RAIRO-Theor. Inf. Appl. 44 (2010) 293–311
DOI: 10.1051/ita/2010016

Available online at:
www.rairo-ita.org

PARALLEL ALGORITHMS FOR MAXIMAL CLIQUES IN CIRCLE GRAPHS AND UNRESTRICTED DEPTH SEARCH [*]

E.N. Cáceres[1], S.W. Song[2] and J.L. Szwarcfiter[3]

Abstract. We present parallel algorithms on the BSP/CGM model, with p processors, to count and generate all the maximal cliques of a circle graph with n vertices and m edges. To count the number of all the maximal cliques, without actually generating them, our algorithm requires $O(\log p)$ communication rounds with $O(nm/p)$ local computation time. We also present an algorithm to generate the first maximal clique in $O(\log p)$ communication rounds with $O(nm/p)$ local computation, and to generate each one of the subsequent maximal cliques this algorithm requires $O(\log p)$ communication rounds with $O(m/p)$ local computation. The maximal cliques generation algorithm is based on generating all maximal paths in a directed acyclic graph, and we present an algorithm for this problem that uses $O(\log p)$ communication rounds with $O(m/p)$ local computation for each maximal path. We also show that the presented algorithms can be extended to the CREW PRAM model.

Mathematics Subject Classification. 68W10, 05C85, 05C69.

Keywords and phrases. BSP/CGM algorithm, PRAM algorithm, circle graph, maximal clique, unrestricted depth search.

[*] *Partially supported by CNPq Proc. Nos. 55.0895/07-8, 30.5362/06-2, 30.1652/09-0, 62.0171/06-5, FUNDECT 41/100.115/2006 and FAPERJ.*

[1] Universidade Federal de Mato Grosso do Sul, Faculdade de Computação, 79070-900 Campo Grande, MS, Brazil; edson@facom.ufms.br

[2] Universidade de São Paulo, Instituto de Matemática e Estatística, 05508-900 São Paulo, SP, Brazil; visiting professor of Universidade Federal do ABC; song@ime.usp.br

[3] Universidade Federal do Rio de Janeiro, Instituto de Matemática, Núcleo de Computação Eletrônica and COPPE, 21.945-970 Rio de Janeiro, RJ, Brazil; jayme@nce.ufrj.br

1. Introduction

In this paper we consider the enumeration problem for maximal cliques in circle graphs. By an enumeration problem, we mean counting and/or generating the set of all solutions to a given problem. While counting the number of solutions gives only one answer, namely an integer count, the generation of all solutions actually involves the listing of all the solutions. In this paper we deal with both problems, that is, we count the number of maximal cliques in a circle graph, and also generate all the maximal cliques. We present parallel algorithms for these problems under the BSP/CGM and CREW PRAM models. In our approach, we first present parallel algorithms, again under both the BSP/CGM and CREW PRAM models, to perform an *unrestricted depth search* in a directed acyclic graph D, to generate all maximal paths in D. In summary, in this paper we deal with three problems: counting the number maximal cliques in a given circle graph, generating all the maximal cliques in the circle graph, and generating all the maximal paths in a directed acyclic graph. In our work we deal only with simple paths.

We summarize the main results as follows[1].

- To count the number of maximal cliques in a circle graph of n nodes and m edges, we present a BSP/CGM parallel algorithm that requires $O(nm/p)$ local computation time and $O(\log p)$ communication rounds, where p is the number of processors. We also present an $O(\log^2 n)$ time CREW PRAM algorithm with $nM(n)$ processors, where $M(n)$ is the number of processors to compute matrix product on a CREW PRAM.
- To generate all the maximal cliques of a circle graph, the proposed BSP/CGM algorithm requires $O(\log p)$ communication rounds and $O(nm/p)$ local computation to generate the first maximal clique and $O(\log p)$ communication rounds and $O(m/p)$ local computation to generate each of the subsequent $\alpha - 1$ maximal cliques, where α is the number of maximal cliques. The time complexity of the proposed CREW PRAM algorithm is $O(\log^2 n)$ time with n^3 processors for the first maximal clique and $O(\log n)$ time with m/p processors for the subsequent maximal cliques.
- To generate all the maximal paths of the problem of unrestricted depth search in a directed acyclic graph, the proposed BSP/CGM algorithm requires $O(\log p)$ communication rounds with $O(m/p)$ local computation for each one of the β maximal paths. The CREW PRAM algorithm uses $O(\log n)$ time with m processors for each one of the maximal paths.

Observe that the number of maximal cliques α can be large compared to n. In fact, Moon and Moser [23] described the family of graphs having the maximum (exponential) number of maximal cliques, among all graphs. It happens that all these graphs are circle graphs. Therefore in some cases, it can be very inefficient to compute α by generating all maximal cliques and counting them. This is a motivation for describing an efficient parallel algorithm for computing such number. To our knowledge the algorithms proposed in this paper are the first known

[1] The summary also appear in Table 5 in the conclusion section.

parallel algorithms under the BSP/CGM and PRAM computing models for the counting and generation of all the maximal cliques in a circle graph. Part of the presented results appeared in conference extended abstracts [4,5]. This paper is an extended version with complete proofs, and contains new results for the CREW PRAM model.

This paper is organized as follows. In Section 2 we describe previous works and give motivation. In Section 3 we present the computation model and some definitions and terminology. In Section 4 we describe sequential and BSP/CGM algorithms for the unrestricted depth search of a connected digraph. In Section 5 we present the BSP/CGM algorithm to generate all maximal cliques of a circle graph. In Section 6 we extend the results to the CREW PRAM model. We conclude in Section 7.

2. Motivation and previous works

Circle graphs are a special kind of intersection graphs, to be shown shortly in the next section. Recognition of circle graphs has been an open problem for many years until the mid-eighties when several researchers discovered sequential polynomial time algorithms [3,13,15,24,27]. A number of NP-complete problems for general graphs, including the maximum weighted independent set problem and the maximum weighted clique problem, can be solved in polynomial time for circle graphs [16,27].

