JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 15, 407-417 (1999)

Short Paper

Parallel Decomposition of Generalized Series-Parallel Graphs*

CHIN-WEN HO, SUN-YUAN HSIEH+ AND GEN-HUEY CHEN+
Department of Computer Science and Information Engineering
National Central University
Chung-Li, Taiwan 320, R.O.C.
E-mail: hocw@csie.ncu.edu.tw
+*Department of Computer Science and Information Engineering*
National Taiwan University
Taipei, Taiwan 106, R.O.C.
E-mail: ghchen@csie.ntu.edu.tw

Generalized series-parallel (*GSP*) graphs belong to the class of decomposable graphs which can be represented by their decomposition trees. Given a decomposition tree of a *GSP* graph, there are many graph-theoretic problems which can be solved efficiently. An efficient parallel algorithm for constructing a decomposition tree of a given *GSP* graph is presented. It takes $O(\log n)$ time with $C(m, n)$ processors on a CRCW PRAM, where $C(m, n)$ is the number of processors required to find connected components of a graph with m edges and n vertices in logarithmic time. Based on our algorithmic results, we also derive some properties for *GSP* graphs, which may be of interest in and of themselves.

Keywords: parallel algorithm, generalized series-parallel graph, CRCW PRAM, decomposable graph, decomposition tree

1. INTRODUCTION

Generalized series-parallel (*GSP*) graphs are those graphs which can be obtained from a set of single-edges by applying, recursively, series, parallel and generalized-series compositions. Such graphs belong to the class of k-terminal graphs defined in [1] since each *GSP* graph G has two special vertices, called terminals, and satisfies the condition that G is generated by the above composition rules acting only at terminals. *GSP* graphs include series-parallel (*SP*) graphs, outplanar graphs, trees, unicyclic graphs, C_N-trees, C-trees, 2-trees, cacti and filaments (square, triangular and hexagonal) [1].

K-terminal graphs, also called decomposable graphs [2], can be decomposed into a set of primitive graphs by following a certain set of composing rules. The decomposition structure of the given decomposable graph can be represented by a decomposition tree in

Received July 17, 1997; accepted May 1, 1998.
Communicated by Wen-Lian Hsu.
*A preliminary version of this paper appeared in the proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97) Las Vegas, Nevada, pp. 890-896, 1997.
Research supported in part by NSC under Grant 83-0408-E-008-004.

which each leaf represents a primitive graph, and each internal node represents an appropriate composition operation. Given a *GSP* graph in the form of its decomposition tree, there exist linear-time sequential algorithms for solving many graph-theoretic problems such as the maximum cut set, the maximum cardinality of a minimal dominating set, etc. [3]. Furthermore, Bern, Lawler and Wong [2] have shown that by providing a decomposition tree for a given decomposable graph, many combinatorial optimization problems related to finding an optimal subgraph can be solved in linear time by using a dynamic programming approach if the desired subgraph satisfies some property with respect to the composition rules. For example, such problems include the maximum independent set, maximum matching set and minimum dominating set problems on several subclasses of decomposable graphs including *GSP* graphs [2]. For parallel computations, cost optimal parallel algorithms for solving the above optimal subgraph problems and several others can be obtained by applying a binary tree contraction technique [4] to a decomposition tree of a decomposable graph, which takes $O(\log n)$ time with $O(n/\log n)$ processors on an EREW PRAM, where n is the size of the decomposition tree [4-8]. Consequently, this implies that a wide range of optimal subgraph problems on *GSP* graphs can be solved optimally if their decomposition trees are given. Hence, the problem of constructing decomposition trees is crucial for both sequential and parallel *GSP* graph computations.

The sequential $O(n)$ time algorithm of recognition *GSP* graphs was presented in [1], where n is the number of vertices of an input graph. In this paper, we develop a parallel strategy for constructing a decomposition tree of a given graph G if G is recognized as a *GSP* graph in our algorithm. The time complexity of the algorithm is $O(\log n)$ time, and the number of processors used is $C(m, n)$, where $C(m, n)$ is the number of processors required to compute the connected components of a graph with m edges and n vertices in logarithmic time. Note that since *GSP* graphs are planar, in our complexity result, m is of the same order as n. The best result for $C(m, n)$ is $O((m+n)\alpha(m, n)/\log n)$, where α is the inverse ackermann function [9]. Our algorithm can run on a deterministic parallel random access machine that permits concurrent reads and concurrent writes (CRCW) in its shared memory; in case of a write conflict, the model allows an arbitrary processor to succeed [10].

