PARALLEL ALGORITHMS FOR P4-COMPARABILITY

Stavros D. Nikolopoulos and Leonidas Palios

20 – 2001

Preprint, no 20 – 01 / 2001

Department of Computer Science
University of Ioannina
45110 Ioannina, Greece

Parallel Algorithms for P_4-comparability Graphs

Stavros D. Nikolopoulos and Leonidas Palios
Department of Computer Science, University of Ioannina
GR-45110 Ioannina, Greece
e-mail: {stavros,palios}@cs.uoi.gr

Abstract: We consider two problems pertaining to P_4-comparability graphs, namely, the problem of recognizing whether a simple undirected graph is a P_4-comparability graph and the problem of producing an acyclic P_4-transitive orientation of a P_4-comparability graph. Sequential algorithms for these problems have been presented by Hoàng and Reed and very recently by Raschle and Simon, and by Nikolopoulos and Palios. In this paper, we establish properties of P_4-comparability graphs which allow us to invent parallel algorithms for the recognition and orientation problems on this class of graphs; for a graph on n vertices and m edges, our algorithms run in $O(\log^2 n)$ time and require $O((n + m^2)/\log n)$ processors on the CREW PRAM model. Thus, in view of the fact that the currently fastest sequential algorithms for these problems require $O(n + m^2)$ time, this behaviour is cost efficient. Our approach relies on the parallel computation and proper orientation of the P_4-components of the input graph.

Keywords: Parallel algorithms, perfectly orderable graphs, P_4-comparability graphs, P_4-components, recognition, acyclic P_4-transitive orientation, PRAM computation.

1. Introduction

Let $G = (V, E)$ be a simple non-trivial undirected graph. An *orientation* of the graph G is an antisymmetric directed graph obtained from G by assigning a direction to each edge of G. An orientation $U = (V, F)$ of G is called *transitive* if U satisfies the following condition: if abc is a chordless path on 3 vertices in G, then $\overrightarrow{ab}$ and $\overleftarrow{bc}$, or $\overleftarrow{ab}$ and $\overrightarrow{bc}$ in U, where by $\overrightarrow{uv}$ or $\overleftarrow{vu}$ we denote an edge directed from u to v. The relation F is called a *transitive orientation* of $E(G)$ or of G [14]. An orientation U of a graph G is called P_4*-transitive* if the orientation of every chordless path on 4 vertices of G is transitive; an orientation of such a path $abcd$ is transitive if and only if $\overrightarrow{ab}$, $\overleftarrow{bc}$ and $\overrightarrow{cd}$, or $\overleftarrow{ab}$, $\overrightarrow{bc}$ and $\overleftarrow{cd}$. The term borrows from the fact that a chordless path on 4 vertices is denoted by P_4.

A graph which admits an acyclic transitive orientation is called a *comparability graph* [12, 14]; Figure 1(a) depicts a comparability graph. A graph is a P_4*-comparability graph* if it admits an acyclic P_4-transitive orientation [16, 17]. In light of these definitions, every

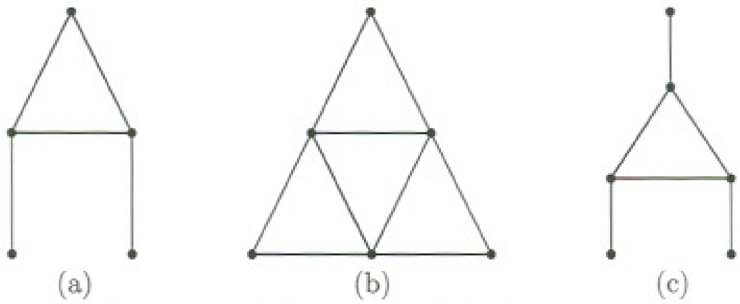

Figure 1: (a) a comparability graph, (b) a P_4-comparability graph, (c) a graph which is not P_4-comparability.

comparability graph is a P_4-comparability graph. Moreover, there exist P_4-comparability graphs which are not comparability; Figure 1(b) depicts such a graph, which is often referred to as a pyramid. The graph shown in Figure 1(c) is not a P_4-comparability graph.

In the early 1980s, Chvátal introduced the class of *perfectly orderable* graphs [5]. This is a very important class of graphs, since a number of problems, which are NP-complete in general, can be solved in polynomial time on its members [3, 5]; unfortunately, it is NP-complete to decide whether a graph admits a perfect order [24]. Chvátal showed that the class of perfectly orderable graphs contains the comparability and the chordal graphs [5]; thus, it also contains important subclasses of comparability and chordal graphs, such as the bipartite graphs, permutation graphs, interval graphs, split graphs, cographs, threshold graphs [14]. Later, Hoàng and Reed introduced the classes of the P_4-comparability, the P_4-indifference, the P_4-simplicial and the Raspail graphs, and proved that they are all perfectly orderable [17]. Moreover, the class of perfectly orderable graphs also includes a number of other classes of graphs which are characterized of important algorithmic and structural properties; we mention the classes of brittle, co-chordal, HHD-free, Meyniel $\cap$ co-Meyniel, P_4-sparse, ptolemaic [14]. We note that the class of perfectly orderable graphs is a subclass of the well-known class of perfect graphs.

Many researchers have devoted their work to the study of perfectly orderable graphs. They have proposed both sequential and parallel algorithms for many different problems on subclasses of perfectly orderable graphs; for example, problems for finding maximum cliques, maximum weighted cliques, maximum independent sets, optimal coloring, breadth-first search trees and depth-first search trees, hamiltonian paths and cycles, testing graphs for isomorphism [1, 6-10, 12, 15, 16, 19-22, 25-30, 32].

The comparability graphs in particular have been the focus of much research which culminated into efficient recognition and orientation algorithms [14, 22, 23, 25, 32]. Golumbic presented algorithms for recognizing and assigning transitive orientations on comparability graphs in $O(d\,m)$ time and $O(n+m)$ space, where d is the maximum degree of the graph's vertices [13, 14]. Due to the work of McConnell and Spinrad [22, 23], the graph modular decomposition and graph transitive orientation problems can be solved in $O(n+m)$ time. This gives linear time bounds for maximum clique and minimum vertex coloring on comparability graphs, and other combinatorial problems on comparability graphs and their complements. Recently, Morvan and Viennot [25], presented parallel algorithms for recognizing and assigning transitive orientation of comparability graphs; their algorithms run in $O(\log n)$ time and require $O(d\,m)$ processors on the CRCW PRAM model. They also presented a modular decomposition parallel algorithm which runs in $O(\log n)$ time with $O(n^3)$ processors on the same model of parallel computation.

On the other hand, the P_4-comparability graphs have not received as much attention, despite the fact that the definitions of the comparability and the P_4-comparability graphs rely on the same principles [11, 16, 17, 29, 30]. Hoàng and Reed addressed the problems of recognition and acyclic P_4-transitive orientation on the class of P_4-comparability graphs and they described polynomial time algorithms for their solution [16, 17]. Their recognition and orientation algorithms require $O(n^4)$ and $O(n^5)$ time respectively, where n is the number of vertices of G. Newer results on these problems were provided by Raschle and Simon [30]; their algorithms for either problem run in $O(n + m^2)$, where m is the number of edges of G. Recently, Nikolopoulos and Palios [29] presented different $O(n + m^2)$-time recognition and acyclic P_4-transitive orientation algorithms for P_4-comparability graphs of n vertices and m edges. We note that Hoàng and Reed [16, 17] also presented algorithms which solve the recognition problem for P_4-indifference graphs in $O(n^6)$ time. The recognition and orientation problems for P_4-indifference graphs were also studied by Raschle and Simon [30] who achieved $O(n + m^2)$ time complexities for both problems.

In this paper, we present parallel algorithms for the recognition and the acyclic P_4-transitive orientation problems on P_4-comparability graphs and analyze their time and processor complexity on the PRAM model of computation [2, 4, 18, 31]. Both algorithms run in $O(\log^2 n)$ time using a total of $O((n + m^2)/\log n)$ processors on the CREW PRAM model, where n and m are the number of vertices and edges of the input graph. They rely on structural properties of P_4-comparability graphs, and on efficient parallel algorithms for the computation and P_4-transitive orientation of the P_4-components of the input graph. To the best of our knowledge, the currently fastest algorithms for the recognition and acyclic P_4-transitive orientation problems of P_4-comparability graphs require $O(n + m^2)$ time in a sequential process environment [29, 30]. Thus, our algorithms are cost efficient.

The paper is structured as follows. In Section 2 we review the terminology that we will be using throughout the paper and we state some useful lemmata. We describe and analyze the recognition and acyclic P_4-transitive orientation algorithms in Section 3 and Section 4, respectively, while in Section 5 we conclude with a summary of our results, extensions and open problems.

2. Theoretical Framework

Let $G = (V, E)$ be a simple non-trivial graph on n vertices and m edges. A *path* in a graph $G = (V, E)$ is a sequence of vertices $(v_0, v_1, \ldots, v_k)$ such that $v_{i-1}v_i \in E$ for $i = 1, 2, \ldots, k$; we say that this is a path from v_0 to v_k and that its *length* is k. A path in a graph G is undirected or directed depending on whether G is an undirected or a directed graph; a *directed* path $(v_0, v_1, \ldots, v_k)$ is a path such that $\overrightarrow{v_0v_1}$, $\overrightarrow{v_1v_2}$, $\ldots$, $\overrightarrow{v_{k-1}v_k}$. A path is called *simple* if none of its vertices occurs more than once; it is called *trivial* if its length is equal to 0. A path (simple path) $(v_0, v_1, \ldots, v_k)$ is called a *cycle* (*simple cycle*) of length $k+1$ if $v_0v_k \in E$. A simple path (cycle) $(v_0, v_1, \ldots, v_k)$ is *chordless* if $v_iv_j \notin E$ for any two non-consecutive vertices v_i, v_j in the path (cycle). Throughout the paper, the chordless path (chordless cycle, respectively) on n vertices is denoted by P_n (C_n, respectively). In particular, a chordless path on 4 vertices is denoted by P_4.