The generation of all the maximal cliques of a graph is a core problem in gene expression networks analysis, *cis* regulatory motif finding, and the study of quantitative trait loci for high-throughput molecular phenotypes [34]. There are many previous sequential and parallel solutions to generate all the maximal cliques of a graph. Sequential algorithms for generating all maximal cliques are presented by Makino and Uno [22] and by Tsukiyama *et al.* [30], the latter algorithm uses $O(nm\alpha)$ time. Chiba and Nishizeki [8] present results for this problem for graphs of bounded arboricity. Dahlhaus and Karpinski [9] present a PRAM algorithm for computing all maximal cliques in a general graph. Their algorithm takes $O(\log^3(n\alpha))$ parallel time and uses $\alpha^6 n^2$ processors. Wang and Chang [32] present a PRAM algorithm for finding all the maximal cliques of an interval graph. Klein [21], Naor *et al.* [25], Ho and Lee [18] present parallel algorithms for this problem for Chordal Graphs. Distributed algorithms for this problem are presented by Jennings and Motyckova [19] for network graphs and by Protti *et al.* [26] for general graphs. An approximation algorithm is presented by Gupta *et al.* [17] for unit disk graphs. To count and generate all the maximal cliques of a circle graph, Szwarcfiter and Barroso [28] present sequential algorithms that takes $O(nm)$ to count the maximal cliques and $O(n(m + \alpha))$ to generate all the maximal cliques. Our PRAM results for circle graphs have better bounds compared with the result for general graphs [9]. The BSP/CGM algorithms proposed in this paper require a number of communication rounds that are independent of the input size.

The proposed parallel algorithms to generate all the maximal cliques of a circle graph are based on an algorithm to perform an *unrestricted depth search* in a directed acyclic graph D. The unrestricted search in a graph generates all the maximal paths starting from a given vertex r called the root of the search. Given a graph $D = (V, A)$ and a vertex $r \in V$, the problem of generating a maximal path consists of finding a simple path P starting at r such that P cannot be extended without finding a vertex that already belongs to P. It has been shown that the algorithm for computing a maximal path with a greedy strategy does not have an efficient PRAM parallelization [2].

Unrestricted depth search has been shown to be an important technique for the design of algorithms. It has been used to generate all the maximal cliques in a circle graph [28]. The sequential algorithm for unrestricted depth search has $O(m + \beta n)$ time complexity, where β is the number of maximal paths.

On the other hand, the problem of parallel restricted depth search does not appear to have a simple solution as in the sequential case. At the moment the only known solutions in NC (PRAM Model) are for particular kinds of graphs [14,33]. In general, it is known that lexicographic depth search is *P-complete* [20].

Notice the difference between previous works and the results presented in this paper. Previous works either present sequential algorithms for the enumeration problem of maximal cliques of circle graphs, or parallel PRAM algorithms for this same problem but for general graphs. Thus this paper presents for the first time parallel BSP/CGM and PRAM algorithms for the enumeration problem of maximal cliques in circle graphs. Our approach is based on the parallelization of the *unrestricted depth search* in a directed acyclic graph.

3. Computational model, notation and terminology

We consider a version of the *Bulk Synchronous Parallel* (BSP) model [31] referred to as the *Coarse-Grained Multicomputer* (BSP/CGM) model [10]. A BSP/CGM consists of a set of p processors $P_1, \ldots, P_p$ with $O(N/p)$ local memory per processor and each processor is connected by a router that can send messages in a point-to-point fashion (or shared memory). A BSP/CGM algorithm consists of alternating local computation and global communication rounds separated by a barrier synchronization. The BSP/CGM model uses only two parameters: the input size N and the number of processors p. In a computing round, each processor runs a sequential algorithm to process its data locally. A communication round consists of sending and receiving messages, in such a way that each processor sends at most $O(N/p)$ data and receives at most $O(N/p)$ data. We require that all information sent from a given processor to another processor in one communication round be packed into one long message, thereby minimizing the message overhead.

In the BSP/CGM model, the communication cost is modeled by the number of communication rounds which we wish to minimize. In a good BSP/CGM algorithm the number of communication rounds does not depend on the input size

N. The ideal algorithm requires a constant number of communication rounds. If this is not possible, we attempt to get an algorithm for which this number is independent on N but depends on p. This is the case of the present paper.

The BSP/CGM model has the advantage of producing results that are close to the actual performance of commercially available parallel machines. Some algorithms for computational geometry and graph problems require a constant number or $O(\log p)$ communication rounds [11]. The BSP/CGM model is particularly suitable for current parallel machines in which the global computation speed is considerably greater than the global communication speed.

We also designed our algorithms for the standard CREW PRAM model [20].

This paper deals with an enumeration problem, in our case, all maximal cliques of a circle graph. In a general graph the number of maximal cliques could be exponential and computing such number is a $\#P$-complete problem. As for circle graphs, the number of maximal cliques might still be exponential. However, as mentioned in the paper, there is a polynomial time sequential algorithm to compute this number [28].

Since we deal with an enumeration problem, let us state two premises, employed by many enumeration algorithms, and adopted in this paper.

(1) The time performance of the algorithm is measured by its delay complexity, that is, the worst case time between the enumeration of any two consecutive maximal cliques, also considering the time needed for the enumeration of the first and the last maximal cliques of the collection.
(2) By enumerating a maximal clique C, we mean finding the set of vertices which forms C, and making it available in a memory space. The time needed for explicitly listing C, for instance, is not taken into account. This is the way in which most algorithms for enumeration problems are usually handled.

The following definition and notation will be used in this paper.

Consider a digraph (directed graph) $D = (V, A)$. For each vertex $v \in V$ we define the *successor* of v, denoted $\mathrm{suc}[v]$, as a fixed element of the adjacency list of v. Let $e = (u, v) \in A$, we define the *successor* of e as the edge $(v, \mathrm{suc}[v])$. A *walk* in D is a sequence of disjoint edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{q-1}, v_q)$. A *kl-walk* is a walk P in D, with initial edge (k, l) and such that every edge of P, except (k, l), is the successor of another edge of P.

For an undirected graph $G = (V, E)$, we consider each edge (u, v) as two distinct directed edges (u, v) and (v, u).

Since the successor of each vertex is fixed, all the edges on a kl-walk entering a vertex v have the same successor $(v, \mathrm{suc}[v])$. We will prove that a kl-walk can be a simple path, a cycle or a path together with a cycle. In the case where $D = (V, A)$ is an acyclic digraph, the kl-walks are formed by simple paths.

Let $\mathscr{F}$ be a family of nonempty sets, the *intersection graph* of $\mathscr{F}$ is obtained by representing each set in $\mathscr{F}$ by a vertex and connecting two vertices by an edge if and only if their corresponding sets intersect [16]. *Circle graphs* are intersection graphs where $\mathscr{F}$ is a a family of chords of a circle. In other words, consider a

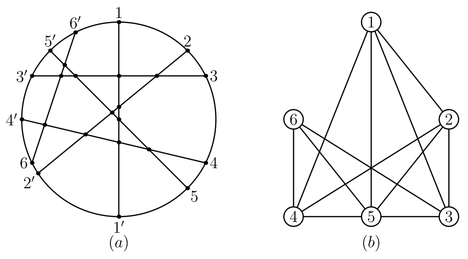

FIGURE 1. (a) Family of chords and (b) the corresponding circle graph $G = (V, E)$.

