An Efficient Parallel Strategy for Computing K -Terminal Reliability and Finding Most Vital Edges in 2-Trees and Partial 2-Trees¹

Chin-Wen Ho²

*Department of Computer Science and Information Engineering, National Central University,
Chung-Li, Taiwan*

and

Sun-Yuan Hsieh and Gen-Huey Chen

*Department of Computer Science and Information Engineering, National Taiwan University,
Taipei, Taiwan*

Received July 18, 1996; revised March 2, 1998; accepted March 18, 1998

In this paper, we first develop a parallel algorithm for computing K -terminal reliability, denoted by $R(G_K)$, in 2-trees. Based on this result, we can also compute $R(G_K)$ in partial 2-trees using a method that transforms, in parallel, a given partial 2-tree into a 2-tree. Finally, we solve the problem of finding most vital edges with respect to K -terminal reliability in partial 2-trees. Our algorithms take $O(\log n)$ time with $C(m, n)$ processors on a CRCW PRAM, where $C(m, n)$ is the number of processors required to find the connected components of a graph with m edges and n vertices in logarithmic time. © 1998 Academic Press

Key Words: Network reliability, K -terminal reliability, partial 2-tree, 2-tree, most vital edge.

1. INTRODUCTION

Analysis of network reliability is of major importance in computer, communication, power, and other networks. Because some components of a particular network may be subjected to random failure, we need to compute, as efficiently as possible, the probability that the network is still functional.

Let $G = (V, E)$ be a graph with vertex set V and edge set E , and let K be a specified subset of V . Each vertex is assumed to be perfectly reliable, and each edge $e \in E$ may fail independently with known probability $1 - \text{Pro}(e)$ (i.e., $\text{Pro}(e)$ is the

¹ This work was partially supported by the National Science Council of Taiwan under Grant NSC84-2213-E-008-005. We thank the anonymous referees for helping us improve the presentation of this paper.

² Corresponding author. Address: Department of Computer Science and Information Engineering, National Central University, Chung-Li, Taiwan 32054. E-mail: hocw@csie.ncu.edu.tw.

probability that the edge is functional). The K -terminal reliability of G , denoted by $R(G_K)$, is the probability that all vertices in K are connected. Generally, the K -terminal network reliability problem belongs to a class of NP-hard problems [5, 13, 21, 25]. Even for planar graphs, the computation of $R(G_K)$, for $K \neq V$, is also an NP-hard problem [20, 22]. There are linear time algorithms for solving the problem when $|K| = 2$ or $K = V$ for some special graphs [20]. For any $K \subseteq V$, Satyanarayana and Wood [23] developed a linear time algorithm to solve the problem in series-parallel networks. Wald and Colbourn [26] also developed a sequential linear time algorithm to solve the problem in partial 2-trees. Partial 2-trees, also called generalized series-parallel graphs [16, 29], include properly the class of series-parallel graphs.

Since a 2-tree is a biconnected series-parallel graph, it can be represented by a decomposition tree [6, 11, 17]. In this paper, given a 2-tree, we first represent it using a special form of its decomposition tree and then develop a parallel strategy to solve K -terminal reliability problems in 2-trees using the tree contraction technique [1]. In addition, based on structure characterization of 2-trees, we can first transform a given connected partial 2-tree into a biconnected one and further transform it into a 2-tree by adding dummy edges with probability measure zero. Thus, $R(G_K)$ in partial 2-trees can be computed by using the algorithm we have developed for 2-trees. Finally, we consider an application that finds the *most vital edge* (the edge whose deletion yields the largest decrease in the K -terminal reliability) in a partial 2-tree. The time-processor complexity for finding most vital edges is the same as the result for computing $R(G_K)$ in partial 2-trees. Our algorithms take $O(\log n)$ time with $C(m, n)$ processors on a CRCW PRAM, where $C(m, n)$ is the number of processors required to deterministically find connected components of a graph with m edges and n vertices in logarithmic time. The best result for $C(m, n)$ is $O((m+n)\alpha(m, n)/\log n)$, where α is the inverse Ackermann function [9].

The computation model used in this paper is the deterministic parallel random access machine model (PRAM) which permits concurrent or exclusive reads and writes, including CRCW, CREW, and EREW, in its shared memory. In CRCW PRAM, if a write conflict occurs, the model allows an arbitrary processor to succeed [4].

2. PRELIMINARIES

In this paper, all of the input graphs are finite with no loops or multiple edges (the edges have two common end vertices). However, during the process of handling an input graph by means of some graph operations, we allow the graphs appearing in the process to contain multiple edges. We use standard graph theoretical terminology compatible with [7]. Given a graph $G = (V, E)$, let $n = |V|$ and $m = |E|$. Let $G(V')$, $V' \subseteq V$, be the subgraph of G induced by V' . Let (a, b) denote an undirected edge with end vertices a and b , and let $a \rightarrow b$ denote a *directed edge* (or an *arc*) from a to b . An induced path is a subgraph induced by a vertex set $\{v_1, v_2, \dots, v_k\}$, such that $(v_i, v_{i+1}) \in E$ for all $1 \leq i \leq k-1$, and there is no edge joining two nonconsecutive vertices of the path. An s - t *numbering* of G is a one-to-one

function $\rho: V \rightarrow \{1, \dots, n\}$ satisfying $\rho(s) = 1$, $\rho(t) = n$, and for each $v \in V - \{s, t\}$, there exist adjacent vertices v_1 and v_2 such that $\rho(v_1) < \rho(v) < \rho(v_2)$.

An undirected graph G is called *chordal* if every cycle of length greater than three has a chord, that is, an edge joining two nonconsecutive vertices of the cycle. Equivalently, G does not contain an induced subgraph isomorphic to a k -cycle C_k for $k > 3$.

A 2-tree is defined recursively as follows:

(1) A single edge is a 2-tree.

(2) Given any 2-tree G , let (x, y) be an edge of G . Adding a new vertex z , and the two edges (x, z) and (y, z) produces a 2-tree.

An ordering of the vertex set V , say $v_1, v_2, \dots, v_n$, is called a *perfect elimination order* (PEO) of G if for $i \geq 1$, v_i and its adjacent vertices form a clique in $G(\{v_i, v_{i+1}, \dots, v_n\})$. It is well known that a graph is chordal if and only if it has a PEO [15]. The recursive definition of 2-trees implies that they have the PEO property. Hence, 2-trees are chordal graphs.

Partial 2-trees are simply subgraphs of 2-trees. The following lemma provides a characterization of partial 2-trees.

LEMMA 1 [27]. *Partial 2-trees are exactly those graphs having no subgraph homeomorphic to K_4 .*

Series-parallel (SP) graphs are defined recursively as follows.

(1) A graph consisting of two vertices u and v , and a single edge (u, v) is a primitive SP graph with terminals u and v .

(2) If G_1 and G_2 are two-terminal SP graphs with terminals $\{u_1, v_1\}$ and $\{u_2, v_2\}$, respectively, then the graph obtained by either of the following two operations is an SP 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.

LEMMA 2 [10]. *Let G be a biconnected undirected graph. G is an SP graph if and only if it contains no subgraph homeomorphic to K_4 .*

From Lemma 1 and Lemma 2, we have:

LEMMA 3. *G is a partial 2-tree if and only if each block of G is an SP graph.*

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 G is two; the *series reduction* of two edges in a series, $e_1 = (u, w)$ and $e_2 = (w, v)$, is an operation of replacing e_1 and e_2 with a new edge $e = (u, v)$. The *parallel reduction* of two multiple edges with end vertices u and v is

an operation of replacing the two edges with a new edge $e = (u, v)$. The following lemmas give characterizations of SP graphs based on the above reductions.

LEMMA 4 [10, 17]. *G is an SP graph if and only if G can be reduced to a single edge by a sequence of series and parallel reductions.*

LEMMA 5 [17]. *G is a biconnected SP graph if and only if G can be reduced to any edge of G by a sequence of series and parallel reductions.*

It is well known that SP graphs are a class of decomposable graphs which can be decomposed into a set of primitive graphs by using a certain set of composing rules [6]. The decomposition structure of an SP graph can be represented by a decomposition tree.

A decomposition tree T of an SP graph is defined as follows.

(1) The tree consisting of a single node labeled $e = (u, v)$ is a decomposition tree of the primitive SP graph $G = (\{u, v\}, \{(u, v)\})$.

