Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Algorithms 51 (2004) 77–104
Journal of Algorithms
www.elsevier.com/locate/jalgor

Parallel algorithms for P_4-comparability graphs

Stavros D. Nikolopoulos * and Leonidas Palios

Department of Computer Science, University of Ioannina, GR-45110 Ioannina, Greece

Received 21 June 2002

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 such a 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 describe 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 $O(nm/\log n)$ processors on the CREW PRAM model. Since the currently fastest sequential algorithms for these problems run in $O(nm)$ time, our algorithms are cost-efficient; moreover, to the best of our knowledge, this is the first attempt to introduce parallelization in problems involving P_4-comparability graphs. 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 U contains $\overrightarrow{ab}$ and $\overleftarrow{bc}$, or $\overleftarrow{ab}$ and $\overrightarrow{bc}$, where by $\overrightarrow{uv}$ or $\overleftarrow{vu}$ we denote an edge directed from u to v. The relation F is

* Corresponding author.
E-mail addresses: stavros@cs.uoi.gr (S.D. Nikolopoulos), palios@cs.uoi.gr (L. Palios).

0196-6774/$ – see front matter
doi:10.1016/j.jalgor.2003.11.005

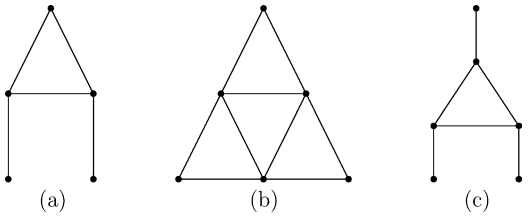

Fig. 1. (a) A comparability graph, (b) a P_4-comparability graph, (c) a graph which is not P_4-comparability.

called a *transitive orientation* of E or equivalently of the graph 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]; Fig. 1(a) depicts a comparability graph. A graph is a P_4*-comparability graph* if it admits an acyclic P_4-transitive orientation [15,16]. In light of these definitions, every comparability graph is a P_4-comparability graph. Moreover, there exist P_4-comparability graphs which are not comparability; Fig. 1(b) depicts such a graph, which is often referred to as a pyramid. The graph shown in Fig. 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 is perfectly orderable [23]. 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 (also known as bipolarizable) graphs, and proved that they are all perfectly orderable [16]. Moreover, the class of perfectly orderable graphs also includes a number of other classes of graphs which are characterized by important algorithmic and structural properties; we mention the classes of brittle, co-chordal, HHD-free, Meyniel ∩ 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, such as, recognition as well as problems pertaining to finding maximum cliques, maximum weighted cliques, maximum independent sets, optimal coloring, breadth-first search trees and depth-first search trees, hamiltonian paths and cycles, and testing graphs for isomorphism [1,3,6–13,15,16,18–22, 24–29,31].

The comparability graphs in particular have been the focus of much research in recent years which culminated into efficient recognition and orientation algorithms. Golumbic presented algorithms for recognizing and assigning transitive orientations on comparability graphs in $O(dm)$ time and $O(n+m)$ space, where n, m, and d are the number of

vertices, the number of edges, and the maximum degree of the input graph respectively [13,14]. Due to the work of McConnell and Spinrad [21,22], the modular decomposition and transitive orientation problems for comparability graphs can be solved in $O(n+m)$ time. This gives linear time bounds for maximum clique and minimum vertex coloring on comparability graphs, as well as other combinatorial problems on comparability graphs and their complements. Recently, Morvan and Viennot [24] presented parallel algorithms for the recognition and the computation of a transitive orientation of comparability graphs; their algorithms run in $O(\log n)$ time and require $O(dm)$ 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 [11,15,16,28,29]. 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 [15,16]. 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 [29]; their algorithms for either problem run in $O(n+m^2)$, where m is the number of edges of G. Different $O(n+m^2)$-time recognition and acyclic P_4-transitive orientation algorithms for P_4-comparability graphs were presented by Nikolopoulos and Palios [27], while recently the same authors improved their algorithms achieving an $O(nm)$-time and $O(n+m)$-space complexity [28].

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 [2,17,30]. Both algorithms run in $O(\log^2 n)$ time using a total of $O(nm/\log n)$ 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. Our algorithms are cost efficient and, to the best of our knowledge, they are the first parallel algorithms for problems involving P_4-comparability graphs.

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 lemmas. We describe and analyze the recognition and acyclic P_4-transitive orientation algorithms in Sections 3 and 4, respectively, while in Section 5 we conclude with a summary of our results, extensions and open problems.

2. Theoretical framework

We consider simple non-trivial graphs. Let G be such a graph; we denote the vertex set and edge set of G by $V(G)$ and $E(G)$ respectively. A *path* in G is a sequence of vertices $v_0v_1\ldots v_k$ such that $v_{i-1}v_i \in E(G)$ 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. The path is undirected or directed depending on whether G is an undirected or a directed graph; a *directed* path $v_0v_1\ldots v_k$ 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) $v_0v_1 \ldots v_k$ is called a *cycle* (*simple cycle*) of length $k+1$ if $v_0v_k \in E(G)$. A simple path (cycle) $v_0v_1 \ldots v_k$ is *chordless* if $v_iv_j \notin E(G)$ 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 the *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 a 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 [16] 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 [16, Theorem 3.1] can be paraphrased in terms of the P_4-components as follows.

Lemma 2.1 [16]. *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 after inversion of the orientations 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 [16].

Lemma 2.2 [16, Lemma 3.5]. *If a proper orientation of an interesting graph is cyclic, then it contains a directed triangle.*[1]

[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.

For a non-trivial P_4-component $\mathcal{C}$, the set of vertices $V(G) - V(\mathcal{C})$ can be partitioned into three sets: the set R contains the vertices of $V(G) - V(\mathcal{C})$ which are adjacent to some (but not all) of the vertices in $V(\mathcal{C})$, the set P contains the vertices of $V(G) - V(\mathcal{C})$ which are adjacent to all the vertices in $V(\mathcal{C})$, and the set Q contains the vertices of $V(G) - V(\mathcal{C})$ 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 [29], 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 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 [29, 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:

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 $r \in R$, $v_1, v_1' \in V_1$, $v_2 \in V_2$.

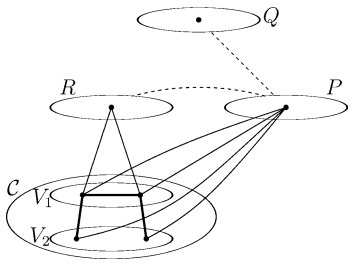

Fig. 2.

[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.

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 [29, Lemma 3.4], which implies that:

Lemma 2.4 [29]. *Let $\mathcal{C}$ be a non-trivial P_4-component of a graph G. 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 $V(\mathcal{C}) \subset V(G)$, and let $S_{\mathcal{C}}$ be the set of non-trivial P_4-components of G which have a vertex belonging to $V(\mathcal{C})$ as well as a vertex not in $V(\mathcal{C})$. Then, each of the P_4-components in $S_{\mathcal{C}}$ contains a P_4 of type (1)–(8) with respect to $\mathcal{C}$. Additionally, if we take Lemma 2.4 into account, we can partition the elements of $S_{\mathcal{C}}$ into two sets as follows.

- *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 lemma is the heart of our algorithm for computing an acyclic P_4-transitive orientation of a P_4-comparability graph (for the proof, see [28]).

Lemma 2.5. *Let $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_\ell$ 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 $S_i = \{\mathcal{C}_j \mid j < i$ and $\mathcal{C}_i$ 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 $(1 \leqslant i \leqslant \ell)$ does not contain a directed cycle.*

Notation. Let G be a simple graph. Hereafter, the subgraph of G induced by a vertex subset $S \subseteq V(G)$ is denoted by $G[S]$ and the subgraph spanned by an edge subset $W \subseteq E(G)$ is denoted by $G\langle W\rangle$. In the case that all the edges in W have been assigned an orientation, the directed subgraph spanned by the oriented edges is denoted by $G\langle \vec{W}\rangle$.

Additionally, we will be assuming that the input graph G has k P_4-components, among which ℓ are non-trivial $(1 \leqslant \ell \leqslant k)$. Without loss of generality, we assume that the ℓ non-trivial P_4-components are $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_\ell$ in order of increasing vertex number, while the trivial ones are $\mathcal{C}_{\ell+1}, \ldots, \mathcal{C}_k$.

Finally, with a slight abuse of notation, we will be using 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 be a connected simple graph on n vertices and m edges. Then, $n = O(m)$ and $\log m = \Theta(\log n)$.

Let $E_{\mathcal{C}}$ and $E_{\mathcal{T}}$ be the sets of the edges of all the non-trivial and trivial P_4-components of G respectively; because the edges in $E_{\mathcal{T}}$ 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 $E(G)$ is equal to the disjoint union of $E_{\mathcal{C}}$ and $E_{\mathcal{T}}$.

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

Algorithm REC_P4G (*for the recognition of a P_4-comparability graph G*).

Phase I. *Compute the P_4-components of the graph G and a P_4-transitive orientation* (*if one exists*) *of each one of them.*

1. Compute the P_3s that participate in P_4s in G: compute the BFS-trees (up to the 3rd level) of the complement $\overline{G}$ of the graph G rooted at each of G's vertices and extract from them the sought P_3s.
2. Compute and P_4-transitively orient the P_4-components of G: construct an auxiliary graph $\widehat{G}$ which has $2m$ vertices (two vertices $\hat{u}_{xy}$ and $\hat{u}_{yx}$ for each edge xy of G), and $O(nm)$ edges recording information on the P_3s of G which participate in P_4s; two vertices $\hat{u}_{xy}$ and $\hat{u}_{zy}$ (respectively $\hat{u}_{yx}$ and $\hat{u}_{yz}$) are adjacent in $\widehat{G}$ iff xyz is a P_3 in G participating in a P_4 in G. Then, the connected components of $\widehat{G}$ yield the P_4-components of G while the subscripts of the vertices in these connected components yield the P_4-transitive orientations of the P_4-components.

