1991

# Parallel and Distributed Algorithms for a Class of Graph-Related Computational Problems.

Rajanarayanan Subbiah
*Louisiana State University and Agricultural & Mechanical College*

Order Number 9219577

# Parallel and distributed algorithms for a class of graph related computational problems

Subbiah, Rajanarayanan, Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1991

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

# Parallel and Distributed Algorithms for a Class of Graph Related Computational Problems

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by

Rajanarayanan Subbiah

M.Sc. (Tech), Birla Institute of Technology & Science, Pilani, India, 1985
M.S., Louisiana State University, Louisiana, 1989

December, 1991

## Acknowledgements

I extend my sincere thanks to my major advisor, Prof. Sitharam Iyengar, without whose support, encouragement, criticisms and suggestions this work would not have been a success. His guidance at every step of my work was invaluable. This research was also partially funded by the LEQSF Board of Regents grant to Prof. Iyengar. I would like to thank Prof. Donald H. Kraft for all his support and especially for his humor. I will always have "Golab Jolabs" for you whenever and wherever we meet in the future. My sincere thanks are due to the members of the graduate committee: Prof. Bush Jones (I can never forget CSC 7001, Fall 1986), Dr. Doris Carver (I wish I had taken Software Engineering), Prof. Bert Boyce (Thanks for everything, I cannot forget the cup with the Whale).

I would like to thank Dr. David Cordes and Dr. Mike Stinson for the wonderful environment they provided while I was still a fledgling researcher at LSU.

I would like to thank Dr. Sridhar Radhakrishnan from the bottom of my heart for making me believe that I had the potential to make a name for myself. He knew me well and trusted me, I am ever grateful to him. I enjoyed collaborating with him in all my research endeavors and hope to keep this an ever lasting relationship. Thanks are also due to MaK (M.A. Krishna) for housing and feeding me at New Orleans.

I am very grateful to Dr. Krishnakumar Narayan (KK) whose encouragement and support helped me complete my graduate studies at LSU. Thanks to Mrs. KK (Sunita) for all the "idlis". Every one of my friends has helped me in some way or the other. Bardhan (too many good times to mention), Rao (he knows me in and out), Hegde

(friend, philosopher and guide, He is a GEM), Phatak (hope you learnt something from all the cooking lessons I gave you, He is the best friend one can have). My friends are my life, and would like to thank each and every one of them for their company and goodwill.

I would specially like to thank Shankar for those long nights when he helped me proof read this document. I owe a lot to Dr. Mohan, starting from "Digger" to Distributed Algorithms, he helped me improve and become more positive in life.

I owe this chance to study abroad to my brother (Balaji). I know that this entire experience has made me a better person and would like to thank my parents for every bit of understanding.

# Table of Contents

## Abstract

The subject of parallel computing has moved from the exotic to mainstream computer science within a decade. The recent developments in parallel processing technologies has induced the evolution of many new parallel hardware architectures, interconnection technologies, and algorithm paradigms. There exist at least two models of parallel computing, namely, shared-memory and message-passing. This research addresses problems in both these types of systems, and proposes efficient parallel (Shared-Memory Model) and distributed (message-passing) algorithms for a variety of graph related computational problems. The algorithms developed can add the primitives required in the design of a distributed operating system.

The contribution of the dissertation is two fold. In part I, we design algorithms for three generic problems in distributed systems: set manipulation, network structure recognition and facility placement. In part II, we design parallel algorithms for facility placement and $p$-center problems in trees using an EREW-PRAM model of computation. We also design efficient parallel algorithms for range searching using distributed data structures.

Network structure recognition and facility placement are two important graph problems in distributed systems. Knowledge of the structure of a network helps in the design of efficient algorithms for routing, center location, sorting etc. We present optimal distributed algorithms for recognizing rectangular-mesh networks. The time and message complexity of our algorithm is linear in the number of nodes in the network. We also lay the foundation for the recognition of 2-reducible, outer-planar and cactus graphs. These algorithms have a message complexity of $O(kn)$, where, $k$ is

the number of isolated two degree nodes in the network.

Facility placement or $r$-domination is NP-complete on general networks. Many variations of the domination problem have been studied on restricted graph structures: trees, chordal, interval graphs etc. We introduce the problem of reliable $r$-domination and design unified optimal distributed algorithms for the total, reliable and independent $r$-domination on trees. The time and message complexity of our algorithm is $O(n)$, where $n$ is the number of nodes in the tree.

In the domain of set manipulation we design optimal algorithms for determining the intersection of sets in a distributed environment, where each processor is assumed to have its own set. In many situations, the intersection would be null, where, we propose optimal algorithms for determining the mode (element occurring maximum number of times). The time and message complexity of our set intersection algorithm is $O(mn)$, where $m$ is the cardinality of the smallest set.

In part II of our research we design optimal algorithms for $r$-domination and efficient parallel algorithms for the $p$-center problems on trees. We also present an optimal algorithm for computing the maximum independent set on intervals i the EREW-PRAM model and such a set is required by our algorithms on trees. The $r$-domination problem on trees can now be solved in $O(logn)$ time with $O(n/logn)$ processors using the EREW-PRAM model.

Range Search has important applications in the areas of databases and computational geometry. The range search problem is to obtain a set of data points (tuples, records) satisfying a query which specifies a range of values on each dimension (attribute) of the data. A parallel algorithm for range searching is developed here using the

concept of distributed data structures. We use the range tree proposed by Bentley as our data structure to be distributed. We show that $O(logn)$ search time can be effected for a range search on $n$ 3-dimensional points using $(2.\log^2 n - 14.logn + 8)$ processors and this is optimal for the range tree distribution. We present a non-trivial implementation technique on the hypercube parallel architecture with which the above time and processor bound can be achieved without any communication overhead. Our algorithm can easily be generalized for the case of $d$-dimensional range search.

# Chapter One

# INTRODUCTION

## 1.1 Overview

The recent developments in parallel processing technologies has induced the evolution of many new parallel hardware architectures, interconnection technologies, and algorithm paradigms. The cost of hardware processing units has been drastically reduced by the sheer diversity of this technological innovation. The result has been the development of computer networks, and interconnection of various computers, which have become viable and cost-effective. Besides being reliable, computer networks offered several advantages to the users including interactive computing, resource sharing, and cooperation. Compared to the traditional monolithic computing systems ( controlled by a centralized operating system ), these 'coordinated' multiprocessor computing systems offer reduced incremental cost, modularity, better reliability response, and performance, and improved utilization. Thus, the feasibility and realization of multiprocessor computing has opened avenues to challenging applications and research problems.

There are two categories of multiprocessor systems: a multiple computer system, and a multiprocessing system. A *multiple computer system* consists of several autonomous computers which may or may not communicate with each other. A *multiprocessing system* is controlled by one operating system which provides interaction between processors and their programs at the process level, data set level, and element

level. The communication in both these systems may occur by sending messages from one processor to the other or by sharing a common memory. There are some similarities between multiprocessors and multicomputer systems since both are motivated by the same basic goal - the support of concurrent operations in the system. However, there exists an important distinction between multiple computer systems and multiprocessors, based on the extent of resource sharing and cooperation in the solution of a problem. A multiple computer system is characterized by the lack of shared memory. These systems coordinate their activities by message passing. This research addresses problems related to both types of systems.

Another classification of multiprocessor system is based on the architectural model used. Two categories under this classification are the tightly coupled multiprocessr and the loosely coupled multiprocessor. The first category is also referred to as a parallel machine where individual processors communicate via shared memory. The second category is referred to as a distributed system where autonomous multiple processors communicate via message passing.

## 1.2 Parallel vs Distributed Computation

Parallel algorithms refer to the algorithms that are executed on a parallel computer, which is generally a tightly-coupled multiprocessing system with shared memory. Synchronization and communication are two important mechanisms that are used to resolve contentions, and subsequently achieve a common goal. Distributed systems or loosely coupled multicomputer systems are characterized by the absence of a shared memory. Hence, the mechanisms of synchronization and communication

have to be redefined for these systems. Since message passing is an important part of a distributed system, message primitives should be defined for sending and receiving messages.

In any sequential or parallel system, control information is available with the operating system, which allows the synchronization and control of various events. In a distributed system, no such global control is available. No single node/processor has complete control information. There is also no system wide clock to synchronize events.

The performance of parallel algorithms is usually characterized by computing the total elapsed time or processing time, and the number of processors used in parallel computing. In distributed systems, the communication costs play a major role in defining the efficiency of an algorithm. Processing time is generally considered negligible in the analysis. The time complexity is usually measured in terms of time units, where the time taken for a message to traverse from one node to another is taken as one time unit. Hence the measure of the efficiency of a distributed algorithm is expressed as the total number of messages required to execute an algorithm.

Compared to sequential or parallel algorithms, the design of distributed algorithms is more complicated. Due to unpredictable delays in communication, the order of computation and hence the operational results become unpredictable during the execution of distributed algorithms. In the following table we illustrate the basic differences between distributed and non-distributed systems.

| Algorithm | Distributed | Non-Distributed |
|---|---|---|
| 1. Control Information | Central | At Individual Sites |
| | | No Global Information |
| 2. Order of Operational Results | Predictable | Unpredictable |
| 3. Event control | At one site | Different Sites |
| | Synchronous | Asynchronous |
| 4. Communication | Shared Memory | Messages |
| 5. Performance Measures | Time Complexity | Time Complexity |
| | Space Complexity | Message Complexity |

Table 1.1

## 1.3 An Overview of Distributed Algorithms

Consider a distributed system where a set of physically distinct computational units work on a common problem, while their operation is coordinated via a communication medium connecting some or all of of these units. Each computing unit has certain processing and memory capabilities, and is preprogrammed to perform its part

of the computation as well as send and receive control/data messages over the communication links. The program residing on each node is called a node algorithm and the ensemble of all these node algorithms is called a *Distributed Algorithm*[Sega83]. Based on the control mechanism, distributed algorithms may be centralized or non-centralized.

In centralized control, one node is designated as the control node; this controls access to a shared object. Whenever a node wishes to gain access to the shared object, it sends a REQUEST message to the control node, which returns a REPLY (permission) message when the shared object becomes available (allocated) to the node. These algorithms are characterized by the following properties:

1. Only the central (control) node makes the decision.
2. All the necessary information is concentrated in the central control node.

The weaknesses of this type of system are its vulnerability to failure of the central control and the possibility that the central control may become a bottleneck. If these problems can be tolerated, then the central control method is suitable. Some variations of this idea exist, such as the idea that the central control can migrate from node to node.

In non-centralized control, no single node is designated as the control node. More than one node can start the algorithm and control is generally shared or passed from node to node. A "fully distributed" algorithm is characterized by these properties:

1. All nodes have equal amount of information.

2. All nodes make decisions based solely on local information.

3. All nodes bear equal responsibility for the final decision.

4. All node expend equal effort in effecting a final decision.

5. Failure of a node, in general, does not result in a total system collapse.

In practice, there are very few fully distributed algorithms. Some algorithms may dictate that every node bears an equal amount of responsibility, while the amount of effort expended by individual nodes to obtain, for example, mutual exclusion is not equal. For other algorithms, the reverse might be true. Still others may lead to a total system collapse if a single node fails.

Regardless of the manner and degree of distribution, there are two fundamental assumptions common to all *distributed algorithms*:

1. Each node has only a partial picture of the total system and must make decisions based on this information.

2. There exists no system-wide common clock.

The latter assumption is a major constraint since the temporal ordering of events is a fundamental concept in our view of computer systems.

Distributed algorithms for a number of problems have been reported in the literature; for example, network traversal algorithms and global state determination algorithms [Rayn88]. Extensive literature for a variety of problems such as distributed mutual exclusion, electing a leader, termination, failure detection, synchronization and graph algorithms (median, center, shortest paths, and minimal spanning trees ) may be found in [Chan82, Chan85, Chan83, Dech87, Dijk80, Gall83, Hela88, Moha89,

Hirs80, Kora84, Laks87, Lamp78, Lamp, Less81].

Performance of distributed algorithms is measured in terms of both time complexity and message complexity. The time complexity of a distributed algorithm is the maximum time elapsed from the beginning to the termination of the algorithm, assuming that the time for delivering a message over a link is at most one unit of time. Generally, the time for receiving a message, local processing and sending a message over a link is considered negligible, since the communication costs are very high. The communication or message complexity of a distributed algorithm is the total number of messages transmitted during the execution of the algorithm.

## 1.4 An Overview of Parallel Algorithms

Research and development of multiprocessing systems are aimed at improving throughput, reliability, flexibility, and availability. These systems contain two or more processors of approximately comparable capabilities. All processors share access to common sets of memory modules, I/O channels and peripheral devices. Most importantly, the entire system must be controlled by a single integrated operating system, providing interactions between processors and their programs at various levels. Besides the shared memories and I/O devices, each processor has its own local memory and private devices. Interprocessor communications can be done through the shared memories or through an interrupt network.

The subject of parallel computing has moved from the exotic to mainstream computer science within a decade. This is mostly because of possible limits to hardware speed and software efficiency in the realm of sequential computing. Even

before the existence of real parallel machines, computer scientists were developing a theoretical framework to develop the model of parallel computations based on processor capability, memory accessibility, and the pattern of interconnection among processors. Though there is no consensus on a model of parallel computation, many studies have focussed on the *parallel-random-access-machine* (PRAM) model (see Karp and Ramachandran [Karp90]). The PRAM model is a parallel analog of the sequential RAM. It consists of several independent sequential processors, each with its own private memory, communicating with each other through a global memory. In one unit of time, each processor can read one global or local memory location, execute a single RAM operation, and write into one global or local memory location.

PRAMs can be classified according to restrictions on the global memory access. In *EREW* (Exclusive Read Exclusive Write) PRAMs simultaneous access to any memory location for both reading and writing is forbidden. In a *CREW* (Concurrent Read Exclusive Write) PRAM simultaneous reads are allowed but not simultaneous writes. A *CRCW* (Concurrent Read Concurrent Write) PRAM allows both simultaneous reads and writes. In the case of a concurrent write we assume that an arbitrary processor succeeds, though other assumptions are possible (see Moitra and Iyengar [Moit87]). These models are increasingly powerful in that order.

It is known that the CREW and the CRCW PRAM models can be simulated by an EREW-PRAM model in $O(logP)$ time with $O(P)$ extra processors or with no extra processors in $O(\log^2 P)$ time [Ecks79]. It was shown in [Upfa84], that all PRAM models with $P$ processors can be simulated by an ultracomputer (bounded-degree network of processors with no global memory) in $O(logP\ (loglogP)^2)$ time per step and

with no extra processors. Having described the models of parallel computation we will be using in our work, we next turn our attention to the class of algorithms we will consider. In our discussion below, we only consider sequential algorithms having polynomial-time complexity.

The most natural and a very practical idea of parallelism is to make use of a fixed number of processors say, P (independent of the size of the input say, n). In this case, it is not hard to see that the speedup of a parallel algorithm (over the sequential one) for a problem will depend on the input size and when the input size is increased, the number of processors has to be dependent on the input size. The agreement in this case is to make use of only a reasonable amount of hardware i.e., a polynomial number of processors. The polynomiality of resources (time and processors) has been accepted as a reasonable demand.

Having discussed the use of a polynomial number of processors let us consider the issue of *time*. The worst-case time-complexity of a parallel algorithm is a measure of the maximum time taken by any of the processors over all inputs. Using a polynomial number of processors we may hope to do as well as getting a constant-time parallel algorithm. However, creating a polynomial number of processors requires nonconstant time in a reasonable model of parallel computation. For example, if we want to create say $O(n^r)$ processors, we need $O(\log_2 n^r)$ time assuming a binary-tree configuration for processor creation. Hence reasonably good speed up can be said to have been achieved when the parallel time-complexity is a *polylog* (polynomial in the logarithm) of $n$. Note that in this case, the speedup is exponential! Without much

further ado, let us say that the class of *NC algorithms* (due to Nicholas Pippenger [Pipp87]) consists of algorithms which run in $O(\log^r n)$ time and make use of a polynomial number of processors. NC algorithms have been found for several important problems in areas like algebra, graph theory, and computational geometry. For a good survey on NC algorithms see Karp and Ramachandran [Karp90].