(2) Let G be the SP graph generated by a composition of two SP graphs, G_1 and G_2 . Then, the decomposition tree T of G is the tree with the root r labeled by an appropriate composition (S or P , depending on which composition is applied to generate G) and with the roots of T_1 and T_2 as children of r , where T_1 and T_2 are decomposition trees of G_1 and G_2 , respectively.

It is easy to see that 2-trees are biconnected SP graphs. Thus, each 2-tree can be represented by a decomposition tree, which is useful for solving the K -terminal reliability problem in our algorithm.

In this paper, we make use of the *binary tree contraction technique* described in [1] to reduce a binary tree to the root node by using a sequence of independent vertex removals. This technique can be regarded as a general method for scheduling parallel computations on trees and has been applied to some problems on graphs (including cographs, SP graphs, etc.) and to the evaluation of binary arithmetic computation trees [1, 17]. This technique recursively applies two operations, *Prune* and *Bypass*, to a given binary tree. *Prune*(v) is an operation which removes a leaf node v from the current tree, and *Bypass*(u) is an operation (which follows a *Prune* operation) which removes a node u with exactly one child w and then lets the parent of u become the new parent of w .

The algorithm initially numbers the leaves in a left to right order and then repeats the following steps. In each step, the operation pairs *Prune* and *Bypass* work only on the leaves with odd index and their parents. Hence, these operation pairs can be performed independently and can delete about half of the leaves and their parents on the binary tree in each step. Therefore, the tree will be reduced to a three-node tree after repeating the steps in $\lceil \log n \rceil$ times.

LEMMA 6 [1]. *If the Prune operation and Bypass operation can be performed by one processor in constant time, the binary tree contraction algorithm can be implemented in $O(\log n)$ time with $O(n/\log n)$ processors on an EREW PRAM, where n is the number of nodes in the input binary tree.*

3. A PARALLEL STRATEGY FOR COMPUTING $R(G_K)$ IN 2-TREES

Wald and Colbourn [26] developed a linear time sequential algorithm to solve the K -terminal reliability problem in partial 2-trees. They first transform an arbitrary partial 2-tree into a 2-tree by adding dummy edges with probability measure zero. Then, by reversing a recursive construction process for the given 2-tree G (i.e., repeatedly deleting a vertex with degree two until the remaining graph is a single edge), they use the following method to compute $R(G_K)$: when a vertex with degree two is removed from G , they summarize the information about the triangle $\{x, y, z\}$, where y is the vertex with degree two, on the arcs $x \rightarrow z$ and $z \rightarrow x$ prior to deleting y . (Note that two arcs $x \rightarrow z$ and $z \rightarrow x$ may have different associated probability measures.) With each arc $\alpha = x \rightarrow z$, there are six associated probability measures which summarize the probabilities examined in a subgraph S which has so far been reduced to the arc α . In order to make this paper self-contained, we will briefly introduce the probability measures:

1. $sc(\alpha)$ is the probability of obtaining a subgraph of S connecting all the specified vertices K in S in addition to both x and z ;
2. $dc(\alpha)$ is the probability of obtaining a subgraph of S in which each specified vertex is connected to x or to z , but there is no path from x to z ;
3. $yn(\alpha)$ is the probability of obtaining a subgraph of S not containing z in which all the specified vertices are connected to x ;
4. $ny(\alpha)$ is the probability of obtaining a subgraph of S not containing x in which all the specified vertices are connected to z ;
5. $nn(\alpha)$ is the probability of obtaining a subgraph of S not containing x or z in which all the specified vertices are connected to one another;
6. $none(\alpha)$ is the probability that S contains no specified vertices.

The initial measures for each arc $\alpha = x \rightarrow z$ corresponding to an edge $e = (x, z)$ are as follows:

1. $sc(\alpha) = Pro(e)$.
2. $dc(\alpha) = 1 - Pro(e)$.
3. $yn(\alpha) = 0$ if $z \in K$, $1 - Pro(e)$ otherwise.
4. $ny(\alpha) = 0$ if $x \in K$, $1 - Pro(e)$ otherwise.
5. $nn(\alpha) = 0$.
6. $none(\alpha) = 0$ if $x \in K$ or $z \in K$, 1 otherwise.

Let $\{x, y, z\}$ be a triangle and y be the vertex of degree two. Let $L = x \rightarrow y$, $R = y \rightarrow z$, $M = x \rightarrow z$. Before deleting y , the probabilities are updated for M by using six equations:

1. $sc(M) = sc(M) \cdot (sc(L) \cdot sc(R) + sc(L) \cdot dc(R) + dc(L) \cdot sc(R) + yn(L) \cdot ny(R)) + dc(M) \cdot sc(L) \cdot sc(R)$.
2. $dc(M) = dc(M) \cdot (sc(L) \cdot dc(R) + dc(L) \cdot sc(R) + yn(L) \cdot ny(R))$.
3. $yn(M) = yn(M) \cdot (sc(L) \cdot yn(R) + yn(L) \cdot none(R))$.

4. $ny(M) = ny(M) \cdot (ny(L) \cdot sc(R) + none(L) \cdot ny(R)).$
5. $nn(M) = nn(M) \cdot none(L) \cdot none(R) + none(M) \cdot (nn(L) \cdot none(R) + none(L) \cdot nn(R) + ny(L) \cdot yn(R)).$
6. $none(M) = none(M) \cdot none(L) \cdot none(R).$

These measures are computed symmetrically for $z \rightarrow x$. We repeatedly execute the above process until G is reduced to a single edge (x, y) with arcs $\alpha = x \rightarrow y$ and its reverse $\beta = y \rightarrow x$. Finally, the probability $R(G_K)$ is taken to be $sc(\alpha) + yn(\alpha) + ny(\alpha) + nn(\alpha)$. Note that the same probability can be obtained from β .

Observing a reduction process generated in the sequential algorithm, we can easily see that the reduction of removing a degree-two vertex is composed of two consecutive reductions: one series reduction, followed by one parallel reduction (for convenience, we call such a reduction a *2-tree reduction*). On the other hand, given a decomposition tree T of a 2-tree G , we can also find a set of reducing sequences $\mathcal{A} = \{\delta_1, \delta_2, \dots, \delta_k\}$, where each δ_i can reduce G to a single edge. (We call δ_i a *reducing sequence corresponding to T* .) Consider the following scheme: find some internal node³ μ in T whose two children are leaves representing two edges e_1 and e_2 of G . Then, apply some appropriate reduction to e_1 and e_2 according to the label (P or S) of μ . By the definition of reduction, e_1 and e_2 are replaced by 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 μ with the new leaf node v_e . Clearly, if we repeatedly execute the above scheme, then G can be finally reduced to a single edge. However, during a reduction process generated from T , we cannot ensure that each series reduction will always be followed by a parallel reduction. In Fig. 1, there are two reducing sequences $\delta_1 = S_1, S_2, P_3, P_4$ and $\delta_2 = S_2, S_1, P_3, P_4$ corresponding to T . Neither of the sequences is a sequence of 2-tree reductions.

From another point of view, given a reducing sequence δ which can reduce G to a single edge, we can 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 [17]. During the reduction process, each edge is associated with a tree structure, and when the reduction process terminates, the tree associated with the single remaining edge is a decomposition tree of G . At the beginning, each edge e of G is associated with a tree consisting of a single node labeled e . When a series (respectively, parallel) reduction is applied to two edges, e_1 and e_2 , we create a new node μ labeled S (respectively, P) and let the trees associated with e_1 and e_2 be the children of μ .

Combining the points of the above discussion, we know that any decomposition tree representing G corresponds to a set of reducing sequences. Moreover, since each reducing sequence corresponds to a unique decomposition tree, the set of all the reducing sequences can be partitioned into several disjoint subsets so that each

³To avoid unnecessary confusion, we use Greek letters to denote the vertices of T in order to distinguish them from the vertices of G ; and, throughout this paper, we call the vertices of T “nodes” and call those of G “vertices”.

FIG. 1. A decomposition tree T of G corresponding to a reducing sequence $\delta = S_1, S_2, P_3, P_4$.

subset consists of those reducing sequences corresponding to the same decomposition tree. In particular, if T corresponds to a sequence of 2-tree reductions, we call T a *canonical tree*. By definition, each 2-tree contains a canonical tree as its decomposition tree. However, not every decomposition tree is canonical. For example, the tree shown in Fig. 1 is not canonical. In order to apply the tree contraction technique to compute $R(G_K)$ efficiently, our method (described in section 3.3) requires that the given decomposition tree be canonical. In the following, we will describe some properties of canonical trees and give a method for transforming any decomposition tree into a canonical one.

