1988

Some optimally adaptive parallel graph algorithms on erew pram model

Sajal K. Das
University of Central Florida

UNIVERSITY OF CENTRAL FLORIDA

OFFICE OF GRADUATE STUDIES

DEFENSE OF DISSERTATION

THE UNDERSIGNED VERIFY THAT THE FINAL ORAL DEFENSE OF THE DOCTOR OF PHILOSOPHY DISSERTATION OF Sajal K. Das

HAS BEEN SUCCESSFULLY COMPLETED ON June 27, 1988

TITLE OF DISSERTATION: "Some Optimally Adaptive Parallel Graph Algorithms on EREW PRAM Model"

MAJOR FIELD OF STUDY: Computer Science

COMMITTEE:

Narsingh Deo

Chairperson – Narsingh Deo

Ronald Dutton

Member – Ronald D. Dutton

A. Mukherjee

Member – Amar Mukherjee

Brian Petrasko

Member – Brian E. Petrasko

Ratan Kumar Guha

Member – Ratan K. Guha

APPROVED:

L. M. Trefonas 7/5/88

Louis M. Trefonas
Dean of Graduate Studies

Date

SOME OPTIMALLY ADAPTIVE PARALLEL GRAPH ALGORITHMS ON EREW PRAM MODEL

by

SAJAL K. DAS

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
the Department of Computer Science at
the University of Central Florida
Orlando, Florida

August 1988

Major Professor: Narsingh Deo

SOME OPTIMALLY ADAPTIVE PARALLEL GRAPH ALGORITHMS ON EREW PRAM MODEL

Sajal K. Das
University of Central Florida
Orlando, FL 32816
August 1988

Major Professor: Narsingh Deo

ABSTRACT

The study of graph algorithms is an important area of research in computer science, since graphs offer useful tools to model many real-world situations. The commercial availability of parallel computers have led to the development of efficient parallel graph algorithms.

Using an exclusive-read and exclusive-write (EREW) parallel random access machine (PRAM) as the computation model with a fixed number of processors, we design and analyze efficient parallel algorithms for seven undirected graph problems, such as, connected components, spanning forest, fundamental cycle set, bridges, bipartiteness, assignment problem, and approximate vertex coloring. For all but the last two problems, the input data structure is an unordered list of edges, and divide-and-conquer is the paradigm for designing algorithms. One of the algorithms to solve the assignment problem makes use of an appropriate variant of dynamic programming strategy. An elegant data structure, called the adjacency list matrix, used in a vertex-coloring algorithm avoids the sequential nature of linked adjacency lists.

Each of the proposed algorithms achieves optimal speedup, choosing an optimal granularity (thus exploiting maximum parallelism) which depends on the density or the number of vertices of the given graph. The processor-(time)^2 product has been identified as a useful parameter to measure the cost-effectiveness of a parallel algorithm. We derive a lower bound on this measure for each of our algorithms.

Dedicated to my loving parents

ACKNOWLEDGEMENTS

It has been my great pleasure to have the honor and privilege of working with Professor Narsingh Deo. I enjoyed his friendly cooperation, in academic as well as non-academic affairs, throughout my graduate studies at Washington State University and University of Central Florida. Professor Deo introduced me to the field of Parallel Algorithms. Month after month he suggested new problems (some of them are not at all related to my dissertation) and advised me through many hours of helpful discussions. Without his encouragement and guidance, this dissertation would not have been written. I also owe him a lot for improving my technical writing skill. I express my sincere gratitude and appreciation to Professor Deo.

Many persons have, directly or indirectly, contributed to the success of my dissertation. I am thankful to the members of my committee — Professors Amar Mukherjee and Ron Dutton, and Drs. Brian Petrasko and Ratan Guha. They read the dissertation carefully and offered valuable suggestions which improved the presentation. Professors Bob Brigham and Ron Dutton are my inspirations for active group research. Professors Lokenath Debnath and Amar Mukherjee and Dr. Ratan Guha, apart from their overall friendliness, provided advice and information during my stay at Orlando. My first apprentice in Computer Science started with the late Professor A. K. Choudhury of Calcutta University while Professor L. M. Patnaik at the Indian Institute of Science, Bangalore, taught me how to enjoy research. I am grateful for their contributions. I am also indebted to the teachers of Dainhat High School and Narendrapur Ramakrishna Mission Residential College for their help in building my

basic foundation in science and technology.

I deeply acknowledge the understanding, affection, and constant encouragements of my parents. To fulfill their own ambitions, they have guided me in the proper direction from childhood, and have patiently spent about three years in India without me. My elder brother and best friend, Utpal Das, deserves thanks. In order to see his brother become doctorate, he has sacrificed a great deal by taking responsibilities of our family. Sweet letters from my relatives and friends in India brought occasional mental boost. Special thanks are due to Reeta, my girl friend. Without her continuing moral support, patience, love and promising wait, I could not have completed this research work.

Grateful acknowledgements are given to the Computer Science Departments at WSU and UCF for financial assistantships. The departmental secretaries smoothed out many problems whenever they had arisen. Nancy and Delyn of the Interlibrary Loan Office at UCF offered necessary help by supplying journal materials not available here. Susanne Payne did an excellent job in editorial assistance. I would like to thank my colleague and roommate, Sushil Prasad, for his unselfish help on many an occasion. I enjoyed the company of Mahesh, Greg, Teresa, Art, Shiv, Ranga, and Uma-di. Last but not the least, my thanks to the members of the Bengali Society of Central Florida for keeping my cultural activities alive.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF SYMBOLS AND ABBREVIATIONS

G	A graph
V_G	The set of vertices of G
E_G	The set of edges of G
n	The number of vertices in graph G
m	The number of edges in G
G'	A subgraph of graph G
l	A positive integer
LIST	A two-dimensional array of unordered list of edges of G
e_i	The ith edge of a graph
i	Index
j	Index
u	A vertex
v	A vertex
u_i	The ith vertex in a path
Y	A set
$\|Y\|$	The cardinality of set Y
Z	A set
T	A spanning tree of G

E_T	The set of edges in T
F	A spanning forest of G
p	The number of processors
ε	A real number, $0 < \varepsilon \leq 1$
O	Order no greater than
θ	Order exactly
Ω	Order at least
a	A real number
$\lfloor a \rfloor$	Floor function of a
$\lceil a \rceil$	Ceiling function of a
PRAM	Parallel random access machine
EREW	Exclusive-read and exclusive-write
CREW	Concurrent-read and exclusive-write
CRCW	Concurrent-read and concurrent-write
B	A one-dimensional array
X	A one-dimensional array
X_i	The ith element in the array X
X'	A one-dimensional array
N	A nonnegative integer
d	Diameter of graph G
$\alpha(m, n)$	Inverse Ackermann's function
ROOT	A one-dimensional array

j	Index
v_i	The ith vertex
P_i	The processor with index i
V^i	The set of vertices belonging to a component numbered i
G_i	A subgraph of G allocated to processor P_i
R_i	ROOT subarray of processor P_i
f_i	The forest induced by R_i
y	A vertex or an element of a set
z	A vertex or an element of a set
w	A vertex
ru	Root of the component to which vertex u belongs
rw	Root of vertex w
b	A positive integer
β	Index
k	A positive integer
π	A problem to be solved
T_1^π (*or* T_1)	Worst-case time required by the best-sequential algorithm to solve π
T_p^π (*or* T_p)	Worst-case time to solve π by a parallel algorithm using p processors
S_p^π (*or* S_p)	Speedup of a parallel algorithm solving π using p processors
E_p^π (*or* E_p)	Efficiency of a parallel algorithm solving π using p processors
f	A function from nonnegative integers to reals

g	A function from nonnegative integers to reals
IN	The set of nonnegative integers
R^+	The set of positive real numbers
x	A nonnegative integer
x_0	A nonnegative integer
CON	Abbreviation for the connected-components problem
M^{CON}	Time required by the procedure MERGE_CONNECT
t_c	Time to count the number of distinct entries in ROOT[1 .. n]
K_i	A positive constant, $1 \leq i \leq 4$
∂	Operator for partial derivative
Q_i	A (priority) queue with label i
F_i	A spanning forest of subgraph G_i
FOR	Abbreviation for the spanning-forest problem
G^*	The transitive closure of a graph G
CT	The co-tree corresponding to a spanning tree T
e	An edge of a graph
FC_i	The fundamental cycle created by a co-tree edge e_i
FCS	Abbreviation for a fundamental cycle set
G_e	A graph G after the removal of an edge e
V_1	A subset of vertices of G
V_2	A subset of vertices of G

BRI	Abbreviation for bridge-finding problem
BIP	Abbreviation for bipartiteness-checking
MARK	A one-dimensional array
i'	Index
i''	Index
$PATH_i$	The path in a spanning tree between the end-vertices of edge e_i
L	Label of a vertex in a breadth-first search
IDENTITY	A one-dimensional array
DEPTH	A one-dimensional array
PARITY	A one-dimensional array
t	A spanning tree corresponding to a connected component
W_i	The ith worker
J_j	The jth job
c_{ij}	The nonnegative cost of assigning W_i to J_j
x_{ij}	A Boolean variable
CM	The cost matrix in the assignment problem
CC	A copy of matrix *CM*
cc_{ij}	An element of matrix *CC*
MATCH	A one-dimensional array
ROWMIN	A one-dimensional array
ZROW	A one-dimensional array

CROW	A one-dimensional array
ZCOL	A one-dimensional array
RCOL	A one-dimensional array
GLOMIN	The global minimum of the elements in array *ROWMIN*
r	Index
s	Index
jm	Index
im	Index
rm	Index
sm	Index
CM_i	A matrix, $1 \leq i \leq 3$
n'	Label of the nth node in Stage 3 in a flow network
SO	The source node in a flow network
SI	The sink node in a flow network
q	A stage in a multi-stage flow network, $1 \leq q \leq 4$
$Fq[i]$	The return of node labeled i at stage q
$F2$	A one-dimensional array
$F3$	A one-dimensional array
DSO	A one-dimensional array
DSI	A one-dimensional array
D	A matrix representing the costs of infinite-capacity arcs in a flow network
$\chi(G)$	The chromatic number of graph G

AL	A sequential approximate algorithm
PAL	A parallel approximate algorithm corresponding to AL
T_{AL}	Execution time of the algorithm AL
T_{PAL}	Execution time of the algorithm PAL
S_{PAL}	Speedup of the algorithm PAL
Δ	Maximum vertex-degree in a graph
$d(i)$	Degree of ith vertex v_i
λ	A positive integer
μ	A positive integer
DEGREE	A one-dimensional array
COLOR	A one-dimensional array
SORT	A one-dimensional array
c	A color number
γ	Index
ϕ	Index
LF	Abbreviation for largest-degree-first
DB	Abbreviation for Dutton and Brigham
PLF	Parallel LF
PDB	Parallel DB
T_d	Time required to compute the degrees of vertices in a graph
T_i	Initialization time for the PLF-algorithm

T_s	Sequential time for LF-ordering of vertices
T_c	Time for assigning colors to vertices using the algorithm PLF
C_i	A positive constant, $1 \leq i \leq 3$
C'_i	A positive constant, $1 \leq i \leq 4$
δ	Degree of a regular graph
A	Adjacency matrix of a graph
a_{ij}	An element of matrix A
$\bar{E}_G$	The set of nonadjacent vertex-pairs of graph G
$\bar{V}_{ij}$	The set of common adjacent vertices of a non-adjacent pair (v_i, v_j)
CA_{ij}	The cardinality of $\bar{V}_{ij}$
INDEX	A Boolean matrix
R_{ij}	The jth record in the priority queue Q_i
CB	The change-bit vector
VERTEXLIST	A two-dimensional array
NEIGHBOR_COLOR	A two-dimensional array

CHAPTER 1
INTRODUCTION

The rapid growth of VLSI technology and the decreasing cost of processor-hardware have made feasible highly parallel computers in which a large number of processors work simultaneously such that the total execution time to solve a single problem is reduced in comparison with the time required by a sequential computer. The computational speed achieved by such computers certainly overcomes the limitations of sequential von Neumann type computers, the speed of which cannot be increased indefinitely for physical reasons. The idea of extracting the inherent parallelism present in a problem and the commercial availability of parallel computers have motivated researchers to develop a new field of study, namely, the design and analysis of parallel algorithms.

There are two broad directions of research in parallel algorithms and computations:

1. To establish theoretical bounds on the inherent parallel complexity. Even if no restriction is imposed on the power of the parallel computation model, there exist lower bounds on the computation of problems. This is attributed to the *intrinsic parallel complexity* of problems, which limits the ultimate speedup achievable by parallelism. Examples include proving a lower bound of the

order of $\log N$ time to compute the sum of N integers on a concurrent-read and concurrent-write parallel random access machine, so long as the number of processors is bounded by any polynomial in N (Beame 1988); or proving a lower bound of the order of $\log N$ time to find the smallest of N elements on a concurrent-read and exclusive-write model, independent of the number of processors, size of the shared memory, or the instruction set of a processor (Cook, Dwork, and Reischuk 1986). Also included in this category are algorithms and theoretical results which prove that a problem belongs to *NC* (Nick's Class) or *log-space complete for P* (Cook 1985). *NC* is the class of problems which can be solved in time polynomial in the logarithm of the input size (also called *poly-logarithmic* or *poly-log* time), using polynomial number of processors. However, this approach often calls for an unrealistic number (a higher exponent in the problem-size) of processors. This is referred to as *unbounded parallelism.*

2. To design parallel algorithms implementable on realistic computers. This assumes *bounded parallelism,* where a large but fixed number of processors are available. Though the architectural development of parallel computers is quite advanced, the lack of efficient parallel algorithms and data structures poses a bottleneck to the wide applicability of parallel computers. Therefore, there are ample scopes to enrich this fertile area by designing efficient parallel algorithms which can be directly implemented on realistic computers.

The work presented in this dissertation falls in the second direction of research. Note that if a fast algorithm is designed under the assumption of unbounded parallelism, its adaptation to computers with bounded parallelism is a nontrivial problem. The Theorem of Brent (1974) and its proof gives some idea how to manage this, but the resulting algorithm is only a crude simulation on a finite number of processors and often not very efficient. Thus, if possible, a better approach is to come up with parallel algorithms, keeping a bounded number of processors in mind.

1.1 Motivation

An increasing proportion of computations are nonnumeric in nature, such as sorting, searching, graph processing, and so on. Of particular interest are graph problems, which are often abstractions of important real-world situations, such as communication and transportation networks, VLSI design, program optimization, automata theory, crypto systems, artificial intelligence, image processing, and applications in other fields of science and engineering. Therefore, there is always a demand for fast solutions to frequently-occurring graph problems. The objective of this dissertation is to develop efficient parallel algorithms for several graph problems (specific applications are cited in respective chapters) on a synchronous, general-purpose, shared-memory model of parallel computation. The problems include finding the connected components, a spanning forest, a fundamental cycle set, and the bridges, determining bipartiteness, a minimum-weight bipartite matching (also called the

assignment problem), and vertex-coloring of a given undirected graph. The last two are combinatorial optimization problems.

Except vertex-coloring, all other problems of our interest have polynomial-time complexity on sequential computers. For the graph-coloring problem, being NP-hard, all known algorithms have exponential (in the problem-size) time complexities. Hence there is a great deal of motivation to parallelize polynomial-time approximation algorithms for vertex-coloring.

A significant body of literature is available on parallel graph algorithms on shared memory machines (see Tables 3.1, 4.1, and 6.1). The majority of these algorithms is developed assuming unbounded parallelism. Many of them allow simultaneous reading from and/or simultaneous writing into the same memory cell. Since relatively little has been done in designing efficient graph algorithms on the weakest albeit most practical shared memory models (with bounded parallelism), which do not allow simultaneous access to a memory cell, we pursue this subject here.

Another implicit assumption in most of the previous work is that the input graph is dense so that the adjacency matrix can be used as a data structure with no penalty. However, except for the assignment problem, our graph algorithms are intended to manipulate large, randomly sparse graphs. The dynamic way in which these graphs are modified makes the choice of data structures an important consideration in order to exploit sparsity while designing parallel graph algorithms. Sequential data structures, such as linked lists, stacks, or queues, are not very effective in supporting

parallel operations. Thus, we use alternative data structures, wherever possible, to handle sparse as well as dense graphs with efficiency.

An important issue in the design of parallel algorithms is a careful balancing of computation and communication time complexities. Usually, the problem decomposition controls the granularity or grain-size — the amount of task performed by each processor. If the granularity is too fine, communication and synchronization overhead predominate. On the other hand, too coarse granularity may cause load unbalance and inefficient processor utilization. Both situations degrade the speedup. To make a proper compromise between computation and communication, we introduce a new performance measure for parallel algorithms. The motivation is to choose an optimal number of processors (as a function of problem-size) such that both speedup and efficiency are maximized.

1.2 Related Research

In this section, we briefly review the research related to the design and analysis of parallel algoritnms for realistic computers.

1.2.1 Existing Parallel Computers

The absence of a universal model of parallel computers has encouraged researchers to propose and design widely varying parallel architectures like systolic arrays, tree machines, vector processors, multiprocessors, dataflow-processors, and so

on. Introductions and surveys of advanced parallel architectures are given in Almasi (1985), Dongarra and Duff (1985), Hwang and Briggs (1984), Quinn (1987), and Te Riele (1987). Multiprocessors have further been classified according to instruction and data streams. We are interested in commercially available, general-purpose multiprocessors of the multiple-instruction and multiple-data (MIMD) type. Two major approaches of building such computers are

1. Shared memory computers: These computers have a global shared memory either with a shared bus or a multistage interconnection network between processors and storage for interprocessor communication. For example, Encore's Multimax/320 (1987) and Sequent's Balance/21000 (1986) are computers with shared bus, while BBN's Butterfly/GP1000 (Howe 1988) and Alliant's FX/8 (Babb II 1988) are computers with interconnection networks.
2. Fixed connection computers: These computers do not have global shared memory. Processors, each having local memory, are connected by a fixed topology, such as mesh, hypercube, pyramid, etc. The interprocessor communication takes place via message-passing. Examples of hypercube-based machines are Intel's iPSC/d7 (1986), NCUBE's NCUBE/10 (Hayes et al. 1987), Thinking Machines' Connection machine (Hillis 1985), and Ametek's S/14 (Dongarra and Duff 1985).

These two classes of machines have merits as well as demerits. For example, it is easier to program on a shared memory computer while it is cheaper to build a fixed

connection computer. However, a shared memory machine is more versatile in the sense that it can simulate the message-passing primitives of the fixed connection machine, but the converse is not true (Seitz 1985).

Several general-purpose parallel computers have been or are being designed as research machines. To name a few, Ultracomputer at New York University, Cedar at University of Illinois, Cosmic Cube at California Institute of Technology, Research Parallel Processor Prototype (RP3) at IBM, NON-VON at Columbia University, Partitionable SIMD/MIMD Multicomputer (PASM) at Purdue University, Texas Reconfigurable Array Computer (TRAC) at University of Texas. Detailed description and design philosophy of these machines are available in Lipovski and Malek (1987), where authors have also presented a theoretical basis for comparing different parallel computers.

Throughout this dissertation, our model of computation is an exclusive-read and exclusive-write (EREW) parallel random access machine (PRAM), which can be treated as an abstract generalization (with possibly additional power) of general-purpose, shared memory parallel computers. (The details of PRAMs are described in Section 2.2.) The idea behind the choice of PRAM as a model is to assure that our proposed algorithms are independent of the target machine architecture.

1.2.2 Parallel Algorithm Design Strategies

Though a relatively young discipline, parallel algorithms are under extensive

study and significant results are being established. Although some work has been reported, a general framework for the design and representation of parallel algorithms is still missing. The following literature deals with design strategies: parallel greedy (Anderson and Mayr 1984), parallel divide-and-conquer (Horowitz and Zorat 1983; Tang and Lee 1984; Nelson 1987), parallel branch-and-bound (Lai and Sahni 1984; Lai and Sprague 1985; Li and Wah 1986), parallel dynamic programming (Li and Wah 1985, Veldhorst 1986), binary tree method (Dekel and Sahni 1983), filtration and funnelled pipelining (Hochschild, Mayr, and Siegel 1983), deterministic coin tossing and accelerating cascades (Cole and Vishkin 1986), compute-aggregate-broadcast (Nelson 1987), and parallel symmetry-breaking (Goldberg, Plotkin, and Shannon 1987). To the best of our knowledge, no book or survey paper discusses systematically all of these paradigms for designing parallel algorithms. For details on stepwise parallel program design and correctness proofs, readers are encouraged to consult Chandy and Misra (1988). The state of the art in software tools for programming commercially available parallel computers is reported in Babb II (1988). Jamieson, Gannon, and Douglass (1987) and Quinn (1987) provide reference sources for several important issues on parallel algorithm design.

1.2.3 Parallel Data Structures

Designing appropriate data structures is an art in traditional algorithm design. To achieve higher speedup in parallel processing, suitable parallel data structures are also

required. A *parallel data structure* is a single coherent data structure in which each processor accesses the part allocated to it. The allocation is made either by partitioning the data structure into disjoint portions or by replicating some parts of it. The literature on parallel and concurrent data structures includes adjacency list matrix (Ecstein and Alton 1977), doubly-linked adjacency list (Wyllie 1979), parallel linked list (Kruskal, Rudolph, and Snir 1986), partial sum's tree (Shiloach and Vishkin 1982; Vishkin 1984), linked array, parallel semiqueue and deque (Quinn and Yoo 1984), parallel heap (Kwan and Ruzzo 1984; Quinn and Yoo 1984), parallel 2-3 trees (Ellis 1980a, Paul et al. 1983), parallel PQ-trees (Klein and Reif 1986), parallel binary tree traversal (Moitra and Iyenger 1987), concurrent binary search tree (Manber 1984), concurrent AVL trees (Ellis 1980b), and concurrent priority queues (Rao and Kumar 1988).

1.3 Executive Summary

The principal contribution of this dissertation is designing deterministic, optimal parallel algorithms for several undirected graph problems on a synchronous, shared memory model of computation, which forbids simultaneous read or write access to a memory cell. The problems of our interest belong to different classes so far as their applications are concerned. Also, the sequential algorithms corresponding to these problems have different time complexities. In particular, connected components, spanning forest, bridge-detection, and bipartiteness-checking problems can be solved

in linear (in the edges of the graph) time. The sequential algorithms for fundamental-cycle-set and the assignment problem have cubic (in the vertices) time complexities. We consider two approximate (sequential) vertex-coloring algorithms, which require linear and cubic times, respectively.

For all but the assignment and coloring problems, the data structure for the input graph is an unordered list of edges. This simple data structure avoids the sequential access of a linked adjacency list and also requires optimal space as opposed to the extra space used by an adjacency matrix to represent sparse graphs. The divide-and-conquer strategy is the underlying paradigm for designing parallel algorithms for these problems. In the divide-and-conquer strategy, the given problem is divided into subproblems which can be executed independently by different processors. The sub-solutions obtained are then merged step by step to reach the final solution.

We develop two parallel algorithms for the assignment problem. One of them is a parallelization of the classical Hungarian method, while the other performs by finding a min-cost flow in an appropriate layered network. The min-cost flow is computed by applying a variant of the dynamic programming technique. Since the input graph is dense for the assignment problem, we use a cost matrix as the data structure. Finally, two approximate parallel algorithms are designed for coloring the vertices of a graph. One of these algorithms uses an elegant data structure, called the adjacency list matrix, to alleviate the inherent sequential nature of linked adjacency lists. The other algorithm uses an adjacency matrix. Problem decomposition is the

basic design strategy for the parallel coloring algorithms.

The processor-(time)^2 product has been chosen as a useful parameter to measure the cost-effectiveness and to derive optimality conditions of parallel algorithms. This parameter is a proper compromise between speedup and efficiency. We compute a lower bound on processor-(time)^2 for each of our algorithms. The parallel algorithms for connected-components, spanning-forest, fundamental-cycle-set, and bipartiteness-checking achieve optimal speedup for dense as well as sparse graphs, and are optimally scalable up to a large number of processors, which depends on the density of the input graph. The algorithms for bridge-detection and the assignment problem are optimal for dense graphs only. One of the parallel coloring algorithms is efficient for regular or near-regular graphs, and the other is efficient for graphs of widely varying chromatic numbers.

1.4 Overview of Dissertation

Chapter 2 first presents the terminology and notation used throughout this dissertation. This is followed by the description and relative power of different classes of parallel random access machine models. We then introduce a new performance measure, called *processor-(time)*^{2}, and justify its usefulness in designing optimal parallel algorithms.

Chapters 3 through 6 are devoted to designing and analyzing several efficient parallel algorithms for undirected graphs on exclusive-read and exclusive-write, paral-

lel random access machines. Each chapter contains a brief discussion of the previous research on the problems being examined.

In Chapter 3, we present parallel divide-and-conquer algorithms for determining connected components and a spanning forest. These algorithms are then used as subroutines, in Chapter 4, to design algorithms for finding a fundamental cycle set, for bridges of a connected graph and for determining bipartiteness of a graph.

Chapter 5 develops two optimal parallel algorithms for the assignment problem or a minimum-weight matching in a complete bipartite graph. Vertex-coloring of a graph is considered in Chapter 6. Chapter 7 concludes the dissertation and explores possible future work.

CHAPTER 2

BACKGROUND CONCEPTS

2.1 Terminology and Notation

We begin this section by defining the graph theoretic terms and other notation used throughout this dissertation. Definitions pertinent to a specific chapter are given in that chapter. Deo (1974) and Harary (1969) provide a general introduction to graph theory.

An *undirected graph* $G = (V_G, E_G)$ consists of a finite, nonempty set V_G of *vertices* (or nodes) and a finite set E_G of *edges*. An edge (u, v) is an unordered pair of distinct vertices. Vertices u and v are *adjacent* if $(u, v) \in E_G$. We consider *simple* (i.e., without self-loops and parallel edges) graphs of n vertices and m edges. For a vertex $u \in V_G$, $adj(u) = \{v \mid (u, v) \in E_G\}$ is called the set of *neighbors* of u. The collection of such sets for all vertices form the *adjacency list* of the graph G. For $V_G = \{v_1, v_2 \ldots, v_n\}$, the matrix $A = [a_{ij}]_{n \times n}$ is called the *adjacency matrix* of G if

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E_G \\ 0 & \text{if } (v_i, v_j) \notin E_G \end{cases}$$

A *path of length* l from u to v in G is a sequence $u = u_1, u_2, \ldots, u_l = v$ of distinct vertices such that $(u_i, u_{i+1}) \in E_G$ for $1 \le i \le l - 1$. A *cycle* is a path with

$u_1 = u_l$. A graph without cycles is called *acyclic*. A graph $G' = (V_{G'}, E_{G'})$ is a *subgraph* of the graph G if $V_{G'} \subseteq V_G$ and $E_{G'} \subseteq E_G$.

