Parallel algorithms for distributed systems and software engineering

Hu, Jie, Ph.D.

The University of Texas at Dallas, 1992

PARALLEL ALGORITHMS FOR DISTRIBUTED SYSTEMS AND SOFTWARE ENGINEERING

by

JIE HU, B.S., M.S.

DISSERTATION

Presented to the Graduate School of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

August 1992

PARALLEL ALGORITHMS FOR DISTRIBUTED SYSTEMS AND SOFTWARE ENGINEERING

APPROVED BY SUPERVISORY COMMITTEE

Eliezer Dekel

Dr. Eliezer Dekel, Chairman

[illegible]

Dr. Larry Ammann

[illegible]

Dr. Ivor Page

William Pervin

Dr. William Pervin

[illegible]

Dr. Haim Schweitzer

Copyright Disclaimer

THE UNIVERSITY OF TEXAS AT DALLAS

CERTIFICATE

I hereby certify that any extensive copyrighted material which I have utilized in the manuscript of my dissertation is with the written permission of the copyright owner. I hereby agree to indemnify and save harmless The University of Texas at Dallas from any and all claims which may be asserted or which may arise from any copyright violation.

[illegible]	8-24-92
Signature	Date

This dissertation is dedicated to

my wife, Weiping Lu,

my daughter, May Hu

and

my parents.

ACKNOWLEDGMENTS

I wish to sincerely thank my advisor, Eliezer Dekel, Professor in the Computer Science Program, for his guidance and encouragement. He is truly a good teacher who has had the greatest influence on me.

I also wish to thank the members of my committee, Professors Larry Ammann, Ivor Page, William Pervin, and Haim Schweitzer for their time, suggestions, and support.

As always, I am very grateful to my family. Specifically, I thank my wife, Weiping Lu and my daughter, May Hu, for their love, encouragement and endurance. I thank my parents, my sister, my brother, and my in-laws for their continuing support and unlimited careness and love. I also thank the friends of my family, Ban and Jan Capron, Dennis and Pat Bull, Jim and Judy Wimberley for their friendship, compassion, and prayers.

Finally, I would like to thank my friends Yuval Caspi, Jesse Chen, Mohammad Heydari, Tony Juang, Wen Ouyang, and Lu Tian for their concern and friendship.

June, 1992

Parallel Algorithms for Distributed Systems

and Software Engineering

Publication No.____________________

Jie Hu, Ph.D.

The University of Texas at Dallas, 1992

Supervisor Professor: Eliezer Dekel

Networks, database systems, computer processes and programs are often presented by graphs. Parallel methods for solving graph problems provide the means of parallel processing for distributed systems and software engineering.

In this dissertation we present a new parallel technique for graph decomposition, *pruning decomposition*, which partitions a graph into certain disjoint structures. Using the pruning decomposition, we introduce the efficient methods for computing st-numbering, finding biconnected components and ear decomposition on the EREW P-RAM model of computation.

Based on these results, we give some efficient parallel algorithms for other problems such as vertex location trees, strong orientation, and minimum cutset on reducible graphs.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

Distributed systems, such as communication networks, database systems, etc. are often presented by graphs. In software development, processes and programs are represented by directed graphs so that evaluation, testing, verification and compilation can be performed effectively. The vertices can indicate process units, the edges (with weight or direction) may present the relation among processes. Therefore methods for solving graph problems provide very useful tools for related transactions or operations of the distributed systems and software engineering.

To fulfill the rapidly increasing requirements of fast and massive computations, parallel processing is becoming a dominant theme in many areas. Different parallel computers have been built and are being built. Of course, some parallel computational models are proposed in order to study the logical structure of parallel computations. Among the theoretical parallel computation models the parallel random-access machine (P-RAM) has been proven to be an extremely useful model to pursue this study. It is assumed in P-RAM, in addition to the private memories of processors, there is a shared memory and the cells of which can be accessed by each processor in unit time.

In the P-RAM model there is the possibility of read- and write-conflicts, in which two or more processors try to read from or write into the same memory cell simultaneously. Distinctions in the way these conflicts are handled lead to several different variants of the model. The weakest of these is the exclusive-read exclusive-write (EREW) P-RAM, in which concurrent reading and writing are forbidden. The strongest one is the concurrent-read concurrent-write (CRCW) P-RAM, in which more than one processor can read from or write into a same memory cell concurrently with certain conventions, such as only one value can be written or priorities are assign to the processors. The intermediate one is concurrent-read exclusive-write (CREW) P-RAM.

Theoretically the EREW P-RAM model can be simulated by CRCW model by time increasing by a O(log$p$) factor, where p is the number of processors, by sorting the processors whenever the memory accessing conflicts occur [V1][KR][Co]. Nevertheless, developing EREW algorithms is of practical interest because the inefficiency and large overhead of this kind of simulation. In addition, it is shown that EREW P-RAM can simulate most of the existing parallel computer structures [HB] and is considered to be the one that is closest to the existing parallel computer systems.

The P-RAM models lead the classification of problems according to their difficulty. The complexity for parallel computation has to include computation time and number of processors. For example, *NC* problems are those problems which can be solved in polylogrithmic time using polynomial number of processors based on the size of the problems. Similarly, an *NC* algorithm indicates a parallel algorithm which can be performed in $O(\log^k N)$ time using $O(N^c)$ processors, where k,c are constants and N is the size of the problem. We refer the readers to the review by R. M. Karp and V. Ramachandran [KP] for more details of the parallel computation models and complexity of parallel algorithms.

In this dissertation, we introduce a new technique, pruning decomposition search (PDS), which partitions a graph into a set of subgraphs so as to provide the tools for parallel processing. Using PDS, some basic graph problems such as st-number numbering, biconnected components and ear decomposition can be solved on EREW P-RAM efficiently with very small overhead and simple data structures. Based on those solutions, we present efficient parallel algorithms for several other graph problems directly related to distributed systems, such as bipartitioning of biconnected graphs, centroided tree construction, centered tree construction, strong orientation, building biconnected certificate and finding minimum cutset for reducible graphs.

In the following we introduce some preliminaries that will be used in most of the following chapters. More definitions will be introduced in the chapters in which the related problems are discussed.

A *graph* $G=(V,E)$ is a structure which consists of a set of *vertices* $V=\{v_1, v_2,..., v_N\}$ and a set of *edges* $E=\{e_1, e_2,..., e_M\}$. Each edge e is incident to a pair of vertices (u,v) which are not necessarily distinct. If (u,v) is ordered then G is a *directed* graph, otherwise G is an *undirected* graph. To facilitate the following discussion, we use "graph" to indicate undirected graph and assume $|V|=N$ and $|E|=M$.

If edge $e=(u,v)$ then u,v are *endpoints* of e. The degree of a vertex u is the number of times u is used as an endpoint.

A *path* is a sequence of edges $e_1, e_2,..., e_k$ s.t. e_i shares one of its endpoints with e_{i-1} and the another with e_{i+1} for $i=2,...,k-1$. A graph $G=(V,E)$ is *connected* if there is a path between any pair of vertices of G.

A vertex is an *articulation point* of G if by removing it G will be disconnected. An edge is a *bridge* of G if by removing it G will be disconnected. A connected graph is *biconnected* if there is no articulation point in it. A connected graph is 2*–edge connected*, or *bridgeless*, if there is no bridge in it.

A graph $G=(V,E)$ is a *tree* if G is connected and $|E|=|V|-1$. A vertex u of a tree is a *leaf* if the degree of u is one, i.e., there is only one edge incident to it.

A graph $G'=(V',E')$ is a *subgraph* of $G=(V,E)$ if $V'\subseteq V$ and $E'\subseteq E$. A subgraph of G which contains all of its vertices and is a tree is called a *spanning tree* of G.

We refer the readers to the book "Graph Algorithms" by S. Even [E] for more graph concepts and definitions.

This dissertation is organized as following:

In Chapter 2 we introduce the pruning decomposition search (PDS). The decomposition can be achieved in $O(\log^2 N)$ time using $N^2/\log N$ processors on an EREW P-RAM. We also present a modified version of PDS in which the spanning tree construction is the only step that needs $O(\log^2 N)$ time, while all the other steps run in $O(\log N)$ time. This implies for graphs in which the spanning trees can be found efficiently, the PDS has even better performance. The PDS is applied for solving most of the problems in this dissertation.

In Chapter 3 we present an efficient EREW algorithm for finding biconnected components of graphs in $O(\log N)$ time using $N^2/\log N$ processors based on the information from the PDS. The 2-edge connected components can be found efficiently as well using the similar method.

In Chapter 4 we give a new method for finding biconnected certificates based on NF-trees. Then we derive an efficient EREW algorithm for ear decomposition in $O(\log N)$ time using $N^2/\log N$ processors based on the information from the PDS.

In Chapter 5 we first introduce an EREW algorithm for finding an st-numbering for biconnected graphs which can be performed in O(log$N$) time using N processors based on the results of the ear decomposition. From the obtained st-numbering we describe efficient parallel algorithms for biconnected graph bipartitioning, centroided tree and centered tree constructions and strong orientations on P-RAM.

In Chapter 6 we present an EREW algorithm for finding a minimum cutset of reducible graphs. The pruning decomposition search is applied to directed graphs here. A heuristic for finding a minimal cutset of general graphs is introduced too. They can be found in O($\log^3 N$) time using O($N^3/\log N$) processors on an EREW P-RAM.

CHAPTER 2

PRUNING DECOMPOSITION SEARCH

2.1 Introduction

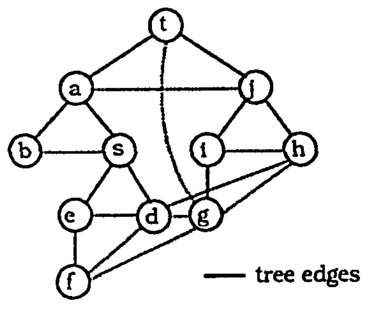
Problem decomposition is a key to several algorithmic techniques. Methods of decomposition have the same basic feature. They break apart a complex structure of a given combinatorial object into smaller and simpler components. These simpler components are then processed in parallel to produce an efficient solution to the problem. Considerable effort in developing decomposition methods is reported in the literature. Many graph decompositions have been introduced for different types of graphs on different models of parallel machines. For example, ear decomposition is for biconnected graphs on CRCW or CREW P-RAM [MSV], tree contraction [GMT][DNP][KR] and the centoid decomposition [CV2] are for tree decompositions on EREW P-RAM.

In this chapter we introduce *pruning decomposition*, a new graph decomposition, for general graphs on EREW P-RAM. Pruning decomposition partitions a graph of N vertices into $K \leq \log N$ auxiliary graphs. Each of these auxiliary graphs consists of some simple structured components. The iterative nature of the pruning decomposition has the flavor of a general search technique in graphs (undirect and directed). It arranges the vertices of the graph by partitioning them into ordered sets of "chains". In this context we refer to the technique as *pruning decomposition search* (PDS). We demonstrate the utility of the PDS by developing several efficient EREW algorithms. Some of them are finding biconnected components of graphs[DH1], ear decomposition and st-numbering for biconnected graphs[DH3][DH4] and minimum cutset for reducible graphs [DH5]. These will be presented in the following chapters of this dissertation.

In the following, section 2.2 we introduce the pruning decomposition and describe the structure of the auxiliary graphs. We also present the parallel implementation of pruning decomposition. In section 2.3 we modify the pruning decomposition such that all the steps can be performed in O(log$N$) time using $N^2/\log N$ processors except the construction of the spanning tree. This modified version can be applied to problems for which the spanning tree can be efficiently found on the EREW P-RAM.

2.2 Pruning Decomposition of Graphs

As we mentioned in the introduction, decomposition methods partition a problem so as to handle it in parallel. The pruning decomposition partitions the graph into *branches* (to be defined). It consists of three steps. In the following, we give the description of these three steps, and then present our parallel decomposition algorithm.



$$\begin{array}{c|ccccccccccc}
 & a & b & d & e & f & g & h & i & j & s & t \\
\hline
a & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
b & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
d & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\
e & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
f & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
g & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
h & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\
i & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
j & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
s & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
t & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0
\end{array}$$

Figure 2.2.1 Graph G, spanning tree T and matrix M

Consider a rooted spanning tree T of an undirected graph G (Fig. 2.2.1). We define an *active chain* to be the longest subpath on the path from a leaf vertex to the root of T, starting from the leaf and including only vertices with degree two or less (the root of the tree is included only if its degree is one or less). The *root of the chain* is the end of the chain that is not the leaf. For a given tree *T*, we can find *m* active chains if there are *m* leaves. The first step of the decomposition is to move those active chains to an auxiliary graph, namely, "prune". The remaining tree has some new leaves generated, from which the new active chains can be found and moved to another auxiliary graph. We say a vertex is *activated* (hence a *chain vertex*) when it is moved to an auxiliary graph. These iterative steps terminate once the root of the tree becomes active. Each auxiliary graph A_i (generated in the i^{th} iteration) consists of some active chains. Obviously every vertex is activated exactly once. Actually this step partitions *T* into several sets of chains.

It is instructive to follow the decomposition with an example. Figure 2.2.1 gives an undirected graph *G*, its spanning tree *T* with root *t* and its adjacency matrix *M*. Figure 2.2.2 shows the results of the first step: auxiliary graph A_1 with five chains, A_2 with two chains $a-s, j$ and A_3 with one chain *t*.

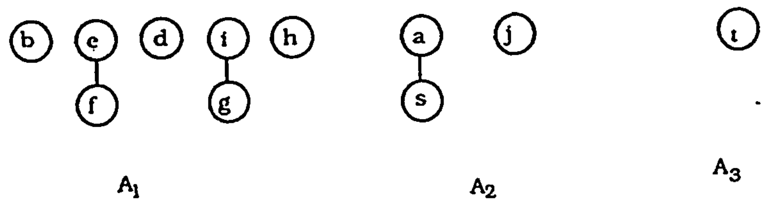


Figure 2.2.2 Auxiliary graphs of G after the first step

In the second step we augment the information in the auxiliary graphs by adding "super leaves" (to be defined) to the chains. Let *v* be a chain vertex and *u* be a child of *v* in the spanning tree *T*. Vertex *u* was a root of an active chain in a previous iteration. We attach $\bar{u}$ to *v* as a *super leaf*. This super leaf represents the vertex set of the subtree of *T* rooted at u. The auxiliary graphs A_i's of our example together with the new super leaves are presented in Fig.2.2.3: super leaves $\bar{b},\bar{e},\bar{d},\bar{i},\bar{h}$ are added into A_2 and $\bar{a},\bar{j}$ are added into A_3. In the following discussion we say that vertex x is a member of $\bar{u}$, $x \in \bar{u}$ (x is represented by $\bar{u}$), if x is a descendant of u in T. Obviously there are no super leaves in A_1.

To complete our pruning decomposition, we need to add the edges among the vertices of every chain in the auxiliary graphs (including chain vertices and super leaves). This is done in the third and last step of the decomposition. We add the following edges:

- Add $(\bar{u},\bar{w})$ if there is a non-tree edge (x,y) of T s.t. $x \in \bar{u}$, $y \in \bar{w}$ and $\bar{u},\bar{w}$ are super leaves of the same chain;
- Add $(\bar{u},v)$ if there is a non-tree edge (x,v) of T s.t. $x \in \bar{u}$ and v is a chain vertex;
- Add (u,v) if u,v are chain vertices and (u,v) is a non-tree edge (i.e., a forward edge of T whose both endpoints are chain vertices of a same chain).

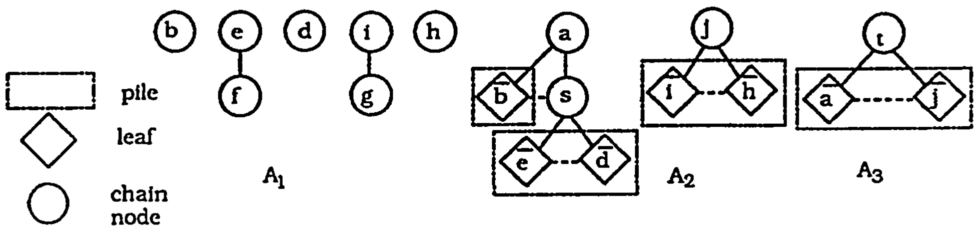


Figure 2.2.3 Auxiliary graphs of G

A chain augmented with super leaves and the above edges is referred to as a *branch*, and the root of the chain is referred to as the *root of branch*. In our discussion, we say that edge (x,y) is represented by edge $(\bar{u},\bar{v})$ if $x \in \bar{u}$ and $y \in \bar{v}$, edge (x,y) is represented by $(x,\bar{u})$ if x is a chain vertex and $y \in \bar{u}$ or (x,y) is represented by itself when x,y are both chain vertices. The branches that are created as a result of the pruning decomposition have enough information in them of the original graph to allow parallel computation. Various operations that are based on the pruning decomposition require some additional information. To that end, we classify the edges of the branch. Consider the relation just among the super leaves of a branch, some of them are connected. We define a *pile* to be a set of super leaves that are connected to each other in a branch. In the following, a pile is denoted by p_u if u is the vertex with the largest label in the pile (assume the vertices are labeled in postorder of the spanning tree T). Let b_w be a branch with root w; a pile p_u of b_w is *local* if there is no edge (x,y) of G s.t. x is in p_u and and y is not in b_w. Otherwise the pile is *non−local*. In our example in A_2 of Fig.2.2.3, p_d consisting of $\bar{e},\bar{d}$ is a non-local pile because $(d,g)\in G$ has $d\in p_d$ but $g\notin b_a$. Clearly, p_b is a local pile of b_a.

Furthermore, the edges of a branch can be partitioned into:

- Chain edges (the endpoints of the edges are chain vertices, e.g., (a,s) of A_2 in Fig.2.2.3);
- Edges between non-local piles and chain vertices (e.g., $(\bar{e},s)$ of A_2 in Fig.2.2.3);
- Edges between local piles and chain vertices (e.g., $(a,\bar{b})$ of A_2 in Fig.2.2.3);

This classification provides more information to the pruning decomposition for various applications. In addition, this decomposition can partition the edges of G into tree edges, forward edges and cross edges according to the spanning tree T. An edge (x,y) is a cross edge iff its representative is either (1) an edge between two chain leaves $\bar{u}$ and $\bar{v}$ s.t. $x\in\bar{u}$, $y\in\bar{v}$, or (2) an edge $(x,\bar{u})$ s.t. x and u are not descendant to each other. Edges that are neither tree edges (part of the spanning tree), nor cross edges are forward edges. Therefore every edge of G can be identified as tree edge, cross edge or forward edge.

This information may allow some other applications for pruning decomposition [DH5]. Before presenting the parallel algorithm for the pruning decomposition, we need to prove some properties of the pruning decomposition.

Lemma 2.2.1: The total number of auxiliary graphs is no more than $\log N$, where N is the number of vertices in G.

Proof: In an arbitrary tree T, if vertex v is the only child of vertex u then we say vertices u,v are in a same chain (this relation is transitive). A chain is *active* if it contains a leaf (it is exactly the definition of active chain in pruning decomposition) and is otherwise *non−active*. In the process of pruning decomposition, vertices on the same chain are on the same active chain in some iteration. Hence representing a non-active chain by one vertex will not change the number of iterations. Reduce T into T' by replacing each chain by a vertex so that every internal vertex of T' has at least two children, the number of leaves (i.e., the active chains) in T' is at least half of the vertices. This implies the total number of iterations is bounded by $\log N$.□

Lemma 2.2.2: Every edge of G is represented in the auxiliary graphs exactly once.

Proof: We prove this lemma by considering the type of edges based on the spanning tree T of G.

Case 1: edge (u,v) is a tree edge of T. Without lost of generality, let v be the child of u in T. If u,v are in the same active chain, then edge (u,v) is a chain edge which is represented by itself in the auxiliary graph where u is a chain vertex. If v is activated before u then v must be a root of a branch, hence $\overline{v}$ is a super leaf where u is a chain vertex. Edge (u,v) is represented only in the auxiliary graph that is created when u is activated.

Case 2: edge (u,v) is a forward edge of T. Without lost of generality, let v be the descendant of u in T. If u,v are in the same active chain, then similar to the case 1, (u,v) is represented by itself when u,v are chain vertices. If v is activated before u, then (u,v) is represented by $(u,\overline{w})$ where w is the child of u and w is an ancestor of v. Edge $(u,\overline{w})$ is in the auxiliary graph where u is a chain vertex. Hence (u,v) is represented only in the auxiliary graph where u is a chain vertex.

Case 3: edge (u,v) is a cross edge of T. We know that the representing edge for (u,v) is added only when u,v belong to the same branch. Let w be the lowest common ancestor of u and v in T. Consider the iteration when the representing edge of (u,v) is added in, if u is a chain vertex then (u,v) is represented by $(u,\overline{x})$ where x is the child of w and x is an ancestor of v. If v is a chain vertex the (u,v) is represented by $(\overline{y},v)$ where y is the child of w and y is an ancestor of u. If none of u,v are chain vertices then (u,v) is represented by $(\overline{x},\overline{y})$ where x,y are children of w, u is a descendant of x and v is a descendant of y. It is impossible for u and v be both chain vertices in a same chain since (u,v) is a cross edge. So (u,v) is represented only when the lowest common ancestor of u,v is activated.

Notice that every vertex of G is activated (i.e., to be a chain vertex) exactly once. Every edge of G is represented exactly once since it is represented only when certain vertex is an activated chain vertex.□

We are now ready to describe our parallel implementation of pruning decomposition, Algorithm 2.2.1. In step 1 of PRUNING_DECOMPOSITION, we construct a spanning tree of the input graph. This can be done in $O(\log^2 N)$ time using $N^2/\log N$ processors on EREW P-RAM [NM]. The postorder of T in step 2 can be computed in O(log$N$) time using N/log$N$ processors on EREW [TV][KR].