3.1. Properties of Canonical Trees

Suppose that T is a decomposition tree of a 2-tree G . For each internal node v of T , let $T(v)$ denote the subtree of T with v as the root, and let $G(v)$ denote the subgraph of G with $T(v)$ as its decomposition tree. The following results give some characterizations of T which are useful for constructing canonical trees.

LEMMA 7. *Suppose that T is a decomposition tree of a 2-tree G . There is no S node v in T such that one child μ of v is also an S node*

Proof. Assume that v and μ are both S nodes. Let ω be the other child of v . By definition, $G(v)$ is an SP graph, with two terminals a and b , obtained by applying a series composition to $G(\mu)$ and $G(\omega)$ with terminals $\{a, b'\}$ and $\{b', b\}$, respectively. Similarly, $G(\mu)$ is obtained from two SP graphs G_1 and G_2 with terminals $\{a, a'\}$ and $\{a', b'\}$, respectively, by means of a series composition. Clearly, we can find three induced paths of G_1 , G_2 , and $G(\omega)$ which are between a, a' ; a', b' ; and b', b , respectively. Since G is biconnected, there exists another induced path between a and b passing through no vertices of G_1 , G_2 , and $G(\omega)$. Hence, there is a cycle

C contributed by the above four induced paths. Moreover, by observing the process of generating $G(v)$ from $T(v)$, we can find that there is no chord in C . Thus, C is an induced cycle with a length of at least four, which contradicts that G is a 2-tree. ■

LEMMA 8. *Suppose that T is a decomposition tree of a 2-tree G . T is a canonical tree if and only if each P node of T has two children: one is an S node, and the other is a P node or a leaf node.*

Proof. We will first prove the if part. If each P node satisfies the condition described in this lemma, we can show by induction on k , the number of P nodes, that T corresponds to a sequence of 2-tree reductions. The base case for T containing a single node representing a primitive 2-tree is trivially correct. Assume that the lemma is correct when the number of P nodes is less than k . Now, suppose that T has k P nodes. Consider a P node v with the largest level in T (the level of v is defined as the distance between v and the root of T). By the assumption of this lemma and the selection rule of v , the vertex v has two children: one is an S node μ , and the other is a leaf node representing an edge of G . Clearly, each child of μ is neither a P node (by the selection rule of v) nor an S node (by Lemma 7); thus, both children of μ are leaf nodes. Hence, $T(v)$ corresponds to a 2-tree reduction which can reduce a triangle $G(v)$ to an edge e . By replacing $T(v)$ in T with a node representing e , we can further find a 2-tree reducing sequence from the resulting tree, which can reduce the graph $G - G(v) + e$ to an edge by the induction hypothesis. Therefore, T corresponds to a 2-tree reducing sequence; thus, it is a canonical tree.

As for the only if part, consider a canonical tree of G . By definition, the tree corresponds to a 2-tree reducing sequence. Based on the scheme for constructing the unique decomposition tree from a reducing sequence, it is clear that the constructed tree of the 2-tree reducing sequence has the desired structure characterizations. ■

A t -fan is a 2-tree with the vertex set $V = \{x, y_1, y_2, \dots, y_t, z\}$ and the edge set $E = \{(x, z), (x, y_1), (y_1, z), (x, y_2), (y_2, z), \dots, (x, y_t), (y_t, z)\}$. The graph shown in Fig. 1 is a 2-fan.

LEMMA 9. *Suppose that T is a decomposition tree of a 2-tree G . If T contains no P node whose parent is an S node, then T is a decomposition tree of a t -fan.*

Proof. By the assumption and Lemma 7, each S node contains no P or S node as a child. Hence, the two children of each S node in T are both leaves. Let v be the root of T , and let x and z be the two terminals of $G(v) = G$. Note that v must be a P node since G is biconnected. We will further show that the two children of each S node represent two edges (x, y) and (y, z) of G .

Assume that the two children of an S node μ represent edges (x', y) and (y, z') with either $x' \neq x$ or $z' \neq z$. Without loss of generality, assume that $x' \neq x$. Notice that x' is a terminal of $G(\mu)$ but not a terminal of $G(v)$. By the composition process of generating $G(v)$ from $G(\mu)$, a series composition must be worked on x' . This implies that the node μ has an ancestor ω in $T(v)$ which is an S node. If $\omega = \text{par}(\mu)$,

this contradicts that G is a 2-tree by Lemma 7. Hence, we conclude that there is some P node whose parent is the S node ω , which contradicts the assumption.

Let α be a P node with the largest level in $T(v)$. Then, one of the children of α must be an S node (otherwise, G contains multiple edges). From the above result, we know that the corresponding parallel composition of α must apply to two sub-graphs of G with terminals x and z . By induction on the number of P nodes, it can be shown that each P node in $T(v)$ has the same property. Hence, G is a t -fan, where t is the number of S nodes of $T(v)$. ■

3.2. An Algorithm for Constructing a Canonical Tree

In the following, we shall develop an algorithm which can transform an arbitrary decomposition tree into a canonical tree. Before proceeding to the description of the algorithm, we will give some definitions.

Let $sib(v)$ (respectively, $par(v)$) denote the sibling (respectively, parent) of v for each internal node v of T . Recall that each S node v represents an SP graph $G(v)$ which is constructed by applying a series composition to two SP graphs G_1 and G_2 . By definition, there exists a vertex of $G(v)$ which is a terminal of both G_1 and G_2 . We call such a vertex the *joint-vertex* of $G(v)$.

ALGORITHM CONSTRUCT_A_CANONICAL_TREE

Input: An arbitrary decomposition tree T of a given 2-tree.

Output: A canonical tree T' .

Step 1. For each S node v of T , find, in parallel, the joint-vertex and the two terminals of $G(v)$.

Step 2. For each S node v of T with joint-vertex y and two terminals x and z , construct, in parallel, a tree-structure t_v as follows: make a P node r as the root of t_v ; let the edge (x, z) and the three-node subtree, obtained by making an S node with two children (x, y) , (y, z) , be the two children of r . We call t_v the tree-structure corresponding to (x, z) .

/* Note that each t_v is a decomposition tree corresponding to a 2-tree reduction. Such a reduction can reduce the triangle $\{x, y, z\}$ to edge (x, z) . */

Step 3. /* Construct a canonical tree from a set of tree-structures */

3-1. For each edge e of G , find, in parallel, the k ($k \geq 0$) tree-structures corresponding to e . If $k \geq 2$, assign an arbitrary order $1, \dots, k$ to those tree-structures. Let t_{v_i} denote the tree-structure corresponding to e with order i (note that t_{v_i} contains a leaf node μ_e representing edge e). Then, replace μ_e of t_{v_i} with the tree-structure $t_{v_{i+1}}$ for all $1 \leq i \leq k-1$.

/* After execution of step 3-1, the resulting trees form a forest $\mathcal{F}$. Each tree-structure in $\mathcal{F}$ is a decomposition tree of some t -fan. Moreover, for each edge in G , there exists at most one tree-structure in $\mathcal{F}$ corresponding to it. */

3-2. For each leaf v_e of the tree-structures in $\mathcal{F}$, replace, in parallel, v_e with the unique tree-structure which corresponds to e (exclusive of the one containing v_e) if such a tree-structure exists in $\mathcal{F}$.

Figure 2 shows an example of constructing a canonical tree of a 2-tree G with six vertices. The decomposition tree shown in Fig. 2(a) contains four S nodes. Thus, after executing Step 2, four tree-structures t_v , t_μ , t_ω , t_τ are generated as shown in Fig. 2(b). Each dashed arrow in Fig. 2(b) represents a replacement executed in Step 3. For example, since t_v and t_μ correspond to the same edge (a, b) , a replacement is executed in Step 3-1. Now, we will show the correctness of the algorithm. Now, we will show the correctness of the algorithm.

THEOREM 1. *Given an arbitrary decomposition tree T of a 2-tree G , algorithm **Construct_a_Canonical_Tree** can correctly construct a canonical tree T' of G .*

Proof. We will prove this by induction on k , the number of the S nodes of T . The base case ($k=0$) for T containing a single node which represents a primitive 2-tree (an edge) is trivially correct. Assume that the theorem is correct when the number of S nodes is less than k . Now, suppose that T contains k S nodes.

If the structure of T satisfies the conditions of Lemma 9, then T is a decomposition tree of a t -fan. Clearly, by executing Steps 1, 2, and 3-1 of the algorithm, a canonical tree will be constructed correctly by Lemma 8.