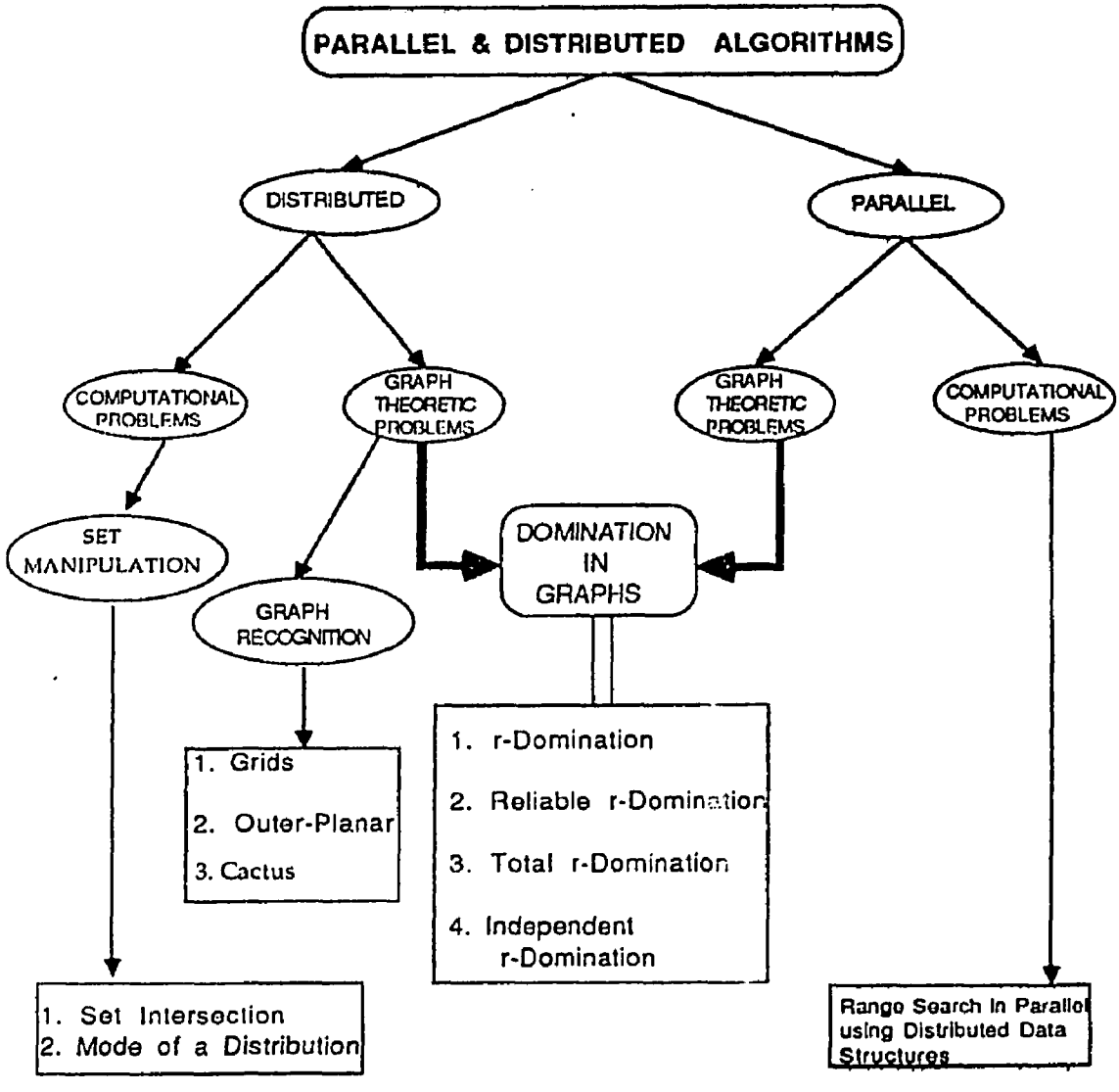



fig 1.1 Contributions of the Dissertation

## 1.5 Contributions of the Dissertation

In fig 1.1 we provide a schematic representation of the organization and flow of this research. We consider three generic problems in distributed systems, namely, set intersection and other set manipulation problems, facility placement or r-domination problem and its variations; and network recognition problems. We also design optimal parallel algorithms for the r-domination problem using an EREW-PRAM model. Using these algorithms we also solve the p-center problem in trees.

In distributed systems, no global information is maintained at each node. In an office environment, the problem of scheduling a meeting can be abstracted into the set intersection problem [Rama88]. Scheduling meetings among personnel is one of the most common tasks in an office environment. Each person can be considered to have a collection of *available slots of time* in such a setting and the task is to find a slot in their intersection. The above problem can be abstracted as that of finding an element in the intersection of the sets.

The problem of determining the mode of the distribution or finding the element that occurs most number of times in the system is an extension of the set intersection problem wherein the intersection may be empty. In these cases, it is important to find that time-slot that is most preferred. Another variation of this problem is that of electing a representative for a task in an office environment. Each person is allowed to choose any number of representatives. The problem is to find the one with the maximum number of votes.

In general, many distributed algorithms for general networks are expensive in terms of both time and message complexities, whereas algorithms for certain

topologies are much faster and easy to design. Many problems are much simpler on tree structures, but are NP-complete on general graphs. Message routing becomes easy to handle on mesh-connected or planar networks. This is the basic motivation behind designing distributed algorithms for recognizing the structure of a network. The design of efficient distributed algorithms for network recognition has many advantages because knowledge of the structure of the network simplifies many problems. We design optimal algorithms for the recognition of rectangular-mesh connected networks. These algorithms can be extended to recognize generalized boolean n-cube structured networks. We also design efficient algorithms for the recognition of 2-reducible, outer-planar and cacti networks.

The problem of finding a minimum r-dominating set for a network $G=(V,E)$ is that of selecting a subset $D_r$ such that every other vertex of $G$ is at a distance $r$ or less from some vertex in $D_r$. We present optimal distributed algorithms for determining the minimum $r$-dominating set when G is a tree. The domination problem is to determine a dominating set $D$ with minimum cardinality. The dominating set is used in solving facility placement problems in networks, where each node represents a customer and/or a potential site for placement of a facility. A feasible solution corresponds to a set of facilities located at $D \subseteq V$, such that each customer is adjacent to or at a distance at most $r$ to some facility. Several variations of the domination problem have been studied in the past [Corn86, He90, Slat76, Kari79, Keil86]. We introduce the problem of reliable r-domination and provide unified optimal distributed algorithms to solve the total, reliable and independent r-domination problems on trees.

## 1.6 Scope and Organization of the Dissertation

The dissertation deals with the design of efficient algorithms for multiprocessor systems. We have discussed the basic differences between a parallel system and a distributed system in Chapter 1. Here, we also have provided definitions for the measure of the performance of parallel algorithms and distributed algorithms. The dissertation is organized into two parts. In part I we concentrate on the design and analysis of efficient distributed algorithms to solve three generic problems, namely, network recognition, set manipulation and facility placement. We list several paradigms, discuss concepts and techniques used by researchers to solve various fundamental problems in distributed computation. We also provide the basic model of computation used in all our distributed algorithms in chapter 2.

In part II, we discuss the design of efficient parallel algorithms for the facility placement problem and the p-center problem in tree networks. We also provide an efficient algorithm for the range search problem in parallel using distributed data structures.

The first part (Chapters 3 and 4) deal with the set manipulation problem in distributed systems. Here, we provide optimal distributed algorithms for the set intersection problem and extend this algorithm to determine certain values of statistical interest, namely the mode of the distribution.

The problem of network learning or recognition of network structures is dealt with in chapter 5. We present solutions for classical problems in network recognition [also see[Rama89].] We provide the basic motivation for the recognition of network structures and provide an optimal algorithm for the recognition of rectangular-mesh

connected networks. We provide insight into the design of efficient algorithms for the recognition of 2-reducible, outer-planar, and cactus (edge disjoint cycle ) graphs.

The problem of domination is discussed in detail in chapter 6. We discuss the nature of the problem and provide insight into the variety of domination problems discussed in the literature in the recent past. We list several classical domination problems and their complexities. We discuss the NP-completeness of the domination problem, its close and natural relationship to other NP-complete problems, and the subsequent interest in finding polynomial time solutions to domination problems in special classes of graphs.

Several variations of the domination problem have been proposed in the literature. In chapter 7, we introduce the problem of reliable $r$-domination and provide optimal distributed algorithms for the reliable and total dominating set problem in trees.

In part II of this dissertation, we provide parallel algorithms for several problems. In chapter 8, we provide optimal parallel algorithms for the $r$-domination problem in trees, using an EREW-PRAM model. We also provide efficient algorithms for the $p$-center problem in tree model using the same model of computation. In chapter 9, we briefly discuss the design of efficient parallel algorithms for the range search problem using distributed data structures.

At the end of each chapter we provide a summary which details the salient features of that chapter. In chapter 10, we provide the conclusions of this dissertation and provide an insight into the future course of action in the realm of distributed and parallel computation.

# Chapter Two

## PRELIMINARIES

## 2.1 Introduction

Formally, distributed systems, refers to the integration of autonomous systems that cooperate to achieve a common goal. There are basically two schemes for building such systems. In a tightly coupled system, the processors share memory and clock. In these *multiprocessor* systems, communication usually takes place through the shared memory. In a loosely coupled system, the processors do not share memory or a clock. Instead, each processor has its own local memory. The processors communicate with each other through various communication lines, such as high-speed busses or telephone lines. These systems are referred to as *Distributed systems*.

In general, a distributed system is defined as a system where the entities that form the system cooperate in achieving a common aim, by exchanging messages. In [CORN85, Film84, M.Ra85, Hela88, A.Se83, Less81, Less81] distributed systems are characterize as being completely accurate and nearly autonomous systems. These systems are accurate because they operate on complete and correct information. They are nearly autonomous because each processor has an almost complete database of the information it needs. Lynch and Fischer [Lync85] present a mathematical model of the distributed system and a mathematical model of their input/output behaviors.

## 2.2 Taxonomy of Distributed Systems

The processors in a distributed system may vary in size and function. They may

include small microprocessors, work stations, minicomputers, and large general purpose computer systems. These processors are referred to by a number of names such as *sites, nodes, computers*, and so on, depending on the context in which they are mentioned.

There are four major reasons for building distributed systems: resource sharing, computation speed-up, reliability, and communication.

### Resource Sharing

If a number of different sites (with different capabilities) are connected to each other, then a user at one site may be able to use the resources available at another. In general, resource sharing in a distributed system provides mechanisms for sharing files at remote sites, processing information in a distributed database, printing files at remote sites, using remote specialized hardware devices, such as high-speed array manipulation hardware, and other operations.

### Computation Speed-up

If a particular computation can be partitioned into a number of subcomputations that can run concurrently, then the availability of a distributed system may allow us to distribute its computation among the various sites in order to run the computation concurrently. In addition, if a particular site is currently overloaded with jobs, some of them may be moved to other lightly loaded sites. This movement of jobs is called load sharing.

### Reliability

If one site fails in a distributed system, the remaining sites can potentially continue operating. If the system is composed of a number of autonomous installations (general purpose computers), the failure of one of them should not affect the rest. The failure of one site must be detected by the system and appropriate action may be needed to recover from the failure.

### Communication

When a number of sites are connected to each other via a communication network, the users at different sites have the opportunity to exchange information. Many systems have modeled this communication system on the post office. In a distributed system we refer to such activity as electronic mail. In a message system, the processors are allowed to communicate through explicit messages. The responsibility for providing the communication rests with the operating system.

## 2.3 Features of Distributed Algorithms

One of the main characteristics of distributed algorithms is that control is distributed throughout the algorithm. No fixed hierarchical or prioritized relation exists among the set of processors, over which control may be distributed. Each processor is as powerful as the other in the system, which makes the design of distributed algorithms very difficult and challenging. Several features of distributed algorithms must be considered while designing new algorithms. We list a few salient features below.

## Symmetry

An important feature of a distributed algorithm is the extent to which an algorithm is partitioned. There are several levels of symmetry.

1. Textual symmetry: The text executed by each processor is the same, but the names or ID of each processor is distinct.
2. Strong symmetry: The program texts are identical, and there is no reference to processor names. Identical or different behaviors are possible, according to the messages received.
3. Total symmetry: The program texts are identical and all the processors behave in the same way.
4. Asymmetry: Processors can be interchanged.

Most distributed algorithms either assume a distinct ID or name for processors, since many contentions can be resolved based on this alone. Two processors which send the same message to another processor can be distinguished only by their names. The distributed algorithms considered in this dissertation fall into the category of textual symmetry.

## Fault Tolerance

A distributed system should not collapse under some node/link failures. Therefore, it is imperative that the distributed algorithms are made resilient to failures. Since no node has any special control, these algorithms will work for any network topology.

### Network Models

When designing algorithms for distributed systems, it is desirable to make the fewest possible assumptions regarding the system. The fewer the assumptions, the more sophisticated the algorithm. A general assumption made is that the topology of the network does not change during the execution of the algorithm.

### Algorithm Complexity

The total number of messages used during the execution of a distributed algorithm is a measure of the efficiency of that algorithm. If a lesser number of messages is used, the network traffic will be low, and hence the delays for transmission of individual messages will also be low.

### Global State

No global state information is available at any one processor; therefore, algorithms have to be designed such that every processor can make decisions with just the neighborhood or local information. The absence of a global clock necessitates the design of asynchronous algorithms.

## 2.4 Concepts and Techniques

Many standard techniques such as acknowledge messages, broadcast etc., are used in distributed algorithms. We shall list three main techniques, namely, diffusing computations, circulating token, and time stamping.

i) Diffusing Computations: This principle was proposed by Dijkstra and scholten[Dijk80]. A spanning tree representing the network is used to detect ter-

mination of processes. Here the root of the spanning tree plays a different role compared to other nodes in the network. This also allows each processor to defined in a parent-child relationship with some other processor.

ii) Circulating Token: A special message referred to as a *token* is circulated in the network. A processor possessing the token at any point in time is said to be privileged and hence can execute some special action, like accessing a shared resource. This is a simple and powerful technique in solving mutual exclusion problems.

iii) Time stamping: Lamport[Lamp78] introduced the concept of logical clocks. An assumption is made that the event of "sending" a message occurs before the event of "receiving" the same message. These assumptions naturally provide a partial ordering of events in the system, which is termed the *happened-before relation*. Denoted by "$\rightarrow$", the happened-before relation is precisely defined as follows:

1. $a \rightarrow b$

   * if $a$ and $b$ are events in the same process such that $a$ occurs before $b$, or

   * if $a$ is the sending of the message by one process and $b$ is the receipt of the same message by another process.

2. The relation is transitive: if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

3. if no ordering exists between two distinct events, then $a$ and $b$ are said to be concurrent.

For each process $P_i$, we assume a logical clock $C_i$ exists that assigns a number $C_i$(a) to an event $a$ in that process. For a system of logical clocks C to be correct, the following condition must hold.

1. For two events $a$ and $b$ in process $P_i$, such that $a$ comes before $b$, $C_i(a) < C_i(b)$.
2. if $a$ is the event of sending a message $m$ by process $P_i$, and $b$ is the receipt of $m$ by process $P_j$, then $C_i(a) < C_j(b)$.

