \(A_B[1..n]\) of size \(n\) such that \(A_f[x] = f(x)\), and \(A_B[x] = A_B[y]\) iff both \(x\) and \(y\) are in the same set of \(B\). We desire to determine the output as the array \(A_Q[1..n]\) of size \(n\) such that \(A_Q[x] = A_Q[y]\) iff both \(x\) and \(y\) are in the same set of \(Q\). Thus, the single function coarsest partition problem can be regarded as labelling each element of \(S\) according to the partition \(Q\) (\(Q\)-labeling), given the function \(f\) and the partition \(B\) (or \(B\)-labels). Let \(f^0(x) = x\) and \(f^i(x) = f(f^{i-1}(x))\), \(i > 0\). The following simple lemma from \[16\] is helpful in motivating our solution.

**Lemma 2.1.** The following statements hold:

(i) \(\forall x, y \in S, A_Q[x] = A_Q[y]\) iff \(A_B[x] = A_B[y]\) and \(A_Q[f(x)] = A_Q[f(y)]\).

(ii) \(\forall x, y \in S, A_Q[x] = A_Q[y]\) iff \(A_B[f^i(x)] = A_B[f^i(y)]\), \(i = 0, 1, \ldots, n\).

We can translate this problem into the following graph problem (cf. \[14\]). Create a directed graph \(G = (V, E)\) such that \(V = S = \{1, 2, \ldots, n\}\) and \((x, f(x)) \in E, \forall x \in V\). Each node \(x\) is \(B\)-labeled, i.e. assigned the label \(A_B[x]\). We have to relabel each node such
that any two nodes $x$ and $y$ are assigned the same $Q$-label iff both $x$ and $y$ are in the same set of $Q$.

Since the outdegree of each vertex is one, the graph $G=(V,E)$ is a pseudo-forest. Each component of $G$ is a pseudo-tree in which there is exactly one cycle and all the paths end in the cycle. Clearly, statements (i) and (ii) of Lemma 2.1 can be expressed as follows:

(i) Any two nodes $x$ and $y$ of $V$ have the same $Q$-label iff $x$ and $y$ have the same $B$-label, and the parents of $x$ and $y$ have the same $Q$-label.

(ii) Let $(x=x_0, x_1, \ldots, x_n)$ and $(y=y_0, y_1, \ldots, y_n)$ be two directed paths of length $n$ starting from $x$ and $y$, respectively. Note that $x_i = f^i(x)$ and $y_i = f^i(y)$, $i=0, 1, \ldots, n$. Then nodes $x$ and $y$ have the same $Q$-label iff nodes $x_i$ and $y_i$ have the same $B$-label, where $i=0, 1, \ldots, n$.

Example 2.2. Given a function $f$ and a partition $B$ represented by the arrays $A_f[1..16]=\{2,4,6,8,10,12,1,3,5,7,9,11,14,15,16,13\}$ and $A_p[1..16]=\{1,2,1,1,2,2,3,3,1,1,3,1,2,1,3\}$. Then $B = \{B_1, B_2, B_3\}$ and $B_1 = \{1,3,4,9,10,12,13,15\}$, $B_2 = \{2,5,6,14\}$, and $B_3 = \{7,8,11,16\}$. The corresponding graph is shown in Fig. 1. Note that it consists of two simple cycles. The $B$-label of a node is listed outside the circle representing the node. Note that nodes 1, 3 and 13 will have the same $Q$-label, and that nodes 1 and 4 cannot have the same $Q$-label.

Determining the $Q$-labels of all the nodes of $V$ can be done by implementing the following strategy on the directed graph.

Algorithm coarsest partition

Step 1: Mark all the cycle nodes in the pseudo-forest.
Step 2: Find the Q-labels of the cycle nodes.
Step 3: Find the Q-labels of the remaining tree nodes.

We explain the implementations of steps 2 and 3 in the following two sections respectively, and the remaining details are presented in the last section.