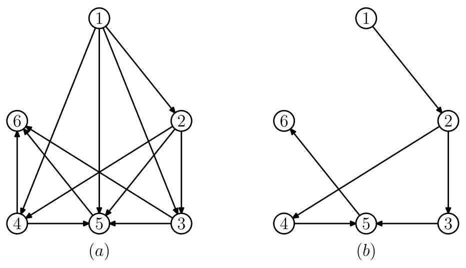

FIGURE 2. (a) A digraph $\vec{G}$ and (b) the transitive reduction $\vec{G}_R$.

family of n chords on a circle C, numbered as $1-1', 2-2', \ldots, n-n'$. Assume that any two chords do not share a same endpoint. In the corresponding circle graph, each chord corresponds to a vertex and there is an edge (u, v) if chord u intersects chord v. See Figure 1.

A *circular sequence* S of G is the sequence of the $2n$ distinct endpoints of the chords on circle C, by traversing C in a chosen direction, starting at a given point in C. Denote by $S_1(v)$ and $S_2(v)$, respectively, the first and second instances or occurrences of the chord corresponding to $v \in V$ in the sequence S. Denote by $S_i(v) < S_j(w)$ (for $i, j = 1, 2$) when $S_i(v)$ precedes $S_j(w)$ in S. We have $S_1(v) < S_2(v)$.

Let $G = (V, E)$ be a circle graph. A circular sequence S of G induces an orientation on the edges of G, so that G can be transformed into an acyclic digraph (V, A), called an S_1-*orientation* $\vec{G}$ of G. Observe that $\vec{G}$ is an orientation in which any directed edge $(v, w) \in A$ satisfies $S_1(v) < S_1(w)$. Figure 2a shows the acyclic digraph $\vec{G} = (V, A)$.

Let $\vec{G}$ denote an acyclic orientation of G, $N_v^+(\vec{G})$ and $N_v^-(\vec{G})$ are the subsets of vertices that leave and enter v, respectively. For $v, w \in V$, v is an *ancestor* of w in $\vec{G}$ if the directed graph contains a path $v - w$. In this case, w is a *descendant* of v. Denote by $N_v^*(\vec{G})$ the set of descendants of v. If $w \in N_v^*(\vec{G})$ and $v \neq w$, then v is a *proper ancestor* of w and w a *proper descendant* of v. When $(v, w), (w, z) \in A$ implies $(v, z) \in A$, $\vec{G}$ is denominated a *transitive digraph* with respect to edges.

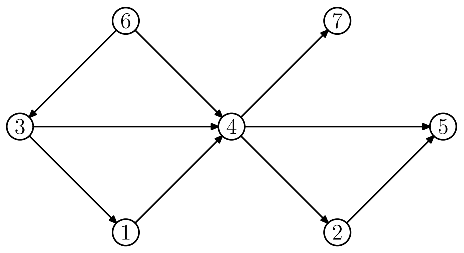

FIGURE 3. A locally transitive digraph $D = (V, A)$.

The *transitive reduction* $\vec{G}_R$ is the subgraph of $\vec{G}$ formed by the edges that are not implied by transitivity (see Fig. 2b).

Let $G = (V, E)$ be an undirected graph, $|V| > 1$ and $\vec{G}$ an acyclic orientation of G. Let $v, w \in V$, we denote by $Z(v, w) \subset V$ the subset of vertices that are simultaneously descendants of v and ancestors of w in $\vec{G}$. An edge $(v, w) \in A$ *induces local transitivity* when $\vec{G}(Z(v, w))$ is a transitive digraph. Since $\vec{G}(Z(v, w))$ is a transitive digraph the vertices of any path from v to w induce a clique in G. Furthermore, (v, w) *induces maximal local transitivity* when there does not exist $(v', w') \in A$ different from (v, w) such that v' is simultaneously an ancestor of v and w' a descendant of w in $\vec{G}$. In this case (v, w) is called a *maximal edge*. The orientation $\vec{G}$ is *locally transitive* when each of its edges induces local transitivity. Figure 3 shows an example of a locally transitive orientation.

Based on the following theorem, one can use locally transitive orientations to find maximal cliques.

Theorem 3.1. *Let $G = (V, E)$ be a graph, $\vec{G}$ a locally transitive orientation of G and $\vec{G}_R$ the transitive reduction of $\vec{G}$. Then there exists a one-to-one correspondence between the maximal cliques of G and paths $v - w$ in $\vec{G}_R$, for all maximal edges $(v, w) \in A$.*

The proof can be found in [28].

4. UNRESTRICTED SEARCH

Depth search and breadth search in a graph $G = (V, E)$ starting from a given vertex $v \in V$ determines a tree that corresponds to the execution of the respective search method. In these search methods, each edge $e \in E$ is traversed a *constant* number of times. We use the term *restricted* to identify a search method with such a characteristic. On the other hand, we define *unrestricted search* in a graph $G = (V, E)$ from a given vertex as a systematic process of traversing G in such a way that each edge is visited a *finite* number of times. The action taken in each visit depends on the particular application. The only restriction is that the process must terminate.

The (classical) sequential unrestricted depth search algorithm of a connected graph $G = (V, E)$ is a variation of the (restricted) depth search algorithm [29]. It is given in Algorithm 1. The algorithm starts with a given vertex v chosen as root and uses a stack Q.

Algorithm 1 `Unrestricted Search`(v)

Input: A connected graph $G = (V, E)$ and $v \in V$, the root of the search.
Output: All maximal paths of G starting at v.

```
1: Mark $v$
2: Push $v$ onto stack $Q$
3: for each vertex $w$ adjacent to $v$ do
4:    if $w$ is not in $Q$ then
5:       visit edge $(v, w)$
6:       Unrestricted Search($w$)
7:    end if
8: end for
9: Pop $v$ from $Q$
10: Unmark $v$
```

To design a parallel algorithm for unrestricted search, we must avoid the problems that occur when the path reaches a cycle in the graph. Furthermore, we need to know in which parts of the graph the path is located at each step of the algorithm. For this purpose, we use a decomposition of the graph called kl-walk, to be seen later.

Unrestricted search applies to both undirected and directed graphs. In this paper we use unrestricted search in an acyclic orientation of G, which is an acyclic digraph. Thus in the following we will describe the parallel algorithms for unrestricted search for acyclic digraphs.