Let $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_\ell$ be the non-trivial P_4-components of the input graph G and let E_1, E_2, $\ldots$, E_ℓ be their edge sets.

Phase II. *Check for directed cycles in the oriented P_4-components.*

3. Combine the oriented non-trivial P_4-components: compute appropriate inversions (if needed) of the P_4-transitive orientations $G\langle\overrightarrow{E_1}\rangle, G\langle\overrightarrow{E_2}\rangle, \ldots, G\langle\overrightarrow{E_\ell}\rangle$ of the non-trivial P_4-components $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_\ell$, so that the directed graph $G\langle\overrightarrow{E_{\mathcal{C}}}\rangle$, spanned by the directed edges in $\overrightarrow{E_{\mathcal{C}}} = \overrightarrow{E_1} \cup \overrightarrow{E_2} \cup \cdots \cup \overrightarrow{E_\ell}$, is acyclic if each P_4-transitive orientation $G\langle\overrightarrow{E_i}\rangle$ is acyclic.
4. Detect directed cycles in the P_4-transitive orientations of the non-trivial P_4-components: locate all those trivial edges of the graph G for which the directed graph $G\langle\overrightarrow{E_{\mathcal{C}}}\rangle$ contains a directed path from one of their endpoints to the other, and orient them so that no cycle is formed (i.e., the orientation matches the direction of the path). Add the oriented edges to $G\langle\overrightarrow{E_{\mathcal{C}}}\rangle$ producing $G\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$. This has the effect that any directed cycles in $G\langle\overrightarrow{E_{\mathcal{C}}}\rangle$ give rise to a directed triangle (C_3) or a directed C_4 in $G\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$. Then,

G is a P_4-comparability graph if and only if $G\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$ does not contain a directed C_3 or a directed C_4.

Steps 1 and 2 compute and orient the P_4-components of the input graph G; note that step 2 detects if a P_4-component cannot admit a P_4-transitive orientation (for example, if the graph contains a C_5), in which case, the algorithm reports that G is not a P_4-comparability graph and terminates. Step 3 computes appropriate orientation inversions of the non-trivial P_4-components based on Lemma 2.5, which guarantees that if G is a P_4-comparability graph then the resulting directed graph $G\langle\overrightarrow{E_{\mathcal{C}}}\rangle$ spanned by the edges of the non-trivial P_4-components has an acyclic P_4-transitive orientation.

Remark 3.1. 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 in the directed graph $G\langle\overrightarrow{E_i}\rangle$ corresponding to the P_4-component $\mathcal{C}_i$ can be done either by computing the transitive closure of $G\langle\overrightarrow{E_i}\rangle$ or by the method employed in step 4 above. The former approach turns out to be rather expensive in the number of processors; recall that the P_4-components share vertices and that the computation of the transitive closure of a graph on n vertices takes $O(\log^2 n)$ time using $O(M(n)/\log n)$ processors on the CREW PRAM, where $M(n) \simeq n^{2.376}$ is the best-known sequential bound for the multiplication of two $(n \times n)$-size arrays [17]. The latter approach also proves to be expensive; in order to be immune to orientation conflicts on the trivial edges arising from the fact that each P_4-component is assigned an orientation independently from the orientations of the remaining P_4-components, one needs to maintain separate orientation assignments for the trivial edges for each non-trivial P_4-component. As there may exist $\Omega(m)$ trivial edges and $\Omega(m)$ non-trivial P_4-components, one needs to be able to process $\Omega(m^2)$ amount of information which may very well exceed the desired $O(nm\log n)$ cost. Therefore, both of the above approaches exhibit high computational cost.

In the following paragraphs, we present parallel implementations for each step of the proposed algorithm.

3.1. Computing the P_3s that participate in P_4s in the graph G

The P_3s that participate in P_4s in G are computed by means of the BFS-trees of the complement $\overline{G}$ of the graph G rooted at each of G's vertices. The approach is the one used in [28]; we give next the basic ideas. It is important to observe that if $abcd$ is a P_4 of G then its complement is the P_4 $bdac$ and it belongs to $\overline{G}$. If we consider the BFS tree $T_{\overline{G}}(b)$ of $\overline{G}$ rooted at b, then the vertices b, d, and a have to belong to the 0th, 1st, and 2nd level respectively; the vertex c may belong to the 2nd or 3rd level, but not to the 1st level since c is not adjacent to b in $\overline{G}$ (see Fig. 3).

In particular, a P_3 abc of the graph G participates in a P_4 in G if and only if in the BFS-tree $T_{\overline{G}}(b)$ rooted at b either exactly one of a, c belongs to the 2nd level and the other one to the 3rd level (see Fig. 3, case on the left) or both a and c belong to the 2nd level and there exists at least one vertex in the 1st level which is adjacent to exactly one

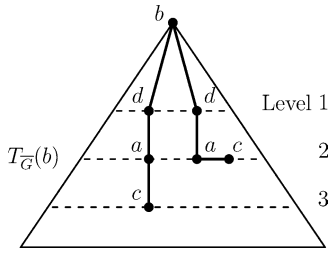

Fig. 3. The two placements of the P_4 $bdac$ in the tree $T_{\overline{G}}(b)$.

of them (see Fig. 3, case on the right). From a complexity point of view, working with the complement $\overline{G}$ of G has the important benefit that the vertices in the 2nd and 3rd level of the BFS-tree of $\overline{G}$ rooted at a vertex v are all adjacent to v in G, and thus their total number does not exceed the degree of v in G.

Next, we present a formal description of the construction procedure. The procedure uses a $(2m \times n)$-size array $M[\,]$, whose entries are denoted by $M[(a, b), c]$ where $a, b, c \in V(G)$ and a, b are adjacent in G.

Procedure COMPUTE_P3s (*for the computation of the P_3s that participate in P_4s in the graph G*).

1. Initialize to 0 all the entries of the array $M[]$.
2. For each vertex v of the graph G do in parallel
   - 2.1 compute the sets $L_1(v)$, $L_2(v)$, and $L_3(v)$ of vertices in the 1st, 2nd, and 3rd level of the BFS tree $T_{\overline{G}}(v)$ of the complement $\overline{G}$ rooted at v;
   - 2.2 partition the set $L_2(v)$ into subsets of vertices so that two vertices belong to the same subset iff they have (in $\overline{G}$) the same neighbors in $L_1(v)$;
   - 2.3 for each vertex $x \in L_2(v)$ do in parallel
     - 2.3.1 for each vertex $y \in L_3(v)$ do in parallel
       if x, y are not adjacent in G
       then $M[(v, x), y] \leftarrow 1$ {P_3 xvy participates in a P_4 in G};
       $M[(v, y), x] \leftarrow 1$;
     - 2.3.2 for each vertex $y \in L_2(v)$ do in parallel
       if x, y are not adjacent in G and x, y belong to different partition sets of $L_2(v)$, then $M[(v, x), y] \leftarrow 1$ {P_3 xvy participates in a P_4 in G}.

The correctness of Procedure COMPUTE_P3s follows from the fact that both P_3s of each P_4 of the graph G are taken into account: for a P_4 $abcd$ of G, the P_3 abc will be reported by the algorithm while processing the BFS-tree of $\overline{G}$ rooted at b, while the P_3 bcd will be reported while processing the BFS-tree rooted at c. Note that for the case on the left in Fig. 3, the P_3 abc will be reported by means of substep 2.3.1; the case on the right in Fig. 3 is covered by substep 2.3.2.

3.1.1. Time and processor complexity

We shall use a step-by-step analysis for computing the time and processor complexities of each step of Procedure COMPUTE_P3s.

1. The initialization of the $(2m \times n)$-size array $M[]$ can be carried out in $O(1)$ time using $O(nm)$ processors, or in $O(\log n)$ time using $O(nm/\log n)$ processors on the EREW PRAM model.

2. This step is executed for each vertex $v \in V(G)$ and consists of two substeps. In substep 2.1 we compute the sets $L_1(v)$, $L_2(v)$, and $L_3(v)$ of the vertices of the 1st, 2nd, and 3rd level respectively of the BFS tree $T_{\overline{G}}(v)$ of the complement $\overline{G}$ rooted at v. For this computation, we work as follows.

Computation of all the vertices of the 1*st,* 2*nd and* 3*rd levels of* $T_{\overline{G}}(v)$:

(i) compute the vertex sets $N(v)$ and $V(G) - N(v)$ of the graph G;
$L_1(v) \leftarrow V(G) - N(v)$;
(ii) $L_2(v) \leftarrow \emptyset$;
for each vertex $x_i \in N(v)$ do in parallel
compute the number n_i of neighbors of x_i (in G) which belong to $L_1(v)$;
if $n_i < |L_1(v)|$ then add x_i to $L_2(v)$;
(iii) $L_3(v) \leftarrow \emptyset$;
for each vertex $x_i \in N(v) - L_2(v)$ do in parallel
compute the number n_i of neighbors of x_i (in G) which belong to $L_2(v)$;
if $n_i < |L_2(v)|$ then add x_i to $L_3(v)$.

(i) We use an auxiliary array $A_v[]$ of length n and set $A_v[i] \leftarrow 1$, for $1 \leqslant i \leqslant n$. Then, for each vertex u_i adjacent to vertex v, we set $A_v[u_i] \leftarrow 0$. Using parallel prefix and array packing computations on $A_v[]$, we can compute the vertices of the set $V(G) - N(v)$, i.e., the vertices of the set $L_1(v)$, in $O(\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM model.