3. Labelling of cycle nodes

In this section, we consider the coarsest partition problem for a function whose graph consists of a set of cycles with no entering edges. We begin with a few definitions. Given a string \( S \) and a positive integer \( i \), \( S^i \) represents the string \( S \) concatenated with itself \( i \) times. The smallest repeating prefix of a string \( S \) is the shortest prefix \( P \) of \( S \) such that \( P^j = S \), for some \( j > 0 \). Note that in this case \( P \) is a period of \( S \). If \( x \) is any node of a cycle \( C \) of length \( k \), then \( C \) can be represented as the circular string \( (x, f(x), f^2(x), \ldots, f^{k-1}(x)) \), together with the B-label string \( (A_B[x], A_B[f(x)], A_B[f^2(x)], \ldots, A_B[f^{k-1}(x)]) \). Let \( P \) be the smallest repeating prefix of the B-label string of \( C \). Consider the sets

\[
C_i = \{ f^j(x) | j = 0, \ldots, k-1 \text{ and } j = i \mod |P| \}, \quad i = 0, \ldots, |P| - 1.
\]

Then, by Lemma 2.1(ii), any two nodes \( x \) and \( y \) from the same set \( C_i \) have the same Q-label, since \( A_B[f^j(x)] = A_B[f^j(y)] \), \( i = 0, 1, \ldots, n \). Similarly, any two nodes from different such sets can not have the same Q-label. Thus, given any two nodes \( x \) and \( y \) in \( C \), \( A_Q[x] = A_Q[y] \) iff \( x, y \in C_i \), for some \( i \). If \( |P| = |C| \), the B-label string of \( C \) is not repeating, and hence every node in \( C \) has a different Q-label.

Example 3.1. Given the function \( f \) and the partition \( B \) introduced in Example 2.2, the corresponding graph has two cycles \( C \) and \( D \). Cycle \( C \) and its B-label string are given by \( (1, 2, 4, 8, 3, 6, 12, 11, 9, 5, 10, 7) \) and \( (1, 2, 1, 3, 1, 2, 1, 3, 1, 2, 1, 3) \), respectively. Hence, the smallest repeating prefix \( P \) of the B-label string is \( (1, 2, 1, 3) \), and

\[
C_0 = \{1, 3, 9\}, \quad C_1 = \{2, 6, 5\}, \quad C_2 = \{4, 12, 10\}, \quad C_3 = \{8, 11, 7\}.
\]

Cycle \( D \) and its B-label string are given by \( (13, 14, 15, 16) \) and \( (1, 2, 1, 3) \), respectively, and hence \( D_0 = \{13\} \), \( D_1 = \{14\} \), \( D_2 = \{15\} \), \( D_3 = \{16\} \). Note that the nodes in \( C_i \cup D_i \), \( i = 0, 1, 2, 3 \), have the same Q-label. If we set \( Q_{i+1} = C_i \cup D_i \), for \( i = 0, 1, 2, 3 \), the output is given by

\[
A_Q[1 \ldots 16] = [1, 2, 1, 3, 2, 2, 4, 4, 1, 3, 4, 3, 1, 2, 3, 4].
\]

Given two distinct cycles \( C \) and \( D \), let \( B_C \) and \( B_D \) be their corresponding B-label strings, and let \( P_C \) and \( P_D \) be the smallest repeating prefixes of \( B_C \) and \( B_D \), respectively. We say that \( P_C \) and \( P_D \) are cyclic shift equivalent (or \( P_C \equiv P_D \)) iff one is the cyclic shift of the other. We also define the two cycles \( C \) and \( D \) to be equivalent iff \( P_C \equiv P_D \). Note that \( C \) and \( D \) need not have the same length even if \( C \) and \( D \) are equivalent. For example, cycles \( C \) and \( D \) of Examples 3.1 are equivalent.
Let \( x \) and \( y \) be a pair of nodes such that \( x \in C \) and \( y \in D \), where \( C \) and \( D \) are equivalent, and let \( |P_C| = |P_D| = l \). Assume that \( P_C = (A_B[x], A_B[f(x)], \ldots, A_B[f^{l-1}(x)]) \) and \( P_D = (A_B[y], A_B[f(y)], \ldots, A_B[f^{l-1}(y)]) \), and \( A_B[f^i(x)] = A_B[f^i(y)] \), \( i = 0, 1, \ldots, l-1 \). Clearly, this can be achieved by shifting \( P_C \) or \( P_D \) cyclically whenever \( C \) and \( D \) are equivalent. Then, \( f^i(x) \) and \( f^i(y) \) must have the same \( Q \)-label, \( i = 0, 1, \ldots, l-1 \). Moreover, if we let
\[
C_i = \{ f^i(x) \mid j = 0, \ldots, |C| - 1 \text{ and } j = i \mod l \}, \quad i = 0, \ldots, l-1
\]
and
\[
D_i = \{ f^i(y) \mid j = 0, \ldots, |D| - 1 \text{ and } j = i \mod l \}, \quad i = 0, \ldots, l-1,
\]
then all the nodes in \( C_i \cup D_i \) have the same \( Q \)-label. That is \( \forall x, y \in C \cup D \), \( A_Q[x] = A_Q[y] \) \iff both \( x \) and \( y \) are in \( C_i \cup D_i \), for some \( i \).