Otherwise, assume that T does not satisfy the condition of Lemma 9. Hence, there exists an S node with one child v which is a P node. Clearly, $G(v)$ contains

FIG. 2. (a) A 2-tree G with its decomposition tree T . (b) A canonical tree T' can be constructed by our algorithm.

no induced four cycle; hence, $G(v)$ is chordal and a biconnected SP, which implies that it is a non-primitive 2-tree with the decomposition tree $T(v)$. Let a and b be the two terminals of $G(v)$. We can easily observe that (a, b) is an edge of $G(v)$. Moreover, since $par(v)$ is an S node, there is no other S node μ in $T - T(v)$ (i.e., by removing $T(v)$ from T) with a and b as the two terminals of $G(\mu)$. For the time being, we can assume that the S nodes in $T(v)$ and $T - T(v)$ are manipulated separately by the algorithm since any two tree-structures generated for an S node in $T(v)$ and another S node in $T - T(v)$ cannot correspond to the same edge. Now, both $T(v)$ and $T - T(v)$ contain less than k S nodes. By the induction hypothesis, our algorithm can construct a canonical tree T_1 of $G(v)$ and a canonical tree T_2 of $G - G(v) + (a, b)$. Note that T_1 must correspond to edge (a, b) and that T_2 has a leaf representing (a, b) . Therefore, after execution of Step 3-2, a canonical tree T' of G will be generated by replacing (a, b) in T_2 with T_1 . ■

To achieve efficient parallel implementation of this algorithm, we first transform the given 2-tree (an SP graph) into a directed SP (DSP) graph. The definition of DSP graphs is similar to that of SP graphs except that two terminals of each DSP are considered to be ordered as left and right terminals; the left and right terminals of the primitive DSP $(\{a, b\}, \{a \rightarrow b\})$ are a and b , respectively; the series composition of two DSP graphs is operated by identifying the left terminal of one graph with the right of the other; the parallel composition is operated by identifying the left (respectively, right) terminal of one graph with the left (respectively, right) of the other. The definition of decomposition trees of DSP graphs is similar to that of SP graphs. The transformation can be done by using the algorithm presented in [12] to compute an s - t numbering: First, select the end vertices of an arbitrary edge of G as s and t , assign an s - t numbering ρ to G , and then transform G into a directed graph by replacing each edge (a, b) with arc $a \rightarrow b$ if $\rho(a) < \rho(b)$. Let $\vec{G}_\rho$ denote such a directed graph. The transformation takes $O(\log n)$ time with $C(m, n)$ processors on a CRCW PRAM. He [17] has shown that $\vec{G}_\rho$ is a DSP graph. From now on, we sometimes may regard the given 2-tree (or partial 2-tree in the following sections) as such a DSP graph if an s - t numbering is given, and we may use the notation G instead of $\vec{G}_\rho$ for convenience whenever no ambiguity occurs.

Eppstein [11] developed a parallel algorithm for recognizing DSP graphs and constructing their decomposition trees in $O(\log n)$ time with $C(m, n)$ processors on a CRCW PRAM. We utilize Eppstein's algorithm to construct a decomposition tree T for the given "directed" 2-tree and use T as the input decomposition tree of our algorithm. The constructed tree T has a property which is crucial for implementing Step 1: *For each internal node v of T with two children μ and ω , the left (respectively, right) terminal of $G(v)$ is the minimum (respectively, maximum) s - t number of terminals of $G(\mu)$ and $G(\omega)$.* (This property is obvious since the DSP is constructed according to the s - t numbering.)

In the following, we will first describe a bottom-up sequential algorithm to implement Step 1 and then describe its parallel implementation. Suppose that T is a decomposition tree of a given 2-tree G . Initially, we maintain an ordered pair $\langle a, b \rangle$ for each leaf of T , where a and b are the s - t numbers of the left and right terminals of a primitive 2-tree, respectively (thus, $a < b$). The algorithm processes T

from the leaves to the root γ level by level: For each internal node v which has two children μ and $sib(\mu)$ associated with the ordered pairs $\langle a, b \rangle$ and $\langle c, d \rangle$, respectively, maintain an ordered pair $\langle \min(a, c), \max(b, d) \rangle$ for v . We repeatedly execute the above scheme until the root γ is processed. Thus, for each node v , the two terminals (and the joint-vertex as well) of $G(v)$ can be obtained from the above algorithm. The correctness of the above algorithm can be easily proved by induction on the number of vertices of G .

To implement in parallel the above computation, we initially maintain an ordered pair $\langle a, b \rangle$ for each leaf of T and assign, for each internal node v of T , an ordered pair $\langle \infty, -\infty \rangle$. During execution of tree contraction for contracting some leaf v and $par(v) = \mu$ (i.e., a prune operation on v , followed by a bypass operation on μ), we assume that $\langle a, b \rangle$, $\langle e, f \rangle$, and $\langle c, d \rangle$ are ordered pairs of μ , v , and $sib(v)$, respectively. After executing the contraction, the ordered pair $\langle \min(\min(a, e), c), \max(\max(b, f), d) \rangle$ is maintained for $sib(v)$. It is easy to verify that the above computation satisfies the conditions of applying tree contraction (i.e., the binary operators $\min$ and $\max$ are associative [18]). After executing the above algorithm, the root γ of T will be associated with an ordered pair containing two elements, which are the two terminals of $G(\gamma) = G$. By modifying slightly the tree contraction algorithm, the desired ordered pair for each internal node of T can be computed without increasing the time-processor complexity of the algorithm [1]. Hence, Step 1 can be done correctly in $O(\log n)$ time with $O(n/\log n)$ processors on an EREW PRAM by Lemma 6.

Clearly, Step 2 can be done in $O(1)$ time with $O(k)$ processors, where k is the number of S nodes in T . This can be simulated in $O(\log k)$ time with $O(k/\log k)$ processors based on Brent's theorem [8]. Throughout this paper, in all the implementations which take $O(1)$ time with $O(k)$ processors, Brent's theorem can be applied to achieve the above time-processor complexity.

As for the implementation of Step 3-1, we first remove leaves and those edges (μ, v) from T where μ is a child of v and v is an S node. Let $\tilde{T}$ be the resulting forest. According to Lemma 7 and the proof in Lemma 9, we can see that the nodes of each tree in $\tilde{T}$ are exactly those nodes having the same terminal pair. For each tree in $\tilde{T}$, we can assign an order to the S nodes of the tree by first finding an order (any order such as preorder or postorder) of the nodes in the tree and then using a prefix-sums computation. Both the ordering operation and the prefix-sums computation can be solved optimally [18]. With these orders, Step 3-1 can be implemented in $O(1)$ time with $O(m)$ processors. Hence, the bottleneck is the operation of finding components of $\tilde{T}$. Step 3-2 can be implemented in $O(1)$ time with $O(m)$ processors. Combining these results, we have the following theorem.

THEOREM 2. *The algorithm **Construct_a_Canonical_Tree** can be implemented in $O(\log n)$ time with $C(m, n)$ processors on a CRCW PRAM.*

COROLLARY 1. *A 2-tree G can be correctly recognized in $O(\log n)$ time with $C(m, n)$ processors on a CRCW PRAM. Moreover, its perfect elimination ordering can be computed within the same bounds.*

Proof. Based on the algorithm presented in [11] and the property that each 2-tree contains exactly $2n - 3$ edges [14], we can easily recognize a 2-tree by checking whether an input graph is a biconnected partial 2-tree with $2n - 3$ edges. The time-processor complexity is bounded by the recognition algorithm in [11], which takes $O(\log n)$ time with $C(m, n)$ processors on a CRCW PRAM. Moreover, after executing the recognition algorithm, a decomposition tree T of G can be constructed. By implementing Step 1 of the algorithm **Construct_a_Canonical_Tree**, we can compute, for each S node of T , its joint-vertex. Then, a perfect elimination ordering of G can be computed in the following way: First, traverse T in postorder and assign to each node of T a postorder number. Then, order the joint-vertices of the S nodes in T from 1 to $n - 2$ according to their post-ordered numbers. Finally, after assigning arbitrarily two numbers $n - 1$ and n to the remaining unordered vertices in G , we have a PEO of G . (Note that each vertex is the joint-vertex of some S node in T except for the two terminals u and v of G , where (u, v) is an edge of G .) The correctness can be easily shown by induction on the number of S nodes of T . Clearly, the above method can be implemented by using the Euler-Tour technique and parallel prefix-sums computation [18]. ■

Adhar and Peng [2] presented a parallel algorithm for k -tree recognition by using Klein's algorithm [19] to compute a perfect elimination order for chordal graphs (note that each k -tree is chordal) [19]. It takes $O(\log^2 n)$ time with $O(m + n)$ processors on a CRCW PRAM. Hence, the above result shows that the recognition and the computation of a PEO for k -trees can be improved when $k = 2$.