2. PRELIMINARIES

Let $V(G)$ and $E(G)$ stand, respectively, for the vertex set and the edge set of an undirected graph G, and let $n = |V(G)|$ and $m = |E(G)|$. We denote an edge between x and y as (x, y). An undirected graph $G = (V, E)$ is *connected* if there exists a path between any pair of vertices in V. A *connected component* for a graph G is a maximal induced connected subgraph of G. A vertex $v \in V$ is an *articulation vertex* or *cut vertex* of a connected undirected graph $G = (V, E)$ if the subgraph induced by $V-\{v\}$ is not connected. G is *biconnected* if it contains no articulation point. A *bicomponent* (or *block*) of G is a maximal induced biconnected subgraph of G. In this paper, the graphs we discuss are all connected.

Generalized-series-parallel (*GSP*) graphs are defined recursively as follows.

Definition 1: (1) A graph consisting of two vertices u and v, and a single edge (u, v) is a primitive *GSP* graph with terminals u and v. (2) If G_1 and G_2 are two *GSP* graphs with terminals $\{u_1, v_1\}$ and $\{u_2, v_2\}$, respectively, then the graph obtained by means of any of the following three operations is a *GSP* graph:

(a) The series composition of G_1 and G_2: identifying v_1 with u_2 and specifying u_1 and v_2 as the terminals of the resulting graph.
(b) The parallel composition of G_1 and G_2: identifying u_1 with u_2 and v_1 with v_2, and specifying u_1 and v_1 as the terminals of the resulting graph.
(c) The generalized-series composition of G_1 and G_2: identifying v_1 with u_2 and specifying u_2 and v_2 as the terminals of the resulting graph.

The family of *series-parallel* (*SP*) graphs consists of those *GSP* graphs that are obtained by using only the series and parallel compositions of Definition 1. A characterization of *SP* graphs can be obtained by using the following definitions of two inverse operations of series and parallel compositions. Suppose that the degree of a vertex w in $V(G)$ is two; the *series reduction* of two edges in series $e_1 = (u, w)$ and $e_2 = (w, v)$ is an operation in which e_1 and e_2 are replaced by a new edge $e = (u, v)$. The *parallel reduction* of two parallel edges (two edges with common end vertices) $e_1 = (u, v)$ and e_2 is an operation in which e_1 and e_2 are replaced by a new edge $e = (u, v)$.

Lemma 2.1: [6, 11] A graph G is an *SP* graph if and only if G can be reduced to a single edge by means of a sequence of series and parallel reductions.

A *GSP* graph G can be represented by a decomposition tree T which is defined as follows.

Definition 2: (1) The tree consisting of a single vertex labeled $e = (u, v)$ is a decomposition tree of the primitive *GSP* graph $G = (\{u, v\}, \{(u, v)\})$. (2) Let G be the *GSP* graph generated by some composition of two *GSP* graphs G_1 and G_2, and let T_1 and T_2 be the decomposition trees of G_1 and G_2, respectively. Then, the decomposition tree T of G is the tree with the root r labeled by an appropriate composition ("G", "P" or "S", depending on which composition is applied to generate G), and with the two roots of T_1 and T_2 as the left and right children of r, respectively.

The definition of a decomposition tree of an *SP* graph is similar to Definition 2 but without an internal node labeled "G" since each *SP* graph is generated only by means of series and parallel compositions. Fig. 1 shows a *GSP* graph with its decomposition tree.

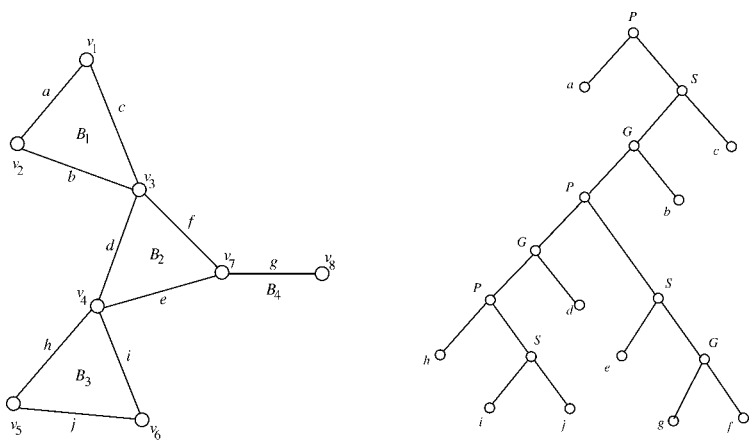

Fig. 1. A *GSP* graph with its decomposition tree.