Step 4 constructs the auxiliary graphs consisting of chain vertices and super leaves. Let N_i be the number of chain vertices in A_i and S_i be the number of super leaves in A_i. We know $S_i \le N_{i-1}$. Therefore step 4.1.2 can be done in O(logN_i) time using N_i/logN_i processors and step 4.1.3 can be performed in O(1) time using N_{i-1} processors or in O(logN_{i-1}) time using N_{i-1}/logN_{i-1} processors. By Lemma 2.2.1, there are at most log$N$ iterations, step 4.1 runs in O(log^2N) time using N/log$N$ processors on EREW P-RAM. Step 4.2 finds all the representing edges. This is done by using the information in adjacency matrix M. By the definition of the adjacency matrix, element $a(i,j)$ of M indicates there is an edge between vertices i and j in G. We construct a matrix M_1 with element $R(i,j)$ s.t.$R(i,j)$=1 if there exists an element $a(i,x)$=1, where x is a descendant of j in T, i.e., $R(i,j)$ indicates that there is an edge between vertex i and the subtree of T with root j, T_j. The matrix M_1 indicates the relation between every vertex and every subtree of T. We can compute $R(i,j)$ for column i of M on T by Euler Tour technique [TV]. Using our example graph of Fig.2.2.1, computing $R(b,j)$ for column b by T is illustrated in Fig.2.2.4. In the example, $a(b,s)$=1 and $a(b,a)$=1 in M, then $R(b,s)$=1, $R(b,a)$=1 and $R(b,t)$=1 This can be done in O(log$N$) time using N/log$N$ processors on EREW P-RAM using Euler Tour technique [TV]. To obtain $R(i,j)$'s for all the columns of M_1 we need N^2/log$N$ processors. We perform the similar computations for each row of M_1 and keep the results in M_2 in which element $R(i,j)$=1 means there is an edge between vertices of T_i and T_j. Now we can add representing edges into auxiliary graphs from the information in M_1 and M_2. Consider a branch of A_i, if i is a chain vertex, $\bar{j}$ is a super leaf and

```
Procedure PRUNING_DECOMPOSITION($G$)
{Input: An undirected graph $G=(V,E)$ represented by an $N \times N$ adjacency matrix M,
$N=|V|$;
 Output: A set of auxiliary graphs $A_i$, $1 \le i \le K$, where $K \le \log N$}
1. Construct a rooted spanning tree $T$ of $G$.   {Any vertex can be the root.}
2. Relabel the vertices by the postorder of $T$ and arrange M according to the new label;
3. { Initialization }
          3.1 $T_{org} \leftarrow T$;
          3.2 $i \leftarrow 0$;
4. { Construct the auxiliary graphs iteratively.}
     4.1 Repeat
          4.1.1 $i \leftarrow i+1$;
          4.1.2 find all the active chains and move them from $T$ to $A_i$;
          4.1.3 add super leaves for each chain of $A_i$;
        until $T$ is empty;
     4.2 Add representing edges for $A_i$'s; {complete branch construction }
     4.3 $K \leftarrow i$;        { K auxiliary graphs are formed}
5. { Find all the piles.}
          5.1 Construct $G_l=(V_l, E_l)$ where
                    $V_l=\{\bar{x} | \bar{x} \in A_i$ for some i};
                    $E_l=\{(\bar{x},\bar{y}) | (\bar{x},\bar{y}) \in A_i$ for some i and $\bar{x},\bar{y} \in V_l\}$;
          5.2 find a spanning forest $F_l$ of $G_l$;
          { Every spanning tree of the super leaves indicates a pile.}
6. for i=1 to K do  {Identify the type of piles and type of edges for each branch.}
     for each branch $b_w$ with root $w$ of $A_i$ in parallel do
          6.1 for each super leaf $\bar{u}$ of $b_w$ in parallel do
               $\bar{u}$ is non-local if there is an edge $(x,y)$ in $G$ s.t. $x \in \bar{u}$ and y is not a des-
               cendant of w in $T_{org}$, otherwise $\bar{u}$ is local;
          6.2 for each pile of $b_w$ in parallel do
               the pile is non-local if there is a $\bar{u}$ in the pile is non-local, otherwise the
               pile is local;
          6.3 classify the edges of the branch;
end;{ PRUNING_DECOMPOSITION }
```

Algorithm 2.2.1. Algorithm for pruning decomposition

$b(i,j)=1$ in M_1, then add representing edge $(i,\bar{j})$ because there is an edge between vertex

i and T_j in G. Similarly, if $\bar{i}$, $\bar{j}$ are super leaves and $c(i,j)$=1 in M_2, add representing edge $(\bar{i},\bar{j})$. So step 4.2 runs in O(log$N$) time using N^2/log$N$ processors and hence step 4 can be done in O(log^2N) time using N^2/log$N$ processors on EREW P-RAM.

Notice that if a super leaf $\bar{u}$ appears in A_i, then u must be the root of a branch in A_{i-1}. This means the chain leaves of all the auxiliary graphs are disjoint and so are the representing edges. There is no conflict in generating a graph G_l that consists only the super leaves of all the auxiliary graphs and the edges among the super leaves in step 5.1 (Fig 2.2.5). In the spanning forest F_l of G_l that is found in step 5.2, every set of connected chain leaves (or every spanning tree in F_l) is a pile. For our example, F_l is the same as G_l, we can see that there are four piles p_b, p_d, p_h and p_j (Fig.2.2.5). The complexity of step 5 is dominated by the spanning forest construction, which can be found in O(log^2N) time using N^2/log$N$ PE's on EREW [NM].

Step 6 has K≤log$N$ iterations. In the i^{th} iteration, 6.1 and 6.2 can be performed based on the matrices M_1 and M_2 Applying step 6 on our example, p_j and p_b are found to be local piles while the rest are non-local piles. This can be done in O(log$N$) time using N^2/log$N$ PE's on EREW P-RAM. As long as we have the information of chain

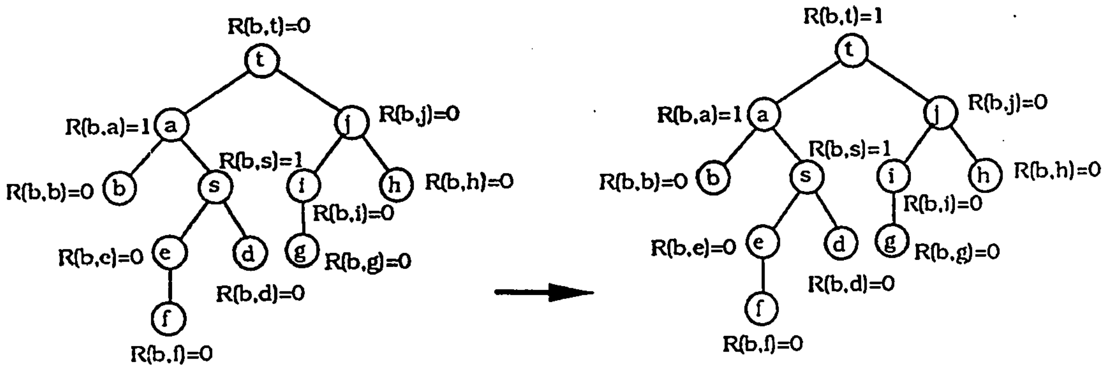


Figure 2.2.4 Computing $R(b,j)$ for column b of M on T

vertices, super leaves, piles (local or non-local) and F_l, the classification of edges can be performed in O(1) time using N^2 PE's or in O(log$N$) time using N^2/log$N$ PE's. Therefore step 6 can be done in O(log$N$) time using N^2/log$N$ PE's on EREW.

In conclusion, the pruning decomposition of a given graph with N vertices can be achieved in O(log^2N) time using N^2/log$N$ PE's on EREW P-RAM based on the tree and matrix data manipulations. This decomposition partitions an undirected graph into a set of auxiliary graphs, each of them consists of some branches which contains information about edges among the set of vertices the branch represents. This information can be utilized in obtaining efficient parallel solution for many graph problems.

2.3 Modified Pruning Decomposition

In the first step (step 4.1.2 of Algorithm 2.2.1) of the pruning decomposition described in previous section, we found the longest chain from each leaf. This is required for solutions of some graph problems [DH5][HHT]. A more efficient version of pruning decomposition can be obtained when the chain does not necessarily have to be the longest possible. In this decomposition, the construction of the spanning tree dominates the complexity, while all the other steps can be done in O(log$N$) time using N^2/log$N$ processors on EREW P-RAM. Hence in graphs for which a spanning tree can be found efficiently on EREW P-RAM (such as planar graphs, serial parallel graphs, directed

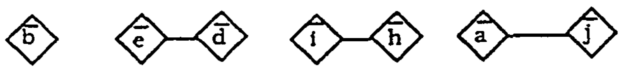


Figure 2.2.5 Spanning forest of super leaves

acyclic graphs, etc.), the pruning decomposition can be obtained much faster. Utilizing this modified pruning decomposition, many problems on those graphs can be solved efficiently. For example, st-numbering [DH1], ear decomposition [DH3], biconnected components [DH2], bipartition of biconnected graphs and node location tree constructions [DH3] and centroid tree (median tree) construction of trees [CDH].

The main change here is to use the iterative operations of COMPRESS and RAKE of the tree contraction [NDP][KR] on the spanning tree T of given graph G. In tree contraction, operation RAKE removes all the leaves from the tree. Here we move the leaves from T to an auxiliary graph A_i, if it is in the i^{th} iteration. As we know, a leaf that is moved by RAKE is actually a chain. This chain was compressed into a node by previous COMPRESS operations. We define DECOMPRESS to be an operation that recovers the chain from the node by simply tracing COMPRESS's back. Performing DECOMPRESS for the leaves moved into the auxiliary graphs will not change the complexity of tree contraction. We use these recovered chains as the chains of the auxiliary graphs and add super leaves and edges to form the branches of the auxiliary graphs as described in previous section. The only difference now is the chains of the branches are not necessary the longest chains from the leaves. In tree contraction there are O(log$N$) operations of COMPRESS (therefore O(log$N$) DECOMPRESS's) and K=O(log$N$) iterations (i.e., K auxiliary graphs). Let N_i be the number chain vertices in A_i, D_i be the number of DECOMPRESS's in A_i. From the property of tree contraction, we have $\sum_{i=1}^{K} D_i = O(\log N)$.

The algorithm of this modified pruning decomposition is derived from the procedure PRUNING_DECOMPOSITION (Algorithm 2.2.1). The only change is in the

Modified Step 4

```
4. {Construct the auxiliary graphs}
    4.1 Repeat
            4.1.1 i←i+1;
            4.1.2 COMPRESS;
            4.1.3 move the leaves to $A_i$;   {modified RAKE}
            4.1.4 DECOMPRESS;                 {obtain chains}
            4.1.5 add super leaves for each chain of $A_i$;
        Until $T$ is empty;
    4.2 Add representing edges for $A_i$; {complete branch construction}
    4.3 K←i;          {K=O(log$N$) auxiliary graphs}
```

Figure 2.3.1 Modified step 4 of PRUNING_DECOMPOSITION

construction of auxiliary graphs (step 4 of Algorithm 2.2.1). Replacing step 4 of Algorithm 2.2.1 by modified step 4 (Fig.2.3.1) gives the implementation of modified pruning decomposition.

It is easy to see that the modified step 4 of pruning decomposition can be computed in O(log$N$) time using N^2/log$N$ processors on EREW P-RAM. Step 4.1.1-4.1.3 is tree contraction which can be obtained in O(log$N$) time using N/log$N$ processors on EREW P-RAM[GMT][NDP][KR]. Step 4.1.4, DECOMPRESS takes O(D_i) using N_i/logN_i processors, step 4.1.5 can be done in O(1) time using at most N_i processors. We know $\sum_{1}^{K} D_i$=O(log$N$). Step 4.2 can be achieved in O(log$N$) time using N^2/log$N$ processors on EREW P-RAM as discussed in the previous section.

The algorithm of modified pruning decomposition is dominated by spanning tree construction (step 1 and step 5 of Algorithm 2.2.1). All the other steps can be achieved in O(log$N$) time using N^2/log$N$ processors. For the graphs whose spanning tree can be found efficiently, the pruning decomposition can be obtained faster. When a graph is a tree, no spanning tree needs to be constructed and no representing edges need to be updated, the pruning decomposition can be done in O(log$N$) time using O(N) processors on EREW P-RAM.

CHAPTER 3

BICONNECTED COMPONENTS ON EREW P-RAM

3.1 Introduction

A *biconnected component* of a graph is a maximal subgraph that contains no articulation points. Finding all the biconnected components (and finding all the articulation points) of a graph is a basic problem in graph theory and is used very often for many graph problems. The sequential algorithm to find the articulation points is based on depth first search (DFS) which has time complexity O(M)[E] where M is the number of edges. The parallel algorithm on CRCW P-RAM takes O(log$N$) time using M+N PE's[TV] where N is the number of vertices of the graph. This algorithm can be improved to O(log$N$) time using $(M+N)\alpha(M,N)/\log N$ processors on CRCW, where α is the inverse Ackermann function [KR]. On CREW this can be done in O($\log^2 N$) time using O($N\lceil N/\log^2 N\rceil$) PE's[TC] or O(N^2/p) time using p PE's[TV] where $p \leq N^2/\log^2 N$. On EREW P-RAM this problem can be done by transitive closure where O($N^3/\log N$) PE's are required [DNS][H].

In this chapter we present a method based on the pruning decomposition that finds all the articulation points and biconnected components on EREW P-RAM with very small overhead and simple data structure. Our algorithm can be done in O(log$N$) time using $N^2/\log N$ processors on EREW P-RAM based on the results of pruning decomposition. We present this algorithm in next section. To facilitate the discussion, we consider only connected graphs in the following. For the case that a graph is not connected, each connected component can be treated as a connected subgraph on which our algorithm can be applied.

3.2 Biconnected Components on EREW P-RAM

We first introduce the idea of how our algorithm finds the biconnected components based on the pruning decomposition and then proceed to discuss the details of the parallel implementation.

Let BC be the set of biconnected components and AP be the set of articulation points of graph G. A *pseudo-graph* G_{ps} can be derived from G in the following way:

1. Every biconnected component $B_i \in BC$ is a node in G_{ps};
2. Two biconnected components B_i and B_j are "connected" by a vertex u if $u \in B_i$, $u \in B_j$ and $u \in AP$.

Figure 3.2.1 shows an example of pseudo-graph G_{ps} of G. Notice that G_{ps} has a structure similar to a tree. A biconnected component B is a *pendant* if it contains only one articulation point (we define this articulation point the *anchor* of the biconnected

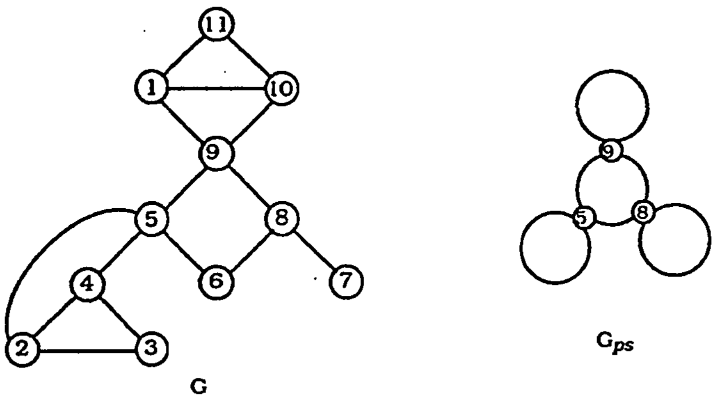


Figure 3.2.1 Pseudo-graph of G

component). There must be some pendants in any G_{ps} according to the tree like structure. If we remove the pendants (except the articulation points which are contained in the non-pendants also), some new pendants will be generated in the remaining part of G_{ps}. The strategy that keeps finding and removing the pendants makes every B a pendant exactly once and leads us to a method which can be modified into a parallel algorithm for finding the biconnected components of a graph. Clearly every biconnected component has exact one anchor, except the last remained biconnected component.

We are using the results of the pruning decomposition to perform this strategy. For the purpose of finding biconnected components, we need to define some more terms related to the auxiliary graphs of pruning decomposition for further presentation. In a branch b_w, a chain vertex u is *non−local* if u can reach out of b_w (i.e., a vertex v that is not a descendant of w in T) without passing the parent of u. Otherwise u is *local.* A super leaf $\bar{u}$ is *attached* to a chain vertex x if there is an edge $(x,\bar{u})$ in the auxiliary graph. Similarly, a chain vertex x is attached by a pile if there is a super leaf attaches to x belongs to the pile. Let $max(p_u)$ ($min(p_u)$) be the largest (smallest) chain vertex attached by p_u. A chain vertex x is *internal* to p_u if $max(p_u) > x > min(p_u)$.

We proceed to show that the articulation points can be found locally in the branches of the auxiliary graphs from the pruning decomposition of G.

Consider a branch of an auxiliary graph (Fig.3.2.2). There are some local and non-local piles attach to the chain vertices. Assume the vertices are labeled in postorder of T. We have the following observations:

1. If a chain vertex u is non-local and chain vertex $x > u$, then x is also non-local (in Fig.3.2.2, vertex 7 is nonlocal because vertex 6 is nonlocal).

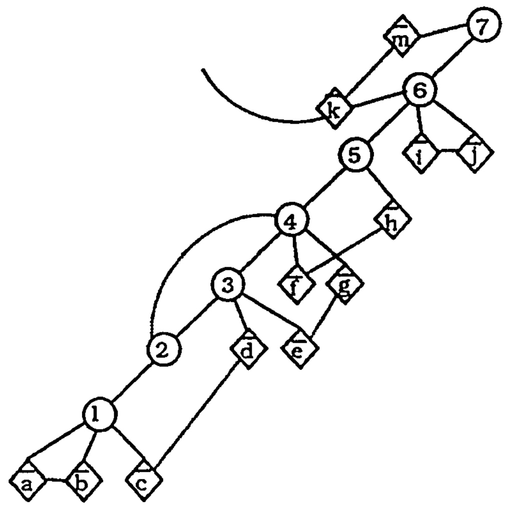


Figure 3.2.2 Finding articulation points in a branch

2. If a local pile p_u attaches only to one chain vertex x, i.e., $max(p_u)=x=min(p_u)$, x is an articulation point (called *type I* articulation point) that separates the vertices belong to p_u from other vertices (in Fig.3.2.2, vertex 1,6 are type I articulation points).

3 If a local chain vertex (or the smallest non-local chain vertex but not the smallest chain vertex) x is not internal to any of local piles, than x is an articulation point (called *type II* articulation point) that separates its parent from its children (in Fig.3.2.2, vertices 4,5 and 6 are type II articulation points).

Lemma 3.2.1: A vertex u is an articulation point of G iff u is identified in a branch as an articulation point of type I or type II.

Proof:

"==>": If u is an articulation point of G, u must separate its parent v from one of its children x. In the branch where u is a chain vertex, we have the following cases:

Case 1: both u and x are chain vertices of the branch. Because u separates v and x, there is no pile (or forward edge) that attaches chain vertices larger than u and smaller than u, i.e., u is not internal to any local pile (or forward edge). It is identified as a type II articulation point.

Case 2: u is a chain vertex but $\bar{x}$ is a super leaf which belongs to a local pile p_y. If p_y attaches only to u then u is identified as a type I articulation point. If p_y attaches to other chain vertices, since v cannot reach x without passing u, u must be either $max(p_y)$ or $min(p_y)$. Therefore u is not internal to any other local pile. It is identified as an articulation point of type II.

Case 3: u is the root of the branch. If x is a chain vertex, then u is either a local or the smallest non-local chain vertex. It cannot be internal to any pile or forward edge since it is the root of the branch. Hence it is identified as an articulation point of type II. If x is a super leaf, it belongs to some local pile p_y. Similar to case 1 and 2, u is identified as an articulation point of type I or type II.

"<==": Trivial.□

Consider a branch without any of the local piles that attach to only one chain vertex. If there is no type II articulation point then all the chain vertices belong to a same biconnected component, because no chain vertices can separate the chain. Therefore all the vertices of super leaves belong to the same biconnected component. If there are some type II articulation points, let x be the largest type II articulation point. Because x separates the upper part of the branch from the lower part of the branch and there are no type II articulation point greater than x, all the chain vertices that are not smaller than x

and all piles attach only to those chain vertices belong to a same biconnected component. All the other vertices of the branch are of other biconnected components. If we remove those vertices of other biconnected component, $\overline{w}$ will represent vertices of a same biconnected component in some auxiliary graph later (defined to be *purified* super leaf), such as $\overline{i}, \overline{j}$, 7 and 8 will be represented by purified super leaf $\overline{8}$. Since the root of the branch is never removed and all the removed vertices are either local chain vertices or of local piles, the structure of all the branches will not change.

We proceed to discuss how to identify biconnected components locally in a branch.

Given a branch of with all the super leaves "purified". We have:

1. If u is a type I articulation point and p_x is a local pile that attaches only to u, then vertices of p_x and u belong to a same biconnected component (such as $\overline{a}, \overline{b}$ and vertex 1 of Fig.3.2.2). According to the property of purified super leaf, there are no other vertices belong to this biconnected component. It is a biconnected component of *type I* and u is its anchor.

2. If u be a type II articulation point and v be the greatest chain vertex that is smaller than u and is not internal to any of the local piles (such as 4 and 1 in Fig.3.2.2). All the chain vertices between u and v (including u and v) and local piles attached to those vertices are of a same biconnected component (such as 1, 2, 3, 4, $\overline{c}$, $\overline{d}$, $\overline{e}$ and $\overline{g}$ in Fig.3.2.2). By the property of purified super leaf, it is a biconnected component of *type II* and u is its anchor.

Lemma 3.2.2: A subgraph $B=(V_B, E_B)$ is a biconnected component of G iff V_B is identified as a biconnected component in some branch.

Proof:

"==>:" Let B be a biconnected component of G and u be its anchor. By the structure of G_{ps}, the anchor of B has the greatest preorder among the vertices of B. Hence every vertex of B is either a chain vertex or belongs to some super leaf in the branch where u is a chain vertex. Here we have two cases:

Case 1: u is the only vertex of B that is a chain vertex of the branch, while all the other vertices of B are represented by some super leaves. Since B is a connected component, those super leaves belong to pile p_y. Since B is biconnected maximally, p_y is local. Obviously p_y attaches only to u and u is a type I articulation point. By the discussion above, all the vertices of B belong to $p_y \cup \{u\}$ and $p_y \cup \{u\}$ contains only vertices of B. It is identified as a type I biconnected component in this branch.