3.3. Rules for Computing $R(G_K)$

From the sequential algorithm described at the beginning of Section 3, we observe that each undirected edge (x, z) is initially associated with two probability measures for two arcs $x \rightarrow z$ and $z \rightarrow x$. During each removal of degree-2 vertex y of a triangle $\{x, y, z\}$, the corresponding probability measures are summarized for arcs $x \rightarrow z$ and $z \rightarrow x$. As a matter of fact, we can in advance orient the edges of G in some way to make G become a directed graph $\vec{G}$. Then, $\vec{G}$ can be reduced to one arc by repeatedly removing a degree-2 vertex (in-degree = out-degree = 1) y of a triangle $\{x, y, z\}$ (with the arcs $x \rightarrow y$, $y \rightarrow z$, and $x \rightarrow z$) until one arc remains. Hence, for each edge of G , we may summarize its probability measures only once, which reduces by half the number of computations of the sequential algorithm. Given an undirected 2-tree G , such an orientation can be achieved by using an s - t numbering. (We have described the orientation in the implementation of the algorithm **Construct_a_Canonical_Tree**.) The correctness of the above method can be easily verified by induction on the number of vertices of G .

Once a canonical tree T is constructed, we can compute $R(G_K)$ of a 2-tree by dividing each 2-tree reduction into one series reduction, followed by one parallel reduction. When a series reduction is applied to two edges, $e_1 = x \rightarrow y$ and $e_2 = y \rightarrow z$, to form the edge $e_3 = x \rightarrow z$, we use Rule A (described below) to summarize the probability measures (containing eight terms which are derived from the

sequential algorithm in [26]) in e_3 . When a parallel reduction is applied to two edges $e_1 = e_2 = x \rightarrow y$, we use Rule B to summarize the probability measures for the reduced edge $x \rightarrow y$.

Let $\{x, y, z\}$ be a triangle and y be a vertex of degree two, and let $L = x \rightarrow y$ and $R = y \rightarrow z$. The initial measures for each arc $e = a \rightarrow b$ are defined as follows: $sc^1 = \text{Pro}(e)$; $sc^2 = 0$; $dc = 1 - \text{Pro}(e)$; $yn = 0$ if $b \in K$ or 1 otherwise; $ny = 0$ if $a \in K$ or 1 otherwise; $nn^1 = nn^2 = 0$ and $none = 0$ if $a \in K$ or $b \in K$, 1 otherwise. Now, we can formalize both rules as follows:

Rule A:

1. $sc^1(m^*) = sc^1(L) \cdot sc^1(R) + sc^1(L) \cdot dc(R) + dc(L) \cdot sc^1(R) + yn(L) \cdot ny(R)$
2. $sc^2(m^*) = sc^1(L) \cdot sc^1(R)$
3. $dc(m^*) = sc^1(L) \cdot dc(R) + dc(L) \cdot sc^1(R) + yn(L) \cdot ny(R)$
4. $yn(m^*) = sc^1(L) \cdot yn(R) + yn(L) \cdot none(R)$
5. $ny(m^*) = ny(L) \cdot sc^1(R) + none(L) \cdot ny(R)$
6. $nn^1(m^*) = none(L) \cdot none(R)$
7. $nn^2(m^*) = nn^2(L) \cdot none(R) + none(L) \cdot nn^2(R) + ny(L) \cdot yn(R)$
8. $none(m^*) = none(L) \cdot none(R)$,

where $m^* = x \rightarrow z$ is the edge formed by applying a series reduction to edges L and R .

Rule B:

1. $sc^1(M) = sc^1(M) \cdot sc^1(m^*) + dc(M) \cdot sc^2(m^*)$
2. $sc^2(M) = 0$
3. $dc(M) = dc(M) \cdot dc(m^*)$
4. $yn(M) = yn(M) \cdot yn(m^*)$
5. $ny(M) = ny(M) \cdot ny(m^*)$
6. $nn^1(M) = 0$
7. $nn^2(M) = nn^2(M) \cdot nn^1(m^*) + none(M) \cdot nn^2(m^*)$
8. $none(M) = none(M) \cdot none(m^*)$,

where M is an edge of G or an edge formed by applying a parallel reduction, and m^* is an edge formed by applying a series reduction.

Finally, when the given 2-tree has been reduced to an edge e , $R(G_K)$ is taken to be $sc^1(e) + yn(e) + ny(e) + nn^2(e)$. A bottom-up sequential implementation for computing $R(G_K)$ by applying rules A and B on a canonical tree of G is obvious. Detailed parallel implementation for computing $R(G_K)$ by means of tree contraction is described in the Appendix, and we have the following result:

THEOREM 3. *$R(G_K)$ in 2-trees can be solved in $O(\log n)$ time with $C(m, n)$ processors on a CRCW PRAM.*

4. A PARALLEL STRATEGY FOR COMPUTING $R(G_k)$ IN PARTIAL 2-TREES

In this section, we will extend the result described in Section 3 to solve the K -terminal reliability problem in partial 2-trees. The strategy is to first transform a given connected partial 2-tree into a biconnected one and to further transform it into a 2-tree. Thus, the algorithm we have developed in Section 3 can be applied to the resulting 2-tree.

4.1. Transform a Connected Partial 2-Tree into a Biconnected One

In this subsection, we will present a parallel algorithm modified from the sequential algorithm in [26] to transform a connected partial 2-tree into a biconnected one by adding dummy edges each with a probability measure of zero. Before presenting the algorithm, we need the following definitions.

Suppose that $a_1, a_2, \dots, a_k$ are the cut vertices of G and that $B_1, B_2, \dots, B_l$ are the blocks of G . The *block tree* BT [17] is defined as follows. The vertex set of BT is $a_1, a_2, \dots, a_k, b_1, b_2, \dots, b_l$, and (a_i, b_j) is an edge of BT if and only if a_i is a vertex of B_j . In addition, we call each b_i a *block-vertex* of BT. It can be shown that BT is a tree [3]. Let BT' be the rooted tree formed by selecting one block-vertex b_r of BT as the root and assigning directions to the edges. 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 when we traverse path P from b_i , we call B_j (respectively, B_i) the *parent block* (respectively, a *child block*) of B_i (respectively, B_j). In particular, $par(B_i)$ denotes the parent block of B_i and we call the block with no child block the *leaf block*.

ALGORITHM BECOME_BICONNECTED

Input: A connected partial 2-tree.

Output: A biconnected partial 2-tree.

Step 1. Find the blocks $B_1, B_2, \dots, B_k$ of a connected partial 2-tree G .

Step 2. /* Prepare to add dummy edges */

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

2-2. Transform BT into a rooted tree BT' by selecting an arbitrary block-vertex b_r as the root.

/* Steps 2-3 and 3 are executed in parallel for each block B_i ($i = 1, \dots, k$). */

2-3. Find the parent block of B_i using BT'.

Step 3. Arbitrarily select one edge $e = (v, w)$ of B_i and an edge (u, v) of $par(B_i)$, where v is the cut vertex connecting B_i to $par(B_i)$. Then, add a dummy edge (w, u) to G with probability zero.

The resulting graph is a biconnected partial 2-tree since it contains no cut vertex and no homeomorph of K_4 is introduced after adding the dummy edges.

The processor-time complexity of the algorithm is analyzed as follows. In Step 1, finding the blocks of G takes $O(\log n)$ time with $C(m, n)$ processors on a CRCW PRAM [12, 24]. In Step 2-1, the block tree BT can be constructed in $O(1)$ time with $O(m)$ processors [17]. Step 2-2 constructs the rooted tree BT' from BT by using the Euler-Tour technique described in [18]. It takes $O(\log k)$ time with $O(k/\log k)$ 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 ($1 \leq i \leq k$) can be found by using BT' since $b_j = \text{par}(\text{par}(b_i))$. This can be done in constant time with $O(k)$ processors. In Step 3, the cut vertex v in B_i connecting to $\text{par}(B_i)$ can be determined in $O(1)$ time. Thus, the selection of some edge of B_i ($1 \leq i \leq k$) and addition of dummy edges can be done in constant time with $O(k)$ processors. Hence, we have the following theorem.