3. AN ALGORITHM FOR DECOMPOSING *GSP* GRAPHS

In this section, we will first discuss two important properties of *GSP* graphs and one result of *SP* graphs, and then present our decomposition algorithm.

A characterization of *GSP* graphs can be derived by using three specified operations: series, parallel and generalized-series reductions, which can be viewed as the inverse operations of series, parallel and generalized-series compositions. (Series and parallel reductions are defined in Section 2.) The *generalized-series reduction* of two edges $e_1 = (u, v)$ and $e_2 = (v, w)$, where u is a *pendent* vertex (a vertex with degree one), is an operation in which e_1 and e_2 are replaced by a new edge $e = (v, w)$.

Suppose that T is a decomposition tree of a given *GSP* graph G. Following the idea in the proof of Lemma 2.1, it can be shown that T corresponds to a set of sequences of series, parallel and generalized-series reductions such that each sequence of reductions can reduce G to a single edge. Consider the following scheme: Find some internal node u of T whose left and right child represent two edges e_1 and e_2 of G. Then, apply some appropriate reduction to e_1 and e_2 according to the label of u. That is, if u is labeled "*G* ", then generalized-series reduction is applied, and other cases for "P" and "S" can be applied similarly. By the definition of reduction, e_1 and e_2 are replaced with a new edge e. Thus, the original graph G becomes another "smaller" graph and has the decomposition tree obtained from T by replacing the subtree rooted at u, whose two children are e_1 and e_2, with the new leaf node e. Clearly, if we repeatedly execute the above scheme, then G can be finally reduced to a single edge.

Conversely, given a reducing sequence δ which can reduce G to a single edge, an approach described in [6] can be slightly modified to construct a unique decomposition tree T corresponding to δ. The construction of T follows the process that reduces G to a single edge by δ. Hence, we have the following lemma.

Lemma 3.1: A graph G is a *GSP* graph if and only if it can be reduced to a single edge by a sequence of series, parallel and generalized-series reductions.

Assume that G is a *GSP* graph generated by a sequence of compositions $\delta_1, \delta_2, ..., \delta_k$. If some δ_i is the generalized-series composition applied to two *GSP* graphs G_1 and G_2 with terminals $\{u_1, v_1\}$ and $\{u_2, v_2\}$, respectively, then the vertex $v_1 (= u_2)$ will be a cut vertex of G. From the above observation, it is easy to derive the following characterization of *GSP* graphs.

Lemma 3.2: [1] A graph G is a *GSP* graph if and only if each block of G is an *SP* graph.

Each block of an *SP* graph is also an *SP* graph. However, the converse is not true. For example, each tree is a *GSP* graph since each block of a tree is a primitive *SP* graph (the graph consisting of only one edge); each tree containing a node of degree greater than two is not an *SP* graph since no sequence of series and parallel reductions can reduce such a tree to a single edge.

Based upon an open ear decomposition, Eppstein presented an efficient parallel algorithm for recognizing biconnected *SP* graphs [12]. Given a biconnected *SP* graph G with an open ear decomposition $\{P_0, P_1, ..., P_{r-1}\}$ (the definition of an open ear decomposition

and the details of implementation for finding one can be found in [13]), the algorithm can construct a decomposition tree corresponding to a reducing sequence which can reduce G to P_0, where P_0 is an edge of G. Since any edge e of G can be chosen to construct an open ear decomposition $\{P_0, P_1,..., P_{r-1}\}$ with $P_0 = e$ [13], we have the following result.

Lemma 3.3: [12, 13] Given a biconnected *SP* graph G and an arbitrarily selected edge e of G, a decomposition tree can be constructed in $O(\log n)$ time using $C(m, n)$ processors on a CRCW PRAM. Moreover, such a tree corresponds to a reducing sequence which can reduce G to e.

According to Lemma 3.2 and the algorithm presented in [12], we can easily recognize *GSP* graphs, but to construct their decomposition trees efficiently, we need some useful strategies described in our algorithm. Before presenting our algorithm, we provide the following definitions which are necessary for construction of a decomposition tree of a *GSP* graph. Suppose that $a_1, a_2,..., a_l$ are the cut vertices of G, and that $B_1, B_2,..., B_k$ are the blocks of G. The block-cut vertex tree BT is defined as follows [14]. The vertex set of BT is $\{a_1, a_2,..., a_l, b_1, b_2,..., b_k\}$, and (a_i, b_j) is an edge of BT if a_i is a vertex of B_j. In addition, we call each b_i (respectively, a_i) a *block-vertex* (respectively, a *non-block vertex*) of BT. Let BT' be the rooted tree by selecting one block-vertex b_r of BT as the root. Then, for each block-vertex b_i ($i \neq r$), there is a unique directed path P from b_i to the root b_r. If b_j is the first block-vertex of BT' encountered by b_i in P, we call b_j the *parent* of b_i(denoted as $Par(b_i)$), and B_j (respectively, B_i) is the *parent block* (respectively, a *child block*) of B_i (respectively, B_j). We denote $Child(B_i)$ as the set of child blocks of B_i and call the block with no child block the *leaf block*.