Both the above conditions can be satisfied by implementing the clocks so that the following conditions hold:

- $P_i$ increments $C_i$ between any two successive events.
- if $a$ denotes the event of sending a message $m$ by process $P_i$, then the message $m$ bears the timestamp $T_m = C_i(a)$.
- When the message $m$ is received, process $P_i$ sets $C_j$ to a value that is greater than or equal to its present value and greater than $T_m$.

It is now possible to induce a total ordering on the collection of events in our distributed system. Event $a$ in process $P_i$ precedes (denoted by =>) event $b$ in process $P_j$ if and only if

1. $C_i(a) < C_j(b)$, or
2. $C_i(a) = C_j(b)$ and $P_i << P_j$, where the relation << is an arbitrary total ordering of the processes.

A simple way to implement the relation << is to assign a unique process number to each process and to define $P_i << P_j$ iff $i < j$.

## 2.5. The Model

A *Distributed algorithm* is a collection of automata, where one automaton is associated with each processor in the system. Given an instance $I$ of input size $n$ of a problem and an algorithm $A$ which solves the problem, the *message complexity* of $A$ on $I$ is the total number of messages generated by all the processors during the execution of $A$.

We consider an asynchronous point-to-point connected communication network. At each node there exists an independent processor. We shall be referring to this communication network as a distributed system. Each processor is assumed to have a distinct $id_i$, and a private set $S_i$ of values. Two processors can exchange information only through explicit messages on a communication medium; the communication provides a point to point communication capability (thus, at a given instance, a processor can send a message to one/more of its neighbors). The distributed algorithm is initiated at one or more nodes/processors (usually by a command given by the user or generated by a system operating at a site). A processor can send and/or receive messages only to/from processors that are adjacent to it. The communication links are bidirectional. Each processor does some local computation and sends out messages to its neighbors (some, all or none at all). We assume that it can receive messages from all its neighbors and that no message is lost in transit. All messages are guaranteed delivery within an arbitrary but finite amount of time. The order in which the messages are received by any node is immaterial for the execution of this algorithm.

The exchange of messages between two processors is *asynchronous* in that the sender always hands over the message to the communication subsystem and proceeds

with its local computation if any. A similar model of distributed computing may be found in [Chan82] and [A.Se83].

# Chapter Three

## SET MANIPULATION

### 3.1. Introduction

In this chapter, we discuss the problem of set manipulation in distributed systems. First, we shall look into the set intersection problem. Scheduling meetings among personnel is one of the most common tasks in an office environment. Each person can be considered to have a collection of *available slots of free time* in such a setting and the task is to find a slot in their intersection. We use the same scenario as provided in [Rama88]. The above problem can be abstracted as that of finding an element in the intersection of the sets.

Another situation where set intersection plays a vital role is in a distributed database system. Each processor in the system is given a column fragment from a global relational database. The only assumption being that the union of all the fragments will result in the original database. A query posed to the system is transmitted to all the sites. Each processor computes the solution to the query based on its local information. This solution is basically a set of tuples that satisfy the query based on the attributes of interest. The answer to the query is the intersection of all these individual sets possessed by each processor.

Now we shall formulate the set intersection problem in a distributed environment. Consider an asynchronous point-to-point communication network. Each node is an independent processor. Each processor $i$ in this network contains a set $S_i$. The problem is to find the intersection of all the sets in the network. We shall be referring

to the communication network as a distributed system. In a recent paper [Rama88] , an optimal distributed algorithm was presented to solve the set intersection problem with a message complexity of $\Theta(mn)$, where $m$ is the cardinality of the smallest sized subset and $n$ is the number of processors in the system. However, the decentralized algorithm and its complexity analysis are complicated. In the rest of this chapter we shall refer to the algorithm presented in [Rama88] as Algorithm A. In this paper, we present a distributed decentralized algorithm which uses lesser number of messages than required by algorithm A and furthermore, our algorithm is simpler and does not require each processor to know the Universal set $U$ (the set of all likely elelments). Our centralized and decentralized algorithms use at most $3.m(n-1)+3.(n-1)$ messages, whereas algorithm A proposed in [Rama88] uses $5.mn+2.n-4.m$ messages in the decentralized version. It will be evident that our algorithm is considerably simpler than the algorithm previously proposed to solve the set intersection problem.

## 3.2. The Model

An asynchronous network is a point-to point (or store and forward) communication network, described by an undirected communication graph $G=(V,E)$, where the set of nodes $V$ represents processors of the network and the set of edges $E$ represents the bidirectional non-interfering communication channels operating between neighboring nodes. There is no shared memory between processors in the system. Each node processes messages received from its neighbors, performs local computational, and sends messages to its neighbors, all in negligible time. Each processor is assumed to have a distinct $id_i$, and a private set $S_i$ of values. The communication complexity, $C$, is the worst case total number of elementary messages sent

during the execution of the algorithm. The time complexity, $T$, is the maximum possible number of time units from the start to the completion of the algorithm, assuming that the inter-message delay and the propagation delay of an edge is at most one time unit. The model assumed is a common model for communication networks [Awer84, Gall82].

## 3.3. Motivation for the Distributed Set Intersection Problem

Scheduling meetings among personnel is one of the most common tasks in an office environment. Each person can be considered to have a collection of *available slots of time* in such a setting and the task is to find a slot in their intersection. We use the same scenario as provided in [Rama88]. The above problem can be abstracted as that of finding an element in the intersection of the sets.

## 3.4. Description of Previous Work

We shall look into the decentralized version of algorithm A in [Rama88]. The basic idea behind the algorithm is as follows: the algorithm is divided into several *phases* and a possibly different processor directs the processing during each phase; in the first phase, R, the designated root of the underlying spanning tree of the network, finds a processor with the smallest sized set and passes the control to that processor while simultaneously reorienting the tree to be rooted at that processor; the root of the tree now simply sends its values one at a time down that tree. Each processor computes the size of the intersection of its set with the set received during that phase and sends this number to its parent (in a bottom-up fashion). Thus, at the end of each phase, a new root is found which has the smallest intersection with the set received

*during that phase.* The algorithm is terminated when all processors have responded with exactly the same number of values as the number of values sent by the root during that phase.

The process during each phase is as follows: each processor has a candidate set at the beginning of the first phase; the (new) root of the tree sends its values down the tree one at a time; each processor, after receiving these values, computes its intersection with this set and treats the intersection as the new candidate set. Each leaf processor sends (processor-id, size-of-candidate-set) to its parent. An internal processor receiving messages from all of its children finds the minimum of the numbers received. The root finally computes the minimum of all the processors. If this minimum is equal to the size of its own candidate set , then it sends a "terminate" message on the tree and moves into a final state, marking the candidate set as the intersection. Each of the other processors, after receiving the terminate message, marks its candidate set as the intersection and enters a final state. If, on the other hand, the number sent by one of the processors is smaller, then a processor with the smallest value is chosen as the next root and a message is sent to that processor. This message triggers all the processors on that path to change the directions of the edges on the path (so that the tree is reoriented) and the new root is selected to start the next phase.

## 3.5. The Proposed Algorithm

We assume that there exists a spanning tree in the network. Let P be the root of this tree. The first step in the algorithm is to find that processor which has the smallest

sized set. This is achieved as follows (by a standard process): P sends a "*Compute* min" message down the tree. Each processor upon receiving such a message simply forwards it to its children. A leaf processor upon receiving the message sends an ordered pair (Processor id, set-size) to its parent in the tree. An internal processor upon receiving messages from all its children computes the minimum set size among all of its children and itself and passes to its parent (processor-id, set size) corresponding to the minimum. Thus, P can compute the minimum size of the sets and the processor with that set (a tie being arbitrarily broken). The procedure *ComputeMin* uses exactly $2^*(n-1)$ messages.

Let the processor with the minimum sized set be R. We now reorient the tree so as to make R the new root of the tree. This process of reorientation of the parent child relationship in the tree takes no more than $n-1$ messages.

The root processor R now sends its entire set down the tree. The size of this message is at most $m$. Each internal processor when it receives this set will compute the intersection of the set received with its own set. It will then send a message containing the elements of this intersection down the tree to its children. When a leaf node/processor receives a message it computes the intersection of its own set with the set received from its parent. The leaf processor now sends this intersection set to its parent. An internal node upon receiving the intersection sets from all its children then computes the intersection of all the sets it received. This information is then sent to its parent and this process goes on until the root note receives the intersection sets from all its children. Upon receiving the messages (intersection sets) from all its children, the root node computes the intersection of all the sets in the system. This

information is then sent down the tree to all the processors. For more clarity, we now present a step wise description of the algorithm.

### 3.5.1 Description of the Set Intersection Algorithm

Each processor in the system has a set of elements $S_i$ which we shall refer to as its *Candidate Set* $C_i$.

1. Find the processor/node in the system/network which has the minimum sized set using procedure *Compute* min. (Uses exactly 2* ($n-1$) messages)
2. Reorient the tree such that the node selected in step 1. is made the root. (Uses exactly $n-1$ messages)
3. The root sends the elements of its candidate set down the tree to all its children.
   - 3.1 Each Intermediate node, upon receiving the set of elements from its parent, computes the intersection of its own set with the set received and then sends this newly computed set to all its children. This newly computed set now becomes its candidate set $C_i$.
   - 3.2 A leaf node upon receiving the elements of the set from its parent computes the intersection of its own set with the set received and sends the newly computed candidate set $C_i$ up the tree to its parent.
4. Upon receiving the sets from all its children, an intermediate node computes the intersection of all these sets received. It also forms a list of all the sets received along with the *id* of the processor from which it was received. It then sends this intersection set up the tree to its parent.

5. The root upon receiving the sets from all its children, computes the intersection of these sets. This set referred to as INT is the final intersection of all the sets in the system. The root also forms a list of all the sets received along with their processor *id*. This information is required to minimize the number of messages required to broadcast the final intersection set. ( Steps 3, 4, 5 use at most $2.(n-1).m$ messages)

6. The root then computes the set difference $D_i$, which is $C_i$ - INT for each of the sets received. If $|D_i| \leq |\text{INT}|$ then the root sends $D_i$ to processor $i$, one element at a time with a flag which tells that processor to discard the element from its candidate set, otherwise it sends INT to processor $i$. The intermediate nodes follow the same procedure as the root. (requires at most $m(n-1)$ messages)

## 3.6. Time and Message Complexity Analysis

In the above section, the sequence of steps used in the algorithm is outlined along with the message complexity for each step. In steps 3,4, and 5, during the execution of the algorithm each node receives exactly one message from its parent (other than the root node), the size of which cannot exceed $m$. Each child processor sends exactly one message to its parent, which in turn sends one to its parent and so on, until the messages reach the root. The message complexity of this entire process is no more than $2.(n-1).m$. This is because, there are exactly $n-1$ messages transmitted during that phase (there are primarily only two phases in this algorithm, the forward phase refers to messages being sent from the root down the tree and the backward phase to messages being sent from the leaves up the tree to the root) and the size of

any single message cannot exceed $m$. The last part of the algorithm deals with the broadcast of the result to all the processors which again requires at most $(n-1).m$ messages. Therefore, the overall message complexity of the entire algorithm is $3.(n-1)+3.m.(n-1)$.

The time complexity of the above algorithm is $O(mn)$. It is a common assumption in this model of computation that the transmission of a message takes one unit of time. Therefore in any tree computation which is basically top-down or bottom-up (root to leaf or leaves to root) the time complexity is $O(h)$, where $h$ is the height of the tree. Since the value of $h$ can be at most $n$, and the size of each message can be at most 5, the overall time complexity of our algorithm is $O(mn)$.

## 3.7. Comparison of Performance

Our algorithm is similar to the decentralized algorithm proposed in [Rama88]. In fig 3.1, we compare the performance of our algorithm with that of algorithm A. An illustrative example is provided in the Appendix. The proposed algorithm is better than the previous one [Rama88], for the following reasons.

- Message complexity analysis of our algorithm is simple.
- Our algorithm involves only one phase, whereas algorithm A in [ Rama88] can have many phases (the notion of a phase is as described in section 4) in the execution of the algorithm with *each phase requiring a reorientation of the tree*.

| Factors | Ramarao's Algorithm | Our Algorithm (Subbiah & Iyengar) |
|---|---|---|
| Number of Phases | At most $m$ | 1 |
| Reorientation of Tree | At most $m$ | 1 |
| Computation of Min | At most $m$ times | Once |
| Message Complexity Bound | $(5mn + 2n - 4m)$ | $(3m(n-1) + 3(n-1))$ |

Table 3.1 Table of Comparison

where $m$ is the cardinality of the smallest sized subset and $n$ is the number of processors.

- Our algorithm does not necessitate the fact that all the processors have knowledge of the universal set. *The computation of the message complexity of algorithm A also involves the fact that each of the processors has information about the universal set.*
- The complexity analysis of the proposed algorithm takes into account the number of messages needed for the reorientation of the tree. *The messages required for reorienting the tree at each phase has not been taken into account in the calculation of the bound for the message complexity of algorithm A.*

- The time complexity of our algorithm is $O(mn)$. In [Rama88], no mention is made of the time complexity for the distributed set intersection algorithm in the decentralized case. It is apparent from the nature of algorithm A, that the time complexity is of $O(m^2n)$, since there could be at most $m$ phases in that algorithm.

Algorithm A perfoms well when the number of phases is at most one or two. This is reflected by the fact that there is no need for broadcasting the results to all the processors at the completion of the algorithm. The major emphasis of this chapter is in the bound of the message complexity, which is considerably lower.

The proposed decentralized algorithm is message-optimal for the set-intersection problem according to the optimality criterion provide in section 2 of [Rama88].

## 3.8. Summary

We have presented a simple decentralized distributed algorithm for the set intersection problem which uses at most $3.m(n-1)+3.(n-1)$ messages, where $m$ is the cardinality of the smallest sized subset and $n$ is the number of processors. This improves the result of [Rama88] where, a centralized and a decentralized distributed algorithm using $3.m(n-1)+3.(n-1)$ and $5.mn+2.n-4.m$ messages, respectively was presented. The time complexity of our algorithm is $O(mn)$. The emphasis of this chapter is more on the simplicity of the proposed algorithm and the reduction in message complexity and time complexity.

It is interesting to note that there may be many cases where the intersection is empty. In these cases, we would be interested in the *most preferred* time slot

[Iyen90]. The next chapter provides an efficient algorithm to determine the mode of the distribution of the sets.

# Chapter Four

## DETERMINATION OF THE MODE

### 4.1 Introduction

In the previous chapter, we analyzed a situation where our point of interest was the intersection of sets in a distributed environment. Consider an office environment, where people communicate by electronic mail. If the employees wanted to elect a representative for a task, they could vote for one or more out of several candidates. The universal set would the set of all eligible candidates. Each individual set would be the list of candidates which the person voted for. In this case, we would be more interested in knowing who secured the maximum number of votes. This maximum is otherwise called the mode of the distribution. We shall define the exact model of the problem in the following section.

Consider the following problem: each processor $i$ in a distributed system contains a subset $S_i$ of a universal set $U$; we wish to determine the number of messages necessary and sufficient for each processor to know which element occurs with the highest frequency (MODE) in the entire network. A distributed algorithm with a message complexity of $O(kn)$ is presented to solve the above problem, where $k$ is the number of distinct elements and $n$ is the number of processors in the system.

### 4.2 Motivation

In the previous chapter we had considered those cases where the intersection of the sets is non-empty. But, what if the intersection is empty? We analyze this

situation where the intersection of the sets may be empty and propose distributed algorithms to find that slot of time *most preferred*. The mode of a distribution is that element which occurs with the highest frequency in a distribution. The mode, as we have already discussed, is of importance to those situations where the intersection is empty.

## 4.3 The problem

Electing a representative for a task is one of the most common requirements in an office environment. Each person is allowed to chose any number of representatives. The problem is to find the one with the maximum number of votes. The case of a tie being arbitrarily settled.