The total number of elements in a set Y is $|Y|$. Notation $Y \subseteq Z$ (or $Y \subset Z$) means that Y is a subset (or proper subset) of Z. The union, intersection, and difference of two sets are represented by $\bigcup$, $\bigcap$, and $-$ respectively.

$$Y \bigcup Z = \{y \mid y \in Y \text{ or } y \in Z\}$$

$$Y \bigcap Z = \{y \mid y \in Y \text{ and } y \in Z\}$$

$$Y - Z = \{y \mid y \in Y \text{ and } y \notin Z\}$$

We use O, θ, and Ω to mean upper bound, exact bound, and lower bound, respectively. Let $f, g : IN \rightarrow R^+$ be two functions from the set, IN, of nonnegative integers to the set, R^+, of positive real numbers. Then the *order* notations are formally defined as follows (Baase 1988):

(i) $f(x) = O(g(x))$ if there exist $c \in R^+, x_0 \in IN$ such that for all $x \geq x_0$, $|f(x)| \leq c \; |g(x)|$.

(ii) $f(x) = \Omega(g(x))$ if there exist $c \in R^+, x_0 \in IN$ such that for all $x \geq x_0$, $|f(x)| \geq c \; |g(x)|$.

(iii) $f(x) = \theta\,(g(x))$ if $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$.

For any real number $a \in R^+$, $\lfloor a \rfloor$ denotes the greatest integer less than or equal to a, and $\lceil a \rceil$ is the least integer greater than or equal to a. Throughout all logarithms are to the base 2 and $\log n$ denotes $\lceil \log_2 n \rceil$. We define i^{th} iterate of the log function as

$\log^{(i)} n = \log^{(i-1)} \log n$ for $n \geq 1$, and $\log^{(0)} n = n$.

2.2 Shared Memory Parallel Computation Model

Parallel random access machines (PRAMs) are well accepted shared memory models for synchronous parallel computation, and have been widely used for parallel algorithm design. It is convenient to express parallel algorithms on PRAMs because one may concentrate on the problem of *parallelizing,* i.e., decomposing the problem at hand into simultaneously executable tasks, without having to worry about the communication between the tasks.

Formally, a PRAM (pronounced "p ram") consists of a finite number p of unit-cost, general-purpose, sequential processors or RAMs (Aho, Hopcroft, and Ullman 1974), each equipped with a small amount of local memory, operating synchronously in parallel. Each processor knows its own index or identification number P_i, $1 \leq i \leq p$; can perform any scalar arithmetic, comparison, or boolean operation in one time unit; and can read from and write into its own local memory. There is a common (global) shared, random access memory, each *cell* of which can be read from or written into by any processor. Program and input data reside in the common memory. From the view point of designing algorithms, we assume a single instruction stream, i.e., all processors execute a single program. But the identification number of a processor can control the sequence of steps to be executed, and different processors may do different things. Hence the net effect is that of a multiple instruc-

tion stream. At any instant, a processor is either masked (i.e., inactive) or executes the same instruction as all other processors but each on a different data set. The necessary synchronization and communication among the processors take place via global variables stored in the shared memory. For example, when two processors wish to communicate, one processor writes a datum in the shared memory which is subsequently read by the other processor.

Different processors can simultaneously access the common shared memory. Whenever more than one processor attempts to read from (or write into) the same memory cell at the same time, a read (or write)-conflict takes place. Depending on whether or not read- or write-conflicts are allowed, we distinguish three main classes of PRAM models (Borodin and Hopcroft 1985; Snir 1985).

1. Exclusive-read and exclusive-write (EREW) PRAM: Neither read- nor write-conflicts are allowed. This model is the same as PRAC or parallel random access computer due to Lev, Pippenger, and Valiant (1981).
2. Concurrent-read and exclusive-write (CREW) PRAM: Only read-conflicts are allowed but not write-conflicts. This model is first defined as P-RAM by Fortune and Wyllie (1978).
3. Concurrent-read and concurrent-write (CRCW) PRAM: Both read- and write-conflicts are allowed, with some rule defining the exact semantics of simultaneous writing. This model is also referred to as WRAM in the literature.

While using EREW (or CREW) PRAM model, an algorithm that would have read/write (or write)-conflict is considered an illegal algorithm. Three subclasses of the CRCW PRAM model have been suggested, which differ in the way write-conflicts are resolved. These variants are (Fich, Ragde, and Wigderson 1988):

(i) COMMON: All processors attempting to write into the same shared memory cell write a common value; otherwise the program is illegal.

(ii) ARBITRARY: If more than one processor attempts to write into the same cell, an arbitrary one succeeds.

(iii) PRIORITY (MINIMUM): Among all processors which simultaneously attempt writing into the same memory cell, the one with the highest priority (minimum index) will succeed. This subclass is essentially identical to SIMDAG (single instruction stream, multiple data stream, global memory) of Goldschlager (1978).

All of these PRAM models have been used for implementing parallel algorithms. For example, Cole and Vishkin (1986) and Kruskal et al. (1986) use EREW; Chin et al. (1982) and Hirschberg et al. (1979) use CREW; Shiloach and Vishkin (1981) and Vishkin (1984) use COMMON; Cole and Vishkin (1986) and Shiloach and Vishkin (1982) use ARBITRARY; and Awerbuch and Shiloach (1983) use MINIMUM.

2.2.1 Relative Powers

In the previous discussion, all classes and subclasses of PRAMs are listed in increasing order of their strengths. For example, the MINIMUM model is at least as powerful as the ARBITRARY model. This is because if an algorithm performs irrespective of which processor succeeds in writing, then it will perform unaltered if the lowest-indexed processor is allowed to succeed. Similar argument shows that the ARBITRARY model is no less powerful than the COMMON model. Also, the COMMON model is at least as powerful as the CREW PRAM which, in turn, is at least as powerful as the EREW PRAM. Moreover, Cook, Dwork, and Reischuk (1982) have shown that the CREW PRAM is strictly less powerful than the CRCW PRAM, by proving that the logical OR of N bits can be computed in one step on the COMMON model whereas it requires $\Omega(\log N)$ steps using a CREW PRAM model. By considering the problem of searching for a key in a list of ordered elements, Snir (1985) has demonstrated that the EREW PRAM is strictly less powerful than the CREW PRAM. Thus,

$$\text{EREW} \subset \text{CREW} \subset \text{COMMON} \subseteq \text{ARBITRARY} \subseteq \text{MINIMUM}.$$

Relative powers of different variants of CRCW PRAM have been rigorously studied by Fich, Ragde, and Wigderson (1988). The weakest albeit most practical EREW PRAM model can simulate the most powerful CRCW (MINIMUM) PRAM with a delay of $O(\log p)$ per step using $O(p)$ additional processors or with a delay of $O(\log^2 p)$ per step without additional processors (Vishkin 1983a).

Not only does the CRCW PRAM provide an elegant framework for the design and analysis of parallel algorithms, but it is also closely related to the unbounded fan-in circuit, another abstract model of computation. An unbounded fan-in Boolean circuit, is an acyclic directed graph — each node of which is labeled as either an input node, an AND-gate, an OR-gate, or a NOT-gate. Input nodes have fan-in zero while NOT-gates must have fan-in one. In addition, certain nodes are designated as output nodes. The *size* of a circuit is the number of edges, and the *depth* is the length of a longest path from some input to some output. Stockmeyer and Vishkin (1984) have shown that parallel time and number of processors of a PRAM correspond, respectively, to depth and size of a circuit. The time-depth correspondence is to within a constant factor and the processor-size correspondence is to within a polynomial. Therefore, a PRAM is a robust, abstract model of parallel computation.

2.2.2 Realizability

The entire family of PRAM models (also termed as *paracomputers,* Schwartz 1980) is idealistic because of physical fan-in limitations; present and foreseeable technology does not seem to allow more than a constant number of processors to simultaneously access the same memory module. Nevertheless, Schwartz (1980) noted that such models "can play a useful role as theoretical yardsticks for measuring the limits of parallel computation" (p. 486). The so-called most practical and the weakest EREW PRAM model can be made realizable to some extent by incorporat-

ing a broadcast facility (Section 2.3), by which a processor communicates with all others in more than a single step. This mechanism reduces the number of processors having simultaneous access to a memory module.

Another pragmatic approach toward the realization of PRAM is to have only limited number of processors with read or write accesses to each memory cell and to have each processor directly communicating with a fixed number of other processors. This bounded-degree network model is known as an *ultracomputer* (for example, a perfect-shuffle interconnection machine, Schwartz 1980). Of particular interest is the NYU Ultracomputer, a general-purpose MIMD machine accessing a global shared memory via a multistage perfect-shuffle interconnection (called the *Omega* network), which can be regarded as an approximate realization of paracomputers (Gottlieb et al. 1983). Making use of the fetch-and-add synchronization primitive along with the *serialization principle,* the NYU Ultracomputer accomplishes the effect of simultaneous access to the shared memory. For details on the implementation and choice of abstract parallel machine models, see Vishkin (1983b).

The fact that the PRAM model (though conceptually very convenient to develop algorithms) is not very practical, has motivated researchers to efficiently simulate PRAM computations on feasible parallel models, particularly models without global shared memory. It can be shown that each step of a p-processor PRAM can be simulated in $O(\log p (\log \log p)^2)$ steps on a bounded-degree network of p processors (Upfal and Wigderson 1984). Therefore, if we develop algorithms for PRAM,

they can be easily translated to algorithms for actual machines.

2.3 Algorithmic Constructs

Parallel algorithms will be represented by employing the usual fork and join statements, denoted by a *parallel for* construct with the syntax:

```
for all index, expression ≤ index ≤ expression, do
    parbegin
        statement-list
    parend;
```

For example,

```
for all i, 1 ≤ i ≤ p, do
    parbegin
        Statement 1;
        Statement 2;
            .
            .
        Statement S;
    parend;
```

indicates that the single process executing this statement is to fork into p parallel processes (corresponding to processors P_i, $1 \le i \le p$), each sharing the environment of the original process with its own unique value of the index i. The index may be referenced inside the parallel structure, but it must not be modified. All p processes simultaneously execute the "statement-list," and each processor executes statements 1 through S sequentially, and then join into a single process at the corresponding

parend. Thus, the global synchronization is achieved, and no processing occurs beyond the **parend** until all of the forked processes have completed "statement-list." When there is a single statement within the parallel structure, we sometimes omit the words **parbegin** and **parend.**

A sequential loop execution is distinguished by the construct

```
for each index, expression ≤ index ≤ expression, do
   begin
      statement-list
   end;
```

For example, when we use **for each** j, $(i-1)\left\lceil \frac{N}{p} \right\rceil + 1 \le j \le i\left\lceil \frac{N}{p} \right\rceil$, **do . . .**, where $1 \le i \le p$, it is assumed that the sequential loop is executed so long as $j \le N$. The symbol $B[i\ ..\ j]$ means that it is an array with constant-time access to each of its elements $B[i], B[i+1], \ldots, B[j]$.

We illustrate the preceding syntactic constructs with an example program. If all p processors in an EREW PRAM model simultaneously need a shared datum, a broadcast operation is performed. The algorithm BROADCAST is adopted from Akl (1986), where B is an array (in the shared memory) of length p which is initialized to zeros.

```
procedure BROADCAST;
  begin
    Processor P_1 copies the shared datum into B[1];
    for each i, 0 ≤ i ≤ log p − 1, do
        for all j, 2^i + 1 ≤ j ≤ 2^(i+1), do
            parbegin
                P_j copies B[ j − 2^i ] into B[ j];
            parend;
  end.
```

Clearly, after $O(\log p)$ time, each of the p processors receives the shared datum. We will use the procedure BROADCAST in Section 4.3.2, in the context of designing a better algorithm for a fundamental cycle set.

2.4 Performance Measures

Usually, three measures are considered by the algorithm designers to evaluate the performance of a new parallel algorithm. These are speedup, efficiency, and cost, as explained in the following. Given a problem π, let T_1^{π} and T_p^{π} be, respectively, the worst-case running times required to solve π by the best-known sequential algorithm and by a given parallel algorithm using p processors. The uniform cost criterion is assumed for the worst-case time. Over all inputs of a given problem-size, the worst-case time for the sequential algorithm is the maximum of the time required for its execution, whereas the worst-case time of a parallel algorithm is the maximum of the time elapsed from when the first processor starts execution until the last processor terminates it. The parallel time complexity, also referred to as the *depth* of a

parallel algorithm, does not include input/output time.

The *speedup* S_p^π of a parallel algorithm running on p processors is defined as the ratio of T_1^π to T_p^π. Clearly, the larger the ratio, the better is the algorithm. A trivial bound is $1 \le S_p^\pi \le p$, because the best upper bound on the parallel running time for an algorithm using p processors is $\frac{T_1^\pi}{p}$. Otherwise, by simulating the parallel algorithm on a sequential computer, we obtain a faster sequential algorithm. However, in practice, a speedup of p is often difficult to achieve due to data dependency in the problem itself and/or synchronization and communication overhead among processors. The *efficiency* (or processor-utilization) E_p^π of a parallel algorithm is the ratio of the speedup to the number p of processors used. Obviously, $\frac{1}{p} \le E_p^\pi \le 1$. The *hardware cost* of a parallel algorithm is defined as the product pT_p^π. That is, the cost represents the worst-case number of operations while executing the parallel algorithm. When it is clear from the context, for brevity, the performance parameters will be denoted as T_p, S_p, and E_p.

A parallel algorithm is *optimal* or is said to have *optimal speedup* if its speedup is proportional to p (i.e., $S_p = \theta(p)$ or efficiency is $O(1)$). In other words, the cost of an optimal parallel algorithm solving a problem matches (within a multiplicative constant) to the worst-case number of operations required by the best-known sequential algorithm solving it. Of course, if we have an optimal parallel algorithm with running time T_N using N processors, then (by the obvious processor simulation) we

also have an optimal algorithm even by employing fewer processors, i.e., it runs in time $T_p = \theta\,(\frac{N}{p}\,T_N)$ for all $p \le N$ processors. Such algorithms, also known as *optimally adaptive* or *scalable,* are useful from a practical point of view where we have a limited number of processors.

2.4.1 A New Measure $p\,(T_p)^2$

In designing many parallel algorithms, one often minimizes the parallel time T_p, employing as many processors p as possible. This may maximize the speedup S_p, but the efficiency E_p may be poor. On the other hand, if we try to minimize only the cost, we might end up sacrificing the speedup, although the efficiency may be high. For a proper trade-off, one should try to employ an optimal number of processors such that the product $S_p E_p$ is maximized. In other words, one should minimize the product $p\,(T_p)^2$ with respect to p. While designing systolic algorithms for dynamic programming problems, Li and Wah (1985) have also recognized that $p\,(T_p)^2$ is an appropriate measure of the performance in parallel processing. This *processor-(time)*^{2} complexity in parallel algorithms has the similar flavor as *area-(time)*^{2} complexity in the context of designing VLSI circuits (Thompson 1979), where the objective is to find the minimum area (which includes the total size of basic components and the total length of interconnecting wires) for fabricating a chip and to minimize the total time (including input/output, computation, and communication delay times) required to solve a problem.

Note that the parameter $p(T_p)^2$ is also an appropriate measure for comparing parallel algorithms for a single problem on a particular model. The lower its value, the better is the performance and the corresponding algorithm is said to be more *parallelizable*. (By parallelizability, we mean how well the problems can take advantage of multiple processors.) Since the product $S_p E_p$ yields the speedup-to-cost ratio, a lower value of $p(T_p)^2$ will obviously lead to a higher value of speedup-to-cost ratio.

Let us justify the utility of our new performance measure with a familiar parallel algorithm which computes the minimum among N elements using $p \leq N$ processors on an EREW PRAM model (Baase 1988). The input elements are $X[1 \,.\,.\, N]$, stored in the shared memory. We use an auxiliary array X' of size p. Initially, processor P_i, $1 \leq i \leq p$, operates on $\left\lceil \frac{N}{p} \right\rceil$ elements given by $X[(i-1)\left\lceil \frac{N}{p} \right\rceil + 1, \ldots, i\left\lceil \frac{N}{p} \right\rceil]$. It finds (sequentially) the minimum of these elements in $\left\lceil \frac{N}{p} \right\rceil - 1$ steps, and stores in $X'[i]$. Then parallel merging takes place.

The execution of the parallel algorithm can be depicted in the form of a binary tree (Figure 2.1) as follows. After the initial computation, p local minima, $X'[1 \,.\,.\, p]$, are produced which form the leaves of the binary tree. Next, the processors are assigned such that all the computation at a level can be done as one step. When a processor P_i compares elements $X'[i]$ and $X'[j]$, where $i < j$, the resulting minimum is stored in $X'[i]$. In this approach, half of the processors used in a

step is reassigned in the following step. The global minimum is found after $\log p$ merging steps, when the processor P_1 at the root of the binary tree completes its computation. The parallel algorithm is formally described as follows.

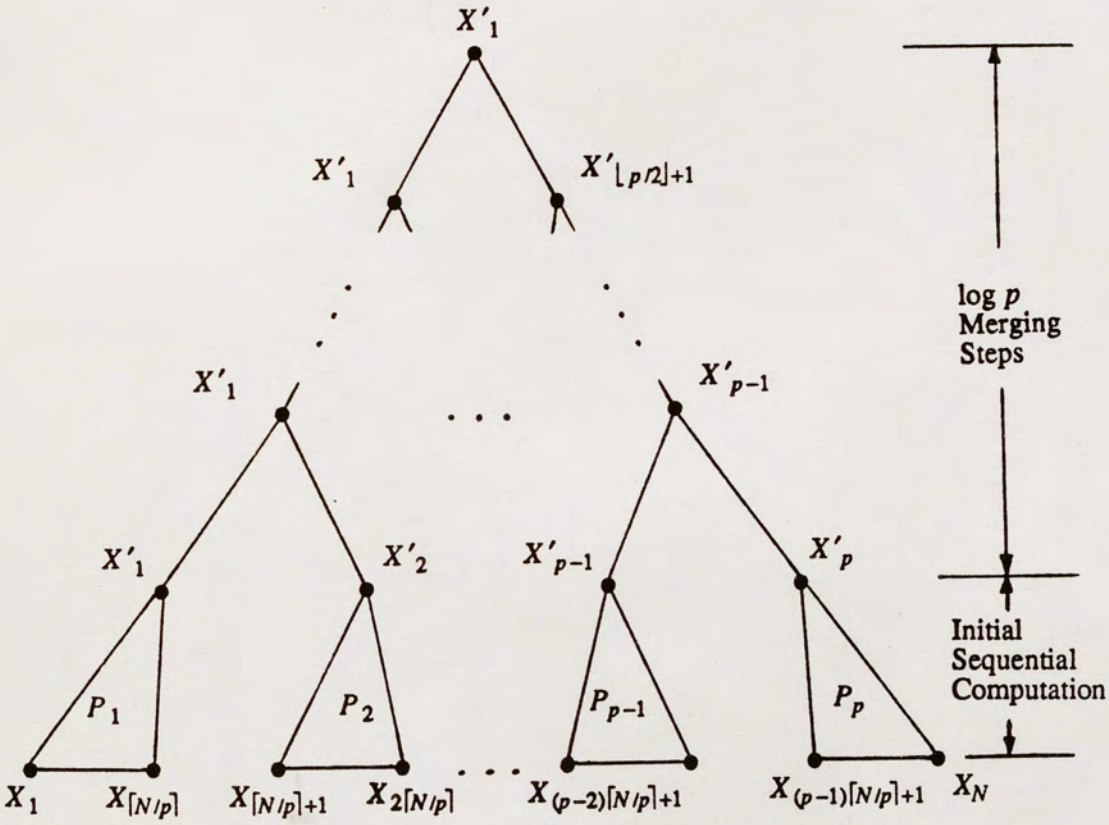

Figure 2.1. Execution Behavior of the Algorithm SMALLEST.

```
procedure SMALLEST;
begin
  for all i, 1 ≤ i ≤ p, do    (* initial computation *)
      parbegin
          P_i computes the minimum of ⌈N/p⌉ elements and stores in X'[i];
      parend;
  for each j, 1 ≤ j ≤ log p, do
    for all i, 0 ≤ i ≤ ⌊(p − 2^(j−1) − 1)/2^j⌋, do    (* merging *)
      parbegin
        (* P_(1+i2^j) compares elements X'[1 + i2^j] and X'[1 + i2^j + 2^(j−1)] *);

        if X'[1 + i2^j + 2^(j−1)] < X'[1 + i2^j]
           then X'[1 + i2^j] := X'[1 + i2^j + 2^(j−1)];
      parend;
end.
```

It is clear that the parallel algorithm SMALLEST has time complexity

$$T_p = \left\lceil \frac{N}{p} \right\rceil - 1 + \log p = \theta \left(\frac{N}{p} + \log p \right).$$

Also, the best-known sequential algorithm finds the smallest of N elements in $T_1 = N - 1 = \theta\,(N)$ time.

Special Case: Consider $p = \frac{N}{K}$, where K is a positive constant. We get $T_{\frac{N}{K}} = K - 1 + \log \frac{N}{K} = \theta\,(\log N)$, and this is the best that can be obtained because it meets the asymptotic lower bound for finding the minimum of N elements employ-

ing N processors on the EREW PRAM model (Cook, Dwork, and Reischuk 1986). Consequently, the speedup and efficiency are

$$S_{\frac{N}{K}} = \frac{T_1}{T_{\frac{N}{K}}} = \theta\left(\frac{N}{\log N}\right) < \theta\left(\frac{N}{K}\right) = \theta(p),$$

$$E_{\frac{N}{K}} = O\left(\frac{K}{\log N}\right) < O(1).$$

Hence the algorithm SMALLEST does not attain optimal speedup, by definition, when $\frac{N}{K}$ processors are used.

As pointed out at the beginning of this subsection, the raw speedup of the preceding algorithm can be increased by minimizing T_p with respect to p, the number of processors, crudely assuming that T_p is a continuous function of p. Following the rule of finding minima in Calculus (Fulks 1961), we make the partial derivative $\frac{\partial T_p}{\partial p} = 0$ and reach the condition $p = \frac{N}{\log e}$, which is to be satisfied. (Here $e \approx 2.718$ is the base of the natural logarithm.) As seen earlier, this condition fails to yield optimal speedup. On the other hand, if we make $\frac{\partial(pT_p)}{\partial p} = 0$ in order to maximize efficiency or minimize cost, we derive the condition $\log p = -\log e$, which cannot be satisfied. Physically, this result implies that $pT_p = \theta(N + \log p)$ is an increasing function of p (≥ 2), for a given value of N.

Our aim, now, is to show that the new performance measure $p(T_p)^2$ can be used to correctly derive the optimality condition. Minimizing this parameter with

respect to p would maximize speedup as well as efficiency. Accordingly, if we let $\frac{\partial(p (T_p)^2)}{\partial p} = 0$, we get

$$p (\log p + 2 \log e) = N \tag{2.1}$$

Since Equation (2.1) is transcendental in nature, it does not have an algebraic solution. However, for large values of N and p, there are asymptotic solutions to the corresponding inequality

$$p \log p \leq N \tag{2.2}$$

For example, $p \leq O(\frac{N}{\log N})$ is a solution. This form of solution will only be considered in later chapters of this dissertation wherever we encounter equations like (2.1) or inequalities like (2.2). We claim that the use of $p = \frac{N}{\log N}$ processors renders the parallel algorithm to be optimal. Though the parallel time T_p is still $\theta (\log N)$, the speedup now becomes $S_p = \theta (\frac{N}{\log N}) = \theta (p)$. Also, the algorithm is optimally adaptive for $p \leq \frac{N}{\log N}$.

In fact, any asymptotic solution to Equation (2.1) leads to optimal number of processors to be used. It implies that even if we have more processors at hand, say N in this example, we should not be tempted to grab all of them in order to solve the given problem. Otherwise many processors might remain idle, leading to inefficient processor-utilization. Physically, satisfying Equation (2.1), we make a trade-off between the initial computation time and the communication time (due to merging) in

the algorithm SMALLEST.

Note: In this context, it is worth mentioning that an exact solution to the equation $p \log p = N$ can be shown to be

$$p = \frac{N}{\sum_{i=1}^{\infty} (-1)^{i+1} \log^{(i)} N}.$$

We close this subsection with the following theorem and its corollary.

Theorem 2.1: An asymptotically general solution to Inequality (2.2) is given by $p = N^{1-\varepsilon}$, for $0 < \varepsilon \leq 1$.

Proof: Substituting $p = N^{1-\varepsilon}$ in Inequality (2.2), we obtain

$$(1 - \varepsilon) \log N \leq N^{\varepsilon}, \quad \text{for } \varepsilon > 0.$$

For $\varepsilon = 0$, this yields $\log N \leq 1$ which is satisfied only for $N \leq 2$. And for $\varepsilon = 1$, the computation model reduces to a uniprocessor system. Now,

$$\lim_{N \to \infty} \frac{N^{\varepsilon}}{\log N} = \lim_{N \to \infty} \frac{\varepsilon N^{\varepsilon - 1}}{(\log e)/N}, \quad \text{by L'Hospital's Rule (Fulks 1961).}$$

$$= \lim_{N \to \infty} \left(\frac{\varepsilon}{\log e} N^{\varepsilon}\right) \to \infty, \quad \text{for } \varepsilon > 0.$$

It implies that $\log N < N^{\varepsilon}$, for $0 < \varepsilon \leq 1$, and Inequality (2.2) is asymptotically satisfied for our choice of p. □

Corollary 2.1: The parallel algorithm SMALLEST is optimally adaptive for $p \leq N^{1-\varepsilon}$ processors, for $0 < \varepsilon \leq 1$.

Proof: The parallel time is $T_p = \theta\left(\frac{N}{p} + \log p\right) = \theta\,[N^{\varepsilon} + (1-\varepsilon)\log N] = \theta\,(N^{\varepsilon})$.

Therefore, the speedup is $S_p = \frac{T_1}{T_p} = \theta\,(N^{1-\varepsilon}) = \theta\,(p)$. □

Based on the preceding discussions, we conclude that the new performance measure $p\,(T_p)^2$ can be used to compute an optimal number of processors for a parallel algorithm as a function of the input size. The constant ε in Corollary 2.1 may be called a *scale factor*. By varying ε in the interval (0, 1], we can choose p.

We make one assumption in the asymptotic analysis of our algorithms in the rest of this dissertation. Whenever we take the derivative of $\log p$ with respect to p, we do not write the constant, $\log e \approx 1.44$, associated with the result.

CHAPTER 3

CONNECTED COMPONENTS AND SPANNING FOREST

Based on the divide-and-conquer strategy, we design two parallel algorithms — one for computing the connected-components and the other for a spanning-forest in an undirected graph. Initially, the connected components (or a spanning forest) of different subgraphs of the original graph are (or is) computed in parallel by different processors, each using an optimal sequential algorithm. Then the subsolutions are gradually merged to obtain the final solution. The input graph is represented by an unordered list of edges, and the use of simple and elegant data structures avoids memory read- and write-conflicts. Both the proposed algorithms achieve optimal speedups for all graphs using an appropriate number of processors, which is shown to be dependent on the density of the input graph.

The rest of the chapter is organized as follows. Section 3.1 defines the terminology and notation. Section 3.2 reviews briefly the previous works on the parallel connected-components and spanning-forest algorithms. In Sections 3.3 and 3.4 new algorithms are presented, and simple proofs of correctness have been provided. A lower bound on the processor-(time)^2 product for these parallel algorithms is also derived. Section 3.5 discusses the salient features of our algorithms and their design strategies.