(ii) Initially, we set $A_v[i] \leftarrow 0$, for $1 \leqslant i \leqslant n$. Then, for each vertex $x_i \in N(v)$, we compute the number n_i of neighbors of x_i which belong to $L_1(v)$; this computation takes $O(\log(\deg(x_i))r)$ time and requires $O(\deg(x_i))$ processors on the CREW PRAM model. Next, we set $A_v[x_i] \leftarrow 1$ if $n_i < |L_1(v)|$; this assignment operation takes $O(1)$ sequential time. Note that the number $|L_1(v)|$ is computed in $O(\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM model. Thus, the whole step takes $O(\log k) = O(\log n)$ time and requires $O(\sum_{i=1}^{|N(v)|} \deg(x_i)r) = O(m)$ processors on the CREW PRAM model, where $k = \max_{1 \leqslant i \leqslant |N(v)|} \deg(x_i) = O(n)$.

(iii) This step is executed within the same time and processor bounds as step (ii).

Complexity of substep 2.1. Since substep 2.1 is executed for each vertex $v \in V(G)$, it requires in total $O(\log n)$ time with $O(nm)$ processors on the CREW PRAM model.

In substep 2.2 we partition the set $L_2(v)$ into subsets of vertices so that two vertices belong to the same subset if and only if they have (in $\overline{G}$) the same neighbors in $L_1(v)$. Let $x_1, x_2, \ldots, x_{k(v)}$ be the vertices of the set $L_2(v)$; clearly, $k(v) \leqslant \deg(v)$ since all the vertices in $L_2(v)$ are adjacent to v in G. Moreover, let $N_x[]$ be an array of size n such that

$N_x[y] = 1$ if $y \in L_1(v)$ and y is a neighbor of x in G and $N_x[y] = 0$ otherwise (note that two vertices of the set $L_2(v)$ have the same neighbors in $L_1(v)$ in the graph G if and only if they have the same neighbors in $L_1(v)$ in the graph $\overline{G}$). It is then obvious that two vertices $x_i, x_j \in L_2(v)$ have (in $\overline{G}$) the same neighbors in $L_1(v)$ if and only if $N_{x_i}[] = N_{x_j}[]$, that is, $N_{x_i}[p] = N_{x_j}[p]$ for $p = 1, 2, \ldots, n$. Thus, we compute an array $S_v[]$ of size $k(v)$ such that, for $x_i, x_j \in L_2(v)$, $S_v[x_i] = S_v[x_j]$ if and only if x_i, x_j have (in $\overline{G}$) the same neighbors in $L_1(v)$.

Computation of the array $S_v[]$:

(iv) $A_v[] \leftarrow [x_1, x_2, \ldots, x_{k(v)}]$, that is, $A_v[i] \leftarrow x_i$ for $i = 1, 2, \ldots, k(v)$;
(v) for each vertex $x_i \in L_2(v)$ do in parallel
  for each vertex $y \in V(G)$ do in parallel
    if $y \in L_1(v)$ and x_i is adjacent to y in G then $N_{x_i}[y] \leftarrow 1$ else $N_{x_i}[y] \leftarrow 0$;
(vi) sort the vertices $x_1, x_2, \ldots, x_{k(v)}$ in the array $A_v[]$ according to their neighbors in the set $L_1(v)$, i.e., $x_i \leqslant x_j$, iff the contents of $N_{x_i}[]$ form a string of 0s and 1s which is lexicographically smaller or equal to the corresponding string of $N_{x_j}[]$;
(vii) construct an auxiliary graph H having vertex set $V(H) = \{x_1, x_2, \ldots, x_{k(v)}\}$ and edge set $E(H) = \emptyset$;
(viii) for each $i = 2, 3, \ldots, k(v)$ do in parallel
  $u \leftarrow A_v[i]$, $w \leftarrow A_v[i-1]$;
  if $N_u[] = N_w[]$ then add the edge uw to $E(H)$;
(ix) compute the connected components of the graph H, and let $C_v[]$ be the output array of length $k(v)$ such that $C_v[x_i] = C_v[x_j]$ iff x_i, x_j belong to the same connected component of the graph H;
copy the contents of $C_v[]$ to $S_v[]$.

(iv) The auxiliary array $A_v[]$ of length $k(v) \leqslant \deg(v)$ can be computed in $O(1)$ time using $O(\deg(v))$ processors on the EREW PRAM model.

(v) The vertex y belongs to $L_1(v)$ if and only if the vertices v, y are not adjacent in G. Since the set $L_2(v)$ contains $O(\deg(v))$ vertices and the adjacency in the graph G can be checked in $O(1)$ sequential time (using an $(n \times n)$-size array), the array $N_{x_i}[]$ can be computed in $O(1)$ time using $O(n \deg(v))$ processors on the CREW PRAM model.

(vi) It is well known that the sorting problem on n elements has an optimal solution which takes $O(\log n)$ time and requires $O(n)$ processors on the EREW PRAM model [2,17]. Moreover, we can check whether the contents of $N_{x_i}[]$ form a string which is lexicographically smaller or equal to that of $N_{x_j}[]$ in $O(\log n)$ using $O(n/\log n)$ processors on the EREW PRAM model. Thus, this step can be executed in $O(\log^2 n)$ time using $O(n \deg(v)/\log n)$ processors on the EREW PRAM model; recall that the array $N_{x_i}[]$ is of length n, $1 \leqslant i \leqslant k(v)$.

(vii) Since $k(v) \leqslant \deg(v)$, we can copy the vertices $x_1, x_2, \ldots, x_{k(v)}$ of $L_2(v)$ in the set $V(H)$ in $O(1)$ time with $O(\deg(v))$ processors on the EREW PRAM model.

(viii) It is easy to see that, for each value of i, the condition of the if-statement can be checked in $O(\log n)$ time using $O(n/\log n)$ processors on the EREW PRAM model. Thus,

the whole step can be executed in $O(\log n)$ time using $O(n\deg(v)/\log n)$ processors on the same model of computation.

(ix) By construction, the graph H has $O(\deg(v))$ vertices and less than $k(v)$, i.e., $O(\deg(v))$, edges. Thus, the connected components of the graph H can be computed in $O(\log n)$ time using a total of $O(\deg(v))$ processors on the EREW PRAM model [4] (for a graph G on n vertices and m edges, the parallel connectivity algorithm in [4] computes the connected components of G in $O(\log n)$ time using $O(n+m)$ processors on the EREW PRAM). Finally, copying $C_v[]$ in $S_v[]$ takes $O(1)$ time and $O(\deg(v))$ processors on the EREW PRAM model.

Complexity of substep 2.2. Since substep 2.2 is executed for each vertex $v \in V(G)$, it requires in total $O(\log^2 n)$ time with $O((\sum_v n\deg(v))/\log nr) = O(nm/\log n)$ processors on the CREW PRAM model.

In substep 2.3, we compute the $(2m \times n)$-size array $M[]$ such that $M[(v,x),y] = 1$ if and only if xvy is a P_3 of the graph G which participates in a P_4 in G. Since $L_2(v) \cap L_3(v) = \emptyset$ and $L_2(v), L_3(v) \subseteq N(v)$, substep 2.3 can be carried out in $O(1)$ time using $O(\deg^2(v))$ processors on the CREW PRAM model; it is assumed that the adjacency in G is checked in constant time using an $(n \times n)$-size array.

Complexity of substep 2.3. This substep is executed for each vertex $v \in V(G)$ and, thus, in total, it requires $O(1)$ time with $O(\sum_v \deg^2(v)r) = O(nm)$ processors on the CREW PRAM model.

Taking into consideration the time and processor complexity of each step of the procedure COMPUTE_P3s, we conclude that it is executed in $O(\log^2 n)$ time with $O(nm/\log n)$ processors on the CREW PRAM model. Thus, we have proved the following result.

Theorem 3.1. *Let G be a connected simple graph on n vertices and m edges. Procedure* COMPUTE_P3s *computes all the P_3s of the graph G that participate in a P_4 in $O(\log^2 n)$ time using $O(nm/\log n)$ processors on the CREW PRAM model.*

3.2. Computing and orienting the P_4-components of the graph G

The algorithm uses an auxiliary graph $\widehat{G}$ which serves as a constructive tool for computing the P_4-components of the graph G and for deciding whether each P_4-component admits a P_4-transitive orientation; if so, the algorithm produces such an orientation for each P_4-component.

The graph $\widehat{G}$ has $2m$ vertices and $O(nm)$ edges, and records all the P_3s which participate in P_4s in G. In particular,

$$V(\widehat{G}) = \{\hat{u}_{xy}, \hat{u}_{yx} \mid xy \in E(G)\}$$

and

$$E(\widehat{G}) = \{\hat{u}_{xy}\hat{u}_{zy}, \hat{u}_{yx}\hat{u}_{yz} \mid xyz \text{is a } P_3 \text{ in a } P_4 \text{ in } G\}.$$

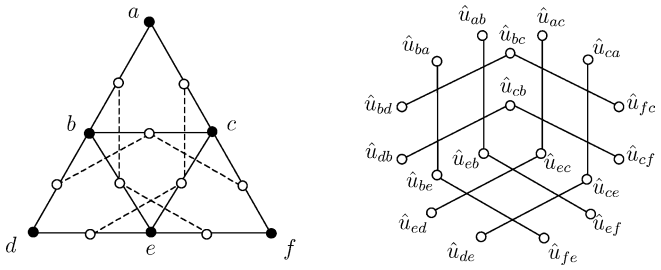

Fig. 4. If G is the pyramid, the graph $\widehat{G}$ has 6 connected components.