Let *abcd* be a P_4 of a graph G. The vertices b and c are called *midpoints* and the vertices a and d *endpoints* of the P_4 *abcd*. The edge connecting the midpoints of a P_4 is called *rib*; the other two edges (which are incident upon the endpoints) are called *wings*. For example, the edge bc is the rib and the edges ab and cd are the wings of the P_4 *abcd*. Two P_4s are called *adjacent* if they have an edge in common. The transitive closure of the adjacency relation is an equivalence relation on the set of P_4s of a graph G; the subgraphs of G spanned by the edges of the P_4s in the equivalence classes are the *P_4-components* of G. Clearly, each P_4-component is connected and for any two P_4s ρ and ρ' which belong to the same P_4-component $\mathcal{C}$, there exists a sequence of adjacent P_4s in $\mathcal{C}$ from ρ to ρ'. With slight abuse of terminology, we consider that an edge which does not belong to any P_4 belongs to a P_4-component by itself; such a component is called *trivial*. A P_4-component which is not trivial is called *non-trivial*; clearly a non-trivial P_4-component contains at least one P_4. If the set of midpoints and the set of endpoints of the P_4s of a non-trivial P_4-component $\mathcal{C}$ partition the vertex set $V(\mathcal{C})$, then the P_4-component $\mathcal{C}$ is called *separable*.

The definition of a P_4-comparability graph requires that such a graph admit an acyclic P_4-transitive orientation. However, Hoàng and Reed [17] showed that in order to determine whether a graph is a P_4-comparability graph one can restrict one's attention to the P_4-components of the graph. In particular, what they proved ([17], Theorem 3.1) can be paraphrased in terms of the P_4-components as follows:

Lemma 2.1. **([17])** *Let G be a graph such that each of its P_4-components admits an acyclic P_4-transitive orientation. Then G is a P_4-comparability graph.*

Although determining that each of the P_4-components of a graph admits an acyclic P_4-transitive orientation suffices to establish that the graph is P_4-comparability, the directed graph produced by placing the oriented P_4-components together may contain cycles. However, an acyclic P_4-transitive orientation of the entire graph can be obtained by inversion of the orientation of some of the P_4- components. Therefore, if one wishes to compute an acyclic P_4-transitive orientation of a P_4-comparability graph, one needs to detect directed cycles (if they exist) formed by edges from more than one P_4-component and appropriately invert the orientation of one or more of these P_4- components. Fortunately, one does not need to consider arbitrarily long cycles as shown in the following lemma [17].

Lemma 2.2. **([17], Lemma 3.5)** *If a proper orientation of an interesting graph is cyclic, then it contains a directed triangle.*[1]

For a non-trivial P_4-component $\mathcal{C}$, the set of vertices $V - V(\mathcal{C})$ can be partitioned into three sets: the set R contains the vertices of $V - V(\mathcal{C})$ which are adjacent to some (but not all) of the vertices in $V(\mathcal{C})$, the set P contains the vertices of $V - V(\mathcal{C})$ which are adjacent to all the vertices in $V(\mathcal{C})$, and the set Q contains the vertices of $V - V(\mathcal{C})$ which are not adjacent to any of the vertices in $V(\mathcal{C})$. The adjacency relation is considered in terms of the input graph G.

In [30], Raschle and Simon showed that, for a non-trivial P_4- component $\mathcal{C}$ and a vertex $v \notin V(\mathcal{C})$, if v is adjacent to the midpoints of a P_4 of $\mathcal{C}$ and is not adjacent to its endpoints, then so is v with respect to every P_4 in $\mathcal{C}$ (that is, v is adjacent to the midpoints and not

[1] An orientation is *proper* if the orientation of every P_4 is transitive. A graph is *interesting* if the orientation of every P_4-component is acyclic.

adjacent to the endpoints of every P_4 in $\mathcal{C}$). This implies that any vertex of G, which does not belong to $\mathcal{C}$ and is adjacent to at least one but not all the vertices in $V(\mathcal{C})$, is adjacent to the midpoints of all the P_4s in $\mathcal{C}$. Based on that, Raschle and Simon showed that:

Lemma 2.3. **([30], Corollary 3.3)** *Let $\mathcal{C}$ be a non-trivial P_4-component and $R \neq \emptyset$. Then, $\mathcal{C}$ is separable and every vertex in R is V_1- universal and V_2-null*[2]*. Moreover, no edge between R and Q exists.*

The set V_1 is the set of the midpoints of all the P_4s in $\mathcal{C}$, whereas the set V_2 is the set of endpoints. Figure 2 shows the partition of the vertices of a graph with respect to a separable P_4-component $\mathcal{C}$; the dashed segments between P and R and P and Q indicate that there may be edges between pairs of vertices in the corresponding sets. Then, a P_4 with at least one but not all its vertices in $V(\mathcal{C})$ must be a P_4 of one of the following types:

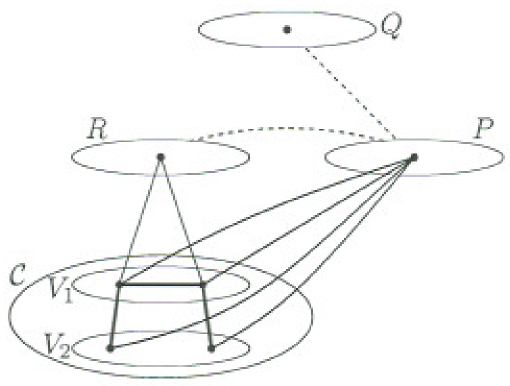

Figure 2

type (1)	vpq_1q_2	where $v \in V(\mathcal{C}),\ p \in P,\ q_1, q_2 \in Q$
type (2)	p_1vp_2q	where $p_1 \in P,\ v \in V(\mathcal{C}),\ p_2 \in P,\ q \in Q$
type (3)	$p_1v_2p_2r$	where $p_1 \in P,\ v_2 \in V_2,\ p_2 \in P,\ r \in R$
type (4)	$v_2pr_1r_2$	where $v_2 \in V_2,\ p \in P,\ r_1, r_2 \in R$
type (5)	rv_1pq	where $r \in R,\ v_1 \in V_1,\ p \in P,\ q \in Q$
type (6)	rv_1pv_2	where $r \in R,\ v_1 \in V_1,\ p \in P,\ v_2 \in V_2$
type (7)	$rv_1v_2v_2'$	where $r \in R,\ v_1 \in V_1,\ v_2, v_2' \in V_2$
type (8)	$v_1'rv_1v_2$	where $r \in R,\ v_1, v_1' \in V_1,\ v_2 \in V_2$

Raschle and Simon proved that neither a P_3 abc with $a \in V_1$ and $b, c \in V_2$ nor a $\overline{P_3}$ abc with $a, b \in V_1$ and $c \in V_2$ exists ([30], Lemma 3.4), which implies that:

Lemma 2.4. **([30])** *Let $\mathcal{C}$ be a non-trivial P_4-component of a graph $G = (V, E)$. Then, no P_4s of type (7) or (8) with respect to $\mathcal{C}$ exist.*

Let us consider a non-trivial P_4-component $\mathcal{C}$ of the graph G such that $V(\mathcal{C}) \subset V$, and let $S_\mathcal{C}$ be the set of non-trivial P_4-components of G which have at least as many vertices as $\mathcal{C}$ and have a vertex in common with $\mathcal{C}$. Then, each component in $S_\mathcal{C}$ contains at least one vertex in $V - V(\mathcal{C})$; otherwise, its vertex set would be equal to $V(\mathcal{C})$ (it cannot have fewer vertices than $\mathcal{C}$, according to the definition of $S_\mathcal{C}$), which would imply that the component would coincide with $\mathcal{C}$ (see [30]), a contradiction. Since each of these components also has a vertex in common with $\mathcal{C}$, it contains a P_4 of type (1)-(8). Then, taking Lemma 2.4 into account, we can partition the elements of $S_\mathcal{C}$ into two sets as follows:

[2] For a set A of vertices, we say that a vertex v is A-universal if v is adjacent to every element of A; a vertex v is A-null if v is adjacent to no element of A.

- *P_4-components of type A:* the P_4 components, each of which contains at least one P_4 of type (1)-(5) with respect to $\mathcal{C}$;
- *P_4-components of type B:* the P_4-components which contain only P_4s of type (6) with respect to $\mathcal{C}$.

Let $\mathcal{B}$ be a P_4-component which is of type B with respect to a P_4-component $\mathcal{C}$. Then, the general form of a P_4 of type (6) with respect to $\mathcal{C}$ implies that every edge of $\mathcal{B}$ has exactly one endpoint in $V(\mathcal{C})$, that if an edge of $\mathcal{B}$ is oriented towards its endpoint that belongs to $V(\mathcal{C})$, then so are all the edges of $\mathcal{B}$, and that the edges of $\mathcal{B}$ incident upon the same vertex v are all oriented either towards v or away from it. The following lemmata establish properties of P_4-components of type A and of type B (proofs can be found in [29]).

Lemma 2.5. *Let $\mathcal{C}$ be a non-trivial P_4-component of a P_4-comparability graph $G = (V, E)$ and suppose that the vertices in $V - V(\mathcal{C})$ have been partitioned into sets R, P, and Q as described earlier in this section. Then, if there exists an edge xv (where $x \in R \cup P$ and $v \in V(\mathcal{C})$) that belongs to a P_4-component $\mathcal{A}$ of type A, then all the edges, which connect the vertex x to a vertex in $V(\mathcal{C})$, belong to $\mathcal{A}$. Moreover, these edges are all oriented towards x or they are all oriented away from x.*

Lemma 2.6. *Let $\mathcal{B}$ and $\mathcal{C}$ be two non-trivial P_4-components of the graph G such that $|V(\mathcal{B})| \geq |V(\mathcal{C})|$ and let $\beta = \sum_{v \in V(\mathcal{C})} d_{\mathcal{B}}(v)$, where $d_{\mathcal{B}}(v)$ denotes the number of edges of $\mathcal{B}$ which are incident upon v. Then, $\mathcal{B}$ is of type B with respect to $\mathcal{C}$ if and only if $\beta = |E(\mathcal{B})|$.*

Lemma 2.7. *Let $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_h$ be the non-trivial P_4-components of a graph G ordered by increasing vertex number and suppose that each component has received an acyclic P_4-transitive orientation. Consider the set $S_i = \{\mathcal{C}_j \mid j < i \text{ and } \mathcal{C}_i \text{ is of type B with respect to } \mathcal{C}_j\}$. If the edges of each P_4-component $\mathcal{C}_i$ such that $S_i \neq \emptyset$ get oriented towards their endpoint which belongs to $V(\mathcal{C}_{\hat{\imath}})$, where $\hat{\imath} = \min\{j \mid \mathcal{C}_j \in S_i\}$, then the resulting directed subgraph of G spanned by the edges of the $\mathcal{C}_i$s $(1 \leq i \leq h)$ does not contain a directed cycle.*

Notation. Let G be a simple graph with vertex set V and edge set E. Hereafter, the subgraph of G induced by a vertex subset $S \subseteq V$ is denoted by $G[S]$ and the subgraph spanned by an edge subset $W \subseteq E$ is denoted by $G\langle S \rangle$.