3.1 Basic Definitions

An undirected graph $G = (V_G, E_G)$ is *connected* if there is a path between every pair of distinct vertices of the graph. A maximal connected subgraph of G is called its *connected component* (or just *component).* More formally, by the *connected components problem,* we mean the problem of computing the function $CON: V_G \rightarrow V_G$ such that

$CON(v_j) = \min \{k \mid k = j \text{ or } v_k \text{ is connected to } v_j \text{ by a path in } G\}$.

A *tree* is a connected acyclic graph. A subgraph $T = (V_G, E_T)$ of a connected graph G is a *spanning tree* of G if it is a tree containing all vertices of G. Clearly, $|E_T| = n - 1$. A *spanning forest* F of G is a collection of spanning trees, one for each connected component. Let $G' = (V_{G'}, E_{G'})$ be a subgraph of G. The set of edges with both end-vertices in $V_{G'}$ is denoted by $E(V_{G'})$. If $E_{G'} = E(V_{G'})$ then G' is the subgraph of G *induced by* $V_{G'}$.

Many polynomial graph theoretic (sequential) algorithms depend on basic search strategies, such as, *depth-first* or *breadth-first* search. In a depth-first search of a graph, we start at a vertex and each time an edge is discovered, the search is continued from the new vertex and is not renewed at the old vertex until all edges from the new vertex are exhausted. In a breadth-first search, on the other hand, we start at a vertex and first search all vertices at a distance of one from it. Next all vertices at a distance of two from the start-vertex are searched and so forth, until the graph is traversed.

The input graph is stored in the common shared memory as an $m \times 2$ array, LIST, an unordered list of edges labeled as $e_1, e_2, \ldots, e_m$. The ith edge $e_i = (u, v)$, with $u < v$ as a convention, is stored in LIST[i], where LIST[i, 1] := u and LIST[i, 2] := v, for $1 \leq i \leq m$ and $1 \leq u < v \leq n$.

3.2 Previous Works

The connected-components and spanning-forest algorithms, besides being important in their own right, can also serve as basic subroutines in designing more complex algorithms, as are evident from the results in Chapter 4. Therefore, considerable work has been done to solve these problems on different classes of PRAM models. Table 3.1 reviews the time and processor complexities of the available literature on fast and efficient, parallel connected-components or spanning-forest algorithms. For detailed discussions on several such algorithms, readers may refer to Moitra and Iyenger (1987) or Quinn and Deo (1984). The basic strategies used in developing these algorithms are breadth-first search, transitive closure, and vertex collapse. As can be observed from Table 3.1, many of the earlier results are based on the assumption of unbounded parallelism. Moreover, most of these algorithms require adjacency matrix as the input data structure so that the underlying graph problem can be solved by manipulating matrices. Consequently, these techniques lead to optimal or near-optimal algorithms only for dense graphs. For example, optimal speedups are achieved for connected-components or spanning-forest algorithms due to Chin, Lam,

and Chen (1982), and Vishkin (1984) using $p \leq \frac{n^2}{\log^2 n}$ processors. Kwan and Ruzzo (1984) implemented a spanning-forest algorithm on the CREW model with $p \leq \frac{m}{\log n}$ processors, taking care of sparse graphs. Using adjacency list as the data structure, Koubek and Krsnakova (1985) designed near-optimal algorithms for connected components on both CREW and EREW models which utilize $O(\frac{m+n}{\log n})$ processors and $O(m+n)$ space. The connected-components algorithms due to Cole and Vishkin (1986) are optimal for $m \geq n \log^* n$, where $\log^* n = \min \{i \mid \log^{(i)} n \leq 1\}$ is the iterated logarithm, when the model is CRCW or CREW and the edges of the graph are represented in a vector of length $2m$ in a forward-star fashion; however, on the EREW model the algorithm is near-optimal. Kruskal, Rudolph and Snir (1986) used an unordered list of edges as the data structure. Their implementation of the connected-components or spanning-forest requires $O(pn + m)$ space. On the CREW model the algorithm attains the optimal speedup satisfying $p \leq \sqrt{\frac{m}{\log m}}$ and $\log p \leq \frac{m}{n}$; while on the EREW model the optimal speedup (for all but the sparsest graphs where $m = \theta(n)$) is achieved when $p \leq m^{\frac{1}{2} - \varepsilon}$, $0 < \varepsilon \leq 1$, and $\log p \leq \frac{m}{n}$.

In this chapter, we develop parallel algorithms for connected components and spanning forest. They achieve optimal speedups for all graphs by using $p \leq \frac{m/n}{\log(m/n)}$ processors. For dense graphs where $m = \theta(n^2)$, our algorithms are

asymptotically faster than those of Kruskal, Rudolph, and Snir (1986). The implementation of these algorithms require $O(pn + m)$ space, which is optimal by our choice of p.

In passing we now mention some algorithms for the problems under consideration on fixed connection computers. Doshi and Varman (1987) have described an optimal algorithm for spanning forest on a fixed-size linear array. Yeh and Lee (1984) have developed connected-components algorithm on a tree-structured parallel computer. Huang (1985) has implemented connected-components and spanning-forest algorithms on mesh-of-trees networks. On mesh-connected computers, Hambrusch (1983), Nassimi and Sahni (1980), and Stout (1985) have developed algorithms for connected components, and Atallah and Kosaraju (1984) have presented an algorithm for spanning forest. The spanning-forest algorithms due to Miller and Stout (1987a, 1987b) are designed for Pyramid and hypercube machines. For an overview of the time and processor complexities of these and several other algorithms, refer to Das, Deo, and Prasad (1988b), where authors have designed optimally adaptive connected-components and spanning-forest algorithms on hypercube computers.

TABLE 3.1. PARALLEL GRAPH ALGORITHMS ON PRAM MODELS †

PROBLEM	MODEL	TIME (T_p)	PROCESSORS	RESEARCHERS
Connected Components, Spanning Forest	CRCW	$O(\log n)$	$n + 2m$	Shiloach & Vishkin (1982)
		$O(\frac{n^2}{p})$	p	Vishkin (1984)
		$O(\log n)$	$O(m + n)$	Koubek & Krsnakova (1985)
		$O(\log n \log^{(2)} n \log^{(3)} n)$	$O((m + n) \frac{\alpha(m, n)}{T_p})$	Cole & Vishkin (1986)
	CREW	$O(\log n \log d)$	$O(\frac{n^3}{\log n})$	Savage & Ja' Ja' (1981)
		$O(\log^2 n)$	$O(m + n \log n)$	Savage & Ja' Ja' (1981)
		$O(\log^2 n)$	$n \left\lceil \frac{n}{\log n} \right\rceil$	Hirschberg et al. (1979)
		$O(\frac{n^2}{p} + \log^2 n)$	p	Chin et al. (1982)
		$O(\log^2 n)$	$O(\frac{m + n}{\log n})$	Koubek & Krsnakova (1985)
		$O(\log^2 n)$	$n + 2m$	Wyllie (1979)
		$O(\frac{m \log n}{p} + \log n \log p)$	p	Kwan & Ruzzo (1984)
		$O(\frac{m}{p} + \frac{n \log p}{p} + p \log p)$	p	Kruskal et al. (1986)
		$O(\log^2 n)$	$O((m + n) \frac{\alpha(m, n)}{T_p})$	Cole & Vishkin (1986)
	EREW	$O(\log^2 n)$	$O(\frac{n^2}{\log n})$	Nath & Maheshwari (1982)
		$O(\frac{m}{p} + p^{1+\varepsilon} + \frac{n \log p}{p})$	p	Kruskal et al. (1986)
		$O(\log^2 n)$	$O(\frac{m + n}{\log n})$	Koubek & Krsnakova (1985)
		$O(\log^2 n)$	$O((m + n) \frac{\log^{(2)} n}{T_p})$	Cole & Vishkin (1986)
		$O(\frac{m}{p} + n \log p)$	p	Das (*this dissertation*)

† d is diameter of graph; $0 < \varepsilon \leq 1$; $\alpha(m, n)$ is an inverse Ackermann's function.

3.3 Connected Components

The parallel algorithm PARALLEL_CONNECT, based on a divide-and-conquer strategy, computes the connected components of an undirected graph, represented as an unordered list (LIST) of edges. The algorithm uses a linear array ROOT (of size pn), stored in the shared memory. At the termination of the algorithm, the number of distinct entries in subarray ROOT[1 . . n] is the number of connected components in the graph. The element ROOT[j], for $1 \le j \le n$, stores the root of the component to which the vertex v_j belongs. The (final) *root* of a component is the smallest-indexed vertex in that component. Initially, ROOT[$(i-1)n + j$] := j for $1 \le j \le n$ and $1 \le i \le p$, indicating that vertex v_j is a component by itself in the subgraph processed by processor P_i. After the execution of the algorithm PARALLEL_CONNECT, the set of vertices $V^i = \{v_j \mid \text{ROOT}[\,j] = v_i$ and $1 \le i \le j \le n\}$ belongs to a connected component numbered i. The parallel algorithm is sketched as follows.

```
procedure PARALLEL_CONNECT;
  begin
    for all i, 1 ≤ i ≤ p, do      (* initialization *)
        parbegin
            for each j, 1 ≤ j ≤ n, do
                    ROOT[(i − 1)n + j] := j;
        parend;
    for all i, 1 ≤ i ≤ p, do      (* initial computation *)
        parbegin
            The processor P_i constructs an adjacency list of a subgraph G_i of n
```

vertices and $\left\lceil \frac{m}{p} \right\rceil$ edges stored in LIST[$(i-1)\left\lceil \frac{m}{p} \right\rceil + 1 \;..\; i\left\lceil \frac{m}{p} \right\rceil$];
computes the connected components of G_i by a sequential algorithm; and outputs the result in ROOT[$(i-1)n+1 \;..\; in$], where ROOT[$(i-1)n+j$] contains the root of the component of the subgraph G_i to which the vertex v_j belongs.

parend;

MERGE_CONNECT; (* procedure for parallel merging *)

# of connected components <— # of distinct entries in subarray ROOT[$1 \;..\; n$];

end.

Note that the information contained in the ROOT subarray of an individual processor P_i *induces a forest* $f_i = (V_G, E_{f_i})$ of n vertices with the edge-set $E_{f_i} = \{(u, v) \mid \text{ROOT}[(i-1)n+v] = u \text{ and } 1 \le u < v \le n\}$. If a vertex is a root by itself, the induced self-loop is discarded. Some of the edges, say (y, z), corresponding to this forest may not exist in the original subgraph G_i processed by P_i, but arise here due to the existence of a path from y to z in G_i. It implies the following Lemma.

Lemma 3.1: The induced forest f_i preserves the connectedness of subgraph G_i, for $1 \le i \le p$.

Proof: Assume that in the subgraph G_i originally processed by processor P_i, $1 \le i \le p$, there is an edge (u, v) between two given vertices u and v, with $u < v$. Then, after the initial computation of the algorithm PARALLEL_CONNECT, the ROOT subarray of P_i will contain ROOT[$(i-1)n+v$] = ROOT[$(i-1)n+u$]

$= ru$, say, assuming that the sequential connected-components algorithm performs correctly. In the corresponding induced forest f_i, we get a path (u, ru, v) or an edge (u, v) depending on whether vertex u is distinct from or identical to its root-vertex ru.

Next, consider that there is an edge-sequence of length greater than one between the given vertices u and v of G_i. Without loss of generality, let the vertices along the edge-sequence be $(u, y, \ldots, w, \ldots, z, v)$, where any or both of y and z may be absent. Now, if u is the smallest-indexed vertex among these, by the preceding argument either the edge (u, v) or the path (u, ru, v) exists in the generated induced forest. On the other hand, if w is the smallest-indexed vertex, then $\text{ROOT}[(i-1)n + u] = \text{ROOT}[(i-1)n + y] = \ldots = \text{ROOT}[(i-1)n + z] = \text{ROOT}[(i-1)n + v] = \text{ROOT}[(i-1)n + w] = rw$, say. Accordingly, the induced forest will have the path (u, rw, v).

Thus, if two vertices are connected in the subgraph G_i processed by P_i, then they remain connected in the induced forest f_i generated by it. □

As a consequence of Lemma 3.1, the problem of merging the connected components of two subgraphs G_i and G_j computed by processors P_i and P_j, respectively, essentially reduces to the problem of merging two induced forests (available in ROOT subarrays of P_i and P_j), each having at most $n - 1$ edges. To simplify presenting parallel algorithm MERGE_CONNECT, the number of processors will be assumed to be $p = 2^b$ for $b \geq 1$.

procedure MERGE_CONNECT; (* $\log p$ steps of merging *)
begin
for each k, $1 \le k \le b$, **do**

for all i, $0 \le i \le \left\lfloor \frac{2^b - 2^{k-1} - 1}{2^k} \right\rfloor$, **do**

(* Processor P_{1+i2^k} merges its solution with that of $P_{1+i2^k+2^{k-1}}$ *)
parbegin
The processor P_{1+i2^k} extracts the edges (excluding duplicate edges) of the induced forests contained in subarrays ROOT[$(i2^k)n + 1$ .. $(i2^k + 1)n$] and ROOT[$(i2^k + 2^{k-1})n + 1$ .. $(i2^k + 2^{k-1} + 1)n$]; constructs an adjacency list; and computes the connected components of this induced merged-forest by a sequential algorithm. The output is stored in the ROOT subarray of processor P_{1+i2^k}.
parend;
end.

Lemma 3.2: During an iteration k, for $1 \le k \le b = \log p$, when two subsolutions obtained by processors P_{1+i2^k} and $P_{1+i2^k+2^{k-1}}$ are merged, the element ROOT[$(i2^k)n + j$], for $1 \le j \le n$, is assigned the smallest-indexed vertex as the root of the merged component to which the vertex v_j belongs.

Proof: We apply an induction on k. During the first merging iteration (when $k = 1$), processor P_{2i+1}, for $0 \le i \le \left\lfloor \frac{p}{2} - 1 \right\rfloor$, merges its subsolution with that of processor P_{2i+2}. By Lemma 3.1, the forests f_{2i+1} and f_{2i+2} induced by ROOT subarrays of these two processors preserve, respectively, the connectedness of subgraphs

G_{2i+1} and G_{2i+2} originally assigned to them. (This can be treated as the base case when $k = 0$.) Assuming that a single adjacency list can be correctly constructed by extracting the edges from these two induced forests, and that the implementation of the sequential connected-components algorithm is correct, the ROOT subarray of processor P_{2i+1} will contain the merged solution. In other words, the element ROOT[$2in + j$], for $1 \leq j \leq n$, stores the root of the component to which vertex v_j belongs in the so-far merged solution. Similar argument holds for all merging iterations. □

Theorem 3.1: The algorithm PARALLEL_CONNECT correctly computes the connected components of $G = (V_G, E_G)$ without memory read- or write-conflicts.

Proof: We prove the theorem by showing that if any two vertices u and v, with $u < v$, belong to a particular connected component in the original graph G, then they remain so in the forest f_1 induced by the ROOT subarray of processor P_1 at the termination of the algorithm PARALLEL_CONNECT. Assume that there is an edge (u, v) in G which is initially assigned to a processor P_β, $1 \leq \beta \leq p$. The message-flow during an execution of the algorithm in an 8-processor parallel computer is as shown in Figure 3.1. Generalizing this to a p-processor system, there is a unique directed route of communication along which the connected components produced by the processor P_β passes through during different merging steps and finally reaches the processor P_1. Since vertices u and v were connected initially at P_β, they belong

to the same connected component (by Lemma 3.1) in all the forests induced by various ROOT subarrays along this specified route. Also, after each merging iteration, they receive the smallest-indexed vertex as the root of the component to which they belong (by Lemma 3.2).

Next, if the given vertices u and v of G are connected via an edge-sequence of length greater than one, they will belong to the same connected component in the induced forest f_1 because we can apply the preceding argument on each edge in the sequence. Since each processor accesses exclusively different parts of the shared memory, there is no read- or write-conflict of a memory cell at any stage of the parallel algorithm. □

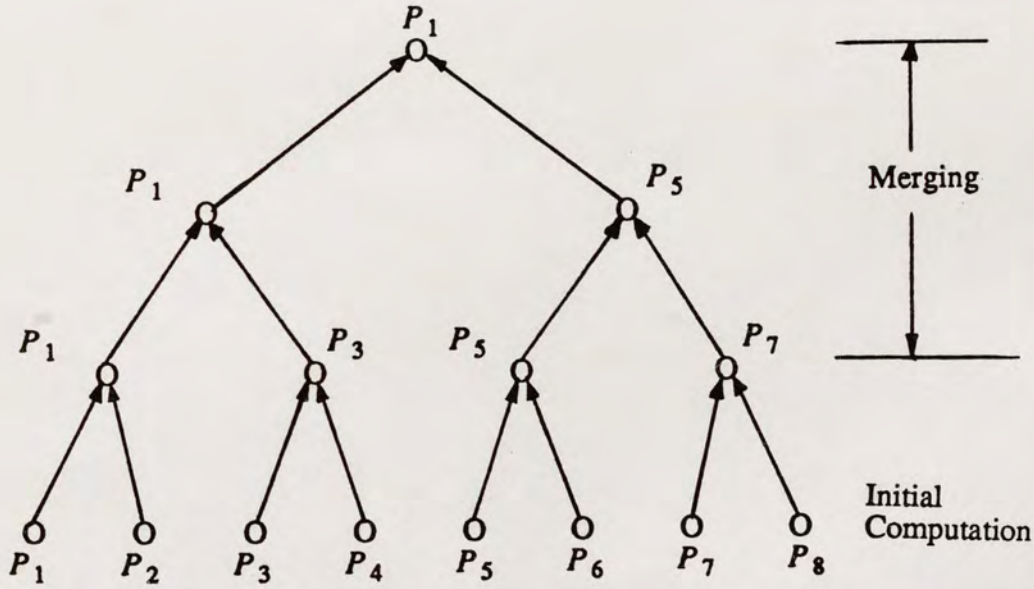

Figure 3.1. Message Communication in an 8-Processor Computer During an Execution of Algorithm PARALLEL_CONNECT.

3.3.1 An Example

We illustrate the algorithm PARALLEL_CONNECT with a graph in Figure 3.2, using $p = 4$ processors. The input consists of $n = 14$ vertices and an unordered list of edges,

$$\text{LIST} = \{(v_1, v_3), (v_3, v_{10}), (v_1, v_{11}), (v_3, v_{11}), (v_1, v_2), (v_7, v_9), (v_8, v_9), (v_7, v_8), (v_{10}, v_{11}), (v_5, v_6), (v_{13}, v_{14}), (v_{12}, v_{13}), (v_{10}, v_{12}), (v_4, v_9), (v_{12}, v_{14})\}.$$

Since there are $m = 15$ edges, initially each of the processors P_1, P_2, and P_3 is allocated 4 edges while P_4 gets the remaining 3 edges. Each processor constructs the adjacency list of its own edges, computes the roots of the components therein, and stores the result in its portion of the array ROOT, which is of size $4 \times 14 = 56$. For ease of understanding, we partition the ROOT array into four separate arrays, namely R_1, R_2, R_3, and R_4 each of size 14. The result of the initial computation of connected components by individual processors is shown in the following.

G_1: $\{(v_1, v_3), (v_3, v_{10}), (v_1, v_{11}), (v_3, v_{11})\}$

R_1: 1 2 1 4 5 6 7 8 9 1 1 12 13 14

G_2: $\{(v_1, v_2), (v_7, v_9), (v_8, v_9), (v_7, v_8)\}$

R_2: 1 1 3 4 5 6 7 7 7 10 11 12 13 14

G_3: $\{(v_{10}, v_{11}), (v_5, v_6), (v_{13}, v_{14}), (v_{12}, v_{13})\}$

R_3: 1 2 3 4 5 5 7 8 9 10 10 12 12 12

G_4: $\{(v_{10}, v_{12}), (v_4, v_9), (v_{12}, v_{14})\}$

R_4: 1 2 3 4 5 6 7 8 4 10 11 10 13 10

The forests induced by root-information of different processors are depicted in Figure 3.3. We see that the forest f_1 induced by R_1 contains the edge (v_1, v_{10}), which did not exist in the original subgraph G_1 allocated to processor P_1. Also, self-loops are not included in the induced forests. The components involving single vertices have not been shown. Next, the subsolutions obtained by different processors are merged as follows, in two iterations. After the first merging iteration, the forests induced by arrays R_1 and R_3 are as shown in Figure 3.4.

(a) First merging iteration:

Merging of f_1 and f_2

R_1: 1 2 1 4 5 6 7 8 9 1 1 12 13 14 (before merging)

1 1 1 4 5 6 7 7 7 1 1 12 13 14 (merged solution)

Merging of f_3 and f_4

R_3:	1	2	3	4	5	5	7	8	9	10	10	12	12	12	(before merging)
	1	2	3	4	5	5	7	8	4	10	10	10	10	10	(merged solution)

(b) Second merging iteration:

Merging of f_1 and f_3

R_1:	1	1	1	4	5	6	7	7	7	1	1	12	13	14	(before merging)
	1	1	1	4	5	5	4	4	4	1	1	1	1	1	(merged solution)

The final contents of array R_1 after the second merging iteration is the output of the algorithm PARALLEL_CONNECT, and is interpreted as follows. There are three connected components of the graph in Figure 3.2, represented by the distinct integers (1, 4, and 5) in R_1. Vertices v_1, v_2, v_3, v_{10}, v_{11}, v_{12}, v_{13}, and v_{14} are in component numbered 1; vertices v_4, v_7, v_8, and v_9 are in component numbered 4; and vertices v_5 and v_6 are in component numbered 5. This can be readily seen from Figure 3.5.

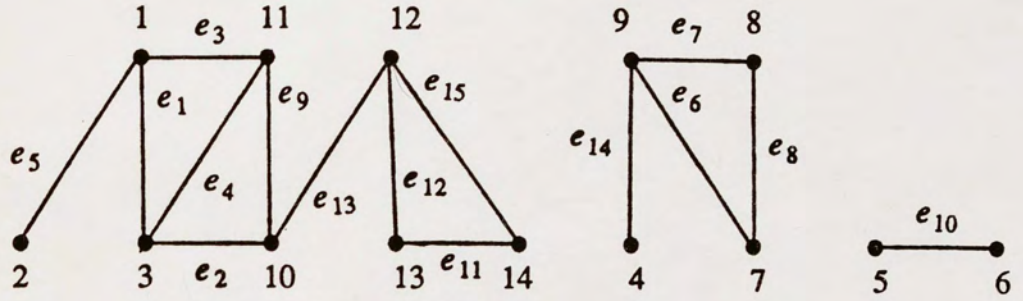

Figure 3.2. A Disconnected Graph.

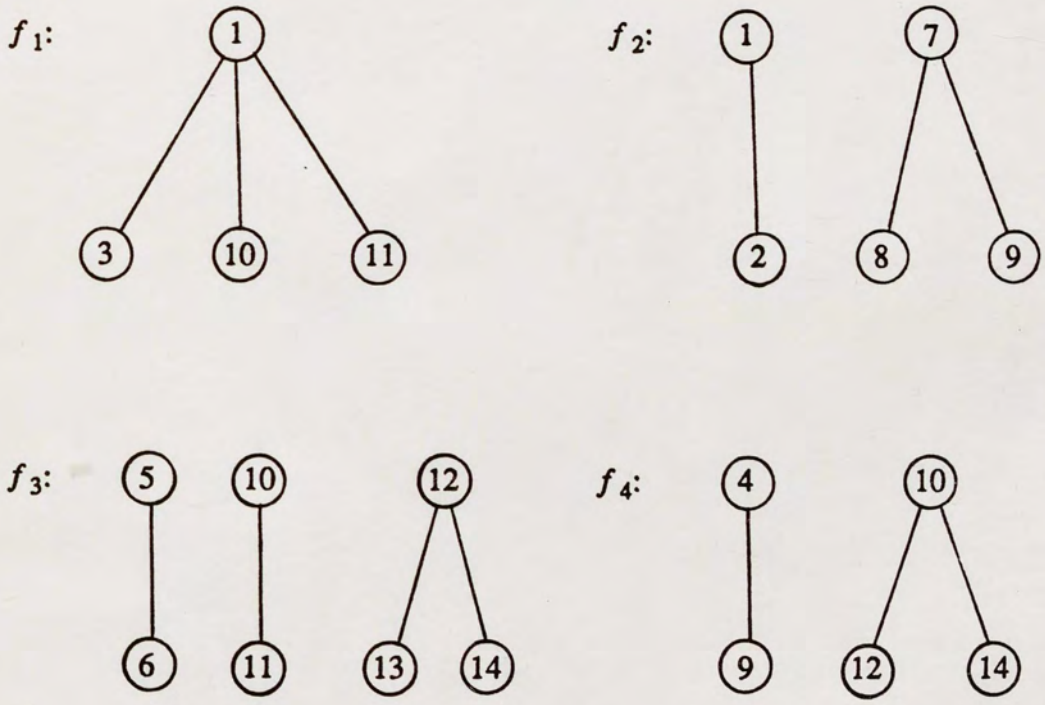

Figure 3.3. Induced Forests After Initial Computation.

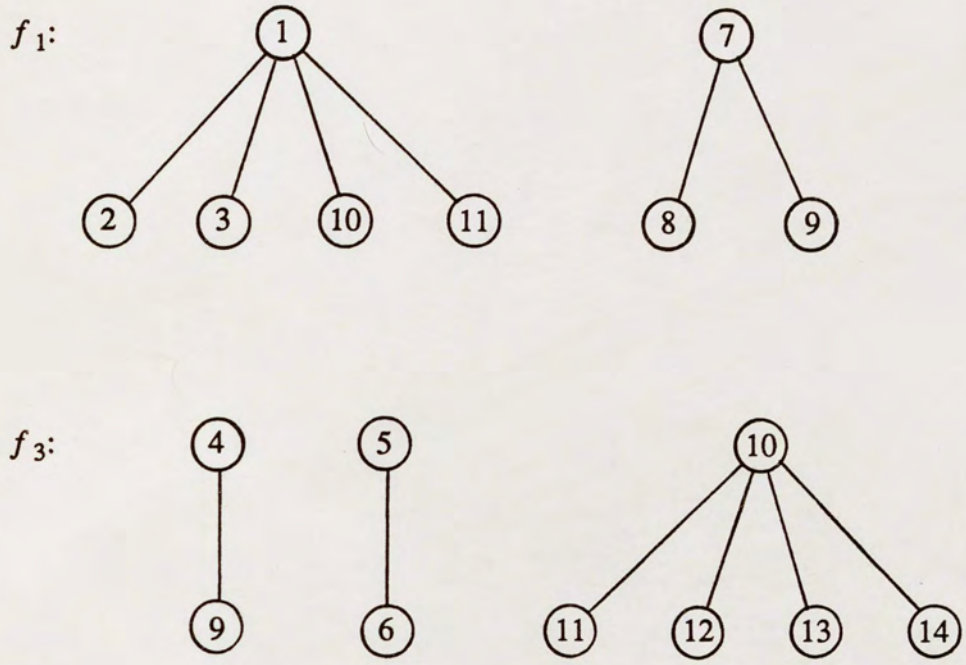

Figure 3.4. Induced Forests After First Merging Iteration.