That is, $\widehat{G}$ has two vertices, say, $\hat{u}_{xy}$ and $\hat{u}_{yx}$, for each edge xy of G, and for each P_3 xyz participating in a P_4 in G, $\widehat{G}$ has an edge connecting the vertices $\hat{u}_{xy}$ and $\hat{u}_{zy}$ and another edge connecting the vertices $\hat{u}_{yx}$ and $\hat{u}_{yz}$. It is important to note the subscripts of the vertices incident upon an edge of $\widehat{G}$: the two edges corresponding to each P_3 in a P_4 of G connect vertices of $\widehat{G}$ whose subscripts correspond to the two transitive orientations of the P_3; in this way, from the two P_3s of a P_4 of G, $\widehat{G}$ will contain two pairs of incident edges which correspond to the two P_4-transitive orientations of the P_4. Figure 4 gives an example of the construction; it depicts the graph $\widehat{G}$ when the graph G is the pyramid.

The definition of the graph $\widehat{G}$ implies that $\widehat{G}$ has the following important properties:

(A1) The edges of G corresponding to the vertices of each connected component of $\widehat{G}$ span a P_4-component of G.

(A2) Each of the P_4-components of the graph G admits a P_4-transitive orientation if and only if for no pair of vertices $x, y \in V(G)$ both $\hat{u}_{xy}$ and $\hat{u}_{yx}$ belong to the same connected component of $\widehat{G}$.

(A3) If each of the P_4-components of the graph G admits a P_4-transitive orientation, then the connected components of $\widehat{G}$ can be partitioned into pairs of "twin" components, i.e., for each vertex $\hat{u}_{ab}$ belonging to the one component in such a pair, the vertex $\hat{u}_{ba}$ belongs to the other component in the pair. If we select one component from each such pair, then, for each edge xy of G, we will pick exactly one of the vertices $\hat{u}_{xy}$ and $\hat{u}_{yx}$; then, we obtain a P_4-transitive orientation of each P_4-component of G by orienting the edge in accordance with the ordering of the subscripts of the selected vertices, i.e., if the vertex $\hat{u}_{ab}$ is selected then the edge of G connecting a and b is oriented from a to b.

Property (A3) relies on the following lemma.

Lemma 3.1. *Let G be a simple graph and let $\widehat{G}$ be the corresponding graph obtained as described in this section. Additionally, let $\widehat{C}_i$ $(1 \leqslant i \leqslant p)$ be the connected components of $\widehat{G}$. If $\hat{u}_{ab} \in \widehat{C}_i$ and $\hat{u}_{ba} \in \widehat{C}_j$, then, for every vertex $\hat{u}_{st} \in \widehat{C}_i$, the vertex $\hat{u}_{ts}$ belongs to $\widehat{C}_j$.*

Proof. Since $\hat{u}_{ab}$ and $\hat{u}_{st}$ both belong to $\widehat{C}_i$, there exists a path from $\hat{u}_{ab}$ to $\hat{u}_{st}$. Then, by the construction of the graph $\widehat{G}$, there exists also a path from $\hat{u}_{ba}$ to $\hat{u}_{ts}$. Thus, $\hat{u}_{ts} \in \widehat{C}_j$. □

The following procedure takes advantage of the properties of $\widehat{G}$ in order to compute and P_4-transitively orient the P_4-components of the input graph G.

Procedure COMPUTE_TRO_P4C (*for computing and P_4-transitive orienting the P_4-components of the graph G*).

1. Construct a graph $\widehat{G}$ having vertex set $V(\widehat{G}) = \{\hat{u}_{xy}, \hat{u}_{yx} \mid xy \in E(G)\}$ and edge set $E(\widehat{G}) = \emptyset$.
2. For each P_3 xyz reported by Procedure COMPUTE_P3s, add the edges $\hat{u}_{xy}\hat{u}_{zy}$ and $\hat{u}_{yx}\hat{u}_{yz}$ to the set $E(\widehat{G})$, where by $\hat{u}_{ab}$ and $\hat{u}_{ba}$ we denote the vertices of $\widehat{G}$ corresponding to the edge ab of G.
3. Compute the connected components $\widehat{C}_1, \widehat{C}_2, \ldots, \widehat{C}_p$ of the graph $\widehat{G}$;
   let $\widehat{C}[]$ be the output array of length $2m$ such that $\widehat{C}[\hat{u}_{xy}] = \widehat{C}[\hat{u}_{zw}]$ iff $\hat{u}_{xy}, \hat{u}_{zw}$ belong to the same connected component; let $\hat{u}_{x_1y_1}, \hat{u}_{x_2y_2}, \ldots, \hat{u}_{x_py_p}$ be the representatives of the connected components $\widehat{C}_1, \widehat{C}_2, \ldots, \widehat{C}_p$ respectively.
4. For each representative $\hat{u}_{x_iy_i} \in \widehat{C}_i$ do in parallel
   if $\widehat{C}[\hat{u}_{x_iy_i}] = \widehat{C}[\hat{u}_{y_ix_i}]$, then $A[x_iy_i] \leftarrow 1$ {$A[]$ is an array of size $2m$ initialized to 0}.
5. Check if all the $2m$ entries of the array $A[]$ have values equal to 0;
   if this is not the case then the graph G is not a P_4-comparability graph; exit
   {*if the procedure does not exit here, then each P_4-component of the graph G admits a P_4-transitive orientation*}.
6. Select $k = p/2$ connected components from $\widehat{C}_1, \widehat{C}_2, \ldots, \widehat{C}_p$ as follows:
   for each representative $\hat{u}_{x_iy_i} \in \widehat{C}_i$, $1 \leqslant i \leqslant p$, do in parallel
   find the representative $\hat{u}_{x_jy_j}$ of the connected component $\widehat{C}_j$ that contains the vertex $\hat{u}_{y_ix_i}$;
   if the index number of $\hat{u}_{x_iy_i}$ is less than the index number of $\hat{u}_{x_jy_j}$
   then select the connected component $\widehat{C}_i$.
7. Let $C_1, C_2, \ldots, C_k$ be the connected components selected in step 6;
   for each vertex $\hat{u}_{xy} \in C_i$, $1 \leqslant i \leqslant k$, do in parallel
   orient the edge xy of G from x to y.

Observe that in step 6 exactly one component from each pair of "twin" connected components will be selected. If $E_1, E_2, \ldots, E_k$ are the edge sets corresponding to the vertex sets of the selected components $C_1, C_2, \ldots, C_k$ respectively, then the graph G has k P_4-components with edge sets $E_1, E_2, \ldots, E_k$.

Additionally, all the edges of G get oriented in step 7; for each edge xy of G, the graph $\widehat{G}$ contains two vertices, $\hat{u}_{xy}$ and $\hat{u}_{yx}$, exactly one of which belongs to one of the C_is, $1 \leqslant i \leqslant k$. The correctness of the algorithm follows from the stated properties of the graph $\widehat{G}$; the correctness of step 4 relies on Lemma 3.1, which implies that if a pair of vertices $\hat{u}_{xy}, \hat{u}_{yx}$ belong to the same connected component $\widehat{C}_i$, then for every vertex $\hat{u}_{ab} \in \widehat{C}_i$, the vertex $\hat{u}_{ba}$ belongs to $\widehat{C}_i$ as well.

3.2.1. Time and processor complexity

We now compute the time and processor complexities of each step of Procedure COMPUTE_TRO_P4C.

1. The $2m$ vertices of the graph $\widehat{G}$ can be easily computed from the adjacency list of the input graph G. Let $v_1, v_2, \ldots, v_n$ be the vertices of G. We use an array $P[]$ of length $2m$ for the implementation of the vertex set $V(\widehat{G})$: for each edge $v_i v_j$ of G, we set $P[k_{ij}] \leftarrow \hat{u}_{v_i v_j}$ and $P[k_{ji}] \leftarrow \hat{u}_{v_j v_i}$, where $k_{ij} = \sum_{\ell=1}^{i-1} \deg(v_\ell) + \text{rank}(v_i, v_j)$, $k_{ji} = \sum_{\ell=1}^{j-1} \deg(v_\ell) + \text{rank}(v_j, v_i)$, and $\text{rank}(v, u)$ denotes the rank of the vertex u in the adjacency list of the vertex v. Furthermore, we establish pointers from each of the two vertices $\hat{u}_{v_i v_j}$ and $\hat{u}_{v_j v_i}$ to the other. It is easy to see that this step can be completed in $O(\log n)$ time with $O(m/\log n)$ processors on the EREW PRAM model, using prefix sums and list ranking computations [2,17].

2. The procedure COMPUTE_P3s computes all the $P_3 s$ that participate in $P_4 s$ in the graph G, and stores this information in the array $M[]$ of size $(2m \times n)$ (see Section 3.1); recall that $M[(v, x), y] = 1$ if and only if xvy is a P_3 participating in a P_4 in G. Since the size of $M[]$ is $O(nm)$, the computation of the edge set $E(\widehat{G})$ of the graph $\widehat{G}$ can be completed in $O(1)$ time using $O(nm)$ processors on the EREW PRAM model.

3. The graph $\widehat{G}$ has $2m$ vertices and $O(nm)$ edges. Thus, the connected components $\widehat{C}_1, \widehat{C}_2, \ldots, \widehat{C}_p$ of the graph $\widehat{G}$ can be computed in $O(\log m)$ time using a total of $O(nm)$ processors on the EREW PRAM model [4]. Clearly, $p \leqslant 2m$.

4. Step 3 computes the connected components of the graph $\widehat{G}$, that is, an array $\widehat{C}[]$ of length $2m$ such that $\widehat{C}[\hat{u}_{xy}] = \widehat{C}[\hat{u}_{zw}]$ if and only if $\hat{u}_{xy}, \hat{u}_{zw}$ belong to the same connected component. Thus, thanks to the pointers between each pair of vertices of $\widehat{G}$ corresponding to the same edge of G, the array $A[]$ can be easily computed in $O(1)$ time using $O(m)$ processors on the EREW PRAM model.