Moreover, with slight abuse of notation, in the following we use vertices or edges to index arrays.

3. P_4-comparability Graph Recognition

We will assume for the time being that the input graph is connected; the case of a disconnected input graph is addressed in Section 3.5. So, let $G = (V, E)$ be a connected simple graph on n vertices and m edges. Then, $n = O(m)$ and $\log n = \Theta(\log m)$. Let E_C and E_T be the sets of the edges of all the non-trivial and trivial P_4-components of G respectively; because the edges in E_T span trivial P_4-components, we will refer to these edges as *trivial edges*. Since an edge belongs to exactly one P_4-component, it follows that $E = E_C \cup E_T$.

Before presenting the algorithm, we will describe the preprocessing. In order to save on the number of processors, we need to be able to determine in constant time the rank of a vertex in the adjacency list of one of its neighbors. To be able to do that, we construct a $(2m)$-array of edges where we place each edge twice, once for each of the two orderings of the two vertices to which it is incident; so an edge incident upon the vertices x and y, will contribute two entries, one for xy and another for yx. Then, we sort the elements of the array based on the index of the first of the two vertices that correspond to the entry; note that the entry which corresponds to xy will be sorted based on the vertex x, whereas the one corresponding to yx will be sorted based on y. After the sorting, all edges incident upon the same vertex occupy consecutive places in the array. Thanks to this sorted array, the rank of a vertex in the adjacency list of one of its neighbors can be computed in constant time. For example, the $\text{rank}(x, y)$ of the vertex y in the adjacency list of x can be computed by adding 1 to the difference of the position of the edge xy in the array minus the minimum position of any edge incident upon x.

The above array can be initialized in $O(1)$ time using $O(m)$ processors on the EREW PRAM model, or in $O(\log^2 n)$ time using $O(m/\log^2 n)$ processors. The sorting can be done in $O(\log m) = O(\log n)$ time using $O(m)$ processors on the CREW PRAM model, or in $O(\log^2 n)$ time using $O(m/\log n)$ processors on the same model [2, 18, 31].

Our P_4-comparability graph recognition algorithm involves the following algorithmic steps.

Algorithm for the Recognition of a P_4-comparability Graph G (P4G_REC)

1. Construct an auxiliary graph $\widehat{G}$ which has m vertices $u_1, u_2, \ldots, u_m$; the vertex u_i corresponds to the edge e_i of G. Two vertices u_i and u_j are adjacent in $\widehat{G}$ iff the corresponding edges e_i and e_j form a P_3 which is contained in a P_4 of G.

2. Compute the connected components of the graph $\widehat{G}$; the edges corresponding to the vertices of each connected component span a P_4-component of G. Then, find a P_4-transitive orientation for each P_4-component. If a P_4-component cannot admit a P_4-transitive orientation, then G is not a P_4-comparability graph; exit.

3. Compute appropriate inversions (if needed) of the orientation of the non-trivial P_4-components of G so that if G is a P_4-comparability graph then the directed graph $G\langle E_C \rangle$, spanned by the edges of its non-trivial P_4-components, is acyclic.

4. For each trivial edge xy (i.e., $xy \in E_T$), check if there exists a directed path from x to y, or from y to x; in the former case orient the edge towards y, in the latter towards x. If during this process, a directed triangle (C_3) or a directed C_4 is formed, then G is not a P_4-comparability graph; exit.

We note that in order to determine if G is a P_4-comparability graph, it would suffice to check whether the P_4-transitive orientation of each P_4-component (after Step 2) is acyclic (Lemma 2.1). Finding a cycle can be done either by computing the transitive closure of each P_4-component or by a method similar to Step 4 above (if this approach is used before Step 3 — that is, the P_4-components have not received compatible orientations —, then a trivial edge may be assigned opposite orientations by different P_4-components; because this does not necessarily imply that the input graph is not a P_4-comparability, we need to keep different copies of the trivial edges, which results into high cost.) Both approaches exhibit high time and processor complexities.

Steps 1 and 2 correctly compute and orient the P_4-components of the input graph G; note that if an edge is assigned incompatible orientations (for example, if the graph contains a C_5) then G is not a P_4-comparability graph and the algorithm terminates. Step 3 computes appropriate orientation inversions of the non-trivial P_4-components based on Lemma 2.7 which guarantees that if G is a P_4-comparability graph then the resulting directed graph spanned by the edges of the non-trivial P_4-components has an acyclic P_4-transitive orientation. So, if G is a P_4-comparability graph, then no directed C_3 or C_4 exists (otherwise a directed cycle would exist), and our algorithm correctly identifies the input graph as a P_4-comparability graph. If G is not a P_4-comparability graph, then either there will be incompatibilities in the assignment of orientations to the edges (which is detected in Step 2), or there is a directed cycle in a non-trivial P_4-component; thus, there exists a directed cycle in $G\langle E_C\rangle$ (Step 3 terminates even if G is not a P_4-comparability graph). If there exists a directed C_3 or C_4 in $G\langle E_C\rangle$, it will be immediately detected and the input graph will be correctly characterized. If there exists a longer cycle, then every triple of consecutive edges of the cycle cannot span a P_4 and hence there exist chords which will receive compatible orientation in Step 4 and will result in the formation of a directed C_3 or C_4; again, the input graph is correctly characterized.

A parallel implementation of each step of the proposed algorithm is presented in the following paragraphs.

3.1. Construction of the Graph $\widehat{G}$. In the construction of the graph $\widehat{G}$, we use two auxiliary arrays for each vertex of the input graph G and an $(m \times m)$-array M. Namely, for vertex v of G, we use an n-array D_v and an array L_v of size $2m \times \text{degree}(v)$. The array D_v contains information about the vertices of G at the 1st and 2nd level of the BFS tree T_v of G rooted at v. In particular, $D_v[x] = 1$, $D_v[y] = 2$, and $D_v[z] = 3$ iff x is a vertex of the 1st level of T_v, y is a vertex of the 2nd level which is incident upon no vertices in the 3rd level, and z is a vertex of the 2nd level which is incident upon a vertex in the 3rd level; for the remaining vertices, the corresponding entry of D_v is equal to 0. The array L_v helps us avoid simultaneous memory accesses for write operations and so does the array M.

The construction algorithm works as follows:

Algorithm for the Construction of the Graph $\widehat{G}$ (G_HAT)

1. Construct a graph $\widehat{G}$ with m vertices $u_1, u_2, \ldots, u_m$ and no edges (the vertex u_i corresponds to the edge e_i of G);

2. Compute the arrays D_v for each vertex v of G; initialize to 0 all the entries of the arrays L_v for each vertex v of G and of the array M;

3. For each vertex v of the graph G, do in parallel
   3.1 for each edge xy of the graph G, do in parallel
      (a) if $D_v[x] = 1$ and $D_v[y] = 3$, then
            $M[e_i, e_j] := 1$, where $e_i = vx$ and $e_j = xy$;
      (b) if $D_v[x] = 3$ and $D_v[y] = 1$, then
            $M[e_i, e_j] := 1$, where $e_i = vy$ and $e_j = xy$;
      (c) if $D_v[x] = 2$ and $D_v[y] = 2$, then
            for each vertex w adjacent to v, do in parallel
               if $D_w[x] = 1$ and $D_w[y] \neq 1$, then $L_v[xy, \text{rank}(v, w)] := 1$;

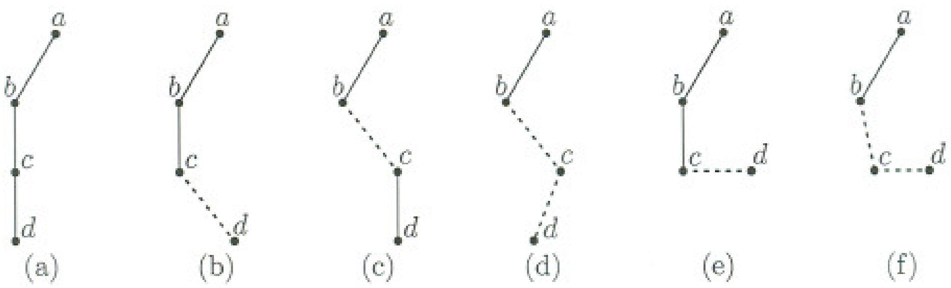

Figure 3: The different positions of a P_4 *abcd* in the BFS tree T_a.

if $D_w[x] \neq 1$ and $D_w[y] = 1$, then $L_v[yx, \text{rank}(v, w)] := 1$;

3.2 for each vertex w adjacent to v, do in parallel
for each vertex x, do in parallel
check if there exists an entry in the subarray $L_v[x*, \text{rank}(v, w)]$ equal to 1; if yes, $M[e_i, e_j] := 1$, where e_i and e_j are the edges of G connecting v and w, and w and x respectively;

4. For $i = 1 \ldots m$ do in parallel
for $j = 1 \ldots m$ do in parallel
if $M[e_i, e_j] = 1$ or $M[e_j, e_i] = 1$, then add the edge $u_i u_j$ in $\widehat{G}$;

We observe that the graph $\widehat{G}$ has m vertices and $O(nm)$ edges. The correctness of the construction algorithm of $\widehat{G}$ follows from the fact that both P_3s of each P_4 of the graph G are taken into account. To see this, consider a P_4 *abcd* of the graph G. We will show that the vertices of $\widehat{G}$ corresponding to the edges ab and bc will be adjacent in $\widehat{G}$; the case for the edges bc and cd is similar. Since the algorithm processes all the vertices of G in Step 3, it will process the vertex a too. Let us investigate the different positions that this path may assume in the BFS tree T_a. Clearly, the vertices a, b, and c have to belong to the 0th, 1st, and 2nd level respectively; the vertex d may belong to the 2nd or 3rd level, but not to the 1st level since d is not adjacent to a. All the possible positions of the path are shown in Figure 3; the solid lines, the slanting dashed lines, and the horizontal lines represent tree edges, cross edges, and level edges respectively. In the first four cases of Figure 3, the P_3 *abc* will be recorded in $\widehat{G}$ by means of the Substeps 3.1(a)-(b); the final two cases of Figure 3 are covered by the Substeps 3.1(c) and 3.2.

Time and Processor Complexity. We shall use a step-by-step analysis.