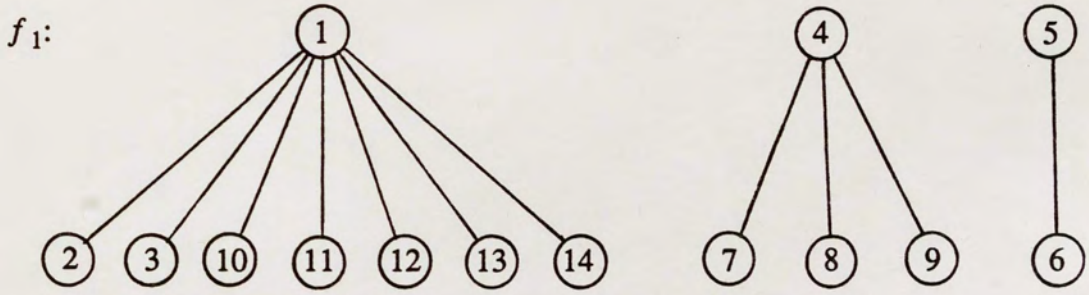

Figure 3.5. Connected Components of the Graph in Figure 3.2.

3.3.2 Complexity Analysis

Let T_p^{CON} be the total time required by the algorithm PARALLEL_CONNECT using p processors, and T_1^{CON} be the time to find the connected components by the best-known sequential algorithm, say the one based on a breadth-first or a depth-first search (Reingold et al. 1977; Tarjan 1972). It is known that $T_1^{CON} = O(m + n)$ for a graph with n vertices and m edges. The same amount of time is needed to form a linked adjacency list given the unordered list of edges as the input. If M^{CON} denotes the time required by the procedure MERGE_CONNECT and t_c , the time required for finding the number of distinct components from the subarray ROOT[1 . . n], we can write $T_p^{CON} = K_1 (\left\lceil \frac{m}{p} \right\rceil + n) + M^{CON} + t_c$, where K_1 is a positive constant. The first term on the right-hand side of the preceding expression represents the time required by each processor for constructing the adjacency list and finding the connected components of $\left\lceil \frac{m}{p} \right\rceil$ edges, including the time required for initializing the array ROOT. Using a single processor, $t_c = n$. Each execution of the **parbegin . . . parend** loop of procedure MERGE_CONNECT requires $K_1 [(2n - 2) + n] = K_1 (3n - 2)$ time units in the worst-case. This time includes constructing an adjacency list of at most $2n - 2$ edges of two induced forests and computing their connected components. Since there are $\log p$ merging iterations, $M^{CON} = K_1 (3n - 2) \log p$. Therefore, the overall time complexity and the speedup of the algorithm PARALLEL_CONNECT are, respectively,

$$T_p^{CON} = K_1 \left(\left\lceil \frac{m}{p} \right\rceil + n \right) + K_1 (3n - 2) \log p + n \ , \quad \text{and}$$

$$S_p^{CON} = \frac{T_1^{CON}}{T_p^{CON}} = \frac{O(m+n)}{K_1 \left(\left\lceil \frac{m}{p} \right\rceil + n \right) + K_1 (3n - 2) \log p + n} \ .$$

The following theorem gives an asymptotic complexity on the performance of the algorithm PARALLEL_CONNECT.

Theorem 3.2: Let the connected components of a graph G of n vertices and m edges be computed using p processors in time T_p by the parallel divide-and-conquer algorithm, PARALLEL_CONNECT. Then $p(T_p)^2 \geq \theta\,(mn \log \frac{m}{n})$, and the equality holds when $p = \theta\,(\frac{m/n}{\log(m/n)})$.

Proof: When m is sufficiently large compared to p, $\left\lceil \frac{m}{p} \right\rceil$ may be approximated to $\frac{m}{p}$. Ignoring the constants in the foregoing analysis of parallel time does not affect the validity of the proof presented below. So we write,

$$T_p \geq \frac{m}{p} + n \log p, \quad \text{for } 1 \leq p \leq \frac{m}{n}$$

i.e.,

$$p(T_p)^2 \geq p\left(\frac{m}{p} + n \log p\right)^2 = \frac{m^2}{p} + 2mn \log p + pn^2 \log^2 p \qquad (3.1)$$

We consider the following three cases to complete the proof.

(i) When $p = \theta\left(\frac{m/n}{\log(m/n)}\right)$, we get $p(T_p)^2 \geq \theta\left(mn \log \frac{m}{n}\right)$.

(ii) When $p < \theta\left(\frac{m/n}{\log(m/n)}\right)$, the first term on the right hand side of Expression (3.1) is $\frac{m^2}{p} > \theta\left(mn \log \frac{m}{n}\right)$.

(iii) When $p > \theta\left(\frac{m/n}{\log(m/n)}\right)$, the third term on the right hand side of (3.1) becomes $pn^2 \log^2 p > \theta\left(mn \log \frac{m}{n}\right)$, because $\log^2 p > \left(\log\left(\frac{m/n}{\log(m/n)}\right)\right)^2 \geq \theta\left(\log^2 \frac{m}{n}\right)$.

The preceding three cases imply that the lower bound on the product $p(T_p)^2$ is given by $\theta\left(mn \log \frac{m}{n}\right)$, which is achieved by the use of $p = \theta\left(\frac{m/n}{\log(m/n)}\right)$ processors. Since the initial granularity (or the amount of data allocated to each processor) of the parallel divide-and-conquer algorithm is $\left\lceil \frac{m}{p} \right\rceil$, the optimal granularity is $\theta\left(n \log \frac{m}{n}\right)$. □

Corollary 3.1:

(a) When the given graph is dense, i.e., $m = \theta(n^2)$, $p(T_p)^2 = \Omega(n^3 \log n)$, and $p = \theta\left(\frac{n}{\log n}\right)$. Hence, $T_p = \Omega(n \log n)$. Since $T_1 = \theta(n^2)$, the speedup $S_p = \theta\left(\frac{n}{\log n}\right) = \theta(p)$.

(b) When the given graph is sparse, i.e., $m = \theta(n)$, $p(T_p)^2 = \Omega(n^2)$, and $p = \theta(1)$. Thus, $T_p = \Omega(n)$. Since $T_1 = \theta(n)$, $S_p = \theta(1) = \theta(p)$. □

In the foregoing analysis, the asymptotic optimal value of p can be derived as follows. The running time of the algorithm PARALLEL_CONNECT is approximated to

$$T_p = K_1 \left(\frac{m}{p} + n\right) + K_1 (3n - 2) \log p + n \tag{3.2}$$

Now, $p(T_p)^2$ achieves a minimum value when $\frac{\partial(p(T_p)^2)}{\partial p} = 0$. That is,

$$(T_p)^2 + 2pT_p \frac{\partial T_p}{\partial p} = 0 \tag{3.3}$$

Computing $\frac{\partial T_p}{\partial p}$ from Equation (3.2) and substituting in Equation (3.3) we get, $T_p - \frac{2mK_1}{p} + 2K_1(3n - 2) = 0$ which yields after the substitution of the value of T_p from Equation (3.2),

$$p \log p \left[3K_1 + \frac{(7K_1 + 1)}{\log p} - \frac{2K_1}{n} - \frac{4K_1}{n \log p} \right] = K_1 \left(\frac{m}{n}\right).$$

For large values of m, n, and p we can write $p \log p \approx \frac{1}{3}\left(\frac{m}{n}\right)$, i.e., $p \le O\left(\frac{m/n}{\log(m/n)}\right)$, following the argument presented in Section 2.4.1. Therefore, the algorithm PARALLEL_CONNECT is optimal for any graph using p processors, where $1 \le p \le \frac{m/n}{\log(m/n)}$. For example, for dense graphs this algorithm is

optimally adaptive up to $1 \leq p \leq \frac{n}{\log n}$, whereas for sparse graphs only up to a constant number of processors. Since the average density of a graph is proportional to $\frac{m}{n}$, the optimal performance is density-dependent.

3.4 Spanning Forest

The parallel algorithm PARALLEL_FOREST finds a spanning forest of a given undirected graph, also based on divide-and-conquer strategy. Each processor P_i has a queue Q_i (stored in the shared memory) of size at most $n - 1$. A subgraph is assigned to P_i which stores in its queue the label of the edges as they are being included in the forest. At the termination of the algorithm, all edges in the spanning forest are available in the queue Q_1. The information regarding the tree-roots of its vertices is stored in an array ROOT[1 . . n], stored in the shared memory. (This is unlike the algorithm PARALLEL_CONNECT, where the array ROOT has size pn.) The *root* of a tree in a forest is the smallest-indexed vertex in that tree. Usually, finding a spanning forest of a graph is concerned with the edges to be included in the forest and may not involve the computation of the array ROOT. However, for detecting bipartiteness (Section 4.5), we need to identify the roots of the trees. Though the algorithmic description of the procedures PARALLEL_FOREST and MERGE_FOREST can be derived (with appropriate modifications) from those of the parallel connected-components algorithm, we present them in the following for completeness.

procedure PARALLEL_FOREST;

begin

for all i, $1 \le i \le p$, **do** (* initialization *)

parbegin

for each j, $(i-1)\left\lceil \frac{n}{p} \right\rceil + 1 \le j \le i\left\lceil \frac{n}{p} \right\rceil$, **do**

ROOT[$(i-1)n + j$] := j;

parend;

for all i, $1 \le i \le p$, **do** (* initial computation *)

parbegin

The processor P_i constructs an adjacency list of a subgraph G_i of n vertices and $\left\lceil \frac{m}{p} \right\rceil$ edges stored in LIST[$(i-1)\left\lceil \frac{m}{p} \right\rceil + 1 \;..\; i\left\lceil \frac{m}{p} \right\rceil$]; computes a spanning forest F_i of G_i by a sequential breadth-first or depth-first search. P_i stores the edges of F_i in the queue Q_i and counts the number of edges included in F_i .

parend;

if the number of edges in any forest F_i, $1 \le i \le p$, is $n-1$

then copy the forest-edges in Q_i to Q_1

else MERGE_FOREST; (* procedure for parallel merging *)

Processor P_1 constructs an adjacency list of the spanning-forest-edges in Q_1 and generates the array ROOT by sequential graph-search;

end.

To simplify presenting the algorithm, we assume that the number of processors is $p = 2^b$, $b \ge 1$.

procedure MERGE_FOREST; (* $\log p$ steps of merging *)

begin

for each k, $1 \le k \le b$, **do**

begin

for all i, $0 \le i \le \left\lfloor \frac{2^b - 2^{k-1} - 1}{2^k} \right\rfloor$, **do**

(* Processor P_{1+i2^k} merges its solution with that of $P_{1+i2^k+2^{k-1}}$ *)

parbegin

Processor P_{1+i2^k} constructs an adjacency list of the edges available in Q_{1+i2^k} and $Q_{1+i2^k+2^{k-1}}$, and computes a spanning forest by a sequential algorithm. The edges of the resulting forest F_{1+i2^k} are stored in the queue Q_{1+i2^k} along with a count on the number of edges therein.

parend;

if any forest contains $n - 1$ edges

then copy those forest-edges into Q_1 and **exit;**

end

end.

Theorem 3.3: The set of edges stored in queue Q_1 of processor P_1 at the termination of the algorithm PARALLEL_FOREST defines a spanning forest of the graph $G = (V_G, E_G)$.

Proof: Assuming that the sequential spanning-forest algorithm performs correctly, the theorem can be proved along the same line as Theorem 3.1. □

3.4.1 An Example

In the following we illustrate the algorithm PARALLEL_FOREST on the graph in Figure 3.2, using 4 processors. The processor P_i for $1 \le i \le 4$ has a queue Q_i. The result of the initial computation of spanning forests by individual processors using breadth-first search is presented below. In this example, for the sake of clarity, a forest-edge in a queue is represented not by its label but by the pair of its end-

vertices, as stored in the array LIST. Note that the spanning forest produced will depend on the order in which the breadth-first traversal works on the vertices of G; here we assume some fixed but arbitrary order.

G_1: $\{(v_1, v_3), (v_3, v_{10}), (v_1, v_{11}), (v_3, v_{11})\}$

Q_1: $\{(v_1, v_3), (v_1, v_{11}), (v_3, v_{10})\}$

G_2: $\{(v_1, v_2), (v_7, v_9), (v_8, v_9), (v_7, v_8)\}$

Q_2: $\{(v_1, v_2), (v_7, v_9), (v_7, v_8)\}$

G_3: $\{(v_{10}, v_{11}), (v_5, v_6), (v_{13}, v_{14}), (v_{12}, v_{13})\}$

Q_3: $\{(v_5, v_6), (v_{10}, v_{11}), (v_{12}, v_{13}), (v_{13}, v_{14})\}$

G_4: $\{(v_{10}, v_{12}), (v_4, v_9), (v_{12}, v_{14})\}$

Q_4: $\{(v_4, v_9), (v_{10}, v_{12}), (v_{12}, v_{14})\}$

Now the subsolutions obtained by different processors are merged. While merging two forests, say F_1 and F_2, the processor P_1 first constructs an adjacency list of the edges in queues Q_1 and Q_2, computes a spanning forest by a breadth-first search and then stores the forest-edges in Q_1. There are two merging iterations in our example. At the end of the second iteration, the array ROOT is computed.

(a) First merging iteration:

Merging of Q_1 and Q_2

Q_1: $\{(v_1, v_3), (v_1, v_{11}), (v_1, v_2), (v_3, v_{10}), (v_7, v_9), (v_7, v_8)\}$ (merged forest)

Merging of Q_3 and Q_4

Q_3: $\{(v_4, v_9), (v_5, v_6), (v_{10}, v_{11}), (v_{10}, v_{12}), (v_{12}, v_{13}), (v_{12}, v_{14})\}$ (merged forest)

(b) Second merging iteration:

Merging of Q_1 and Q_3

Q_1: $\{(v_1, v_3), (v_1, v_{11}), (v_1, v_2), (v_3, v_{10}), (v_4, v_9), (v_5, v_6),$

$(v_7, v_9), (v_7, v_8), (v_{10}, v_{12}), (v_{12}, v_{13}), (v_{12}, v_{14})\}$ (merged forest)

ROOT: 1 1 1 4 5 5 4 4 4 1 1 1 1 1

The output of the algorithm PARALLEL_FOREST is interpreted as follows. A spanning forest of the graph in Figure 3.2 consists of eleven edges contained in the queue Q_1, and is depicted in Figure 3.6. The number of distinct integers in the array ROOT (which is three here) is the number of trees in this spanning forest. With the help of ROOT, we determine the root of a tree to which a forest-edge belongs.

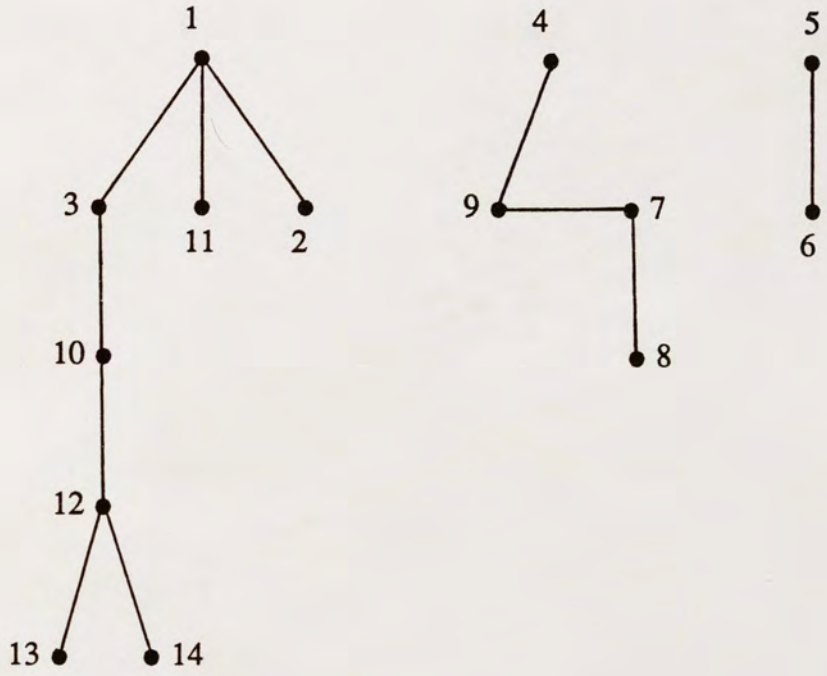

Figure 3.6. A Spanning Forest of the Graph in Figure 3.2.

3.4.2 Time Complexity

The time required by the best-known sequential algorithm (Tarjan 1972) for finding a spanning forest of a graph with n vertices and m edges is $T_1^{FOR} = O(m + n)$. Since a forest has no more than $n - 1$ edges, one iteration of the algorithm MERGE_FOREST requires at most $K_2\ (3n - 2)$ time, where K_2 is a positive constant. There are $\log p$ merging iterations. Each processor has the count of the number of edges included in the forest handled by it, and "whether any forest contains $n - 1$ edges" can be checked in at most $\log p$ time. If the merging terminates before $\log p$ iterations, then copying of the appropriate forest-edges into Q_1 requires

$n - 1$ time in the worst case. Therefore, the total merging time to compute a spanning forest is given by, $M^{FOR} \leq [\, K_2\,(3n - 2) + \log p \,] \log p$. Initialization of ROOT array and its final computation require, respectively, $\left\lceil \frac{n}{p} \right\rceil$ and no more than $K_2\,(2n - 1)$ times. Thus the time complexity of the algorithm PARALLEL_FOREST is

$$T_p^{FOR} \leq K_2\,[\,(\left\lceil \frac{m}{p} \right\rceil + n) + (2n - 1)\,] + [\, K_2\,(3n - 2) + \log p \,] \log p + \left\lceil \frac{n}{p} \right\rceil .$$

The asymptotic performance and the $p\,(T_p)^2$ complexity are the same as those obtained for the algorithm PARALLEL_CONNECT. The algorithm PARALLEL_FOREST also achieves optimal speedup for $1 \leq p \leq \frac{m/n}{\log\,(m/n)}$ processors.

3.4.3 Remarks

(1) The algorithm PARALLEL_FOREST can be used directly to find the connected components of a graph. The number of distinct entries in array ROOT[1 . . n] gives the number of connected components, and ROOT[j] is the component number of the vertex v_j, for $1 \leq j \leq n$.

(2) The algorithm PARALLEL_CONNECT or PARALLEL_FOREST can be applied to design optimal parallel algorithm for computing the transitive closure G^*

of an undirected graph G. In G^* the edge (u, v), where $u < v$, exists if and only if u and v belong to the same component of G.

(3) As suggested by Chin, Lam, and Chen (1982), the algorithm for finding the weakly connected components of a directed graph can be obtained by first ignoring the edge orientations, removing the duplicate edges, and then applying either of the preceding algorithms.

3.5 Discussion

The divide-and-conquer-based parallel algorithms for connected components and spanning forest presented in this chapter are not parallelization of existing sequential algorithms. Since stacks and queues are very sequential in nature as data structures, conflict-free access to their elements by different processors incurs overhead; so direct parallelization of depth-first or breadth-first search techniques to solve these problems is not attractive. From that point of view, our strategy has significance. In the underlying divide-and-conquer strategy, the input graph is partitioned almost equally among processors; each processor operates sequentially on its subproblem, and then subsolutions are merged iteratively to obtain the final solution. Many parallel sorting and selection algorithms use a similar approach. We have also applied this technique to find minimum spanning forest on a weighted graph (Das, Deo, and Prasad 1988a). We believe that this approach will lead to optimal parallel algorithms for other problems as well. For such an algorithm, the time complexity of merging

— which involves critical computation and communication in a shared memory model without concurrent access facility — will be the dominating factor in the overall performance. We choose optimal grain-size to make a proper trade-off between the computation and communication time complexities. The optimal number of processors and hence the optimal granularity are functions of the number of vertices and edges in the graph. To derive the optimality condition, the processor-(time)^2 product has been minimized with respect to the number of processors.

Another novelty of our algorithms is the use of simple data structure, namely an unordered list of edges, in contrast to adjacency matrix or adjacency list. Many existing parallel algorithms are optimal for dense graphs but are unacceptably inefficient for sparse graphs. Our algorithms are equally efficient for dense as well as sparse graphs, and are optimally adaptive within the derived range of processors. The working data structures are also simple, and ensure that the processors do not conflict in reading from or writing into a memory cell. The total required space is optimal by the choice of the number of processors to be used.

Although the algorithms presented in this chapter have been designed for shared memory computers, the use of simple merging algorithms and large grain-size promise their efficient implementation (with less communication, restricted only to neighboring processors) on fixed connection computers as well, such as a hypercube. For details, see Das, Deo, and Prasad (1988b).

CHAPTER 4

FOREST-BASED GRAPH ALGORITHMS

This chapter is devoted to designing three efficient parallel graph algorithms based on spanning forest or, in particular, spanning tree. The problems include computing a fundamental cycle set and the bridges of a connected graph and determining the bipartiteness of a graph. Cycles of a graph give information as to how well the graph is connected. The set of cycles remains invariant under isomorphism of graphs. In certain applications, e.g., the program analysis and evaluation, the theory of data structures, etc., it is advantageous to have a list of all the cycles of a graph. A fundamental set of cycles forms a basis for the cycle space of a graph. Therefore, finding a fundamental cycle set is an important graph connectivity problem. The removal of a bridge increases the number of connected components of a graph by one. Thus the problem of locating bridges is important in order to ascertain the connectedness of a graph.

The algorithms PARALLEL_FOREST and PARALLEL_CONNECT (described in Chapter 3) are used as subalgorithms to efficiently solve the above three problems. Each of our proposed algorithms achieves an optimal speedup using an appropriate number of processors, which is different for different problems and is shown to be dependent on the density of the input graph.

Section 4.2 discusses the previous works; the necessary definitions are provided

in Section 4.1. Sections 4.3, 4.4, and 4.5, respectively, describe the algorithms for computing fundamental cycle set, finding bridges, and determining bipartiteness of a graph. These sections also analyze the performance of the corresponding algorithms, and derive a lower bound on the processor-(time)^2 product for each of them. Finally, Section 4.6 summarizes the result.

4.1 Definitions

A *co-tree* $CT = (V_G, E_{CT})$ of a connected graph $G = (V_G, E_G)$ with respect to a spanning tree $T = (V_G, E_T)$ is the subgraph with the edge-set $E_{CT} = E_G - E_T$ of $m - (n - 1)$ edges, where n and m are, respectively, the number of vertices and edges in the graph G. Any edge e_i of a co-tree is called a *chord* of the spanning tree T. Adding a co-tree edge e_i to T creates a *fundamental cycle,* FC_i. The collection of all $m - n + 1$ cycles with respect to T is called a *fundamental cycle set* (FCS). The importance of an FCS is that any arbitrary cycle in the graph can be expressed as a linear combination of the fundamental cycles by the *symmetric difference* operation, denoted by $\oplus$, where $FC_i \oplus FC_j = \{e \mid e \in FC_i \cup FC_j, e \notin FC_i \cap FC_j\}$. An edge $e \in E_G$ of a connected graph G is called a *bridge* if the graph $G_e = G - \{e\} = (V_G, E_G - \{e\})$ is disconnected. A graph G is *bipartite* if its vertex-set V_G can be partitioned into two subsets V_1 and V_2 such that every edge in E_G has one end-vertex in V_1 and the other one in V_2. As in Chapter 3, we use an array LIST (of size $m \times 2$) of unor-

dered edges as the data structure for the input graph.

4.2 Previous Works

Table 4.1 reviews the literature on fast and efficient parallel algorithms for the aforementioned three problems, employing the PRAM models of computation. As can be observed from Table 4.1, many of the earlier results are based on the assumption of unbounded parallelism. Moreover, most of these algorithms are optimal or near-optimal only for dense graphs. For example, optimal speedups are achieved for bridge-finding algorithms due to Tarjan and Vishkin (1985), and Tsin and Chin (1984) using $p \leq \frac{n^2}{\log^2 n}$ processors. Using adjacency list as the data structure, Koubek and Krsnakova (1985) designed a near-optimal algorithm for bridges on EREW model which utilizes $O(\frac{m+n}{\log n})$ processors and $O(m+n)$ space.

Let us highlight the performances of the parallel algorithms we design in this chapter. The fundamental-cycle-set algorithm attains an optimal speedup using $p \leq \frac{\sqrt{mn}}{\log \sqrt{mn}}$ processors for graphs of any density and using $p \leq \sqrt{mn}$ when $m = n^{1+\varepsilon}$ for $0 < \varepsilon \leq 1$. A modified version of this algorithm is optimally adaptive for $p \leq \frac{m}{\log m}$ processors for all graphs. The bridge-finding algorithm requires $p \leq \frac{n}{\log n}$ processors for optimality and is efficient for dense graphs only. The parallel algorithm for bipartiteness-checking is optimal for $p \leq \frac{m/n}{\log (m/n)}$ processors

on graphs of varying density. The implementation of our algorithms require $O(pn + m)$ space, which is optimal by our choice of p.

On various fixed connection models, several parallel algorithms have been reported for the problems under consideration. Doshi and Varman (1987) have described an optimal algorithm for finding bridges on a fixed-size linear array. Yeh (1986) has designed optimal algorithms for fundamental cycles and bridges on a tree-structured computer. Atallah and Kosaraju (1984) have presented algorithms for the fundamental cycles, the bridges, and for checking bipartiteness for mesh-connected computers. Miller and Stout (1987a, 1987b) have developed algorithms for bridges, and bipartiteness-checking for Pyramid and hypercube machines. For an overview of the time and processor complexities of these algorithms, refer to Das, Deo, and Prasad (1988b), who have designed optimal algorithms for all these problems on hypercube computers.

TABLE 4.1. FOREST-BASED PARALLEL GRAPH ALGORITHMS ON PRAM †

PROBLEM	MODEL	TIME (T_p)	PROCESSORS	RESEARCHERS
1. Fundamental Cycle Set	CREW	$O(\log^2 n)$	$O(n^3)$	Savage & Ja' Ja' (1981)
		$O(\frac{m}{nK}\log n + \frac{n}{K} + \log^2 n)$	$nK,\ K \geq 1$	Tsin & Chin (1984)
		$O(\log^2 n)$	$O(n(m-n+1))$	Ghosh (1986)
		$O(\log n \log d)$	$O(n^3)$	Ghosh (1986)
	EREW	$O(\frac{mn}{p} + p + n \log p)$	p	Das (*this dissertation*)
		$O(\frac{mn}{p} + n \log p)$	p	Das (*this dissertation*)
2. Bridges	CRCW	$O(\log n)$	$O(m+n)$	Tarjan & Vishkin (1985)
	CREW	$O(\log^2 n)$	$O(n^2 \log n)$	Savage & Ja' Ja' (1981)
		$O(\frac{n}{K} + \log^2 n)$	$nK,\ K \geq 1$	Tsin & Chin (1984)
		$O(\frac{n^2}{p})$	p	Tarjan & Vishkin (1985)
		$O(\log^2 n)$	$O(n(m-n+1))$	Ghosh (1986)
		$O(\log n \log d)$	$O(n^3)$	Ghosh (1986)
	EREW	$O(\log^2 n)$	$O(\frac{m+n}{\log n})$	Koubek et al. (1985)
		$O(\frac{m}{p} + \frac{n^2}{p} + n \log p)$	p	Das (*this dissertation*)
3. Bipartite	EREW	$O(\frac{m}{p} + n \log p)$	p	Das (*this dissertation*)

† d is diameter of graph; $0 < \varepsilon \leq 1$; $\alpha(m, n)$ is an inverse Ackermann's function.