We now describe the algorithm for solving the coarsest partition problem when the input consists of a set of cycles.

**Algorithm cycle node labeling**

**Input:** Two arrays \( A_f[1 \ldots n] \) and \( A_B[1 \ldots n] \) representing the input function \( f \) and the initial partition \( B \), respectively. The graph representation of \( f \) consists of a set of cycles.

**Output:** An array \( A_Q[1 \ldots n] \) such that \( A_Q[x] = A_Q[y] \) \iff both \( x \) and \( y \) have the same \( Q \)-label.

**Begin**

**Step 1:** Rearrange the input arrays \( A_f \) and \( A_B \) such that each cycle \( (x, f(x), \ldots, f^{k-1}(x)) \) and its \( B \)-label string \( (A_B[x], A_B[f(x)], \ldots, A_B[f^{k-1}(x)]) \) occupy consecutive memory locations.

**Step 2:** Partition the input cycles according to the cyclic shift equivalence relation defined above, and assign the appropriate \( Q \)-labels as above.

**End**

The correctness of this algorithm follows from the discussion preceding the statement of the algorithm. We now consider the implementation of each step. Step 1 of the algorithm can be implemented as follows. First, we label each cycle with one of the indices of the cycle, and then rank all the nodes in each cycle starting from the chosen index. We can do this by using the list ranking algorithm that runs in \( O(\log n) \) time using \( O(n) \) operations on the EREW PRAM [2]. Once this information is available, we rearrange the input arrays \( A_f \) and \( A_B \) so that each cycle and its \( B \)-label string occupy consecutive memory locations according to the cyclic ordering. Hence, step 1 can be done in \( O(\log n) \) parallel time using a linear number of operations. Step 2 can be divided into two substeps. In the first substep, we find the smallest repeating prefix of the \( B \)-label string of each cycle, and then find the minimal starting point of the prefix. In the second substep, we partition the cycles into equivalence classes and
deduce the $Q$-labels of the nodes. The following two sections provide the details for implementing step 2 in $O(\log n)$ time using $O(n \log \log n)$ operations. Thus, we have the following lemma.

**Lemma 3.2.** The single function coarsest partition problem can be solved in $O(\log n)$ time using $O(n \log \log n)$ operations on the arbitrary CRCW PRAM if the graph representation of the function is a set of cycles with no entering edges.

### 3.1. Finding a minimal starting point of a circular string and sorting strings

Let $C=(c_0, c_1, \ldots, c_{n-1})$ be a circular string over the alphabet $\{1, 2, \ldots, n\}$. Since $C$ is circular, $C$ has $n$ equivalent representations $(c_j, c_{j+1}, \ldots, c_{n-1}, c_0, \ldots, c_{j+1})$, $j=0, \ldots, n-1$. Let $C(j_0)=[c_{j_0}, \ldots, c_{n-1}, c_0, \ldots, c_{j_0-1}]$ be a linear string of $C$ with $j_0$ as a starting point. Given $C(i)$ and $C(j)$, we define $C(i) < C(j)$ iff $[c_i, \ldots, c_{n-1}, c_0, \ldots, c_{j-1}]$ precedes $[c_j, \ldots, c_{n-1}, c_0, \ldots, c_{j-1}]$ in lexicographic order, and $C(i) \leq C(j)$ iff $C(i) < C(j)$ or $C(i)=C(j)$. $C(j_0)$ is minimal if $C(j) \leq C(j_0)$ for all $j=0, \ldots, n-1$. The index $j_0$ is called a minimal starting point of $C$ (m.s.p.).