THEOREM 4. *Algorithm **Become_Biconnected** can correctly transform a connected partial 2-tree G into a biconnected partial 2-tree in $O(\log n)$ time with $C(m, n)$ processors on a CRCW PRAM.*

4.2. Transform a Biconnected Partial 2-Tree into a 2-Tree

In this subsection, we will present a parallel algorithm to add dummy edges each with probability measure of zero on a given biconnected partial 2-tree G to transform it into a 2-tree.

ALGORITHM BECOME_2-TREE

Input: A biconnected partial 2-tree G with its decomposition tree T .

Output: A 2-tree G' with its decomposition tree T' .

Step 1. Duplicate T to obtain another decomposition tree T' .

/ Initially, $T' = T$. After execution of the algorithm, T' is a decomposition tree of a 2-tree G' transformed from G . */*

Step 2. For each S node v of T find, in parallel, the two terminals of $G(v)$.

Step 3. For each group of S nodes having the same pair of terminals, arbitrarily mark an S node in that group.

Step 4. For each marked S node v with terminals a and b , if (a, b) is not an edge, add it to G and update T' as follows: Let $\mu = \text{par}(v)$. Create a leaf node v_e (corresponding to the edge $e = (a, b)$) and a P node ω , and let v and v_e be two children of ω and $\text{par}(\omega) = \mu$.

The correctness of the above algorithm can be easily shown by induction on the number of S nodes of T . Now, we will analyze the time-processor complexity of the algorithm.

Step 1 is trivial to implement. In Step 2, the complexity of finding two terminals of each S node has been analyzed in the algorithm **Construct_a_Canonical_Tree**. In Step 3, to find the number of S nodes having the same pair of terminals, we can use a similar technique for implementing Step 3-1 of the algorithm **Construct_a_Canonical_Tree**. Hence, this step needs $O(\log n)$ time with $C(m, n)$ processors

on a CRCW PRAM. In Step 4, checking the existence of edges and adding dummy edges to G can be done in $O(1)$ time with $O(k)$ processors on a CREW PRAM by maintaining an additional data structure, where k is the number of S nodes of T . The addition of these dummy edges and dummy P nodes takes $O(1)$ time with $O(k)$ processors on an EREW PRAM. Therefore, we have the following theorem.

THEOREM 5. *The algorithm **Become_2-Tree** can correctly transform a biconnected partial 2-tree into a 2-tree in $O(\log n)$ time with $C(m, n)$ processors on a CRCW PRAM.*

After adding the dummy edges, a given biconnected partial 2-tree can be transformed into a 2-tree. Thus, the algorithm developed in Section 3 can be used to compute $R(G_K)$ for biconnected partial 2-trees, which takes $O(\log n)$ time with $C(m, n)$ processors on a CRCW PRAM.

From the above result and the algorithm developed in Section 4.1, we have the following theorem.

THEOREM 6. *The K -terminal reliability in partial 2-trees can be solved in $O(\log n)$ time with $C(m, n)$ processors on a CRCW PRAM.*

5. FINDING MOST VITAL EDGES WITH RESPECT TO K -TERMINAL RELIABILITY

In this section, we will deal with the problem of finding most vital edges with respect to K -terminal reliability in partial 2-trees. The *most vital edge* is the edge whose deletion yields the largest decrease in K -terminal reliability. Wang and Hsu [28] have presented a linear-time sequential algorithm for finding most vital edges with respect to K -terminal reliability in SP graphs. However, this method seems to be difficult to parallelize to achieve efficient implementation. In the following, we will use a different strategy to develop a parallel algorithm for finding such an edge in partial 2-trees.

Let $Pro(G)$ denote the eight probability measures of G corresponding to sc^1 , sc^2 , dc , yn , ny , nn^1 , nn^2 , and $none$ (refer to section 3.3). For convenience, we let “probability measures” represent the above eight terms. The main idea of the algorithm is to first compute a special form of the probability measure $Pro(G, H)$ for some subgraph $H \subseteq G$, where $Pro(G, H)$ is represented by the function $f: R^8 \mapsto R^8$ such that once $Pro(H)$ is known, the probability measures $Pro(G)$ can be computed immediately by using $Pro(H)$ as input to the function $Pro(G, H)$. Assume that H is the graph obtained by applying some composition to subgraphs H_1 and H_2 of G (for convenience, we denote $H = H_1 \circ H_2$, where $\circ$ may represent a series or a parallel composition). By evaluating $Pro(H_2)$ in advance (based on a method which will be described later), we can further transform the original function $Pro(G, H)$ into another function $Pro(G, H_1)$ with the unknown value $Pro(H_1)$. Then, we can repeatedly execute the above process to make the “unknown” part become smaller until $Pro(G, e)$ (for convenience, we use the term $Pro(G, e)$ instead of $Pro(G, G_e)$, where G_e represents the subgraph spanned by e) is obtained. Therefore, if we let

$Pro(e)=0$ in the computation of $Pro(G, e)$, then $R(G_K - e)$ can be obtained immediately. Now, we will describe in detail our parallel algorithm in four phases.

Phase 1. Summarize the probability measures $Pro(G(v))$ for each internal node v . As described in Section 3, our algorithm computes eight probability measures for the root of T to obtain the final result $R(G_K) = sc^1 + yn + ny + nn^2$. By modifying slightly the tree contraction algorithm, the probability measures for all the nodes of T can be computed without increasing the time-processor complexity of the algorithm; this is achieved by reversing the vertex elimination process in the tree contraction algorithm by using some pointers of the bypassed nodes [1].

Phase 2. Compute $Pro(G(par(v)), G(v))$ for each nonroot node v . In this phase, a copy of $Pro(G(v))$ computed in Phase 1 is transferred to $sib(v)$ for each internal node v (thus, v can obtain $Pro(sib(v))$ from $sib(v)$). Let $a = sc^1, b = sc^2, \dots, h = none$ be the eight probability measures $G(sib(v))$ computed in Phase 1. We make eight variables x_i (for all $1 \leq i \leq 8$) represent eight probability measures, corresponding to $sc^1, sc^2, \dots, none$, of $G(v)$. Let the left and right terminals (refer to the description of the implementation of the algorithm **Construct_a_Canonical_Tree** in Section 3.2) of $G(par(v))$ be q_1 and q_3 , respectively, and let the joint-vertex of $G(par(v))$ be q_2 if $par(v)$ is an S node. We compute $Pro(G(par(v)), G(v))$ as follows.

Case 1. $par(v)$ is an S node, and the two terminals of $G(v)$ are q_1 and q_2 (the case for q_2 and q_3 being that the two terminals of $G(v)$ can be computed similarly). Use Rule A defined in Section 3 to compute eight components of the function $Pro(G(par(v)), G(v))$ corresponding to $sc^1, sc^2, \dots, none$ as follows:

$$f_{sc^1} = (a + c) \cdot x_1 + a \cdot x_3 + e \cdot x_4$$

$$f_{sc^2} = a \cdot x_1$$

$$f_{dc} = c \cdot x_1 + a \cdot x_3 + e \cdot x_4$$

$$f_{yn} = d \cdot x_1 + h \cdot x_4$$

$$f_{ny} = a \cdot x_5 + e \cdot x_8$$

$$f_{nn^1} = h \cdot x_8$$

$$f_{nn^2} = h \cdot x_7 + g \cdot x_8 + d \cdot x_5$$

$$f_{none} = h \cdot x_8.$$

Case 2. $par(v)$ is a P node. Use Rule B defined in Section 3 to compute $Pro(G(par(v)), G(v))$ for v according to the following subcases.