We now describe Algorithm 2, the parallel BSP/CGM unrestricted search algorithm. We initially decompose the digraph $D = (V, A)$ into a set of kl-walks, using the definition of the successors of vertices and edges of D. The parallel algorithm initializes the successor of each vertex $v \in V$ as the first element of its adjacency list. Then it explores the edges of the kl-walk such that $k = r$, where r is the root of the search and $l = \text{suc}[r]$. Let $\{(v_0, v_1), (v_1, v_2), \ldots, (v_{t-1}, v_t)\}$ ($r = v_0$ and $l = v_1$) be the visited edges of the kl-walk. If v_t is different from all v_i, $0 \le i \le t-1$ (v_t is a sink), then $P = \{k, l, v_2, \ldots, v_{t-1}, v_t\}$ is a maximal path. If $v_t \in$ the kl-path, but it is the last element in the v_{t-1} adjacency list, then $P = \{k, l, v_2, \ldots, v_{t-1}\}$ is a maximal path. Otherwise, $P = \{k, l, v_2, \ldots, v_{t-1}\}$ is a simple path and let $adj[v_{t-1}] = \{u_0, u_1, \ldots, u_p\}$ be the adjacency list of v_{t-1}, where i is the index of the successor of v_{t-1} in its adjacency list ($u_i = v_t$). Now we try to extend the simple path P. First we change the successor of v_{t-1} to $\text{suc}[v_{t-1}] = u_{i+1}$. Every time we change the successors of any vertex, we have a new decomposition of the digraph into a set of kl-walks. Then we do the previous analysis to $P \cup W$, where W is the set of vertices of the kl-walk such that $k = v_{t-1}$ and $l = \text{suc}[v_{t-1}]$ until we find a maximal path starting at r.

Once we have found a maximal simple path $P = \{v_0, \cdots, v_p\}$ for $r = v_0$ on the kl-walk, it means that either v_p is a sink or $\mathrm{suc}[v_p] \in P$ and it is the last element in the v_p adjacency list. We can obtain a new maximal simple path, if it exists, as follows. Determine the last vertex $v_i \in P$, $i < p$ that has some vertex in its adjacency list that has not been visited yet. The successor of each vertex $v_j \notin \{v_0, \cdots, v_i\}$ is modified to be the first element of the adjacency list of v_j, and the successor of v_i is altered to be the element of the adjacency list of v_i immediately following $\mathrm{suc}[v_i]$. The successors of the vertices $\{v_0, \cdots, v_{i-1}\}$ remain unaltered. Using the previous analysis, we try to find a maximal path in $P \cup W$, where $P = \{v_0, \cdots, v_i\}$ and W are the vertices of the kl-walk in the new decomposition, with $k = v_i$ and $l = \mathrm{suc}[v_i]$. The remaining maximal simple paths, if they exist, are computed analogously.

The vertices w_i that are not reachable from the root vertex r are not included in the search.

Lemma 4.1. *Let $D = (V, A)$ be a digraph. Then a kl-walk is:*

(1) *a simple path;*
(2) *a cycle; or*
(3) *a path followed by a cycle.*

Proof. Let $P = \{v_0, \ldots, v_i\}$ be a simple path on the kl-walk, $k = v_0$ and $l = v_1$. Try to extend the path P. If v_i is a sink then it has no successors and P is a simple path. Otherwise, let $w = \mathrm{suc}[v_i]$. If $w \in P$, we obtain a cycle and since the successors are fixed, $\mathrm{suc}[w] \in P$. If $w \neq v_0$, then the kl-walk is a simple path followed by a cycle. If $w \notin P$, extend P by adding w to it and repeat the same reasoning. □

Corollary 4.2. *Let $D = (V, A)$ be an acyclic digraph. Then a kl-walk is a simple path.*

Lemma 4.3. *Consider the input of Algorithm 2, a weakly connected digraph $D = (V, A)$. Let $v \in V$ be a vertex included i times in the array MP. Denote by MP_i the configuration of MP that precedes $MP[j] = v$, after the inclusion of v in MP for the ith time. Then we have necessarily:*

(1) *The configuration of MP, when v is excluded for the ith time from MP, is also equal to MP_i, and*
(2) *$i \neq j \Rightarrow MP_i \neq MP_j$.*

Proof. The array MP simulates a stack in Algorithm 2. A vertex v is only included in MP if it is unmarked. Therefore the configuration of MP when v is excluded for the ith time is equal to MP_i. A vertex v will only be re-explored, in case there exists a path from a vertex u that precedes v in MP, and the vertices of the path do not belong to MP. Therefore if $i \neq j$ then $MP_i \neq MP_j$. □

The above lemma ensures the termination of the algorithm, because if a vertex v is inserted in MP more than once, then necessarily the configuration in MP, below v, is different in each case. Since there exists a finite number of ways to arrange these configurations, the algorithm necessarily reaches the end. This

Algorithm 2 `Parallel Unrestricted Search`(r)

```
Input: (1) A digraph $D = (V, A)$ given by its adjacency lists; (2) $r \in V$, the root
    of the search.
Output: All maximal paths of $D$ starting at $r$.
1: $MP \leftarrow \emptyset$
2: Set all the vertices $v \in V$ as unmarked
3: Decompose the digraph $D$ into a set of $kl$-walks
4: $k \leftarrow r$
5: repeat
6:    Determine a simple path $P = \{v_0, v_1, \dots, v_p\}$ on the $kl$-walk
7:    Mark all the vertices $\in P$
8:    $MP \leftarrow MP \cup P$
9:    if $adj[v_p] = \emptyset$ OR $suc[v_p]$ is the last element in $adj[v_p]$ then
10:      $MP$ is maximal
11:   end if
12:   Verify the existence of a vertex $v_i \in MP$ such that $v_i$ is the most distant
      vertex from $r$ that has in its adjacency list a vertex not visited yet
13:   if there exists such a vertex $v_i$ then
14:      $MP \leftarrow \{v_0, \cdots, v_{i-1}\}$
15:      suc$[v_i] \leftarrow$ next element in its adjacency list
16:      Unmark $v \notin MP$
17:      Alter the successors of the vertices $v_j \notin MP$ to the first one in each
         adjacency list
18:      $k \leftarrow v_i$
19:   end if
20: until there is no such vertex $v_i \in MP$
```

property is based strongly on the fact that a marked vertex cannot be re-explored. The re-exploration of a marked vertex can cause the appearance of cycles.

Theorem 4.4. *Let $D = (V, A)$ be a weakly connected digraph. Any path MP computed by Algorithm 2 starting from the root vertex r in a kl-walk with $k = r$ is maximal.*

Proof. The exploration of a kl-walk ends when a sink or a cycle is detected. If the last vertex in the kl-walk is a sink, then MP cannot be extended and it is a maximal path. When a cycle is detected, we remove the last vertex of the kl-path and store all the remaining vertices in a simple path P. Then we add P to MP. If the successor of the last vertex in MP is the last vertex in its adjacency list, then MP is maximal. Otherwise we change the successor of the last vertex in MP to the next element on its adjacency list and repeat the above procedure until MP is maximal.

Once a maximal path is found, Step 12 of Algorithm 2 computes the most distant vertex v_i from r in MP that possesses an unmarked element in its adjacency list. If such a vertex exists, we remove from MP all vertices starting at v_i and

Step 15 alters the successors of all vertices that do not belong to MP to the first element in each adjacency list.