5. The decision whether the array $A[]$ contains at least one entry with value equal to 1 can be done in $O(\log m)$ time with $O(m/\log m)$ processors on the EREW PRAM model (it can be done by simply finding the maximum element of $A[]$).

6. Step 5 guarantees that for no $x, y \in V(G)$ the vertices $\hat{u}_{xy}$ and $\hat{u}_{yx}$ both belong to the same connected component $\widehat{C}_i$, $(1 \leqslant i \leqslant p)$. Then, from property (A3) of the graph $\widehat{G}$, the p connected components $\widehat{C}_1, \ldots, \widehat{C}_p$ form $k = p/2$ pairs of "twin" components. In this step, we have to select one component from each such pair, obtaining a collection of components $C_1, C_2, \ldots, C_k$, which implies that if $\hat{u}_{xy} \in C_i$ then $\hat{u}_{yx} \notin C_j$, for $1 \leqslant j \leqslant k$. The vertices of the connected components $C_1, C_2, \ldots, C_k$ can be computed as follows:

Computation of the vertices of the connected components $C_1, C_2, \ldots, C_k$:

(i) for each representative $\hat{u}_{x_i y_i}$ of $\widehat{C}_i$, $1 \leqslant i \leqslant p$, do in parallel
  $\hat{u}_{x_j y_j} \leftarrow \widehat{C}[\hat{u}_{y_i x_i}]$ {*representative of component containing* $\hat{u}_{y_i x_i}$}
  if the index number of $\hat{u}_{x_i y_i}$ is greater than the index number of $\hat{u}_{x_j y_j}$
  then $\widehat{C}[\hat{u}_{x_i y_i}] \leftarrow 0$ {*mark the representative of* $\widehat{C}_i$};

(ii) for each vertex $\hat{u}_{xy} \in V(\widehat{G})$ do in parallel
  if $\widehat{C}[\hat{u}_{xy}] = 0$ or $\widehat{C}[\widehat{C}[\hat{u}_{xy}]] = 0$ {*the representative is marked*}
  then $D[\hat{u}_{xy}] \leftarrow (0, \hat{u}_{xy})$ else $D[\hat{u}_{xy}] \leftarrow (\widehat{C}[\hat{u}_{xy}], \hat{u}_{xy})$;

(iii) Sort the array $D[]$ and, then, delete the pairs $(0, \hat{u}_{xy})$;
the vertices of the selected connected component are in consecutive positions in $D[]$;
assign them to sets $C_1, C_2, \ldots, C_k$.

(i) In order to ensure that the body of the for-loop will be executed exactly once for each representative $\hat{u}_{x_i y_i}$, $1 \leqslant i \leqslant p$, we make a copy of the array $\widehat{C}[]$, sort it, and then execute the for-loop for each entry of the sorted array which differs from the entry preceding it in the array. The if-statement can be executed in $O(1)$ sequential time for each vertex $\hat{u}_{y_i x_i}$, $1 \leqslant i \leqslant p$. Since $p \leqslant 2m$ and n elements can be optimally sorted in $O(\log n)$ time with $O(n)$ processors on the EREW PRAM model [2,17], the whole step takes $O(\log m)$ time and uses $O(m)$ processors on the EREW PRAM model.

(ii) It is easy to see that both steps are executed in $O(1)$ time with $O(m)$ processors on the CREW PRAM model.

(iii) This step is executed in $O(\log m)$ time with $O(m)$ processors on the EREW PRAM model.

Thus, the above described procedure computes the vertices of the connected components $C_1, C_2, \ldots, C_k$ in $O(\log m)$ time using $O(m)$ processors on the EREW PRAM model.

7. The vertices of the connected components $C_1, C_2, \ldots, C_k$ computed in step 6 correspond to edges of the graph G. Thus, having computed the vertices of each connected component $C_1, C_2, \ldots, C_k$, the orientation assignment can be executed in $O(1)$ time using $O(m)$ processors on the EREW PRAM model.

Taking into consideration the time and processor complexity of each step of the procedure COMPUTE_TRO_P4C and the fact that $\log m = \Theta(\log n)$ because the graph G is connected, we conclude that:

Theorem 3.2. *Procedure* COMPUTE_TRO_P4C *runs in* $O(\log n)$ *time using* $O(nm)$ *processors on the CREW PRAM model.*

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

In the following, we will be processing the non-trivial P_4-components. These can be easily located as they contain at least 3 edges, whereas the trivial ones contain exactly one edge.

3.3. Combining the oriented non-trivial P_4-components

The procedure for combining the oriented non-trivial P_4-components relies on Lemma 2.5: if $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_\ell$ are the non-trivial P_4-components of a graph G ordered by increasing vertex number, then the edges of each P_4-component $\mathcal{C}_i$ which is of type (B) with respect to another P_4-component $\mathcal{C}_j$ with $j < i$ are oriented towards their endpoint which belongs to $V(\mathcal{C}_{\hat{\imath}})$, where $\hat{\imath} = \min\{j \mid \mathcal{C}_i$ is of type (B) with respect to $\mathcal{C}_j\}$. If the P_4-component $\mathcal{C}_i$ is not of type (B) with respect to any P_4-component $\mathcal{C}_j$ with $j < i$, then

$\mathcal{C}_i$ may be assigned any P_4-transitive orientation; in particular, the orientation assigned by Procedure COMPUTE_TRO_P4C in Section 3.2 will do. Lemma 2.5 implies that if none of the oriented (non-trivial) P_4-components contains a directed cycle, the resulting directed graph (spanned by their edges) will be acyclic. Conversely, if the resulting directed graph contains a directed cycle, then it must be the case that a P_4-component contains a directed cycle, implying that the input graph G is not a P_4-comparability graph.

The procedure also takes advantage of the following observation, which is a direct consequence of the definition of the P_4-components of type (B) and of the form of the P_4s of type (6):

Observation 3.1. *Let $\mathcal{C}$ and $\mathcal{C}'$ be two non-trivial P_4-components of a graph such that $\mathcal{C}'$ is of type* (B) *with respect to $\mathcal{C}$. Then, for any edge xy of $\mathcal{C}'$, exactly one of x, y belongs to $\mathcal{C}$.*

This observation has two very important consequences. First, it implies that a P_4-component $\mathcal{C}'$ is of type (B) with respect to a P_4-component $\mathcal{C}$ if and only if exactly one endpoint of each edge of $\mathcal{C}'$ belongs to $\mathcal{C}$; note that each edge of a P_4 of types (1)–(5) with respect to $\mathcal{C}$ has at most one endpoint in $\mathcal{C}$, while at least one edge of such a P_4 has none of its endpoints in $\mathcal{C}$. This yields an efficient way to determine whether a P_4-component is of type (B) with respect to another P_4-component. Second, if we are given a P_4-component $\mathcal{C}'$ and we are interested in finding the P_4-components $\mathcal{C}$ such that $\mathcal{C}'$ is of type (B) with respect to $\mathcal{C}$, then we only need to pick an arbitrary edge of $\mathcal{C}'$ and look at the P_4-components that contain the edges incident upon the endpoints of e; this is directly implied by Observation 3.1, since exactly one of the endpoints of the arbitrarily chosen edge of $\mathcal{C}'$ belong to any such P_4-component $\mathcal{C}$. This enables us to determine for each P_4-component of G the P_4-components with respect to which it is of type (B) by checking $O(n\ell) = O(nm)$ pairs of P_4-components instead of the $O(\ell^2) = O(m^2)$ pairs that the brute-force approach would necessitate.

In more detail, the procedure for combining the oriented non-trivial P_4-components works as follows:

Procedure TRO_ALL_P4C (*for combining the oriented non-trivial P_4-components*).

1. Sort the non-trivial P_4-components of the graph G in order of increasing vertex number; let them be $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_\ell$ in that order.
2. For each non-trivial P_4-component $\mathcal{C}_i$ $(1 \leqslant i \leqslant \ell)$, do in parallel
   2.1 select an arbitrary edge $x_i y_i$ of $\mathcal{C}_i$; initialize a set S_i to the set $\{i\}$;
   2.2 for each vertex z adjacent to x_i in G do in parallel
   find the P_4-component $\mathcal{C}_j$ to which the edge $x_i z$ belongs;
   if $j < i$ and exactly one endpoint of each edge of $\mathcal{C}_i$ belongs to $\mathcal{C}_j$
   then add j to the set S_i {*$\mathcal{C}_i$ is of type* (B) *w.r.t. $\mathcal{C}_j$ and $j < i$*};
   2.3 repeat substep 2.2 for y_i instead of x_i;
   2.4 $\hat{\imath} \leftarrow$ minimum element of S_i;
   if $\hat{\imath} \neq i$ then for each edge e of $\mathcal{C}_i$ do in parallel
   orient e towards its (exactly one) endpoint that belongs to $\mathcal{C}_{\hat{\imath}}$.

Note that, after substep 2.3, the set S_i contains i and the index numbers of all the non-trivial P_4-components $\mathcal{C}_j$ such that $j < i$ and $\mathcal{C}_i$ is of type (B) with respect to $\mathcal{C}_j$. Moreover, in substep 2.4, if $\hat{\imath} = i$, then the P_4-component $\mathcal{C}_i$ is not of type (B) with respect to any of the P_4-components $\mathcal{C}_j$ with $j < i$; in this case, nothing needs to be done and the P_4-component $\mathcal{C}_i$ retains the orientation assigned to it by Procedure COMPUTE_TRO_P4C. Then, the correctness of the procedure follows from Lemma 2.5, while the correctness of the orientation assignment follows from Observation 3.1.