In our distributed algorithm, the messages would be of the type $\{n_1, S_1, n_2, S_2, .....\}$, where $n_i$ is the number of occurrences of each of the elements in set $S_i$. If there are $k$ distinct elements in the entire system, then the maximum size of each message would at most be $2k$, since there can be at most $k$ counts and since the sets are disjoint, the cardinality of the union of all the sets could at most be $k$. The message complexity of our algorithm is given by the product of individual message length and the number of such messages sent.

## 4.4 Input to the Algorithm

There are $p$ processors in the distributed system, each having a subset $V_i$ of elements from a Universal set $U$. The elements of $U$ need not ordered in any way for the sake of this algorithm. There may be at most $k$ distinct values in the system, such that $k \leq N = |U|$.

### 4.4.1 The Proposed Algorithm

*A centralized algorithm*

Let P be the prefixed central processor. Let us assume that there exists a spanning tree in the network. Efficient algorithms for constructing spanning trees have been proposed by Gallager et. al in[Gall83]. If this spanning tree is not rooted at P, then we can reorient this tree to be rooted at P. This requires no more than $p$ - 1 messages. The processor P then sends a message 'Compute Number of Occurrences' down the tree. Each processor upon receiving this message forwards it to all its children. A leaf processor, after receiving a 'Compute Number of Occurrences' simply sends a reply to its parent. This reply message is of the form $\{1, V_i\}$, since the elements in any set are unique. An internal processor upon receiving reply messages from all its children, computes the number of occurrences of all the individual elements in its subtree and sends a reply to its parent. This message will be of the form $\{n_1, S_1, n_2, S_2, .....\}$, where $n_i$ is the number of occurrences of each of the elements in set $S_i$. Ultimately, the processor P would have received replies from all its children, whereupon it computes the solution to the problem and sends a termination message down the tree.

### 4.4.2 Message Complexity Analysis

Since the size of any message may be at most $2k$, the total number of messages sent in the execution of the above centralized distributed algorithm is at most $\{(n - 1) + ((n - 1) * 2k) + (n - 1)\}$, giving rise to a message complexity of $O(kn)$.

## 4.5 A Decentralized Algorithm

Given the simplicity of the centralized algorithm, it is intriguing if there is a decentralized or fully distributed algorithm with the same message complexity. We show that there is indeed such a distributed algorithm. We also go on to show that the fully distributed algorithm performs better than the centralized algorithm.

*Algorithm*

As in the case of the centralized algorithm, we assume that there exists a spanning tree in the network. At this stage, it is assumed that the root of the spanning tree is known and that each processor in the network knows who among its neighbors is its parent and who its children. Now, we are set to start the *Find_Mode* algorithm.

We assume that the algorithm is initiated at all the processors. We can assume that the system at each site initiates the algorithm at that site.

The processors in the network can be of three types:

1. Leaf Node : there is only one neighbor in the underlying spanning tree, which incidentally happens to be its designated parent.
2. Internal Node : It is not a leaf node, but is any node other than the root node.
3. Root Node : The unique node designated the root in the underlying spanning tree.

*case* 1 :

Every leaf node $i$ will send a message to its parent which will simply be the set of elements in its set $V_i$. The size of this message can be at most $k$, where $k$ is the

number of distinct elements in the system. The message need not involve a count because the each of the elements in any one set is unique.

*case* 2 :

An internal node upon receiving messages from all its children will perform some local computation and send a message to its parent. This message will be of the form: $\{n_1, S_1, n_2, S_2, .....\}$, where $n_i$ is the number of occurrences of each of the elements in set $S_i$. The length of the message sent by any internal node may never exceed $2k$ because there are at most $k$ distinct elements in the system and they can at most all have unique counts.

*case* 3 :

The root node upon receiving messages from all its children will compute the mode. The root node then sends a termination message down the tree along with mode.

It is interesting to note that at this stage the root node has information about the number of occurrences of each and every element in the system. It also knows the Union of the all the individual sets and hence the non-occurrence of any element can be computed from the set difference with the Universal set $U$, if one such exists.

### 4.5.1 Message Complexity

The message complexity of algorithm described above is calculated by taking the product of individual message size and the number of such messages sent by the processors, which is at most $(n - 1) * 2.k$, where the size of any message is at most $2.k$.

Since, there is no *apriori* knowledge about the distribution of elements among the various sets in the system, the choice of the root node may significantly change the number the messages.

### 4.5.2 Proof of correctness

It is trivially true that Algorithm A1 terminates, because each processor in the system sends exactly one message. The root on receiving the messages from all its children then computes the Mode and then sends a terminate message to all the nodes. The tree structure of the communication network ensures that the interior nodes compute the mode on their individual subtrees, after receiving messages from their respective children. The leaf node/processor has a mode of one, where every element occurs only once.

**Theorem.**

Any algorithm that solves the mode detection problem generates $\Omega(kn)$ messages.

Any algorithm for the detection of the mode in a distributed system, requires each processor to send at least one message, except the root, in the scenario just considered. Since the distribution of the elements is not known a-priori, this message will have to contain the elements of the set in that processor. Since each processor may have at most $k$ distinct elements, the size of the message is of $O(k)$. Since at least $n-1$ such messages will have to transmitted, the order of message complexity is at least $\Omega(kn)$. The proposed algorithm uses only $n-1$ messages of size at most $2k$ each and successfully computes the mode, using $O(kn)$ messages. Therefore $O(kn)$ mes-

sages are necessary and sufficient to find distributively find the mode in a distribution of $p$ sets among $p$ processors in a distributed environment.

**Theorem.**

There exists a message optimal decentralized distributed algorithm to detect the mode of a distribution using $\Theta(kn)$ messages.

The proposed decentralized algorithm solves the mode detection problem using $O(kn)$ messages.■

## 4.6 Summary

We have shown that $\Theta(kn)$ messages are necessary and sufficient to determine the mode of a distribution in a distributed environment. Efficient decentralized algorithms were also presented and shown to be optimal. The mode detection algorithm also provides information about the Union of the sets.

In any Office environment, the employees may want to elect an individual to represent them for a particular task. This election process is nothing but voting for all those persons that a particular person would choose to elect for the task. The person (s) with the maximum number of votes is declared the representative. The vote could also be a weighted vote in which case the algorithm has to be modified slightly to make the correct selection.

# Chapter Five

## NETWORK RECOGNITION

### 5.1 Introduction

Distributed graph algorithms are interesting for many reasons. Several distributed algorithms for computing structured subnetworks (such as spanning trees and hamiltonian cycles) and others for understanding the communicational structures of problems, have been proposed in the past. These algorithms can be used for providing certain basic facilities such as routing and reconfiguration. Learning topological configuration in a distributed system is vital to the design of suitable fast algorithms to solve various computational problems. In the earlier chapter, we used a spanning tree to compute the intersection of sets in a distributed environment. This computation would have been redundant had the network been identified to be a tree in the first place.

In the recent past, several distributed algorithms relating to graph-theoretic problems, such as, shortest paths [Chen82, Rama86, Chan82a], minimum weight spanning trees [Gall83, Park81, Chin85], biconnected components [Chan82b], centers [Kora84], and knots [Misr82], have been proposed. Problems such as message routing, center finding etc, can be solved very easily by imposing certain restrictions on the structure of the network. It is shown in [Fred85] that problems such as shortest path, routing etc., are easy to handle in planar networks. In general, algorithms for general networks are expensive in terms of both time and message complexities,

whereas algorithms for certain topologies are much faster and easy to design. Message routing becomes easy to handle on mesh-connected or planar networks. There exist several problems in the literature which have been shown to be NP-Complete in [Gare79], but when restricted to certain graph structures result in optimal or efficient solutions. One such problem is the domination problem, which is NP-Complete on general graphs and several restricted graphs, but for which linear time solutions exist in tree structures. We shall present efficient distributed algorithms for computing the minimum dominating set in tree structures in chapter 7. This is the basic motivation behind designing distributed algorithms for recognizing the structure of a network. The design of efficient distributed algorithms for network recognition has many advantages, because knowledge of the structure of the network simplifies many problems. These recognition algorithms are basically to be used as *preprocessing* algorithms.

Network recognition algorithms basically perform the task of identifying whether a particular network is one of several known restricted structures. The problem of network recognition was first introduced by Ramarao [Rama89], where several known topologies of networks such as rings, trees, bipartite graphs, star graphs, complete graph structure networks are recognized using a single unified recognition algorithm. The problem of network recognition as stated in [Rama89] is as follows:

- Given a graph G(V,E) underlying a communication network, does its structure belong to the set = { tree, ring, bipartite, star, rectangular mesh, 2-reducible, outerplanar, planar, cactus, complete .....}. The recognition algorithm, as men-

tioned earlier should be treated as a preprocessing algorithm, whose time and message complexities should be considerably lesser than that the actual algorithm on general graphs. Therefore, a robust network recognition should satisfy two basic requirements.

1. The cost of the recognition algorithm should be considerably smaller than the cost of the actual algorithm on general graphs.

2. The algorithm should be a unified algorithm, upon whose termination it should be possible to determine if the structure of the network is one among the known restricted structures listed earlier.

## 5.2 Previous Results

The problem of network recognition was introduced by Ramarao [Rama89], where a unified distributed algorithm is proposed which when executing on a connected network G(V,E) determines whether G is a tree, ring, star graph, complete graph or a bipartite graph or none of them. The general pattern of the algorithm in [Rama89] is as follows:

1. Algorithm Initiation
2. Process Interaction
3. Local Detection
4. Global Detection

In a centrally initiated algorithm, the algorithm is usually initiated at a specific start node. This node referred to as the *central node/processor* starts the algorithm. It

sends a message to all its neighbors requesting them to detect the structure(s) of interest. A processor $p$ receiving such a message becomes active. Processor $p$ arbitrarily selects one of the processors which has sent such a message to it and treats it as the "parent". Processor $p$ sends an "accepted" message to its parent and "rejected" messages to all the other processors from which it received the message. Two case need to be considered depending on the nature of the local condition being checked: 1) the condition does not depend on some information available at a neighbor, and 2) the condition depends on such information. In the first case, $p$ determines immediately after receiving a message, whether its local condition is consistent with that of the structure being sought. If any inconsistency is detected, then it informs all its neighbors and terminates the transaction. If the local conditions are consistent, then it simply forwards the message to its neighbors other than those from which it has received the message previously. If it has no more neighbors to send to, it sends a "yes" message to its parent. It sends a "no" message if the local conditions are inconsistent. On the other hand, if the local condition to be checked depends on the information from some neighbor, then clearly $p$ cannot check the local condition until it receives the information from that neighbor. But $p$ cannot wait for that message without sending out any messages since that neighbor might be waiting for $p$ to send it a message before it can send the message to $p$. Thus $p$ forwards the initial request message it has received with the appropriate changes in it according to the protocol, to all its neighbors. Thus, it is guaranteed that $p$ eventually receives the message it is expecting from its neighbor since that neighbor is also following the same algorithm. $P$ then checks the local condition after it receives all the relevant information from its neigh-

bors of interest, and behaves exactly as in the first case thereafter.

It can be seen that a spanning tree rooted at the central processor has implicitly been constructed during the above process. A node receiving "yes" messages from all of its children sends a "yes" message to its parent only if its local conditions are also consistent. Otherwise, it sends a "no" to its parent and terminates the algorithm. When the answer is "yes", it waits until it receives another message from its parent before it terminates. The central processor, after receiving the responses from all of its children, makes the global decision of whether the structure(s) is (are) present or not.

The individual algorithms for each of the structures being considered are listed below.

- Recognition of a *Bipartite Graph*

  *Message sent:* "Color is, *a* where *a* is blue (black) if the color of the parent is black (blue).

  *Consistency condition:* Two messages with different colors are not received.

- Recognition of a *star*

  *Message sent:* Depth of a node ( distance from central processor)

  *Consistent Condition:* Either the depths of all the nodes from which the message is received is 0 or the number of neighbors is 1.

- Recognition of *Tree*

  *Message sent:* "Your Color is red"

  *Consistent condition:* Messages of this type are received from exactly on node.

- Recognition of a *Ring*

  *Message sent:* "Your Color is green"

  *Consistent condition:* Number of neighbors = 2.

- Recognition of a *Complete Graph*

  *Message sent:* Depth of a node, number of neighbors.

  *Consistency condition:* a) Depth of one of the nodes from which it has received the message is 0, and b) number of neighbors of the node = number of neighbors of its parent.

Thus, it is shown in [Rama89] that there exists a unified centrally-initiated algorithm which uses $O(e)$ messages of $O(\log n)$ bits each and $O(d)$ time to detect if the network is any one of a tree, ring, star graph, complete graph, or bipartite graph. It is also shown that the divide and conquer strategy proposed by Gallager et. al. [Gall83] can be used, to detect the structures mentioned earlier. This approach is generally referred to as the *inductive merge paradigm.* The cost of the decentralized algorithm is higher due to the cost incurred by the divide and conquer strategy. The time and message complexities of the decentralized algorithm are $O(n \log n)$ and $O(e+n \log n)$ respectively, where each message is $O(\log n)$ bits long.

## 5.3 Rectangular-Mesh recognition

In this section, we consider the problem of recognizing whether a given network is a rectangular-mesh. We present an efficient distributed algorithm with a message complexity of $O(N)$, where $N$ is the number of nodes in the network, which is an improvement of a previous algorithm in [Moha90] with a message complexity of $O(N \log N)$. The previous result for recognition of mesh connected networks is one using $O(N \log N)$ messages and $O(N^{1.6})$ time units [Moha90]. The bottleneck in that algorithm is the use of a breadth first search BFS tree [Awer87]. The proposed algorithm is constructive in nature and also assigns coordinates to the nodes of the network. The proposed algorithm does not use a BFS tree in the recognition process. Our algorithm also assigns labels (coordinates) to the nodes which enables each node to know its exact position in the grid. Throughout this section, we shall use the words mesh, rectangular-mesh and grid interchangeably.

### 5.3.1 What is a Mesh?

In the rest of the paper, we consider only the case of non-trivial networks with the number of nodes $N$, being at least four. The trivial case of a network being a single node or a path of nodes is not considered as it is not interesting.

*Necessary Conditions:*

A mesh network must consist of vertices with degrees 2, 3, and 4 only, and with exactly four 2-degree vertices. Let $D_2$, $D_3$ and $D_4$ denote the number of 2-degree, 3-degree and 4-degree nodes respectively. Let the dimensions of the mesh be $(m,n)$, where $N$, the total number of nodes in the network = $(m * n)$ =

$D_2 + D_3 + D_4$. The parameters of the mesh namely, $D_2$, $D_3$, $D_4$, $N$, $m$ and $n$ should satisfy the following conditions.

| | |
|---|---|
| Total number of nodes $N$ | $= m * n$ |
| $D_3 + D_4$ | $= N - 4$ |
| Number of 3-degree nodes, $D_3$ | $= 2 * (m + n - 4)$ |
| $D_3$ | $= 2 * (\frac{N}{m} + m - 4)$ |

Solving for $m$, we get

$$m^2 - m\,(\frac{D_3}{2} + 4) + N = 0 \qquad *$$

**Definition:** A non-trivial network is said to be an $(m, n)$ *mesh-looking* structure if (*) yields two positive (not necessarily distinct) integer solutions.