Case 2: there are some vertices other than u of B are chain vertices of the branch. Let v be the smallest vertex of B that is also a chain vertex. Every chain vertex between v and u is internal to some local piles or forward edges, otherwise v and u can be separated by that vertex. Chain vertex v is not internal to any local pile or forward edge because (1) If v is the smallest vertex of the branch then it is not internal to any local piles, (2) if v has a child x that is also a chain vertex then v is not internal to any local piles, otherwise x belongs to B and $x < v$, which contradicts the assumption. Therefore v is the greatest chain vertex that is smaller than u and is not internal to any local piles. Every vertex of B is either a chain vertex between u and v or belong to some local pile that attaches to those vertices. By the property of purified super leaf, B is identified as a type II biconnected component in this branch.

"<==": Trivial.□

Since the articulation points and biconnected components can be found locally in branches, we are ready to present our method to find articulation points and biconnected components of graph G based on the pruning decomposition. The parallel implementation of our method is shown in Algorithm BI_COMPONENTS (Algorithm 3.2.1).

The inputs of the BI_COMPONENTS, Algorithm 3.2.1, are the results of the pruning decomposition. Initially, the set of articulation points AP and the set of biconnected components BC are empty. The algorithm works locally on each branch. The iterative strategy of pseudo graph G_{ps} can be performed without iterations since the pruning decomposition gives the disjoint branches.

Step 2 through step 5 can be done based on the matrices M, M_1 and M_2 from the pruning decomposition. The operations involve OR, maximum finding, minimum finding and deleting on rows or on columns. These can be performed in O(log$N$) time using N^2/log$N$ processors on EREW P-RAM.

Step 6 identifies the articulation points and biconnected components locally on each branch. In step 6.1, the type I articulation points and biconnected components can be found in O(1) time using N processors. Step 6 can obtains all the type II articulation points in O(1) time using N processors. In step 6.3.1, the greatest non_internal vertex can be found by partial sum which can be achieved in O(logN_b) time using N_b/logN_b processors, where N_b is the number of chain vertices of the branch. Hence it can be achieved in O(log$N$) time using N/log$N$ processors for all the branches. The rest of step 6.3 can be done in O(log$N$) time using N^2/log$N$ processors on EREW P-RAM. Therefore the whole

```
Algorithm BI_COMPONENTS:
{Input: Results of pruning decomposition;
 Output: Set of articulation points $AC$ and set of biconnected component $BC$.}
1. { Initialization }
     $AP \leftarrow \emptyset$;     $BC \leftarrow \emptyset$;
2. Find $max(p_u)$ and $min(p_u)$ for each local pile $p_u$;
3. Mark chain vertices that are internal to some local piles or forward edge;
4. Identify "non_local" chain vertices;
5. "Purifying" super leaves;
6. { Find articulation points and biconnected components }
     For each branch in parallel do
          6.1 { Identify type I articulation points and biconnected components }
               For each local pile $p_u$ with $max(p_u)=min(p_u)$ in parallel d
                    6.1.1 $AP \leftarrow AP \cup \{max(p_u)\}$;
                    6.1.2 $S \leftarrow p_u \cup \{max(p_u)\}$;
                    6.1.3 $BC \leftarrow BC \cup \{S\}$;
          6.2 Identify type II articulation points and add them into $AP$;
          6.3 { Identify type II biconnected components }
               For each type II articulation point $u$ in parallel do
                    6.3.1 $v \leftarrow \max\{x \mid x<u$ is a non_internal chain vertex$\}$;
                    6.3.2 $A \leftarrow \{x \mid v \le x \le u$ and $x$ is a chain vertex$\}$;
                    6.3.3 $B \leftarrow \{x \mid x$ belongs to a pile that attaches only to a vertex of $A\}$;
                    6.3.4 $C \leftarrow A \cup B$;
                    6.3.5 $BC \leftarrow BC \cup C$;
end;
```

Algorithm 3.2.1 Algorithm for biconnected components

algorithm can be done in O(log$N$) time using $N^2/\log N$ processors on EREW P-RAM.

Theorem 3.2.1: Algorithm BI_COMPONENTS (Algorithm 3.2.1) finds all the biconnected components of G in O(log$N$) time using $N^2/\log N$ processors on EREW P-RAM.

Proof: Follows by Lemma 3.2.1, Lemma 3.2.2 and the complexity analysis of the algorithm.□

As an extension of this result, bridges and 2-edge connected components of an undirected graph G can be found with the same complexity.

Notice the facts that every bridge of G must be a tree edge of T and the end point of a bridge must be either an articulation point or a vertex of degree one. So we can focus on just tree edges, articulation points, leaf and root of T. If both the end points of a bridge are activated in the same iteration, i.e., they are adjacent chain vertices then they cannot reach each other without passing the bridge. If the endpoints are activated in different iteration, then the bridge is represented by an only edge between a local pile containing only one super leaf and a chain vertex. Therefore according to the algorithm BI_COMPONENTS (Algorithm 3.2.1), if a chain vertex u is identified as a type I articulation point by step 6.1, let the super leaf attaches to u is $\bar{v}$ (i.e., v is a child of u in T), then (u,v) is a bridge when v is a type II articulation point in the branch with root v. If a chain vertex u is a type II articulation point and in step 5.3.1 v is the only child of u in T, then (u,v) is a bridge when v is also a type II articulation point and no pile attaches to both u and v. Obviously adding these operations into the algorithm will not change the complexity. Moreover, if we delete all the bridges from the graph, every connected component will be a 2-edge connected component. So all the bridges and 2-edge connected components of G can be found in O(log$N$) time using N^2/log$N$ PE's on EREW P-RAM based on the pruning decomposition.

We have shown that all the articulation points, bridges, biconnected components and 2-edge connected components can be found efficiently on EREW P-RAM based on the pruning decomposition. Since the complexity of pruning decomposition is dominated by the spanning tree construction, our algorithm is particularly good for graphs where the spanning tree can be constructed efficiently on EREW P-RAM.

CHAPTER 4

EAR DECOMPOSITION ON EREW P-RAM

4.1 Introduction

An *ear decomposition* D=[$P_0, P_1, ..., P_{r-1}$] of an undirected graph G=(V,E) is a partition of E into an ordered collection of edge-disjoint simple paths $P_0, P_1, ..., P_{r-1}$ called *ears*, such that P_0 is a simple cycle, and for i>0, P_i is a simple path (possibly a simple cycle) with each endpoint belonging to a smaller numbered ear, and with no internal vertices belonging to smaller numbered ears.

An ear with no internal vertices is called a *trivial ear.*

An ear with different endpoints is called an *open ear,* otherwise a *close ear.*

An *open ear decomposition* is an ear decomposition in which none of the ears is a close ear.

It is known that a graph has an ear decomposition if and only if it is 2-edge connected (bridgeless) and a graph has an open ear decomposition if and only if it is biconnected[W].

Lovasz[L] showed that ear decomposition problem has an NC parallel algorithm. Y. Maon, B. Schieber and U. Vishkin proved the open ear decomposition is in NC and gave a very efficient parallel algorithm for ear decomposition running in O(log$N$) time using M+N processors on CRCW P-RAM, where M and N are the number of edges and vertices in the graph respectively. With improved implementation, the number of processors can be reduced to $(M+N\log N)/\log N$ [MSV] if the consecutive ear numbers are not required, otherwise it needs $O((M+N)\alpha(M,N))$ works, where $\alpha(M,N)$ is the inverse Ackermann function [KR]. As mentioned in [MSV], the ear decomposition has the flavor of a general search technique in graphs. It arranges the vertices of the graph by partitioning them into paths. This enables exploration of the graph in an orderly manner, which is called *ear decomposition search.* The ear decomposition has been applied to st-numbering computing[MSV], planarity test[RR], triconnectivity test[FRT][RV][MR], strong orientation[V2][KR], etc. The existing parallel algorithm for ear decomposition is based on CRCW P-RAM[MSV] and so are the problems solved based on the ear decomposition as mentioned above. In this chapter, we present an efficient parallel algorithm for open ear decomposition which runs on EREW P-RAM with very small overhead and simple data structures. The spanning tree construction dominates the complexity while all the other steps can be achieved in $O(\log N)$ time using $N^2/\log N$ processors on EREW P-RAM where N is the number of vertices of a given graph. Our algorithm gives consecutive ear numbers.

In section 4.2 we develop a method to reduce a given biconnected graph to a sparse biconnected graph, biconnected certificate (to be defined), so as to avoid massive read or write conflicts. This method brings a new concept "NF-tree" (to be defined) derived from pruning decomposition [DH1] which allows us to construct a biconnected certificate without using transitive closure. In section 4.3 an algorithm is presented to perform the ear decomposition on the sparse graph and further the ear decomposition for the original graph. The preliminaries will be introduced in the related sections.

4.2. NF-trees and biconnected certificates

A spanning tree T of an undirected graph $G=(V,E)$ is called an *NF−tree* if no edge of E is a forward edge for T (NF stands for no forward edges). In another word, every non-tree edge is a cross edge for T. As an example, a breadth first search tree (BFS tree) is an NF-tree.

A *biconnected certificate* $\overline{G}=(V,\overline{E})$ of a biconnected graph $G=(V,E)$ is a sparse subgraph of G with O(|V|) edges s.t. $\overline{G}$ is biconnected if only if G is biconnected. The concept of the *k−connected* certificate was introduced by J. Cheriyan and R. Thurimella [CT]. They also showed a method for finding a *k−connected* certificate for a k-connected graph based on the BFS trees construction. Here we improve the way to find a biconnected certificate which is based on the NF-trees that are efficient for parallel computation.

As introduced in Chapter 2, we can partition the edge set E of a graph $G=(V,E)$ into tree edge set E_t, cross edge set E_c and forward edge set E_f of spanning tree T of G.

An NF-tree can be formed by changing the forward edges to tree edges or cross edges. The Algorithm NF_TREE, Algorithm 4.2.1, gives the implementation of the method in which we assume the vertices are labeled in postorder according to T of G. Figure 4.2.1 gives an example of how an NF-tree is formed, and this example will be used continuously in further discussions.

In Algorithm 4.2.1, Step 1 is to construct a spanning tree and step 2 partitions the edge set of G by pruning decomposition which can be obtained in O($\log^2 N$) time using $N^2/\log N$ processors on EREW P-RAM[NM][DH1]. In rest of the steps the major opera-

```
Algorithm NF_TREE
{Input: A biconnected graph $G=(V,E)$;
 Output: A NF-tree $\bar{T}$ with root $r$. }
1. Construct a spanning tree $T=(V,E_t)$ of $G$ with root $r$;
2. Partition $E$ into $E_t$, $E_f$ and $E_c$ of $T$;
3. $\bar{T}=(V,\bar{E})$ with root $r$ where $\bar{E}\leftarrow E_t$;
4. For every vertex $x$ with parent $v$ in parallel do
      4.1 max_ancestor($x$)=max$\{y \mid x<y$ and $(x,y)\notin E_c\}$;
      4.2 If max_ancestor($x$)$\neq v$
            then $\bar{E}\leftarrow\bar{E}\cup\{(x,$max_ancestor$(x))\}-\{(x,v)\}$;
```

Algorithm 4.2.1 Algorithm for finding an NF-tree

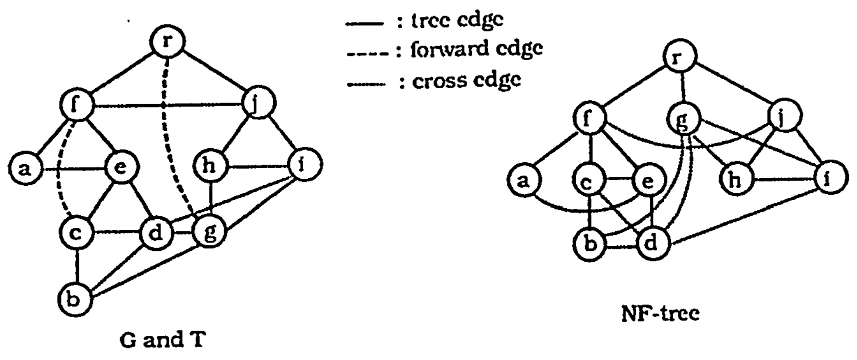


Figure 4.2.1 A biconnected graph and its NF-tree

tion is to find the farthest ancestor for every vertex, which can be done in O(log$N$) time using N^2/log$N$ processors on EREW P-RAM. In the example of Figure 4.2.1, max_ancestor(c)=f and max_ancestor(g)=r so (f,c) and (r,g) are tree edges of $\bar{T}$ that are different from T. Now we show the Algorithm 4.2.1 successfully constructs an NF-tree.

Lemma 4.2.1: Let $A(x)$ be the set of proper ancestors of vertex x in T and $B(x)$ be the set of proper ancestors of vertex x in $\bar{T}$, then $B(x)\subseteq A(x)$.

Proof: Assume on the contrary there is a vertex $y \in B(x)$ but $y \notin A(x)$. Let the path from x to the root r in $\bar{T}$ be x, v_1, it must be the case that max_ancestor(x)=v_1, max_ancestor(v_1)=v_2,..., man_ancestor(v_k)=y in T. Then y must be an ancestor of v_k, ..., v_1 and x in T, i.e., $y \in A(x)$. It is a contradiction to the assumption.□

Corollary 4.2.1: If (x,y) is a cross edge for T then (x,y) is also a cross edge for $\bar{T}$.

Proof: Since (x,y) is a cross edge for T, $x \notin A(y)$ and $y \notin A(x)$. By Lemma 4.2.1, $x \notin B(y)$ and $y \notin B(x)$, then (x,y) is a cross edge for $\bar{T}$.□

Lemma 4.2.2: Let the path from vertex x to the root r of T be x, v_1, ..., v_k, ..., r and max_ancestor(x)=v_k. Then in $\bar{T}$, x and v_j (j<k) are neither ancestor nor descendent to each other.

Proof: From Algorithm 4.2.1, x is a child of v_k in $\bar{T}$. For any vertex v_j (j<k), if $v_k \in B(v_j)$ then v_j must be a descendent of some v_l s.t. j≤l<k and v_l is a another child of v_k (see Fig.4.2.2-a). Therefore $x \notin B(v_j)$ and $v_j \notin B(x)$. If $v_k \notin B(v_j)$ then $x \notin B(v_j)$ and $v_j \notin B(x)$ because $v_j \notin B(v_k)$(see Fig. 4.2.2-b). □

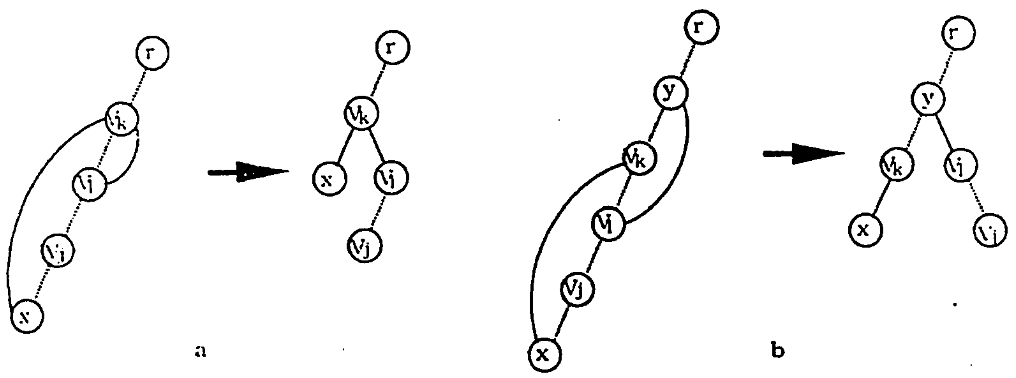


Figure 4.2.2 Cases of transformation to NF-trees

Corollary 4.2.2: If (x,y) is a forward edge of tree edge for T and max_ancestor(x)$\neq y$ then (x,y) is a cross edge for $\bar{T}$.

Proof: Follows from Lemma 4.2.2.□

Theorem 4.2.1: An NF-tree of a graph can be found in O($\log^2 N$) time using $N^2/\log N$ processors on EREW P-RAM.

Proof: From Corollary 4.2.1 and Corollary 4.2.2, we know every edge of G is either a tree edge or a cross edge of $\bar{T}$. Hence Algorithm 4.2.1 correctly finds an NF-tree of a graph. According to the complexity analysis, this can be performed in O($\log^2 N$) using $N^2/\log N$ processors on EREW.□

If a graph is not connected, then a *NF-forest* can be found based on the connected components of the graph in the same complexity. Now we are ready to construct a biconnected certificate for a biconnected graph. The technique used here is similar to the one gave by [CT] except it is based on the NF-trees instead of BFS trees. The idea is to find an NF-tree, delete the tree edges from the graph, find another NF-tree (or NF-forest) and glue those two NF-trees together. Algorithm BI_CERTIFICATE, Algorithm 4.2.2, gives the implementation and Figure 4.2.3 is the example continued from the previous step.

Obviously Algorithm 4.2.2 works in O($\log^2 N$) time using $N^2/\log N$ processors on EREW P-RAM and $\bar{E}$ has O(N) edges which makes $\bar{G}$ a sparse graph. We need to show $\bar{G}$ is a biconnected certificate for G.

```
Algorithm BI_CERTIFICATE
{Input: A biconnected graph $G=(V,E)$;
 Output: A biconnected certificate $\bar{G}=(V,\bar{E})$ of $G$. }
1. Construct an NF-tree $T=(V,E_t)$ of $G$;
2. $G'=(V,E')$ where $E'=E-E_t$;
3. Construct an NF-forest $F=(V,E_f)$ of $G'$;
4. $\bar{G}=(V,\bar{E})$ where $\bar{E}=E_t \cup E_f$;
```

Algorithm 4.2.2 Algorithm for finding a biconnected certificate

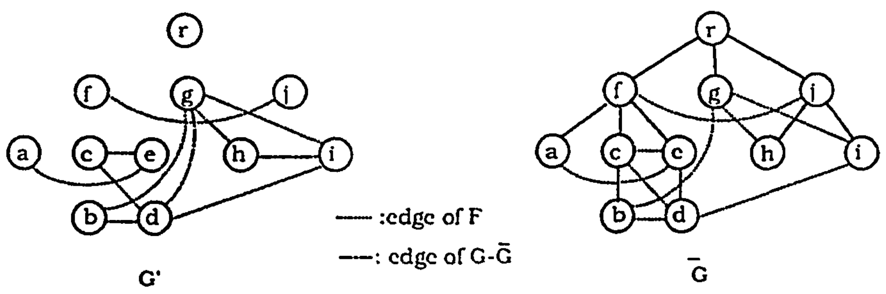


Figure 4.2.3 Example of a biconnected certificate

Lemma 4.2.3: $\bar{G}$ is biconnected if only if G is biconnected.

Proof:

"==>": Trivial.

"<==": Assume on the contrary there is an articulation point m in $\bar{G}$ but G is biconnected. Obviously G has more than two vertices. We have three cases (see Fig.4.2.4):

Case 1: m is the root of T and m has only one child (Fig.4.2.4-a). Because there is no forward edge in T, m has only one incident edge in G also, which implies G is not biconnected.

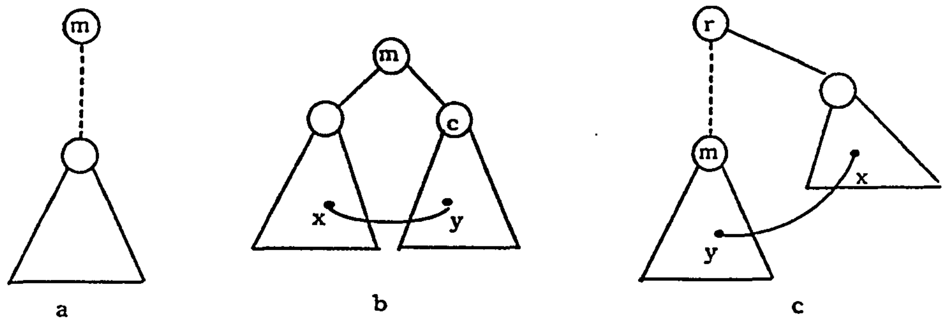


Figure 4.2.4 Cases of articulation points

Case 2: m is the root of $\overline{T}$ and m has more than one children (Fig.4.2.4-b). There must be a subtree t_c rooted with c, a child of m, s.t. there is no edge (x,y) in F has $x \notin t_c$ and $y \in t_c$ since m is an articulation point of $\overline{G}$. But such an edge exists in G because G is biconnected and this edge must be a cross edge for T thus it is in G'. This implies there is an edge (x,y) in F s.t. $x \notin t_c$ and $y \in t_c$. This is a contradiction.

Case 3: m is not the root of $\overline{T}$ (Fig.4.2.4-c). Since m is an articulation point of $\overline{G}$, there is no edge (x,y) in F s.t. $x \notin t_m$ and $y \in t_m$ where t_m is the subtree of T with root m. But there exists such an edge in G because G is biconnected and this edge must be a cross edge for T thus it is in G'. This implies there is an edge (x,y) s.t. $x \notin t_m$ and $y \in t_m$. Again, this is a contradiction.□

Theorem 4.2.2: A biconnected certificate $\overline{G}$ of a biconnected graph G can be constructed in O($\log^2 N$) time using $N^2/\log N$ processors on EREW P-RAM.

Proof: Follows from Lemma 4.2.3 and complexity analysis of Algorithm 4.2.2.□

An NF-tree has the property that every non-tree edge is a cross edge, which is an important property of BFS tree. In parallel computation, BFS tree construction needs

$O(\log^2 N)$ time by $O(N^3/\log N)$ processors on EREW [DNS][H]. In the cases that BFS tree construction is necessary because of this property, the NF-tree can be used instead of the BFS tree. Biconnected certificate is one of the example. The NF-trees can be used for k-connected sparse certificates instead of the BFS tree[CT]. This will reduce the number of processors from $O(N^3/\log N)$ to $O(N^2/\log N)$. Notice that in Algorithm 4.2.2, only the spanning forest construction needs $O(\log^2 N)$ time, while all other steps can be achieved in $O(\log N)$ time.