4.3 Computing Fundamental Cycle Set

For this section, the given graph G is assumed to be connected. (Note that, if necessary, we can always check for connectedness of the graph with the help of the algorithm PARALLEL_CONNECT.) The best-known sequential algorithm, based on either breadth-first or depth-first search, finds a fundamental cycle set (FCS) of a connected graph in $T_1^{FCS} = O(mn)$ time (Reingold, Nievergelt, and Deo 1977). In the proposed algorithm PARALLEL_FCS, a spanning tree T is first computed. Then a co-tree is identified in parallel with the help of a boolean array MARK of size m, each bit of which is assigned to an edge. Initially, MARK[i] := 0, for $1 \le i \le m$. Note that a row-index of the array LIST gives the label of an edge of the input graph. After the execution of the algorithm PARALLEL_FOREST, the labels of all edges in the spanning tree T are stored in the queue Q_1; and for these tree-edges the corresponding MARK bits are 1's. The unmarked edges are those of the co-tree CT. Each of the co-tree edges forms a fundamental cycle when added to a subset of edges in the corresponding spanning tree. Each processor scans its share of edges and, if a particular edge belongs to the co-tree, it finds the associated fundamental cycle. Additional storage is required by the following algorithm to store the fundamental cycles.

procedure PARALLEL_FCS;
  begin
    PARALLEL_FOREST ; (* find a spanning tree T *)
    for all i, $1 \le i \le p$, **do** (* construct the co-tree CT of G *)
      parbegin
        for each j, $(i-1)\left\lceil \frac{m}{p} \right\rceil + 1 \le j \le i\left\lceil \frac{m}{p} \right\rceil$, **do**

```
            begin
                MARK[ j ] := 0;
            end;
        for each j, (i − 1)⌈(n − 1)/p⌉ + 1 ≤ j ≤ i⌈(n − 1)/p⌉, do
            begin
                MARK[ Q1 [ j ] ] := 1;
            end;
    parend;
for all e_i = (v_i', v_i'') ∈ E_G do
    parbegin
        if MARK[i] = 0 then        (* execute only for co-tree edges *)
            begin
                find the path PATH_i from v_i' to v_i'' in T;
                FC_i := PATH_i ∪ {e_i};    (* i^th fundamental cycle *)
            end;
    parend;
end.
```

4.3.1 Complexity Analysis

The construction of the co-tree CT requires $\theta\left(\frac{m+n}{p}\right)$ time. In the foregoing algorithm, the second **parbegin . . . parend** loop is executed for only those edges which are in CT. Each processor works on the queue Q_1 and constructs for itself a linked adjacency list of the edges in the spanning tree T. The concurrent reading of the queue is avoided by pipelining the access to the edges by different processors. For a connected graph, a spanning tree has $n - 1$ edges; hence the adjacency lists can be constructed in $O[(n-1)+(p-1)+n]$ time. Next each processor is in

charge of $\left\lceil \frac{m}{p} \right\rceil$ edges of LIST, and each sequentially finds the path $PATH_i$ in T corresponding to a co-tree edge $e_i = (v_{i'}, v_{i''})$ as indicated below. The vertex $v_{i'}$ is labeled 1. Then starting at $v_{i'}$ and using a breadth-first search, each vertex u is labeled $L + 1$ where L is the label of the predecessor of u. We stop when $v_{i''}$ acquires a label. Clearly, the labeling of the vertices requires $O(n)$ time. The required path $PATH_i$ is traced by starting at vertex $v_{i''}$ and proceeding backwards so that the next vertex visited has a label one less than that of the current vertex. While tracing the path, the total time spent on scanning edges at individual vertices is $O(n)$ because each edge, except the initial and final ones, is scanned at most twice. The length of the path is $n - 1$ in the worst case, and so the entire path-finding procedure requires no more than $O(n)$ time. Therefore each processor spends at most $O(\frac{mn}{p})$ time to find the cycles corresponding to all edges assigned to it. The asymptotic time complexity of the algorithm PARALLEL_FCS is

$$
\begin{aligned}
T_p^{FCS} &\leq T_p^{FOR} + O(\frac{m+n}{p}) + O(n+p) + O(\frac{mn}{p}) \\
&= O(\frac{m}{p} + n) + O(n \log p) + O(\frac{m+n}{p}) + O(n+p) + O(\frac{mn}{p}) \\
&\approx O(\frac{mn}{p}) + O(n \log p) + O(p) \qquad (4.1)
\end{aligned}
$$

The speedup is given by,

$$
S_p^{FCS} = \frac{O(mn)}{O(\frac{mn}{p}) + O(n \log p) + O(p)} = \frac{O(p)}{O(1) + O(\frac{p \log p}{m}) + O(\frac{p^2}{mn})}.
$$

Now, for optimal speedup, $p \log p \leq m$ and $p \leq \sqrt{mn}$, both of which are satisfied for any graph by a choice of $p \leq \dfrac{\sqrt{mn}}{\log \sqrt{mn}}$.

Theorem 4.1: If a fundamental cycle set of a graph of n vertices and m edges is computed using p processors in time T_p by the algorithm PARALLEL_FCS, then $p(T_p)^2 \geq \theta\left((\sqrt{mn})^3 \log \sqrt{mn} \right)$, and the lower bound is achieved when $p = \theta\left(\dfrac{\sqrt{mn}}{\log \sqrt{mn}} \right)$.

Proof: Ignoring the constants in the asymptotic time complexity, the lower bound on T_p can be written as $T_p \geq \dfrac{mn}{p} + n \log p + p$

or, $p(T_p)^2 \geq \dfrac{m^2n^2}{p} + 2mn^2 \log p + pn^2 \log^2 p + p^3 + 2mnp + 2p^2n \log p$.

Considering three cases corresponding to whether p is less than, equal to, or greater than $\theta\left(\dfrac{\sqrt{mn}}{\log \sqrt{mn}} \right)$, the proof follows. □

Theorem 4.2: If $p = \sqrt{mn}$, then $p \log p \leq m$ is asymptotically satisfied for graphs with large number n of vertices and $m = n^{1+\varepsilon}$ edges, for $0 < \varepsilon \leq 1$.

Proof: Let $p = \sqrt{mn}$. Then $p \log p \leq m$ yields

$$\log \sqrt{mn} \leq \sqrt{\frac{m}{n}} \tag{4.2}$$

Consider a graph having $m = n^{1+\varepsilon}$ edges, for $0 < \varepsilon \leq 1$. Then $\sqrt{\dfrac{m}{n}} = n^{\frac{\varepsilon}{2}}$ and

$\sqrt{mn} = n^{1+\frac{\varepsilon}{2}}$. From Inequality (4.2), we get

$$(1 + \frac{\varepsilon}{2}) \log n \leq n^{\frac{\varepsilon}{2}} \tag{4.3}$$

Now, $\lim_{n \to \infty} \frac{n^{\frac{\varepsilon}{2}}}{\log n} = \lim_{n \to \infty} \frac{\frac{\varepsilon}{2} n^{\frac{\varepsilon}{2}-1}}{\frac{1}{n}}$, by L'Hospital's Rule.

$$= \lim_{n \to \infty} (\frac{\varepsilon}{2} n^{\frac{\varepsilon}{2}}) \to \infty, \quad \text{for } \varepsilon > 0.$$

It implies that Inequality (4.3) is satisfied for $0 < \varepsilon \leq 1$, since for a graph without multiple edges between two vertices, ε cannot be greater than 1 for the chosen value of m. Hence the proof. □

4.3.2 A Modified Implementation

The parallel time in Expression (4.1) as well as the performance of the algorithm PARALLEL_FCS according to Theorems 4.1 and 4.2 are based on the linear pipelined access of the edges in the queue for the construction of a co-tree. As we have seen, by this mechanism, adjacency lists corresponding to a spanning tree are available at all processors in $O(n + p)$ time. However, a simple but elegant improvement is as follows. By using the BROADCAST subroutine (Section 2.3), the spanning-tree-edges from the queue can be accessed by all processors in a binary tree-like pipelined fashion. Consequently, the construction of adjacency lists require a

total of $O[(n-2)+\log p+n]$ time. Thus, the overall asymptotic time complexity of the algorithm for computing fundamental cycles is improved to

$$T_p = O(\frac{mn}{p}) + O(n \log p) \tag{4.4}$$

The modified speedup is $S_p = \dfrac{O(p)}{O(1) + O(\frac{p \log p}{m})}$.

Therefore, the asymptotic optimal number of processors is given by the inequality $p \log p \le m$ and the corresponding time is $T_p = O(\frac{mn}{p})$.

Theorem 4.3: The modified implementation of the PARALLEL_FCS-algorithm satisfies $p(T_p)^2 \ge \theta(mn^2 \log m)$, and the lower bound is attained for $p = \theta(\frac{m}{\log m})$.

Proof: Starting with $T_p \ge \frac{mn}{p} + n \log p$, when we ignore the multiplicative constants in the order notation, the proof is simple. □

It is to be noted that the modified implementation of the preceding algorithm has better performance because of its lower value of $p(T_p)^2$ and because it is optimally adaptive up to a larger number (namely, $p \le \frac{m}{\log m}$) of processors compared to the earlier version. It is the modified time complexity of the co-tree generation which will be used to analyze the bridge-finding algorithm in the next section.

4.4 Finding Bridges

We assume a connected graph G for this section also. The algorithm PARALLEL_BRIDGE, outlined below, is a parallelization of Corneil's (1971) sequential algorithm to identify the bridges in a connected graph G. It utilizes the fact that a bridge must belong to every spanning tree of G. Corneil's algorithm first forms a spanning tree and the corresponding co-tree of G. Then it collapses into *supervertices* all the vertices of the spanning tree belonging to a particular component of the co-tree. An edge in the resulting graph (which might have parallel edges) is a bridge if and only if it was a bridge in G (Corneil 1971). For illustration, consider a connected subgraph (Figure 4.1) of the graph in Figure 3.1. The edge $e_{13} = (v_{10}, v_{12})$ is a bridge. A spanning tree for this graph and the corresponding co-tree are shown in Figure 4.2. The vertices v_3, v_{10}, and v_{11} are collapsed together in the spanning tree; so are the vertices v_{13} and v_{14}. This results in a graph shown in Figure 4.3, which also has the edge e_{13} as a bridge.

For the implementation of the algorithm PARALLEL_BRIDGE, we use a bit vector IDENTITY of size m, which is initialized to all zeros and stored in the shared memory. When an edge e is detected as a bridge, IDENTITY[e] is set to 1. The algorithm is formally presented in the following.

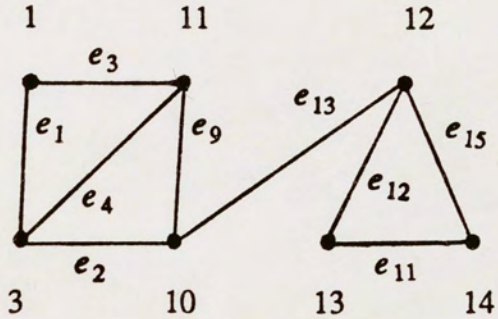

Figure 4.1. A Graph With a Bridge.

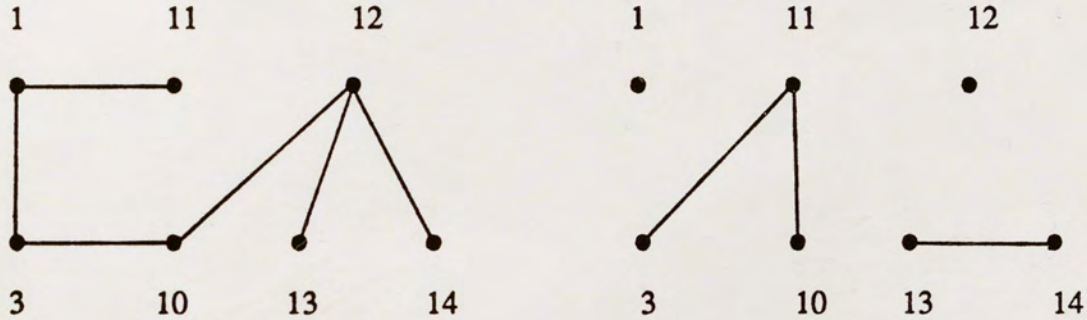

Figure 4.2. (a) A Spanning Tree. (b) The Corresponding Co-tree.

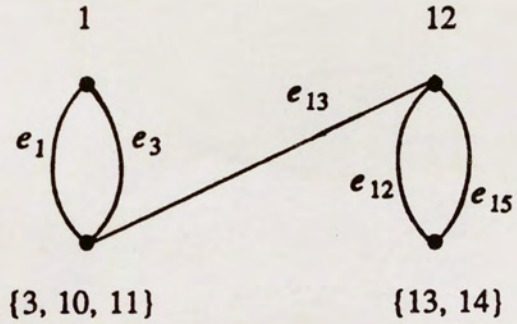

Figure 4.3. The Resulting Graph With Collapsed Vertices.

```
procedure PARALLEL_BRIDGE;
  begin
    PARALLEL_FOREST;        (* find a spanning tree T *)
    construct the co-tree CT of G;
    PARALLEL_CONNECT;       (* find the connected components of CT *)
    construct a new graph H = (V_H, E_H) such that
        V_H = {(i | i is a connected component of CT }
        E_H = {(i, j) | (y, z) is an edge of T, y ∈ component i
                                  and z ∈ component j of CT};
    for all e ∈ E_H do
      parbegin
        if H − {e} is connected then e is not a bridge
          else IDENTITY[e] := 1;
      parend;
  end.
```

4.4.1 Time Complexity

As shown in the (modified) asymptotic analysis of the algorithm PARALLEL_FCS, a spanning tree and its co-tree are formed in $O(\frac{m}{p} + n) + O(n \log p) + O(\frac{m+n}{p}) + O(n + \log p)$ time, using a binary-tree like pipelining. The connected components of CT are computed in $O(\frac{m}{p} + n) + O(n \log p)$ time. The new graph H is constructed sequentially in $O(n)$ time, since the number of components in the co-tree CT is no more than $n - 1$, and $|E_H| = n - 1$. In the **parbegin . . . parend** loop, each processor examines the connectivity of $H - \{e\}$ in $O(n)$ time. Since each processor handles $\left\lceil \frac{n-1}{p} \right\rceil$ edges, the total required time is

$O(\frac{n^2}{p})$. The overall asymptotic time complexity of the algorithm PARALLEL_BRIDGE is given by

$$T_p^{BRI} = O(\frac{m}{p}) + O(n \log p) + O(\frac{n^2}{p}).$$

The best-known sequential algorithm computes the bridges in $T_1^{BRI} = O(m + n)$ time (Tarjan 1974). Therefore, the speedup of the algorithm PARALLEL_BRIDGE is

$$S_p^{BRI} = \frac{O(p)}{O(1) + O(\frac{np \log p}{m}) + O(\frac{n^2}{m})}.$$

For dense graphs, with $m = \theta(n^2)$, an asymptotic optimal number of processors is given by the inequality $p \log p \leq \theta(n)$ which yields $p = \theta(\frac{n}{\log n})$, $T_p = O(\frac{n^2}{p})$, and the speedup is optimal.

Theorem 4.4: For the PARALLEL_BRIDGE-algorithm, $p(T_p)^2 = \Omega(n^3 \log n)$; and the lower bound is achieved when $p = \theta(\frac{n}{\log n})$.

Proof: Starting with $T_p \geq \frac{n^2}{p} + n \log p$, and approaching along the same line as Theorem 3.1, the proof is straightforward. □

4.4.2 Remark

Another possible algorithm for finding the bridges could utilize the property that a bridge must not belong to any fundamental cycle. Thus, if all the edges in a fundamental cycle set are removed from a graph, the remaining edges are the bridges. However, this approach has worse complexity for dense as well as sparse graphs in comparison to the bridge-finding algorithm we have presented.

4.5 Determining Bipartiteness

The algorithm PARALLEL_BIPARTITE determines whether an undirected graph G is bipartite. It first selects a spanning forest F of G, by applying the algorithm PARALLEL_FOREST, whose output is a list of forest-edges along with the array ROOT[1 . . n], which gives the roots of the trees in the forest. Recall that the smallest-indexed vertex in a tree is its root. For each rooted tree, the algorithm computes the depths of its vertices by a breadth-first search. We use an array DEPTH of size n, where DEPTH[i] stores the depth of the vertex v_i from the root of the tree to which it belongs. Designate a vertex to be in the subset V_1 if its depth is even, and in V_2 if its depth is odd. Now, the graph G is bipartite if and only if the depths assigned to the end-vertices of every edge of G is of different parity. In the following algorithm, each processor verifies this condition for a subset of edges assigned to it. (Similar approach has been taken by Miller and Stout (1987a) on the Pyramid machine.) We use a bit vector PARITY of size m, which is initialized to all 1's and

stored in the shared memory. When an edge e has the same parity for the depths of its two end-vertices, then PARITY[e] is set to 0. Thereafter, log n steps of communication are required to collect the final result at processor P_1. Formally, the algorithm is described below.

```
procedure PARALLEL_BIPARTITE;
  begin
     PARALLEL_FOREST;  (* compute a spanning forest F *)
     for all tree  t ∈ F  do
          parbegin
               determine the depth of each vertex in  t  from its root
               and store the depths into the array DEPTH;
          parend;
     for all  e ∈ E_G  do
          parbegin
               if both end-vertices of  e  have either even or odd depths
                    then  PARITY[e] := 0;
          parend;
     if any of the PARITY bits is 0 then G  is not bipartite
               else G  is bipartite;
  end.
```

4.5.1 Time Complexity

Computing array DEPTH requires no more than $O[(n-1)+n]$ time. To avoid having to make concurrent reading of this array in the next stage, all processors copy it in a binary-tree pipelined fashion in $O[(n-1)+\log p]$ time. The generation of the PARITY bit-vector for the edges and checking bipartiteness are performed in

parallel in $O(\frac{m}{p}) + \log p$ time. Therefore, the asymptotic time required by PARALLEL_BIPARTITE is

$$T_p^{BIP} \leq O(\frac{m}{p} + n) + O(n \log p) + O(n) + O(n + \log p) + O(\frac{m}{p}) + \log p$$

$$\approx O(\frac{m}{p}) + O(n \log p).$$

Using depth-first search, a sequential algorithm for checking the bipartiteness of a graph achieves the lower bound of $T_1^{BIP} = O(m + n)$ time (Reingold et al. 1977); therefore the asymptotic speedup of our parallel algorithm is

$$S_p^{BIP} = \frac{O(p)}{O(1) + O(\frac{np \log p}{m})}.$$

For optimality of S_p we derive the condition $p \log p \leq \frac{m}{n}$, which is satisfied by choosing $p \leq \frac{m/n}{\log(m/n)}$. We state the following theorem which can be proved along the same line as Theorem 3.1.

Theorem 4.5: The algorithm PARALLEL_BIPARTITE satisfies $p(T_p)^2 = \Omega(mn \log \frac{m}{n})$.

4.6 Discussion

We have presented efficient parallel algorithms for finding a fundamental cycle set and the bridges of a connected graph and for determining bipartiteness of a graph. The divide-and-conquer-based parallel algorithms for a spanning forest and the connected components presented in Chapter 3 have been used as subroutines to design these algorithms. Except the one for finding bridges (which is efficient for dense graphs only), the algorithms achieve optimal speedups for graphs of varying densities. The modified implementation of the fundamental-cycle-set algorithm has better performance. The optimal number of processors and hence the optimal granularity for each algorithm are found to be functions of the number of vertices and edges in the graph. A simple data structure, namely an unordered list of edges, has been used. The working data structures require optimal space. Although the algorithms presented in this chapter have been designed for shared memory computers, the use of large grain-size promises their efficient implementation (with less communication cost) on fixed connection computers as well, such as a hypercube (Das, Deo, and Prasad 1988b).

CHAPTER 5

THE ASSIGNMENT PROBLEM

The *assignment* problem is an important combinatorial optimization problem, which finds a minimum-cost (or maximum-profit) assignment of n workers to n jobs in a one-to-one fashion, given that assigning a worker W_i to a job J_j is associated with a nonnegative cost (or profit), c_{ij}. In this chapter, we present two parallel algorithms for solving an $n \times n$ assignment problem on an EREW PRAM model. The performance analysis reveals that each of the proposed algorithms achieves optimal speedup for dense graphs, and is optimally scalable up to a certain number of processors. A lower bound on the processor-(time)^2 product for each algorithm is also derived.

In Section 5.1, we present a formulation of the problem. Section 5.2 describes a parallelization of the Hungarian method. This algorithm runs in $O(\frac{n^3}{p} + n^2 \log p)$ time and achieves optimal speedup using $1 \le p \le \frac{n}{\log n}$ processors. The second algorithm, based on a variation of dynamic programming strategy, has been sketched in Section 5.3. This algorithm is designed by finding a min-cost flow in an appropriate network in $O(\frac{n^3}{p} + pn)$ time, and is optimal for $1 \le p \le n$. Section 5.4 concludes the chapter.

5.1 Background

In order to build a mathematical model, for $1 \le i \le n$ and $1 \le j \le n$, define the variables

$$x_{ij} = \begin{cases} 1, & \text{if } W_i \textit{ is assigned } J_j \\ 0, & \textit{otherwise} \end{cases}$$

Formally, the minimum-cost assignment problem is defined as:

$$\text{Minimize} \quad \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij}\, x_{ij} \quad \text{subject to}$$

$$\sum_{j=1}^{n} x_{ij} = 1, \text{ for } 1 \le i \le n, \quad \text{and} \quad \sum_{i=1}^{n} x_{ij} = 1, \text{ for } 1 \le j \le n.$$

This problem is also known as the *minimum-weight matching* in a complete bipartite graph of $2n$ vertices, where each of the two subsets of the vertex-set contains n vertices. (A *matching* in a graph $G = (V_G, E_G)$ is a subset of edges, no two of which have a common end-vertex. Given nonnegative weights to the edges in E_G, the minimum-weight matching problem is to find a matching that minimizes the sum of the weights on matched edges.)

The assignment problem can be solved by the Hungarian method (Papadimitriou and Steiglitz 1982) or by finding a min-cost flow in an appropriate network (Lawler 1976) — each requiring $O(n^3)$ time with cost matrix, $CM = [c_{ij}]_{n \times n}$, as the data structure. The best-known sequential implementation of an algorithm in the second

class uses Fibonacci heap as the data structure, and has time complexity $O(mn + n^2 \log n)$ for a $2n$-vertex bipartite graph of m edges (Fredman and Tarjan 1985). Though sequentially faster on sparse graphs, this algorithm does not appear to be easily parallelizable. We observe that the Hungarian and the min-cost flow algorithms, with cost matrix as input stored in the shared memory, have potential parallelism and are efficient for dense graphs. In comparison with many other polynomial-time-solvable graph problems (Das and Deo 1988; Quinn and Deo 1984), virtually no attempt has been made to design deterministic parallel algorithms for the assignment problem. Fast parallel, randomized algorithms for a special case of this problem are reported by Galil (1986). Only recently, Pawagi (1987) has developed a divide-and-conquer-based algorithm to compute a maximum-weight matching in an n-vertex tree (represented by the adjacency list of its vertices), requiring $O(\log^2 n)$ time with $O(n)$ processors on a CREW PRAM model.