Algorithm Decomposing GSP.

Step 1: Find the blocks $B_1, B_2,..., B_k$ of G.

Step 2: /* Prepare for construction of the decomposition tree of G */

2-1 Construct the block-cut vertex tree BT of G.

2-2 Transfer BT to a rooted tree BT' by selecting an arbitrary block-vertex b_r as the root. /* Thus, B_r is the root block of G */

/* The following steps: 2-3, 2-4, 2-5, and Step 3 are executed in parallel for each block B_i, $1 \le i \le k$. */

2-3 Find the parent and child blocks for B_i by using BT'.

2-4 For the cut vertex v connected to $Par(B_i)$, select one edge $e = (v, w)$ of B_i, and mark it as the *main edge* t_i. For the root block B_r, select one arbitrary edge as the main edge.

2-5 For each cut vertex v connected to some child of B_i, select one edge $e = (u, v)$ of B_i, and mark it as the *reducing edge* r_v. Compute the number m of the edges t_j, where each t_j is the main edge of some child block of B_i which is connected to B_i by v. Then, construct a right-skew binary tree structure R_v with internal nodes labeled "G" and leaves labeled t_j and e as follows: order those edges t_j from 1 to m and construct m "G" nodes denoted by "$\mathbf{G}_1$", "$\mathbf{G}_2$",..., "$\mathbf{G}_m$", such that the right child of each "$\mathbf{G}_l$" ($1 \le l \le m$-1) and "$\mathbf{G}_m$" is the node "$\mathbf{G}_{l+1}$" and e, respectively, and the left child of each "$\mathbf{G}_l$" ($1 \le l \le m$) is a node labeled t_j for some j (according to

the ordering associated with it). /* There may be some edge e of B_i which is marked as the main edge and also as a reducing edge, but this does not affect the result of this algorithm */

Step 3: Apply the recognition of the *SP* graph algorithm [12] to B_i and construct its decomposition tree T_i corresponding to a reducing sequence which can reduce B_i to its main edge. If one of the blocks is not an *SP* graph, reject it. /* G is not a *GSP* graph */

Step 4: /* Construct the decomposition tree T of G. Steps 4-1 and 4-2 are executed in parallel for each reducing edge and main edge, respectively */

4-1 For each reducing edge $r_v = (u, v)$ of T_i, if r_v is also marked as the reducing edge for u, then replace r_v with the root of R_u, and replace the edge (u, v) (which appears as a leaf of R_u)with the root of R_v. Otherwise, replace r_v with the root of R_v.

4-2 Replace each t_i node with the root of T_i.

Fig. 2 shows the construction of a decomposition tree for the *GSP* graph G with four blocks B_i $(1 \le i \le 4)$ in Fig. 1. We first select B_1 as the root in Step 2-2 and find $Child(B_1) = \{B_2\}$ and $Child(B_2) = \{B_3, B_4\}$ in Step 2-3. In Step 2-4, the edges $t_1 = a$, $t_2 = d$, $t_3 = h$ and $t_4 = g$ are selected as the main edges of B_i $(1 \le i \le 4)$, respectively. Step 2-5 selects the edges $r_{v_3} = b$ and $\{r_{v_4} = d, r_{v_7} = \mathrm{f}\}$ as the reducing edges of B_1 and B_2, respectively, and then constructs the tree structures R_{v_3}, R_{v_4} and R_{v_7} for the cut vertices v_3, v_4 and v_7. Step 3 constructs in parallel the decomposition trees T_i $(1 \le i \le 4)$, where each T_i corresponds to a reducing sequence which can reduce B_i to its main edge t_i. Finally, a decomposition tree T of G can be generated in Step 4 by replacing r_{v_3}, r_{v_4} and r_{v_7} with the roots of R_{v_3}, R_{v_4} and R_{v_7}, respectively (each dashed arrow represents a replacement), and by replacing each t_i with the root of T_i $(1 \le i \le 4)$.