Step 1: A graph with m vertices and no edges can be constructed in $O(1)$ time using $O(m)$ processors on the EREW PRAM model.

Step 2: The initialization of all the arrays L_v ($\forall v \in V$) can be carried out in $O(\log^2 n)$ time using $O((n+m^2)/\log^2 n) = O(m^2/\log^2 n)$ processors since the total number of entries of these arrays is $4m^2$. Similarly, the array M can be initialized in $O(\log^2 n)$ time using $O(m^2/\log^2 n)$ processors. Next, we present a CREW PRAM computation of the n-array D_v, where $v \in V(G)$. We will use the array L_v (that we saw earlier) and another auxiliary array N_v of size $2m$. In the array N_v, we mark the edges connecting a vertex of the 2nd level of T_v to a vertex of the 3rd level of T_v. We work as follows:

Computation of all the vertices of the 1st and 2nd levels of T_v:

(i) we initialize to 0 all the entries of the arrays D_v and L_v for all $v \in V$;

(ii) for each vertex x, we do:
 (a) if $\text{rank}(v, x) > 0$ (i.e., v and x are adjacent), then $D_v[x] := 1$;
 (b) else for each vertex y adjacent to x, we do
  if $\text{rank}(v, y) > 0$ (i.e., $vx \notin E$, $vy \in E$), then $L_v[xy, \text{rank}(v, y)] := 1$;
 (c) we check whether there exists an entry in the subarray $L_v[x*, *]$ with value equal to 1; if there exists, then we set $D_v[x] := 2$; $\{$*not necessarily the final value*$\}$

Computation of all the 2nd level vertices, which are adjacent to vertices of the 3rd level:

(iii) we initialize all the entries of N_v to 0;

(iv) for each edge xy, we do:
 if $D_v[x] = 2$ and $D_v[y] = 0$, (i.e., x in 2nd, y in 3rd level), then $N_v[xy] := 1$;
 if $D_v[x] = 0$ and $D_v[y] = 2$, (i.e., x in 3rd and y in 2nd level), then $N_v[yx] := 1$;

(v) for each vertex x, we check whether there exists an entry in the subarray $N_v[x*]$ with value equal to 1; if there exists, (i.e., x is adjacent to a vertex in the 3rd level of T_v), then we set $D_v[x] := 3$;

By the end of steps (i)-(v), the arrays D_v are correctly updated. In steps (ii)(c) and (v), the test whether there is an entry equal to 1 in the subarrays $L_v[x*, *]$ and $N_v[x*]$, which are of sizes $\text{degree}(x) \times \text{degree}(v)$ and $\text{degree}(x)$ respectively, is done by means of an interval prefix computation on the entire arrays L_v and N_v which are of sizes $2m \times \text{degree}(v)$ and $2m$ respectively; the interval prefix computation on an array of N elements can be carried out on the EREW PRAM in $O(\log N)$ time with $O(N/\log N)$ processors [2, 18], or in $O(\log^2 N)$ time with $O(N/\log^2 N)$ processors. Thus, the arrays D_v can be computed for all the vertices v of G in $O(\log^2 n)$ time using a total of $O(m^2/\log^2 n)$ processors on the CREW PRAM model; note that $\log n \leq \log(m \times \text{degree}(v)) \leq \log n^3$.

Step 3: It is not difficult to see that Substep 3.1 can be executed in $O(\log^2 nm)$ time using $O(nm/\log^2 nm)$ processors on the CREW PRAM model. The maximum in Substep 3.2, is computed by using interval prefix computation. Thus, the processing takes $O(\log^2 n)$ parallel time and requires $O(m^2/\log^2 n)$ processors on the CREW PRAM model.

Step 4: Step 4 can be executed in $O(1)$ time using $O(m^2)$ processors on the EREW PRAM model, or in $O(\log^2 n)$ time using $O(m^2/\log^2 n)$ processors on the same model.

Thus, we have proved the following result.

Theorem 3.1. *Let G be a connected simple graph on n vertices and m edges. Algorithm G_HAT constructs the graph $\widehat{G}$ of G in $O(\log^2 n)$ time using a total of $O(m^2/\log^2 n)$ processors on the CREW PRAM model.*

3.2. P_4-transitive Orientation of each P_4-component. The algorithm relies in the computation and processing of the connected components of the graph $\widehat{G}$; note that the edges of G corresponding to the vertices of such a component span a P_4-component of G.

Algorithm for the P_4-transitive Orientation of each P_4-component (P4C_TRO)

1. Compute the connected components $\widehat{C}_1, \widehat{C}_2, \ldots, \widehat{C}_\ell$ of the graph $\widehat{G}$;

2. For every connected component $\widehat{C}_i$, $1 \leq i \leq \ell$, do in parallel
   compute a spanning tree T_i of $\widehat{C}_i$ and the set B_i of its non-tree edges;

3. For every tree T_i, $1 \leq i \leq \ell$, do in parallel
   Construct the tree R_i using the tree T_i as follows:
   3.1 for every vertex u of T_i which corresponds to the edge xy of G, do in parallel
       add two vertices a_u and b_u in $V(R_i)$;
       add the edge $a_u b_u$ in $E(R_i)$;
       set $lbl(a_u) := x$ and $lbl(b_u) := y$;
   3.2 for every edge uv of T_i where u and v correspond to the edges xy and yz of G, do in parallel
       if $lbl(a_u) = y$, then $a := a_u$ else $a := b_u$;
       if $lbl(a_v) = y$, then $b := a_v$ else $b := b_v$;
       add the edge ab in $E(R_i)$; {*note that*, $lbl(a) = lbl(b) = y$}

4. For every tree R_i, $1 \leq i \leq \ell$, do in parallel
   4.1 root the tree R_i at an arbitrary vertex r_i;
   4.2 for every vertex v of R_i, do in parallel
       while $p(v)$ and $p(p(v))$ are defined and $lbl(p(v)) = lbl(p(p(v)))$ do
           set $p(p(v))$ to be the parent of v; {$p(v)$ *denotes* v*'s parent in* R_i}
   4.3 for every vertex v of R_i, do in parallel
       if v is a leaf and $lbl(v) = lbl(p(v))$, then delete v from R_i; {*note:* $v \neq r_i$}

5. For every tree R_i, $1 \leq i \leq \ell$, do in parallel
   5.1 compute the level of each vertex of the tree R_i;
   5.2 for each edge ab of R_i, do in parallel
       if $level(a)$ is even, then orient edge ab away from a;
       otherwise, orient edge ab towards a;

6. For every component $\widehat{C}_i$, $1 \leq i \leq \ell$, do in parallel
   6.1 if for every pair of edges ab and cd in R_i such that $lbl(a) = lbl(d)$, and $lbl(b) = lbl(c)$, it holds that $\overrightarrow{ab}$, $\overleftarrow{cd}$ (or $\overleftarrow{ab}$, $\overrightarrow{cd}$), then R_i is a "good-R$_i$" tree;
   6.2 for every edge uv of B_i where u and v correspond to the edges xy and yz of G, do in parallel
       if for every pair of edges ab and cd in R_i such that $lbl(a) = x$, $lbl(b) = lbl(c) = y$ and $lbl(d) = z$, it holds that $\overrightarrow{ab}$, $\overleftarrow{cd}$ (or $\overleftarrow{ab}$, $\overrightarrow{cd}$), then R_i is a "good-B$_i$" tree;

7. If there exists a tree R_i, $1 \leq i \leq \ell$, which is not a "good-R$_i$" or a "good-B$_i$" tree, then G is not a P_4-comparability graph;

Time and Processor Complexity. We now compute the time and processor complexity of the proposed parallel algorithms on the CREW PRAM model of computation. We shall use a step-by-step analysis.

Step 1: The graph $\widehat{G}$ has m vertices. Thus, the connected components $\widehat{C}_1, \widehat{C}_2, \ldots, \widehat{C}_\ell$ of the graph $\widehat{G}$ can be computed in $O(\log^2 m)$ time using a total of $O(m^2/\log^2 m)$ processors on the EREW PRAM model [26].

Hereafter, m_i denotes the number of vertices of the connected component $\widehat{C}_i$, $1 \leq i \leq \ell$.

Step 2: A spanning forest of a graph on N vertices is computed in $O(\log^2 N)$ time using a total of $O(N^2/\log^2 N)$ processors on the EREW PRAM model [26]. (Note that an algorithm that finds a spanning forest of a graph also finds the connected components of this graph; the converse, however, is not necessarily true.) The number of vertices of the component $\widehat{C}_i$ of the graph $\widehat{G}$ is m_i; thus, a spanning tree T_i of the component $\widehat{C}_i$ is computed in $O(\log^2 m_i)$ time using $O(m_i^2/\log^2 m_i)$ processors, $1 \leq i \leq \ell$. Since $m_1 + m_2 + \ldots + m_\ell \leq m$, we obtain that the whole substep can be executed in $O(\log^2 m)$ time using a total of $O(m^2/\log^2 m)$ processors. (We use a dummy vertex w and make it to be adjacent with a vertex of each connected component $\widehat{C}_1, \widehat{C}_2, \ldots, \widehat{C}_\ell$; vertex w connects $\widehat{C}_1, \widehat{C}_2, \ldots, \widehat{C}_\ell$ into a connected component C; then, we compute a spanning tree of C, we delete the vertex w, and we have the connected components of the resulting graph.)

Step 3: The connected component $\widehat{C}_i$ of the graph $\widehat{G}$ has m_i vertices, and, thus, T_i is a tree on m_i vertices and $m_i - 1$ edges, $1 \leq i \leq \ell$. Clearly, since $m_1 + m_2 + \ldots + m_\ell \leq m$, both Substeps 3.1 and 3.2 are executed in $O(\log^2 m)$ time using a total of $O(m^2/\log^2 m)$ processors on the EREW PRAM model. (We use the array packing technique to compute the sets $V(R_i)$ and $E(R_i)$ for each tree R_i, $1 \leq i \leq \ell$.)