5.2 Parallel Hungarian Algorithm

A solution to the assignment problem is to select n elements in the cost matrix CM such that there is exactly one element in each row, and exactly one element in each column. A set of zeros satisfying these two requirements must yield an optimal solution because all costs are nonnegative. The Hungarian method appropriately transforms the cost matrix to produce a desired set of zeros without altering the set of optimal solutions to the original problem.

```
procedure PARALLEL_HUNGARIAN;
 (* Input:  An n × n matrix CM = [c_ij] of nonnegative integers *)
 (* Output: An optimal assignment in an array MATCH *)
begin
Step0: make a copy CC of the cost matrix CM;
Step1: for all i, 1 ≤ i ≤ p, do
       parbegin
           (* for each row in CM subtract the smallest entry in the row from every
           entry in it *)
         for each r, (i − 1)⌈n/p⌉ + 1 ≤ r ≤ i⌈n/p⌉, do
           begin
             ZROW[r] := 0;  CROW[r] := 0;
             for each j, 1 ≤ j ≤ n, do
               find jm := j such that CM[r, j] is minimum;
             for each j, 1 ≤ j ≤ n, do
               begin
                 CM[r, j] := CM[r, j] − CM[r, jm];
                 if CM[r, j] = 0 then ZROW[r] := ZROW[r] + 1;
               end
           end;
       parend;
Step2: for all j, 1 ≤ j ≤ p, do
       parbegin
           (* for each column with all positive entries subtract the smallest entry in
           the column from every entry in it *)
         for each s, ( j − 1)⌈n/p⌉ + 1 ≤ s ≤ j⌈n/p⌉, do
           begin
             ZCOL[s] := 0;  RCOL[s] := 0;
             for each i, 1 ≤ i ≤ n, do
               find im := i such that CM[i, s] is minimum;
```

```
        if CM[im, s] > 0 then
          for each i, 1 ≤ i ≤ n, do
            CM[i, s] := CM[i, s] − CM[im, s];
        for each i, 1 ≤ i ≤ n, do
            if CM[i, s] = 0 then ZCOL[s] := ZCOL[s] + 1;
      end;
    parend;
Step3: for all i, 1 ≤ i ≤ n, do MATCH[i] := 0;   N := 0;
    while there exist active rows and/or columns do
    (* repeat until each zero has at least one vertical/horizontal line through it *)
      begin
        for all r, 1 ≤ r ≤ n, do
          find rm := r such that r is an active row and ZROW[r] is minimum;
        for all s, 1 ≤ s ≤ n, do
          find sm := s such that s is an active column and ZCOL[s] is
           minimum;
        if ZROW[rm] ≤ ZCOL[sm] then
          (* row rm has the least number of zeros *)
          begin
           (* the first active column containing a zero in row rm is made inac-
           tive, and a vertical line is drawn through it *)
            for all j, 1 ≤ j ≤ n, do
                CROW[rm] := smallest active column-index j such that
                CM[rm, j] = 0;
            N := N + 1;  MATCH[rm] := CROW[rm];
            ZCOL[CROW[rm]] := 0;  RCOL[CROW[rm]] := 0;
            for all r, 1 ≤ r ≤ n, do
                if (ZROW[r] > 0) and ( CM[r, CROW[rm]] = 0) then
                 ZROW[r] := ZROW[r] − 1;  ZROW[rm] := 0;
          end
        else  (* column sm has the least number of zeros *)
          begin
           (* the first active row containing a zero in column sm is made
```

```
        inactive, and a horizontal line is drawn through it *)
          for all i, 1 ≤ i ≤ n, do
              RCOL[sm] := smallest active row-index i such that
              CM[i, sm] = 0;
          N := N + 1;  MATCH[RCOL[sm]] := sm;
          ZROW[RCOL[sm]] := 0;  CROW[RCOL[sm]] := 0;
          for all s, 1 ≤ s ≤ n, do
            if (ZCOL[s] > 0) and (CM[RCOL[sm], s] = 0) then
                ZCOL[s] := ZCOL[s] − 1;  ZCOL[sm] := 0;
        end;
    end;   (* while *)
Step4: if N = n then   (* a feasible solution of all zero entries is found *)
        the array MATCH contains an optimal assignment, compute optimal cost
        from the matrix CC, and exit
      else   (* N < n, an optimal set of zeros not yet found *)
        begin
          for all i, 1 ≤ i ≤ p, do
            parbegin
              for each r, (i − 1)⌈n/p⌉ + 1 ≤ r ≤ i⌈n/p⌉, do
                if CROW[r] > 0 then
                  for each j, 1 ≤ j ≤ n, do
                    begin
                      find jm := j such that RCOL[j] > 0 and CM[r, j] is
                      minimum;
                      ROWMIN[r] := CM[r, jm];
                    end
            parend;
          find the global minimum GLOMIN among those in array ROWMIN;
          (* GLOMIN is the minimum entry with no line through it in the
          transformed cost matrix *)
          for all i and j, 1 ≤ i, j ≤ n, do
            parbegin
```

```
        (* subtract GLOMIN from each entry with no lines through it *)
          if (CROW[i] > 0) and (RCOL[ j] > 0) then
             CM[i, j] := CM[i, j] - GLOMIN;
        (* add GLOMIN to each entry with both horizontal and vertical lines
        through it *)
          if (CROW[i] = 0) and (RCOL[ j] = 0) then
             CM[i, j] := CM[i, j] + GLOMIN;
      parend;
      (* activate all rows and columns *)
    for all r, 1 ≤ r ≤ n, do compute ZROW[r];
    for all s, 1 ≤ s ≤ n, do compute ZCOL[s];
    go to Step3;
  end;
end.
```

5.2.1 An Example

Let us illustrate the algorithm PARALLEL_HUNGARIAN by finding a minimum-cost assignment for matrix CM of order 5×5. Instead of overwriting on CM itself, for simplicity the matrices obtained after different steps of the algorithm will be given different names.

$$CM = \begin{bmatrix} 5 & 7 & 5 & 1 & 6 \\ 3 & 9 & 11 & 12 & 7 \\ 4 & 10 & 2 & 5 & 8 \\ 7 & 12 & 3 & 9 & 8 \\ 3 & 4 & 9 & 1 & 5 \end{bmatrix} \begin{matrix} (1) \\ (3) \\ (2) \\ (3) \\ (1) \end{matrix}$$

In Step1 we subtract the minimum entry in each row (shown in parentheses on the

right) from that row, which produces the matrix CM_1. We count the number of zeros in each row. Applying Step2, only 2nd and 5th columns are found to have all positive entries. Their minimum entries are shown in parentheses at the bottom of CM_1. Subtracting them from corresponding columns generates the matrix CM_2. The number of zeros in each column is also counted.

$$CM_1 = \begin{bmatrix} 4 & 6 & 4 & 0 & 5 \\ 0 & 6 & 8 & 9 & 4 \\ 2 & 8 & 0 & 3 & 6 \\ 4 & 9 & 0 & 6 & 5 \\ 2 & 3 & 8 & 0 & 4 \end{bmatrix} \qquad CM_2 = \begin{bmatrix} 4 & 3 & 4 & 0 & 1 \\ 0 & 3 & 8 & 9 & 0 \\ 2 & 5 & 0 & 3 & 2 \\ 4 & 6 & 0 & 6 & 1 \\ 2 & 0 & 8 & 0 & 0 \end{bmatrix}$$

$$\begin{array}{ccccc} & (3) & & & (4) \end{array}$$

The minimum possible number of lines are drawn in Step3 in order to cross out all the zeros in CM_2. The zeros selected to draw these lines are enclosed in circles. The contents of arrays $ZROW$, $CROW$, $ZCOL$, and $RCOL$ after different iterations in the **while** loop of Step3 (so long as an active row or column exists), are shown in Table 5.1. We see that only four lines are used, which is less than the order (i.e., five) of the matrix. Therefore the **else** part of Step4 is executed — first finding 1 as the minimum uncrossed entry in CM_2, then subtracting 1 from all uncrossed entries and adding it to all doubly crossed entries. Next, all rows and columns are activated by recomputing arrays $ZROW$ and $ZCOL$. The newly-transformed matrix is CM_3.

$$CM_3 = \begin{bmatrix} 3 & 2 & 4 & 0 & 0 \\ 0 & 3 & 9 & 10 & 0 \\ 1 & 4 & 0 & 3 & 1 \\ 3 & 5 & 0 & 2 & 0 \\ 2 & 0 & 9 & 1 & 0 \end{bmatrix}$$

Repeating Step3 we now need exactly five lines to cross out all the zeros. Once again a snapshot of the contents in $ZROW$, $CROW$, $ZCOL$, and $RCOL$ in various iterations is provided in Table 5.2. The zeros within circles in CM_3 correspond to an optimal solution (out of possibly several), and is given by the array $MATCH$.

$MATCH =$

1	2	3	4	5
4	1	3	5	2

This implies $x_{14} = x_{21} = x_{33} = x_{45} = x_{52} = 1$, while all other x_{ij}'s are zeros. Hence a minimum-cost assignment is given by the worker-job pairs: {(1, 4), (2, 1), (3, 3), (4, 5), (5, 2)}, as shown in Figure 5.1; its cost is calculated with the help of the original matrix CM as: 2 + 8 + 1 + 3 + 4 = 18. As expected, the minimum cost is equal to the total amount subtracted from the original matrix in Step1 and Step2, plus the 1 subtracted in Step4.

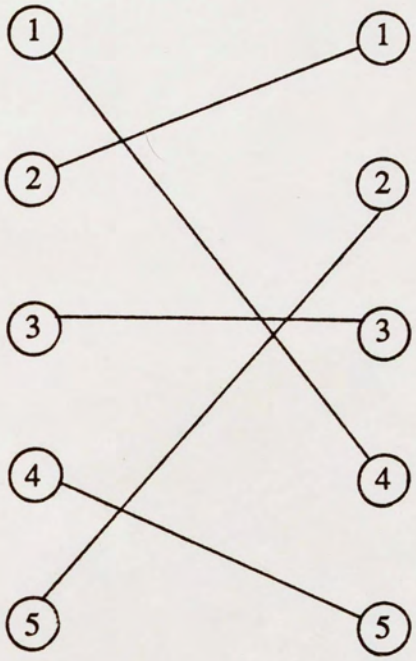

Figure 5.1. An Optimal Assignment for Matrix *CM*.

TABLE 5.1. EXECUTION SNAPSHOTS ON MATRIX CM_2

	(*ZROW*[*r*], *CROW*[*r*])					(*ZCOL*[*s*], *RCOL*[*s*])				
r and *s* →	1	2	3	4	5	1	2	3	4	5
Initially	(1, 4)	(2, 1)	(1, 3)	(1, 3)	(3, 2)	(1, 2)	(1, 5)	(2, 3)	(2, 1)	(2, 2)
After choosing edge (1, 4)	(0, 4)	(2, 1)	(1, 3)	(1, 3)	(2, 2)	(1, 2)	(1, 5)	(2, 3)	(0, 0)	(2, 2)
After choosing edge (3, 3)	(0, 4)	(2, 1)	(0, 3)	(0, 3)	(2, 2)	(1, 2)	(1, 5)	(0, 0)	(0, 0)	(2, 2)
After choosing edge (2, 1)	(0, 4)	(0, 0)	(0, 3)	(0, 3)	(2, 2)	(0, 2)	(1, 5)	(0, 0)	(0, 0)	(1, 5)
After choosing edge (5, 2)	(0, 4)	(0, 0)	(0, 3)	(0, 3)	(0, 0)	(0, 2)	(0, 5)	(0, 0)	(0, 0)	(0, 5)

TABLE 5.2. EXECUTION SNAPSHOTS ON MATRIX CM_3

	(*ZROW*[*r*], *CROW*[*r*])					(*ZCOL*[*s*], *RCOL*[*s*])				
r and *s* →	1	2	3	4	5	1	2	3	4	5
Initially	(2, 4)	(2, 1)	(1, 3)	(2, 3)	(2, 2)	(1, 2)	(1, 5)	(2, 3)	(1, 1)	(4, 1)
After choosing edge (3, 3)	(2, 4)	(2, 1)	(0, 3)	(1, 5)	(2, 2)	(1, 2)	(1, 5)	(0, 0)	(1, 1)	(4, 1)
After choosing edge (4, 5)	(1, 4)	(1, 1)	(0, 3)	(0, 5)	(1, 2)	(1, 2)	(1, 5)	(0, 0)	(1, 1)	(0, 0)
After choosing edge (1, 4)	(0, 4)	(1, 1)	(0, 3)	(0, 5)	(1, 2)	(1, 2)	(1, 5)	(0, 0)	(0, 0)	(0, 0)
After choosing edge (2, 1)	(0, 4)	(0, 1)	(0, 3)	(0, 5)	(1, 2)	(0, 0)	(1, 5)	(0, 0)	(0, 0)	(0, 0)
After choosing edge (5, 2)	(0, 4)	(0, 1)	(0, 3)	(0, 5)	(0, 2)	(0, 0)	(0, 0)	(0, 0)	(0, 0)	(0, 0)

5.2.2 Complexity Analysis

We analyze the total parallel time T_p required by the preceding algorithm, applying the known result that the minimum of n elements can be found in $\left\lceil \frac{n}{p} \right\rceil + \log p$ time using p processors on the EREW PRAM model (Section 2.4.1). Step0 and Step1, respectively, can be performed in $n\left\lceil \frac{n}{p} \right\rceil$ and $(3n + 2)\left\lceil \frac{n}{p} \right\rceil$ time. Time taken by Step2 is less than or equal to the time required by Step1. The **if . . . then** or the **else** clause in the **while** loop of Step3 can be executed in at most $(2\left\lceil \frac{n}{p} \right\rceil + \log p + 4)$ time. Since the **while** loop has no more than n iterations, Step3 requires $n(5\left\lceil \frac{n}{p} \right\rceil + 3 \log p + 5)$ time in the worst case. Step4 takes $(6n\left\lceil \frac{n}{p} \right\rceil + \left\lceil \frac{n}{p} \right\rceil + \log p)$ time in the worst case. It is easy to see that the number of iterations involving Step3 and Step4 is at most n. Therefore, the total computation time, with p processors, is

$$T_p \leq 11n^2\left\lceil \frac{n}{p} \right\rceil + 8n\left\lceil \frac{n}{p} \right\rceil + 3n^2 \log p + 5n^2 + n \log p + 4\left\lceil \frac{n}{p} \right\rceil.$$

For large n, $\left\lceil \frac{n}{p} \right\rceil \approx \frac{n}{p}$ and, therefore, in the worst case,

$$T_p \approx \frac{11n^3}{p} + \frac{8n^2}{p} + 3n^2 \log p + 5n^2 + n \log p + \frac{4n}{p}.$$

Thus, the following theorem.

Theorem 5.1: An $n \times n$ assignment problem can be solved in $O(\frac{n^3}{p} + n^2 \log p)$ time, with p processors on an EREW PRAM model, for $1 \le p \le n$. □

In the following we compute a lower bound on $p(T_p)^2$ complexity and the optimal number of processors to be used. The measure $p(T_p)^2$ achieves a minimum when $\frac{\partial(p(T_p)^2)}{\partial p} = 0$, which, in our case, yields $T_p + 2p \frac{\partial T_p}{\partial p} = 0$. Substituting the values for T_p and $\frac{\partial T_p}{\partial p}$ we get, after some simplification,

$$p \log p \; [3 + \frac{1}{n} + \frac{11}{\log p} + \frac{2}{n \log p}] = 11n + 8 + \frac{4}{n}.$$

For large values of n and p, this may be simplified as $p \log p \approx \frac{11n}{3}$, i.e., $p = O(\frac{n}{\log n})$. Therefore the algorithm PARALLEL_HUNGARIAN achieves optimal speedup using p processors, in the range $1 \le p \le \frac{n}{\log n}$.

Theorem 5.2: For the PARALLEL_HUNGARIAN algorithm, $p(T_p)^2 \ge \theta(n^5 \log n)$ and the equality holds when $p = \theta(\frac{n}{\log n})$.

Proof: For large p and n, and ignoring the multiplicative constants, we can write

$$T_p \ge \frac{n^3}{p} + n^2 \log p, \quad \text{and therefore}$$

$$p(T_p)^2 \geq \frac{n^6}{p} + 2n^5 \log p + pn^4 \log^2 p \tag{5.1}$$

We consider the following three cases to complete the proof.

(i) When $p = \theta\left(\frac{n}{\log n}\right)$, we get $p(T_p)^2 = \theta(n^5 \log n)$;

(ii) When $p < \theta\left(\frac{n}{\log n}\right)$, the first term on the right hand side of Expression (5.1) is $\frac{n^6}{p} > \theta(n^5 \log n)$; and

(iii) When $p > \theta\left(\frac{n}{\log n}\right)$, the last term on the right hand side of (5.1) is $pn^4 \log^2 p > \theta(n^5 \log n)$ because $\log^2 p > \left(\log\left(\frac{n}{\log n}\right)\right)^2 \geq \theta(\log^2 n)$.

Hence the claim that $\theta(n^5 \log n)$ is a lower bound on the product $p(T_p)^2$, which is achieved with $\theta\left(\frac{n}{\log n}\right)$ processors. □

5.3 Parallel Min-Cost Flow Algorithm

An $n \times n$ assignment problem can be reduced to a *min-cost flow* problem in a $(2n + 2)$-vertex network as shown in Figure 5.2, and can be solved with exactly n flow augmentations (Lawler 1976). A *flow network* is a directed graph with a source (no edges going into it) and a sink (no edges going out of it); each directed edge has a capacity and a cost per unit flow. The *min-cost flow* problem finds a flow pattern in a given network that minimizes total cost. Syslo, Deo, and Kowalik (1983) may be consulted for more formal definitions.

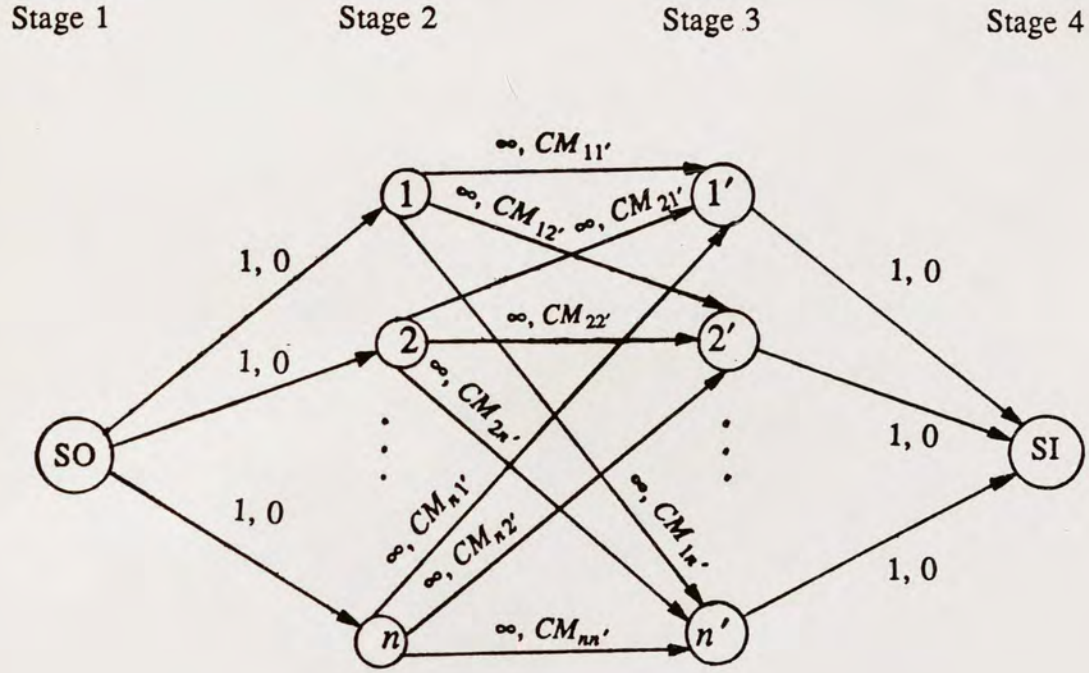

Figure 5.2. Flow Network for the Assignment Problem.

In Figure 5.2, the vertices SO and SI represent, respectively, the source and the sink. The first element on each arc (or directed edge) denotes its capacity and the second, its cost. The job-indices (as depicted in Stage 3 of Figure 5.2) have been primed in order to distinguish from the worker-indices in Stage 2. Each flow augmentation in this network is carried out along a shortest path from the source to the sink, followed by the creation of back edges and modification of the network. For a detailed description of a sequential algorithm (due to Busacker and Gowen) for solving the min-cost flow problem, refer to Syslo et al. (1983). On the EREW PRAM model, a parallel implementation of Dijkstra's shortest path algorithm requires

$O(\frac{n^2}{p} + n \log p)$ time (Paige and Kruskal 1985). This leads to a parallel algorithm for the assignment problem having the same performance as achieved by the PARALLEL_HUNGARIAN algorithm.

Alternately, exploiting the fact that the flow-network and its modifications have arcs only between adjacent stages, we parallelize a variation of a dynamic programming algorithm to find shortest paths. For each vertex (also called *state),* a *return* is defined as its shortest distance from the source. Let $Fq[i]$ be the return of vertex i when the system is at stage q, $1 \leq q \leq 4$. The working data structures for the parallel algorithm consist of a variable $F4[SI]$ which contains the shortest distance from source to sink, and four linear arrays — $F2$, $F3$, DSO, and DSI — each of size n. Arrays $F2$ and $F3$ store the returns of the vertices in Stages 2 and 3, respectively. Initially DSO and DSI contain all zeros. The condition $DSO[i] = 0$ implies that there exists an arc $< SO, i >$ with unit capacity and zero cost, while $DSO[i] = \infty$ means the arc $< SO, i >$ is saturated (i.e., no longer exists) and an arc $< i, SO >$ is created having unit capacity and zero cost. Similarly, $DSI[j'] = \infty$ implies that $< j', SI >$ is saturated and there is an arc $< SI, j' >$ with unit capacity and zero cost, while $DSI[j'] = 0$ indicates the existence of the arc $< j', SI >$. We also use an $n \times n$ matrix D to represent the costs of infinite-capacity arcs between Stages 2 and 3. The $(ij')^{th}$ element of this matrix is a two-tuple: $(D[i, j'], D[j', i])$. Initially, $D[i, j'] := CM[i, j']$, the cost of assigning the worker W_i to the job $J_{j'}$, and $D[j', i] := \infty$ signifying that the arc $< j', i >$ is absent. When an arc $< i, j' >$ is

included in a shortest path, we assign $D[\,j', i\,] := -D[i, j'\,]$ as the cost of the new arc $< j', i >$. Since only one unit of flow is augmented in each iteration, this new arc has unit capacity. On the other hand, if $< j', i >$ is an arc in a shortest path, we assign $D[\,j', i\,] := \infty$. At termination of the min-cost flow algorithm, the arcs $< i, j' >$ with $D[\,j', i\,] < 0$ correspond to an optimal solution to the assignment problem. Assuming that the return of the source $F1[SO] := 0$, the returns of other vertices are computed by the following procedure.

procedure SHORTEST_DISTANCE;
begin
  for all i, $1 \le i \le n$, **do** $F2[i] := DSO[i]$; (* Step 1 *)
  for all j', $1' \le j' \le n'$, **do**
    $F3[\,j'\,] := \min\,\{F2[i] + D[i, j'\,] \mid 1 \le i \le n\}$; (* Step 2 *)
  for all i, $1 \le i \le n$, **do**
    $F2[i] := \min\,\{F2[i], \min\,\{F3[\,j'\,] + D[\,j', i\,] \mid 1' \le j' \le n'\}\}$; (* Step 3 *)
  for all j', $1' \le j' \le n'$, **do**
    $F3[\,j'\,] := \min\,\{F3[\,j'\,], \min\,\{F2[i] + D[i, j'\,] \mid 1 \le i \le n\}\}$; (* Step 4 *)
  $F4[SI] := \min\,\{F3[\,j'\,] + DSI[\,j'\,] \mid 1' \le j' \le n'\}$; (* Step 5 *)
end.

Theorem 5.3: Prior to each flow-augmentation, the algorithm SHORTEST_DISTANCE correctly computes the shortest distance from the source to the sink.

Proof: Let u' be a vertex in Stage 3 satisfying minimum $F3[u'\,]$ for $1' \le u' \le n'$, as

computed by Step 5 of the algorithm, and the arc $< u', SI >$ is in a shortest path. Also, let $CM[\, y, u' \,] \leq CM[i, u' \,]$ for $1 \leq i \leq n$, and $F2[\, y \,] := 0$ where y is a vertex in Stage 2. Then the output of Step 2 is $F3[u' \,] := CM[\, y, u' \,]$.

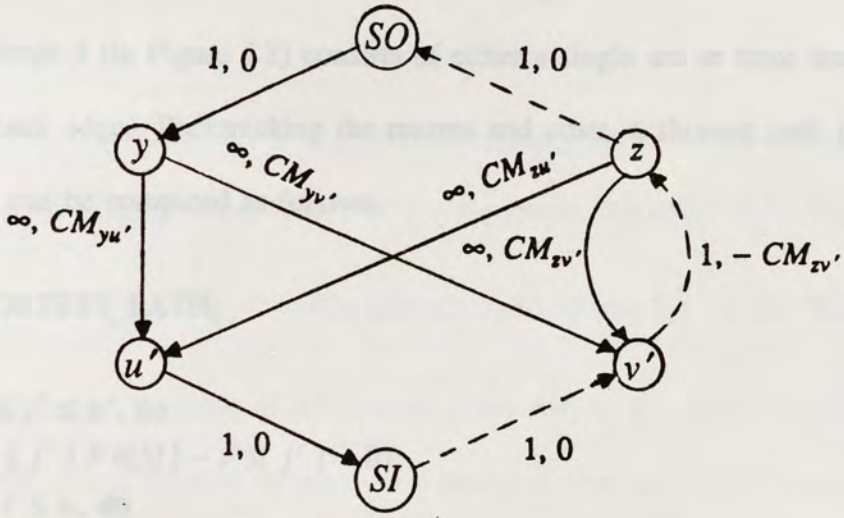

Figure 5.3. Proof of Correctness of the Algorithm SHORTEST_DISTANCE.

Assume the existence of vertices z and v' in Stages 2 and 3, respectively, such that $D[v', z] := -D[z, v'] := -CM[z, v']$ and $F3[v'] := CM[\, y, v' \,]$. Since $< v', z >$ is a back edge with unit capacity, the back edges $< z, SO >$ and $< SI, v' >$ will be present, i.e., $DSO[z] := DSI[v'] := \infty$. This is depicted in Figure 5.3. Now, if $CM[\, y, v' \,] - CM[z, v'] + CM[z, u'] < CM[\, y, u' \,]$ then after the execution of Step 4 we get $F3[u'] := CM[\, y, v' \,] - CM[z, v'] + CM[z, u']$. Therefore,

$$F3[u'] := \min \{CM[y, u'], CM[y, v'] - CM[z, v'] + CM[z, u']\},$$

over all triplets y, v', and z such that $1 \le y,\ z \le n$, $1' \le v' \ne u' \le n'$, and $< v', z >$ is a back edge with $D[v', z] < 0$. □

As a corollary to Theorem 5.3, note that a shortest path from a vertex in Stage 2 to a vertex in Stage 3 (in Figure 5.2) consists of either a single arc or three arcs, one of which is a back edge. Backtracking the returns and costs, a shortest path (out of possibly many) can be computed as follows:

```
procedure SHORTEST_PATH;
begin
  for all j', 1' ≤ j' ≤ n', do
    j1 := min { j' | F4[SI] − F3[ j' ] = 0};
  for all i, 1 ≤ i ≤ n, do
   i1 := min { i | F3[ j1] − F2[i] − D[i, j1] = 0};
  for all j', 1' ≤ j' ≤ n', do
   j2 := min { j' | F2[i1] − F3[ j' ] + D[i1, j' ] = 0};
  if j2 ≠ j1 then
    begin
      for all i, 1 ≤ i ≤ n, do
        i2 := min { i | F3[ j2] − F2[i] − D[i, j2] = 0;
      the shortest path is given by (SO, i2, j2, i1, j1, SI);
      D[ j2, i2] := − D[i2, j2]; D[ j2, i1] := ∞;
      D[ j1, i1] := − D[i1, j1]; DSO[i2] := ∞; DSI[ j1] := ∞;
    end
```

```
else
begin
  the shortest path is given by (SO, i1, j1, SI);
  D[ j1, i1] := - D[i1, j1]; DSO[i1] := ∞; DSI[ j1] := ∞;
end;
end.
```

5.3.1 Time Complexity

The computation progresses stage-by-stage, allocating one processor to each vertex in a stage. Each of Step 1 and Step 5 of the algorithm SHORTEST_DISTANCE performs without memory read- or write-conflicts, and requires $\left\lceil \frac{n}{p} \right\rceil$ time. However, a straightforward implementation of other steps gives rise to the concurrent reading of a memory cell; this is avoided by pipelining the operations of different processors which can be performed in $\left\lceil \frac{n}{p} \right\rceil (n + p - 1)$ time. In the worst case, a shortest path can be found from algorithm SHORTEST_PATH in $4 (\left\lceil \frac{n}{p} \right\rceil + \log p) + 5$ time without any memory conflict. Therefore, the total parallel time T_p required by n flow augmentations is given by,

$$T_p = n \; [\, 3 \left\lceil \frac{n}{p} \right\rceil (n + p + 1) + 4 \log p + 5]$$

$$\leq n \; [\, 3 \, (\frac{n}{p} + 1) \, (n + p + 1) + 4 \log p + 5 \,] \; = \; O(\frac{n^3}{p} + pn) \, .$$

Theorem 5.4: The parallel min-cost flow algorithm corresponding to an assignment problem attains the optimal speedup for $p = O(n)$, and it satisfies $p(T_p)^2 \geq \theta(n^5)$.

Proof: Letting $\frac{\partial(p(T_p)^2)}{\partial p} = 0$ we get, $p\ [\ 6 + \frac{9p}{n} + \frac{16}{n} + \frac{4 \log p}{n}\ ] = 3n + 3$. Hence the optimal speedup is achieved when $p = O(n)$. Following the reasoning used in the proof of Theorem 5.2, we can show that the product $p(T_p)^2$ attains the asymptotic lower bound of $\theta(n^5)$ for $p = \theta(n)$. □

Since finding all ordered pairs $< i, j' >$ with negative $D[\ j', i\ ]$ values requires $n\left\lceil \frac{n}{p} \right\rceil$ time, the parallel algorithm for the assignment problem exhibits the same asymptotic performance as stated in Theorem 5.4.

5.4 Discussion

Two deterministic parallel algorithms have been presented to solve the assignment problem (i.e., to find a minimum-weight matching in a complete bipartite graph). For dense graphs, each algorithm is optimal when employing up to a certain number of processors (which is a function of the problem size). The algorithm, which solves the assignment problem by computing a min-cost flow, is more parallelizable than the Hungarian method, because the former is optimally scalable up to a larger number of processors and has a lower value for the processor-(time)^2 product than the latter.