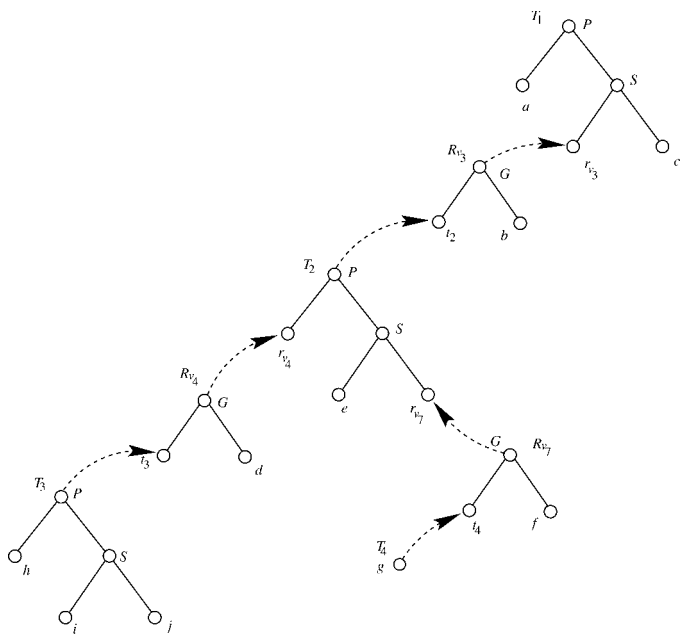

Fig. 2. A decomposition tree T of the *GSP* graph shown in Fig. 1 can be constructed by the algorithm.

Theorem 3.4: The algorithm **Decomposing GSP** can recognize a *GSP* graph and construct its decomposition tree correctly.

Proof: If G is not a *GSP* graph, then by Lemma 3.2, some block of G is not an *SP* graph; thus, G will be rejected in Step 3. Conversely, if G is a *GSP* graph, the blocks of G are all *SP* graphs by Lemma 3.2. The following claim shows that a decomposition tree T of G can be constructed correctly by our algorithm. This claim can be proved by induction based on k, the number of blocks.

Claim A: If G is a *GSP* graph, then the algorithm can construct a decomposition tree T of G, such that T corresponds to a reducing sequence which can reduce G to the main edge of the root block B_r selected in our algorithm.

If $k = 1$, then G contains only one block which will be selected as the root B_r in Step 2-2. By Lemma 3.3, we can construct a decomposition tree corresponding to a reducing sequence which can reduce B_r to its main edge; thus, the claim is correct. Assume the claim is correct for any *GSP* graph with the number of blocks less than k. Now, let G be a *GSP* graph with k blocks $B_1, B_2,.., B_k$. We select one block B_r as the root and consider the case of removing B_r (except those cut vertices in B_r) from G. Then, the resulting graph contains several connected components $C_1, C_2,..., C_m$, where $m < k$. Clearly, the number of blocks of each C_i ($1 \le i \le m$) is less than k. According to the induction hypothesis, our algorithm can construct a decomposition tree T_{c_i} of C_i such that T_{c_i} corresponds to a reducing sequence which can reduce C_i to the main edge $t_i = (u_i, v_i)$ of the root block B_i of C_i. The block B_i is a child block of B_r which is connected to B_r by the cut vertex v_i. After reducing each C_i to t_i (by applying the reducing sequence corresponding to T_{c_i}), the graph G is reduced to another graph G', where G' contains the block B_r and the edges t_i, $1 \le i \le m$ (each of which is connected to B_r by v_i). Note that the end vertex u_i of each $t_i = (u_i, v_i)$ is a pendent vertex. Then, we reduce each t_i by applying generalized series reduction to the edge t_i and some reducing edge r_{v_i} of B_r, where r_{v_i} and t_i are two edges incident to v_i. Such reduction can be represented by the tree structure R_{v_i} generated in Step 2-5. When each t_i has been reduced, the resulting graph contains only one block B_r. By Lemma 3.3, our algorithm can construct a decomposition tree T_r corresponding to a reducing sequence which can reduce B_r to its main edge. Finally, by replacing each reducing edge r_{v_i} of T_r with some tree structure and each t_i with the root of T_{c_i}, the decomposition tree T of G can be generated in Step 4. This is a decomposition tree corresponding to a reducing sequence which can reduce G to the main edge of B_r. By induction, we have proven Claim A. □

Theorem 3.5: The algorithm **Decomposing GSP** can be implemented in $O(\log n)$ time using $C(m, n)$ processors on a CRCW PRAM.

Proof: In Step 1, finding the blocks of G taes $O(\log n)$ time using $C(m,n)$ processors on a CRCW PRAM [9].