**Remark:** An $(m, n)$ *mesh-looking* structure need not be a rectangular-mesh.

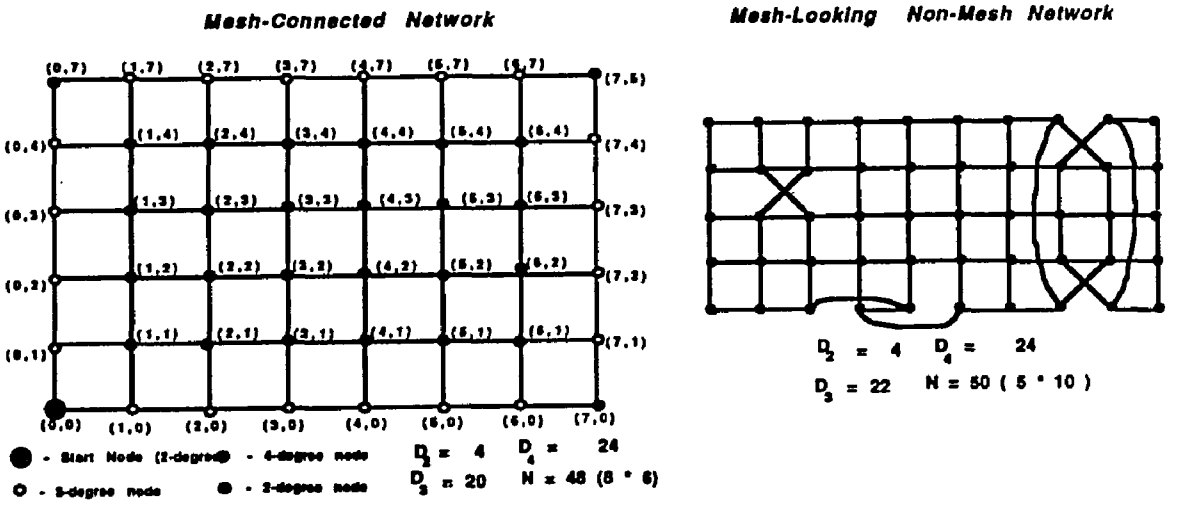



Fig: 5.1.

**Notation:** For $n \geq 1$, let $[n-1]$ denote the set $\{0, 1, \ldots n-1\}$.

*Sufficient Conditions:*

**Theorem:** An $(m, n)$ *mesh-looking* structure is a rectangular-mesh *iff* there exists an injective map $F : V \rightarrow [m-1] \times [n-1]$, such that for any $u, v \in V$ with $F(u) = (i, j)$ and $F(v) = (i', j')$, whenever, $uv \in E$ then

$$|i - i'| + |j - j'| = 1.$$

It can be easily verified, that except for symmetry, the mapping F is unique. For any 2-degree node $v$, $F(v) \in \{(0,0), (m-1,0), (0,n-1), (n-1,m-1)\}$. Once this is fixed, because of adjacency constraints, a 3-degree node can assume labels from the set { (1,0), (2,0) ... $(m-2,0)$, (0,1), (0,2), .. $(0,n-2)$, $(1,n-1)$, $(2,n-1)$, .. $(m-2,n-1)$, $(m-1,1)$, $(m-1,2)$ .. $(m-1,n-2)$}.

### 5.3.2 Description of the Previous Algorithm

The algorithm proposed in [Moha90] works in two phases. Our algorithm differs from the previous algorithm mainly in the second phase. The first phase simply involves determining whether the necessary conditions for a network to be a rectangular-mesh are met. At the termination of Phase I, two adjacent 2 degree vertices are located. They then initiate the second phase of the algorithm. A breadth first search is then performed with $v_1$ and $v_2$ (the two 2-degree vertices) as the roots of the BFS tree. It is this breadth first search that contributes to the increased message complexity of their algorithm. At each node the level number according to each of the two breadth first search trees is stored. Based on these level numbers, a consistency condition is then checked. If these conditions are satisfied then the network is mesh-connected, otherwise not.

## 5.4 Informal Description of the Proposed Algorithm

The algorithm consists of two phases. In the first phase, the necessary conditions for a network to be a mesh are checked, which is very similar if not the same as that of the algorithm proposed in [Moha90]. If these conditions are met, then the second phase strips the mesh layer by layer, each time leaving behind a mesh of lesser dimension, until an intermediate mesh structure of dimension (1,*), (*,1), (2,*) or (*,2) is identified. During this phase, nodes in the mesh network are assigned coordinates. A successful assignment of coordinates to the nodes, satisfying the adjacency criteria, ensures recognition of a mesh structure. The algorithm rejects any other structure.

### 5.4.1 Phase I

During Phase I of the recognition algorithm, the following tasks are performed.

1) Compute the values of $D_3, D_2, D_4, m, n, N$.

2) Check necessary conditions.

3) Send information about $m$ and $n$ to all nodes.

4) Instruct nodes to send *id* and degree to all their neighbors.

5) Select any 2-degree node and initiate Phase II.

In this phase, the values of the various mesh parameters are determined. For this we use an underlying spanning tree of the network (if no such one exists then we can construct one by a simple depth first search algorithm using $O(N)$ messages). We

shall now informally discuss the working of the algorithm. The root of this spanning tree generates a "Count" message, which has the following format:

$$(D_2, D_3, D_4, N, \text{Flag})$$

the flag field is Boolean (presence or absence of nodes with degree other than 2, 3, or 4. This message is trickled down to the leaf nodes, which return the message after updating the values of the various fields. An internal node receiving a return message from all its children updates (counts the number of 2-degree, 3-degree and 4-degree nodes in its subtree) the value of the various fields and sends it to its parent, and this process goes on until the root receives messages from all its children. At this stage the root node determines the values of the various parameters and checks whether the necessary conditions have been satisfied.

If any of the necessary conditions is violated, then the algorithm is terminated. The root node sends a message down the tree, containing the expected dimensions of the mesh $(m, n)$. This message is propagated by the internal nodes of the tree, until a leaf node is reached. When a node receives this message about the dimensions of the mesh, it sends information about its *id* and degree to every one of its neighbors in the network and also propagates the dimensions down to its children. Then, some 2-degree node is identified and a message is sent to that node to initiate the second phase of the algorithm. Hence at the conclusion of Phase I, all necessary conditions have been satisfied and the structure has been identified as a *mesh-looking* structure. Additionally, every node knows the *id* and degree of all its neighbors and the dimensions of the mesh $(m, n)$. The newly selected 2-degree node now starts Phase II.

### 5.4.2 Phase II

The algorithm described in this section is used only for the recognition of those mesh structures whose dimensions are $(m, n)$, where $m, n > 2$. A slight modification of the algorithm can be used to recognize the special case of an $(m, 2)$ or $(2, n)$ mesh structures. The algorithm for the recognition of mesh-structures with dimensions $(m, 2)$ or $(2, n)$ is presented in section 10.

Each node has 3 variables, namely, its coordinate (x, y) {set to (0,0) initially}, static degree $D_s$, and dynamic degree $D_d$ (initially = $D_s$ = no: of neighbors). Each node also maintains an active neighbor list $N_l$ of all its neighbors and their static degrees. The algorithm in this phase is constructive in nature. This phase is divided into many stages. At each stage, a layer of the mesh is peeled (much like an onion peel). This peeling of a particular layer affects the $N_l$ list and dynamic degrees of the nodes in the next inner layer. Stage I differs from the other stages in that $D_d = D_s$ for all the nodes participating in this stage. The peeling of a layer is broken into four different directions namely (+x, +y, -x, -y). These directions aid in the generation of coordinates for the nodes. The order of the directions is quite important, but it does not matter as to which direction is chosen first. The need for the direction will be clarified in the subsequent sections.

#### 5.4.2.1 Stage I of Phase II

At the beginning of Phase II, each node sets all the members of its neighborhood list to be ACTIVE. A node enters the DEAD state when its neighborhood list becomes empty. A successful termination of the algorithm results in every node in the

network reaching the DEAD state. A node updates its neighborhood list when a message, is either sent or received by that node. The process of updating is the deletion of that member from/to whom the message was received/sent. The process of peeling the outer most layer is initiated by the 2-degree node selected at the end of Phase I. This node sends a message to one of its 3-degree neighbors. This message has the following format:

{Message_Type, $id$, $S_{num}$, Coordinates($x_i$, $y_i$), Direction}

where $S_{num}$ denotes the stage number (= 1 in this case), and Dir can be any one of {+x, +y, -x, -y}. The actual message sent in this case would be

{"Propagate", $id$, 1, (0,0), +x }

When a 3-degree node receives this message, it sets its coordinates to (1,0), based on the coordinate of the sender(0,0) and the direction(+x). The rest of the description of stage I will be illustrated with an example as shown in Fig 5.2a.

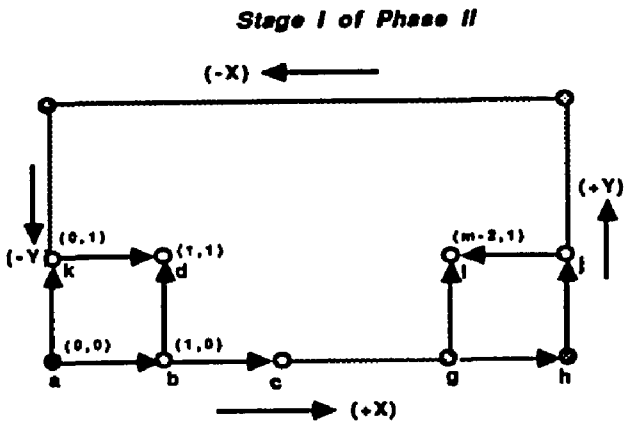



Fig. 5.2a

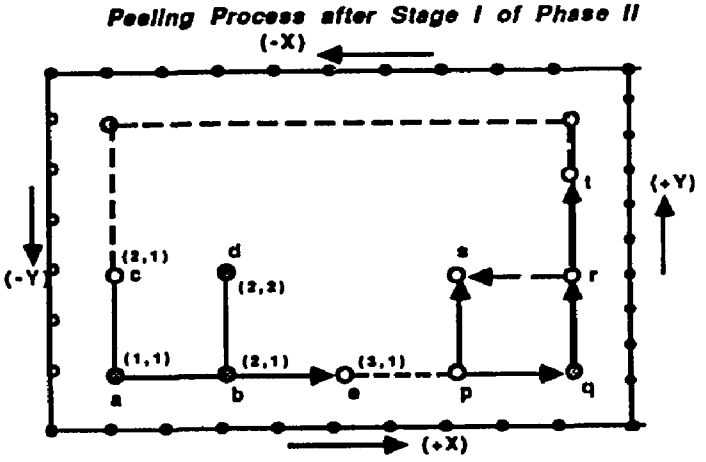



Fig. 5.2b

Node $b$ has received the message from node $a$. Node $b$ has two other neighbors namely, $c$ and $d$, which have a degree of 3 and 4 respectively. Node $d$ does not participate in the peeling process, but learns its coordinates based on the message it receives from $b$. The message sent by node $b$ to node $d$ is

("Propagate", $b$, 1, (1,0), +y)

Node $d$ on receipt of this message sets its coordinates to (1,1). In this stage all direction information is taken "as is" for the computation of the coordinates, whereas in subsequent stages the 4-degree nodes will have to interpret the directions.

Node $b$ sends the following message to $c$

("Propagate", $b$,1, (1,0), +x)

Node $c$ upon receiving this message sets its coordinates to (2,0) and propagates this message to its other 3-degree neighbor, who performs similar actions as $b$.

This process is continued until node $g$ as shown. Node $g$ does not have a 3-degree neighbor, other than the one from which it received the propagation message. Hence it sends the updated propagation message to its 2-degree neighbor. Node $g$

also sends a message to its 4-degree neighbor $i$, which sets its coordinates to $(m-2, 1)$. If a node receiving a message, already has its coordinates set, then the information from the message received is used for confirming its coordinates or for rejection. In case of a rejection, the node sends a special message to its parent in the spanning tree, to terminate the recognition process. In subsequent stages, a node simply sends a "Disagree" message to the originator of the message, who must decide whether or not to terminate the process. It is to be noted that node $i$ receives messages both from node $g$ and $j$. It is immaterial as to which message is received first.

Node $h$ upon receiving a propagate message from node $g$, sets its coordinates, and verifies the dimension of the mesh based on its coordinates and the stage number. Node $h$ also changes the direction from '+x' to '+y', and sends the following message to $j$.

("Propagate", $h$, 1, $(m-1, 0)$, +y)

This process continues until node $k$ sends a message to nodes $d$ and $a$. Node $d$ upon receipt of this second message, confirms its coordinates and initiates the second stage of Phase II. At each stage $i$, the node with coordinates (i-1, i-1) and $|N_l| = 2(D_d = 2)$, will initiate the peeling process of that stage.

### 5.4.2.2 Rest of Phase II

The second stage of Phase II is initiated by the node which has the coordinates (1,1) and a dynamic degree of 2. It is important to understand the fact that each node only knows the static degree of its neighbors. The nodes which already have a coordinate assigned in the previous stage, participate in the peeling process and the nodes in

the next layer learn and set their coordinates which will be used in the next stage. Hence peeling of one layer, is not completely independent from the peeling of the previous or the next layer. This transfer of information to the next layer ensures the proper working of the algorithm.

When a particular layer of the network is being peeled, the nodes in the next layer (4-degree nodes usually) learn their coordinates. They compute their coordinate value based on the values of the various fields of the received message. Since they are 4-degree nodes, they do not propagate this message. The propagation of the peeling message is done by the dynamic 3-degree and 2-degree nodes. Since these nodes do not propagate the message, they know that the direction field of the message received is wrong, and hence use the succ(Dir) to compute their coordinate values. These computed values are then confirmed, when the next layer is peeled when these nodes become 3-degree nodes are used to propagate the peeling message.

Every node upon receiving a message computes a coordinate value. If its coordinates are not already known, then the newly computed values are its coordinates. If its coordinates are already set, then the newly computed value should equal its coordinates. If the node is a 3-degree node, then it confirms its coordinates with the newly computes values and continues the propagation process. If it finds that the two values are not the same, then it recomputes the coordinates with succ(Dir) and then checks if its coordinates are equal to this newly computed value, as in the case of node $s$ in Fig 5.2b. If the values are found to be equal, then it sends a "Disagree" message to the sender indicating that there is a problem. If the values are found to be different, then this node recognizes that the network is not a rectangular-mesh and hence sends a

special "Reject" message to its parent in the spanning tree constructed during phase I. We shall explain the working of the algorithm by identifying 3 cases as shown in Fig 5.2b.

1. Node $a$ has $b$ and $c$ as active members of its $N_l$ list. Each of these nodes has a static degree of 4 and a dynamic degree of 3. Node $b$ has a coordinate of (2,1) and $c$ has a coordinate of (1,2). To ensure the same direction of peeling in this layer, as in the previous layer, only $b$ has to propagate the message. Node $a$ sends the following message to both $b$ and $c$.

   ("Propagate", $a$, 2, (1,1), +x)

   Node $c$ will reject the message since the computed coordinate does not match its coordinate. It then sends the following message to $a$, informing it about the disagreement in the computed coordinate values.

   ("Disagree", $c$, 2, (1,2))

   Node $a$ will ignore this message, since it expects to receive a "Disagree" message from one of its active neighbors. If a node receives "Disagree" messages from all its active neighbors, then it sends a special "REJECT" message to its parent in the spanning tree used in Phase I.

2. Node $b$ has two active members in its $N_l$ list namely $d$ and $e$, both with a static degree of 4. Node $e$ has a dynamic degree of 3, whereas $d$ has a dynamic degree of 4. Node $b$ sends the following message to both $d$ and $e$.

   ("Propagate", $b$, 2, (2,1), +x)

   The interpretation of this message by $d$ and $e$ is quite different.

'*d*' Since *d* does not have its coordinates set, it learns about the value of its coordinates from the message it received. This computation is not the same as in Phase I. The message received by *d* is a propagate message, but its dynamic degree is not 3. So it assumes that this message was wrongly sent. To compute its coordinates, *d* chooses the direction following that which was received in the message, from the set {+x, +y, -x, -y}. In this case the direction chosen would be +y. Hence *d* would set its coordinates to (2,2), and send a reject message to *b*.