CHAPTER 6

APPROXIMATE COLORING OF GRAPHS

A variety of problems in production scheduling (Christofides 1975; Matula, Marble, and Isaacson 1972), construction of examination timetables (Syslo et al. 1983; Welsh and Powell 1967), register allocation in compiler code generation and optimization (Chaitan et al. 1981) can be expressed as graph coloring problems. A *vertex coloring* of a graph $G = (V_G, E_G)$ is an assignment of positive integers, the *colors,* to the vertices of G such that no two adjacent vertices are of the same color. Throughout this chapter, "coloring" will always mean vertex coloring. A k*-coloring* of G is a coloring of G with at most k colors. The smallest integer k for which G is k-colorable is called its *chromatic number,* $\chi(G)$. Since the determination of $\chi(G)$-coloring of a graph (even 3-coloring of a planar graph with maximum vertex-degree 4) is an NP-complete problem (Garey and Johnson 1979), various approximate algorithms (sequential) have been proposed to produce k-coloring in polynomial time such that $\chi(G) \leq k \leq n$ (Christofides 1975; Dutton and Brigham 1981; Matula et al. 1972; Syslo et al. 1983). Another effort has been to develop polynomial-time sequential algorithms for coloring planar graphs using fixed number of colors. (Note that a planar graph can be exactly colored using ≤ 4 colors.) Because of the availability of parallel computers, some effort has also gone into designing parallel algorithms.

Though very recent work concentrates on parallel coloring of restricted graphs, relatively little attempt has been made in speeding up approximate algorithms for coloring general graphs by solving them on realistic parallel computers.* In this chapter, using an EREW PRAM model, we parallelize two known approximate graph-coloring algorithms, namely, the largest-degree-first (LF) algorithm originally proposed by Welsh and Powell (1967), and an algorithm (henceforth referred to as the DB algorithm) due to Dutton and Brigham (1981).

For analyzing the performance of parallel approximate coloring algorithms, we compare the execution time of a given parallel algorithm with that of the corresponding sequential algorithm rather than considering the (possibly none) best sequential algorithm. The notion of "best" is not properly defined for approximate algorithms due to the fact that the algorithm which has the faster execution time, may not necessarily require the smaller number of colors and vice versa. Therefore, if AL is a sequential approximate algorithm which solves a problem in time T_{AL}, and PAL is the corresponding parallel algorithm requiring T_{PAL} time, then the *speedup* S_{PAL} of the parallel approximate algorithm PAL using p processors will be defined as the ratio of T_{AL} to T_{PAL}. The definitions of efficiency and optimal speedup remain the same as used in Chapter 2.

* In this context it is worth mentioning a strictly distributed algorithm from Shamir and Upfal (1984), which runs in $O(max(\hat{d}(n), \log n))$ time for random graphs with mean degree $\hat{d}(n)$. The required number of colors is "almost surely" bounded by $\frac{\hat{d}(n)}{\log \hat{d}(n)}$.

Section 6.1 presents the literature survey. Sections 6.2 and 6.3, respectively, describe the implementation and complexity analysis of the parallel LF (PLF) and the parallel DB (PDB) algorithms. We derive bounds for the optimal number of processors, and indicate the class of graphs for which these algorithms attain optimal speedup. The DB algorithm is found to be more easily parallelizable than the LF algorithm. Section 6.4 concludes the chapter.

6.1 Previous Works

Table 6.1 summarizes the salient results on parallel algorithms for vertex coloring. As can be observed, except for the one due to Goldberg (1986), these fast algorithms deal with restricted classes of graphs, namely, planar, embedded planar, constant-degree, and so on. Diks (1986) has presented a parallel algorithm to color outerplanar graphs with minimum possible number (at most 3) of colors. Karloff's (1986) algorithm works only for Brooks graphs, for which maximum vertex-degree $\Delta \geq 3$ and the complete graph on $\Delta + 1$ vertices is not a subgraph. This algorithm runs in poly-logarithmic time for such graphs with $\Delta = O(\log^{O(1)} n)$. Bauernöppel and Jung (1985) have considered a special (λ, μ)-type graphs, for $\lambda, \mu \geq 1$, which include fixed-degree, fixed-genus, planar, and outerplanar graphs. For example, planar graphs are of (4, 1)-type and can be colored with at most eight colors within $O(\log^2 n)$ depth on a uniform Boolean circuit of polynomial size. Goldberg (1986) has designed from scratch a new parallel approximate algorithm which bisects the

graph (by partitioning the vertex-set into two almost equal-sized subsets such that the number of edges cut is minimized), and recursively colors each of the subgraphs using disjoint sets of colors. The algorithm requires poly-logarithmic time using linear number of processors.

TABLE 6.1. PARALLEL ALGORITHMS FOR VERTEX COLORING

CLASS OF GRAPHS	UPPER BOUND ON COLORS	MODEL	TIME	NO. OF PROCESSORS	RESEARCHERS
Outerplanar	3	CRCW	$O(\log n)$	$O(n)$	Diks (1986)
Planar	5	EREW	$O(\log^5 n)$	$O(n^3)$	Naor (1987)
	5	CRCW	$O(\log^3 n)$	polynomial	Boyar & Karloff (1987)
	6	CRCW	$O(\log n)$	$O(n^4)$	Diks (1986)
	7	CRCW	$O(\log \log^* n)$	$O(n)$	Goldberg et al. (1987)
	7	EREW	$O(\log^2 n)$	$O(n)$	Goldberg et al. (1987)
Embedded planar	5	CRCW	$O(\log \log^* n)$	$O(n)$	Goldberg et al. (1987)
	5	CRCW	$O(\log \log^* n)$	$O(n)$	Boyar & Karloff (1987)
Constant-degree (Max degree = Δ)	$\Delta + 1$	EREW	$O(\log^* n)$	$O(n)$	Goldberg & Plotkin (1987)
	$\Delta + 1$	EREW	$O(\Delta \log \Delta (\log^* n + \Delta))$	$O(n)$	Goldberg et al. (1987)
(λ, μ)-type $\lambda, \mu \geq 1$	$2^{\lambda+\mu-2}$	Boolean circuit	$O(\log^2 n)$	$O(n^{O(1)})$	Bauernöppel & Jung (1985)
Max degree = Δ	$\Delta + 1$	EREW	$O(\log^2 n)$	$O(n^2 m \Delta)$	Luby (1986)
	Δ	CRCW	poly-logarithmic	polynomial	Karchmer & Naor (1988)
Brooks' Theorem	$\Delta \geq 3$	EREW	$O(\min(\Delta, \sqrt{n}) \log^{O(1)} n)$	$O(n^2 m \Delta)$	Karloff (1986)
Unrestricted	$\sqrt{2m + \frac{1}{4}} + \frac{1}{2}$	EREW	$O(\log^3 n)$	$O(m + n)$	Goldberg (1986)

6.2 The PLF Algorithm

In the sequential LF algorithm (Matula et al. 1972; Syslo et al. 1983; Welsh and Powell 1967), the vertices are sorted by their degrees in a nonincreasing order (in case of a tie, the vertex with the larger index appears first). Such an ordering is called the largest-degree-first (LF)-ordering. Initially, the vertex with the largest degree is assigned color 1. At each iteration, an uncolored vertex from the sorted list is assigned the smallest possible color number. A linked adjacency list provides an efficient data structure for the sequential algorithm.

In order that all the neighbors of a vertex v_i, $1 \leq i \leq n$, can be accessed simultaneously in the parallel LF (PLF) algorithm, we use an adjacency list matrix first proposed by Eckstein and Alton (1977), consisting of an array VERTEXLIST of size $n \times \Delta$ as the data structure, where $\Delta = \max \{d(i) \mid 1 \leq i \leq n\}$ and $d(i)$ is the degree of vertex v_i. For each vertex v_i, the first $d(i)$ locations of the row VERTEXLIST[i] contain the neighbors (in any order) of v_i and the remaining $\Delta - d(i)$ locations contain 0's. The output is the number k of distinct colors needed to color the graph along with an array COLOR of size n which gives the colors assigned to the vertices. Initially, COLOR[i] = 0 for $1 \leq i \leq n$. As working data structures, the PLF algorithm uses two arrays, DEGREE and SORT, each of size n. Entry DEGREE[i] is the degree of the vertex v_i. The LF-ordering of the vertices is contained in the array SORT. We use another array, NEIGHBOR_COLOR, of size $n \times \Delta$. For each uncolored vertex v_i, the ith row of NEIGHBOR_COLOR contains the information

regarding the colors of its neighbors. Initially, NEIGHBOR_COLOR[i, j] = 0, for $1 \le i \le n$ and $1 \le j \le d(i)$. All these data structures are stored in the global shared memory.

The PLF algorithm works in two phases. In the first phase with VERTEXLIST as the input, the algorithm computes the degree of each vertex; and sorts the vertices according to LF-ordering. In the second phase, colors are assigned to vertices in array SORT. When a vertex u is assigned a color c, we record that information (in parallel) for all of its neighbors. That is, NEIGHBOR_COLOR[v, c] is assigned 1 if $(u, v) \in E_G$. Thus, at any instant, the smallest possible color for vertex v corresponds to the first 0 entry in the vth row of NEIGHBOR_COLOR.

```
procedure PLF;
 begin
  for all i, 1 ≤ i ≤ p, do
    parbegin
      for each j, (i − 1)⌈n/p⌉ + 1 ≤ j ≤ i⌈n/p⌉, do
        begin
          COLOR [ j] := 0;
          DEGREE [ j] := 0;
          for each l, 1 ≤ l ≤ Δ , do
            if VERTEXLIST [ j, l] > 0 then DEGREE [ j] := DEGREE [ j] + 1
                else go to 1;
    1:    for each γ, 1 ≤ γ ≤ DEGREE [ j], do
                        NEIGHBOR_COLOR [ j, γ] := 0;
        end;
    parend;
    sort the vertices according to LF-ordering;
```

```
k := 0;    (* k is an estimate of χ(G) *)
for each i, 1 ≤ i ≤ n do
  begin
    for all j, 1 ≤ j ≤ DEGREE [SORT [i]], do
      parbegin
        find the smallest j, say c, such that NEIGHBOR_COLOR [SORT
        [i], c] = 0. If no such j exists, then c := DEGREE [SORT [i]] + 1;
      parend;
    COLOR [SORT [i]] := c;
    if k < c then k := c;
    for all j, 1 ≤ j ≤ DEGREE [SORT [i]], do
      parbegin
        NEIGHBOR_COLOR [VERTEXLIST [SORT [i], j], c] := 1;
      parend;
  end
end.
```

It is easy to show that the proposed implementation of the PLF algorithm performs correctly without memory read- or write-conflict, and it colors a graph exactly the way the sequential LF algorithm does. Also, the use of an elegant data structure (namely, adjacency list matrix) alleviates the inherent sequential nature of the linked adjacency list.

6.2.1 Complexity Analysis

Using $p \le n$ processors, we calculate the time required by the PLF algorithm. The parallel time T_d to compute the degrees of the vertices is given by

$$T_d = \max_{1 \le i \le p} \left\{ \sum_{j=(i-1)\lceil n/p \rceil+1}^{i\lceil n/p \rceil} d(j) \right\} \le C_1 \left\lceil \frac{n}{p} \right\rceil \Delta ,$$

where C_1 is a positive constant and Δ is the maximum vertex-degree. The total initialization time $T_i \leq \left\lceil \frac{n}{p} \right\rceil (\Delta + 1)$. Since the degree of a vertex is no more than $n - 1$, the LF-ordering is obtained in $T_s = C_2\, n$ time by sequential radix sorting. While coloring the vertex u, we allocate $\left\lceil \frac{d(u)}{p} \right\rceil$ vertices to each processor, which works on its portion of the array NEIGHBOR_COLOR and finds the smallest index j such that NEIGHBOR_COLOR $[u, j] = 0$. If no such j exists, it returns $d(u) + 1$. Then merging takes place in order to find the smallest integer, say c, among the indices obtained by individual processors. (There are $\log p$ merging iterations.) The vertex u is assigned the color c, which can be broadcasted to all processors in $\log p$ time. Now, the assignment of 1 to NEIGHBOR_COLOR $[v, c]$, where $(u, v) \in E_G$, requires $\left\lceil \frac{d(v)}{p} \right\rceil$ time. The total time T_c for assigning colors is given by

$$T_c = C_3 \sum_{u \in V} \left(\left\lceil \frac{d(u)}{p} \right\rceil + \log p \right) + \sum_{v \in V} \left(\left\lceil \frac{d(v)}{p} \right\rceil + \log p \right)$$

$$\leq C_3 \left(\frac{2m}{p} + n + n \log p \right) + \left(\frac{2m}{p} + n + n \log p \right).$$

Therefore, the overall time complexity T_{PLF} of the PLF algorithm is

$$T_{PLF} = T_d + T_i + T_s + T_c$$

$$\leq (C_1 + 1)\left\lceil \frac{n}{p} \right\rceil \Delta + \left\lceil \frac{n}{p} \right\rceil + (C_2 + C_3 + 1)\, n + 2\,(C_3 + 1)\, \frac{m}{p} + (C_3 + 1)\, n \log p$$

$$= C'_1 \frac{n}{p} \Delta + \frac{n}{p} + C'_2\, n + C'_3 \frac{m}{p} + C'_4\, n \log p \;, \quad \text{for large } n.$$

It is easy to show that the sequential LF algorithm requires $T_{LF} = O(m)$ time. Then the speedup of the PLF algorithm is given by

$$S_{PLF} = \frac{T_{LF}}{T_{PLF}} = \frac{O(m)}{C_1' \frac{n}{p} \Delta + \frac{n}{p} + C_2'\, n + C_3' \frac{m}{p} + C_4'\, n \log p} \;.$$

For optimal speedup, i.e., $S_{PLF} = O(p)$ for large n and m, we derive two conditions

$$n\, \Delta \approx O(m) \tag{6.1}$$

$$p \log p \approx O\left(\frac{m}{n}\right) \tag{6.2}$$

Clearly, Condition (6.1) is satisfied by regular or near-regular graphs. For any graph satisfying (6.1), Condition (6.2) yields an optimal granularity. The optimal number of processors is $p \leq O\left(\frac{m/n}{\log (m/n)}\right)$. For a regular graph of degree δ, we get $p \leq O\left(\frac{\delta}{\log \delta}\right)$. However, the PLF algorithm is inefficient for those sparse graphs which have $m = O(n)$ edges and a few but fixed number of vertices, each of degree $O(n)$. For such graphs, computing the degrees of the vertices becomes the bottleneck and Condition (6.1) is not satisfied.

6.3 The PDB Algorithm

The sequential DB algorithm (Dutton and Brigham 1981) aims at creating a complete graph by successively merging nonadjacent vertex-pairs. At the end, the size of the complete graph gives an estimate of the chromatic number. The heuristic selects at each iteration that vertex-pair for merger which has the maximum number of common adjacent vertices. This ensures that the formation of a complete graph requires more iterations, and hence fewer colors, to color the original graph (Williams and Milne 1984). At any instant, 'vertex' v_i represents the original vertex v_i along with those merged 'into' v_i, directly or indirectly, by previous iterations. All vertices merged into the vertex v_i, $1 \le i \le n$, are assigned the ith color.

The adjacency matrix $A = [a_{ij}]_{n \times n}$ of the graph G is stored in the common shared memory. Let $\bar{E}_G = \{(v_i, v_j) \mid \ 1 \le i \le n-1, j > i \text{ and } (v_i, v_j) \notin E_G\}$. Physically, $(v_i, v_j) \in \bar{E}_G$ iff $a_{ij} = 0$. Clearly, $|\bar{E}_G| = \frac{n(n-1)}{2} - m$. For each $(v_i, v_j) \in \bar{E}_G$, define $\bar{V}_{ij} = \{l \mid (v_i, v_l) \text{ and } (v_j, v_l) \in E_G\}$, and $CA_{ij} = |\bar{V}_{ij}|$. Observe that the number CA_{ij} of common adjacent vertices of a nonadjacent vertex-pair (v_i, v_j) is nothing but the number of 1's in the resultant bit vector obtained by ANDing the rows i and j of the adjacency matrix A. Similarly, the merging of vertex v_j into v_i is essentially replacing the row (and also the column) i by the resultant bit vector obtained by ORing the rows i and j and logically deleting row (and column) j in matrix A. The vertex with larger index will be merged into the one with smaller index. The colors assigned to the vertices are available in the array

COLOR of size n. The implementation of the parallel DB (PDB) algorithm uses the following data structures stored in the shared memory.

Index Matrix (INDEX): This is an $n \times n$ boolean matrix. Initially INDEX[i, i] := 1 for $1 \le i \le n$, and INDEX[i, j] := 0 for $1 \le i < j \le n$. When vertex v_j is merged into v_i, $j > i$, then INDEX[i, j] := 1 and INDEX[j, j] := 0. The diagonal entries of the INDEX matrix serve as *status bits* for the vertices. For example, INDEX[j, j] = 0 indicates that the row (and the column) j of the adjacency matrix A is logically deleted. After the termination of the PDB algorithm, the rows i for which INDEX[i, i] = 1 correspond to the vertices of the resulting complete graph; and the columns j such that INDEX[i, j] = 1 correspond to the set vertices merged into the vertex v_i including itself. A record R_{ij} (defined below) for which either INDEX[i, i] or INDEX[j, j] is 0, is considered to be inactive.

Priority Queue (Q_i) and Record (R_{ij}): For each vertex v_i, $1 \le i \le n$, there is a priority queue Q_i of size at most $n - 1$. An element of Q_i is a two-field record $R_{ij} = (j, C_{ij})$, where $j > i$ and $(v_i, v_j) \in \bar{E}_G$. The top element of this queue contains the record corresponding to the lexicographically largest CA_{ij} value in the following sense. It is that R_{ij} which satisfies $CA_{ij} > CA_{rs}$ for all $(v_r, v_s) \in \bar{E}_G$ and $i \ne r$, $j \ne s$. If $CA_{ij} = CA_{rs}$ we assume, without loss of generality, that $CA_{ij} > CA_{rs}$ if either $i < r$, or $i = r$ and $j < s$. A queue Q_ϕ with the status bit INDEX[ϕ, ϕ] = 0 is said to be inactive.

Change-Bit (CB) Vector: One bit is assigned to each vertex. At the beginning of each iteration, the CB vector is initialized to zero. When merging a vertex, say w, into u changes the value of the element a_{uv} in the adjacency matrix from 0 to 1, then CB[v] := 1. Physically it means that (v, w) was an edge before merger, and merging w into u, the pair of vertices u and v are no longer nonadjacent. Setting CB[v] to 1 helps in eliminating the record $R_{uv} = (v, CA_{uv})$ from queue Q_u and in recomputing the CA_{vj} values (if any) for the records in Q_v.

```
procedure PDB;
  begin
    for all l, 1 ≤ l ≤ p, do
      parbegin
        for each i, (l − 1)⌈n/p⌉ + 1 ≤ i ≤ l⌈n/p⌉, do
          begin
            INDEX[i, i] := 1;
            for each j, i + 1 ≤ j ≤ n, do
              begin
                INDEX[i, j] := 0;
                if a_ij = 0 then insert vertex v_j in queue Q_i;
              end;
          end;
      parend;
    for each i, 1 ≤ i ≤ n, do
      for each v_j ∈ Q_i do
        begin
          parallel ANDing of rows i and j of the adjacency matrix A.
          Compute the CA_ij value, create the record R_ij, and store it in Q_i;
        end;
```

```
for all l, 1 ≤ l ≤ p, do
    parbegin
        for each i, (l − 1)⌈n/p⌉ + 1 ≤ i ≤ l⌈n/p⌉, do
            begin
                build the priority queue Q_i by heap management such
                that the top element corresponds to the lexicographically
                largest record in that queue;
            end
    parend;
k := n;
while there is an active queue with an active record do
    begin
        initialize the CB vector to 0's;

        find the lexicographically largest active record, say R_ij, among the
        top elements of all active queues;    (* vertex v_j is to be merged
        into v_i *)
         INDEX[i, j] := 1;    INDEX[ j, j] := 0;
        parallel ORing of rows i and j of A. Overwrite the row and
        column i of A by the ORed bit vector. At the same time, update
        the CB vector;
         for each v, i + 1 ≤ v ≤ n, do
            begin
              if CB[v] = 1 then
                begin
                  recompute CA_vq values of the records in Q_v;
                  eliminate inactive records from Q_v and readjust it;
                end;
            end;
         readjust the queue Q_i;
         if the top element of any other queue is inactive then
              delete it and bring an active record to the top of that queue;
```

```
            k := k - 1;      (* at termination, k is an estimation of χ(G) *)
        end;
    c := 1;
    for each i, 1 ≤ i ≤ n, do
        begin
          if INDEX[i, i] = 1 then
              begin
                for all j, i ≤ j ≤ n, do
                    parbegin
                      if INDEX[i, j] = 1 then COLOR[ j] := c;
                    parend;
                  c := c + 1;
              end;
        end;
end.
```

Note that letting different processors compute different CA_{ij} values would require concurrent reading of a memory cell. This is avoided by employing all p processors in computing one C_{ij} value at a time. Similarly, the records cannot be generated while storing the vertices in different queues. It can be easily shown that the proposed parallel DB algorithm performs correctly without memory read- or write-conflicts.

6.3.1 Complexity Analysis

Using $p \leq n$ processors, the worst-case parallel time required by the PDB algorithm is calculated as follows. Initialization of the INDEX matrix and finding the

elements of the queues (i.e., the set $\overline{E}_G$ of nonadjacent vertex-pairs) require $K_1 \frac{n\ (n-1)}{2\,p}$ time, for some constant K_1. Two bit-vectors, each of size n, can be ANDed in $\left\lceil \frac{n}{p} \right\rceil$ time, and the number of 1's in the resulting bit vector can be counted in $\left\lceil \frac{n}{p} \right\rceil + \log p$ time. Thus the total time needed to compute all CA_{ij} values is $K_2 \left(\frac{n\ (n-1)}{2} - m \right) \left(\left\lceil \frac{2n}{p} \right\rceil + \log p \right)$. Each queue can accommodate at most $n - 1$ records. So each priority queue is built in time at most $K_3\, n$ and the thus total parallel time for queue building is no more than $K_3 \frac{n^2}{p}$ units. The vertex-pair to be merged can be found in $\leq \left\lceil \frac{n}{p} \right\rceil + \log p$ time, and the ORing of two n bit vectors requires $\left\lceil \frac{n}{p} \right\rceil$ time. Updating the CA_{ij} values of relevant records in the queue can be performed in at most $K_4 \left[n \left(\left\lceil \frac{2n}{p} \right\rceil + \log p \right) \right]$ time. Since the PDB algorithm has $n - k$ iterations, its worst-case time complexity T_{PDB} is given by,

$$T_{PDB} \leq K_1 \frac{n\ (n-1)}{p} + K_2 \left(\frac{n^2 - n - 2m}{2} \right) \left(\left\lceil \frac{2n}{p} \right\rceil + \log p \right) + K_3 \frac{n^2}{p}$$
$$+ (n-k) \left(\left\lceil \frac{2n}{p} \right\rceil + \log p \right) + (n-k) \left[K_3\, n + K_4 \left[n \left(\left\lceil \frac{2n}{p} \right\rceil + \log p \right) \right] \right].$$

Next, we discuss the bounds on the asymptotic performance of the PDB algo-

rithm for large graphs. Clearly no iteration is needed to color a complete graph, so a regular graph of degree $\delta = n - 2$ will provide a lower bound on T_{PDB} . In this case an estimate for chromatic number is $k = n - 1$, and $T_{PDB} = O\left[\frac{n^2}{p} + n \log p\right]$. Since the sequential time complexity is $T_{DB} = O(n^2)$, the speedup is obtained as

$$S_{PDB} = \frac{T_{DB}}{T_{PDB}} = \frac{O(p)}{O(1) + O\left[\frac{p \log p}{n}\right]} .$$

To attain the optimal speedup we satisfy $p \log p \approx O(n)$, i.e., $p \leq O\left[\frac{n}{\log n}\right]$. On the other hand, the upper bound on T_{PDB} is achieved by 2-chromatic (i.e., $k = 2$) graphs such as a bipartite graph. For these graphs, $T_{DB} = O(n^3)$ and $T_{PDB} = O\left[\frac{n^3}{p} + n^2 \log p\right]$. Once again the optimal speedup is achieved by using $p \leq O\left[\frac{n}{\log n}\right]$ processors. Therefore, the PDB algorithm is efficient for graphs of varying chromatic numbers or densities; this means that the DB algorithm is easily parallelizable.

6.4 Discussion

We have parallelized two known approximate graph-coloring algorithms for a shared memory parallel computation model, which does not allow concurrent read or

write from the same memory cell. Using an elegant data structure, the parallel largest-degree-first algorithm has been implemented avoiding write-conflict. This algorithm works efficiently for regular or near-regular graphs. Its implementation can be directly applied to parallelize the degree-saturation (DSATUR) algorithm due to Brelaz (1979), where an uncolored vertex with the maximum number of differently colored adjacent vertices is the next possible candidate for coloring. The parallel DSATUR algorithm will be optimal for any graph using $p \leq O\left[\frac{n}{\log n}\right]$ processors. We also believe that for other variations of the LF algorithm (Christofides 1975; Matula et al. 1972; Syslo et al. 1983), our parallelization technique can be efficiently adopted. The second algorithm discussed in this paper is found to be costlier but more easily parallelizable, and yields optimal speedup for graphs of varying densities. Each of our parallel algorithms colors a graph exactly the way its sequential counterpart does; so we need not recompute an upper bound on the number of colors used.

CHAPTER 7

CONCLUSIONS AND FUTURE RESEARCH

With the motivation of contributing to the area of parallel algorithms, we have designed efficient parallel algorithms to solve several problems on undirected graphs. According to the sequential time complexities, the problems of our interest can be classified as follows.

(i) Linear time: connected components, spanning forest, bridges, and bipartiteness.

(ii) Cubic time: fundamental cycle set, and assignment problem.

(iii) Exponential time: vertex-coloring. Two approximate coloring algorithms are considered; one of linear time and the other of cubic time complexity.

The model of computation is an EREW PRAM consisting of a fixed number of processors, which is an abstract generalization of shared memory computers available commercially. Consequently, our parallel algorithms are independent of specific architectural features of a target shared memory machine. It has further been shown by Das, Deo, and Prasad (1988b) that many of these algorithms yield similar asymptotic performance even on hypercube computers, which are of fixed connection type without global shared memory and communication is via a message-passing mechanism.

Divide-and-conquer strategy has been chosen as a paradigm for designing most of our parallel algorithms. In this strategy, the given problem is partitioned into subproblems of almost equal size. Each of these subproblems are solved by different processors, and the ultimate solution is obtained by gradual merging of subsolutions. Interprocessor communication is required during merge phases. To achieve maximum speedup as well as efficiency, two critical factors — computation and communication times — need to be balanced as much as possible. In order to satisfy this condition, we define a new performance measure for parallel algorithms. This new measure, called processor-(time)^2, helps us to choose an optimal number of processors to be employed so that the parallel algorithm is optimally adaptive.

For conflict-free random access by different processors, we have used three simple and known data structures for input graphs, such as unordered list of edges (Chapters 3 and 4), adjacency list matrix (Section 6.2), and cost/adjacency matrix (Chapter 5 and Section 6.3). The parallel algorithms based on an unordered list of edges are optimally adaptive for sparse as well as dense graphs. Since they do not directly parallelize depth-first or breadth-first searches, they avoid the use of the so-called sequential data structures, stacks and queues. Similarly, in one of our parallel graph-coloring algorithms, the inherent sequential nature of linked adjacency lists is alleviated by using adjacency list matrix, which can be accessed randomly and yet preserve sparsity to some extent.