In Step 2-1, the block-cut vertex tree BT can be constructed in $O(1)$ time with $O(n+m)$ processors [6]. This can be simulated in $O(\log n)$ time using $O((n+m)/\log n)$ processors by Brent's scheduling principle [10]. In the following steps, all the implementations which take $O(1)$ time using a linear number of processors can apply Brent's scheduling principle to achieve the desired complexity.

Step 2-2 constructs the rooted tree BT' from BT by using the Eulerian tour technique described in [15, 16]. This takes $O(\log k)$ time using $O(k/\log k)$ processors on an EREW PRAM, where k is the number of blocks of G.

In Step 2-3, the parent block B_j of B_i $(1 \le i \le k)$ can be found by using BT' since $b_j = par(par(b_i))$. This can be done in $O(1)$ time with $O(k)$ processors. The child blocks of B_i can be obtained by merging the child lists of children of b_i in BT'. This can be implemented in $O(\log n)$ time with $O(n/\log n)$ processors on a CRCW PRAM.

In Step 2-4, the cut vertex v in B_i connected to $Par(B_i)$ can be determined in $O(1)$ time; thus, the main edge of B_i $(1 \le i \le k)$ can be selected in constant time with $O(k)$ processors.

In Step 2-5, first select an edge $e = (u, v)$ of B_i for each cut vertex v connected to some child of B_i and mark it as a reducing edge r_v. This can be done in $O(1)$ time with $O(k)$ processors. Then, compute the number of the edges t_j, where each t_j is the main edge of some child block of B_i which is connected to B_i by v, by using optimal parallel prefix sum computation [15]. Ordering the edges t_j and making up a right-skew tree structure R_v can be done by means of optimal parallel list ranking [15]. Thus, Step 2-5 can be done in $O(\log n)$ time with $O(n/\log n)$ processors on an EREW PRAM.

In Step 3, recognition of the *SP* graph and constructing a decomposition tree for each block take $O(\log n)$ time with $C(m, n)$ processors on a CRCW PRAM by Lemma 3.3.

In Step 4, for each reducing edge $r_v = (u, v)$ of B_i, checking whether r_v is also the reducing edge selected for u can be accomplished in constant time. It is clear that the other implementations of Step 4-1 and 4-2 can be achieved in $O(1)$ time using $O(k)$ processors. □

4. SOME PROPERTIES OF *GSP* GRAPHS

In this section, we derive two properties of *GSP* graphs from the algorithmic results obtained in the previous section, which may be of interest in and of themselves.

Suppose that G is a *GSP* graph. Then, by definition, there exist two special vertices u and v as the two terminals of G. By Lemma 3.1, we observe that G is a *GSP* graph with terminals $\{u, v\}$ if and only if G can be reduced to a single edge $e = (u, v)$ by a sequence of series, parallel and generalized-series reductions. Hence, we can not select arbitrarily any two vertices as the terminals of the given *GSP* graph. The graph shown in Fig. 3 is a *GSP* graph with terminals $\{v_2, v_3\}$ or $\{v_3, v_6\}$ but is not a *GSP* graph with terminals $\{v_2, v_5\}$ since no reducing sequence can reduce the graph to the edge $e = (v_2, v_5)$.

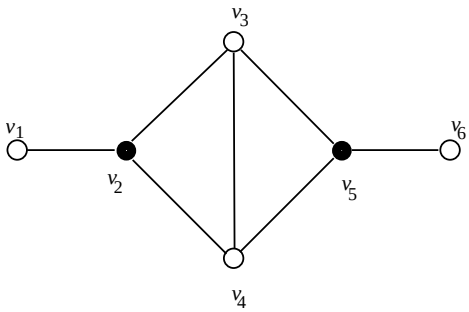

Fig. 3. The graph is not a *GSP* graph with respect to v_2 and v_5.

Theorem 4.1: Let G be a *GSP* graph. Then, for any edge $e = (u, v)$ of G, G is a *GSP* graph with terminals $\{u, v\}$.

Proof: Let G be a *GSP* graph, and let $e = (u, v)$ be an edge of some block B_r. If we select B_r as the root and mark e as the main edge of B_r, then from the proof of Theorem 3.4, we can construct a decomposition tree corresponding to a reducing sequence which can reduce G to the edge $e = (u, v)$. Hence, G is a *GSP* graph with terminals $\{u, v\}$. □

Theorem 4.2: Let u and v be two vertices of G. G is a *GSP* graph with terminals $\{u, v\}$ if and only if $G' = (V(G), E(G) \cup \{(u, v)\})$ is a *GSP* graph.