Given a circular string, the problem for finding a m.s.p. can be sequentially solved in linear time [5, 17]. In a private communication, Vishkin observed that this problem could be solved in $O(\log n)$ parallel time using $O(n \log n)$ operations by constructing an appropriate suffix tree, and in [12], Iliopoulos and Smyth developed an $O(\log n \log \log n)$ time algorithm using $O(n \log \log n)$ operations. Both algorithms run on the arbitrary CRCW PRAM. We present a parallel algorithm that solves the problem in $O(\log n)$ time using $O(n \log \log n)$ operations on the same model. The algorithm needs $O(n \log \log n)$ operations because we make use of the integer sorting algorithm appearing in [4]. All the other steps can be performed within a linear number of operations. Note that this algorithm can be implemented in linear time on a sequential machine.

In this section, we assume that the input circular string is not repeating, since if it is repeating, we can find its smallest repeating prefix in $O(\log n)$ time and $O(n)$ operations [6, 20]. Clearly, the m.s.p. of a smallest repeating prefix is also an m.s.p. of the original string. One can easily check that there is only one m.s.p. in a nonrepeating circular string. We begin with a simple $O(\log n)$ time algorithm that uses $O(n \log n)$ operations. We use it later to derive a more efficient algorithm. This simple algorithm is based on the following lemma from [17].

**Lemma 3.3.** Let $C=(c_0, c_1, \ldots, c_{n-1})$ be a circular string with a unique m.s.p. If $c_{i+1}=c_{j+1}$ for all $i=1, \ldots, s$, then any of $\{i+1, \ldots, i+s+1\} \cap \{j+1, \ldots, j+s+1\}$ cannot be the m.s.p.
Algorithm simple m.s.p.

**Input:** A nonrepeating circular string \( C = (c_0, c_1, \ldots, c_{n-1}) \).
(Assume for convenience that \( n = 2^k \), for some integer \( k > 0 \).)

**Output:** The m.s.p. of the input string.

**begin**
1. Mark all \( n \) positions as candidates for the m.s.p.
2. for \( i = 1 \) to \( \log n \) do
   for each block of size \( 2^i \) pardo
   (* Note that each block starts at position \( j2^i \), where
   \( 0 \leq j \leq n/2^i - 1 \), and consists of two subblocks of size \( 2^{i-1} \);
   each of the two subblocks has exactly one candidate.*)
   Compare the two overlapping strings each of length \( 2^i \) starting at each of these
   two candidate positions in the block.
   Eliminate one of the two candidates.

**end**

We assign \( 2^i \) processors to each block of size \( 2^i \), and compare the two strings of
length \( 2^i \) each starting from its candidate position. If the two strings are different,
we find the smaller string and mark its starting position as a further candidate. If the two
strings are the same, we mark the first of the two candidates as a further candidate by
using Lemma 3.3. This can be done in constant time using \( O(2^i) \) operations since we
know how to find the position of the first 1 in a Boolean array within the same
complexity bounds [9]. Thus, the above algorithm finds the m.s.p. of a nonrepeating
circular string in \( O(\log n) \) time and \( O(n \log n) \) operations on the common CRCW
PRAM. We now describe the more efficient algorithm.

Algorithm efficient m.s.p.

**Input:** A nonrepeating circular string \( C = (c_0, c_1, \ldots, c_{n-1}) \).
(Assume for convenience that \( n = 2^k \), for some integer \( k > 0 \).)

**Output:** The m.s.p. of the input string.

**begin**

**Step 1:** Let \( m \) be the smallest element in the circular string. Mark all the positions
\( j \) in the string if \( c_j = m \) and \( c_{(j-1) \mod n} \neq m \). Obviously, any of the unmarked positions
cannot be the m.s.p. If only one element is marked, output the position as the m.s.p.
and stop.

**Step 2:** Starting from each marked position, group the elements in ordered pairs
until the next marked position is reached. Note that if the number of elements in this
substring is odd, the last group of the substring has only one element \( c \), and in this case
we represent it as the pair \((c, m)\). For each pair, we maintain its starting point in the
original string.
Step 3: Sort all the ordered pairs lexicographically and replace each pair by its rank. Note that numbers in the range \([1, 2, \ldots, 2n/3]\) are sufficient for this assignment.

Step 4: Run steps 1–3 recursively until the length of the resulting string is at most \(n/\log n\).

Step 5: Apply Algorithm simple m.s.p. on the resulting string and find the final m.s.p.