Since the successors changed, we have a new decomposition of D into a different set of kl-walks. We repeat Steps 5 to 11 of Algorithm 2 to the new decomposition fo D. This is repeated while there is a vertex v_i that satisfies Step 12. □

Theorem 4.5. *Let $D = (V, A)$ be a weakly connected digraph. All maximal paths of D starting at the root vertex r will be found by Algorithm 2 at least once.*

Proof. Once a maximal path MP starting at r is found, Step 4 of Algorithm 2 determines the vertex v_i in P that is most distant from r. Since MP is a maximal path, v_i is not the last vertex in MP. All the vertices after v_i in MP are removed from MP. Also all the adjacencies of the vertices that do not belong to MP point to the first element in their respective adjacency lists. Finally the current successor of v_i points to the next element in the adjacency list (with respect the previous MP). After this, the graph is decomposed into a new set of kl-walks, with $k = r$.

The path MP obtained in the new kl-walk is different from the previous one, because the vertex to be included in MP after v_i is different from the previous vertex in MP.

Since we explore all the possible adjacencies of each path MP, all the simple maximal paths are computed. □

Theorem 4.6. *Let $D = (V, A)$ be a weakly connected digraph. No maximal path of D starting at the root vertex r will be found by Algorithm 2 more than once.*

Proof. By Lemma 4.3, if a vertex v is included in a path more than once, then the configuration in MP of the vertices that precede v is different, each time v is included in a path. Therefore the algorithm does not compute the same path more than once. □

Theorem 4.7. *Algorithm 2 obtains an unrestricted search of the digraph $D = (V, A)$ with n vertices and m edges using $O(\beta\gamma \log p)$ communication rounds with $O(m/p)$ local computation, where β is the number of maximal paths, γ the number of cycles found in D and p is the number of processors.*

Proof. First we rank the adjacency lists of all $v \in V$ and define the successor of v to be the first element of the adjacency list of v.

We start with the root r and the defined successors. If a cycle is found in the kl-walk, we have to detect the cycle and remove all duplicate vertices. Using list ranking, this can be done in $O(\log p)$ rounds [11]. After this, we check if the kl-walk is maximal. If the last vertex of the kl-walk has any unmarked adjacent vertex, we change the successor of v_i to the next unmarked vertex in its adjacency list and add this new path to the kl-walk. If we find any cycle in this path, we proceed as above. At the end, after γ_i cycles (γ_i can be zero), we have a maximal path of D.

After this, we mark the last vertex of the path and change the successor (if there is one) of the tail of this path and compute another maximal path. Otherwise, we can backtrack on this path and visit a vertex that has not been visited before and apply the same procedure.

With $\gamma = \max\{\gamma_1, \ldots, \gamma_\beta\}$, the number of communication rounds will be $O(\beta\, \gamma \log p)$. □

Corollary 4.8. *The unrestricted search in an acyclic digraph $D = (V, A)$ with n vertices and m edges can be done in $O(\beta \log p)$ communication rounds with $O(m/p)$ local computation, where β is the number of maximal paths found in D.*

Using Algorithm 2 for directed acyclic graphs we will generate in parallel all the maximal cliques of a given circle graph. We will show that we can also efficiently count the number of maximal cliques of the circle graph, without actually generating them.

5. Generating all maximal cliques

In this section we present Algorithm 3 to generate all the maximal cliques of a given circle graph G. An S_1-orientation $\vec{G}$ of G can be easily obtained through its circular sequence. We thus assume that an S_1-orientation is given as input.

Algorithm 3 `All Maximal Cliques`(G)

```
Input: An $S_1$-orientation $\vec{G}$ of a given circle graph $G$.
Output: All maximal cliques of $G$.
1: Construct the transitive reduction $\vec{G}_R$
2: Find all maximal edges of $\vec{G}$
3: For each maximal edge $(v, w) \in A$, find all paths $v - w$ in $\vec{G}_R$ (each path
   defines a maximal clique)
```

We now describe Algorithm 3, the BSP/CGM all the maximal cliques generation algorithm.

Line 1 computes the transitive reduction $\vec{G}_R$ of $\vec{G}$. This is done by using the transitive closure algorithm [1,6] with p processors using $O(\log p)$ communication rounds with $O(nm/p)$ local computation time.

Line 2 determines all the maximal edges of $\vec{G}$. Based on [28], observe that if $\vec{G}$ is locally transitive, then $(v, w) \in A$ is a maximal edge if and only if $N_v^+(\vec{G}) \cap N_w^+(\vec{G}) = N_v^-(\vec{G}) \cap N_w^-(\vec{G}) = \emptyset$. The adjacency lists $N_v^-(\vec{G})$ for all the vertices of $\vec{G}$ can be easily computed with a constant number of communication rounds with $O(m/p)$ local computation, by using a sort algorithm [7,12], and partitioning the edge lists into sublists.

In line 3 we first compute for each $v \in V$ the subsets $W(v)$ formed by vertices w such that (v, w) is a maximal edge. This can be done using a constant number of communication rounds with $O(m/p)$ local computation time. We then construct the subgraph $\vec{H}$ of $\vec{G}_R$ induced by the vertices that are simultaneously descendants of v and ancestors of any $w \in W(v)$ (Fig. 4). This can be done by computing

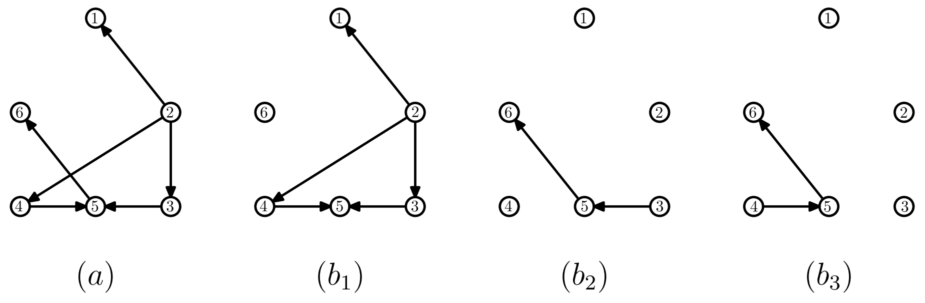

FIGURE 4. (a) A digraph $\vec{G}_R$ and (b) the digraphs $\vec{H}$.

the transitive closure [1,6] of $\vec{G}_R$ and through the intersection of the vertices that leave v with those that enter each of $w \in W(v)$. The subgraph $\vec{H}$ can be computed using $O(\log p)$ communication rounds with $O(nm/p)$ local computation time. The paths $v - w$ in $\vec{G}_R$ taken from a certain vertex v are exactly the source-sink paths in $\vec{H}$. Observe that digraph $\vec{H}$ is acyclic. Using Algorithm 2 we determine each one of the maximal paths of $\vec{H}$ using $O(\log p)$ communication rounds with $O(m/p)$ local computation time. The source-sink paths of $\vec{H}$ so obtained form the maximal cliques of G. The first maximal clique is obtained in $O(\log p)$ communication rounds and $O(nm/p)$ local computation, and each of the remaining $\alpha - 1$ maximal cliques is obtained in $O(\log p)$ communication rounds and $O(m/p)$ local computation.