'*e*' Node *e* already has its coordinates set, and simply confirms that the computed value is the same as its coordinates. Then *e* follows the same procedure as *b*.

3. Node *s* receives two messages, one each from *p* and *r*. The message received from *p* is (*p*, 2, (*m*-3, 1), +x). Since *s* is 4-degree node, it computes its coordinate value with the direction changed to +y, (succ(+x)). The message received from *r* is (*r*, 2, (*m*-2, 2), +y). At this point in time, *s* would have a dynamic degree of 3, since it has already received the message from *p*. So it computes its coordinate value with the same direction field value as that received. It tries to confirm its value, but when it finds that the computed value is different from its previously computed coordinates, it recomputes the new value with succ(dir). It is immaterial for the successful execution of the algorithm as to which of these two messages reaches *s* first. Node *s* learns its coordinates from the first message and confirms this with the second message. If the learned value and the

computed value are not the same, then a special "REJECT" message is sent to its parent in the spanning tree. The algorithm is then terminated. The exact working of the algorithm at each node upon receiving a message is provided in section 5.

During any stage of the peeling process, there exist 4 nodes, whose dynamic degree is 2. One of these initiates the algorithm for that stage/layer. The other three perform the following functions.

- The direction field of the next propagate message is changed to that which appears next (successor) in the list {+x, +y, -x, -y}.

- The current dimensions of the mesh are confirmed based on its coordinate value and the stage number.

This process of peeling is stopped when a degenerate intermediate mesh structure of dimension (1,*), (*,1), (2,*) or (*,2) is identified. This decision is made by the two degree node which initiates the peeling process at that stage, since it has knowledge of the current dimensions of the mesh. These special trivial structures will be recognized using a slightly different algorithm. In the following section, a pseudo code version of the algorithm is presented. The algorithm is initiated at a particular 2-degree node. Other nodes in the system are initially INACTIVE. When a node receives a message, it executes the following algorithm. Messages have the following format in general:

(Mesg_Type, *id*, *Stage_num*, Coordinates (*x*,*y*), *Direction*)

Messages are treated as record structures for the purpose of the description of the algorithm. The field Mesg_Type can take the following values:

**"Propagate"**

The peeling process in any stage is propagated from one node to another using the "Propagate" message.

**"Disagree"**

When a node receives a "Propagate" message and finds that the computed value of its coordinates does not agree with its learned value, then it sends a "Disagree" message to the node from which the original message was received.

**"REJECT"**

When a node is able to determine that the structure is not a rectangular-mesh, then it sends a special "REJECT" message to its parent in the spanning tree. This message indicates that the algorithm is to be terminated and hence this message is propagated by its parent up to the root.

The following code is executed by a node $i$ receiving a message $m$ from its neighbor. Each node has the following local variables:

| | |
|---|---|
| $(x_i, y_i)$ | Coordinates of the node |
| set | True if coordinates have been set, else false |
| $N_{dis}$ | Number of disagree messages received. |
| $N_l$ | A list of neighbors, initially all active. |

(M, N) Dimensions of the Mesh.

**Procedure Recognize_Mesh ( $i, m$);**

```
        i : id;
    m : message;
/* The following code is executed by a node i receiving a message m from its neighbor. */
begin

Case m.type      of

"Propagate":
        If stage 1 then
        begin
            If Coordinates set then
                If old value ≠ new computed value then
                    send (" REJECT ", P(i));
                else
                    remove sender from active neighbors list
                    If (|N_i| = 2) and (x_i, y_i) = (m.stage, m.stage ) and NOT degenerate then
                    /* checking for an intermediate case of (2,*) or (1,*) */
                        for every neighbor j do
                            Send ( "Propagate, i, m.stage + 1, +x );
                            remove recipient from active neighbors list
                            { Start Next Stage }
            else /* coordinates not set */ Begin
                Compute new coordinates (same direction)
                Case dynamic degree Of
                3 :  remove sender from active neighbors list
                     for every neighbor j do
                     Case degree of neighbor j Of
                         2, 3 : send ( "Propagate", i, 1, (x_i, y_i), m.dir);
                                remove recipient from active neighbors list
                         4    : send ( "Propagate", i, 1, (x_i, y_i), SUCC (m.dir));
                                remove recipient from active neighbors list
                     end; { Case Statement - N_i(i)(j).degree }
                4 :  remove sender from active neighbors list
                2 :  remove sender from active neighbors list
                     To your neighbor do
                     Send ( "Propagate", i, (x_i, y_i), SUCC (m.dir));
                     If Not Dimension_Confirm then
                         send (" REJECT ", P(i));
```

```
                    end; { Case Statement - |N_i| }
                end;
            end
            else  { Stage No: ≥ 2 }
                If coordinates not set then
                    If |N_i| = 4 then
                        (x_i, y_i) = Compute ( m.(x, y), SUCC (dir));
                        Send ( "Disagree", m.id);
                        remove sender from active neighbors list
                    else
                        Send ("REJECT", P(i));
                else { Coordinate Set During Previous Stage }
                begin
                    If (x_i, y_i) = Compute ( m.(x, y), m.dir) then
                        remove sender from active neighbors list
                        for every active neighbor j do
                            Send ( "Propagate, i, m.stage, (x_i, ~y_i), m.dir );
                            remove recipient from active neighbors list
                            { Continue Same Stage }
                    else
                    If (x_i, y_i) = Compute ( m.(x, y), SUCC (m.dir)) then
                    begin
                        Send ( "Disagree", m.id);
                        { When current dimension becomes (2,*) }
                        If |N_i| ≠ (2 or 3) then
                        remove sender from active neighbors list
                        If |N_i| = 2 then
                            If (x_i, y_i) = (m.stage, m.stage) and NOT degenerate then
                                for every neighbor j do
                                    Send ( "Propagate, i, m.stage + 1, (x_i, ~y_i), +x );
                                    remove recipient from active neighbors list
                                    { Start Next Stage }
                        else If (degenerate) then
                            /*  An intermediate mesh of dimension
                              (1,*), (*,1), (2,*) or (*,2) is identified.   */
                            Recog_degenerate;/* in the next section*/
                    end
                    else
                        send ("REJECT", P(i));
                end; { End PROPAGATE }
"Disagree":
            N_dis = N_dis + 1;
            If N_dis = 2 then
                Send ("REJECT", P(i));
"REJECT":
```

```
            If not root(Spanning Tree) then
                  Send ("REJECT", P(i))
            else
                  Start ( Terminate ( REJECT ));
end;  { End of Case Statement }
```

The following procedure is followed when a degenerate case is identified during the recognition procedure.

Note: *These procedures are used only when an intermediate mesh of dimension (1,*), (*,1), (2,*) or (*,2) is identified.*

The computation of the coordinate value is performed locally by each node upon receipt of a message, by using the procedure "Compute".

---

**Procedure Compute** $(x,y)$**, dir);**

```
      input : Sender's Coordinate value and direction
output : New coordinate values

begin
      Case dir of
      '+x' : return ( (x+1,y));
      '+y' : return ( (x,y+1));
      '-x' : return ( (x-1,y));
      '-y' : return ( (x,y-1));
      end;
end;
```

---

**Procedure Recognize_Intermediate_(1,*)_Mesh** $( i, m)$

```
      i : id;
    m : message;
begin
    If coordinates are not set then
        send ("REJECT", P(i));
    else
    If message type = "X dimension = 1" then
        remove sender from active neighbor list;
        If active neighbor list not empty then
            send to unique active neighbor
            ("X dimension = 1", i, m.stage, (x_i, y_i), m.dir);
```

```
        else
            Send ("FINISH", P(i));
end;
/* A similar procedure is followed if the "Y dimension is 1"   */
```

---

**Procedure Recognize_Intermediate_(*,2)_Mesh ( $i$, $m$);**

```
begin
 If coordinates not set then
      Send ("REJECT", P(i));
 else
 If message type = "X dimension = 2" then
      If m.(x,y) ∈ { } then
           To both active neighbors do
           send ("X dimension = 2", i, m.stage, (x_l, y_l), NO Direction)
           remove sender from active neighbor list;
           If (active neighbor list is empty) and (x_l, y_l) = (M - m.stage, m.stage - 1) then
                send ("FINISH", P(i));

end;
/* A similar procedure is followed if the "X dimension is 2"   */
/* The reporting node has a coordinate of (M - 1 - m.stage, N - 1 - m.stage)   */
```

---

### 5.4.3 Termination of Recognition

The algorithm is terminated when a message of type "FINISH" is received by the root of the spanning tree. If the root node is in the "DEAD" state then it sends a "check" message to all its children. Any node receiving the "check" message will either propagate it down to its children, if its state is "DEAD" i.e. its coordinates are set, or send a "NO" message to its parent. When the leaf nodes receive this "check" message, they send a "yes" if the state of the node is "DEAD" and a "NO" otherwise.

If a node receives a "NO" message then it propagates this message to its parent. If all children send a "yes", then it propagates a "yes" to its parent. If the root node receives a "yes" from all its children, then the network is a rectangular-mesh.

## 5.5 Proof of Correctness

Any structure accepted by the algorithm is a rectangular-mesh structure. The algorithm ensures that the necessary conditions for the network to be mesh-connected, are satisfied. Now, it is sufficient to label the nodes with coordinate values, which satisfy the sufficiency conditions, as specified in the definition of a mesh. Every node receives its coordinate value from a neighboring node which already has its coordinates set. A node does not change its coordinates during the execution of the algorithm. First, we need to prove that no two nodes have the same coordinate value.

Let us consider the case of a node with coordinates $(p, q)$. No neighbor of this node has the same coordinates. Without loss of generality let us assume that the direction of propagation is +x. Any node receiving its coordinates in the same direction has the first coordinate element greater than $p$. For this coordinate to decrease two changes in direction should take place. But the first change in direction would change the second coordinate element. With this simple argument we can establish that no two nodes can have the exact same coordinate values. □

From the definition of a mesh (section 3), it is apparent that apart from symmetry conditions, there exists exactly one mapping from $V \rightarrow [m-1] \times [n-1]$. Therefore, the failure of our algorithm to assign labels to all nodes in the network signifies the fact

that the structure in question was not a mesh-connected structure in the first place.

## 5.6 Informal Discussion of (2,*) Mesh Recognition

It can be seen that the algorithm described in the previous sections does not recognize the specific case of a (2,*) or (*,2) mesh. Here, we shall describe a different algorithm by illustrating it with an example as shown in Fig 5.3.

The algorithm will recognize a mesh structure, by assigning coordinates to the nodes, just as we did in the previous algorithm. If during the process of assigning coordinates, a non-mesh structure is recognized, then the algorithm is terminated. The algorithm works by assigning coordinates to two nodes at a time. We shall give an informal description of the working of the algorithm.

It is assumed that all nodes know the *id* and degree of their neighbors. During phase I, all nodes also know the expected dimensions of the mesh. The only possibility of the existence of a non-mesh structure, is shown in Fig 5.3. The algorithm basically checks to see that there are no "cross-connections".

To start with, a 2 degree node is identified, which starts the recognition process, by setting its coordinates to (0,0), and sending a message to its unique 2-degree neighbor. We shall describe the rest of the algorithm by the example shown in Fig 5.3.

Node $f$ upon receiving a message from node $a$, sets its coordinates to (0,1), and sends an "accept" message back to node $a$. The next step is to assign coordinates to nodes $b$ and $g$, and then to nodes $c$ and $h$, and so on ...

Node $a$ sends the message ("New", 1, (0,0), $a$) to node $b$. When node $b$ receives

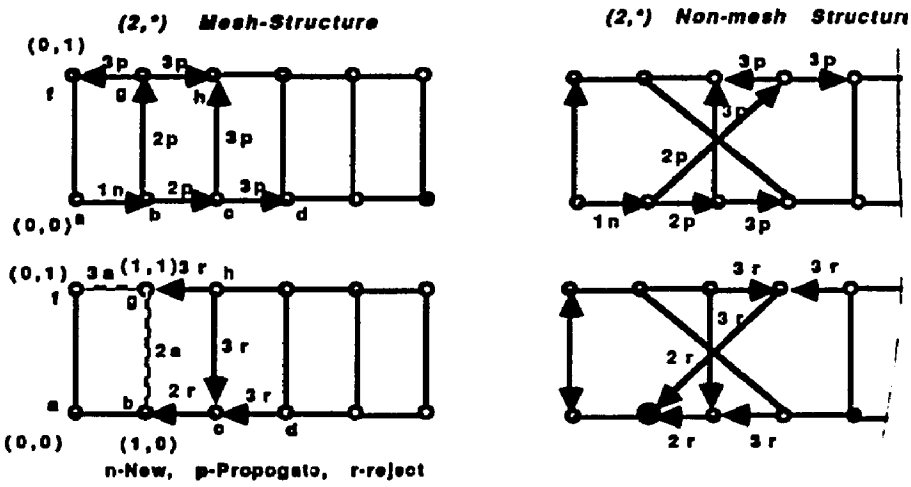



Fig 5.3.

this message, it sends a propagate message ("Propagate", 2, (0,0), *b*) to nodes *g* and *c*. Node *c* upon receiving this message propagates it to its active neighbors *d* and *h*, by sending the message ("Propagate", 3, (0,0), *c*) to them. A node receiving a message with a distance field of 3 does the following.

1. If its dynamic degree is 2 and that it has its coordinates set, then it sends an "accept" message to the sender.

2. Otherwise, it sends a "reject" message to the sender.

In this case, node *c* receives two "reject" messages, upon which it propagates this "reject" back to node *b*. Node *g* receives a "reject" from *h* and an "accept" from node *f*. Since node *g* expects to receive only one accept message, it sends an accept to node *b*, after removing *f* from its active neighborhood list, and setting its coordinates to (0+1, 0+1). Node *b* upon receiving an "accept" from *g*, sets its coordinates to (0+1, 0), and sends the message ("New", 1, (1,0), *b*) to node *c* and the process goes on,

until node $x$ receives a message of type "New", where it sets its own coordinates and sends a special message to its unique 2-degree neighbor which sends a "Finish" message to its parent in the spanning tree used in Phase I.

In the case of non-mesh structures, both nodes $c$ and $g$ would have received 2 "reject" messages each and would in turn propagate "reject" messages to node $b$. Node $b$ after receiving two "reject" messages would in turn send a "reject" message to node $a$, which would terminate the recognition recognition process.

The rest of the algorithm proceeds along the same lines as described above and hence needs no more explanation.

## 5.7 Complexity Analysis

First, we shall establish a lower bound on the number of messages needed to recognize the structure of a network. Any algorithm that recognizes the structure of a network requires at least $\Omega$(N) messages, where $N$ is the number of nodes in the network. This is true because each node has only local knowledge of its neighbors, and hence at least one message needs to be sent by every node, even to count the number of nodes in the network.

The algorithm presented in this paper to recognize rectangular-mesh connected networks uses $O(N)$ messages, and has a time complexity of $O(N)$. The algorithm works in two phases. In the first phase, it uses $O(N)$ messages to construct an underlying spanning tree and then uses at most $2N$ messages to compute the values of $D_2$, $D_3$ and $D_4$ and the dimensions of the mesh. In the second phase of the algorithm, each node sends at most 3 messages, where the size of the message is fixed and not depen-

dent on the size of the network. The final phase of terminating the algorithm uses at most $2N$ messages. Therefore the total message complexity of the algorithm is $O(N)$.