3.3.1. Time and processor complexity

In order to be able to determine whether a vertex belongs to a non-trivial P_4-component, the procedure uses an auxiliary $(n \times \ell)$-size array $A[]$, which records for each vertex of the graph G the non-trivial P_4-components to which it belongs: if vertex v is a vertex of the P_4-component $\mathcal{C}_i$ $(1 \leqslant i \leqslant \ell)$, then the entry $A[v, i]$ is equal to 1, otherwise it is 0. Initializing all the entries of the array $A[]$ to 0 is done in $O(1)$ time using $O(n\ell) = O(nm)$ processors on the EREW PRAM model. We set the appropriate entries of $A[]$ to 1 as follows: for each vertex $\hat{u}_{xy} \in C_i$, we assign $A[x, i] \leftarrow 1$ and $A[y, i] \leftarrow 1$, where $C_1, C_2, \ldots C_k$ are the selected connected components of the graph $\widehat{G}$ of Section 3.2 (see steps 6 and 7 of Procedure COMPUTE_TRO_P4C). Assigning 1 to the appropriate entries of $A[]$ takes $O(1)$ time using $O(m)$ processors on the EREW PRAM.

Below we give a step-by-step analysis of the time and processor complexity of the procedure TRO_ALL_P4C. Recall that $\ell \leqslant m$.

1. It is well known that n elements can be sorted in $O(\log n)$ time with $O(n)$ processors on the EREW PRAM model [2,17]. Thus, this step is executed in $O(\log m)$ time with $O(m)$ processors on the EREW PRAM model.

2. Substep 2.1 can be carried out in $O(1)$ time on the EREW PRAM using $O(m)$ processors. In substep 2.2, finding $\mathcal{C}_j$ is done in $O(1)$ time using an array that indicates for each edge of G the P_4-component to which it belongs; since $O(\sum_{1 \leqslant i \leqslant \ell} \deg(x_i) r) = O(nm)$, this part of substep 2.2 can be carried out in $O(1)$ time using $O(nm)$ processors on the CREW PRAM.

The set S_i is implemented by means of an $(n \times |\mathcal{C}_i|)$-size array $B_i[]$ (note that the arrays $B_i[]$, $1 \leqslant i \leqslant \ell$, can all be placed in a single $(n \times m)$-array if the edges are re-indexed so that the edges of each P_4-component are assigned consecutive index numbers; this can be easily achieved by sorting in $O(\log m)$ time with $O(m)$ processors on the EREW PRAM). The entries of each array $B_i[]$ are initialized to i (in this way, invalid entries do not affect the subsequent minimum computations); the initialization of these arrays for all $i = 1, 2, \ldots, \ell$ takes $O(1)$ time with $O(nm)$ processors on the EREW PRAM. Then, for each vertex z adjacent to x_i and for each edge e of $\mathcal{C}_i$ we check if $j < i$ and if e has exactly one endpoint in $\mathcal{C}_j$, and, if this is true, we set the entry $B_i[z, e]$ equal to j; otherwise, we set it to 0. Next, for each row r of the array $B_i[]$, we compute the minimum over the entries in the row r; let it be p_r. Then, if $p_r = 0$ (meaning that an edge of $\mathcal{C}_i$ has failed the test) or $p_r = i$ (meaning that the vertex corresponding to the row r is not adjacent to x_i), we set the entry $B_i[r, 1]$ equal to i; otherwise (i.e., $p_r < i$ and $\mathcal{C}_i$ is of type (B) with respect to $\mathcal{C}_{p_r}$) we set $B_i[r, 1]$ equal to p_r. In this way, the minimum element of S_i among those contributed in substep 2.2 can be computed as the minimum over the entries $B_i[*, 1]$ in the first column of the array $B_i[]$. The above description implies that managing the sets S_i for all $i = 1, \ldots, \ell$

in substep 2.2 takes $O(\log n)$ time using $O(nm/\log n)$ processors on the EREW PRAM. In total, substep 2.2 runs in $O(\log n)$ time using $O(nm/\log n)$ processors on the CREW PRAM. In a similar fashion, substep 2.3 takes $O(\log n)$ time and $O(nm/\log n)$ processors on the CREW PRAM model.

As suggested, the computation of $\hat{\imath}$ for $\mathcal{C}_i$ in substep 2.4 is achieved by means of a minimum computation over the entries of two arrays of size n; for all $i = 1, \ldots, \ell$, this takes $O(\log n)$ time using $O(nm/\log n)$ processors on the EREW PRAM. Thanks to the array $A[]$, the rest of substep 2.4 is completed for all i in $O(1)$ time and $O(m)$ processors on the CREW PRAM model.

Thus, we have the following result.

Theorem 3.3. *Given a P_4-transitive orientation of each P_4-component of a connected simple graph G, Procedure* TRO_ALL_P4C *produces a P_4-transitive orientation of the graph $G\langle E_{\mathcal{C}}\rangle$ in $O(\log n)$ time using $O(nm/\log n)$ processors on the CREW PRAM model. The resulting orientation is acyclic if and only if the P_4-transitive orientation of each of the P_4-components is acyclic.*

3.4. Detecting directed cycles in the non-trivial P_4-components

Recall that if the input graph G is a P_4-comparability graph, then Procedure TRO_ALL_P4C produces an acyclic P_4-transitive orientation $G\langle\overrightarrow{E_{\mathcal{C}}}\rangle$ of the graph $G\langle E_{\mathcal{C}}\rangle$ spanned by the edges of the non-trivial P_4-components $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_\ell$ of G; note that $E_{\mathcal{C}} = E_1 \cup E_2 \cup \cdots \cup E_\ell$, where E_i is the set of edges of the P_4-component $\mathcal{C}_i$, $1 \leqslant i \leqslant \ell$.

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 has been checked during the execution of Procedure COMPUTE_TRO_P4C (see Section 3.2); if not, the procedure stops and reports 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 $G\langle\overrightarrow{E_i}\rangle$ of the non-trivial P_4-component $\mathcal{C}_i$, $1 \leqslant i \leqslant \ell$.

In order to do this, we use a procedure which orients the trivial edges whose endpoints are connected by a directed path in the graph $G\langle\overrightarrow{E_{\mathcal{C}}}\rangle$; we call these edges *forced*, because their orientation is constrained to match the direction of the path, otherwise a directed cycle is formed (see Fig. 5(a)). The procedure assigns to the forced trivial edges the appropriate

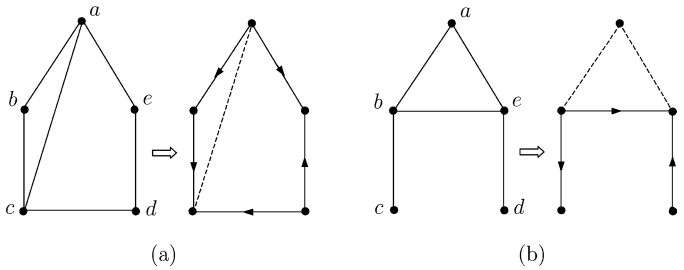

Fig. 5. (a) the edge ac is a forced trivial edge; (b) the edges ab and ae are trivial but they are not forced.

orientation by means of an iterative procedure, which keeps short-cutting directed paths (respectively cycles) until they become of length at most 2 (respectively 4). At each iteration, it also checks for the existence of directed triangles, i.e., C_3s, or C_4s which proves to be sufficient for ensuring the detection of directed cycles of any length in $G\langle\overrightarrow{E_C}\rangle$. The oriented forced trivial edges are added to the set $\overrightarrow{E_C}$ producing a set $\overrightarrow{E_{C,F}}$. The procedure uses an auxiliary array $R[]$ of size $n \times n$; for any two vertices x and y of G, $R[x, y]$ is set equal to 1 if a directed P_3 from x to y has been found in the current iteration.

Procedure DDC_P4C (*for detecting directed cycles in the P_4-components of G*).

1. $\overrightarrow{E'_C} \leftarrow \emptyset$; $Q \leftarrow \overrightarrow{E_C}$.
2. while $Q \neq \emptyset$ do
   - 2.1 $\overrightarrow{E'_C} \leftarrow \overrightarrow{E'_C} \cup Q$; $Q \leftarrow \emptyset$; initialize all the entries of array $R[]$ to 0;
   - 2.2 for every vertex x of G do in parallel
     for every vertex $z \neq x$ do in parallel
     for every vertex y adjacent to both x and z do in parallel
     if both $\overrightarrow{xy} \in \overrightarrow{E'_C}$ and $\overrightarrow{yz} \in \overrightarrow{E'_C}$
     then if $\overrightarrow{zx} \in \overrightarrow{E'_C}$
     then there exists a directed cycle; exit;
     else $R[x, z] \leftarrow 1$;
   - 2.3 for every pair of vertices x, z of G do in parallel
     - 2.3.1 if $R[x, z] = 1$ and $R[z, x] = 1$
       then there exists a directed cycle; exit;
     - 2.3.2 if $R[x, z] = 1$ and x, z are adjacent in G
       then if the edge xz belongs to $\overrightarrow{E'_C}$
       then if it is oriented from z to x
       then there exists a directed cycle; exit;
       else add $\overrightarrow{xz}$ to Q;
     - 2.3.3 if $R[z, x] = 1$ and x, z are adjacent in G
       then if the edge xz belongs to $\overrightarrow{E'_C}$
       then if it is oriented from x to z
       then there exists a directed cycle; exit;
       else add $\overrightarrow{zx}$ to Q;
   - 2.4 for every edge xy of G which does not belong to $\overrightarrow{E'_C}$ do in parallel
     for every vertex z of G adjacent to x do in parallel
     [2.4.1] if $\overrightarrow{xz} \in \overrightarrow{E'_C}$ and $R[z, y] = 1$ then add $\overrightarrow{xy}$ to Q;
     [2.4.2] if $\overrightarrow{zx} \in \overrightarrow{E'_C}$ and $R[y, z] = 1$ then add $\overrightarrow{yx}$ to Q;
   - 2.5 if there exists an edge xy of G such that both $\overrightarrow{xy}, \overrightarrow{yx} \in Q$
     then there exists a directed cycle; exit.