4.3 Ear decomposition on EREW P-RAM

In this section first we give an algorithm for ear decomposition of a biconnected certificate, then for ear decomposition of a biconnected graph. We also show it can be used to find ear decompositions for 2-edge connected graphs.

The idea of the algorithm is close to the CRCW algorithm [MSV]. Since we works on a certificate with only $O(N)$ edges, complexity is $O(\log N)$ time using $N^2/\log N$ PE's on EREW P-RAM if a biconnected certificate is known. It is implemented by Algorithm EAR_DECOMPOSITION, Algorithm 4.3.1 and the example is shown in Figure 4.3.1. We assume the vertices are labeled in preorder on T.

Since F has at most $k<N$ edges, the Algorithm 4.3.1 allows k copies of T to avoid read and write conflicts especially in step 1 and step 3. By the Euler tour technique [TV], in each of those T's the lowest common ancestor of a pair of cross edge endpoints can be found in $O(\log N)$ time using $N/\log N$ processors on EREW P_RAM. Other operations are sorting and minimum finding. The whole algorithm can be done in $O(\log N)$ time using

```
Algorithm ERA_DECOPOSITION
{Input: $G$, $T$, $G'$, $F$ and $\overline{G}$;
 Output: An ear decomposition $P_0, ..., P_k$ of $\overline{G}$.}
1. For every edge $(x,y)$ of $F$ in parallel do
        $LCA(x,y)$=lowest common ancestor of $x$ and $y$;
2. Sort the edges of $F$ in increasing order of $LCA$ and the endpoints;
   Let $ORD(x,y)$ be the position of $(x,y)$ in the sorted list;
3. For every edge $(x,y)$ of $F$ in parallel do
        For every edge $(u,v)$ that is between $x$ (or $y$) and $LCA(x,y)$ do
                $LABEL(u,v,ORD(x,y))=ORD(x,y)$;
4. For every tree edge $(u,v)$ of $T$ in parallel do
        $ORD(u,v)=\min\{LABEL(u,v,i)\}$;
5. Ear $P_i=\{(u,v) | ORD(u,v)=i\}$.
```

Algorithm 4.3.1 Ear decomposition for certificate

$N^2/\log N$ processors on EREW P-RAM. In the example of Figure 4.3.1-a, step 2 actually sorts the edges of F in the order of the 3-tuple (preorder of LCA, preorder of left endpoint, preorder of right endpoint) for each edge. Therefore the sorted edges of F are: (f,j)=(1,2,9), (b,g)=(1,5,8), (d,i)=(1,7,11), (g,h)=(1,8,10), (g,i)=(1,8,11), (a,e)=(2,3,6), (c,e)=(2,4,6), (c,d)=(2,4,7) and (b,d)=(2,5,7). Step 3 gives the same label for the edges of every cycle created by each edges of F. For example, edge (b,d) creates a cycle $b-d-e-f-c-b$. Hence all the edges involved will get a label from (b,d), which is 9, from the sorted list. A tree edge may have more than one labels in step 3 (as we mentioned in step 1, k copies of T' can avoid the access conflicts). For example, tree edge (f,c) has four labels: 2 from (b,g), 5 from (c,e), 6 from (c,d) and 7 from (b,d). Then in step 6 the order of (f,c) is 2.

After the above steps, every edge has a unique *ORD* value that indicates to which ear it should belong. Step 5 does the collection and results for the example are shown in Fig.4.3.1-b.

We can see that the operations involved are sorting and minimum finding, the whole algorithm can be done in O(log$N$) time using N^2/log$N$ processors on EREW P-RAM.

Theorem 4.3.1: An open ear decomposition of a biconnected certificate can be found by EAR_DECOMPOSITION in O(log$N$) time using N^2/log$N$ processors on EREW P-RAM.

Proof: Followed by the complexity analysis of Algorithm 4.3.1 and the correctness proof of CRCW algorithm in [MSV].□

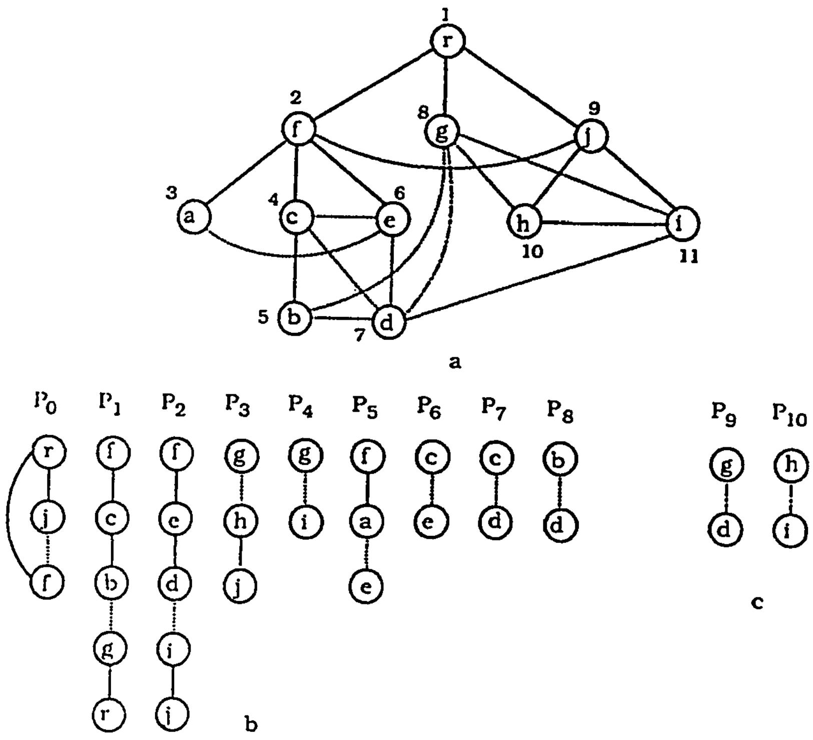


Figure 4.3.1 Ear decomposition

Theorem 4.3.2: An open ear decomposition of a biconnected graph can be found by EAR_DECOMPOSITION in O(log$N$) time using N^2/log$N$ processors on EREW P-RAM.

Proof: Let the certificate $\overline{G}$ have ears $P_0, ..., P_k$. Let every edge (x,y) that is in G but not in $\overline{G}$ be a trivial ear P_l. As long as l>k, it will be an open ear decomposition for G. The only thing need to be done is to sort those trivial ears according to their endpoints. Then the open ear decomposition for G is $P_0, ..., P_k, P_{k+1}, ..., P_{k+h}$, where h is the number of edges that are in G but not in $\overline{G}$ (in the example of Figure 4.3.1-c, $P_9=\{(d,g)\}$ and $P_{10}=\{(h,i)\}$). Since h is O(N^2), the sorting will not dominate the complexity of the algorithm.□

This method can be modified to find an ear decomposition for more general cases, i.e., for 2-edge connected (bridgeless) graphs. In Chapter 3 we know that G_{ps} is a tree-like structure of size O(N). Therefore every biconnected component has an articulation point that "connects" it to its "parent biconnected component" except the "root". Let this articulation point be the root of an NF-tree for the biconnected component and find an ear decomposition for the biconnected component. Since the biconnected components can be labeled in preorder of G_{ps}, all the ears can be arranged in the order s.t. ears of a biconnected component have the number larger than the number of the ears of its "parent biconnected component". Therefore all the ears will have the numbers satisfying the conditions of ear decomposition. Obviously the complexity is the same because the biconnected components can be found in the same complexity [DH2].

CHAPTER 5

ST-NUMBERING AND APPLICATIONS

5.1 Introduction

Given a biconnected graph $G=(V,E)$ that $|V|=N$ and $s,t \in V$, A one-to-one function f from V to $\{1,...,N\}$ is called *st−numbering* of $\{s,t\}$ if it satisfies (i) $f(s)=1$ and $f(t)=N$, and (ii) for each $v \in V-\{s,t\}$ there exist adjacent vertices x and y s.t. $f(x)<f(v)<f(y)$.

The serial algorithm for st-numbering depends on depth first search (DFS)[E]. The existing parallel algorithms for st-numbering that run on CRCW or CREW models of computation. Y. Maon, B. Schiever and U. Vishkin gave a CRCW st-numbering algorithm running in O(log$N$) time on $M+N$ processors, where M is the number of edges and N is the number of vertices of the graph. The improved implementation of this algorithm runs in O(log$N$) time using $(M+N\log N)/\log N$ processors on CRCW P-RAM model[MSV].

In this chapter, we present an efficient parallel algorithm for st-number computing which runs in O(log$N$) time using N processors on EREW P-RAM with very small overhead and simple data structure based on the ear decomposition of Chapter 4.

Many problems that are solved sequentially using depth first search can be attacked in parallel by using st-numbers. The st-numbering plays an important role in many graph problems such as planarity testing and triconnectivity. Some communication network related problems related to st-numbering will be introduced in this chapter.

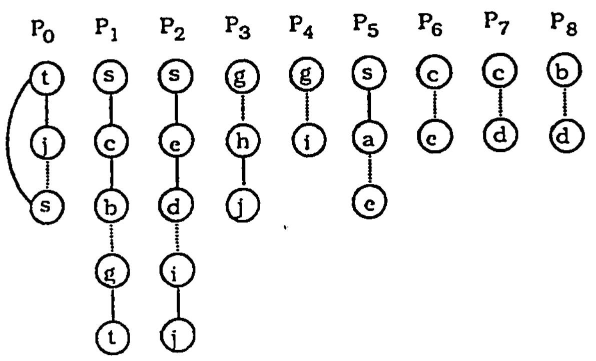
In section 5.2 we present the algorithm for computing an st-numbering of biconnected graphs based on the results of ear decomposition of Chapter 4. In Section 5.3 we apply the st-numbering to other graph problems including bipartitioning of biconnected graphs, constructing centroided tree or centered tree of biconnected graphs and strong orientation.

5.2 St-numbering on EREW P-RAM

In Chapter 4 we can obtain an open ear decomposition with consecutive ear numbers $P_0,...,P_{r-1}$ from a biconnected certificate $\overline{G}$ of a biconnected graph G. By the fact that an st-numbering of $\overline{G}$ is also an st-numbering of G, we introduce a very simple method to compute an st-numbering for $\overline{G}$ (hence for G), on EREW P-RAM in O(log$N$) time using N processors based on the result of ear decomposition of $\overline{G}$.

From Algorithm 4.3.1 and example of Fig.4.3.1 (for convience, we copy the ears of $\overline{G}$ of Fig.4.3.1 to Fig.5.2.1 and let $s=f$, $t=r$), we have the following observations:

- Every ear of $\overline{G}$ contains exact one cross edge of T (e.g., (b,g) is the only cross edge in P_1 of Fig.5.2.1).
- Let vertices x,y be the endpoints of ear P_i and all the rest vertices of P_i be *internal vertices* of P_i, then $x \in P_j$ and $y \in P_l$ s.t. $j \leq l < i$ (e.g., in P_2 of Fig.5.2.1, $f \in P_0$, $j \in P_1$ and e,d,i are internal vertices).
- If vertex u is an internal vertex of P_i and vertex v is the parent of u in T, then v is either an internal vertex or an endpoint of P_i (e.g., e is the parent of d in T of $\overline{G}$ of Fig.4.3.1).



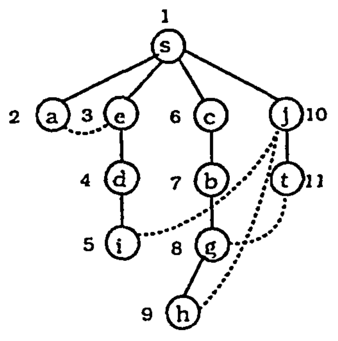


Figure 5.2.1 Ears of $\overline{G}$ and T_{st}

Lemma 5.2.1: The internal vertices of all the ears of $\overline{G}$ partition the vertex set of V-$\{s,t\}$.

Proof: The ear decomposition partitions the edge set of $\overline{G}$, so every vertex appears in some ears. Let s and t be the endpoints of P_0, then every vertex of V-$\{s,t\}$ appears as an internal vertex of some ears. We need to proof that each every vertex appears as an internal vertex at most once. Assume on the contrary, vertex u appears as an internal vertex of P_i and P_j. Let v be the parent of u. Edge (u,v) will appear in both P_i and P_j, contradiction to the fact that ear decomposition partitions the edges of $\overline{G}$.□

We define an ear index function p of vertices to be $p(u)$=i if vertex u is an internal vertex of ear P_i. By Lemma 5.2.1 we know that $p(u)$ for u is unique. Therefore in ear P_i, we have $p(x)$=g and $p(y)$=h, where x,y are endpoints and $p(u)$=i for every internal vertex u. Let y be the *head* of the ear and x be the *tail* of the ear if g≤h (e.g., in P_2 of Fig.5.2.1, s is the head and j is the tail). Let $p(s)=p(t)=0$. We construct a tree T_{st} in the following way:

- If u is an internal vertex of P_i and v is the vertex that adjacent to u and closer to the head of P_i (v could be the head), then v is the parent of u in T_{st}. In example of

Fig.5.2.1, internal vertex d of P_2 has parent e in T_{st}.

- If u,v are children of x in T_{st}, then u is on the left of v in T_{st} if $p(u)>p(v)$. In example of Fig.5.2.1, a,e,c,j are children of s. Because $p(a)>p(e)>p(c)>p(j)$, s has children in order of a,e,c,j in T_{st}.

Lemma 5.2.2: The preorder of T_{st} is an st-numbering of $\overline{G}$.

Proof: Let $g(u)$ be the preorder of u in T_{st}. The root of T_{st} is s, which has $g(s)=1$. Vertex t is the right most leaf of T_{st} according to the ordered construction, hence $g(t)=N$. We need to show that for every other vertex u, there must be $g(x)<g(u)<g(y)$, where x,y are neighbors of u in $\overline{G}$. Consider vertices of V-$\{s,t\}$ in T_{st}, we have two cases:

Case 1: u is not a leaf of T_{st}. It has $g(x)<g(u)<g(y)$ where x is its parent and y is its child.

Case 2: u is a leaf of T_{st}, u must be an internal vertex of P_i s.t. u is next to the tail y of P_i. Then (u,y) is a cross edge of T_{st}. By the order of children for every vertex, if $p(u)>p(y)$ then y is always on the right side of u, i.e., $g(u)<g(y)$. Let x be the parent of u in T_{st}, we have $g(x)<g(u)<g(y)$.□

The procedure ST_NUMBER, Algorithm 5.2.1, describes the method. Referring the example of Fig.5.2.1, after step 1, $p(s)=p(j)=p(t)=0$, $p(c)=p(b)=p(g)=1$,...etc. This can be done by identifying endpoints in O(1) time using N processors. Step 2 can be achieved by list ranking in O(log$N$) time using N/log$N$ processors. The sorting of step 3 can be done in O(log$N$) time using N processors [C]. The preorder can be computed in step 5 in O(log$N$) time using N/log$N$ processors. Therefore Algorithm 5.2.1 runs in

```
Procedure ST_NUMBER
{Input: Ears $P_0,...,P_k$;
 Output: st-numbers of $V$.}
1. Compute $p(u)$ for each internal vertex of each ear;
2. Identify parent for every internal vertex of each ear;
3. Sort children for each vertex;
4. Construct $T_{st}$;
5. Compute preorder of $T_{st}$;
```

Algorithm 5.2.1 Algorithm for st-numbering

O(log$N$) time using N Processors.

Theorem 5.2.1: Procedure ST_NUMBER computes an st-numbering of a biconnected graph G.

Proof: By Lemma 5.2.2 and complexity analysis of Procedure ST_NUMBER.□

5.3 Applications of st-numbering

As we mentioned before, st-numbering can be used to solve many graph problems in parallel. In this section we apply the st-numbering to some network related problems.

5.3.1 Bipartition of biconnected graphs

The problem of bipartition of biconnected graphs is defined to be:

Given a biconnected graph $G=(V,E)$, $s_1,s_2\in V$, $n_1+n_2=|V|$, partition V into V_1 and V_2 s.t. $s_1\in V_1$, $s_2\in V_2$, $|V_1|=n_1$, $|V_2|=n_2$ and both V_1,V_2 are connected subgraphs of G by edges of G.

The bipartition of biconnected graphs partitions a system with specified vertices in different connected portions of desired sizes. It has applications in job scheduling, fault tolerant routing, distributed system management and various communication problems.

H. Suzuki, N. Takahashi and T. Nishizeki found a serial algorithm in 1990 for bipartition of biconnected graphs of time complexity O(M+N), which is based on the DFS tree of the graph [STN]. Here we introduce a solution for this problem that is easy to be parallelized:

Let function f be a st-numbering of $\{s_1, s_2\}$ for V, i.e., let s_1 be s and s_2 be t. We partition vertex set V into: $V_1=\{v|f(v)\leq n_1\}$ and $V_2=\{v|f(v)>n_1\}$.

Claim 5.3.1: V_1 and V_2 is a bipartition of V.

Proof: Obviously $s_1\in V_1$, $s_2\in V_2$, $|V_1|=n_1$ and $|V_2|=n_2$. If we assign direction for every edge of G from u to v if $f(u)<f(v)$, a directed acyclic graph (*DAG*) is formed. Every vertex v has a directed path from s_1 to v s.t. any vertex u on the path has $f(s)<f(u)<f(v)$. If $f(v)\leq n_1$ then every vertex u on the path has $f(u)\leq n_1$. Therefore they are all in V_1. An outtree with root s_1 can be found with vertices of V_1. In the undirected sense every vertex v is connected to s_1 if $f(v)\leq n_1$. Hence vertices of V_1 are connected. Similarly, an intree with root s_2 with vertices of V_2 can be found and in the undirected sense, the vertices of V_2 are connected.□

Clearly the running time of this partition is determined by the st-numbering computing.

5.3.2 Centroided Trees

To establish a connected system so that a certain server is located in a position to meet some criteria is frequently required in networks. Generally they are referred as location problems.

We first give the preliminaries of the location problems:

The *distance* of two vertices is the length of the shortest path between the two vertices. The distance of a vertex u from a vertex set S is the shortest distance between u and any vertex of S. The *distancesum* of vertex u is the total distances from u to all the vertices in $V-u$. The distancesum of vertex set S is the total distances from S to all the vertices in $V-S$. The *median* of a graph is the vertex which has the minimum distancesum. Particularly in a tree, a *centroid* is the vertex that by taking it away, the largest subtree has size$\leq 1/2N$. It is known that in a tree, a centroid is a median[Sl].

The problem of constructing a centroided tree of a biconnected graph is to find a spanning tree of the graph with specified vertex as the centroid. A serial algorithm of time complexity O(M) was introduced in 1984 by G. A. Cheston [Ch]. Here we present a parallel algorithm for this problem that is based on the st-numbering and bipartition of biconnected graphs.

The idea of centroided tree algorithm is as following. Let t be a neighbor of s and bipartition the G by $s_1=s$, $s_2=t$, $n_1=\lceil 1/2N \rceil$ and $n_2=\lfloor 1/2N \rfloor$. Since t is connected to s, a centroided tree can be formed if we combine the intree with root t and the outtree with root s (and then disregard the directions of the edges) into a spanning tree of the original graph that satisfies: (1) s is the root and (2) t is the child of s.

```
Procedure CENTROIDED_TREE
    {Input: Biconnected graph $G=(V,E)$ and vertex $s$ of $V$;
    Output: A spanning tree $T$ of $G$ s.t. $s$ is the centroid.}
1. $t=\min\{x|(s,x)\in G\}$;  {choose a child of $s$ to be $t$}
2. Computer an st-numbering $f$ of $V$ for $\{s,t\}$;
3. {Form a $DAG$}
   For each edge $(u,v)$ in parallel do
       Assign direction from $u$ to $v$ if $f(u)<f(v)$;
4. Bipartition $G$ with $s_1=s$, $s_2=t$, $n_1=\lceil 1/2N \rceil$ and $n_2=\lfloor 1/2N \rfloor$ into $V_1$, $V_2$;
5. { Construct an outtree $T_{out}$ with root $s$ }
   For each vertex $u$ of $V_1-\{s\}$ in parallel do
       $(v,u)\in T_{out}$ if $f(v)=\min\{f(x)|(x,u)\in DAG\}$;
6. { Construct an intree $T_{in}$ with root $t$ }
   For each vertex $u$ of $V_2-\{t\}$ in parallel do
       $(u,v)\in E_{in}$ if $f(v)=\max\{f(x)|(u,x)\in DAG\}$;
7. Construct a centroided tree $T=(V,E_T)$ with root $s$ s.t.
       $E_T=E_{in}\cup E_{out}\cup\{(s,t)\}$;
       Remove the direction for every edge;
```

Algorithm 5.3.2 Algorithm of centroided tree construction

The procedure CENTROIDED_TREE (Algorithm 5.3.2) shows how the method proceeds. Figure 5.3.1 illustrates the method by an example. Figure 5.3.1-a gives a biconnected graph G and a specified vertex s. An st-number of $\{s,t\}$ and a DAG are shown in Figure 5.3.1-b. The bipartition is done by $s_1=s$, $s_2=t$ and $n_1=\lceil 1/2N \rceil=6$ by step 4. The outtree and the intree that imply two subtrees are formed by steps 5 and 6 respectively based on st-numbers and the DAG directly, as shown in figure 5.3.1-c. The combining is trivial which is done by step 7.