Example 3.4. Given a circular string \((3, 2, 1, 3, 2, 3, 4, 3, 1, 2, 3, 4, 2, 1, 1, 3, 2, 2)\). Since \(m = 1\), we mark three 1's as follows: \((3, 2, 1, 3, 2, 3, 4, 3, 1, 2, 3, 4, 2, 1, 1, 3, 2, 2)\). After step 2, we have pairs \((1, 3), (2, 3), (4, 3), (1, 2), (3, 4), (2), (1, 1), (1, 3), (2, 2), (3, 2)\). In step 3, the pairs are sorted into \((1, 1), (1, 2), (1, 3), (1, 3)(2), (2, 2), (2, 3), (3, 2), (3, 4), (4, 3)\), and the numbers \(1, 2, 3, 3, 4, 5, 6, 7, 8, 9\) are assigned to the corresponding pairs. Hence, the resulting circular string is \((7, 3, 6, 9, 2, 8, 4, 1, 3, 5)\) with pointers to the appropriate positions in the original string.

Lemma 3.5. The m.s.p. of the modified circular string after step 3 of Algorithm efficient m.s.p. maintains the same m.s.p. as the original one.

Proof. Let \(j_0\) be the m.s.p. of the original circular string the \(j\) any other position such that \(c_j=m\). Then there is \(k>0\) such that \(c_{j_0} = c_j = \ldots = c_{j_0+k} = c_{j+k}\), and \(c_{j_0+k+1} < c_{j+k+1}\) since \(j_0\) is the m.s.p. Since the substring in the modified circular string corresponding to \((c_{j_0}, \ldots, c_{j_0+k+1}, \ldots)\) still precedes the string corresponding to \((c_j, \ldots, c_{j+k+1}, \ldots)\) lexicographically, the m.s.p. is still \(j_0\).

The proof of the following lemma is straightforward.

Lemma 3.6. The length of the modified circular string after step 3 is at most \(2n/3\).

Lemma 3.7. Given a circular string of length \(n\), Algorithm efficient m.s.p. finds its m.s.p. in \(O(\log n)\) time using \(O(n \log \log n)\) operations on the arbitrary CRCW PRAM.

Proof. The correctness of Algorithm efficient m.s.p. follows from Lemma 3.3 and the correctness of Algorithm simple m.s.p. The number of iterations of steps 1–3 is \(O(\log \log n)\) by Lemma 3.6. Steps 1–3 can be done by using the algorithms for computing prefix sums and for integer sorting in \(O(\log n/\log \log n)\) time using \(O(n \log \log n)\) operations. Thus, steps 1–4 can be done in \(O(\log n)\) time and \(O(n \log \log n)\) operations. Step 5 requires \(O(\log n)\) time using \(O(n)\) operations.

We now show that this algorithm can be modified to solve the following sorting problem: Given an alphabet \(\Sigma\) with \(n^{O(1)}\) symbols and a list \(\mathcal{L} = (X_1, X_2, \ldots, X_m)\) of \(m\) strings over the alphabet \(\Sigma\), where \(\sum_{i=1}^{m} |X_i| = n\), rearrange \(\mathcal{L}\) into a sorted list of the strings. There are several algorithms for the string sorting problem. In particular, Aho et. al. [1] provide a sequential algorithm that runs in \(O(n)\) time; Hagerup and Petersson [11] provide a parallel algorithm that runs in \(O(\log^2 n/\log \log n)\) time using...
Algorithm efficient m.s.p., we present an improved simple parallel algorithm that solves the string sorting problem in $O(\log n)$ time using $O(n \log \log n)$ operations on the arbitrary CRCW PRAM.

Algorithm sorting strings

\textbf{Input:} A list $\mathcal{L} = (X_1, X_2, \ldots, X_m)$ of $m$ strings over the alphabet $\Sigma$, where $\sum_{i=1}^m |X_i| = n$ and $|\Sigma| = n^{O(1)}$.

\textbf{Output:} The list $\mathcal{L}$ in sorted order.

\begin{algorithm}
\textbf{begin}

\textbf{Step 1:} Sort the input strings according to their first symbols by using the integer sorting algorithm of [4]; in case of tie, we let unit-length strings precede strings of length greater than one. Since strings of length one are in their correct positions in the sorted list, we now sort strings of length greater than one.

\textbf{Step 2:} Each string $X_i$ is partitioned into $\lceil |X_i|/2 \rceil$ ordered pairs each of which is a string of length two, for $1 \leq i \leq m$. If $|X_i|$ is an odd number and the last symbol of $|X_i|$ is $c$, then the last pair will be $(c, \#)$, where $\#$ is the blank symbol which precedes any symbol in $\Sigma$.