Theorem 5.1. *Algorithm 3 generates all the maximal cliques of a circle graph correctly, on a BSP/CGM with p processors, using $O(\log p)$ communication rounds and $O(nm/p)$ local computation to generate the first maximal clique and $O(\log p)$ communication rounds and $O(m/p)$ local computation to generate each of the subsequent $\alpha - 1$ maximal cliques, where α is the number of maximal cliques.*

5.1. Computing the number of maximal cliques

The sequential algorithm for computing the number of maximal cliques in a circle graph presented in [28] first labels all the sink vertices of $\vec{H}$ with 1. When all the adjacent vertices of a vertex v_i have a label, v_i is labeled with the sum of the labels of all its adjacent vertices. This procedure is repeated until all the source vertices have been labeled. This approach is inherently sequential and would give rise to an inefficient parallel solution with $O(n)$ communication rounds in the worst case.

Algorithm 4 computes the number of maximal cliques using p processors. We use the transitive closure algorithm of [1,6], adapted to compute the lengths of the longest paths between vertices: instead of testing if there is a path (i, k), (k, j), for each k, we compute the longest path connecting vertices i and j.

Algorithm 4 `Number of Cliques`(G)

```
Input: An $S_1$-orientation $\vec{G}$ of a given circle graph $G$.
Output: The number of maximal cliques of $G$.
1: Construct the transitive reduction $\vec{G}_R$
2: Find all maximal edges of $\vec{G}$
3: for each maximal edge $(v, w) \in A$ do
4:     Find all paths $v - w$ in $\vec{G}_R$;
5:     Construct the graph $\vec{H}$
6:     Using the transitive closure algorithm [1,6] we compute the matrix $M_d = \{d_{ij}\}$ of the longest paths in $\vec{H}$: $d_{ij} \leftarrow \max\{d_{ij}, d_{ik} + d_{kj}\}$
7: end for
```

Theorem 5.2. *Algorithm 4 correctly computes the total number of maximal cliques in a circle graph in $O(\log p)$ communication rounds with $O(nm/p)$ local computation.*

Proof. Applying Algorithm 4 on graph $\vec{H}$, each $d_{ij} \in M_d$, where v_i is a source vertex and v_j is a sink vertex in $\vec{H}$ represents the existence of a maximal clique of size $d_{ij} + 1$. These numbers are added together to get the number of maximal cliques of the circle graph G. □

6. PRAM algorithms

We now show how the previous algorithms can be implemented on a CREW PRAM.

The implementation of the Parallel Unrestricted Search Algorithm (Algorithm 2) uses basic CREW PRAM techniques, as sum, list ranking, pointer jumping and list (edges and vertices) manipulation. Each path can be computed in $O(\log n)$ parallel time using $m/\log n$ processors. Therefore, the complexity of Algorithm 2 is $O(\beta \log n)$ parallel time with $m/\log n$ processors, where β is the number of maximal paths starting at root r and ending at a sink vertex of an acyclic digraph.

Now we describe how to implement the main steps of the All Maximal Cliques Algorithm (Algorithm 3) on a CREW PRAM. Line 1 (the transitive reduction $\vec{G}_R$ of $\vec{G}$) can be implemented as follows: we start by computing the adjacency lists $N_v^-(\vec{G})$ for all the vertices of $\vec{G}$. This can be done by partitioning the edge lists into sublists, one for each vertex v. This partition can be obtained in $O(\log n)$ parallel time with $M(n)$ processors on a CREW PRAM, where $M(n)$ is the number of processors needed to multiply two $n \times n$ matrices. We use the array of lists ADJ to store the adjacency lists. Each $ADJ[v_i]$ contains the list of vertices v_j that arrive in v_i. Using pointer jumping or recursive doubling, we remove from each adjacency list $ADJ[v_i]$ all the vertices that are in adjacency lists $ADJ[v_j]$, for all $v_j \in ADJ[v_i]$. This can be done as in Algorithm 5. The adjacency lists $N_{v_i}^-(\vec{G}_R)$

are given by $ADJ[v_i]$. Each one contains at most n vertices. In each step we compress each adjacency list in order to remove repeated vertices.

Algorithm 5 `Transitive Reduction(`G`)`

```
Input: An $S_1$-orientation $\vec{G}$ of a given circle graph $G$.
Output: The transitive reduction $\vec{G_R}$.
 1: for all $v_i \in V$ in parallel do
 2:     $ADJ[v_i] \leftarrow N^-_{v_i}(\vec{G})$
 3:     $BADJ[v_i] \leftarrow \cup_{v_j \in ADJ[v_i]} ADJ[v_j]$
 4: end for
 5: for $j = 1$ to $\lceil \log n \rceil$ do
 6:     for all $v_i \in V$ in parallel do
 7:         $BADJ[v_i] \leftarrow BADJ[v_i] \cup \{\cup_{v_j \in BADJ[v_i]} BADJ[v_j]\ \}$
 8:     end for
 9: end for
10: for all $v_i \in V$ in parallel do
11:     $ADJ[v_i] \leftarrow ADJ[v_i] - BADJ[v_i]$
12: end for
```

Lemma 6.1. *Graph $\vec{G_R}$ obtained by Algorithm 5 is the transitive reduction of $\vec{G}$.*

Proof. At each iteration we compute for each vertex v the set $BADJ[v]$ formed by the union of the adjacency lists of the vertices that arrive at v. The set $BADJ[v]$ contains the vertices for which there exists at least one path of length $\geq 2^i$ to v. At each iteration i, for each vertex v, we remove those vertices that belong to $BADJ[v]$ from the adjacency lists of the vertices that arrive at v. After $\lceil \log n \rceil$ iterations, all the vertices that are ancestors of v belong to $BADJ[v]$. Thus the adjacency lists will not contain vertices that generate transitive edges. □

Lines 6 to 8 of Algorithm 5 are repeated $\log n$ times. Thus line 1 of Algorithm 3 uses $O(\log^2 n)$ parallel time with n^3 processors on a CREW PRAM.

In line 2 of Algorithm 3 we determine all the maximal edges of $\vec{G}$. We observe that if $\vec{G}$ is locally transitive, then $(v, w) \in A$ is a maximal edge if and only if $N^+_v(\vec{G}) \cap N^+_w(\vec{G}) = N^-_v(\vec{G}) \cap N^-_w(\vec{G}) = \emptyset$. We can either assume the adjacency lists as part of the input or easily compute them.