Except step 2 which computes the st-numbering of a biconnected graph, all the other steps of Algorithm 5.3.2 can be performed in O(log$N$) time using N/log$N$

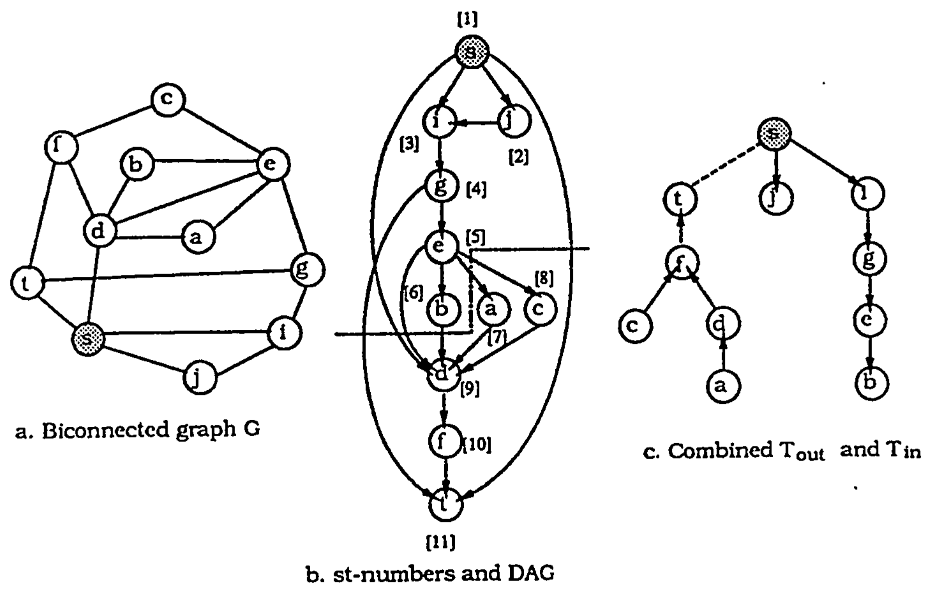


Figure 5.3.1 Centroided tree construction

processors on EREW P-RAM since only operations of comparison and finding minimum (maximum) are performed. Therefore the st-number computing dominates the complexity of centroided tree construction for all different P-RAM models.

Claim 5.3.2: Procedure CENTROIDED_TREE constructs a spanning tree of G with s as the centroid.

Proof: First we notice that T is a tree since edge $(s,t) \in G$ connects two subtrees that partitions the vertices of G. From bipartition we know that each of the subgraphs has at most half of the vertices and s, t belong to different subgraphs. The intree and the outtree are of size at most half of the total vertices. By taking s away from T no subtree has vertices more than half of the total vertices. Hence T is a spanning tree of G with s as the centroid.□

If the problem is to generate a spanning tree with vertex set Q as the centroid of the tree s.t. vertices of Q are connected, we can easily reduce G to G' by shrinking Q into one vertex q and (q,v) is in G' if there is (x,v) and $x \in Q$ in G. Surely G' is connected. If there is any articulation point in G' it must be q. Then every biconnected component contains q. For each biconnected component construct a spanning tree with q as the centroid. Combining those centroided trees forms a centroided tree of G'. Then we expend q into a spanning tree of Q and mark only one edge between vertex in Q and vertex $u \in V-Q$ if there is an edge (q,u) in G'. The resulting tree has Q as the centroid. Obviously every operation mentioned above can be done within the bound of st-numbering.

5.3.3 Centered Trees

The problem centroided tree of a biconnected graph is to construct a tree out of a biconnected graph so the specified vertex (or connected vertex set) is located at the optimal position in the sense of minimum total distance. Instead of distancesum as discussed in the previous section, here we consider the *eccentricity* of vertex u, which is the largest distance from u to any vertex in $V-u$. The eccentricity of vertex set S is the largest distance from any vertex in S to any vertex in $V-S$.

The problem of constructing a centered tree of a biconnected graph is to find a spanning tree of the graph such that a specified vertex is the center. G. Cheston, A. Farley, S. Hedetniemi and A. Proskurowski represented an algorithm run in $O(N^3)$ time to find a centered tree for biconnected graphs in 1989 [CFHP].

In this section we give a parallel algorithm for this problem. This parallel algorithm implies a better serial algorithm with complexity of $O(M)$ comparing to the known $O(N^3)$ method.

Consider a tree T_{cen}:

1. Vertex s is the root and s has at least two children;

2. Let $c_1,...,c_k$ be the children of s; Let $t_1,..., t_k$ be the subtrees with roots $c_1,...,c_k$ respectively; Let $h_1,...,h_k$ be the height of $t_1,...,t_k$ respective; W.l.o.g. let h_1 be the largest and h_2 be the second largest of h_i's then h_1-h_2≤1.

Claim 5.3.3.1: Vertex s is a center of T_{cen}.

Proof: Assume on the contrary s is not a center but vertex u is a center. Vertex u must be in one of t_i, 1≤i≤k. If u is in t_1 then the distance from u to the farthest leaf of t_2 is greater or equal to h_2+2 so that eccentricity of u is at least h_2+2. If u is not in t_1 then the distance from u to the farthest leaf of t_1 is greater or equal to h_1+2 so that the eccentricity of u is at least h_1+2 that is at least h_2+2. We know that the eccentricity of s is h_1+1≤h_2+2. It is a contradiction to the assumption.□

Our method of centered tree construction is to find such a tree T_{cen} in G . Let the *DAG*, the outtree with root s and the intree with root t be achieved by the same way as we described is Section 5.3.2. Let H_{out} and H_{in} be the heights of the outtree and the intree of Algorithm 5.3.2 respectively. We have the following claim.

Claim 5.3.3.2: There exists a cut of the *DAG* that makes H_{out}-H_{in}=1 or H_{out}-H_{in}=0.

Proof: Assume a cut of the *DAG* leaves k vertices in the outtree and leaves $N-k$ vertices in the intree and H_{out}-H_{in}≤0. Such k must exist since k=1 satisfies the condition. Consider the cut of $k+1$ vertices for the outtree and $N-k-1$ vertices for the intree. The only change compared to the cut of k vertices is the vertex u with $f(u)$=k is moved from the outtree to the intree (and correspondent edge changes of cause). The changes of H_{in} and H_{out} could have four cases:

1. H_{in} increases one and H_{out} decreases one;
2. H_{in} increases one and H_{out} does not change;
3. H_{in} does not change and H_{out} decreases one;
4. Neither H_{in} nor H_{out} is changed.

Let $D(k)=H_{out}-H_{in}$ for the outtree of size k and intree of size $N-k$. From the above cases we notice that $D(k)$ is a non decreasing function. Notice that $D(1)\leq 0$, $D(N-1)\geq 0$ and $D(k+1)-D(k)<=2$. There must be a $1\leq k<N$ s.t. $D(k)=1$ or $D(k)=0$.□

The idea to find the required cut is as following. Let t be a neighbor of s. First we construct an intree of size N with root t and an outtree of size N with root s based on the st-numbering and the DAG that are obtained by the same way as discussed in the previous section. Then the level for every vertex is computed in both trees. Based on the levels, an $h_{in}(u)$ for every vertex u in intree can be achieved by the scheme similar to prefix

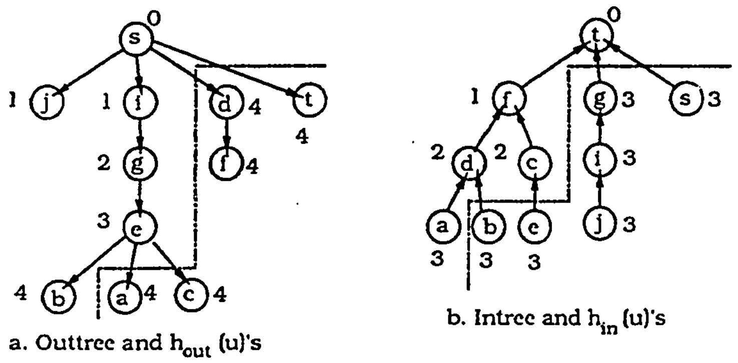


Figure 5.3.2 Example of centered tree

sum computation s.t. $h_{in}(u)$ indicates the height of the intree with vertices having st-number greater or equal to $f(u)$. Similarly, the $h_{out}(u)$ for every vertex u in the outtree can be computed. The cut will surely be found by knowing $h_{out}(u)-h_{in}(u)$ for every vertex u. The Procedure CENTERED_TREE, Algorithm 5.3.3, shows the parallel algorithm for finding a centered tree of a biconnected graph.

We take the same example in Figure 5.3.1-a. Node t is a neighbor of s. The st-numbering and the DAG are shown in Figure 5.3.1-b. The intree and the outtree of size N are shown in Figure 3.5.1 in which the $h_{in}(u)$'s and $h_{out}(u)$'s are also illustrated. Notice that $D(b)=h_{out}(b)-h_{in}(a)=4-3=1$. Therefor $k=6$. The cut is visually shown in Figure 5.3.2. The combined tree T is the same as the centroided tree shown in Figure 5.3.1-c.

Claim 5.3.3.3: The spanning T' constructed by CENTERED_TREE has s as the center.

Proof: First we know that T' is a spanning tree of G since $(s,t) \in G$ and it connects two subtrees that partitions V. Step 8 can be successfully performed because there must be a k, $1 \le k < N$, s.t. $D(k)=1$ or $D(k)=0$, by Claim 5.3.3.2. Taking away s from the outtree T_{out}, the highest subtree left has the height one less than T_{out}. It has height either the same as the intree T_{in} or one less than T_{in}. Therefore in T', the subtree with root t, which is derived from T_{in}, is the highest subtree with root t as a child of s and the subtree derived from the highest subtree of T_{out} is the second highest subtree. According to the discussion of T_{cen} at the beginning of this section, T' is a spanning tree of G with s as the center.□

Different from steps in CENTROIDED_TREE, here we need to compute level for each vertex and h_{in} (h_{out}) in the intree with root t (the outtree with root s) for all the vertices. This can be done in O(log$N$) time using N/log$N$ processors. Other operations are

```
Procedure CENTERED_TREE:
    {Input: A biconnected graph $G=(V,E)$ and vertex $s$ of $V$;
    Output: A spanning tree $T'$ of $G$ with $s$ as the center.}
1. $t=\min\{x|(s,x)\in G\}$;
2. Computer an st-numbering $f$ of $V$ for $\{s,t\}$;
3. {Form a $DAG$}
   For each edge $(u,v)$ in parallel do
       Assign direction from $u$ to $v$ if $f(u)<f(v)$;
4. { Construct an outtree $T_{out}$ with root $s$ of size $N$ }
   For each vertex $u$ of $V$-$\{s\}$ in parallel do
       $(v,u)\in T_{out}$ if $f(v)=\min\{f(x)|(x,u)\in DAG\}$;
5. { Construct an intree $T_{in}$ with root $t$ of size $N$ }
   For each vertex $u$ of $V_2$-$\{t\}$ in parallel do
       $(u,v)\in E_{in}$ if $f(v)=\max\{f(x)|(u,x)\in DAG\}$;
6. For each vertex $u$ of $V$ in parallel do
       $l_{in}(u)$=level of $u$ in the intree;
       $l_{out}(u)$=level of $u$ in the outtree;
7. For every vertex $u$ of $V$ in parallel do
       $h_{in}(u)=\max\{l_{in}(x)|f(x)\geq f(u)\}$;
       $h_{out}(u)=\max\{l_{out}(x)|f(x)\leq f(u)\}$;
       $D(u)=h_{out}(u)-h_{in}(v)$ where $f(v)=f(u)+1$;
8. If there is a $x$ s.t. $D(x)=1$ then $f(k)=\min\{f(x)|D(x)=1\}$;
   else $k=\min\{x|D(x)=0\}$;
9. Reduce $T_{out}$ to size of $k$ by $u\in T_{out}$ if $f(u)\leq k$;
   Reduce $T_{in}$ to size of $N-k$ by $u\in T_{in}$ if $f(u)>k$;
10. Construct a centered tree $T'=(V,E_T)$ with root $s$ s.t.
       $E_T=E_{in}\cup E_{out}\cup\{(s,t)\}$;
       Remove the direction for all the edges;
```

Algorithm 5.3.3. Algorithm of centered tree construction

minimum and maximum finding, which can be computed in O(log$N$) time using N/log$N$ processors. Hence the complexity of CENTERED_TREE is the same as CENTROIDED_TREE.

If the problem is to generate a spanning tree with vertex set Q as the center of the tree s.t. vertices of Q are connected, we can use the method similar to the case in centroided tree construction. For each biconnected component, instead of running CENTROIDED_TREE we need to run CENTERED_TREE. The rest will be the same. The complexity is the same as shown above.

This algorithm implies a serial algorithm of time complexity O(M) to find a centered tree for a biconnected graph. This is better than the known method that runs in O(N^3) time [CF].

3.6 Strong Orientation

The *strong orientation* of an undirected graph assigns direction for every edge of the graph so that every vertex can reach any vertex of the graph by some directed path (i.e., the directed graph is strongly connected). Obviously the strong orientation can be done only for bridgeless graphs.

The strong orientation has a linear sequential algorithm based on DFS of the graph[A]. M. Atallah gave an O(log$N$) time by O(N^3) processors on CRCW for this problem which runs in O(N^3/p+$\log^2 N$) time by p processors on CREW[A]. Vishkin gave an O(log$N$) time by N+M processors on CRCW and an alternative implementation of the algorithm in O(N^2/p) time by $p \leq N^2/\log^2 N$ processors on CREW[V]. Here we present a very simple method which works on EREW P-RAM.

Notice the orientation from the previous section ensures every vertex having at least one incoming and one outgoing edges except s and t. This implies any vertex can reach t and s can reach any vertex by some directed paths. If we choose a pair of adjacent vertices to be s and t, assign the edge (s,t) from t to s will give a strong orientation for a biconnected graph.

In general, the bridgeless graph may have some articulation points. Every biconnected component of a connected bridgeless graph has at least one articulation point which belongs to another biconnected component. The edges of the graph are partitioned by the biconnected components. Combining the strongly oriented biconnected components yields a strong orientation for the graph. The only thing we need to show is that the total number of vertices of all the biconnected components of a bridgeless graph with N vertices is $O(N)$, so that the number of processors we need is the same as for a biconnected graph. Let G be a connected bridgeless undirected graph and G' be the graph consists of all the biconnected components of G.

Lemma 5.3.4: Each biconnected component of G adds one more vertex in G'.

Proof: A biconnected component contains only one articulation point is called a *pendant*. G has at least two pendants. We construct G' from G in the following way: if a biconnected component B is a pendant with articulation point a_B then move B from G to G' s.t. vertex a_B is duplicated into two, one is in B of G' and one is still in G but not necessary an articulation point. By moving all the pendants the new pendants are generated. At last G will be biconnected, hence moving G to G' ends the construction. Obviously each pendant splits one articulation point, hence each pendant adds one more vertex in transmitting from G to G'. Precisely, if G has k biconnected components then G' has $(k-1)$ vertices than G.□

Corollary 5.3.1: G' has $O(N)$ vertices.

Proof: Followed by Lemma 5.3.4.□

Clearly the st-numbering and biconnected components are the main parts of the strong orientation. Therefore the spanning tree construction dominates the complexity.

CHAPTER 6

MINIMUM CUTSETS FOR REDUCIBLE GRAPHS

6.1 Introduction

Processes, in particular computer programs, are often represented by directed graphs. The analysis and manipulation of systems modeled as graphs require a selection of minimal subset of the vertices that cuts all the cycles in the graphs. This set is referred to as a minimal cutset (or feedback vertex set). Two common application areas that use cutsets are program verification and code optimization [HU1][F]. Reducing the size of the vertex set, that cuts all the cycles, usually leads to a simpler and more efficient analysis.

The problem of finding a minimal cutset in a directed graph is NP-complete [K]. A. Shamir showed that for an important class of directed graphs, reducible graphs , there is a serial linear time algorithm for finding a minimum cutset [Sh]. Most directed graphs produced from flow charts of computer programs, (or flow graphs), are reducible. We refer the reader to the paper by A. Shamir [Sh] for more background for the problem.

The ongoing research in the area of vectorizing and parallelizing compilers brought more interest in reducible graphs. Recognizing loops and optimizing execution around them are of key importance in this area. Recently J. Lucas and M. Sackrowitz showed that testing whether a given directed graph with N vertices is reducible can be done in $O(\log^2 N)$ time using $O(N^3/\log N)$ PE's [LS].

In this chapter we present a parallel algorithm for finding a minimum cutset in a reducible graph in $O(\log^3 N)$ time using $O(N^3/\log N)$ PE's on EREW P-RAM. Finding an efficient parallel algorithm for this problem is particularly interesting because the sequential algorithm employs a Depth First Search Tree [Sh]. Depth First Search is believed to be inherently sequential [R]. The pruning decomposition search is applied to the directed graphs here.

First we introduce some definitions related to the problem. We refer to the papers by M. Hecht and J. Ullman [HU1][HU2], and the paper by A. Shamir [Sh], for more definitions and detailed discussions of reducible graphs and their properties.

We use $G(V,E)$ to denote a directed graph. V is the set of vertices and $E \subseteq V \mathrm{x} V$ is the set of edges. A vertex is a *cutpoint* of a cycle if by taking it away, the cycle is broken. A cutset of a graph $G(V,E)$ is a subset of the vertices $S \subseteq V$ such that any cycle in graph G contains at least one vertex from S. A *minimum cutset* of a given graph is a cutset S' such that any other cutset for this graph has at least the same number of vertices.

A graph $G(V,E,r)$ is a *rooted graph* if it has a vertex r, *root*, with the property that for any $v \in V$ there is a path from r to v. Vertex v *dominates* vertex u in a rooted graph $G(V,E,r)$ if every path from r to u passes v. The *DAG* of a directed graph G is a maximal acyclic graph that is a subgraph of G.

The following are equivalent for a *reducible graph*[Sh]:

1. A directed rooted graph $G=(V,E,r)$ is reducible;
2. The DAG of G is unique;

3. The set of edges E of graph G can be partitioned into two subsets E_1 and E_2. E_1 is the set of edges of the DAG of G. $E_2=E-E_1$ and for every edge (u,v) in E_2 either $u=v$ or v dominates u in G, as well as in the DAG of G.

We now define a few terms that will help us in presenting the serial algorithm [Sh]. Let v be a vertex in the reducible graph. Vertex v is a *head* if there is an edge (u,v) in E_2 for some vertex u. Vertex u is a *tail*. Let part of the cycles in $G(V,E)$ be cut by a set $S \subseteq V$ of vertices. We denote head v as *active* if there is some path from v to a corresponding tail that is not cut by any vertex from S in the DAG of G. An active head is *maximal* if none of its proper descendants in the DAG of G is an active head.

```
Procedure Serial_Min_Cutset:
{Serial algorithm for finding minimum cutset $S$ for a given reducible graph $G$ [Sh].}
1.  $S <- \emptyset$;
2.  Select a maximal active head $v$ in $G$ with respect to the current set $S$. If there is
    none, stop; otherwise set $S <- S \cup \{v\}$ and repeat step 2.
```

Algorithm 6.1.1 Serial algorithm for minimum cutset

Procedure Serial_Min_Cutset (Algorithm 6.1.1) is based on the following principle: If a vertex u is a head and there is a path from u to one of its tails such that every vertex (except u itself) on the path has been determined not to be in the cutset, then u is determined to be in cutset. We later refer to this principle as the *decision principle*. It implies that an algorithm in which every vertex in cutset is selected according to the decision principle gives a minimum cutset. The complexity of this serial algorithm is $O(|V|+|E|)$. pp In Section 6.2 we present a parallel algorithm for finding the minimum cutset in a special type of reducible graph. The main result is presented in Section 6.3. We also present a heuristic for finding a cutset in general graphs in section 6.4.

6.2. Minimum Cutset of a Branch

In developing our parallel algorithm we first handle a special type of reducible graphs. We name this type of reducible graph a 'branch'. A reducible graph K(V,E) is a *branch* if the DAG of K is a chain.

The following are equivalent for a branch.

(1) K is a reducible graph and it has unique DAG that is a chain.

(2) The edges of K can be partitioned into chain edges and back edges.

(3) Each back edge creates a cycle in K.

In a branch K, let U_1 and U_2 denote the vertex sets of cycles C_1 and C_2 respectively. If U_1 is contained in U_2 then a cutpoint of C_1 is also a cutpoint of C_2.

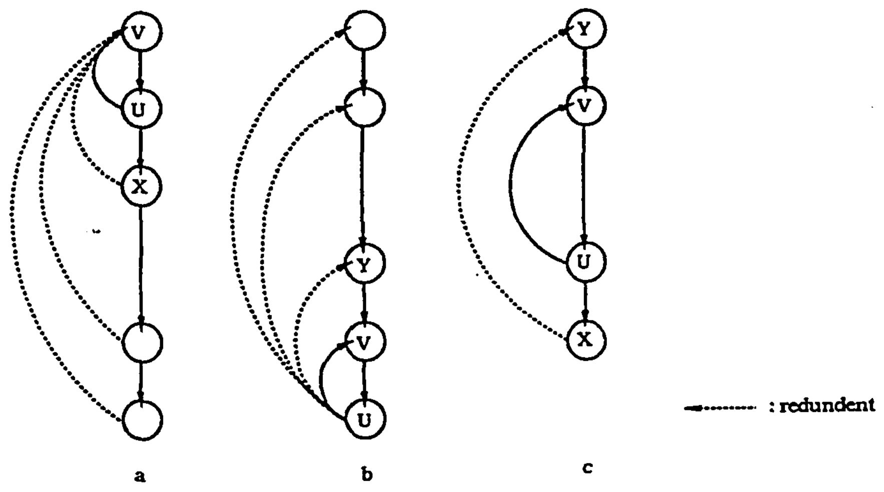


Figure 6.2.1 Example of redundant edges

Let E_1 denote the set of edges of the DAG of K (the edges of the chain) and E_2 denote the set of backedges. The vertices of K are labeled according to the postorder of the DAG of K (In Shamir's serial algorithm preorder is used. Our algorithm can work by either postorder or preorder.). Applying the decision principle to the postorder labeled DAG of K we observe the following property: let (u,v) and (x,y) be two backedges of K, if $x \leq u$ and $y \geq v$ then backedge (x,y) is redundant for the purpose of finding a minimal cutset in K and can be ignored (Fig. 6.2.1).

After deleting all the redundant back edges from a branch K we get a graph K' that we call a *simple branch.* A simple branch K' with n vertices has at most (N-1) back edges. Each vertex can be a head at most once and be a tail at most once. It is easy to see that a minimal cutset of a simple branch is also a minimal cutset of the corresponding branch.