Step 4: Rooting a tree with N vertices can be optimally done on the EREW PRAM using the Euler-tour technique; that is, this computation needs $O(\log N)$ time and $O(N/\log N)$ processors [18]. Thus, Substep 4.1 is executed in $O(\log^2 m)$ time using a total of $O(m^2/\log^2 m)$ processors on the EREW PRAM model. (Let $r_1, r_2, \ldots, r_\ell$ be arbitrary vertices of the trees $R_1, R_2, \ldots, R_\ell$, respectively; we use a dummy vertex w and make it to be adjacent to the vertices $r_1, r_2, \ldots, r_\ell$; vertex w connects $R_1, R_2, \ldots, R_\ell$ into a tree R; then, we root the tree R at vertex w; we compute the subtrees $R_1, R_2, \ldots, R_\ell$ of the tree R rooted at $r_1, r_2, \ldots, r_\ell$, respectively.) Substep 4.2 implements the pointer jumping technique on R_i, $1 \leq i \leq \ell$; this technique on a tree of N vertices needs $O(\log N)$ time and $O(N)$ processors on the CREW PRAM model [2, 18]. The tree R_i has m_i vertices; thus, Substep 4.2 is executed in $O(\log m)$ time with $O(m)$ processors on the CREW PRAM model. Substep 4.3 is clearly executed in $O(1)$ time with $O(nm)$ processors, or in $O(\log^2 n)$ time using a total of $O(nm/\log^2 n)$ processors on the EREW PRAM model.

Step 5: Since computing the level function of a tree on N vertices can be done on the EREW PRAM in $O(\log N)$ time with $O(N/\log N)$ processors using the Euler-tour technique [18], Substep 5.1 requires $O(\log^2 m)$ time and $O(m^2/\log^2 m)$ processors on the EREW PRAM model. (Let $r_1, r_2, \ldots, r_\ell$ be arbitrary vertices of the trees $R_1, R_2, \ldots, R_\ell$, respectively; we use a dummy vertex w and make it to be adjacent to the roots $r_1, r_2, \ldots, r_\ell$; vertex w connects $R_1, R_2, \ldots, R_\ell$ into a tree R; then, we compute the level of each vertex of the tree R rooted at vertex w, which is 1 plus the level of the vertex in the tree R_i to which it belongs.) Regarding Substep 5.2, we note that, after executing Substep 4.3, the number of vertices in each tree R_i equals the number of edges in its corresponding connected component $\widehat{C}_i$, $1 \leq i \leq \ell$; that is, each tree R_i contains at most $2m_i$ vertices. Thus, this substep can be executed in $O(\log^2 m)$ time using a total of $O(m/\log^2 m)$ processors on the CREW PRAM model.

Step 6: Consider the following implementation of this step on the EREW PRAM model:
Substep 6.1:

(i) construct the vertex sets W_1 and W_2 such that:
W_1 contains all the vertices a of $R_1, R_2, \ldots, R_\ell$ such that $level(a)$ = even;
W_2 contains all the vertices b of the same trees such that $level(b)$ = odd;

(ii) construct an auxiliary $(2m \times m)$-array L such that:
$L[ab, xy] = 1$, iff ab is an edge in some R_i, $a \in W_1$, $lbl(a) = x$ and $lbl(b) = y$;
$L[ab, xy] = -1$, iff ab is an edge in some R_i, $a \in W_1$, $lbl(a) = y$, $lbl(b) = x$;
initially, all the entries of the array L are 0; note that, $a \in W_1$ implies $\overrightarrow{ab}$;

(iii) compute the maximum and minimum among the elements of each column xy of the array L; let them be $\max(xy)$ and $\min(xy)$ respectively;

(iv) for every column xy of the array L, do
if $\max(xy) \leq 0$ or $\min(xy) \geq 0$ then
set good-R := true; {*i.e., all the R_is are "good-R_i" trees*}

If at the end, good-R = true, then the edges of the input graph G which correspond to the edges of all the R_is are compatibly oriented.

Let us now compute the time and processor complexity of this procedure. Substep (i): We have seen that the trees $R_1, R_2, \ldots, R_\ell$ contain $O(m)$ vertices in total. Thus, the vertex sets W_1 and W_2 can be constructed in $O(1)$ time with $O(m)$ processors, or in $O(\log^2 n)$ time with $O(m/\log^2 n)$ processors on the EREW PRAM model. Substep (ii): It is easy to see that the array L can be computed in $O(1)$ time with $O(m^2)$ processors, or in $O(\log^2 n)$ time with $O(m^2/\log^2 n)$ processors on the EREW PRAM model. (Note that since the tree R_i has at most $2m_i$ vertices and thus $O(m_i)$ edges, the array L suffices to accommodate all pairs of an edge of a tree R_i and an edge of G.) Substep (iii): The maximum and minimum among m elements are computed in $O(\log^2 m)$ time with $O(m/\log^2 m)$ processors on the EREW PRAM model. Substep (iv): Obviously, this substep is executed in $O(\log^2 m)$ time with $O(m/\log^2 m)$ processors on the EREW PRAM model. Thus, the entire Substep 6.1 is executed in $O(\log^2 n)$ time using a total of $O(m^2/\log^2 n)$ processors on the EREW PRAM model.

Substep 6.2: After the (successful) completion of Substep 6.1, all the edges of the P_4-components of G have been assigned an orientation. So, in order to complete this substep, we need to check for every edge of every set B_i, which corresponds to a P_3, say, xyz, whether the edges xy and yz have compatible orientations. If it is so for all these edges, then a variable good-B is set to true. (This means that all the trees R_i are good-B_i trees.)

Step 7: Thanks to the variables good-R and good-B of the previous step, Step 7 can be executed in constant sequential time by simply checking whether they are both true.

Taking into consideration the time and processor complexity of each step of the algorithm P4C_TRO, we conclude that:

Theorem 3.2. *Algorithm P4C_TRO runs in $O(\log^2 n)$ time using a total of $O(m^2/\log^2 n)$ processors on the CREW PRAM model.*

Corollary 3.1. *The non-trivial P_4-components of a connected simple graph G on n vertices and m edges can be computed and P_4-transitive oriented in $O(\log^2 n)$ time using a total of $O(m^2/\log^2 n)$ processors on the CREW PRAM model.*

3.3. Combining the P_4-transitive orientations of the P_4-components. The algorithm relies on Lemma 2.7 and it is using three auxiliary arrays: an $(n \times k)$-array A, an $(m \times k)$-array B, and a $(k \times k)$-array H, where k is the number of non-trivial P_4-components of the input graph G. The array A records to which P_4-components a vertex belongs; if vertex v is a vertex of the i-th P_4-component, then the entry $A[v, i]$ is equal to 1, otherwise it is 0. The array B records the P_4-components to which an edge is adjacent, that is, the components which contain one of its endpoints (note that the general form of the P_4s of type (1)-(6) with respect to a P_4-component $\mathcal{C}$ implies that at most one of the endpoints of an edge of such a P_4 belongs to $\mathcal{C}$); the corresponding entries are equal to 1 while the remaining ones are equal to 0. The array H stores for a P_4-component $\mathcal{C}$ the P_4-components $\mathcal{C}'$ is of type B such that $\mathcal{C}$ is of type B with respect to $\mathcal{C}'$; then, the entry in the row corresponding to the P_4-component $\mathcal{C}$ and the column corresponding to the P_4-component $\mathcal{C}'$ is 1.

Algorithm P_4-Transitive Orientation of all the P_4-components (P4C_ALL_TRO)

1. Sort the non-trivial P_4-components of the graph G in increasing order of their vertex number; let them be $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k$ in that order;

2. Initialize all the entries of the arrays A, B, and H to 0; Set $H[i, i] := 1$;
   For each P_4-component $\mathcal{C}_i$ $(1 \leq i \leq k)$ do in parallel
       form a $(2m)$-array containing the vertices of the m_i edges of $\mathcal{C}_i$ $(1 \leq i \leq k)$;
       sort this array, use array packing on the sorted array, and
       update the corresponding entries of the array A;

3. For every P_4-component $\mathcal{C}_i$, $2 \leq i \leq k$, do in parallel
   3.1 for every edge xy of $\mathcal{C}_i$
       for every P_4-component $\mathcal{C}_j$, $1 \leq j < i$, do in parallel
           if $A[x, j] = 1$ or $A[y, j] = 1$ (i.e., x or y belongs to $\mathcal{C}_j$)
           then $B[xy, j] := 1$;
   3.2 for every P_4-component $\mathcal{C}_j$, $1 \leq j < i$, do in parallel
       if all the entries of B for the rows which correspond to all the edges of $\mathcal{C}_i$
           and the column which corresponds to $\mathcal{C}_j$ are equal to 1,
       then $H[i, j] := 1$;   {*$\mathcal{C}_i$ is of type B w.r.t. $\mathcal{C}_j$*}
   3.3 find the minimum among the indices of the non-zero entries in $H[i, *]$; let it be $\hat{\imath}$;
   3.4 if $i \neq \hat{\imath}$ then
       for each edge xy of $\mathcal{C}_i$, do in parallel
           if $A[x, \hat{\imath}] = 1$ (i.e., x is a vertex of $\mathcal{C}_{\hat{\imath}}$)
           then orient the edge xy towards x;
           otherwise orient xy away from x;

Time and Processor Complexity. We now compute the time and processor complexity of the proposed parallel algorithm on the CREW PRAM model of computation.

Step 1: It is well known that N elements can be sorted in $O(\log N)$ time with $O(N)$ processors on the CREW PRAM model [2, 18]. Thus, this step is executed in $O(\log m)$ time with $O(m)$ processors on the same model of computation; note that $1 \leq k \leq m$.

Step 2: The initialization of the arrays A, B, and H can be done in $O(1)$ time using $O(kn + km + kk) = O(m^2)$ processors on the EREW PRAM model; equivalently, it can be done in $O(\log^2 n)$ time using $O(m^2/\log^2 n)$ processors on the same model.
Sorting the array of size $2m$ containing vertices of $\mathcal{C}_i$ takes $O(\log m)$ time using $O(m)$ processors on the CREW PRAM model. Array packing on the sorted array takes $O(\log m)$ time using $O(m/\log m)$ processors on the EREW PRAM model. The entries of the array A are updated in constant time using as many processors as the size of the packed sorted array corresponding to $\mathcal{C}_i$; hence, $O(m)$ processors suffice. Thus, the P_4 component $\mathcal{C}_i$ can be processed in $O(\log m)$ time using $O(m)$ processors on the CREW PRAM model. Since $\log m = \Theta(\log n)$ because the graph G is connected, the processing of $\mathcal{C}_i$ can be completed in $O(\log n)$ time using $O(m)$ processors, or in $O(\log^2 n)$ time using $O(m/\log n)$ processors on the CREW PRAM model. Summing over all P_4-components, Step 2 can be executed in $O(\log^2 n)$ time using $O(m^2/\log n)$ processors on the CREW PRAM model.