\textbf{Step 3:} Sort all the ordered pairs lexicographically and replace each pair by its rank in the sorted list. Note that this step produces a new list of at most $m$ strings whose total number of symbols is at most $2n/3$, and whose relative ordering is the same as that of the strings of the original list.

\textbf{Step 4:} Run steps 1–3 recursively until the total number of symbols in the resulting strings is at most $n/\log n$.

\textbf{Step 5:} Apply Cole’s mergesort algorithm [8] on the resulting strings and find the final sorted list. This step can be done in $O(\log m)$ time using $O((n/\log n) \log m) = O(n)$ operations on the common CRCW PRAM by using the fact that any two strings can be compared in $O(1)$ time with linear work.

\textbf{end}

\textbf{Lemma 3.8.} Given a list of strings containing altogether $n$ characters drawn from an alphabet of size polynomial in $n$, these strings can be sorted lexicographically in $O(\log n)$ time using $O(n \log \log n)$ operations on the arbitrary CRCW PRAM.

3.2. Partitioning cycles into equivalence classes

In this section, we address the problem of partitioning a set of $k$ cycles, stored in an array of size $n$, into equivalence classes with respect to the coarsest partition relation. We are assuming that each cycle is nonrepeating; otherwise, we can find its smallest repeating prefix and replace the cycle with this prefix. We are also assuming that all the cycles have the same length $l (= n/k)$, since any two nonrepeating cycles of different lengths cannot be equivalent.

Let the $k$ cycles be $C_i = (x_i, f(x_i), \ldots, f^{l-1}(x_i))$, $i = 1, \ldots, k$, and their $B$-labels be $B_{C_i} = (A_B[x_i], A_B[f(x_i)], \ldots, A_B[f^{l-1}(x_i)])$, $i = 1, \ldots, k$. We assume that for each $i$, $C_i$
and \( B_C \), appear consecutively in arrays \( A_C \) and \( A_B \) as \( A_C[1..n]=\{x_1, f(x_1), \ldots, f^{l-1}(x_1), \ldots, x_k, f(x_k), \ldots, f^{l-1}(x_k)\} \) and \( A_B[1..n]=\{A_B[x_1], A_B[f(x_1)], \ldots, A_B[f^{l-1}(x_1)], \ldots, A_B[x_k], A_B[f(x_k)], \ldots, A_B[f^{l-1}(x_k)]\} \), respectively. We also assume that each \( B_C \) starts with its m.s.p. Recall that two nonrepeating cycles \( C_i \) and \( C_j \) are equivalent iff \( B_{C_i} = B_{C_j} \), and note that two cycles \( C_i \) and \( C_j \) are equivalent iff \( B_{C_i} = B_{C_j} \). Given a set of arbitrary cycles, we can manipulate the cycles to satisfy all the above assumptions within our complexity bounds.

We now present an algorithm that partitions the set of cycles into equivalence classes in \( O(\log n) \) time using \( O(n) \) operations. Clearly, this partitioning problem can be done in \( O(1) \) time and \( O(nk) \) operations on the common CRCW PRAM by comparing every pair of cycles concurrently, and then determining the equivalence classes. For convenience, we assume that \( l=2^h \), for some integer \( h>0 \). The algorithm can easily be modified to handle the general case.

The basic idea of the algorithm is to assign a unique label to each node based on the sequence of \( B \)-labels of the cycle starting at that node by using a two-dimensional array \( BB[1..n, 1..n] \). In the \( j \)th iteration, all the starting positions \( i \) of the paths of length \( 2^j \) that have the same sequence of \( B \)-labels will be encoded by the same number as follows. Let \( d_1=i+2^j-1 \) be such a starting position and let \( d_2=d_1+2^j-1 \). Then processor \( P_i \) corresponding to position \( d_1 \) attempts to write its position \( d_1 \) into \( BB[EQ[d_1], EQ[d_2]] \), where \( EQ[x] \) is the encoding of \( x \) generated during the \( j \)th iteration. Note that several processors may try to write their positions into the same location of \( BB \), and only one of them will succeed. Then processor \( P_i \) reads the same location back and assign the value read to \( EQ[d_2] \).

Hence, after \( \log l \) iterations, the starting positions of two cycles will have the same \( EQ \)-label if and only if the cycles are equivalent. Note that a straightforward implementation would have used \( O(n \log n) \) operations.