Let C_u denote the cycle of a simple branch K' that is formed by back edge (v,u).

Let t_u denote the tail of head u in simple branch K'.

Lemma 6.2.1: If a head x is in cycle C_u and $x<v<u$ where v is also a head, then: (a) x is in cycle C_v, and (b) v is in cycle C_u.

Proof: (See Fig. 6.2.2-a)

a. Assume to the contrary that x is not in cycle C_v, then if $v>x$ and v is a head, the tail of v, t_v, must be greater than x. Then any back edge (y,u) s.t. $t_v<y\leq t_u$ would have been deleted in the construction of K', since back edge (t_u,v) and (y,u) have $t_u>y$ and $v<u$. Hence x is not in C_u, contradiction.

b. Assume to the contrary that v is not in cycle C_u, then u cannot reach x. In other words, x is not in cycle C_u, contradiction.□

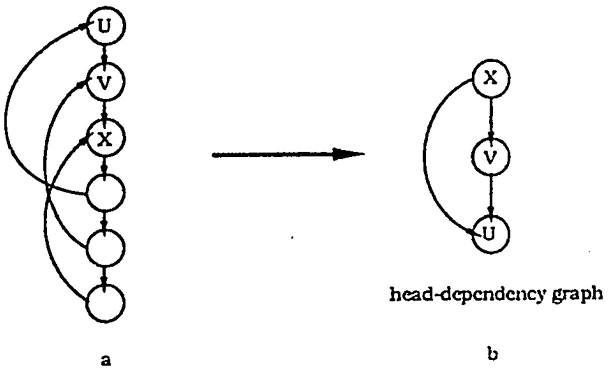


Figure 6.2.2 Head-dependency graph

We now construct D, the *head–dependency* graph of a simple branch K'. The vertices of a head-dependency graph are the heads of the simple branch and (u,v) is an edge in D iff u is in cycle C_v. Fig. 6.2.2-b is an example of the head-dependency graph. We proceed to show how to find a cutset of a simple branch (hence a branch) using the head-dependency graph of a simple branch (Algorithm 6.2.1).

```
Procedure Serial_Branch_MinCut (D)
{Input is a head-dependency graph D}
1.  $S_1 \leftarrow \{v \mid v$ has no incoming edge$\}$;
2.  $S_2 \leftarrow \emptyset$;
3.  repeat
4.      if a vertex $u$ has an immediate predecessor that is in $S_1$, then put $u$ in $S_2$.
5.      if a vertex $u$ has all its immediate predecessors in $S_2$ then put $u$ in $S_1$.
6.  until every vertex is either in $S_1$ or in $S_2$.
7.  $S_1$ is a cutset.
```

Algorithm 6.2.1 Serial algorithm for branch minimum cutset

Procedure Serial_Branch_MinCut uses two sets to partition the vertices of the head-dependency graph D. In steps 1 and 2 of the procedure we initialize these sets. Clearly all vertices of head-dependency graph D are candidates for membership in a minimal cutset. Set S_1 is initialized with vertices that have to be in the cutset. These are the vertices that have no incoming edges in D. Set S_2 has vertices that are determined not to be in the cut set as members and it is initially set to be empty. The loop in lines 3 to 6 is executed until all vertices are determined either not in the cutset or as members of the minimal cutset. The correctness of the algorithm is proved in the following lemmas.

Lemma 6.2.2: In every iteration of the loop, (lines 3-6, Algorithm 6.2.1), there is at most one immediate predecessor in S_1 for any vertex.

proof: Assume to the contrary that vertices x,y are in S_1 and both are immediate predecessors of vertex u. We can assume without loss of generality that $x>y$, by Lemma 6.2.1 and the construction of the head-dependency graph, y is also an immediate predecessor of x. According to Algorithm 6.2.1, x should be in S_2, contradiction.□

Lemma 6.2.3: A vertex u is placed in S_1 by procedure Serial_Branch_MinCut, (Algorithm 6.2.1), iff there is a path in K' from u to t_u such that every vertex on that path (except u) is either in S_2 or is non-head.

Proof: Assume that vertex u is in S_1 then all its immediate predecessors in D are in S_2. It implies that every head in cycle C_u other than u is not in cutset; that is, there is a path from u to t_u with every vertex either in S_2 or non-head. Clearly, if there is a path in K' from u to t_u such that every vertex on the path is either in S_2 or is not a head then u is placed in S_1.□

Lemma 6.2.4: S_1 is a minimum cutset of simple branch K'.

Proof: Follows from Lemma 6.2.3, the construction of head-dependency graph and decision principle.□

Corollary 6.2.1: S_1 is a minimum cutset of K.

Proof: Follows from Lemma 6.2.4 and previous discussion.□

In our discussion we assume that the information for the branch includes the partition of its edges to DAG edges and back edges. Although this information is not typically available as input, it is available in our context as we shall show later. To reduce a branch to a simple branch takes O(M) time where M is the number of edges in the branch. The construction of the head-dependency graph takes O(m) time where m is the number of edges in the head-dependency graph. Procedure Serial_Branch_MinCut, (Algorithm

```
Procedure P_BRANCH_CUTSET (K);
{ Input is a branch K. Vertices are labeled in postorder of its DAG. Edges are partitioned
into DAG edges and back edges. }
A.  Simple_Branch;  {Produce simple branch K' by deleting redundant back edges.}
B.  Head_Graph;     {Construct head-dependency graph from K'.}
C.  {Find a minimum cutset for branch K}
1   Cutset of K←∅;
2   for each block of the head-dependency graph in parallel do
3           Block_Cutset;  {Determine the minimum cutset for each block.}
4   Cutset of K← Cutset of K∪ cutset of block;
end;{P_BRANCH_CUTSET}
```

Algorithm 6.2.2 Skeleton for branch minimum cutset

6.2.1) takes O(m) time, since every edge is checked by some vertex once. Thus we can find a minimum cutset of a branch in linear time.

A head-dependency graph may contain several weakly connected components. We call such a component a *block*. Procedure P_BRANCH_CUTSET is a parallel algorithm for computing a minimum cutset for a given branch K, (Algorithm 6.2.2). It is a parallel implementation of the ideas in Procedure Serial_Branch_MinCut, Algorithm 6.2.1, and the discussion leading to it.

There are three main steps in Procedure P_BRANCH_CUTSET, (Algorithm 6.2.2). In the subsequent paragraphs we discuss the parallel implementation of each of these steps. It is instructive to follow the algorithm with an example. A branch with 11 vertices (Fig 6.2.3-a) is simplified in step A of Procedure P_BRANCH_CUTSET to produce

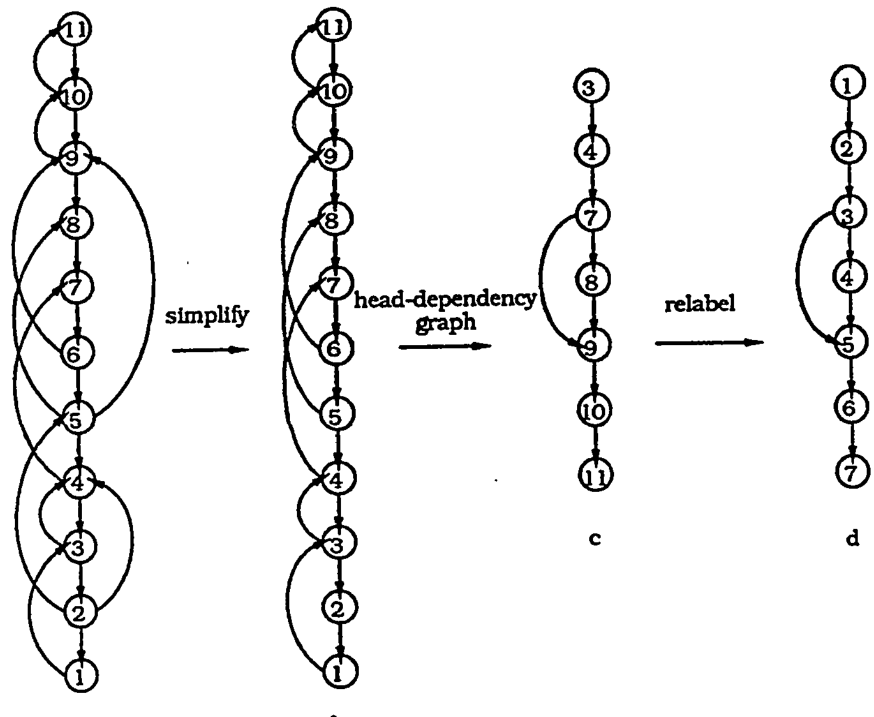


Figure 6.2.3 Construction of head-dependency graph

```
procedure Simple_Branch;
1.    if a vertex $u$ is a head then in parallel do
          $v \leftarrow \max\{\text{tails of } u\}$;   {find the closest tail}
      Delete the backedges $(w,u)$ if $w \neq v$;{case of Fig. 3.1-a}
2.    if a vertex $v$ is a tail then in parallel do
          $u \leftarrow \min\{\text{heads of } v\}$;   {find the closest head}
      Delete backedge $(v,w)$ if $w \neq u$; {case of Fig.3.1-b}
3.    Sort backedges in increasing order according to the tails;
      for each backedge $(u,v)$ in parallel do {case of Fig.3.1-c}
      if the next backedge $(x,y)$ has $x > u$ and $y < v$ then delete $(u,v)$;
end; {Simple_Branch}
```

Algorithm 6.2.3 Algorithm for simple branch

the simple branch in Fig. 6.2.3-b. This simple branch is processed in step B to produce the head-dependency graph (Fig. 6.2.3-c).

Procedure Simple_Branch (Algorithm 6.2.3) has 3 steps. each step corresponds to one of the observations presented earlier in Fig. 6.2.1. The implementation of these steps is quite straight forward. In our example (Fig.6.2.3), vertex 9 is head for backedges (5,9) and (6,9). Edge (5,9) is found redundant and is deleted in step 1. Step 1 also deletes edge (2,4). Processing in step 2 doesn't discover any new redundant backedges. In step 3 edge (2,5) is found redundant after comparing it to edge (3,4).

Procedure Simple_Branch, (Algorithm 6.2.3), can perform its job in O(log$N$) time using $N^2/\log N$ PE's. Using an adjacency matrix to represent the graph, we can find the tail with the minimal label for each vertex in O(log$N$) time with O($N/\log N$) PE's [H][DNS]. Deletion of the redundant edges found in step 1 or step 2 can be done in O(log$N$) time using $N/\log N$ PE's, for each head. Therefore steps 1 and 2 of the procedure

```
procedure Head_Graph;
{Input is a simple branch $K'$. Output is a set of head-dependency graphs.}
1)    {initialize}
      Mark all heads;
      Eliminate all non-head vertices from consideration;
      Initiate a head-dependency graph $G^*=(N^*,E^*)$ where
            $N^*\leftarrow$ {all heads};
            $E^*\leftarrow\emptyset$;
2)    {build  the head dependency graph}
      for each head w in parallel do
            for each backedge $(u,v)$ in parallel do
                  if $v>w\geq u$ then $E^* \leftarrow E^* \cup \{(w,v)\}$;
3)    {identify blocks}
      Every vertex in $G^*$ without incoming edges is the beginning of a block;
      Every vertex in $G^*$ without outgoing edges is the end of a block;
      for each beginning of a block u in parallel do {Construct a block}
            block_end($u$)$\leftarrow\min\{v \mid v$ is an end of block and $v\geq u\}$;
            vertices of block$\leftarrow\{v \mid u\leq v\leq$block_end($u$) and $v$ is a head$\}$;
            edges of block$\leftarrow\{(x,y) \mid x,y\in E^*\}$;
end.{Head_Graph}
```

Algorithm 6.2.4 Algorithm for head-dependency graph

```
procedure Block_Cutset;
{Input is a head dependency graph $D(N_D, E_D)$. Output is the cutset.}
1.  Relabel the vertices according to their ranks in $D$;
2.  for every vertex u in parallel do min_adjacent($u$)$\leftarrow$min{$v|(v,u)\in E_D$};
3.  for every vertex u in parallel do construct $g_u(N_u,E_u)$ where
    $N_u \leftarrow \{y | y\in N_D$ and $y \geq u\}$;
    $E_u \leftarrow \emptyset$;
    for each vertex $y > u$ in parallel do {each vertex has one in-edge}
        if min_adjacent($y$)$\leq u$ then $E_u = E_u \cup (u,y)$
            else $E_u \leftarrow E_u \cup$(min_adjacent($y$),$y$);
4.  for all vertices $i$ and for all $g_u$ in parallel do
      S($i,u$)$\leftarrow$ 'undetermined';  { S($i,u$) denotes the status of vertex $i$ in $g_u$ }
    for all $g_u$ in parallel do
        S($u,u$)$\leftarrow$'in-cutset';
        max-in($u$)$\leftarrow u$;
5.  for i=1 to $\lceil \log N_D \rceil$ do
        for each $g_u$ in parallel do
            5.1 for all vertices $x\in N_u$ do
            cp($x,u$)$\leftarrow$0; { zero the number of copies requested }
            5.2 for each edge ($x,y$) in parallel do
            {directly identify vertices that are not in cutset}
             if ($u \leq x < u+2^{i-1}$) and (($u+2^{i-1}$)$\leq$y$<u+2^i$) then
                5.2.1 if S($x,u$)='in-cutset' then S($y,u$)$\leftarrow$'out';
                5.2.2 if S($x,u$)='out' and $x<$max-in($u$) then S($y,u$)$\leftarrow$'out';
            5.3 $w$=min{$x$| S($x,u$)='undetermined' and $x \leq u+2^i$};
            5.4 if there is such $w$ then do
                5.4.1 cp($u,w$)$\leftarrow$1;
                5.4.2 {to avoid conflicts in 5.4.3 }
                  make $\sum_{x\in N_u}$ cp($x,u$) copies of S(*,$u$);
                5.4.3 {merge in values from $g_w$}
                  for each $x\in N_u$ in parallel do
                      if $w \leq x < u+2^i$ then S($x,u$)$\leftarrow$S($x,w$);
            5.5 max-in($u$)$\leftarrow$max{$y$|S($y,u$)=in-cutset};
6.  Cutset of the block$\leftarrow${$u$| S($u$,1)=in-cutset};
end.{Block_Cutset}
```

Algorithm 6.2.5 Algorithm for block cutset

can be performed in O(log$N$) time using N^2/log$N$ PE's for the whole graph. There are at most N-1 edges left in step 3 of the procedure, sorting can be performed in O(log$N$) time with $N^2/\log N$ PE's [C]. The edge deletion can be done in constant time.

Continuing with our example we use the simple branch in Figure 6.2.3-b as input for our head dependency construction procedure (Algorithm 6.2.4). Step 1 is an initialization step. Clearly only head vertices of the input branch are in the head-dependency graph. The set of edges is initially empty (step 1). In our example the set of head vertices is $\{3, 4, 7, 8, 9, 10, 11\}$. The relationship between these vertices is computed in step 2 of procedure Head_Graph, (Algorithm 6.2.4). In this stage each backedge is considered against all head vertices to check whether the vertex is on the cycle created by this edge. Edges are added to show this relationship. For example, in Fig. 6.2.3-c, backedges (3,4), (4,7), (7,8), (7,9), (8,9), (9,10) and (10,11) are in head dependency graph. In step 3 of the procedure we identify the blocks in the graph. In our example there is only one block.

It is easy to see that procedure Head_Graph, (Algorithm 6.2.4), can be performed in O(log$N$) time using O($N^2/\log N$) PE's on the EREW P-RAM model of computation. The initialization step can be done in constant time using O(N) PE's. In step 2, a backedge may produce O(N) edges in the head-dependency graph. Since the input is a simple branch there are no write conflicts. This step can be performed in O(1) time using O(N^2) PE's or in O(log$N$) time using O($N^2/\log N$) PE's. In step 3, finding the minimum can be done in O(log$N$) time using o($N^2/\log N$) PE's for all vertices that are the beginning of blocks. Block identification can be done in O(1) using O(N^2) PE's or O(log$N$) using O($N^2/\log N$) PE's.

In the first step of procedure Block_Cutset, (Algorithm 6.2:5), we relabel the vertices for each block (The root of the block is determined in step 3 of procedure Head_Graph, (Algorithm 6.2.4)). For our example the relabeling is presented in

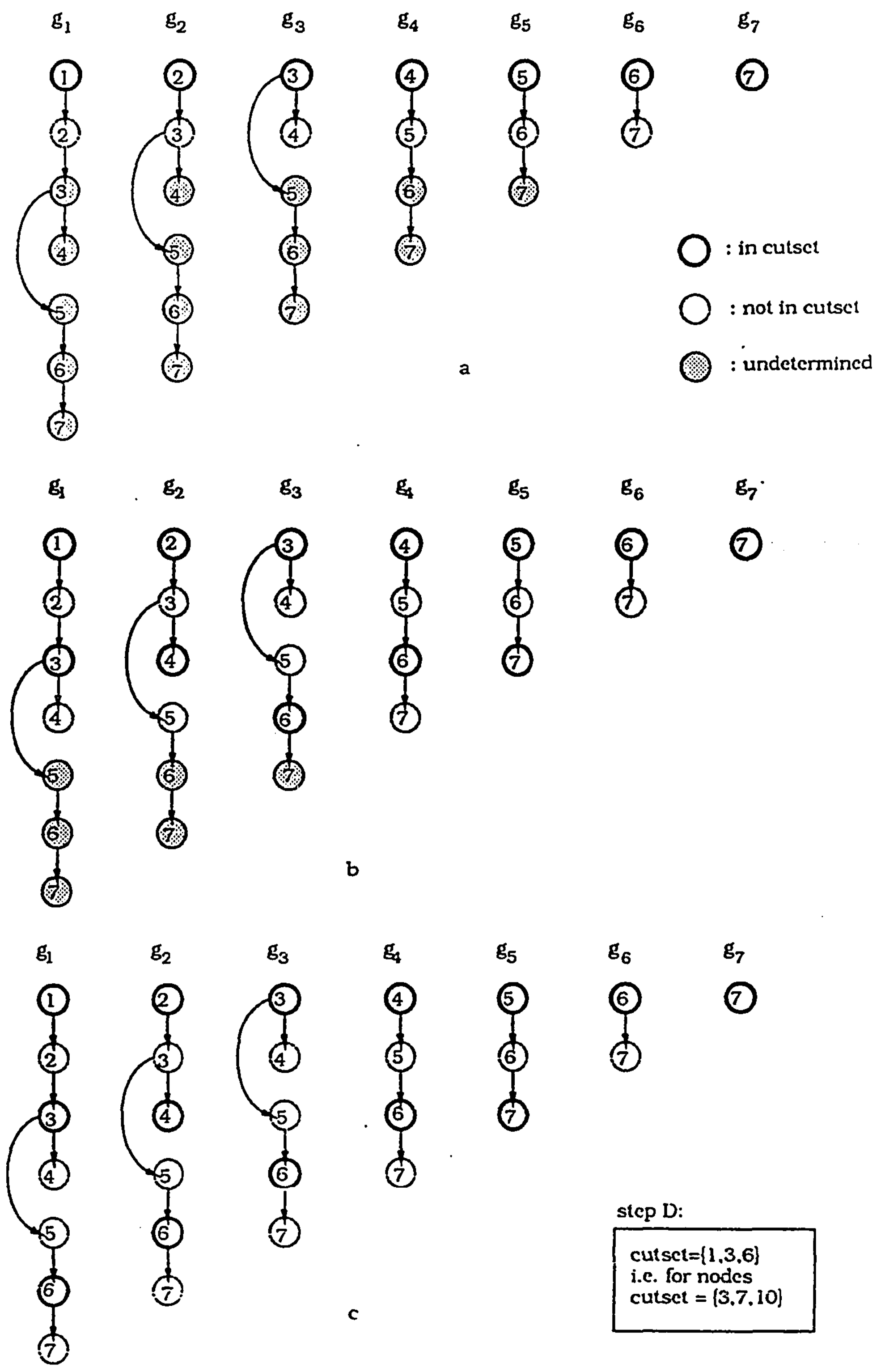


Figure 6.2.4 Determine a cutset of a branch

Fig.6.2.3-d. In step 2 of the procedure we compute the variable min_adjacent. This

variable is used in step 3 where we construct a subgraph $g_u(N_u,E_u)$for each vertex u in the input graph. The min_adjacent value of vertices 2, 3, 4, 5, 6 and 7 are 1, 2, 3, 3, 5, and 6 respectively. $g_u(N_u,E_u)$ is the subgraph rooted at u. We will see these g_u's allow us to maximize the parallel computation without conflicts. Each vertex is connected from the min_adjacent vertex computed in step 2. Adjustments are made for cases where min_adjacent is smaller than the root u, (Fig. 6.2.4). S(i,u), the status of vertex i in subgraph g_u, is initially set to 'undetermined'. This is done for all vertices, in step 4 of the procedure. In this step we also initialize S(u,u) to 'in_cutset' and set the max_in variable associated with each root vertex u to u. In our example, consider the construction for vertex 5, $g_5(N_5, E_5)$, $N_5=\{5, 6, 7\}$, S(5,5)$\leftarrow$'in-cutset', S(6,5)=S(7,5)='undetermined' and max_in(5)$\leftarrow$5.