All of our parallel algorithms are deterministic in nature, in contrast to randomized algorithms, where coins are flipped and correct answers are returned with high probability. Though randomization may lead to better performance in many cases, it is rather difficult to apply.

There are several scopes for extending the work presented in this dissertation. We consider them by chapter and offer some general thoughts. One scope for further work is to obtain faster merging algorithms for connected-components and spanning-forest algorithms on an EREW PRAM model with bounded parallelism. As a guideline, if idling of processors can be avoided during merge phases (on the average, 50% of the processors are idle), the worst-case parallel time required by these algorithms could be reduced further. Achieving this would mean that all algorithms in Chapters 3 and 4 would remain optimally adaptive even for a larger number of processors.

It is still not known whether the assignment problem can be solved in deterministic (or randomized), poly-logarithmic time using a polynomial number of processors. Designing optimal parallel algorithms for a minimum-weight matching in sparse bipartite graphs is another interesting topic to be investigated.

Since there are classes of "bad" graphs for which every polynomial-time coloring algorithm performs poorly (Johnson 1974; Mitchem 1976), it is worth parallelizing other approximation algorithms in order to have several parallel coloring algorithms to choose from.

Theory and practice make research complete, especially in a new area like parallel computing, where there is a wide variety of computation models and different *ad hoc* techniques for algorithm design. Many intricacies are taken care of while coding and running an algorithm on an actual parallel machine, which otherwise may remain unnoticed. For example, computing experience gives insight on methods to minimize implementation overheads to increase speedup and efficiency, methods for the design of efficient data structures, the size of the constants involved in the order-analysis of time complexities, and so on. Therefore, experiments can be conducted to study empirical performance of the proposed algorithms on commercial shared memory (as well as fixed connection) computers, with random graphs as input. Experimental results will demonstrate variations in speedup and efficiency as functions of grain-size, number of processors, size and density of input graphs, load balancing, etc. Ultimately, we expect to build a library of efficient parallel programs for solving graph problems on commercial parallel machines.

From the view point of systematic algorithm design, the suitability of divide-and-conquer strategy in designing efficient parallel algorithms for other classes of graph problems, such as shortest path, max-flow, cardinality matching, etc., may also be explored.

APPENDIX

VITA

VITA

Name: Sajal K. Das

Education: Doctor of Philosophy (Computer Science), August 1988
University of Central Florida
Orlando, FL 32816

Master of Engineering (Computer Science), December 1984
Indian Institute of Science
Bangalore - 560 012, INDIA

Bachelor of Technology (Computer Science), July 1983
University of Calcutta
Calcutta - 700 009, INDIA

Bachelor of Science (Physics Honors), April 1980
Ramkrishna Mission Residential College at Narendrapur
University of Calcutta, INDIA

Publications:

1. Das, S. K., and N. Deo. 1986. "Stirling Graphs and Their Properties." *17th Southeastern International Conference on Combinatorics, Graph Theory, and Computing*. In *Congressus Numerantium,* vol. 54 (Dec.), pp. 5-20.

2. Das, S. K., and N. Deo. 1987. "Parallel Coloring of Graphs: Two Approximate Algorithms." *Joint National Meeting, TIMS/ORSA,* May 4-6, New Orleans, Louisiana. Also *Tech. Rep. CS-TR-87-09,* Dept. Comput. Sci., Univ. Central Florida, Orlando, FL.

3. Das, S. K., V. K. Agrawal, D. Sarkar, L. M. Patnaik, and P. S. Goel. 1987. "Reflexive Incidence Matrix (RIM) Representation of Petri Nets." *IEEE Transactions on Software Engineering* SE-13 (June): 643-653.

4. Das, S. K., and N. Deo. 1987. "Rencontres Graphs: A Family of Bipartite Graphs." *The Fibonacci Quarterly* 25 (Aug.): 250-262.

5. Das, S. K., and N. Deo. 1987. "Square-Star Reducibility and its Forbidden Graphs." In *Congressus Numerantium,* vol. 58 (Dec.), 277-290.

6. Das, S. K., and N. Deo. 1987. "Parallel Algorithms for the Assignment Problem." In *Proceedings of International Symposium on Electronic Devices, Circuits and Systems, ISELDECS-87,* vol. 2 (Dec.), pp. 358-360, IIT, Kharagpur, India. Also *Tech. Rep. CS-TR-87-15,* Dept. Computer Sci., Univ. Central Florida, Orlando, FL.

7. Das, S. K., D. Sarkar, V. K. Agrawal, and L. M. Patnaik. 1987. "Invariant-Preserving Petri Net Reduction and Conditions for Invariant-Existence." *Tech. Rep. CS-TR-87-12,* Dept. Comput. Sci., Univ. Central Florida, Orlando, FL.

8. Das, S. K., and N. Deo. 1988. "Divide-and-Conquer-Based Optimal Parallel Algorithms for Some Graph Problems on EREW PRAM Model." *IEEE Transactions on Circuits and Systems* 35 (Mar.): 312-322. (Special Issue on Computational Graph Theory: Algorithms and Applications.)

9. Prasad, S., S. K. Das, and N. Deo. 1988. "Solving a Class of Graph Problems on Hypercube Computers." *Second International SIAM Conference on Vector and Parallel Computing,* June 6-10, Tromso, Norway. Also *Tech. Rep. CS-TR-87-18,* Dept. Comput. Sci., Univ. Central Florida, Orlando, FL.

10. Das, S. K. 1988. "Wheel-Augmented Binary Trees." *International Journal of Computer Mathematics* 24, in press.

11. Das, S. K., and N. Deo. 1988. "Parallel Hungarian Algorithm." *BIT,* to appear.

12. Das, S. K., and N. Deo. 1988. "Parallel Min-Cost Flow Algorithm to Solve the Assignment Problem." *BIT,* to appear.

13. Das, S. K., N. Deo, and S. Prasad. 1988. "Two Minimum Spanning Forest Algorithms on Hypercube Computers." *Tech. Rep. CS-TR-88-04,* Dept. Comput. Sci., Univ. Central Florida, Orlando, FL.

14. Das, S. K., N. Deo, and S. Prasad. 1988. "Parallel Graph Algorithms for Hypercube Computers." *Tech. Rep. CS-TR-88-05,* Dept. Comput. Sci., Univ. Central Florida, Orlando, FL.

LIST OF REFERENCES

Aho, A. V., J. E. Hopcroft, and J. D. Ullman. 1974. *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley.

Akl, S. G. 1986. "An Adaptive and Cost-Optimal Parallel Algorithm for Minimum Spanning Trees." *Computing* 36: 271-277.

Almasi, G. S. 1985. "Overview of Parallel Processing." *Parallel Computing* 2: 191-203.

Anderson, R., and E. Mayr. 1984. *Parallelism and Greedy Algorithms*. Tech. Rep. STAN-CS-84-10003, Dept. Comput. Sci., Stanford Univ., Stanford, CA.

Atallah, M. J., and S. R. Kosaraju. 1984. "Graph Problems on a Mesh-Connected Processor Array." *J. ACM* 31 (July): 649-667.

Awebuch, B., and Y Shiloach. 1987. "New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM." *IEEE Trans. Comput.* C-36 (Oct.): 1258-1263.

Baase, S. 1988. *Computer Algorithms: Introduction to Design and Analysis*. Reading, MA: Addison-Wesley.

Babb II, R. G., ed. 1988. *Programming Parallel Processes*. Reading, MA: Addison-Wesley.

Bauernöppel, F., and H. Jung. 1985. "Fast Parallel Vertex Coloring." In *Lecture Notes in Comput. Sci.*, vol. 199, pp. 28-35, ed. L. Budach. NY: Springer-Verlag.

Beame, P. 1988. "Limits on the Power of Concurrent-Write Parallel Machines." *Inform. Comput.*, 76 (Jan): 13-28.

Borodin, A., and J. E. Hopcroft. 1985. "Routing, Merging, and Sorting on Parallel Models of Computation." *J. Comput. Syst. Sci.* 30 (Feb.): 130-145.

Boyar, J., and H. Karloff. 1987. "Coloring Planar Graphs in Parallel." *J. Algorithms* 9 (Dec.): 470-479.

Brelaz, D. 1979. "New Techniques to Color the Vertices of a Graph." *Commun. ACM* 22 (Apr.): 251-256.

Brent, R. P. 1974. "The Parallel Evaluation of General Arithmetic Expressions." *J. ACM* 21 (Apr.): 201-206.

Chaitan, G. J., M. A. Auslander, A. K. Chandra, J. Coke, M. E. Hopkins, and P. W. Markstein. 1981. "Register Allocation Via Coloring." *Comput. Langs.* 6: 47-57.

Chandy, K. M., and J. Misra. 1988. *Parallel Program Design: A Foundation.* Reading, MA: Addison-Wesley.

Chin, F. Y., J. Lam, and I. -N Chen. 1982. "Efficient Parallel Algorithms for Some Graph Problems." *Commun. ACM* 25 (Sept.): 659-665.

Christofides, N. 1975. *Graph Theory: An Algorithmic Approach.* NY: Academic Press.

Cole, R., and U. Vishkin. 1986. "Deterministic Coin Tossing and Accelerating Cascades: Micro and Macro Techniques for Designing Parallel Algorithms." In *Proc. 18th Annu. ACM Symp. Theo. Comput.* (May): 206-219.

Cook, S. A. 1985. "A Taxonomy of Problems With Fast Parallel Algorithms." *Inform. Control* 64 (Jan.): 2-22.

Cook, S. A., C. Dwork, and R. Reischuk. 1986. "Upper and Lower Time Bounds for Parallel Random Access Machines Without Simultaneous Writes." *SIAM J. Comput.* 15 (Feb.): 87-97.

Corneil, D. G. 1971. "An n^2 Algorithm for Determining the Bridges of a Graph." *Inform. Process. Lett.* 1 (Feb.): 51-55.

Das, S. K., and N. Deo. 1988. "Divide-and-Conquer-Based Optimal Parallel Algorithms for Some Graph Problems on EREW PRAM Model." *IEEE Trans. Circuits and Systems* 35 (Mar.): 312-322.

Das, S. K., N. Deo, and S. Prasad. 1988a. *Two Minimum Spanning Forest Algorithms on Hypercube Computers.* Tech. Rep. CS-TR-88-04, Dept. Comput. Sci., Univ. Central Florida, Orlando, FL.

Das, S. K., N. Deo, and S. Prasad. 1988b. *Parallel Graph Algorithms for Hypercube Computers*. Tech. Rep. CS-TR-88-05, Dept. Comput. Sci., Univ. Central Florida, Orlando, FL.

Dekel, E., and S. Sahni. 1983. "Binary Trees and Parallel Scheduling Algorithms." *IEEE Trans. Comput.* C-32 (Mar.): 307-315.

Deo, N. 1974. *Graph Theory With Applications to Engineering and Computer Science*. Englewood Cliffs, NJ: Prentice-Hall.

Diks, K. 1986. "A Fast Parallel Algorithm for Six-Coloring of Planar Graphs." In *Lecture Notes in Comput. Sci.,* vol. 233, pp. 273-382. NY: Springer-Verlag.

Dongarra, J. J., and I. S. Duff. 1985. *Advanced Architecture Computers*. Tech. Memo. 57, Math. and Comput. Sci. Div., Argonne Nat. Lab., IL.

Doshi, K. J., and P. J. Varman. 1987. "Optimal Graph Algorithms on a Fixed-Size Linear Array." *IEEE Trans. Comput.* C-36 (Apr.): 460-470.

Dutton, R. D., and R. C. Brigham. 1981. "A New Graph Coloring Algorithm." *Comput. J.* 24 (Feb.): 85-86.

Eckstein, D. M., and D. A. Alton. 1977. "Parallel Graph Processing Using Depth-First Search." In *Proc. Conf. Theo. Comput. Sci.,* pp. 21-29.

ENCORE Computer Corp. 1987. *Multimax Technical Summary*. 726-01759, Rev. D.

Ellis, C. S. 1980a. "Concurrent Search and Insertion in 2-3 Trees." *Acta Inform.* 14:63-86.

Ellis, C. S. 1980b. "Concurrent Search and Insertion in AVL Trees." *IEEE Trans. Comput.* C-29 (Sept.): 811-817.

Fich, F., P. Ragde, and A. Wigderson. 1988. "Relations Between Concurrent-Write Models of Parallel Computation." *SIAM J. Comput.* 17 (June): 606-627.

Fortune, S., and J. Wyllie. 1978. "Parallelism in Random Access Machines." In *Proc. 10th ACM Symp. Theo. Comput.* (May): 114-118.

Fredman, M. L., and R. E. Tarjan. 1985. "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms." In *Proc. 26th Annu. IEEE Symp. Foundations Comput. Sci.* (Oct.): 338-346.

Fulks, W. 1961. *Advanced Calculus: An Introduction to Analysis.* NY: John Wiley & Sons.

Galil, Z. 1986. "Sequential and Parallel Algorithms for Finding Maximum Matchings in Graphs." In *Annu. Rev. Comput. Sci.,* vol. 1, pp. 197-224, eds. J. F. Traub et al. Palo Alto, CA: Annual Reviews Inc.

Garey, M. R., and D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco: W. H. Freeman and Co.

Ghosh, R. K. 1986. "Parallel Algorithms for Connectivity Problems in Graph Theory." *Int. J. Comput. Math.* 18: 193-218.

Goldberg, A. V., and S. A. Plotkin. 1987. "Parallel ($\Delta + 1$)-Coloring of Constant-Degree Graphs." *Inform. Process. Lett.* 25 (June): 241-245.

Goldberg, A. V., S. A. Plotkin, and G. E. Shannon. 1987. "Parallel Symmetry-Breaking in Sparse Graphs." In *Proc. 19th Annu. ACM Symp. Theo. Comput.* (May): 315-324.

Goldberg, M. K. 1986. "Parallel Algorithms for Three Graph Problems." *Congressus Numerantium* 54 (Dec.): 111-121.

Goldschlager, L. M. 1978. "A Unified Approach to Models of Synchronous Parallel Machines." In *Proc. 10th ACM Symp. Theo. Comput.* (May): 89-94.

Gottlieb, A., R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. 1983. "The NYU Ultracomputer — Designing an MIMD Shared Memory Parallel Computer." *IEEE Trans. Comput.* C-32 (Feb.): 175-189.

Hambrusch, S. E. 1983. "VLSI Algorithms for the Connected Component Problem." *SIAM J. Comput.* 12 (May): 354-365.

Harary, F. 1969. *Graph Theory.* Reading, MA: Addison-Wesley.

Hayes, J. P., T. N. Mudge, Q. F. Stout, S. Colley, and J. Palmer. 1986. "Architecture of a Hypercube Supercomputer." In *Proc. Int. Conf. Parallel Process.* (Aug.): 653-660.

Hillis, W. D. 1985. *The Connection Machine.* Cambridge, MA: MIT Press.

Hirschberg, D. S., A. K. Chandra, and D. V. Sarwate. 1979. "Computing Connected Components on Parallel Computers." *Commun. ACM* 22 (Aug.): 461-464.

Hochschild, P. H., E. W. Mayr, and A. R. Siegel. 1983. "Techniques for Solving Graph Problems in Parallel Environments." In *Proc. 24th Annu. IEEE Symp. Foundations Comput. Sci.* (Oct.): 351-359.

Horowitz, E., and A. Zorat. 1983. "Divide-and-Conquer for Parallel Processing." *IEEE Trans. Comput.* C-32 (June): 582-585.

Howe, C. D. 1988. "An Overview of the Butterfly GP1000: A Large-Scale Parallel UNIX Computer." In *Proc. Third Int. Conf. Supercomputing,* vol. II, pp. 134-141, eds. L. P. Kartashev and S. I. Kartashev.

Huang, M. A. 1985. "Solving Some Graph Problems With Optimal or Near-Optimal Speedup on Mesh-of-Trees Networks." In *Proc. 26th Annu. IEEE Symp. Foundations Comput. Sci.* (Oct.): 232-240.

Hwang, K., and F. A. Briggs. 1984. *Computer Architecture and Parallel Processing.* NY: McGraw-Hill.

INTEL Corp. 1986. *iPSC Systems Overview.* Order No. 175278-002, Beaverton, OR.

Jamieson, L. H., D. B. Gannon, and R. J. Douglass, eds. 1987. *The Characteristics of Parallel Algorithms.* Cambridge, MA: MIT Press.

Johnson, D. S. 1974. "Approximate Algorithms for Combinatorial Problems." *J. Comput. Syst. Sci.* 9 (Dec.): 256-278.

Karchmer, M., and J. Naor. 1988. "A Fast Parallel Algorithm to Color a Graph With Δ Colors." *J. Algorithms* 9 (Mar.): 83-91.

Karloff, H. J. 1986. *An NC Algorithm for Brooks' Theorem.* Preprint.

Klein, P. N., and J. H. Reif. 1986. "An Efficient Parallel Algorithm for Planarity." In *Proc. 26th Annu. IEEE Symp. Foundations Comput. Sci.* (Oct): 465-477.

Koubek, V., and J. Krsnakova. 1985. "Parallel Algorithms for Connected Components of a Graph." In *Lecture Notes Comput. Sci.,* vol. 199, pp. 208-217, ed. L. Budach. NY: Springer-Verlag.

Kruskal, C. P., L. Rudolph, and M. Snir. 1986. "Efficient Parallel Algorithms for Graph Problems." In *Proc. Int. Conf. Parallel Process.* (Aug.): 869-876.

Kwan, S. C., and W. L. Ruzzo. 1984. "Adaptive Parallel Algorithms for Finding Minimum Spanning Trees." In *Proc. Int. Conf. Parallel Process.* (Aug.): 439-443.

Lai, T. -H., and S. Sahni. 1984. "Anomalies in Parallel Branch-and-Bound Algorithms." *Commun. ACM* 27 (June): 594-602.

Lai, T. -H., and A. Sprague. 1985. "Performance of Parallel Branch-and-Bound Algorithms." *IEEE Trans. Comput.* C-34 (Oct.): 962-964.

Lawler, E. L. 1976. *Combinatorial Optimization: Networks and Matroids.* NY: Holt, Reinhart and Winston.

Lev, G., N. Pippenger, and L. G. Valiant. 1981. "A Fast Parallel Algorithm for Routing in Permutation Networks." *IEEE Trans. Comput.* C-30 (Feb.): 93-100.

Li, G. -J., and B. W. Wah. 1985. "Systolic Processing for Dynamic Programming Problems." In *Proc. Int. Conf. Parallel Process.* (Aug.): 434-441.

Li, G. -J., and B. W. Wah. 1986. "Coping With Anomalies in Parallel Branch-and-Bound Algorithms." *IEEE Trans. Comput.* C-35 (June): 568-573.

Lipovski, G. J., and M. Malek. 1987. *Parallel Computing: Theory and Comparisons.* NY: John Wiley & Sons.

Luby, M. 1986. "A Simple Parallel Algorithm for the Maximal Independent Set Problem." *SIAM J. Comput.* 15 (Nov.): 1036-1053.

Manber, U. 1984. "Concurrent Maintenance of Binary Search Trees." *IEEE Trans. Softw. Eng.* SE-10 (June): 777-784.

Matula, D. W., G. Marble, and J. Isaacson. 1972. "Graph Coloring Algorithms." In *Graph Theory and Computing,* pp. 109-122, ed. R. Read. NY: Academic Press.

Miller, R., and Q. F. Stout. 1987a. "Data Movement Techniques for the Pyramid Computer." *SIAM J. Comput.* 16 (Feb.): 38-60.

Miller, R., and Q. F. Stout. 1987b. "Graph and Image Processing Algorithms for the Hypercube." In *Proc. SIAM Conf. Hypercube Multiprocessors,* pp. 418-425.

Mitchem, J. 1976. "On Various Algorithms for Estimating the Chromatic Number of a Graph." *Comput. J.* 19 (May): 182-183.

Moitra, A., and S. S. Iyenger. 1987. "Parallel Algorithms for Some Computational Problems." In *Advances in Computers,* vol. 26, pp. 94-153, ed. M. C. Yovits. NY: Academic Press.

Naor, J. 1987. "A Fast Parallel Coloring of Planar Graphs With Five Colors." *Inform. Process. Lett.* 25 (Apr.): 51-53.

Nassimi, D., and S. Sahni. 1980. "Finding Connected Components and Connected Ones on a Mesh-Connected Computer." *SIAM J. Comput.* 9 (Nov.): 744-757.

Nath, D., and S. N. Maheshwari. 1982. "Parallel Algorithms for the Connected Components and Minimal Spanning Tree Problems." *Inform. Process. Lett.* 14 (Mar.): 7-11.

Nelson, P. A. 1987. *Parallel Programming Paradigms.* Ph. D. Diss., Dept. Comput. Sci., Univ. Washington, Seattle, WA.

Paige, R. C., and C. P. Kruskal. 1985. "Parallel Algorithms for Shortest Path Problems." In *Proc. Int. Conf. Parallel Process.* (Aug.): 14-20.

Papadimitriou, C. H., and K. Steiglitz. 1982. *Combinatorial Optimization: Algorithms and Complexity.* NJ: Prentice-Hall.

Paul, W., U. Vishkin, and H. Wagener. 1983. "Parallel Dictionaries on 2-3 Trees." In *Lecture Notes in Comput. Sci.,* vol. 154, pp. 597-609. NY: Springer-Verlag.

Pawagi, S. 1987. "Parallel Algorithms for Maximum Weight Matching in Trees." In *Proc. Int. Conf. Parallel Process.* (Aug.): 204-206.

Quinn, M. J. 1987. *Designing Efficient Algorithms for Parallel Computers*. NY: McGraw-Hill.

Quinn, M. J., and N. Deo. 1984. "Parallel Graph Algorithms." *Comput. Surveys* 16 (Sept.): 320-348.

Quinn, M. J., and Y. B. Yoo. 1984. "Data Structures for the Efficient Solution of Graph Theoretic Problems on Tightly-Coupled MIMD Computers." In *Proc. Int. Conf. Parallel Process*. (Aug.): 431-438.

Rao, V. N., and V. Kumar. 1988. *Concurrent Access of Priority Queues*. Tech. Rep. TR-88-06, Dept. Comput. Sci., Univ. Texas, Austin, TX.

Reingold, E. M., J. Nievergelt, and N. Deo. 1977. *Combinatorial Algorithms: Theory and Practice*. Englewood Cliffs, NJ: Prentice-Hall.

Savage, C., and J. Ja' Ja'. 1981. "Fast, Efficient Parallel Algorithms for Some Graph Problems." *SIAM J. Comput.* 10 (Nov.): 682-691.

Schwartz, J. T. 1980. "Ultracomputers." *ACM Trans. Progr. Langs. Syst.* 2 (Oct.): 484-521.

Seitz, C. L. 1985. "The Cosmic Cube." *Commun. ACM* 28 (Jan.): 22-33.

SEQUENT Computer Systems Inc. 1986. *Balance Technical Summary*. MAN-0110-00, Beaverton, OR.

Shamir, E., and E. Upfal. 1984. "Sequential and Distributed Graph Coloring Algorithms With Performance Analysis in Random Graph Spaces." *J. Algorithms* 5 (Dec.): 488-501.

Shiloach, Y., and U. Vishkin. 1981. "Finding the Maximum, Merging, and Sorting in a Parallel Computation Model." *J. Algorithms* 2 (Mar.): 88-102.

Shiloach, Y., and U. Vishkin. 1982. "An $O(\log n)$ Parallel Connectivity Algorithm." *J. Algorithms*. 3 (Mar.): 57-67.

Snir, M. 1985. "On Parallel Searching." *SIAM J. Comput.* 14 (Aug.): 688-708.

Stockmeyer, L., and U. Vishkin. 1984. "Simulations of Parallel Random Access Machines by Circuits." *SIAM J. Comput.* 13 (May): 409-422.

Stout, Q. F. 1985. "Tree-Based Graph Algorithms for Some Parallel Computers." In *Proc. Int. Conf. Parallel Process.* (Aug.): 727-731.

Syslo, M. M., N. Deo, and J. S. Kowalik. 1983. *Discrete Optimization Algorithms.* Englewood Cliffs, NJ: Prentice-Hall.

Tang, C. Y., and R. C. T. Lee. 1984. "Optimal Speeding Up of Parallel Algorithms Based Upon Divide-and-Conquer Strategy." *Inform. Sci.* 32: 173-186.

Tarjan, R. E. 1972. "Depth-First Search and Linear Graph Algorithms." *SIAM J. Comput.* 1 (June): 146-160.

Tarjan, R. E. 1974. "A Note on Finding the Bridges of a Graph." *Inform. Process. Lett.* 2: 160-161.

Tarjan, R. E., and U. Vishkin. 1985. "An Efficient Parallel Biconnectivity Algorithm." *SIAM J. Comput.* 14 (Nov.): 862-874.

Te Riele, H. J. J., Th. J. Dekker, and H. A. van der Vorst, eds. 1987. *Algorithms and Applications on Vector and Parallel Computers.* NY: North-Holland.

Thompson, C. D. 1979. "Area-Time Complexity for VLSI." In *Proc. 11th ACM Annu. Symp. Theo. Comput.* (May): 81-88.

Tsin, Y. H., and F. Y. Chin. 1984. "Efficient Parallel Algorithms for a Class of Graph Theoretic Problems." *SIAM J. Comput.* 13 (Aug.): 580-599.

Upfal, E., and A. Wigderson. 1984. "How to Share Memory in a Distributed Environment." In *Proc. 25th Annu. IEEE Found. Comput. Sci.* (Oct.): 171-180.

Vishkin, U. 1983a. "Implementation of Simultaneous Memory Address Access in Models That Forbid It." *J. Algorithms* 4 (Mar.): 45-50.

Vishkin, U. 1983b. *Synchronous Parallel Computation — A Survey.* Tech. Rep. 71, Comput. Sci. Dept., New York Univ., NY.

Vishkin, U. 1984. "An Optimal Parallel Connectivity Algorithm." *Discrete Appl. Math.* 9 (Oct.): 197-207.

Veldhorst, M. 1986. "Parallel Dynamic Programming Algorithms." In *Lecture Notes in Comput. Sci.,* vol. 239, pp. 393-402. NY: Springer-Verlag.

Welsh, D. J. A., and M. B. Powell. 1967. "An Upper Bound for the Chromatic Number and Its Application to Timetabling Problem." *Comput. J.* 10 (Feb.): 85-86.

Williams, M. H., and K. T. Milne. 1984. "The Performance of Algorithms for Coloring Planar Graphs." *Comput. J.* 27 (May): 165-170.

Wyllie, J. C. 1979. *The Complexity of Parallel Computations.* Ph. D. Diss., Dept. Comput. Sci., Cornell Univ., Ithaca, NY.

Yeh, D. Y. 1986. "Finding Fundamental Cycles and Bridges on a Tree-Structured Parallel Computer." *Inform. Sci.* 40 (Nov.): 67-73.

Yeh, D. Y., and D. T. Lee. 1984. "Graph Algorithms on a Tree-Structured Parallel Computer." *BIT* 24: 333-340.