Step 3: Substep 3.1 can be completed in $O(1)$ time using $O(i\, m_i) = O(m\, m_i)$ processors, or in $O(\log^2 n)$ time using $O((m\, m_i/\log^2 n)$ processors on the EREW PRAM model. Substep 3.2 can be accomplished by computing the minimum of the subarrays $B[xy, j]$ ($1 \leq j < i$), where $xy \in E(\mathcal{C}_i)$; if the minimum of such a subarray is equal to 1, then all the entries of the subarray are equal to 1, otherwise there exists at least one which is not. Since the minimum value of the entries of an array of size N can be computed in $O(\log N)$ time with $O(N/\log N)$ processors on the EREW PRAM model, then this computation for $\mathcal{C}_i$ on all the above subarrays can be executed in $O(\log m_i)$ time using $O(i\, m_i/\log m_i) = O(m\, m_i/\log m_i)$ processors on the EREW PRAM model. This implies that it can be executed in $O(\log n)$ time using $O(m\, m_i/\log n)$ processors, or in $O(\log^2 n)$ time using $O(m\, m_i/\log^2 n)$ processors on the EREW PRAM model. In a similar fashion, in Substep 3.3, computing the minimum index of a non-zero entry in $H[i, *]$ takes $O(\log k) = O(\log m)$ time using $O(k/\log k) = O(m)$ processors on the EREW PRAM model, which implies that it can also be done in $O(\log^2 n)$ time using $O(m/\log n)$ processors on the same model. Finally, Substep 3.4 can be carried out in $O(1)$ time using $O(m_i)$ processors, or $O(\log^2 n)$ time using $O(m_i/\log^2 n)$ processors on the EREW PRAM model. Summarizing, since Step 3 involves the execution of the above tasks for (nearly) all the non-trivial P_4-components (whose number k is $O(m)$), it will take $O(\log^2 n)$ time using $O(m^2/\log^2 n)$ processors on the CREW PRAM model.

Thus, we have the following result.

Theorem 3.3. *Given an acyclic P_4-transitive orientation of the P_4-components of a connected simple graph G, Algorithm P4C_ALL_TRO produces an acyclic P_4-transitive orientation of the graph $G\langle E_C \rangle$ in $O(\log^2 n)$ time using a total of $O(m^2/\log^2 n)$ processors on the CREW PRAM model.*

3.4. Detecting directed cycles in P_4-components. We have shown that if the input graph G is a P_4-comparability graph, then Algorithm P4C_ALL_TRO produces an acyclic P_4-transitive orientation $G\langle \overrightarrow{E_C} \rangle$ of the graph $G\langle E_C \rangle$ spanned by the edges of the non-trivial P_4-components of G. If G is not a P_4-comparability graph then a non-trivial P_4-component of G either cannot admit a P_4-transitive orientation or contains a directed cycle. Whether each non-trivial P_4-component admits a P_4-transitive orientation is determined during the execution of Algorithm P4C_TRO; if not, the algorithm stops reporting that the input graph is not a P_4-comparability graph. Therefore, the recognition will be complete after we check whether there exists a directed cycle in the P_4-transitive orientation of the non-trivial P_4-components. In order to do this, we use an algorithm which orients trivial edges of G by means of an iterative procedure, thus gradually shrinking directed paths (cycles) to directed paths (cycles) of length at most 2 (4). The oriented trivial edges are added to the set $\overrightarrow{E_C}$ producing a set $\overrightarrow{E'_C}$.

Algorithm for the Detection of Directed Cycles in the P_4-components of G (P4C_DDC)

1. $\overrightarrow{E'_C} := \overrightarrow{E_C}$;

2. Repeat

   2.1 $Q := \emptyset$;

   2.2 for every edge $\overrightarrow{xy}$ in $\overrightarrow{E'_C}$ do in parallel
   for every vertex z of G adjacent to y do in parallel
   if the edge xz is an edge of G
   then if the edge xz has not yet been oriented
   orient it from x to z, and add $\overrightarrow{xz}$ to Q;
   else if it is oriented from z to x
   there exists a directed cycle; exit;

   2.3 for every edge $\overrightarrow{xy}$ in $\overrightarrow{E'_C}$ do in parallel
   for every edge ab of G do in parallel
   if there exists an edge between a and x and it is $\overrightarrow{ax}$ and
   there exists an edge between y and b and it is $\overrightarrow{yb}$ then
   then if the edge ab has not yet been oriented
   orient it from a to b, and add $\overrightarrow{ab}$ to Q;
   else if it is oriented from b to a
   there exists a directed cycle; exit;
   if there exists an edge between a and y and it is $\overrightarrow{ya}$ and
   there exists an edge between x and b and it is $\overrightarrow{bx}$
   then if the edge ab has not yet been oriented
   orient it from b to a, and add $\overleftarrow{ab}$ to Q;
   else if it is oriented from a to b
   there exists a directed cycle; exit.

   2.4 $\overrightarrow{E'_C} := \overrightarrow{E'_C} \cup Q$;

   until $Q = \emptyset$;

The following lemma is crucial for the operation of the algorithm.

Lemma 3.1. *For every chordless directed path ρ of the graph $G\langle\overrightarrow{E_C}\rangle$ whose length is at least 4, one iteration of the repeat loop of Algorithm P4C_DDC produces another directed path on edges of the graph G with the same endpoints as ρ and whose length does not exceed $5/6$ of ρ's length.*

Proof: Let the length of the path ρ be k; then $k \geq 4$. We see ρ as the concatenation of $\lfloor k/3 \rfloor$ directed P_4s of $G\langle\overrightarrow{E_C}\rangle$, followed by at most two additional edges. Since none of these directed P_4s is a P_4 of the input graph G (because of the orientations of its edges), each such directed P_4 has a chord, which is a trivial edge. The edge may span two or three edges of the directed P_4; in either case, this edge will be assigned an orientation at the execution of the repeat loop (see Substeps 2.2 and 2.3). In this way, there is a directed edge "short-cutting" two or three edges for every one of these directed P_4s. Thus, a new directed path of length at most $k - \lfloor k/3 \rfloor = \lceil 2k/3 \rceil$ with the same endpoints as ρ is produced. Since $\lceil 2k/3 \rceil \leq (2k+2)/3 \leq 5k/6$ for $k \geq 4$, the lemma follows. ∎

The correctness of the algorithm is established by the following two lemmata.

Lemma 3.2. *Algorithm P4C_DDC (upon completion) has oriented every trivial edge for which the graph $G\langle\overrightarrow{E_C}\rangle$ contains a directed path from one endpoint of the edge to the other.*

Proof: Consider a trivial edge xy such that there exists a directed path from x to y in the graph $G\langle\overrightarrow{E_C}\rangle$. Then, there is a chordless such path; let it be ρ. Then, the algorithm will process ρ as described in Lemma 3.1 thus resulting in a cordless directed path from x to y of length less than $5/6$ of the length of ρ. Repeating this over and over, the resulting path will eventually be of length 2 or 3, when this process can no longer be applied; then, the edge xy will be oriented from x to y. ∎

Lemma 3.3. *Algorithm P4C_DDC correctly identifies whether the graph $G\langle\overrightarrow{E_C}\rangle$ contains a directed cycle.*

Proof: If $G\langle\overrightarrow{E_C}\rangle$ contains a cycle, then the algorithm will shrink it as described in Lemma 3.1, eventually yielding a directed triangle (P_3) or a directed P_4. But then, the directed P_3 or directed P_4 will be detected by the algorithm, and the input graph will correctly be characterized as not being a P_4-comparability graphs. On the other hand, if the algorithm reports that there exists a directed cycle, then it detected either a directed P_3 or a directed P_4; this may either belonged to $G\langle\overrightarrow{E_C}\rangle$, or was formed by edges that were oriented because there was a directed path in $G\langle\overrightarrow{E_C}\rangle$ leading from one of their endpoints to the other. In either case, $G\langle\overrightarrow{E_C}\rangle$ contains a directed cycle, and thus the algorithm responded correctly. ∎

Time and Processor Complexity. We mention that each of the sets $\overrightarrow{E'_C}$ and Q of edges is maintained as an array of size m (one for each edge of G) where it is recorded whether the corresponding edge belongs to the set. Moreover, Lemma 3.1 implies that the number of iterations of the repeat loop is $O(\log m) = O(\log n)$: the length of the longest directed path (or cycle) is $O(m)$, and at every iteration each directed path is "short-cut" by a directed path of length which is at most a constant factor (less than 1) of the length of the previous path.

Step 1: It is easy to see that this step can be executed in $O(1)$ time with $O(m)$ processors or in $O(\log^2 n)$ time with $O(m/\log^2 n)$ processors on the EREW PRAM model.

Step 2: Substep 2.1 takes constant time using $O(m)$ processors on the EREW PRAM model. Substep 2.4 can be executed in the same time and processor complexity on the EREW PRAM model, in light of the way the sets $\overrightarrow{E'_C}$ and Q are maintained.

Considering a single iteration of the repeat loop, we note that Substeps 2.2 and 2.3 involve $O(nm)$ tests and $O(m^2)$ tests respectively. Because of the way the set Q is maintained, adding an edge to Q takes constant time on the EREW PRAM model. Therefore, if assigning an orientation to an edge is done in constant time on the CREW PRAM model, then each iteration of the repeat loop can be executed in $O(1)$ time using $O(m^2)$ processors, or in $O(\log n)$ time using $O(m^2/\log n)$ processors on the CREW PRAM model. Assigning the orientation of an edge in a brute force manner does it in constant time, but it has the risk of concurrent write operation on the same memory location. We show next that this can be avoided by maintaining for each edge e an array $K_e[1..m]$ which records the different orientations that are assigned to e; the array $K_e[1..m]$ is cleared to 0 for all the edges of G at the beginning of every iteration of the repeat loop. In particular, if a directed path $\overrightarrow{uwv}$ assigns the orientation $\overrightarrow{uv}$ (Substep 2.2), then the entry $K_{uv}[uw]$ is set equal to 1; if a directed path $\overrightarrow{vpqu}$ assigns the orientation $\overleftarrow{uv}$ (Substep 2.3), then the entry $K_{uv}[pq]$ is set equal to -1. It is not difficult to see that the above method ensures exclusive write. In Substep 2.2, only entries of the form $K_e[e']$, where the edges e and e' are adjacent, are filled. Such an entry, however, is filled (if ever) during the consideration of a single directed path. For example, the entry $K_{uv}[uw]$ is filled during the consideration of the directed path uwv, if such a path exists. In Substep 2.3, only entries of the form $K_e[e']$, where the edges e and e' are not adjacent, are filled. Such an entry, however, is filled (if ever) during the consideration of a directed path with the four vertices of the edges e and e'. There may be only two such paths, but it turns out that they cannot exist simultaneously. For example, the entry $K_{uv}[pq]$ is filled during the consideration of either the directed path $upqv$ or the directed path $uqpv$. However, if both of these paths exist then the triangle upq forms a directed triangle, which would have been detected during the execution of Substep 2.2 at the same iteration, and thus the algorithm would have stopped and would not have entered Substep 2.3. Finally, since different sets of entries of the array K_e are filled in Substep 2.2 and 2.3, there cannot be loss of information due to overwriting during the same iteration.