The cases handled by substeps 2.3.2, 2.4.1 and 2.4.2 are shown in Fig. 6. In each iteration, the procedure orients all the forced edges that short-cut directed P_3s and directed P_4s in $G\langle\overrightarrow{E_C}'\rangle$ and have not yet received an orientation. The case of an edge short-cutting a directed P_3 is handled in substep 2.3.2 or substep 2.3.3; because of substep 2.3.1, at most

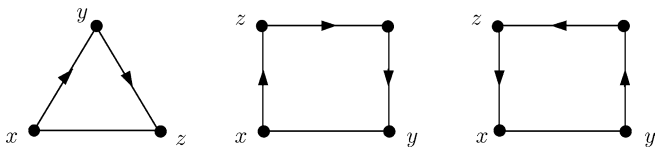

Fig. 6. The cases handled by substeps 2.3.2, 2.4.1 and 2.4.2 of Procedure DDC_P4C.

one of substeps 2.3.2 and 2.3.3 will be executed during each iteration. For an edge ad of G short-cutting a directed P_4 $abcd$ of $G\langle\overrightarrow{E_C}'\rangle$, both $R[a,c]$ and $R[b,d]$ will be set equal to 1 in substep 2.2. Then, no matter whether the edge connecting a and d is considered as the edge ad or da, it will be oriented from a to d in substeps 2.4.1 and 2.4.2 respectively.

Additionally, it is important to observe that, in each iteration of the while-loop of step 2, Procedure DDC_P4C detects all directed C_3s and directed C_4s: the former are detected in substep 2.2; the latter in substep 2.3.1 (for a directed C_4 $abcd$, both $R[a,c]$ and $R[c,a]$ are set to 1), which also implies that $\overrightarrow{yx} \notin \overrightarrow{E_C'}$ in substep 2.4.1, and that $\overrightarrow{xy} \notin \overrightarrow{E_C'}$ in substep 2.4.2.

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

Lemma 3.2. *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 while-loop of Procedure* DDC_P4C *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 $k \geqslant 4$. 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 their 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 while-loop (see substeps 2.3.2, 2.3.3 and 2.4.1, 2.4.2). 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 $\lceil 2k/3 \rceil \leqslant (2k+2)/3 \leqslant 5k/6$ for $k \geqslant 4$, the lemma follows. □

The correctness of the procedure is established by the following two lemmas.

Lemma 3.3. *Upon completion, Procedure* DDC_P4C *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 in $G\langle\overrightarrow{E_C'}\rangle$. For as long as the new chordless path has length at least equal to 4, new short-cutting directed edges will be added to $G\langle\overrightarrow{E_C'}\rangle$. When the length of the resulting chordless path eventually becomes equal to 2 or 3, then substeps 2.3.2–3 and 2.4.1–2 will assign to the edge xy an orientation from x to y. □

Lemma 3.4. *Procedure* DDC_P4C *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 procedure will shrink it as described in Lemma 3.2, eventually yielding a directed C_3 or a directed C_4, which will be detected by the procedure and the input graph will correctly be characterized as not being a P_4-comparability graph. On the other hand, it is not difficult to see that whenever the procedure reports the existence of a directed cycle, it has found two vertices, say, u and v, such that there exists a directed path from u to v and another form v to u. Since the edges of these paths either belonged to $G\langle\overrightarrow{E_C}\rangle$ or were oriented because there was a directed path in $G\langle\overrightarrow{E_C}\rangle$ leading from one of their endpoints to the other, it is clear that $G\langle\overrightarrow{E_C}\rangle$ contains a directed cycle, and thus the procedure responded correctly. □

3.4.1. Time and processor complexity

The sets $\overrightarrow{E'_C}$ and Q of oriented edges are maintained as arrays of size m and $2m$ respectively: for the set $\overrightarrow{E'_C}$, we maintain one entry per edge of G where it is recorded whether the corresponding edge belongs to the set and the assigned orientation; for the set Q, we maintain two entries per edge corresponding to the two opposite orientations of the edge. In this way, testing the membership of a (directed) edge in either of these sets or finding its orientation can be done in constant time.

Moreover, Lemma 3.2 implies that the number of iterations of the while-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.

1. It is easy to see that this step can be executed in $O(1)$ time with $O(m)$ processors on the EREW PRAM model.

2. Let us consider a single iteration of the while-loop. Then, substep 2.1 takes constant time using $O(m)$ processors on the EREW PRAM model in light of the way the sets $\overrightarrow{E'_C}$ and Q are maintained. substep 2.5 can also be executed in constant time using $O(m)$ processors on the EREW PRAM model.

In substep 2.2, for each pair of vertices x, z of the graph G, we have $O(\deg(x)+\deg(z))$ processors, each associated with a neighbor of x or z; each such processor checks whether the associated vertex is adjacent to both x and z and whether the conditions in the if-statement are satisfied, and produces a 1 if it finds that the value of $R[x, z]$ should be set to 1, and a 0 otherwise. Then, $R[x, z]$ is indeed set to 1 if the maximum of the produced values is 1. With the help of an adjacency matrix of G, the computation of each processor takes $O(1)$ time on the CREW PRAM; next, the computation of the maximum can be done in $O(\log n)$ time using $O((\deg(x)+\deg(z))/\log nr)$ processors on the EREW PRAM. Thus, the execution of substep 2.2 can be completed in $O(\log n)$ time using $O((\sum_x \sum_z(\deg(x)+\deg(z))r)/\log nr) = O(nm/\log n)$ processors on the CREW PRAM model.

Substep 2.3 takes $O(1)$ time using $O(n^2)$ processors on the CREW PRAM: for each pair of vertices x, z, the computation takes $O(1)$ time. It is important to observe that no concurrent write occurs as the processor associated with the pair x, z is the only one to add $\overrightarrow{xz}$ or $\overrightarrow{zx}$ to Q.

Similarly to substep 2.2, substep 2.4 takes $O(\log n)$ times using $O(nm/\log n)$ processors on the CREW PRAM: for each edge xy of G not in $\overrightarrow{E'_C}$, we have $O(\deg(x))$ processors, each associated with a neighbor of x; each processor checks if the conditions in the if-statements of substeps 2.4.1 and 2.4.2 hold and produces a 1 if the condition in substep 2.4.1 is true, -1 if the condition in substep 2.4.2 is true, and 0 otherwise (note that since $R[y,z]$ and $R[z,y]$ cannot both be equal to 1 due to substep 2.3.1, at most one of these two conditions will hold). Then, the minimum and maximum of the produced values are computed: if the maximum is 1, then $\overrightarrow{xy}$ is added to Q; if the minimum is -1, then $\overrightarrow{yx}$ is added to Q. The computation of each processor takes $O(1)$ time on the CREW PRAM, while the computation of the minimum and maximum can be done in $O(\log n)$ time using $O(\deg(x)/\log n)$ processors on the EREW PRAM. This implies that, for all such edges xy, substep 2.4 takes $O(\log n)$ time using $O(\sum_{xy} \deg(x)/\log nr) = O(nm/\log n)$ processors on the CREW PRAM model.

Taking into account the time and processor complexities of substeps 2.1–2.5, we have that the execution of one iteration of the while-loop can be completed in $O(\log n)$ using $O(nm/\log n)$ 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 $O(nm/\log n)$ processors on the CREW PRAM model.

Thus, we have the following result.

Theorem 3.4. *It can be decided whether the P_4-transitive orientations of 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 $O(nm/\log n)$ processors on the CREW PRAM model.*

Our results from Section 3 imply the following corollary.

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 $O(nm/\log n)$ 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 Procedure REC_P4G on each one of them. The connected components can be computed on the EREW PRAM model in $O(\log n)$ time using $O(n+m)$ processors [4], or in $O(\log^2 n)$ time using $O((n+m)/\log n)$ processors (see also [19]). If n_i and m_i are the numbers of vertices and edges of the ith connected component, then its processing requires $O(\log^2 n_i)$ time using $O(n_i m_i/\log n_i)$ processors on the CREW PRAM model (Corollary 3.2). If $m_i \geqslant \sqrt{n}$, then $\log m_i = \Theta(\log n)$; the connectivity of the component implies that $\log n_i = \Theta(\log m_i)$, and consequently $O(n_i m_i/\log n_i) = O(n_i m_i/\log m_i) = O(n_i m_i/\log n)$. Moreover, $\log^2 n_i = O(\log^2 n)$. If $m_i < \sqrt{n}$, 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 becomes $O(n_i m_i/\log n)$, since $\log n_i \leqslant \log n$.

In summary, no matter how small or large m_i is, we can process the ith connected component in $O(\log^2 n)$ time using $O(n_i m_i / \log n)$ processors on the CREW PRAM model. Thus, we can process all the connected components in $O(\log^2 n)$ time using a total of $O(nm/\log n)$ 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 $O(nm/\log n)$ 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 $G\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$ produced by the recognition algorithm of the previous section and assigns the final orientations to the remaining edges of the input graph G—these are the trivial edges which are not forced (see Fig. 5(b))—so that no directed cycle is formed in $G\langle\overrightarrow{E(G)}\rangle$. (Note that these edges have received an arbitrary orientation in step 2 of the recognition algorithm REC_P4G, which we ignore.) The algorithm relies on the following two lemmas.