Case 2.1. v is a leaf node or a P node:

$$f_{sc^1} = a \cdot x_1 + b \cdot x_3$$

$$f_{sc^2} = 0$$

$$f_{dc} = c \cdot x_3$$

$$f_{yn} = d \cdot x_4$$

$$f_{ny} = e \cdot x_5$$

$$f_{nn_1} = 0$$

$$f_{nn^2} = f \cdot x_7 + g \cdot x_8$$

$$f_{none} = h \cdot x_8.$$

Case 2.2. v is an S node:

$$f_{sc_1} = a \cdot x_1 + c \cdot x_2$$

$$f_{sc_2} = 0$$

$$f_{dc} = c \cdot x_3$$

$$f_{yn} = d \cdot x_4$$

$$f_{ny} = e \cdot x_5$$

$$f_{nn_1} = 0$$

$$f_{nn^2} = g \cdot x_6 + h \cdot x_7$$

$$f_{none} = h \cdot x_8.$$

Phase 3. Compute $Pro(G, G(v))$ for each nonroot node v of T . In this phase, we make use of the function $Pro(G(par(v)), G(v))$ obtained in Phase 2 to compute $Pro(G, G(v))$. Let $Pro(G(par(v)), G(v))$ and $Pro(G(par(par(v))), G(par(v)))$ be two functions computed in Phase 2 for v and $par(v)$, respectively. Then, the function $Pro(G(par(par(v))), G(v))$ for v can be computed by providing $Pro(G(par(v)), G(v))$ (composed of eight functions corresponding to $sc_1, sc_2, \dots, none$) as an input of $Pro(G(par(par(v))), G(par(v)))$. For the sake of brief description, we call the above operation a substitution of v . It is easy to verify that each of the eight components of the function $Pro(G(par(par(v))), G(v))$ is still a polynomial of eight variables such that the polynomial is expressed as the sum of degree one subpolynomials. In the following, we will apply the two-pass parallel tree contraction technique to compute $Pro(G, G(v))$.

The first pass, namely *forward pass*, is the standard tree contraction process which repeatedly contracts a given binary tree until a three-node tree remains. During the execution of this pass, when some leaf node μ_e and its parent are contracted, we execute a substitution for $sib(\mu_e)$. For convenience, let $sib(\mu_e) = \mu$, $par(\mu_e) = v$, and $par(par(\mu_e)) = \omega$. Through substitution, the function computed for μ is $Pro(G(\omega), G(\mu))$ (see Fig. 3(a)). After the computation of this pass, the resulting tree consists of the root γ with the two children, denoted by v_e and $v_{e'}$, which are two leaves of T , respectively. Moreover, the functions computed for v_e and $v_{e'}$ are $Pro(G, G(v_e)) = Pro(G, e)$, and $Pro(G, e')$.

The second pass, namely the *backward pass*, is the reverse of the tree contraction algorithm; that is, it restores the three-node tree to the original tree T . Since in the forward pass, μ_e is contracted before its parent v , v will be restored before μ_e in the backward pass. During the execution of this pass, when nodes v and μ_e are restored consecutively, we perform the following operation.

Case 1. $\omega \neq \gamma$. We first execute a substitution for v and then execute one for μ_e . Since the function $Pro(G, G(\omega))$ was computed for ω at the time v was restored

FIGURE 3

(this result can be observed from the process of computation), the functions $Pro(G, G(v))$ and $Pro(G, G(\mu_e)) = Pro(G, e)$ are computed for v and μ_e after the above operation, respectively (see Fig. 3(b)).

Case 2. $\omega = \gamma$. We only execute a substitution for μ_e during the process of restoring v and μ_e . Note that the function $Pro(G(\gamma), G(v)) = Pro(G, G(v))$ has been computed for $par(\mu_e)$ in Phase 2. Thus, the function computed for μ_e after this operation is $Pro(G, e)$.

We can show by induction that after T is restored completely, the function $Pro(G, G(v))$ will be computed for each nonroot node v .

Phase 4. Evaluate $R(G_K - e)$ for each edge e . Since $Pro(G, G(v_e))$ for each leaf node v_e of T has been computed in Phase 3, for each v_e representing an edge $e = (a, b)$ of G , we let $Pro(e) = 0$ and set the following values from the equations defined in Section 3 as $sc^1 = Pro(e) = 0$; $sc^2 = 0$; $dc = 1 - Pro(e) = 1$; $yn = 0$ if $b \in K$ or 1 otherwise; $ny = 0$ if $a \in K$ or 1 otherwise; $nn^1 = nn^2 = 0$ and $none = 0$ if $a \in K$ or $b \in K$, 1 otherwise. Then, we use these values to evaluate $Pro(G, e)$ and obtain $R(G_K - e)$ as $sc^1 + yn + ny + nn^2$. Hence, the most vital edge is the edge with the smallest reliability value.

The correctness of the algorithm is straightforward from the computation of those functions described in the previous phases. We will only analyze the time-processor complexity of the algorithm.

Phase 1 takes $O(\log n)$ time with $O(n/\log n)$ processors on an EREW PRAM.

In Phase 2, each internal node v of T sends its label (S or P) to its children. Then, each nonroot node sends its probability measures (computed in Phase 1) to its sibling. Clearly, the computation of this phase can be done in $O(\log n)$ time with $O(n/\log n)$ processors.

In Phase 3, it is not difficult to verify that the operation “substitution” takes constant time with one processor. Since each computation of each component of the function $Pro(G, H)$ remains a sum of degree one polynomials (at most eight terms) during the whole computation, this phase needs $O(\log n)$ time with $O(n/\log n)$ processors on an EREW PRAM by Lemma 6.

In Phase 4, the problem of finding the smallest reliability-function value among all the leaves of T can be easily solved in $O(\log n)$ time with $O(n/\log n)$ processors on a CRCW PRAM [18]. Combining the previous results and this algorithm, we have the following theorem.

THEOREM 7. *The most vital edge with respect to the K -terminal reliability of a partial 2-tree can be found correctly by the algorithm in $O(\log n)$ time with $C(m, n)$ processors on a CRCW PRAM.*

CONCLUSIONS

In this paper, we have developed some parallel algorithms for computing K -terminal reliability in 2-trees and partial 2-trees. Moreover, we have also solved the problem of finding most vital edges in partial 2-trees. One contribution of this paper has been to provide a systematic way to construct a canonical tree for a given 2-tree. In addition to computing K -terminal reliability, there are some other problems on 2-trees, such as finding Steiner trees [27], and finding biconnected Steiner minimal networks [30], which need to summarize some useful information during the process of reducing G by a 2-tree reducing sequences. By providing a canonical tree of a given 2-tree, these problems can be solved using our strategy in a similar way. Moreover, they can also be solved on partial 2-trees by using our transformation method.

APPENDIX

This Appendix describes in detail the implementation of tree contraction in the measurement of $R(G_k)$ in 2-trees.

Given a canonical tree T we initially assign an identity matrix A_v with size 8×8 to each node v of T , and the probability measures represented by a matrix p_{v_e} with size 8×1 to each leaf node v_e representing an edge e of G . The matrices A_v and p_{v_e} are defined as follows:

$$A_v = \begin{pmatrix} sc^1 \\ sc^2 \\ dc \\ yn \\ ny \\ nn^1 \\ nn^2 \\ none \end{pmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$p_{v_e} = \begin{bmatrix} sc^1(e) \\ sc^2(e) \\ dc(e) \\ yn(e) \\ ny(e) \\ nn^1(e) \\ nn^2(e) \\ none(e) \end{bmatrix}$$

Initially, we let $sc^2(e)$ and $nn^1(e)$ be zero and let $sc^1(e) = sc(e)$ and $nn^2(e) = nn(e)$; the other values are the same as in the equations defined in Section 3.

Assume that v_e is some leaf node which will be pruned during execution of the tree contraction algorithm. In the following cases, we will describe the method for reserving useful information on $sib(v_e)$ when v_e and $par(v_e) = \mu$ are contracted. We build a matrix for μ to reserve the probability measures of $G(v_e) = e$, and this matrix will be combined with the matrices of μ and $sib(v_e)$ by means of matrix multiplication.