Let us now see how the orientation of an edge can be extracted from the array K_e. At the end of each iteration of the repeat loop, this array may contain 0s, 1s, and -1s. If it contains both 1s and -1s, then there have been incompatible orientation assignments. In order to detect that, we compute the maximum and the minimum of the elements in the array. If the maximum is equal to 1 and the minimum is equal to -1, then there is incompatibility in the assigned orientations. This implies that there is a directed cycle in a P_4-component and that the input graph is not a P_4-comparability graph. If the maximum is equal to 1, then the final orientation is $\overrightarrow{uv}$. If the minimum is equal to -1, then the final orientation is $\overleftarrow{uv}$. (In the remaining case when both the minimum and the maximum are equal to 0, nothing is done, since no orientation has been assigned during the current iteration of the repeat loop.) Since the minimum and the maximum of the elements of an array of size N can be computed in $O(\log N)$ time using $O(N/\log N)$ processors on the EREW PRAM model, deciding the orientation of an edge takes $O(\log m)$ time using $O(m/\log m)$ processors on the EREW PRAM model. Taking into account the time and processor complexities of the Substeps 2.1-2.4 and the complexity of the handling of the arrays K_e, and that for a connected graph on n vertices and m edges, $O(\log m) = O(\log n)$, we have that the execution of one iteration of the repeat loop can be completed in $O(\log n)$ using $O(m^2/\log n)$ processors on the CREW PRAM model. Since the number of iterations is $O(\log n)$, this implies that the entire Step 2 takes $O(\log^2 n)$ using $O(m^2/\log n)$ processors on the CREW PRAM model.

Thus, we have the following result.

Theorem 3.4. *It can be decided whether the P_4-components of a connected simple graph on n vertices and m edges contain directed cycles in $O(\log^2 n)$ time using a total of $O(m^2/\log n)$ processors on the CREW PRAM model.*

Our results from Section 3 imply the following theorem.

Corollary 3.2. *It can be decided whether a connected simple graph on n vertices and m edges is a P_4-comparability graph in $O(\log^2 n)$ time using a total of $O(m^2/\log n)$ processors on the CREW PRAM model.*

3.5. The Case of a Disconnected Input Graph. If the input graph is disconnected, we compute its connected components, and apply the Algorithm P4G_REC in each one of them. The connected components can be computed in $O(\log^2 n)$ time using $O((n+m)/\log n)$ processors on the EREW PRAM model [20]. If n_i and m_i are the number of vertices and edges of the i-th connected component, then its processing requires $O(\log^2 n_i)$ time using $O(m_i^2/\log n_i)$ processors on the CREW PRAM model (Corollary 3.1). If $m_i \geq n$, then $m_i = \Theta(n)$, and hence $O(m_i^2/\log n_i) = O(m_i^2/\log m_i) = O(m_i^2/\log n)$, since $\log n_i = \log m_i$. Moreover, $\log^2 n_i = O(\log^2 n)$. If $m_i < n$, then we can batch $\log^2 n/\log^2 n_i$ tasks of unit time duration and assign them to a single processor; in this way the needed processors are reduced by a factor of $\log^2 n/\log^2 n_i$, while at the same time the time increases by the same factor. In particular, the time needed becomes $O(\log^2 n)$, while the number of processors $O(m_i^2/\log n)$, since $\log n_i \leq \log n$ and $\log n_i = \Theta(\log m_i)$ because the component is connected. Consequently, no matter whether m_i is less or greater than n, we can process the i-th connected component in $O(\log^2 n)$ time using $O(m_i^2/\log n)$ processors in the CREW PRAM model. Thus, we can process all the connected components in $O(\log^2 n)$ time using a total of $O(m^2/\log n)$ processors on the same model of parallel computation. Therefore, we have the following theorem.

Theorem 3.5. *It can be decided whether a simple graph on n vertices and m edges is a P_4-comparability graph in $O(\log^2 n)$ time using a total of $O((n+m^2)/\log n)$ processors on the CREW PRAM model.*

4. Acyclic P_4-transitive Orientation

The orientation algorithm that we describe here takes advantage of the orientation of the graph $G\langle\overrightarrow{E'_C}\rangle$ produced by the recognition algorithm of the previous section and orients the edges that have not received an orientation at the end of the recognition process. It relies on the following two lemmata.

Lemma 4.1. *In the directed graph $G\langle\overrightarrow{E'_C}\rangle$, the length of the shortest directed path between any pair of vertices does not exceed 2.*

Proof: Suppose for contradiction that there are two vertices such that the length of the shortest directed path from the one to the other exceeds 2. Then, there exist two vertices u and v such that the length of the shortest path from u to v is equal to 3; let $uabv$ be that path. Since this path cannot be a P_4 because of the orientations assigned to its edges, then there must be an edge between at least one of the following pairs of vertices: u and b, u and v, a and v. Note that, in any case, this edge is assigned an orientation during the last step of the recognition algorithm and this orientation is from u to b, from u to v, and from a to v respectively (Lemma 3.2). However, this contradicts the fact that the path $uabv$ is the shortest directed path from u to v, thus establishing the lemma. ∎

Lemma 4.2. *Let $\overrightarrow{ab}$ be a (directed) edge of the graph $G^*\langle\overrightarrow{E'_C}\rangle$, which is the transitive closure of the directed graph $G\langle\overrightarrow{E'_C}\rangle$. Then, the indegree of the vertex b is larger than the indegree of the vertex a.*

Proof: The transitive closure implies that the indegree of a vertex v of $G^*\langle\overrightarrow{E'_C}\rangle$ is equal to the number of vertices of $G\langle\overrightarrow{E'_C}\rangle$ such that there is a directed path from each of these vertices to v. Let $P(a)$ and $P(b)$ be the sets of vertices of $G\langle\overrightarrow{E'_C}\rangle$ such that there is a directed path from each of these vertices to a and b respectively. Then, we need to show that $|P(a)| < |P(b)|$. It is not difficult to see that $P(a) \subset P(b)$: every vertex in $P(a)$ also belongs to $P(b)$, since due to the edge $\overrightarrow{ab}$, a directed path from a vertex to a implies that there is a directed path from that vertex to b; additionally, because there are no directed cycles in $G\langle\overrightarrow{E'_C}\rangle$, $a \notin P(a)$ whereas $a \in P(b)$. ∎

Our orientation algorithm involves the following algorithmic steps.

Algorithm for the Acyclic P_4-transitive Orientation of a Graph G (P4G_TRO)

1. Apply the recognition procedure that we described in the previous section. If the input graph G is not a P_4-comparability graph, then the algorithm stops printing the corresponding diagnostic message; otherwise, the recognition procedure computes the directed graph $G\langle\overrightarrow{E'_C}\rangle$.

2. Compute the transitive closure $G^*\langle\overrightarrow{E'_C}\rangle$ of the graph $G\langle\overrightarrow{E'_C}\rangle$.

3. Compute the indegree(v) of each vertex of the graph $G^*\langle\overrightarrow{E'_C}\rangle$; set the indegree of very vertex of G which is not a vertex of $G^*\langle\overrightarrow{E'_C}\rangle$ equal to 0;

4. Orient the edges of G that have not yet been assigned an orientation: for such an edge xy, if indegree(x) > indegree(y) then $\overleftarrow{xy}$; if indegree(x) < indegree(y) then $\overrightarrow{xy}$; if indegree(x) = indegree(y) then xy is oriented towards that among x and y which has the smaller index (we assume that each vertex of G possesses a distinct index number).

Note that, in light of Lemma 4.1, the computation of the transitive closure $G^*\langle\overrightarrow{E'_C}\rangle$ can be done by adding a directed edge $\overrightarrow{uv}$ for a directed P_3 $\overrightarrow{uwv}$. Therefore, for each directed edge $\overrightarrow{ab}$ of $G\langle\overrightarrow{E'_C}\rangle$, we go through each vertex c of G adjacent to b and check whether the path abc is a directed P_3 of $G\langle\overrightarrow{E'_C}\rangle$; if yes, then the directed edge $\overrightarrow{ac}$ needs to be added. To avoid concurrent writes, for each vertex v we use an $(\text{degree}(v))^2$-array

$H_v\Big[\text{rank}(v,x), \text{rank}(v,y)\Big]$, where x and y are vertices of G. If the edge $\overrightarrow{ab}$ and the vertex c form a directed P_3 abc, then we record the fact that a directed edge $\overrightarrow{ac}$ needs to be added by setting the entry $H_a\Big[\text{rank}(a,c), \text{rank}(a,b)\Big]$ to 1; the entry $H_a\Big[\text{rank}(a,c), \text{rank}(a,b)\Big]$ uniquely corresponds to the path abc. In the end, the transitive closure is produced by adding to $G\langle\overrightarrow{E'_C}\rangle$ the edges $\overrightarrow{uv}$ for which there is a 1 in the subarray $H_u[\text{rank}(u,v), *]$; this can be found in $O(\log^2 n)$ time with $O(m^2/\log^2 n)$ processors using standard interval prefix computations on the EREW PRAM model [2] (note that the total size of all the H arrays is $\sum_v(\text{degree}(v))^2 = O(m^2)$).

The correctness of the algorithm follows from the following lemma.

Lemma 4.3. *For a P_4-comparability graph G, the algorithm P4G_TRO completes all the steps of its description and produces an acyclic P_4-transitive orientation of G.*

Proof: Since the input graph G is a P_4-comparability graph, then Step 1 is completed successfully, and so are the remaining steps of the algorithm. Clearly all the edges of G are assigned an orientation. Furthermore, according to the discussion in the previous section, the orientation of the directed graph $G\langle\overrightarrow{E'_C}\rangle$ is P_4-transitive and therefore so is the resulting orientation. Additionally, since Step 1 of the algorithm P4G_TRO is completed successfully, then the orientation of $G\langle\overrightarrow{E'_C}\rangle$ is also acyclic.