Lemma 4.1. *In the directed graph $G\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$, 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 which has not yet received an orientation and is incident upon at least one of the following pairs of vertices: u and b, u and v, a and v. But, in each of these three cases, this edge is forced and must have been assigned an orientation by Procedure DDC_P4C. In fact, this orientation should be from u to b, from u to v, and from a to v respectively (Lemma 3.3), which 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 transitive closure $G^*\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$ of the acyclic directed graph $G\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$. 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 $G^*\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$ is equal to the number of vertices of $G\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$ 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 $G\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$ 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 $G\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$, $a \notin P(a)$ whereas $a \in P(b)$. □

Our orientation algorithm involves the following algorithmic steps.

Procedure ATRO_P4G (*for the acyclic P_4-transitive orientation of the graph G*).

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 and prints the corresponding diagnostic message; otherwise, the recognition procedure computes the directed graph $G\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$.
2. Compute the transitive closure $G^*\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$ of the graph $G\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$.
3. Compute the indegree(v) of each vertex v of the graph $G^*\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$; set the indegree of every vertex of G which is not a vertex of $G^*\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$ equal to 0.
4. For each (trivial) edge xy of G not in $G\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$, do in parallel
   if indegree$(x) <$ indegree(y), then $\overrightarrow{xy}$;
   if indegree$(x) >$ indegree(y), then $\overleftarrow{xy}$;
   if indegree$(x) =$ indegree(y)
   then if the index number of x is less than the index number of y
   then $\overrightarrow{xy}$ else $\overleftarrow{xy}$.

The index number of a vertex referred to in step 4 is a number which distinguishes one vertex of G from another; it may be the index of the entry that the vertex occupies in the array of vertices of G. Additionally, note that, in light of Lemma 4.1, the computation of the transitive closure $G^*\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$ in step 2 can be done by adding a directed edge $\overrightarrow{uv}$ for each directed P_3 $\overrightarrow{uwv}$. Therefore, for each directed edge $\overrightarrow{ab}$ of $G\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$, we go through each vertex c of G adjacent to b and check whether the path abc is a directed P_3 of $G\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$; if it is so, then the directed edge $\overrightarrow{ac}$ needs to be added. To avoid concurrent writes, for each vertex v we use an array $H_v[x, vy]$ of size $n \times \deg(v)$, where x and y are vertices of G and y is adjacent to v. 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 $H_a[c, ab]$ to 1; the entry $H_a[c, ab]$ uniquely corresponds to the path abc. In the end, the transitive closure is produced by adding to $G\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$ the edges $\overrightarrow{uv}$ for which there is a 1 in the subarray $H_u[v, *]$; this can be found in $O(\log n)$ time with $O(nm/\log n)$ processors using standard interval prefix computations on the EREW PRAM model [2] (note that the total size of all the H arrays is $\sum_v(\deg(v))^2 = O(nm)$).

The correctness of the algorithm follows from the following lemma.

Lemma 4.3. *For a P_4-comparability graph G, Procedure* ATRO_P4G *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 $G\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$ is P_4-transitive and therefore so

is the resulting orientation; note that during the execution of Procedure ATRO_P4G only trivial edges (in particular, those that are not forced) receive an orientation. Additionally, since step 1 of Procedure ATRO_P4G is completed successfully, then the orientation of $G\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$ 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. Consider an ordering of the vertices of G from left to right in increasing order of their indegree in $G^*\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$ (see step 3) and, in case of ties, in increasing order of their index number. Then, according to Lemma 4.2, all the edges of $G^*\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$, and hence of $G\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$, are directed from left to right. Additionally, step 4 guarantees that the edges of G not in $G\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$ are also directed from left to right. Therefore, no directed cycle exists. □

4.1. Time and processor complexity

We assume that the input graph G is connected; thus, $n = O(m)$. Step 1 takes $O(\log^2 n)$ time using a total of $O(nm/\log n)$ processors on the CREW PRAM model (Theorem 3.5). As described above, the process of computing the transitive closure $G^*\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$ is based on the processing of all pairs of an edge and a vertex; thus, it can be carried out in $O(\log n)$ time using a total of $O(nm/\log n)$ processors on the CREW PRAM model. The graph $G^*\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$ has n vertices and $O(n^2)$ edges; therefore, the computation of the indegrees of its vertices can be done in $O(\log n)$ time with $O(n^2/\log n) = O(nm/\log n)$ processors on the EREW PRAM model. Obviously, step 4 takes $O(1)$ time and requires $O(m)$ processors on the CREW PRAM model. If the graph G is disconnected, then we compute its connected components and we apply Procedure ATRO_P4G on each one of them; similarly to the analysis in Section 3.5, we obtain that for a disconnected graph on n vertices and m edges, Procedure ATRO_P4G takes $O(\log^2 n)$ time using $O(nm/\log n)$ 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 $O(nm/\log n)$ processors on the CREW PRAM model.*

It is worth noting that only step 1 of Procedure ATRO_P4G necessitates $O(\log^2 n)$ time; the remaining steps can all be executed in $O(\log n)$ time. Hence, the invention of an algorithm which computes the directed graph $G\langle\overrightarrow{E_{\mathcal{C},F}}\rangle$ in $O(\log n)$ time would imply the computation of the acyclic P_4-transitive orientation of a graph in $O(\log n)$ time.

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 on them. Both algorithms run in $O(\log^2 n)$ time using a total of $O(nm/\log n)$ processors on the CREW PRAM model, where n and m are the number of vertices and edges of the input graph. Our algorithms rely on certain algorithmic and structural properties of the P_4-components of

a graph. To the best of our knowledge, they are the first parallel algorithms for problems involving P_4-comparability graphs; they are cost-efficient, since the currently best sequential algorithms for these problems take $O(nm)$ time [28].

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 cost-efficient algorithms are needed for other well-known and 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.

References

[1] G.S. Adhar, S. Peng, Parallel algorithms for cographs and parity graphs with applications, J. Algorithms 11 (1990) 252–284.
[2] S.G. Akl, Parallel Computation: Models and Methods, Prentice Hall, 1997.
[3] S.R. Arikati, U.N. Peled, A polynomial algorithm for the parity path problem on perfectly orderable graphs, Discrete Appl. Math. 65 (1996) 5–20.
[4] K.W. Chong, Y. Han, T.W. Lam, Concurrent threads and optimal parallel minimum spanning trees algorithm, J. Assoc. Comput. Mach. 48 (2) (2001) 297–323.
[5] V. Chvátal, Perfectly ordered graphs, Ann. Discrete Math. 21 (1984) 63–65.
[6] D.G. Corneil, H. Lerches, L. Burlingham, Complement reducible graphs, Discrete Appl. Math. 3 (1981) 163–174.
[7] D.G. Corneil, Y. Perl, 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. Algorithms 36 (2000) 205–240.
[10] S. de Agostino, R. Petreschi, Parallel recognition algorithms for graphs with restricted neighbourhoods, Internat. J. Found. Comput. Sci. 1 (1990) 123–130.
[11] C.M.H. de Figueiredo, J. Gimbel, C.P. Mello, 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, 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, New York, 1980.
[15] C.T. Hoàng, B.A. Reed, Some classes of perfectly orderable graphs, J. Graph Theory 13 (1989) 445–463.
[16] C.T. Hoàng, B.A. Reed, P_4-comparability graphs, Discrete Math. 74 (1989) 173–200.
[17] J. JáJá, An Introduction to Parallel Algorithms, Addison–Wesley, 1992.
[18] P.N. Klein, Efficient parallel algorithms for chordal graphs, in: Proc. 29th IEEE Symposium on the Foundations of Computer Science (FOCS '89), 1989, pp. 150–161.
[19] C.P. Kruskal, L. Rudolph, M. Snir, Efficient parallel algorithms for graph problems, Algorithmica 5 (1990) 43–64.
[20] R. Lin, S. Olariu, An NC recognition algorithm for cographs, J. Parallel Distrib. Comput. 13 (1991) 76–90.
[21] R.M. McConnell, J. Spinrad, Linear-time modular decomposition and efficient transitive orientation of comparability graphs, in: Proc. 5th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA '94), 1994, pp. 536–545.

[22] R.M. McConnell, J. Spinrad, Linear-time transitive orientation, in: Proc. 8th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA '97), 1997, pp. 19–25.
[23] M. Middendorf, F. Pfeiffer, On the complexity of recognizing perfectly orderable graphs, Discrete Math. 80 (1990) 327–333.
[24] M. Morvan, L. Viennot, Parallel comparability graph recognition and modular decomposition, in: Proc. 13th Symposium on Theoretical Aspects of Computer Science (STACS '96), in: Lecture Notes in Comput. Sci., vol. 1046, Springer-Verlag, 1996, pp. 169–180.
[25] S.D. Nikolopoulos, Constant-time parallel recognition of split graphs, Inform. Process. Lett. 54 (1995) 1–8.
[26] S.D. Nikolopoulos, Coloring permutation graphs in parallel, Discrete Appl. Math. 120 (2002) 165–195.
[27] S.D. Nikolopoulos, L. Palios, Recognition and orientation algorithms for P_4-comparability graphs, in: Proc. 12th International Symposium on Algorithms and Computation (ISAAC '01), in: Lecture Notes in Comput. Sci., vol. 2223, Springer-Verlag, 2001, pp. 320–331.
[28] S.D. Nikolopoulos, L. Palios, Algorithms for P_4-comparability graph recognition and acyclic P_4-transitive orientation, Algorithmica, in press.
[29] T. Raschle, K. Simon, On the P_4-components of graphs, Discrete Appl. Math. 100 (2000) 215–235.
[30] J. Reif (Ed.), Synthesis of Parallel Algorithms, Kaufmann, San Mateo, CA, 1993.
[31] J. Spinrad, On comparability and permutation graphs, SIAM J. Comput. 14 (1985) 658–670.