The loop in step 5 is the most significant part of the algorithm. Its correctness is not apparent and we will discuss it in Lemma 6.2.5. In iteration i of the loop we work on vertices y in the range $u+2^{i-1} \leq y < u+2^i$. We identify the vertices in the cutset in two fashions. One is direct, that is, we determine the vertices that are not in the cutset, (S(y,u)='out'), by looking at their immediate predecessors, step 5.2. Considering g_1 in our example, (Fig. 6.2.4): In the 1st iteration of step 5, S(2,1)$\leftarrow$'out' because S(1,1)='in_cutset', using edge (1,2), $1 \leq 1 < 1+2^0$ and $1+2^0 \leq 2 < 1+2^1$ the conditions for statement 5.2.1 hold (Fig. 6.2.4-a). The second way in which we determine membership in the cutset is by merging values of vertices calculated in other subtrees g_w where $w>u$. This allows us to break the sequential nature of "look at the predecessor" idea. In the 2nd iteration of step 5, (Fig. 6.2.4-b) statement 5.2 doesn't cause any changes. Since 3 is the smallest vertex that is undetermined, statement 5.3 sets w to 3. In statement 5.4.3, S(3,1) is set to 'in_cutset', since $w \leq 3 < 1+2^i$, and S(3,3)= 'in-cutset'. The same statement, (5.4.3) sets the value of S(4,1) to 'out' from S(4,3); In the 3rd iteration, (Fig. 6.2.4-c), Statement 5.2.1 sets S(5,1) to 'out'. In this iteration the value of w is 6, (statement 5.3), and hence the value of S(6,1) and S(7,1) is obtained from S(6,6) and S(6,7) respectively, (statement 5.4.3).

As for the complexity of procedure Block_Cutset (Algorithm 6.2.2): The relabeling can be performed in O(log$N$) using O(N/log$N$) PE's [KR]. If a PE is assigned to each edge in step 2, there will be no read conflict. Computing the minimum can be accomplished in O(log$N$) using O(N^2/log$N$) PE's. In Step 3 we need n copies of the head dependency graphs, in order to avoid read conflict. The replication and the construction of the g_u graphs can be performed in O(log$N$) time with O(N^2/log$N$) PE's. Step 4 can be performed in O(1) using O(N^2) PE's or O(log$N$) time using O(N^2/log$N$) PE's. Finally, Step 5 requires O(log N) iterations. In each iteration we zero out vector cp(*,u) ,(statement 5.1). Variable cp is used in order to ascertain the number of copies of g_u that are needed to avoid read conflicts in statement 5.4.3. Clearly each of the statements, (5.1-5.5), that make up the loop can be performed in O(log$N$) time with O(N^2/log$N$) PE's. Observe that there are at most N copies to be made in statement 5.4.2. By Lemma 6.2.2 there is at most one immediate predecessor that could be in-cutset, there are no read conflicts in statement 5.2. Therefore step 5 can be performed in O(log^2N) time using O(N^2/log$N$) PE's. In conclusion, procedure Block_Cutset can be performed in O(log^2N) time with O(N^2/log$N$) PE's on EREW P-RAM model of computation.

Define D_u to be a subgraph of the head-dependency graph D that contains only vertices with label $\geq u$ and their incident edges. Using D_u, rather then g_u in procedure

Block_Cutset, variables min_adjacent and max-in are no longer required. We do need to use though, O(N^3/log$N$) PE's, since each D_u may have N^2 edges. It is easy to observe that D_u and g_u are equivalent for the purpose of finding the minimal cutset of a brunch. We discuss the correctness of procedure Block_Cutset in the following paragraphs. To facilitate the discussion we use D_u and g_u interchangeably.

Lemma 6.2.5: After the i^{th} iteration of the loop in step 5 of procedure Block_Cutset, S(y,u) for $u \le y < u+2^i$ is set to 'in-cutset' iff y is put in S_1 by procedure Serial_Branch_MinCut, (Algorithm 6.2.1), with input D_u.

Proof: We prove by induction on the number of iterations, i. In first iteration there is only one edge (x,y) in g_u that satisfies $u \le x < u+2^0$ and $u+2^0 \le y < u+2^1$ that is (u,u+1). During the initialization, S(u,u) is set to 'in-cutset'. S(u,u+1) is set to 'out' by step 5.2.1. Evidently u has no predecessor in D_u and u+1 has one immediate predecessor in D_u, therefore Serial_Branch_MinCut puts u in S_1 and u+1 in S_2 when applied to D_u. Assume that the Lemma is true for all iterations i≤n and examine iteration, i=n+1. By our assumption, all S(x,u) where $u \le x < u+2^n$ are set to 'in-cutset' or 'out' iff u is put in S_1 or S_2 correspondingly, by procedure Serial_Branch_MinCut with D_u as input. There are three cases to examine for S(y,u) where $u+2^n \le y < u+2^{n+1}$:

Case 1: S(y,u) is set to 'out' by statement 5.2.1. This takes place if there is an edge (x,y) were $u \le x < u+2^{i-1}$ and S(x,u) was already set to 'in-cutset' in some earlier iteration. By our assumption, this means that x was put in S_1 by procedure Serial_Branch_MinCut. Both in g_u and in D_u, vertex y has an immediate predecessor x in S_1. Therefore procedure Serial_Branch_MinCut places y in S_2.

Case 2: S(y,u) is set to 'out' by statement 5.2.2. This takes place if there is an edge (x,y) were $u \leq x < u+2^{i-1}$ and S(x,u) was already set to 'out' in some earlier iteration. Also $u < x <$max-in(u). By the construction of g_u the vertex with label max-in(u) is an immediate predecessor of y in D_u. But by statement 5.5 and step 4, S(max-in(u),u) = 'in-cutset'. Hence max-in(u)$\in S_1$ by our assumption. Since y has an immediate predecessor in S_1, y must be placed in S_2 by procedure Serial_Branch_MinCut when applied to D_u.

Case 3: S(y,u) is set to 'out' or 'in-cutset' during the merge stage, by statement 5.4.3. This takes place if there is a vertex w that is the minimal vertex in the range $u+2^n \leq w < u+2^{n+1}$ for which S(w,u) is 'undetermined'. We claim that w is placed in S_1 by procedure Serial_Branch_MinCut and S(w,u) is set to 'in-cutset' correctly. To see that, let us consider the immediate predecessors of w. Clearly if w has no incoming edge it is in S_1. If it has one or more immediate predecessors in D_u in the range $[u,u+2^n)$ then they are considered in statement 5.2 of step 5. Since S(w,u)='undetermined' it is not the case that any of the immediate predecessors is predecessors in $[u+2^n$,w-1] then those are determined by statement 5.2 and are all set to 'out' correctly. Hence by all immediate predecessors of w are determined as 'out' and by our induction hypothesis and the discussion in the first 2 cases all of them are in S_2. But then procedure Serial_Branch_MinCut places w in S_1 as claimed. In procedure Block_Cutset the value of S(w,u) is obtained from S(w,w). S(w,w) is set to 'in-cutset' in step 4 during the initialization. Therefore S(w,u) is set correctly to 'in-cutset'. From the construction of g_w and our induction hypothesis it is clear that the rest of the values copied from g_w to g_u, (statement 5.4.3) are correct.

If vertex y placed in S_2 by procedure Serial_Branch_MinCut, it must have an immediate predecessor x in S_1. Vertex x is either in $[u,u+2^n)$ or in $[u+2^{n,y-1}]$. If it is in $[u,u+2^n)$ then by our assumption S(x,u) is determined correctly to 'in-cutset' and therefore statement 5.2.1 sets S(y,u) correctly to 'out'. If $x \in [u+2^{n,y-1}]$ then y is processed by statement 5.4.3. From the discussion of case 3 above, the value of S(y,u) is set to 'out' correctly. If vertex y placed in S_1 by procedure Serial_Branch_MinCut, it must have all its immediate predecessors in S_2 or no predecessor at all. We show in case 3 above that in both cases S(y,u) is set to 'in-cutset', correctly. □

Corollary 6.2.2: After log$N$ iterations, S(y, 1)=in-cutset iff y is in S_1 by procedure Serial_Branch_MinCut, (Algorithm 6.2.1) on head-dependency graph D.

Proof: It is enough to observe that D is D_1. The rest follows from Lemma 6.2.5.□

Theorem 6.2.1: Procedure P_BRANCH_CUTSET finds a minimum cutset for a branch.

Proof: Follows from Corollaries 6.2.1 and 6.2.2.□

We can conclude that Procedure P_BRANCH__CUTSET (Algorithm 6.2.2) can be performed in O($\log^2 N$) time using O($N^2/\log N$) PE's or in O($\log N$) time using O($N^3/\log N$) PE's on the EREW P-RAM model of computation. This can be easily determined from the complexity analysis of the procedures.

6.3. Minimum Cutset of a Reducible Graph

Let T be a directed tree with root r. For every leaf v of T, there is a vertex w that is a predecessor of v such that every vertex between w and v has exactly one incoming edge and one outgoing edge, i. e., it is a chain from w to v in T. We call a chain in T that ends with a leaf an active chain. If we prune all the active chains from the tree *T*, new leaves and new active chains will be created. This pruning operation can continue until the root r becomes part of an active chain. We have shown that there could be at most O(log*N*) pruning iterations for a rooted tree with *N* vertices in Chapter 2. If *T* is a spanning tree of a reducible graph, then the subgraph related to the active chain is a branch. Our algorithm finds a minimum cutset of a reducible graph by successively identifying branches, and finding the cutset of those branches. The procedure P_Cutset, Algorithm 6.3.1, is the skeleton of the algorithm.

Before we present the details of the algorithm we need to establish the correctness of our approach. We know that a reducible graph G has unique backedge set on any Depth First Search (DFS) tree of *G* [Sh]. We first establish that the same is true for a Breadth First Search (BFS) tree of *G*.

```
procedure P_Cutset:
{Input is a reducible graph G.}
1.  Construct a breadth first spanning tree of reducible graph G;
2.  repeat
3.  Identify all the branches;
4.  Prune the branches;
5.  Determine the cutset of each branch;
6.  until the root is determined.
```

Algorithm 6.3.1 Parallel algorithm for minimum cutset of reducible graphs

Lemma 6.3.1: In a reducible graph G, an edge (u,v) is a backedge of a DFS spanning tree of G iff it is a backedge of any BFS spanning tree of G.

Proof: Consider a DFS spanning tree of G and an arbitrary BFS spanning tree of graph G. Assume to the contrary that there is an edge (u,v) that is a backedge of the DFS spanning tree of G but is not a backedge in a BFS spanning tree of G. We examine two cases:

Case 1: (u,v) is a tree edge in the BFS spanning tree. Accordingly, u is a predecessor of v in the tree. That is, there is a path from root to u without passing v and hence v does not dominate u. Since G is a reducible graph and (u,v) is a back edge of the DFS spanning tree of G it implies that v has to dominate u. We have a contradiction.

Case 2: (u,v) is a crossedge in the BFS spanning tree. Accordingly, there is a path from the root to u that does not pass through v. As in case 1, this is inconsistent with the properties of reducible graphs.

In a similar way we can show that any backedge of a BFS spanning tree is also a backedge of a DFS tree.□

Corollary 6.3.1: Any BFS spanning tree of a reducible graph partitions the edge set into backedges and DAG edges and the DAG edge set can be partitioned into tree edges and cross edges.

Proof: It follows from Lemma 6.3.1 and properties of reducible graphs.□

Corollary 6.3.2: An algorithm that is based on the decision principle can find a minimum cutset of a reducible graph using the DAG defined by a BFS tree of the graph.

Proof: It follows from Lemma 6.3.1 and Corollary 6.3.1.□

Consider the first iteration of the repeat loop of procedure P_Cutset, (Algorithm 6.3.1). Initially all vertices of the reducible graph G are undetermined. Using the active chains of the BFS tree T of G, we construct active branches of G. Vertex v is in an active branch of G iff v is in an active chain of T and edge (u,v) is in an active branch iff both u and v are vertices in the branch.

Lemma 6.3.2: In the first iteration of the repeat loop, vertex u is determined to be in the cutset of an active branch by our algorithm iff u is determined to be in cutset of the reducible graph G by the decision principle.

Proof: Let the vertices of branch B be N_B and let the cutset of this branch as determined by Procedure P_BRANCH__CUTSET, (Algorithm 6.2.2), be $C_B \subseteq N_B$. Let the cutset of G produced by procedure Serial_Min_Cutset, (Algorithm 6.1.1), be C_G and let $C'_B = N_B \cap C_G$. We show that $v \in C_B$ iff $v \in C'_B$, in other words $C_B \equiv C'_B$. Since it is the first iteration of procedure P_Cutset, (Algorithm 6.3.1), every vertex involved is either a leaf or has exactly one child in T. If vertex u is a head, its tails must be its descendants in the same branch, otherwise G is not reducible.

Assume to the contrary that there exist a vertex u such that $u \in C_B$ but $u \notin C'_B$. In that case there must be some vertex x between u and its tail v that is determined not to be in C_B, but is in C'_B. Thus x is a head which has a tail that is not in N_B, which implies x has more than one child in T. But this is impossible since every vertex that is considered in the first iteration has exactly one child. Therefore, if $u \in C_B$ it implies that $u \in C'_B$.

To complete the proof we need to show that if $u \in C'_B$ it implies that $u \in C_B$. Assume to the contrary that there exists a vertex u such that $u \in C'_B$ but $u \notin C_B$. In that case there must be a vertex $x \in C_B$ between vertex u and its tail v on the path in B. Also there is a path from u to v that contains vertices that are not in N_B and all the vertices on that path are not in C'_B. It implies that vertex u does not dominate vertex v and G is not reducible, contradiction.□

Corollary 6.3.3: The vertices that are determined to be in the cutset by procedure P_BRANCH_CUTSET in the first iteration are in the minimum cutset of the reducible graph determined by the decision principle.

Proof: Follows from Lemma 6.3.2 and Theorem 6.2.1.□

Some of the successors in G of the active branches in the i^{th} iteration are already processed. Several of those successors are determined to be in the minimum cutset and others are determined not to be in the cutset. The assumptions that allowed us to process the active branches in the first iteration independently do not hold. We now present a

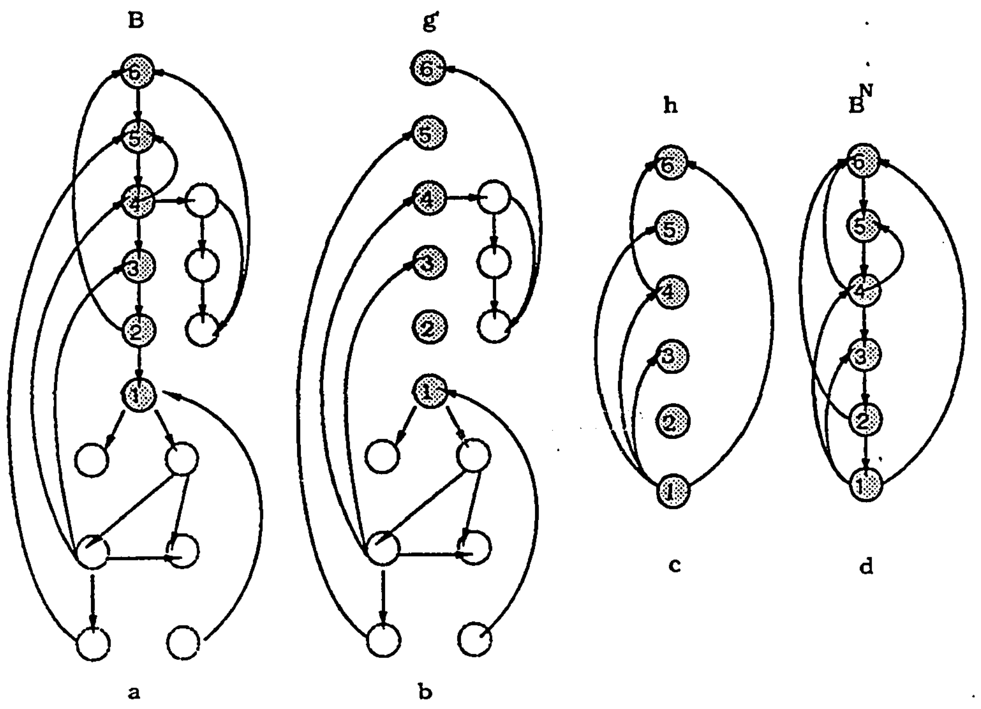


Figure 6.3.1. Example of branch normalizing

transformation that allows us to treat active branches at any iteration independently.

Procedure BRANCH_NORMALIZING (Algorithm 6.3.2) processes an active branch B with some predetermined successors and produces a normalized branch with no successors. Once we obtained the normalized branch we can apply our branch cutset algorithm. In the first step of the procedure we construct a graph g' that includes the vertices of branch B and all vertices that are already determined not to be in the cutset. In Fig. 6.3.1-a, we show an example branch B (dark vertices), its successor vertices that are determined not in the cutset (white vertices) and associated edges. Fig. 6.3.1-b shows graph g' that is obtained from the graph in Fig. 6.3.1-a by deleting all branch edges. In step 2 of the algorithm we compute the transitive closure of g' to find the exact relationships between branch vertices and successor vertices.

```
procedure BRANCH_NORMALIZING;
{Input $B$, a branch that some of its successors have been determined. These vertices are
successors of the current active branch. Edges between two vertices of the branch are ei-
ther tree edges or back edges.}
  1) Construct a graph $g' \leftarrow (N_{g'}, E_{g'})$ such that
        $N_{g'} \leftarrow \{v \mid v$ is in $B$ or $v$ is a vertex that is determined not to be in the cutset$\}$;
        $E_{g'} \leftarrow \{(u,v) \mid u,v$ are in $g'$ but are not both in $B\}$;
  2) Compute the transitive closure on $g'$;
  3) {Construct normalized branch.}
   3.1 Construct a graph $h \leftarrow (N_h, E_h)$, initially
          $N_h \leftarrow \{y \mid y$ is undetermined$\}$;
          $E_h \leftarrow \emptyset$;
   3.2 for each pair of vertices $x,y$ in $B$ in parallel do
          if $x<y$ and $x$ can reach $y$ in $g'$ then
             $E_h \leftarrow E_h \cup \{(x,y)\}$;
   3.3 $B^N \leftarrow h \cup \{$edges of $B\}$;
end.{BRANCH_NORMALIZING}
```

Algorithm 6.3.2 Algorithm for branch normalizing

Based on the computation of step 2 we construct in step 3 our normalized branch. In 3.1 we initialize the graph to contain the vertices of B, (undetermined vertices). The set of edges is initially empty. We next consider the relation between all pairs of vertices in the transitive closure of g' in step 3.2. Edge (x,y) is added to our normalized branch if $x<y$ and there is a path from x to y in g'. The result of this computation on our example is presented in Fig. 6.3.1-c. Edges (1,3), (1,4), (1,5), (1,6) and (4,6) are added in step 3.2. Finally at step 3.3 we add back the original edges of B (Fig. 6.3.1-d), to obtain B^N.

Lemma 6.3.3: In the normalized branch B^N, head u can reach its tail v iff there is a path from u to v in the reducible graph G and every vertex on the path (except u and v) is not included in the cutset by the serial algorithm.

Proof: By the construction of B^N in procedure BRANCH_NORMALIZING (Algorithm 6.3.2).□

The complexity of procedure BRANCH_NORMALIZING is dominated by the transitive closure step. This can be done in O($\log^2 N$) time using $N^3/\log N$ PE's on the EREW P-RAM model of computation [DNS] [H]. The input is represented as an NxN matrix where N is the number of vertices involved in normalizing process. It is easy to see that all the other steps can be computed in O($\log N$) time with O($N^2/\log N$) PE's. Thus BRANCH_NORMALIZING can be wrapped up in O($\log^2 N$) time with O($N^3/\log N$) PE's.

After obtaining B^N we can use P_BRANCH_CUTSET to find a minimum cutset of the branch B^N. Notice that procedure BRANCH_NORMALIZING can produce a branch where some cycles contain more than one backedge (such as 1-4-6 in g' of Fig. 6.3.1-d).

```
Procedure MIN_CUTSET_REDUCIBLE_G;
        {Input is a reducible graph $G$}
1.      Construct a BFS tree $T$ of $G$;
2.      Label the vertices of $T$ in postorder;
3.      for each vertex $u \in T$ do
                DECISION#($u$)←outdegree($u$);
                every vertex is undetermined initially;
4.      repeat
                4.1 Identify all active branches;
                        {find consecutive vertices with DECISION#=1 starting bottom up
                        from each vertex with DECISION#=0}
                4.2 for every active branch $B$ in parallel do
                        4.2.1 BRANCH_NORMALIZING;
                        4.2.2 P_BRANCH_CUTSET;
                4.3 if a vertex $u$ is newly determined and $v$ is its parent in $T$ then
                        DECISION#($v$)←DECISION#($v$)-1;
                4.4 if a vertex $y$ is determined in-cutset then delete $y$ from $G$;
        until root is determined;
5.      Minimum cutset←{$y$|$y$ is determined 'in-cutset'};
end. {MIN_CUTSET_REDUCIBLE_G}
```

Algorithm 6.3.3 Parallel algorithm for minimum cutset of reducible graphs

It suggests that the loop containing (1,6) in B^N is not a simple cycle. Procedure Branch_Normalizing must add (1,4) if (1,6) is added. (1,6) will be deleted as a redundant backedge in P_BRANCH_CUTSET.

We are now ready to present the parallel algorithm for finding a minimum cutset of a reducible graph G (Algorithm 6.3.3):

In Algorithm 6.3.3, variable DECISION# in MIN_CUTSET_REDUCIBLE_G is used to identify the active branches. It is part of the data structure of the tree and is updated after each iteration for the remaining undetermined vertices. If u has DECISION#(u)=0, u will be a leaf in next iteration, if u has DECISION#(u)=1 then u is (or will be) in some active branch.

Lemma 6.3.4: Procedure MIN_CUTSET_REDUCIBLE_G finds a minimum cutset of reducible graph G.

Proof: By previous lemmas and the decision principle.□

It is easy to see that algorithm MIN_CUTSET_REDUCIBLE_G can be done in O($\log^3 N$) time by using O($N^3/\log N$) PE's on EREW P-RAM model of computation. Step 1 can be performed in O($\log^2 N$) using O($N^3/\log N$) PE's,[DNS] [GM]. Step 2 can be done in O($\log N$) with O(N) PE's using the Euler tour technique[TV]. The computations of steps 3 4.1 and 4.3 can be performed in O($\log N$) by O($N^2/\log N$) PE's using standard techniques. As we discussed earlier Step 4.2.1 and 4.2.2 can be done n O($\log^2 N$) with O($N^2/\log N$) PE's. Finally Step 4.4 can be done in O(1) with O(N^2) PE's. The number of iterations is bounded by O($\log N$).