**Algorithm partition**

**Input:** A set of \( k \) cycles containing a total of \( n \) nodes such that \( l=n/k=2^h \), for some positive integer \( h \).

**Output:** An array \( EQ[1..n] \) of size \( n \). On completing the execution of this algorithm, the starting positions of any two equivalent cycles will have the same \( EQ \)-labels.

**begin**
1. Initialize \( EQ[1..n] \cdot A_B[1..n] \);
2. for \( j=1 \) to \( \log l \) do
   for \( 1 \leq i \leq k \) (for each cycle \( C_i \)) pardo
      for \( p=1, 2^j+1, 2^j+2+1, \ldots, p+2^j-1 \leq l \) pardo
         let \( d_1=(i-1)+p \) and \( d_2=d_1+2^j-1 \);
         \( BB[EQ[d_1], EQ[d_2]] \leftarrow d_1 \);
         \( EQ[d_1] \leftarrow BB[EQ[d_1], EQ[d_2]] \);
   end
end
Suppose that, after iteration \((j-1)\) of the above algorithm, \(d\) and \(d'\) are two positions such that \((d - 1)\) and \((d' - 1)\) are multiples of \(2^j\), and \(EQ[d] = EQ[d']\) and \(EQ[d + 2^j - 1] = EQ[d' + 2^j - 1]\); then, after iteration \(j\), \(EQ[d] = EQ[d']\). Moreover, we can prove the following lemma using an inductive argument.

**Lemma 3.9.** Let \(d\) and \(d'\) be such that \(1 \leq d < d' \leq n\). Then, after iteration \(j\), \(EQ[d] = EQ[d']\) if \((d - 1)\) and \((d' - 1)\) are multiples of \(2^j\), and \(A_B[d..(d + 2^j - 1)] = A_B[d'..(d' + 2^j - 1)]\).

**Corollary 3.10.** On completing the execution of Algorithm partition, any two cycles \(C_i\) and \(C_j\) of length \(l\) are equivalent iff \(EQ[l(i - 1) + 1] = EQ[l(j - 1) + 1]\). In other words, two cycles are equivalent iff their first nodes have the same \(EQ\)-label.

**Lemma 3.11.** Given a set of \(k\) cycles each of length \(n/k\), the cycles can be partitioned into equivalence classes in \(O(\log n)\) time using \(O(n)\) operations on the arbitrary CRCW PRAM.

**Remark.** The arbitrary CRCW PRAM is needed for assigning a unique label through the use of the two-dimensional array \(BB[1..n:1..n]\). The memory space used is \(O(n^2)\) but can be reduced to \(O(n^{1+\epsilon})\) for any constant \(\epsilon > 0\) as shown in [3].

### 4. Tree node labeling

Let \(G = (V, E)\) be the directed graph corresponding to an instance of the coarsest partition problem. Assume that all the cycle nodes have already been \(Q\)-labeled. In this section, we describe how to \(Q\)-label the remaining tree nodes. The tree nodes can be classified into two sets; the first consists of the nodes having the same \(Q\)-labels as the cycle nodes, and the second set consists of the remaining nodes. The following lemma is important to \(Q\)-label the former tree nodes. We assume that each tree \(T\) has been rooted at an arbitrary node of the cycle of \(T\).

**Lemma 4.1.** Let \(T \subset G\) be tree whose root \(r\) belongs to the cycle \(C = (r = f^0(r), f(r), \ldots, f^{k-1}(r))\) of length \(k\). Let \(x\) be any node at level \(l\) in \(T\), where the level of \(r\) is zero. Then, \(A_Q[x] = A_Q[f^{k-1}(r)]\) iff \(A_B[f^{j}(x)] = A_B[f^{k-1+j}(r)]\), \(j = 0, \ldots, l\). In other words, \(x\) has the same \(Q\)-label as one of the cycle nodes of its pseudo-tree iff each node in the path from \(x\) to \(r\) has the same \(B\)-label as its corresponding node in the cycle.

**Proof.** The proof follows from Lemma 2.1(ii). \(\Box\)
Lemma 4.1 also implies that if $x$ is a tree node at level $l$ such that $A_{B}[x] \neq A_{B}[f^{k-1}(r)]$, then no descendant node of $x$ can have a $Q$-label that appears in any of the cycles in $G$. We now state our algorithm.