We will only describe the method for building a matrix A_S for series reduction (i.e., μ is an S node). The matrix A_P for parallel reduction can be obtained in a similar way. Let the left and right terminals of $G(\mu)$ be q_1 and q_3 , respectively, and let the joint-vertex of $G(\mu)$ be q_2 . Assume that the two terminals of $G(v_e)$ are q_1 and q_2 (the case that q_2 and q_3 are the two terminals of $G(v_e)$ can be computed similarly). We build A_S for μ as follows. Let a_i , $1 \leq i \leq 8$, denote the $(1, i)$ entry of A_S , and let $b_i, c_i, \dots, h_i$ denote $(2, i), (3, i), \dots, (8, i)$ entries of A_S , respectively. The elements in A_S are derived from Rule A defined in Section 3. Since

$$\begin{aligned} sc^1(m^*) &= sc^1(L) \cdot sc^1(R) + sc^1(L) \cdot dc(R) + dc(L) \cdot sc^1(R) + yn(L) \cdot ny(R) \\ &= (sc^1(L) + dc(L)) \cdot sc^1(R) + sc^1(L) \cdot dc(R) + yn(L) \cdot ny(R), \end{aligned}$$

we let

$$a_1 = sc^1(L) + dc(L), \quad a_3 = sc^1(L), \quad \text{and} \quad a_5 = yn(L).$$

Similarly, we define the other elements of A_S as follows:

$$sc^2(m^*) = sc^1(L) \cdot sc^1(R) = b_1 \cdot sc^1(R)$$

$$\begin{aligned} dc(m^*) &= sc^1(L) \cdot dc(R) + dc(L) \cdot sc^1(R) + yn(L) \cdot ny(R) \\ &= c_1 \cdot sc^1(R) + c_3 \cdot dc(R) + c_5 \cdot ny(R) \end{aligned}$$

$$yn(m^*) = sc^1(L) \cdot yn(R) + yn(L) \cdot none(R) = d_4 \cdot yn(R) + d_8 \cdot none(R)$$

$$ny(m^*) = ny(L) \cdot sc^1(R) + none(L) \cdot ny(R) = e_1 \cdot sc^1(R) + e_5 \cdot ny(R)$$

$$nn^1(m^*) = none(L) \cdot none(R) = f_8 \cdot none(R)$$

$$\begin{aligned} nn^2(m^*) &= nn^2(L) \cdot none(R) + none(L) \cdot nn^2(R) + ny(L) \cdot yn(R) \\ &= g_4 \cdot yn(R) + g_7 \cdot nn^2(R) + g_8 \cdot none(R) \end{aligned}$$

$$none(m^*) = none(L) \cdot none(R) = h_8 \cdot none(R),$$

where $L = (q_1, q_2)$ and $R = (q_2, q_3)$.

Then, A_S is obtained as follows:

$$A_S = \begin{pmatrix} sc^1 & \begin{bmatrix} a_1 & 0 & a_3 & 0 & a_5 & 0 & 0 & 0 \end{bmatrix} \\ sc^2 & \begin{bmatrix} b_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \\ dc & \begin{bmatrix} c_1 & 0 & c_3 & 0 & c_5 & 0 & 0 & 0 \end{bmatrix} \\ yn & \begin{bmatrix} 0 & 0 & 0 & d_4 & 0 & 0 & 0 & d_8 \end{bmatrix} \\ ny & \begin{bmatrix} e_1 & 0 & 0 & 0 & e_5 & 0 & 0 & 0 \end{bmatrix} \\ nn^1 & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & f_8 \end{bmatrix} \\ nn^2 & \begin{bmatrix} 0 & 0 & 0 & g_4 & 0 & 0 & g_7 & g_8 \end{bmatrix} \\ none & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_8 \end{bmatrix} \end{pmatrix}$$

Now, we will describe the two procedures $Prune(v)$ and $Bypass(\mu)$, which are used in the parallel tree contraction algorithm to compute $R(G_K)$.

Procedure $Prune(v)$

/ compute the matrix A_μ for μ after v is pruned */*

$p_v \leftarrow A_v \cdot p_v$; */* compute the probability measures of $G(v)$ */*

Case the node type of μ

begin

case 1: μ is an S node

 Compute A_S for μ from p_v ;

$A_\mu \leftarrow A_\mu \cdot A_S$;

case 2: μ is a P node

 Compute A_P for μ from p_v ;

$A_\mu \leftarrow A_\mu \cdot A_P$;

end/* end-of-case */

Delete v ;

end.

Procedure *Bypass*(μ)

/* compute the matrix $A_{sib(v)}$ for $sib(v)$ after μ is bypassed */

begin

$A_{sib(v)} \leftarrow A_\mu \cdot A_{sib(v)}$;

$par(sib(v)) \leftarrow par(\mu)$;

 Delete μ ;

end.

Since the procedures *Prune* and *Bypass* can be carried out in constant time with one processor, this algorithm can be executed in $O(\log n)$ time with $O(n/\log n)$ processors on an EREW PRAM by Lemma 6.

REFERENCES

1. K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. Przytycka, A simple parallel tree contraction algorithm, *J. Algorithms* **10** (1989), 287–302.
2. G. S. Adhar and S. Peng, Parallel algorithms for k-tree recognition and its applications. *Proceedings of the 27 annual Hawaii international conference on system science*, IEEE Computer Society, Los Alamitos, CA, 1994, 194–201.
3. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, “The Design and Analysis of Computer Algorithms,” Addison-Wesley, Reading, MA, 1974.
4. S. G. Akl, “The Design and Analysis of Parallel Algorithms,” Prentice-Hall, Englewood Cliffs, NJ, 1989.
5. M. O. Ball, Complexity of network reliability computations, *Networks* **10** (1980), 153–165.
6. M. W. Bern, E. L. Lawler, and A. L. Wong, Linear-time computation of optimal subgraphs of decomposable graphs, *J. Algorithms* **8**, 2 (1987), 216–235.
7. J. A. Bondy and U. S. R. Murty, “Graph Theory with Applications,” North-Holland, New York, 1980.
8. R. Brent, The parallel evaluation of general arithmetic expressions, *J. Assoc. Comput. Mach.* **21**, 2 (1974), 201–206.
9. R. Cole and U. Vishkin, Approximate parallel scheduling, II: applications to logarithmic-time optimal parallel graph algorithms, *Inform. Comput.* **92** (1991), 1–47.
10. R. J. Duffin, Topology of series parallel networks, *J. Math. Anal. Appl.* **10** (1965), 303–318.
11. D. Eppstein, Parallel recognition of series-parallel graphs, *Inform. Comput.* **98** (1992), 41–55.

12. D. Fussell, V. Ramachandran, and R. Thurimella, Finding triconnected components by local replacement, *SIAM J. Comput.* **22**, 3 (1993), 587–616.
13. M. R. Garey and D. S. Johnson, “Computers and Intractability: A Guide to the Theory of NP Completeness,” Freeman, San Francisco, 1979.
14. D. Garnot and D. Skorin-Kapov, NC algorithms for recognizing partial 2-trees and 3-trees, *SIAM J. Discrete Math.* **4**, 3 (1991), 342–354.
15. M. C. Golumbic, “Algorithmic Graph Theory and Perfect Graphs,” Academic Press, San Diego, 1980.
16. E. Hare, S. Hedetniemi, R. Laskar, K. Peters, and T. Wimer, Linear-time computability of combinatorial problems on generalized series-parallel graphs, in “Discrete Algorithms and Complexity,” 437–455, Academic Press, San Diego, 1987.
17. X. He, Efficient parallel algorithms for series parallel graphs, *J. Algorithms* **12** (1991), 409–430.
18. J. JáJá, “An Introduction to Parallel Algorithms,” Addison–Wesley, Reading, MA, 1992.
19. P. N. Klein, Efficient parallel algorithms for chordal graphs, *SIAM J. Comput.* **25**, 4 (1996), 797–827.
20. T. Politof and A. Satyanarayana, Efficient algorithms for reliability analysis of planar networks—survey, *IEEE Trans. on Reliability* **35**, 3 (1986), 252–258.
21. J. S. Provan and M. O. Ball, The complexity of counting and of computing the probability that a graph is connected, *SIAM J. Comput.* **12** (1983), 777–788.
22. J. S. Provan, The complexity of reliability computations in planar and acyclic graphs, *SIAM J. Comput.* **15** (1986), 694–702.
23. A. Satyanarayana and R. K. Wood, A linear-time algorithm for computing K-terminal reliability in serial-parallel networks, *SIAM J. Comput.* **14** (1985), 818–832.
24. R. E. Tarjan and U. Viskin, An efficient parallel biconnectivity algorithm, *SIAM J. Comput.* **14**, 862–874.
25. L. G. Valiant, The complexity of enumeration and reliability problems, *SIAM J. Comput.* **8** (1979), 410–421.
26. J. A. Wald and C. J. Colbourn, Steiner trees in probabilistic networks, *Microelectron. Reliability* **23**, 5 (1983), 837–840.
27. J. A. Wald and C. J. Colbourn, Steiner trees, partial 2-trees and minimum IFI networks, *Networks* **13** (1983), 159–167.
28. P. F. Wang and L. H. Hsu, Finding the most vital edge with respect to K-terminal reliability in series-parallel networks, *Comput. and Indust. Engrg.* **25**, (1993), 457–460.
29. T. V. Wimer and S. T. Hedetniemi, K-terminal recursive families of graphs, *Congr. Numer.* **63** (1988), 161–176.
30. P. Winter, Generalized Steiner problem in series-parallel networks, *J. Algorithms* **7** (1986), 549–566.