Theorem 6.3.1: Algorithm MIN_CUTSET_REDUCIBLE_G finds a minimum cutset of a reducible graph in O($\log^3 N$) time by using O($N^3/\log N$) processors on the EREW P-RAM model of computation.

Proof: Follows from Lemma 6.3.4 and complexity analysis.□

6.4 Heuristic for Minimal Cutset of General Graphs

For general graphs the minimum cutset problem (i.e., the feedback vertex set problem) is NP-complete [K]. Barry K. Rosen has presented a linear heuristic method to find a cutset for general graphs [R] based on Shamir's algorithm. We present here a parallel heuristic. The closer is the input graph to the reducible graph, the closer is the cutset to the optimal solution. The heuristic has the same complexity as the parallel algorithm for finding minimum cutset of reducible graphs.

The parallel heuristic for finding a minimal cutset of digraphs is based on procedure MIN_CUTSET_REDUCIBLE_G (Algorithm 6.3.3). For a general graph the cycles may not be handled simply by procedure P_BRANCH_CUTSET as it is called in each iteration of Algorithm 6.3.3. In some iteration, there may exist a cycle of G where every vertex on the cycle is determined not to be in the cutset. To solve this problem, we add a procedure H_CHECKING (Algorithm 6.4.1) after the P_BRANCH_CUTSET of step 4.2 in each iteration of the loop in Algorithm 6.3.3.

Lemma 6.4.1: Every cycle of G is cut by the cutset that is found by the heuristic.

Proof: Assume to the contrary that there is a cycle C that is not cut by the cutset found by the heuristic. In other words, there is a cycle in G where every vertex on the cycle is determined not to be in the cutset. Without loss of generality, we assume that final vertices in the cycle are determined in the i^{th} iteration. In the i^{th} iteration there must be some vertices that belong to cycle C that are B_vertices, (members of current set of active branches). After applying procedure BRANCH_NORMALIZING and procedure P_BRANCH_CUTSET to the active branches, all the cycles without any cross edges B_vertices are cut. If C is not cut there must be some cross edges C. We consider all the possibilities for cycle C containing backedges and tree edges between B_vertices. There are four cases to review:

```
procedure H_CHECKING;
{Input is a set of $B$'s, a set of $B^N$'s that are defined in BRANCH_NORMALIZING (see
Fig. 4.1-a and 4.1-d). Let vertices in $B^N$ be B_vertices (vertices in current branches) and
other vertices be S_vertices (vertices are successors of the B_vertices). Let $S_c$ denote the
set of vertices that are determined in the cutset.}
1.      {Cut cycles that don't contain backedges between B_vertices}
        1.1 for each $B$ construct a graph $B'=B-\{S_c$ and associated edges$\}$;
        1.2 Construct a graph $H=(N_H,E_H)$
                $N_H \leftarrow \{v \mid v$ is in some $B'\}$;
                $E_H \leftarrow \{(u,v) \mid u,v \in N_H, (u,v) \in E$ of $G$ but is
                        not a backedge between B_vertices.$\}$
        1.3 Compute transitive closure on $H$;
        1.4 Let $B'^N=B^N-\{S_c$ and associated edges$\}$;
        1.5 In each $B'^N$ do
                1.5.1 if $y$ can reach $x$ without passing any backedges then
                        $y$ and $x$ are in same subbranch $b^N$;
                1.5.2 for each subbranch $b^N=(N_b,E_b)$ do
                        $E_b \leftarrow \{(x,y) \mid (x,y) \in B'^N$ and $x,y \in N_b\}$;
                        if $x<y$ and $y$ can reach $x$ in $H$ then
                                $E_b \leftarrow E_b \cup \{(x,y)\}$;
        1.6 for each block do P_BRANCH_CUTSET;
2.      {Cut cycles contain backedges and cross edges between B_vertices.}
        2.1 $H \leftarrow H-\{S_c$ and associated edges$\}$;
        2.2 Compute transitive closure on $H$;
        2.3 for each $B'^N$ do
                if $(x,y)$ is a backedge and $y$ can reach $x$ in $H$ then
                        $S_c \leftarrow S_c \cup \{y\}$;
3.      {Cut cycles don't contain backedges and cross edges between B_vertices.}
        3.1 $H \leftarrow H-\{S_c$ and associated edges$\}$;
        3.2 Compute transitive closure on $H$;
        3.3 if $x$ can reach itself in $H$ then
                $S_c \leftarrow S_c \cup \{x\}$;
end; {H_CHECKING}
```

Algorithm 6.4.1Heuristic for minimal cutset

Case 1: Cycle C does not contain any backedges but contains tree edges between B_vertices. It is the case as shown in Fig. 6.4.1-a. By step 1.5, the cycle will be represented independently in the subbranches. Thus step 1.6 of procedure

P_BRANCH_CUTSET will cut *C*. This contradicts our assumption that *C* was not cut.

Case 2: Cycle *C* contains backedges and tree edges between B_vertices. It is the case illustrated in Fig. 6.4.1-b. In step 2.3 every backedge on *C* sets one vertex in the cutset. Hence *C* should be cut at least once. Again, a contradiction.

Case 3: Cycle *C* does not contain tree edge but contains backedges between B_vertices, it is similar to case 2, which handled by step 2.3, contradiction.

Case 4: Cycle *C* does not contain any backedges nor tree edges between B_vertices. It is the case shown in Fig. 6.4.1-c. In step 3. every vertex on *C* is determined to be in the cutset, contradiction.□

When procedure MIN_CUTSET_REDUCIBLE_G is applied to a graph, every vertex belongs to an active branch certain iteration. Hence every cycle will be cut in the

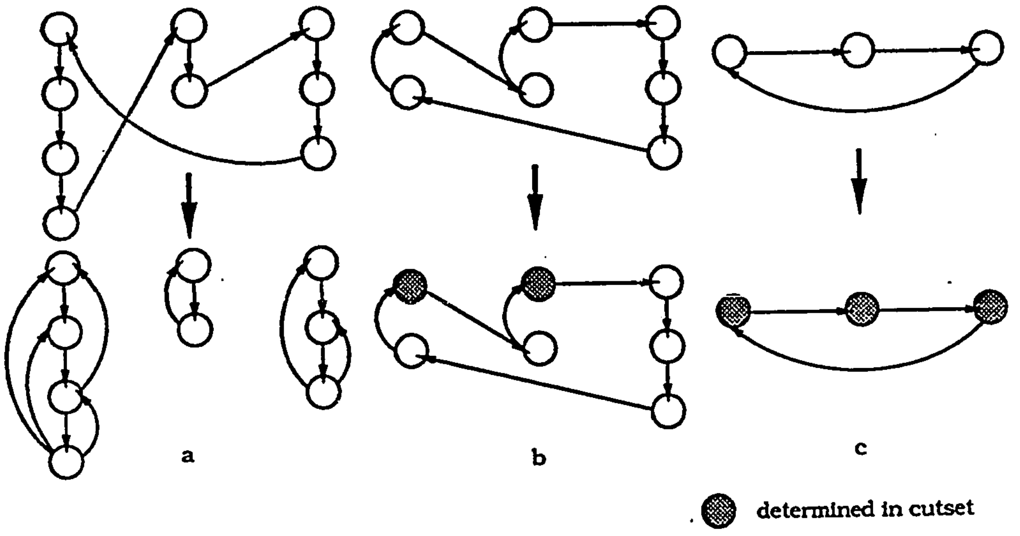


Figure 6.4.1 Cases of remaining cycles

iteration in which the final undetermined vertices of the cycle are all members of active branches. By Lemma 6.4.1, the method finds a cutset of general graph.

Lemma 6.4.2: If the input graph G is reducible, procedure H_CHECKING will not classify any vertices as part of the cutset.

Proof: Assume to the contrary that the input graph is reducible and vertex u is determined to be in the cutset by procedure H_CHECKING. We have three cases to consider:

Case 1: u is determined in the cutset by step 1.7 of H_CHECKING. It implies there is a cycle in G which is similar to the cycle shown in Fig. 6.4.1-a. Deleting any one of the edges on the cycle will produce a different DAG of G. But a reducible graph has a unique DAG, contradiction.

Case 2: u is determined in the cutset by step 2.3 of H_CHECKING. It implies there is a cycle in G which is similar to the cycle shown in Fig. 6.4.1-b. There must be a backedge (x,y), such that the root can reach y without passing x which violates the properties of reducible graphs.

Case 3: u is determined in the cutset by step 3.3 of H_CHECKING. It implies that there is a cycle in G which is similar to the cycle shown in Fig. 6.4.1-c. Deleting any one of the edges on the cycle will produce a different DAG of G. But this again contradict with the fact that G is a reducible graph.□

The proof of Lemma 6.4.2 implies that the procedure H_CHECKING will determine very few vertices in the cutset if the graph is very close to be a reducible graph. Lemma 6.4.1 and Lemma 6.4.2 imply that the parallel heuristic has a good performance on general graphs. This algorithm can recognize whether a graph is reducible or not by checking if any vertices are determined in the cutset by procedure H_CHECKING, though it is not as efficient as the method of J. Lucas and M. Sackrowitz [LS]. The complexity of procedure H_CHECKING is dominated by the complexity of computing the transitive closure. Thus the procedure can be performed in O($\log^2 N$) time using O($N^3/\log N$) PE's and the parallel heuristic can be executed in O($\log^3 N$) time using O($N^3/\log N$) PE's on EREW P-RAM model of computation.

CHAPTER 7

CONCLUSIONS

The basic goal of this study was to find efficient parallel methods in solving graph problems that are related to distributed systems and software engineering.

The primary model of parallel computation, that we have been working with, was EREW P-RAM (exclusive read exclusive write parallel random access machine). It is a multiprocessor, shared memory system that forbids several processors to access the same memory simultaneously. This model is considered closer to practical multiprocessor systems than other models.

This research focused on finding efficient parallel methods for graph problems by partitioning the problems into subproblems so as to distribute them to processors available.

We have developed a parallel pruning decomposition technique (PDS), which partitions the graph into branches and identifies the edges of the graph into different classes. For EREW P-RAM model of computations, the PDS has small overhead and simple data structures. In the modified version of PDS, the bottleneck of complexity is the spanning tree construction. For those graphs in which the spanning tree can be found efficiently on EREW P-RAM, the PDS has even more advantages.

Using the PDS, we have achieved efficient parallel methods for some important graph problems. They are, finding biconnected components (including articulation points and bridges of graphs), ear decomposition and st-numbering computing. Particularly in developing the ear decomposition, we establish a no forward edge tree (NF-tree) to obtain the biconnected certificate of a biconnected graph, which has the potential of substituting breadth first search trees in certain cases.

We apply the st-numbering to some network related problems.

- The bipartition problem is to partition a biconnected system into two connected parts with specific sizes and with a specific vertex in each side. It has applications in message routing, distributed system management and scheduling problems.
- Centroid trees and centered trees constructions in biconnected graphs belong to location problems that are often used to locate servers to meet certain criteria. The minimum total distances and minimum farthest distance are factors considered in centroid tree and centered tree construction respectively. We use st-numbering and bipartition to solve these problems efficiently. Especially for the centered tree problem, our parallel method implies a more efficient serial method. A further consideration can be to find the best centroid tree or centered tree in the sense of distances among all the centroid trees or centered trees.
- The strong orientation problem is to change a two way system into a one way system so that messages can still be routed. We apply the st-numbering here to obtain a very simple solution.

For st-numbering, further research can focus on on-line computing, for example, finding an st-numbering of {x,y} from a given st-numbering of {u,v}. This will greatly reduce the computing time for frequently updating operations on networks.

The PDS can also be applied to directed graphs. Finding a minimum cutset of reducible graphs is one example. Most computer programs can be represented by reducible graphs. To select a set of vertices that cut all the cycles in directed graphs usually leads to a simpler and more efficient analysis and verification of software. We apply the PDS to directed graphs here to achieve a parallel solution for finding a minimum cutset of reducible graphs. Derived from the method, we present a heuristic for general graphs.

REFERENCES

[A] M. Atallah, "Parallel Strong Orientation of an Undirected Graph," *Inf. Proc. Lett.* Vol.18, pp.37-39, 1984.

[ADKP] K. Abrahamson, N. Dadoun, D. A. Kirkpatrick and T. Przytycka, "A Simple Parallel Tree Contraction Algorithm," *Tech. Report 87-30,* Dept. of Computer Science, The University of British Columbia, Vancouver, B.C., Canada, 1987.

[CDH] Y. Caspi, E. Dekel and J. Hu, "On the Centroid Tree of Trees," manuscript, 1992.

[Ch] G. A. Cheston, "Generating a Spanning Tree with a Specified Node as a Centroid Point for Biconnected Graphs," *Congr. Numer.* Vol. 43, pp.211-219, 1984.

[Co] R. Cole, "Parallel Merge Sort," *Proc. 27th FOCS,* pp.511-516, 1986.

[CFHP] G. Cheston, A. Farley, S. Hedetniemi and A. Proskurowski, "Centering a spanning tree of a biconnected Graph," *Info. Proc. Lett.*, Vol. 32, pp.247-250, 1989.

[CLR] T. Cormen, C. Leiserson and R. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[CTa] D. Cheriton and R. E. Tarjan, "Finding Minimum Spanning Tree," *SIAM J. on Comp.*, Vol.5, No.4, pp.724-742, 1976.

[CTh] J. Cheriyan and R. Thurimella, "Algorithms for Parallel k-Vertex Connectivity and Sparse Certificates," *STOC,* 1991.

[CV1] R. Cole and U. Vishkin, "Optimal Parallel Algorithms for Expression Tree Evaluation and List Ranking," *AWOC,* pp.91-100, 1988.

[CV2] R. Cole and U. Vishkin, "Methodological Parallel Evaluation of Expression trees," manuscript, 1986.

[DH1] E. Dekel and J. Hu, "Parallel Pruning Decomposition," *UTD Tech. Rep. UTDCS-11-91*,1991.

[DH2] E. Dekel and J. Hu, "Efficient Parallel Graph Decomposition and Biconnected Components in Graphs on EREW," *UTD Tech. Rep. UTDCS-7-91*,1991.

[DH3] E. Dekel and J. Hu, "Parallel Algorithms for Node(s) Locating Tree Construction on Biconnected Graphs," *UTD Tech. Rep. UTDCS-12-91*, 1991.

[DH4] E. Dekel and J. Hu, "EREW Ear Decomposition Algorithm," *UTD Tech. Rep. UTDCS-13-91*, 1991.

[DH5] E. Dekel and J. Hu, "A Parallel Algorithm for Mininum Cutsets of Reducible graphs," *Proc. 29th Allerton Conf. on Commun., Contro. and Comp.*, 508-517, 1991.

[DNS] E. Dekel, D. Nassimi and S. Sahni, "Parallel matrix and graph algorithms," *SIAM J. Comput.* vol.10, No. 4, pp.657-675, 1981.

[DNP] E. Dekel, S. Ntafos and S. T. Peng, "The compression trees and their applications," *Proc. Int. Conf. Parallel Processing*, pp.132-139, 1987.

[E] S. Even, *Graph Algorithms*, Computer Science Press, 1979.

[F] R. W. Floyd, "Assigning meaning to programs," *Proc. Symp. Appl. Math.*, Vol.19 , pp.19-32, 1967.

[FRT] D. Fussell, V. Ramachandran and R. Thurimella, "Finding Triconnected Components by Local Replacements," *Proc. ICALP 89*, Springer-Verlag LNCS 372, pp.379-393, 1989.

[G] H. Gazit, "Optimal EREW Parallel Algorithms for Connectivity, Ear Decomposition and st-Numbering of Planar Graphs", *5th Inter. Paral. Proc. Symp.*, pp.84-94, 1991.

[GM] H. Gazit and G. L. Miller, "An improved parallel algorithm that computes the BFS numbering of a directed graph," *Inf. Proc. Letters* Vol.28, pp.61-65, 1988.

[GMT] H. Gazit, G. L. Miller and S. H. Teng, "Optimal Tree Contraction in EREW Model," *Proc. 1987 Priceton Workshop on Algorithm, Arcitecture and Technology Issues for Models of Concurrent Computation,* 1987.

[GR] A. Gibsons and W. Rytter, "An Optimal Parallel Algorithm for Dynamic Expression Evaluation and Its Applications," *Symp. on Foundations of Software Technology and Theoretical Comp. Sci.,* Springer Verlag, pp.453-469, 1986.

[H] D. S. Hirschberg, "Parallel Algorithm for the Transitive Closure and the Connected Components Problems," *ACM Symposium on the Theory of Computating,* pp.55-57, 1976.

[HB] K. T. Herley and Gianfranco Bilardi, "Deterministic Simulations of PRAMs on Bounded Degree Networks," *26th Allerton Conf. on Commun., Contr. and Comput.*, pp.1084-1093, 1988.

[HHT] J. Hu, D. Huynh and L. Tian, "Parallel Bisimulation Equivelances of Tree Processes," manuscript, 1992.

[HU1] M. S. Hecht and J. D. Ullman, "Flow graph reducibility," *SIAM J. Comput.* vol 1, No 2, pp.188-202, 1972.

[HU2] M. S. Hecht and J. D. Ullman, "Characterizations of reducible flow graphs," *J. Assoc. Comput. Mech.*, Vol.21, pp.367-375, 1974.

[J] H. Jung, "An optimsl Parallel Algorithm for Computing Connected Components in a Graph," *preprint*, Humboldt-University Berlin, German Democratic Republic, 1989.

[K] R. M. Karp, "Reducibility among combinatorial problems," Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, pp.146-160, 1972.

[KA] A. Kang and D. Ault, "Some Properties of a Centroid of a Free Tree," *Inf. Proc. Lett.* Vol.4, pp.18-20, 1975.

[KD] S. R. Kosaraju and A. L. Delcher, "Optimal Parallel Evaluation of Tree-structured Computations by Raking," extended abstract, The Johns Hopkins University, 1987.

[KR] R. M. Karp and V. Ramachandran, "A Survey of Parallel Algorithms for Shared-Memory Machines," *Handbook of Theoretical Computer Science,* Cambridge, MIT Press, 1990.

[L] L. Lovasz, "Computing Ears and Branchings in Parallel," *Proc. 26th FOCS,* pp.496-503, 1985.

[LS] J. M. Lucas and M. G. Sackrowitz, "Efficient parallel algorithms for path problems in directed graphs," *Proc. SPAA*, pp.369-378, 1989.

[M] E. Minieka, "The Optimal Location of a Path or Tree in a Tree Network," *Network,* Vol.15, 309-321, 1985.

[MR] G. Miller and J. Reif, "Parallel Tree Contraction and Its Application," *Proc. of 26th FOCS,* pp.478-489, 1985.

[MSV] Y. Maon, B. Schieber and U. Vishkin, "Parallel Ear Decomposition Search and st-Numbering in Graphs," *Theo. Com. Sci.,* Vol.47, pp.277-298, 1986.

[NM] D. Nath and S.N. Maheshwari, "Parallel Algorithms for the Connected Components and Minimal Spanning Tree Problems," *Inf. Proc. Lett.,* Vol.14, pp.7-11, 1982.

[P] F. P. Preparata, "New Parallel Sorting Scheme," *IEEE Transactions on Computers*, Vol.C-27, No.7, pp.669-673, 1978.

[R] B. K. Rosen, "Robust linear algorithms for cutsets," *J. Algorithms* Vol.3, pp.205-217, 1082.

[RR] V. Ramachandran and J. Reif, "An Optimal Parallel Algorithm for Graph Planarity," *FOCS*, pp.282-287, 1989.

[RV] V. Ramachandran and U. Vishkin, "Efficient Parallel Triconnectivity in Logarithmic Time," *Proc. 3rd AeGean Workshop on Computing,* Springer-Verlag LNCS 319, pp.33-42, 1988.

[Sh] A. Shamir, "A linear time algorithm for finding minimum cutset in reducible graphs," *SIAM J. Computing* Vol.8, pp.645-655, 1979.

[Sl] P. J. Slater, "Centrality of Paths and Vertices in Graph: Cores and Pits," *Proc. 4th International Graph Theory Conf.*, pp.529-542, 1980.

[STN] H. Suzuki, N Takahashi and T. Nishizeki, "A Linear Algorithm for Bipartition of Biconnected Graphs," *Inf. Proc. Lett.*, Vol.33, pp.227-231, 1990.

[TC] Y. H. Tsin and F. Y. Chin, "Efficient Parallel Algorithms for a Class of Graph Theoretic Problems," *SIAM J. Comput.* Vol.13, No.3, pp.580-599, 1984.

[TV] R. E. Tarjan and U. Vishkin, "An Efficient Parellel Biconnectivity Algorithm," *SIAM J.Comput.*, Vol.14, No.4, pp.862-874, 1985.

[V1] U. Vishkin, "Implementation of Simulaneous Memory Access in Models That Forbid It," *J. of Algorithms,* Vol.4, pp.45-50, 1983.

[V2] U. Vishkin, "On Efficient Parall Strong Orientation," *Inf. Proc. Lett.*, Vol. 20, pp.235-240, 1985.

[W] H. Whitney, "Non-separable and Planar Graphs," *Trans. Amer. Math. Soc.*, Vol.34, pp.339-362, 1932.

VITA

Jie Hu was born in Shanghai, The People's Republic of China, on May 18, 1950, the son of Mr. Ou Hu and Mrs. Hongchun Lan. After studying three years in middle school, he could not continue his formal education because of the Cultural Revolution. He then worked on a farm for seven years and in a grocery store for four years. He then worked as an interpreter in the Scientific and Technical Information Research Institute of Shanghai Petrochemical Complex in 1979. Later an exception was made, due to his lack of a university degree, and he was promoted to an assistant engineer. During these years he studied high school courses by himself and obtained more than sixty self-paced credit hours from the Shanghai Jiaotong University. He married Weiping Lu in 1982 and three years later enrolled in the Computer Science program of the University of Texas at Dallas. He received the degrees of Bachelor of Science in 1987 and Master of Science in Computer Science in 1989. For the past three years, his work has focused on parallel algorithms for distributed systems and software engineering.