**Algorithm tree node labeling**

**Input:** A pseudo-forest $G = (V, E)$. All the cycles of $G$ have been $Q$-labeled and stored in consecutive memory locations. Each tree has been rooted at an arbitrary node of its cycle. The trees are stored in the form of adjacency lists suitable for constructing their Euler tours in a linear number of operations.

**Output:** The $Q$-labels of all the tree nodes.

**begin**

**Step 1:** For each tree node $x$, compute its level.

**Step 2:** For each tree node $x$, compare the $B$-label of $x$ with that of its corresponding node of the cycle in the pseudo-tree containing $x$ (see Lemma 4.1), and mark $x$ if they are the same.

**Step 3:** For each unmarked node, unmark all of its descendants.

**Step 4:** $Q$-label all the marked nodes with the $Q$-labels of their corresponding nodes in the cycles.

**Step 5:** Now, construct a forest consisting of all the unmarked nodes. Note that the parent $f(r_i)$ of the root node $r_i$ in each tree $T_i$ in the resulting forest has already been $Q$-labeled. $Q$-label all the unmarked nodes.

**end**

We need the following lemma for implementing step 5 of this algorithm.

**Lemma 4.2.** Let $T_1, T_2, \ldots, T_k$ be all the trees consisting of unmarked nodes after step 4 and let $r_i$ be the root node of tree $T_i$, $i = 1, \ldots, k$. Let nodes $x$ and $y$ be such that $x \in T_i$ and $y \in T_j$, and let $P_i$ and $P_j$ be the paths from $x$ and $y$ to $r_i$ and $r_j$, respectively, and let $B_{P_i}$ and $B_{P_j}$ be the $B$-labels of $P_i$ and $P_j$, respectively. Then $A_Q[x] = A_Q[y]$ iff $B_{P_i} = B_{P_j}$, and $A_Q[f(r_i)] = A_Q[f(r_j)]$.

**Lemma 4.3.** Given an instance of the coarsest partition problem as described above, Algorithm tree node labeling correctly finds the $Q$-labels of all the tree nodes in $O(\log n)$ time using $O(n)$ operations.

**Proof (sketch).** The correctness of the algorithm is clear. Steps 1–4 can be done within our complexity bounds since we are given the appropriate data structure. Step 5 can be done in $O(\log n)$ time using $O(n \log n)$ operations by using the pointer jumping technique on the directed trees and the technique used in Algorithm partition of Section 3.2. However, the tree nodes can be scheduled so that the number of nodes that participate in successive phases may decrease in a geometric series, and hence this step can be done in a linear number of operations as shown in [15].
5. The single function coarsest partition

The only detail left to complete the description of Algorithm coarsest partition stated in Section 2 concerns step 1 in which the cycle nodes have to be identified. Recall that the input consists of the two arrays $A_f[1..n]$ and $A_B[1..n]$ representing the function $f$ and the $B$-labels, respectively, and these arrays can be interpreted as a directed graph whose nodes have been assigned the $B$-labels. The following algorithm identifies all the nodes lying on a cycle.

Algorithm finding cycle nodes

**Input:** Two arrays $A_f[1..n]$ and $A_B[1..n]$.  
**Output:** All the cycle nodes are marked.

```
begin
  Step 1: For each edge $(x,f(x))$ create its buddy $(f(x), x)$.
  Step 2: Construct an adjacency list of the modified graph and find the Euler tours in the pseudo-forest by using the procedure in [19]. Now, a close observation of the resulting tours as determined by the successor function of [19] indicates that there are two Euler cycles for each pseudo-tree, and that each cycle edge $(x,f(x))$ and its buddy $(f(x), x)$ appear in different Euler cycles, while each tree edge $(y,f(y))$ and its buddy $(f(y), y)$ appear in the same Euler cycle.
  Step 3: Determine and mark the nodes on the cycles.
end
```

The only nontrivial detail is the development of the data structure necessary for applying the Euler tour technique. This can be easily done by using an integer sorting algorithm. Once this data structure is obtained, all the steps of the algorithm can be implemented using essentially the list ranking algorithm. Therefore, we have shown the following theorem.

**Theorem 5.1.** The single function coarsest partition problem can be solved in $O(\log n)$ time using $O(n \log \log n)$ operations on the arbitrary CRCW PRAM.

Acknowledgment

The authors would like to thank Uzi Vishkin for initially raising the authors' curiosity in the problem addressed in this paper.

References

The single function coarsest partition problem