Line 2 of Algorithm 3 can be computed in $O(\log n)$ parallel time with m processors on a CREW PRAM.

Now we describe how line 3 of Algorithm 3 is implemented. First compute for each $v \in V$ the subsets $W(v)$ formed by vertices w such that (v, w) is a maximal edge. We construct now the subgraph $\vec{H}$ of $\vec{G_R}$ induced by the vertices that are simultaneously descendants of v and ancestors of any $w \in W(v)$. This can be done by determining the transitive closure [20] $\vec{G_R}$ and through the intersection of the vertices that leave v with those that enter each of $w \in W(v)$. The paths $v - w$ in $\vec{G_R}$ taken from a certain vertex v are exactly the source-sink paths in $\vec{H}$. These paths can be obtained through the parallel unrestricted depth search algorithm of Section 4. Graph $\vec{H}$ can be constructed in $O(\log^2 n)$ parallel time with $M(n)$ processors on a CREW PRAM, where $M(n)$ is the number of processors needed to multiply two $n \times n$ matrices. Then we do an unrestricted search on $\vec{H}$. Observe that digraph $\vec{H}$ is acyclic. Using Algorithm 2 we determine the maximal paths of $\vec{H}$. Line 3 of Algorithm 3 can be implemented in $O(\log^2 n)$ parallel time with $M(n)$ processors on a CREW PRAM.

The source-sink paths of $\vec{H}$ obtained in line 3 of Algorithm 3 form the maximal cliques of G. Then Algorithm 3 generates the first maximal clique using $O(\log^2 n)$ parallel time with n^3 processors, and each of the remaining $\alpha-1$ maximal cliques is obtained in $O(\log n)$ parallel time with $m/\log n$ processors, where α is the number of maximal cliques.

Now we describe the PRAM algorithm (see Algorithm 6) for computing the number of maximal cliques in a circle graph.

Algorithm 6 `Number of Maximal Cliques`(G)

```
Input: An $S_1$-orientation $\vec{G}$ of a given circle graph $G$.
Output: The number of maximal cliques of $G$.
1: Apply Lines 1 to 3 of Algorithm 3
2: Construct the subgraphs $H$
3: Compute $A^k$, where $A$ is the adjacency matrix for each $H$
```

First we compute the number α_k of maximal cliques of size k and the total number of maximal cliques α. This algorithm uses $O(\alpha \log^2 n)$ parallel time, with n^3 and $nM(n)$ processors on a CREW PRAM, where $M(n)$ is the number of processors needed to multiply two $n \times n$ matrices on a CREW PRAM. To compute the number of maximal cliques on a CREW PRAM we use a different approach, since the sequential solution [28] uses a labeling strategy that seems inherently sequential and difficult to parallelize, as we already mentioned. On the BSP/CGM model we have solved the problem by adapting the transitive closure algorithm. Here we use a similar idea, by using matrix multiplications that can be done efficiently on the PRAM. The matrix A^k gives the number of maximal cliques of size $k+1$. The computation of each matrix A^k can be done in $O(\log n)$ parallel time with $M(n)$ processors on a CREW PRAM [20]. To compute the total number of maximal cliques, we have to compute in parallel each one of the A^k matrices. This can be done in $O(\log^2 n)$ parallel time with $nM(n)$ processors on a CREW PRAM [20]. By using the algorithms in [20], we can compute the number of maximal cliques of size k and the total number of maximal cliques of a circle graph in $O(\log^2 n)$ parallel time with n^3 and $nM(n)$ processors, respectively, on a CREW PRAM.

With the above observations, we can state the following results.

Theorem 6.2. *Algorithm 6 correctly computes the number of maximal cliques of size k and the total number of maximal cliques in a circle graph in $O(\log^2 n)$ parallel time with n^3 and $nM(n)$ processors, respectively, on a CREW PRAM.*

7. Conclusion

We have presented parallel algorithms, on the BSP/CGM and PRAM models, for generating all the maximal cliques in circle graphs. The proposed BSP/CGM algorithm requires $O(\log p)$ communication rounds and $O(nm/p)$ local computation to generate the first maximal clique and $O(\log p)$ communication rounds and $O(m/p)$ local computation to generate each of the subsequent $\alpha - 1$ maximal cliques, where α is the number of maximal cliques.

Algorithm	Generating all the max. cliques	Counting the number of max. cliques	Unrestricted depth search
Sequential$^{(1)}$	$O(nm\alpha)$ [30]	$\#P$-complete	$O(m + \beta n)$
Sequential$^{(2)}$	$O(n(m + \alpha))$ [28]	$O(nm)$ [28]	
CREW-PRAM$^{(1)}$	$O(\log^3(n\alpha))$ with $\alpha^6 n^2$ proc. [9]		$O(\beta\gamma \log n)$ with $m/\log n$ proc. (Sect. 6)
CREW-PRAM$^{(2)}$	$O(\log^2 n)$ with n^3 proc. 1st max. clique. $O(\log n)$ with $m/\log m$ proc. each next max. clique. (Sect. 6)	$O(\log^2 n)$ with $nM(n)$ proc. (Thm. 6.2)	
BSP/CGM$^{(1)}$			$O(\beta\gamma \log p)$ Comm. Rounds with $O(m/p)$ local comp. (Thm. 4.7)
BSP/CGM$^{(2)}$	$O(\log p)$ Comm. Rounds with $O(nm/p)$ local comp. 1st max. clique. $O(\log p)$ Comm. Rounds with $O(m/p)$ local comp. each next max. clique. (Thm. 5.1)	$O(\log p)$ Comm. Rounds with $O(nm/p)$ local comp. (Thm. 5.2)	

Figure 5. Main results for (1) general graphs and for (2) circle graphs, where n is the number of vertices, m the number of edges, α the number of maximal cliques, β the number of maximal paths, γ the number of cycles of the graph and $M(n)$ the number of processors to compute matrix product on a CREW PRAM.

To solve the above problem, we have described a BSP/CGM parallel unrestricted depth search as a technique for computing all maximal paths in an acyclic digraph. This technique requires $O(\beta \log p)$ communication rounds and $O(m/p)$ local computation, where β is the number of maximal paths. We show that both algorithms can also be implemented efficiently on a CREW PRAM using $O(\alpha \log^2 n)$ parallel time with n^3 processors (*maximal cliques*) and $O(\beta \log n)$ parallel time with m processors (*unrestricted depth search*).

Furthermore, we have also presented efficient parallel algorithms for counting the total number of maximal cliques α of a circle graph, without actually generating them.

Table in Figure 5 summarizes the main results.

To our knowledge the algorithms proposed in this paper are the first known parallel algorithm in the literature under the BSP/CGM and PRAM computing model for the counting and generation of all the maximal cliques in a circle graph.