Proof: If G is a *GSP* graph with terminals $\{u, v\}$, then it is clear that G' is a *GSP* graph since G' can be generated by applying a parallel composition to G and the edge $e' = (u, v)$. (Note that each edge (u, v) is a primitive *GSP* graph with terminals $\{u, v\}$.)

Conversely, suppose that G' is a *GSP* graph. Let the added edge $e' = (u, v)$ be an edge in some block B_r of G'. If G' contains only one block B_r, then G is an *SP* graph by Lemma 3.2. According to Lemma 3.3, we can construct a decomposition tree T of G' corresponding to a reducing sequence δ which can reduce G' to the edge e'. Since G' is biconnected, in each reducing step of applying δ to G, the resulting graph remains biconnected. From this observation, we can conclude that the root r of T must not be labeled "S" (otherwise, G' is not biconnected, a contradiction); thus, r is labeled "P". Moreover, consider the path of T from leaf e' to root r. The internal nodes of such path are all labeled "P" since if there exists some internal nodes labeled "S", then vertex u (or v) will be removed by a series reduction, thus contradicting that G' can be reduced to $e' = (u, v)$. Based on the structure of T, we can apply a reducing scheme described in the proof of Lemma 3.1 to reduce G' to another graph. Such a reduced graph has a decomposition tree T' which can be obtained from T by replacing each subtree of T which has a root labeled "S" with an appropriate edge. This edge is generated by executing a reducing sequence corresponding to the above subtree. Hence, all the internal nodes of T' are labeled "P". Then, we can transfer T' to another "equivalent" tree T'' (i.e., both T' and T'' are decomposition trees of the same graph) such that T'' contains the same number of internal nodes as T', each internal node of T'' is labeled "P" and e' is a child of the root of T''. The above observation implies that there exists a decomposition tree of G' with its root r labeled "P", and that e' and the subtree T_G are the two children of r, where T_G corresponds to a reducing sequence which can reduce G to a single edge $e = (u, v)$. Thus, T_G is a decomposition tree of G, and G is a *GSP* graph with terminals $\{u, v\}$.

On the other hand, suppose that G' contains more than one block. We can select B_r as the root and select the edge $e' = (u, v)$ as the main edge in Step 2-4 of our algorithm, such that e' can not also be selected as a reducing edge. This can be achieved since adding e' to G makes the number of edges of B_r larger than one. From the proof of Theorem 3.4, we know that a decomposition tree T of G' can be constructed from the decomposition tree T_r of B_r by replacing each reducing edge of B_r with its corresponding tree structure and then replacing the main edges of $Child(B_r)$ with their associated decomposition trees. Moreover, from the proof of Theorem 3.4 , there exists a reducing sequence which can reduce G' from the leaves to the root until B_r is the only remaining block. Note that any reduction in the above reducing sequence can not replace e' since e' is not a reducing edge of B_r. The above

observation implies that the root of the decomposition tree constructed by our algorithm is labeled "P", and that each internal node on the path from e' to the root is also labeled "P". Hence, following the discussion described in the previous paragraph, we can conclude that there exists a decomposition tree T of G' with the root r labeled "P" and with e' and the subtree T_G as the two children of r, such that T_G corresponds to a reducing sequence which can reduce G to a single edge $e = (u, v)$. Thus, G is a *GSP* graph with terminals $\{u, v\}$. □

5.CONCLUSIONS

In this paper, we have presented an efficient parallel algorithm to construct a decomposition tree for a given *GSP* graph. It takes $O(\log n)$ time with $C(m, n)$ processors on a CRCW PRAM. From this algorithm, we can further obtain another result by considering the special input instance of the algorithm. Recall that trees are contained within the class of *GSP* graphs. If the input graph is known to be a tree, then its decomposition structure can be constructed by our algorithm without executing Step 1 and Step 3 since each block of trees contains only one edge. Moreover, because all of the steps in our algorithm can be done optimally except for the above two steps, the decomposition structure of the given tree can be constructed in $O(\log n)$ time with $O(n/\log n)$ processors on an EREW PRAM. According to the discussion of section 1, this implies that several optimal subgraph problems on trees can be solved optimally by applying the binary tree contraction technique. Note also that the bottleneck in our algorithm is the problem of finding connected components of a graph. According to [17], this problem can be solved by using an optimal randomized parallel algorithm. Hence, combining this result with our algorithm, we can easily obtain an optimal randomized parallel algorithm to construct a decomposition tree for a given *GSP* graph.