Assuming that every message consumes one time unit to reach its destination, it can be easily shown that the algorithm has a time complexity of $O(N)$.

| | Previous Algorithm | Our Algorithm |
|---|---|---|
| Time complexity | $O(N^{1.6})$ | $O(N)$ |
| Message Complexity | $O(N \log N)$ | $O(N)$ |

Table 5.1 Table of Comparison

## 5.8 Recognition of Outer-Planar Graphs

Here, we present a scheme based on the sequential algorithm for recognizing outer-planar graphs proposed by Wagers wagers We shall not provide a detailed description of the algorithm, but merely the technique used and the message complexity. The algorithm proposed by Wagers uses an edge-coloring technique and deletes vertices of degree less than or equal to two, one at a time. We use a similar technique for the distributed algorithm. Here, we shall describe the steps involved in recognizing 2-reducible graphs, which can then be extended to recognizing outer-planar graphs.

### Basic Definitions

The following definitions are directly reproduced from [ Wage ].

- A graph is planar if it can be drawn in the plane without crossing edges.
- A graph is outer planar if it can be drawn in the plane, such that, all vertices lie on the same face (outer-face) and no edges are crossed.
- A graph G = (V,E) is 2-reducible (2-red) *iff*

  1. $E = \phi$ or
  2. $v \in V$ with $deg(v) \leq 1$ and $G_v = G - \{v\}$ is 2-red or
  3. $v \in V$ with $deg(v) = 2$ and neighbors $v_1$ and $v_2$ and $G_v = (G - \{v\}) \cup (\phi, \{\{v_1, v_2\}\})$ is 2-red.

  Basically, $G_v$ arises from the reduction of the vertex $v$.

**Theorem 5.1 (Theorem of Chartrand and Harray)**

A graph is outer planar iff it has no subgraph homoemorphic to $K_4$ or $K_{2,3}$.

For the basic definitions of graph theoretic concepts, the interested reader is referred to Harary [Hara69] The recognition of outer-planar graphs is directly linked to the recognition of 2-reducible graphs, since an outer planar graph is always 2-reducible.

**The Methodology**

Each node maintains the following information in its local memory.

1. The list of neighbors and their IDs.
2. The list of virtual neighbors. (caused by reducing 2-degree vertices)

Initially, all nodes with a single neighbor send a message to their unique neighbors requesting them to delete their ID from the neighbors active neighborhood list.

Hence, without loss of generality, we could assume that the graph under question does not have any 1-degree vertices. Let the number of 2-degree vertices in the graph be $k$. Then, we present an $O(kn)$ algorithm to deduce whether the given graph is 2-reducible or not. We shall illustrate the steps involved in the algorithm with the help of examples. The steps involved in recognizing 2-reducible graphs:

- A 2-degree node reduces itself by sending messages to both its neighbors. The information sent to each neighbor is the ID of the other neighbor. If any of these neighbors is also a 2-degree node then a chain is established. By using standard techniques, we can reduce all such chains by using $O(p)$ messages, where $p$ is the number of nodes participating in the chain. At the end of the reduction of a chain the two end nodes possess knowledge about to whom they are virtually connected.

Obviously, only one of two end nodes of a reduced 2-degree node can itself become 2-degree, after the reduction. If both the nodes are not 2-degree nodes, then a message is sent up the spanning tree, to check if there is a dead lock. So every 2-degree node after reduction either creates another 2-degree node or the reduction process stops with that node when a check is made to determine if there are any other 2-degree nodes that can reduce. If there are none and there exist any unreduced vertex then we conclude the structure was not 2-reducible. If all the vertices have been reduced and have zero degree, then we have a 2-reducible graph.

To recognize an outer planar graph, an additional constraint is added with each reduction. Each edge initially is given a label from the set ( cross, bridge, out). The description of the algorithm and the proof of correctness is presented in [Wage]. It

can be shown that using the methodology described we can recognize outer-planar and 2-reducible graphs using at most $O(kn)$ messages. $k$ is the number of isolated 2-degree nodes in the graph. An isolated 2-degree node is one that does participate in a chain of 2-degree nodes.

We are sure that the complexity of the algorithm can be improved to achieve an algorithm whose message complexity is linear in the number of nodes. Another factor is to include all these algorithms in the unified scheme presented by Ramarao in [Rama89].

## 5.9 Summary

We have presented message optimal $O(N)$ distributed algorithms to recognize rectangular-mesh structured networks. This algorithm is an improvement over a previous presented in [Moha90], as shown in the table above. The algorithm not only recognizes mesh-connected networks but also assigns coordinate labels to each node, which can used for efficient routing. The algorithm described here can be added to the unified optimal distributed algorithms presented in [Rama89] to recognize if a network is a tree, ring, star graph, complete graph or a bipartite graph. Recognition algorithms for other classes of graphs, namely, planar graphs can be constructed under the same framework provided for outer-planar and 2-reducible graphs.

# Chapter Six

## $r$-DOMINATION IN TREES

### 6.1. History

The earliest ideas of dominating sets, it would seem, date back to the origin of the game of chess in India over 400 years ago, in which one studies sets of chess pieces which cover or dominate various opposing pieces or various squares of the chessboard.

The most common definition given of a *dominating set* is that it is a set of vertices $D \subseteq V$ in a graph $G=(V,E)$ having the property that every vertex $v \in V - D$ is adjacent to at least one vertex in $D$. The *domination number* $\gamma(G)$ is the cardinality of a smallest dominating set of $G$.

In more recent times the Eight Queens and Five Queens problems rekindled interest in dominating concepts, e.g., in the books of Ahrens in 1901 and Konig in 1936. Finally with the publications of the books by Berge in 1958 and Ore in 1962 the topic of domination was given formal mathematical definition. Cockayne and Hedetniemi began to study it and ultimately published a 1975 survey of the results that had been obtained by that time. This brought the subject sufficiently into focus to set research on a much larger scale into motion.

In the last 15 years, over 250 papers have been published on the subject. The algorithmic study of domination has exploded onto the scene even more suddenly than the theoretical study of domination. Perhaps the first domination algorithm was an

attempt by Daykin and NG in 1966 to compute the domination number of an arbitrary tree. Cockayne, Goodman and Hedetneimi apparently constructed the first domination algorithm for trees in 1975 and, at about the same time, David Johnson constructed the first [unpublished] proof that the domination problem for arbitrary graphs is NP-complete.

## 6.2 Introduction to $r$-domination

The problem of finding a minimum r-dominating set for a graph $G=(V,E)$ is that of selecting the smallest subset $D_r$ such that every other vertex of $G$ is at a distance $r$ or less from some vertex in $D_r$. In this chapter we present optimal distributed algorithms for determining the minimum $r$-dominating set when G is a tree. We introduce the problem of reliable $r$-domination in the next chapter. In chapter 7, we also present optimal unified distributed algorithms for the total, and reliable r-Domination problem on tree structures with a message and time complexity of $O(n)$, where $|V|=n$.

The dominating set is used in solving facility placement problems in networks, where each node represents a customer and/or a potential site for placement of a facility. A feasible solution corresponds to a set of facilities located at $D \subseteq V$, such that each customer is adjacent to or at a distance at most $r$ to some facility. Several variations of the domination problem have been studied in the past. A dominating set $D$ of a graph $G$ is *independent* if the nodes in $D$ are pairwise non-adjacent, *connected* if the subgraph $G$ [$D$] induced by $D$ is connected, and *total* if $G$ [$D$] has no isolated node.

There are many applications for the $r$-dominating set problem. We shall concentrate on its importance in communication networks, specifically with respect to

distributed systems. We shall also confine ourselves to finding a dominating set in a tree rather than a general graph. The domination problem for general graphs has been proved to be NP-complete in Karp [ Karp72]. The minimum $r$-dominating set is used in networks for placement of special purpose resources, such that every other node in the network has access to these resources within a distance of $r$. Other issues like the reliability and quality of communication links necessitate the identification of such a set of nodes with a special property. In a distributed system, the communication between processors is usually restricted to a spanning tree of edges. The communication subsystem for the purpose of execution of distributed algorithms is a tree. It is for these tree structures that we are interested in finding dominating sets.

Cockayne, Goodman, and Hedetniemi [Cock75] present a (linear) sequential algorithm for finding a minimum $D_r$. A *minimum* $D_r$ is one with the fewest possible vertices. A *minimal* $D_r$ is one such that no proper subset of it is also an $D_r$. Slater [Slat76] also considers the r-Domination in graphs and provides efficient sequential algorithms for these problems.

Until now, no attempts have been made to present distributed algorithms for domination problems in graphs. The problem of $r$-domination also lends itself to the solution of $p$-center problems as illustrated in [He90, Radh90]. Here, we present a message optimal distributed algorithm for computing the minimum $r$-dominating set on trees.

This chapter is organized as follows. Section 6.3 provides a brief insight into the model of the distributed system that we base our algorithm upon. We discuss the format of messages in section 6.4. In section 6.5, we discuss the basic methodology of

the r-dominating set algorithm. Section 6.6 gives a detailed description of our distributed algorithm and its complexity and correctness proofs. A summary is presented in section 6.9.

## 6.3 The Model

An asynchronous network is a point-to point (or store and forward) communication network, described by an undirected communication graph $G = (V, E)$, where the set of nodes $V$ represents processors of the network and the set of edges $E$ represents the bidirectional non-interfering communication channels operating between neighboring nodes. There is no shared memory between processers in the system. Each node processes messages received from its neighbors, performs local computational, and sends messages to its neighbors, all in negligible time.

Several independent processors exist in the system, each with a distinct *id*. Two processors can exchange information only through explicit messages on a communication medium; the communication subsystem provides a point to point communication capability (thus, at a given instance, any processor can be sending/receiving a message to/from any/all of its neighbors). A processor can send and/or receive messages to/from processors that are only adjacent to it. The communication links are bidirectional. The distributed algorithm is initiated at one or more nodes/processors (usually by a command given by the user or generated by a system operating at a site). Each processor does some local computation and sends out messages to its neighbors (some, all or none at all). We assume that all messages are guaranteed delivery within

an arbitrary but finite amount of time, and that no message is lost in transit.

The exchange of messages between two processors is *asynchronous* in that the sender always hands over the message to the communication subsystem and proceeds with its local computation if any.

Before we describe the algorithms, we shall present the format of messages that are transmitted during the execution of the algorithm and the meaning of the various components of the message wherever it is required.

## 6.4. Message Format

Communication between processors is via the communication subsystem, by sending and receiving messages. Every processor has its own local memory and uses many variables to keep track of the local computations. During the execution of these algorithms, we assume that each processor has a unique *id*. Each processor knows the *id* of its parent in the tree. The purpose of these algorithms is to determine the *id* of the nearest dominating vertex and the distance from that dominating vertex, to any vertex *i*. These are computed and stored in two variables which we shall refer to as $ID_{near}$(i) and $D_{near}$(i) respectively.

A message is treated as a record containing certain specific fields. The general format of messages used in the proposed algorithms is as follows:

$$\Phi = (ID, \text{Message Number, Destination, } ID_{near})$$

- *ID* stands for the *id* of the node/vertex sending the message, also referred to $\Phi.id$.

- The message number of a message can be positive, negative or zero. If the message number in a message received by a node $j$ from node $i$ is *negative* say -p (*seen*), then it indicates that node $j$ is within a distance of p from the nearest node in its subtree, which belongs to the dominating set.

- If the message number in a message received by a node $j$ from node $i$ is *positive* say +p (*looking for*), then it indicates that there exists at least one node, belonging to the subtree containing node $i$, which is yet to be *satisfied* and which is at a distance of p from node $j$. A node is said to be *satisfied* if it is at a distance of $r$ or less from any node in the current dominating set.

- If the message number in a message received by a node $j$ from node $i$ is 0 then it indicates that the nodes in the subtree containing node $i$ are all *satisfied*, and the nearest dominating vertex in that subtree is at a distance of $r$ from $i$.

- If the message number in a message received by a node $j$ from node $i$ is $r$, then node $j$ is said to be *Critical Positive*, ie node $j$ should be included in the dominating set.

- If the message number in a message received by a node $j$ from node $i$ is $-r$, then node $j$ is said to be *Critical Negative*, ie node $j$ is the last node that can ever be satisfied by any dominating node in the subtree containing $i$.

- $ID_{near}$ refers to the *id* of the nearest dominating vertex. Every node has two variables namely $ID_{near}$ and $D_{near}$ which it updates during the execution of the algorithm. A node propagating a positive message number will have its $ID_{near}$ set to 0, since it is still looking for a dominating vertex. Every vertex $i$ computes $ID_{near}(i)$ and $D_{near}(i)$, based on the values received from its child nodes.

## 6.5. The Methodology of The Proposed Algorithm

Our algorithms are restricted to tree structures. Let $T$ be the root of a tree $T=(V,E)$. If there exists no such root then we can arbitrarily select any vertex as the root, without loss of generality for the execution of this algorithm.

The algorithm basically works in a bottom-up fashion starting at the leaves and moving to the root where the root decides the termination of the algorithm. There are three types of nodes/vertices namely, leaf, internal and the root node.

- A node with only one neighbor is a leaf node.

- There is a designated root node and all other nodes are internal nodes.

All the nodes in the system are associated with a state. A node may be *IDLE*, *ACTIVE*, *INACTIVE*, or *TERMINAL*. A node has certain transient states like being *saturated* after receiving messages from all its children. At the start of the algorithm all nodes are in the IDLE state. The leaf nodes enter the ACTIVE state. The parent of a node $i$ is denoted by $P(i)$. A leaf node $i$, sends a message to its parent $P(i)$, by executing the

command SEND ($i$, 1, $P(i)$, 0), where $i$ is the *id* of the leaf node, 1 is the value of the message or the message number, and $P(i)$ is the parent of the node $i$, 0 indicating that no node dominates it. The leaf nodes set their $ID_{near}$ to 0. The leaf node then enters the state INACTIVE. Every internal node becomes active when it receives a message from any of its children.

An active internal node upon receiving messages from all its children enters a SATURATED state. After sending the appropriate message to its parent an internal node enters the state INACTIVE. Any node in the SATURATED state executes the following:

- Compute the values of $G_p$, $I_p$, $G_n$, $I_n$, $ID_{near}$, and $D_{near}$. $G_p$ is the value of the largest message number received from any child $i$ and $I_p$ is the *ID* of that vertex (child). $G_n$ is the value of the *largest negative* message number received from any child $i$ and $I_n$ is the *ID* of that vertex (child). $ID_{near}$ and $D_{near}$ refer to the *id* and distance of the nearest dominating vertex respectively. The variables $ID_{near}$ gets a value only when a negative message is propagated, as indicated in the algorithm described in the next section.

The local variables of any internal node are initialized to the following values.

$G_p$ = -MAXINT (the largest negative number)
$I_p = 0$;
$G_n$ = -MAXINT;
$I_n = 0$;
$ID_{near} = 0$;
$D_{near} = 0$;

An interval node $i$ upon receiving a message from one of its children executes

the following procedure *Compute_Local* ($i$) and updates its local variables.