References

[1] C.E.R. Alves, E.N. Cáceres, A.A. Castro Jr, S.W. Song and J.L. Szwarcfiter, Efficient Parallel Implementation of Transitive Closure of Digraphs, in *10th European PVM/MPI Users' Group Conference*. Edited by J. Dongarra, D. Laforenza and S. Orlando, Springer Verlag, Berlin (2003) 126–133.

[2] R. Anderson and E. Mayr, *Parallelism and Greedy Algorithms.* Technical Report STAN-CS-84-1003, Computer Science Department, Stanford University Bonn (1984).

[3] A. Bouchet, Reducing Prime Graphs and Recognizing Circle Graphs. *Combinatorica* **7** (1987) 243–254.

[4] E.N. Cáceres, S.W. Song and J.L. Szwarcfiter, A Coarse-Grained Parallel Algorithm for Maximal Cliques in Circle Graphs, in *Proc. The 2001 International Conference on Computational Science*. Springer Verlag, Berlin (2001) 638–647.

[5] E.N. Cáceres, S.W. Song and J.L. Szwarcfiter, A Parallel Unrestricted Depth Search Algorithm, in *Proc. 2001 International Conference on Parallel and Distributed Processing Techniques and Applications* (2001) 521–526.

[6] E.N. Cáceres, S.W. Song and J.L. Szwarcfiter, A Parallel Algorithm for Transitive Closure, in *Proc. 14th IASTED International Conference on Parallel and Distributed Computing and Systems*. IASTED, Zurich (2002) 116–118.

[7] A. Chan and F. Dehne, A Note on Coarse Grained Parallel Integer Sorting. *Parallel Process. Lett.* **9** (1999) 533–538.

[8] N. Chiba and T. Nishizeki, Arboricity and subgraphs listing algorithms. *SIAM J. Comput.* **14** (1985) 210–223.

[9] E. Dahlhaus and M. Karpinski, A Fast Parallel Algorithm for Computing all Maximal Cliques in a Graph and the Related Problems, in *Proc. Scandinavian Workshop on Algorithm Theory – SWAT* (1988) 139–144.

[10] F. Dehne (Ed.), Coarse grained parallel algorithms. *Algorithmica* **24** (1999) 173–426.

[11] F. Dehne, A. Ferreira, E. Cáceres, S.W. Song and A. Roncato, Efficient Parallel Graph Algorithms For Coarse Grained Multicomputers and BSP. *Algorithmica* **33** (2002) 183–200.

[12] A. Ferreira and N. Schabanel, A randomized BSP/CGM algorithm for the maximal independent set. *Parallel Process. Lett.* **9** (2000) 411–422.

[13] C.P. Gabor, W.L. Hsu and K.J. Supowit, Recognizing Circle Graphs in Polynomial Time. *J. Assoc. Comput. Mach.* **36** (1989) 435–474.

[14] R.K. Ghosh and G.P. Bhattacharjee, A Parallel Search Algorithm for Directed Acyclic Graphs. *BIT* **24** (1984) 134–150.

[15] E. Gioan, C. Paul, M. Tedder and D. Corneil, *Quasi-linear circle graph recognition*. Technical Report, University of Toronto (2009).

[16] M.C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*. Academic Press (1980).

[17] R. Gupta, J. Walrand and O. Goldschmidt, Maximal Cliques in Unit Disk Graphs: Polynomial Approximation, in *Proc. International Network Optimization Conference (INOC)*, Lisbon (2005).

[18] C.W. Ho and R.C.T. Lee, Efficient Parallel Algorithms for Finding Maximal Cliques, Clique Trees, and Minimum Coloring on Chordal Graphs. *Inf. Process. Lett.* **28** (1988) 301–309.

[19] E. Jennings and L. Motyckova, A Distributed Algorithm for finding All Maximal Cliques in a Network Graph, in *Proc. of the 1st Latin American Symposium on Theoretical Informatics (LATIN'92)* (1992) 281–293.

[20] R.M. Karp and V. Ramachandran, Parallel Algorithms for Shared-Memory Machines. Edited by J. van Leeuwen *Handbook of Theoretical Computer Science*. MIT Press/Elsevier (1990) 869–941.

[21] P.N. Klein, Efficient Parallel Algorithms for Chordal Graphs. *SIAM J. Comput.* **25** (1996) 797–827.

[22] K. Makino and T. Uno, New Algorithms for Enumerating All Maximal Cliques, in *Proc. Scandinavian Workshop on Algorithm Theory – SWAT* (2004) 260–272.

[23] J.W. Moon and L. Moser, On cliques in graphs. *Israel J. Math* **3** (1965) 23–28.

[24] W. Naji, Graphes des Cordes, Caractérisation et Reconnaissance. *Disc. Math.* **54** (1985) 329–337.

[25] J. Naor, M. Naor and A.A. Schäffer, Fast Parallel Algorithms for Chordal Graphs. *SIAM J. Comput.* **18** (1989) 327–349.

[26] F. Protti, F.M.G. França and J.L. Szwarcfiter, On Computing All Maximal Cliques Distributedly, in *Proc. Workshop on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR)* (1997) 37–48.

[27] J.P. Spinrad, Recognition of Circle Graphs. *J. Algor.* **16** (1994) 264–282.

[28] J.L. Szwarcfiter and M. Barroso, Enumerating the Maximal Cliques of Circle Graph, *Graph Theory, Combinatorics, Algorithms and Applications.* edited by F.R.K. Chung, R.L. Graham and D.F. Hsu, SIAM Publications (1991) 511–517.

[29] R.E. Tarjan, Depth First Search and Linear Graph Algorithms. *SIAM J. Comput.* **1** (1972) 146–160.

[30] S. Tsukiyama, M. Ide, H. Arujoshi and H. Ozaki, A New Algorithm for Generating the Maximal Independent Sets. *SIAM J. Comput.* **6** (1997) 505–517.

[31] L.G. Valiant, A Bridging Model for Parallel Computation. *Communication of the ACM* **33** (1990) 103–111.

[32] C.S. Wang and R.S. Chang, A Parallel Maximal Cliques Algorithm for Interval Graphs with Applications. *J. Inf. Sci. Eng.* **13** (1997) 649–663.

[33] Y. Zhang, *Parallel Algorithms for Problems Involving Directed Graphs*. Ph.D. thesis, Department of Computer Science, Drexel University (1990).

[34] Y. Zhang, F.N. Abu-Khzam, N.E. Baldwin, E.J. Chesler, N.F. Samatova and M.A. Langston, Genome-Scale Computational Approaches to Memory-Intensive Applications in Systems Biology. *Proceedings of SC*, Seattle, Washington (2005).

Communicated by C. Choffrut.
Received March 30, 2009. Accepted June 2, 2010.