REFERENCES

1. T. V. Wimer and S. T. Hedetniemi, "K-terminal recursive families of graphs," *Congressue Numerantium*, Vol. 63, 1988, pp. 161-176.
2. M. V. Bern, E. L. Lawler and A. L. Wong, "Linear-time computation of optimal subgraphs of decomposable graphs," *Journal of Algorithms*, Vol. 8, No. 2, 1987, pp. 216-235.
3. E. Hare, S. Hedetniemi, R. Laskar, K. Peters and T. Wimer, "Linear-time computability of combinatorial problems on generalized series-parallel graphs," *Discrete Algorithms and Complexity*, Academic Press, 1987, pp. 437-455.
4. K. Abrahamson, N. Dadoun, D. G. Kirkpatrick and T. Przytycka, "A simple parallel tree contraction algorithm," *Journal of Algorithms*, Vol. 10, No. 2, 1989, pp. 287-302.
5. X. He, Yesha and Yaacov, "Binary tree algebraic computation and parallel algorithms for simple graphs," *Journal of Algorithms*, Vol. 9, No. 1, 1988, pp. 92-113.
6. X. He, "Efficient parallel algorithms for series parallel graphs," *Journal of Algorithms*, Vol. 12, No. 3, 1991, pp. 409-430.
7. C. W. Ho, S.Y. Hsieh and G. H. Chen, "An efficient parallel strategy for computing *k*-terminal reliability and finding most vital edges in 2-trees and partial 2-trees," *Journal of Parallel and Distributed Computing*, Vol. 51, No. 2, 1998, pp. 89-113.

8. Yain, Shi-Jim, "Optimal parallel algorithms for decomposable graphs", master thesis, Institute of Computer Science and Electrical Engineering, National Central University, Taiwan, R.O.C., 1993.
9. R. Cole and R. Thurimella, "Approximate parallel scheduling. II: Application to logarithmic-time optimal parallel graph algorithms," *Information and Computation*, Vol. 92, No. 1, 1991, pp. 1-47.
10. R. M. Karp and V. Ramachandran, "Parallel algorithms for shared memory machines," *Handbook of Theoretical Computer Science*, North-Holland, Amsterdan, 1990, pp. 869-941.
11. R. J. Duffin, "Topology of series parallel networks," *Journal of Mathematical Analysis and Applications*, Vol. 10, No. 2, 1965, pp. 303-318.
12. D. Eppstein, "Parallel recognition of series-parallel graphs," *Information and Computation*, Vol. 98, No. 1, 1992, pp. 41-55.
13. Y. Maon, B. Schieber and U. Viskin, "Parallel ear decomposition search (EDS) and st numbering in graphs," *Theoretical Computer Science*, Vol. 47, No. 3, 1986, pp. 277-298.
14. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
15. J. Ja'Ja', *An Introduction to Parallel Algorithms*, Addison Wesley, 1992.
16. R. E. Tarjan and U. Vishkin, "Finding biconnected components and computing tree functions in logarithmic parallel time," *SIAM Journal on Computing*, Vol. 14, No. 4, 1985, pp. 862-874.
17. H. Gazit, "An optimal randomized parallel algorithm for finding connected components in a graph," *SIAM Journal on Computing*, Vol. 20, No. 6, 1991, pp. 1046-1067.

Chin-Wen Ho (何錦文) received the B.S. degree in mathematics from National Taiwan University in 1979, and the M.S. and Ph.D. degrees in computer science from National Tsing Hua University, Hsinchu, Taiwan, in 1984 and 1988, respectively. He is an associate professor in the Department of Computer Science and Information Engineering at National Central University, Chung-Li, Taiwan. He is also a member of the IEEE Computer Society. His research interests include algorithm design and analysis, graph theory, and parallel processing.

Sun-Yuan Hsieh（**謝孫源**）received the B.S. degree and MS degree in computer science from Tamkang and National Central University, Taiwan, in 1992 and 1994, respectively. He is currently a Ph.D. student in the Department of Computer Science and Information Engineering, National Taiwan University. His current research interest is parallel computation.

Gen-Huey Chen（**陳健輝**）was born in Taiwan on October 10, 1959. He received the B.S. degree in computer science from National Taiwan University in June, 1981, and the M.S. and Ph.D. degrees in computer science from National Tsing Hua University, Taiwan, in June,1983, and January, 1987, respectively. He joined the faculty of the Department of Computer Science and Information Engineering, National Taiwan University, in February, 1987, and he has been a professor since August 1992. He received the Distinguished Research Award from the National Science Council, Taiwan, in 1993, 1995, and 1997. His current research interests include graph-theoretic interconnection networks, parallel and distributed computing, and design and analysis of algorithms.