Therefore, we need to show that the edges that were oriented in Step 4 did not cause the formation of a directed cycle. Suppose for contradiction that the resulting directed graph contains a directed cycle. Then, it contains a directed triangle (Lemma 2.2); let its vertices be a, b, c, and suppose without loss of generality that $\text{index}(a) < \text{index}(b) < \text{index}(c)$. Because the orientation of the graph $G\langle\overrightarrow{E'_C}\rangle$ is acyclic and because of Lemma 3.2, at least two of the directed triangle's edges were oriented at Step 4 of the algorithm P4G_TRO.

We distinguish the following cases:

1. *All three edges ab, bc, ac receive their orientations in Step 4 of the algorithm P4G_TRO.*
   (i) the orientation of the edges is: $\overrightarrow{ab}$, $\overrightarrow{bc}$, $\overrightarrow{ca}$. Then, $\overrightarrow{ab}$ implies that $\text{indegree}(a) < \text{indegree}(b)$, $\overrightarrow{bc}$ implies that $\text{indegree}(b) < \text{indegree}(c)$, and $\overrightarrow{ca}$ implies that $\text{indegree}(c) \le \text{indegree}(a)$. These three inequalities lead to contradiction.
   (ii) the orientation of the edges is: $\overleftarrow{ab}$, $\overleftarrow{bc}$, $\overleftarrow{ca}$ In this case, the corresponding inequalities are: $\text{indegree}(b) \le \text{indegree}(a)$, $\text{indegree}(c) \le \text{indegree}(b)$, and $\text{indegree}(a) < \text{indegree}(c)$; a contradiction again.

2. *Two of the three edges ab, bc, ac receive their orientations during Step 4 of the algorithm P4G_TRO.*
   (i) Suppose that the edges which get oriented during Step 4 are the edges ab and bc.
      - ▷ the orientation of the edges is: $\overrightarrow{ab}$, $\overrightarrow{bc}$, $\overrightarrow{ca}$. Then, $\overrightarrow{ab}$ implies that $\text{indegree}(a) < \text{indegree}(b)$, $\overrightarrow{bc}$ implies that $\text{indegree}(b) < \text{indegree}(c)$, and $\overrightarrow{ca}$ implies that $\text{indegree}(a) > \text{indegree}(c)$ (see Lemma 4.2). These three inequalities lead to contradiction.
      - ▷ the orientation of the edges is: $\overleftarrow{ab}$, $\overleftarrow{bc}$, $\overleftarrow{ca}$ In this case, the corresponding inequalities are: $\text{indegree}(b) \le \text{indegree}(a)$, $\text{indegree}(c) \le \text{indegree}(b)$, and $\text{indegree}(c) < \text{indegree}(a)$; a contradiction again.

(ii) The remaining two cases (where the edges that get oriented during Step 4 are the edges ab, ac and bc, ac respectively) are handled similarly. ∎

Time and Processor Complexity. Step 1 takes $O(\log^2 n)$ time using a total of $O((n+m^2)/\log n)$ processors on the CREW PRAM model (Theorem 3.5). As described above, the process of computing the transitive closure $G^*\langle\overrightarrow{E'_C}\rangle$ is based on the processing of all pairs of adjacent edges; thus, it can be carried out in $O(\log^2 n)$ time using a total of $O(m^2/\log^2 n)$ processors on the CREW PRAM model. Moreover, this process implies that the graph $G^*\langle\overrightarrow{E'_C}\rangle$ has n vertices and $O(m^2)$ edges. Therefore, the computation of the indegrees of its vertices can be done in $O(\log n)$ time with $O(m^2/\log n)$ processors (or equivalently in $O(\log^2 n)$ time with $O(m^2/\log^2 n)$ processors) on the CREW PRAM model. Obviously, Step 4 takes $O(1)$ time and requires $O(m)$ processors on the CREW PRAM model. In summary, we have the following result.

Theorem 4.1. *An acyclic P_4-transitive orientation of a simple graph G on n vertices and m edges can be produced in $O(\log^2 n)$ time using a total of $O((n+m^2)/\log n)$ processors on the CREW PRAM model.*

5. Concluding Remarks

In this paper we present efficient parallel algorithms for recognizing P_4-comparability graphs and for computing an acyclic P_4-transitive orientation. Both algorithms run in $O(\log^2 n)$ time using a total of $O((n+m^2)/\log n)$ processors on the CREW PRAM model, where n and m are the number of vertices and edges of the input graph.

Our algorithms are simple and rely on certain algorithmic and structural properties of the P_4-components of a graph [29]. To the best of our knowledge, in a sequential process environment, the currently fastest recognition and acyclic P_4-transitive orientation algorithms for P_4-comparability graphs exhibit a time complexity of $O(n+m^2)$ [29, 30].

We conjecture (but we are unable to prove) that if a P_4-component contains a directed P_3 or a directed P_4 with a trivial edge connecting the endpoints of these paths, then this P_4-component contains a directed cycle. It would be nice to find out whether this is true because it would lead to a faster directed cycle detection algorithm. If the above conjecture is true, then a slightly modified version of our P_4-comparability recognition algorithm would be cost optimal on the CREW PRAM model.

The obvious open question is whether we can design cost-optimal parallel algorithm for the above problems on the CREW PRAM model. Moreover, cost-optimal or at least efficient algorithms are needed for other well-known / important combinatorial and optimization problems on P_4-comparability graphs, such as the coloring problem, the maximum clique problem, the maximal clique and the clique cover problem, etc. We note that, due to the work of Chvátal [5], the coloring problem and the maximum clique problem can be solved in linear sequential time if an acyclic P_4-transitive orientation of the input graph is given.

6. References

[1] G.S. Adhar and S. Peng, Parallel algorithms for cographs and parity graphs with applications, *J. of Algorithms* 11 (1990), 252–284.

[2] S.G. Akl, *Parallel Computation: Models and Methods*, Prentice Hall, 1997.

[3] S.R. Arikati and U.N. Peled, A polynomial algorithm for the parity path problem on perfectly orderable graphs, *Discrete Appl. Math.* 65 (1996), 5–20.

[4] P. Beame and J. Hastad, Optimal bounds for decision problems on the CRCW PRAM, *J. Assoc. Comput. Mach.*, 36 (1989), 643–670.

[5] V. Chvátal, Perfectly ordered graphs, *Annals of Discrete Math.* 21 (1984), 63–65.

[6] D.G Corneil, H. Lerches and L. Burlingham, Complement reducible graphs, *Discrete Appl. Math.* 3 (1981), 163–174.

[7] D.G. Corneil, Y. Perl and L.K. Stewart, A linear recognition algorithm for cographs, *SIAM J. Comput.* 14 (1985), 926–934.

[8] E. Dahlhaus, Efficient parallel recognition algorithms of cographs and distance hereditary graphs, *Discrete Appl. Math.* 57 (1995), 29–44.

[9] E. Dahlhaus, Parallel algorithms for hierarchical clustering and applications to split decomposition and parity graph recognition, *J. of Algorithms* 36 (2000), 205–240.

[10] S. de Agostino and R. Petreschi, Parallel recognition algorithms for graphs with restricted neighbourhoods, *Inter. J. of Found. of Comp. Science* 1 (1990), 123–130.

[11] C.M.H. de Figueiredo, J. Gimbel, C.P. Mello, and J.L. Szwarcfiter, Even and odd pairs in comparability and in P_4-comparability graphs, *Discrete Appl. Math.* 91 (1999), 293–297.

[12] P.C. Gilmore and A.J. Hoffman, A characterization of comparability graphs and of interval graphs, *Canad. J. Math.* 16 (1964), 539–548.

[13] M.C. Golumbic, The complexity of comparability graph recognition and coloring, *Computing* 18 (1977), 199–208.

[14] M.C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, Inc., New York, 1980.

[15] D. Helmbold and E.W. Mayr, Applications of parallel algorithms to families of perfect graphs, *Computing* 7 (1990), 93–107.

[16] C.T. Hoàng and B.A. Reed, Some classes of perfectly orderable graphs, *J. Graph Theory* 13 (1989), 445–463.

[17] C.T. Hoàng and B.A. Reed, P_4-comparability Graphs, *Discrete Math.* 74 (1989), 173–200.

[18] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.

[19] P.N. Klein, Efficient parallel algorithms for chordal graphs, *Proc. 29th Symp. Found. of Comp. Sci.* (1989), 150–161.

[20] C.P. Kruskal, L. Rudolph and M. Snir, Efficient parallel algorithms for graph problems, *Algorithmica* 5 (1990), 43–64.

[21] R. Lin and S. Olariu, An NC recognition algorithm for cographs, *J. of Parallel and Distrib. Comput.* 13 (1991), 76–90.

[22] R.M. McConnell and J. Spinrad, Linear-time modular decomposition and efficient transitive orientation of comparability graphs, *Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms* (1994), 536–545.

[23] R.M. McConnell and J. Spinrad, Linear-time transitive orientation, *Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms* (1997), 19–25.

[24] M. Middlendorf and F. Pfeiffer, On the complexity of recognizing perfectly orderable graphs, *Discrete Math.* 80 (1990), 327–333.

[25] M. Morvan and L. Viennot, Parallel comparability graph recognition and modular decomposition, *Proc. 13th Symposium on Theoretical Aspects of Computer Science STACS '96*, Lecture Notes in Computer Science 1046 (1996), 169–180.

[26] D. Nash and S.N. Maheshwari, Parallel algorithms for the connected components and minimal spanning trees, *Inform. Process. Lett.* 14 (1982), 7–11.

[27] S.D. Nikolopoulos, Constant-time parallel recognition of split graphs, *Inform. Process. Lett.* 54 (1995), 1–8.

[28] S.D. Nikolopoulos, Coloring permutation graphs in parallel, *Elect. Notes Discrete Math.* (Elsevier), 3 (1999), 181–187.

[29] S.D. Nikolopoulos and L. Palios, Recognition and orientation algorithms for P_4-comparability graphs, *Technical Report* TR-23-2000 (2000), Department of Computer Science, University of Ioannina, Ioannina, Greece.

[30] T. Raschle and K. Simon, On the P_4-components of graphs, *Discrete Appl. Math.* 100 (2000), 215–235.

[31] J. Reif (ed.), *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers, San Mateo, California, 1993.

[32] J. Spinrad, On comparability and permutation graphs, *SIAM J. Comput.* 14 (1985), 658–670.