---

```
Procedure Compute_Local (i);
/*    Φ(i) refers to the message received from vertex i.        */
/*    This procedure is executed by an internal node when it   */
/*    when it receives a message from its neighbor (a child)   */
begin
      if 0 < G_p < Φ(i).mesg_num then
      begin
            G_p = Φ(i).mesg_num;
            I_p = Φ(i).ID;
      end;
      if 0 > Φ(i).mesg_num > G_n then
      begin
            G_n = Φ(i).mesg_num;
            I_n = Φ(i).ID;
            ID_near = Φ(i).ID_near;
            D_near = (-1 * G_n);
      end;
end;
```

---

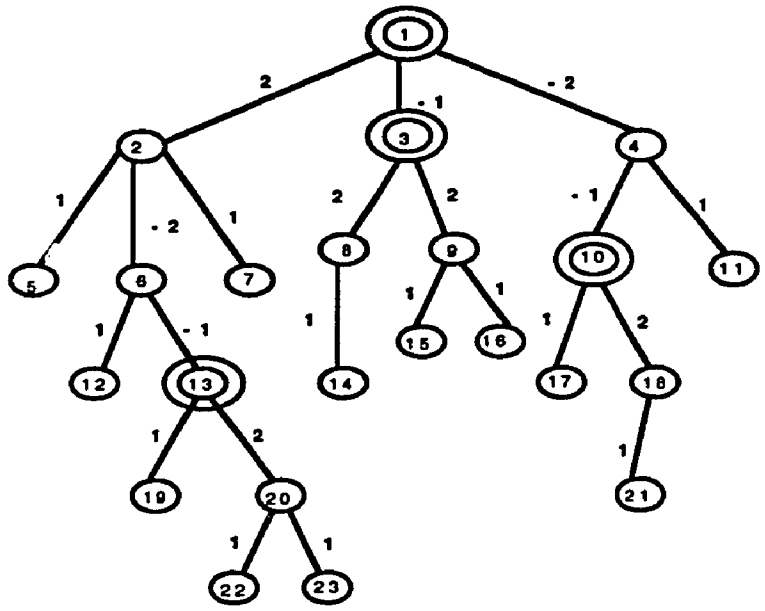
## 6.6. Algorithm for $r$-Dominating Set - Algorithm RDS

In this section we present procedures that are executed by the leaf and internal nodes and a procedure the root node executes after entering the SATURATED state. The internal node after receiving a message from one of its children executes the procedure Compute_Local. After it receives messages from all its children it executes the procedure Intern. Each leaf node sends a message to its parent and enters INACTIVE state and this is given in procedure Leaf. When the root enters saturated state and executes procedure Root, the nodes which has to be in the minimum $r$-dominating set would be selected. We would like to note that the set $D_r$ is never sent to the parent and that nodes which are in the dominating set are informed.

```
Procedure Leaf (i);
/*    All leaf nodes i executes the following code        */
begin
        Send (i, 1, P(i), 0);
        D_near = 0;
        ID_near = 0;
end;

Procedure  Intern (i);
/*    This procedure is executed by an internal node in SATURATED state.  */
/* The vertex i calculates G_p and G_n as      described previously  */
/*    D_r -- The r-dominating set being computed.        */
/*    P (i) -- Parent of the vertex i in the tree.       */
begin
    if (G_p = r) then
    begin
        D_r = {D_r ∪ i}; /* Just indicate to node i that it is in the dominating set */
        ID_near = i;
        D_near = 0;
        Send (i, -1, P(i), ID_near);
    end
    else if  (G_n = 0) then
        Send (i, G_p + 1, P(i), 0)
    else if (G_p = 0) And (G_n ≠ -r) then
        Send ( i, G_n - 1, P(i), ID_near)
    else if (G_p = 0) And (G_n = -r) then
        Send ( i, 0, P(i), 0 )
    else if (G_p - G_n) ≤ r then
        Send (i, G_n - 1, P(i), ID_near)
    else Send (i, G_p + 1, P(i), ID_near);
end;

Procedure  Root ();
begin
    if (G_n = 0) Or ((G_p - G_n) > r ) then
    begin
        Indicate to the root that it is in the dominating set;
        ID_near = root of the tree;
        D_near = 0;
    end
    Terminate; /* Execute the terminate procedure */
end;
```




The vertices in the 2-dominating set marked by double circles are { 1, 3, 10, 13 }

Figure 6.1 - *2-Domination*

### 6.6.1 Illustrative Example

We shall discuss the various steps in the algorithm with an illustrative example as shown in Figure 6.1. The problem we are trying to solve is a 2-domination. The algorithm is initiated at the leaf nodes namely {5, 12, 19, 22, 23, 7, 14, 15, 16, 17, 21, 11}. The leaf nodes send a message to their respective parents {2, 6, 13, 20, 20, 2, 8, 9, 9, 10, 18, 4}, a message with message number = 1. The nodes {20, 8, 9, 18} become saturated and compute their $G_p$, $G_n$, etc as described in section 3 & 4. Since all messages so far have positive message numbers, $G_n = 0$, for all the saturated nodes, and $G_p = 1$.

Node 13 receives a message with message number = 2 (or $r$), and hence becomes a dominating node, because it is critical positive. A dominating node sends a message with message number = -1 to its parent. A similar situation is faced by node 10 and 3.

Node 4 receives a message from node 10 ( a dominating node ), with a message number = -1, and becomes saturated. At node 4, $G_p = 1$, $G_n = -1$, $ID_{near} = 10$. Since $G_p - G_n = 2$, node 4 knows that node 11 is also satisfied (dominated) by node 10. Therefore node 4 sends a message with message number = -2 to its parent. A similar situation is faced by node 6.

When node 2 becomes saturated, its $G_p = 1$, and $G_n = -2$, and $ID_{near} = 13$. Since $G_p - G_n = 3$ (> 2), node 2 knows that there exists nodes in some of its subtrees, which are still not dominated by any node. Hence it sends a message with message number $= G_p + 1$, to its parent in the tree.

Then, Node 1 (root) becomes saturated, with $G_p = 2$, and $G_n = -2$. Since a $G_p$ value of 2 (or $r$) makes it critical, the root (node 1) becomes a dominating node. At this stage the root node (1) starts the termination phase of the algorithm. In this phase, the messages are sent from the parent to the children. Node 1 sends a message to nodes { 2, 3, 4}, a message with $ID_{near} = 1$, and $D_{near} = 1$. Node 2 had its $ID_{near} = 13$, and $D_{near} = 2$, during the first phase (bottom-up) of the algorithm. When it receives a terminal message from its parent with $D_{near} = 1$, it updates its $ID_{near}$ to 1, and $D_{near} = 1$, since distance from node 1 is less than distance from node 13. Node 2 propagates the terminal message down the tree to its children { 5, 6, 7}. Therefore, $ID_{near}(5) = 1$, $ID_{near}(7) = 1$, $D_{near}(5) = 2$, $D_{near}(7) = 2$. Node 6 does not make any change. and propagates the terminal message with $ID_{near} = 13$, and $D_{near} = 2$.

When a leaf node receives a terminal message and sets its variables' values, it sends an acknowledge message to its parent in the tree. We shall stop the discussion here, since the rest of the algorithm is very simple.

### 6.6.2 Termination of the Algorithm RDS

The root initiates the terminate phase by invoking the terminate procedure in the procedure Root During the execution of the terminate procedure first the root sends a special terminate message to all its children. When an internal node receives the terminate message it goes into the terminal state and does the following. The format of a terminate message is as follows:

("terminate", $ID_{near}$, $D_{near}$)

- If it is not a dominating node but has its $ID_{near}$ already set (not zero) then it compares its $D_{near}$ with the $D_{near}$ just received and sets its $D_{near}$ and $ID_{near}$ to that vertex which is nearer (Note: the $D_{near}$ is one greater than that received). It then sends its $D_{near}$ and $D_{near}$ along with the terminate message.

- If it is not a dominating node and does not have its $ID_{near}$ set then it sets its $ID_{near}$ to that ID that was received along with the terminate message. It updates its $D_{near}$ and sends it along with the propagated terminate message.

- If the vertex receiving the terminate message is a leaf then it sets its $ID_{near}$ and $D_{near}$ accordingly and goes into the terminal state and sends an ACKNOWLEDGE message up the tree.

When an internal node receives an acknowledge from all its children, it propagates the acknowledge to its parent. When the root receives an acknowledge message from all its children it confirms the completion of the algorithm.

## 6.7 Message and Time Complexity Analysis

The algorithm runs in two phases, namely the bottom-up phase and the terminal phase. During the bottom-up phase, each vertex sends exactly one message to its unique parent in the tree. This takes $O(h)$ time units, where the height of the tree is $h$. Since there are $|V|$ nodes in the tree, the algorithm uses $|V|-1$ messages. The message complexity is $O(V)$. During the terminate phase of the algorithm, along each edge of the tree exactly one message is sent. The message complexity is $O(n)$, where $n = |V|$. Clearly, in $O(n)$ time the algorithm terminates.

## 6.8 Proof of Correctness of the Algorithm RDS

The correctness of our algorithm can be shown using techniques presented in [Cock75, Slat76]. We present an alternate and simpler proof to show that our algorithm indeed determines a minimum $r$-dominating set. First we prove the following.

**Lemma 6.1**: Every vertex included in the set $D_r$ by the algorithm *uniquely* dominates at least one vertex in the tree and hence, the set $D_r$ is a minimal $r$-dominating set for the given tree.

*Proof*: A vertex is included in the dominating set if and only if it receives a message

with message number equal to $r$. A vertex can receive messages only from the vertices in its subtree. If a message with a message number equal to $r$ is received by a vertex $a$, then it indicates that there exists at least one vertex in the subtree (of which $a$ is the root) which is at a distance of $r$ and not dominated by any other vertex.

The root is included in the dominating set if $G_p - G_n > r$ (The meaning of $G_p$ and $G_n$ are described in section 3) in which case there exists at least one vertex in the tree that is uniquely dominated by the root.

If the $G_p = 0$ and $G_n = 0$ at the root then the root is included in the dominating set because no vertex in $\{D_r$ - root$\}$ dominates the root. Therefore a case may arise where the root of the tree is included in the dominating set but does not dominate any other vertex except itself. (This is a special case which does not alter the proofs presented in the subsequent sections).■

The following lemma is used in proving that the minimal $r$-dominating set obtained by our algorithm is a minimum $r$-dominating set.

Lemma 6.2: Let $l$ and $m$ be any two vertices of a dominating set $D_r$, and let $A$ and $B$ be the set of vertices *uniquely* dominated by $l$ and $m$ respectively. There exists at least two vertices $a \in \{A \cup l\}$ and $b \in \{B \cup m\}$ such that $D(a, b) > 2r$.

*Proof*: For every vertex $u \in D_r$, there exists at least one vertex $v$ in its subtree which is at a distance $r$ from it (Lemma 6.1). The only exception to the above claim is the case where the dominating vertex, in question, is the root. This situation is discussed in Case 3. below.

Case 1: Let us consider the case where the root is not involved. Let $l$ and $m$ be in different subtrees, i.e $l$ is not contained in a subtree of $m$ and vice versa. Then it is trivially true from the algorithm that there exists a vertex $a \in A$ which is uniquely dominated by $l$ and which is at a distance of $r$ from it. Similarly for $m$ there exists a vertex $b \in B$, which is uniquely dominated by $m$. It is true that the distance $D(a, b) > 2r$, since $a$ and $b$ are not directly connected except through $l$ and $m$.

Case 2: Let $m$ be contained in some subtree of $l$. There exists a vertex $a \in A$ which is at a distance of $r$ from $l$ in some subtree of $l$. There exists a vertex $b \in B$ which is at a distance of $r$ from $m$ in some subtree of $m$. Let $m$ be contained in the subtree rooted at $a$. It is obvious that $a$ is at a distance greater than $r$, from $m$, since it is our assumption that $a$ is uniquely dominated by $l$. Furthermore, $b$ is at a distance of $r$ from $m$. Therefore $a$ and $b$ are more than $2r$ levels apart in the tree or $a$ and $b$ are at a distance of least $2r$ apart.

If $m$ is not in the same subtree as $a$ then we use the same argument as Case 1. to prove that $D(a, b) > 2r$.

Case 3: Let $l$ be the root of the tree and let $A$ be empty. If $l$ does not dominate any other vertex, then it is trivially true from the nature of the algorithm that $l$ by itself is at a distance greater than $r$, from any dominating vertex $m$. Therefore there exists a $b \in B$ such that $D(l,b) > 2r$.

- If $A$ is not empty, there exists a vertex $a \in A$ which is dominated uniquely by $l$. Let $m$ be a vertex in the subtree at $a$. This is the same as Case 2. above.

- If $m$ is not contained in a subtree rooted at $a$, then, the algorithm ensures that $a$ is at a distance greater than $r$, from any dominating vertex. Since $b$ is a vertex in the subtree rooted at $m$, and there is no direct path from $a$ to $b$, it is ensured that the distance between $a$ and $b$, $D(a,b) > 2r$. ■

**Theorem 6.3**: The set $D_r$ determined by algorithm RDS is a *minimum* $r$-dominating set.

*Proof*: Let $l$ and $m$ be two vertices in $D_r$ computed by the algorithm RDS. Let the set of uniquely dominated vertices of sets $l$ and $m$ be $A$ and $B$ respectively. We shall prove that there does not exist any other minimal $r$-dominating set whose cardinality is less than that of $D_r$, by showing that there does not exist a vertex $k$ which dominates the vertices in $\{A \cup l\}$ and $\{B \cup m\}$. This fact is trivially proven using Lemma 6.2 as follows. If $A$ and $B$ are unique vertex sets of vertices $l$ and $m$, then there exists a vertex $a \in \{A \cup l\}$ and a vertex $b \in \{B \cup m\}$ such that $D(a, b)$ is greater than $2r$. Therefore, there does not exist a vertex $k$ which can dominate the vertices of both the sets $\{A \cup l\}$ and $\{B \cup m\}$. Thus, we can conclude that there is no other minimal r-dominating set whose cardinality is less than the cardinality of $D_r$. Hence the theorem.■

## 6.9 Summary

We have presented efficient distributed algorithms for $r$-domination in tree networks with a message complexity of $O(n)$, where $n = |V|$. The time complexity of the algorithms is $O(h)$, where $h$ is the height of the tree in consideration. The value of $h$ is

obviously bounded by $n$, which is the number of vertices in the tree. It is interesting to note that both the time and message complexities are independent of the value of $r$. This fact is reflected from the nature of the algorithm, which is simple and elegant. This algorithm lends itself to the design of efficient parallel algorithms for r-domination set problems as shown in [Radh90]. Many variations of the $r$-domination problems have been studied earlier. Unified distributed algorithms for the total, reliable and independent dominating set problems can be found in [Subb90]. In the next chapter, we provide optimal distributed algorithms for the reliable and total $r$-dominating set problem in trees. In part II of this dissertation, we provide optimal parallel algorithms for the $r$-domination problem on trees using an EREW-PRAM model.

# Chapter Seven

## VARIATIONS OF $r$-DOMINATION IN TREES

### 7.1 Introduction

In the previous chapter, we considered the problem of $r$-domination in trees and provided optimal distributed algorithms for the same. In this chapter, we analyze three variations of the $r$-domination problem. We also introduce the problem of reliable $r$-domination. We present optimal unified distributed algorithms for the total, and reliable r-Domination on tree structures with a message and time complexity of $O(n)$, where $|V| = n$. A dominating set $D$ of a graph $G$ is *Independent* if the nodes in $D$ are pairwise non-adjacent, *connected* if the subgraph $G[D]$ induced by $D$ is connected, *total* if $G[D]$ has no isolated node. In this chapter, we introduce the problem of *reliable* $r$-domination. The reliable $r$_Dominating set referred to as $RD_r$, is a subset of the set $V$, such that for every vertex $a \in V - RD_r$, there exists a vertex $b \in RD_r$ and the distance between $a$ and $b$ in $G$, denoted by $D(a,b)$, is less than or equal to $r$, and no vertex in $RD_r$ is isolated. A $r$-reliable dominating set is important, since failure of any single facility at a dominating vertex/node still ensures that the distance of any node from any facility is increased by at most one. We use the acronyms RDS, RRDS, and TRDS representing the algorithms for $r$-domination, reliable $r$-domination, and total $r$-domination respectively. The model of computation used is the same as that provided in chapter 6.

## 7.2 Reliable $r$-domination

The algorithm for computing the reliable $r$-dominating set is similar to that provided for the $r$-dominating set in chapter 6. We refer to the reliable $r$-dominating set algorithm by the acronym RRDS.

### 7.2.1 Message format for Algorithm RRDS

Messages are of the form :

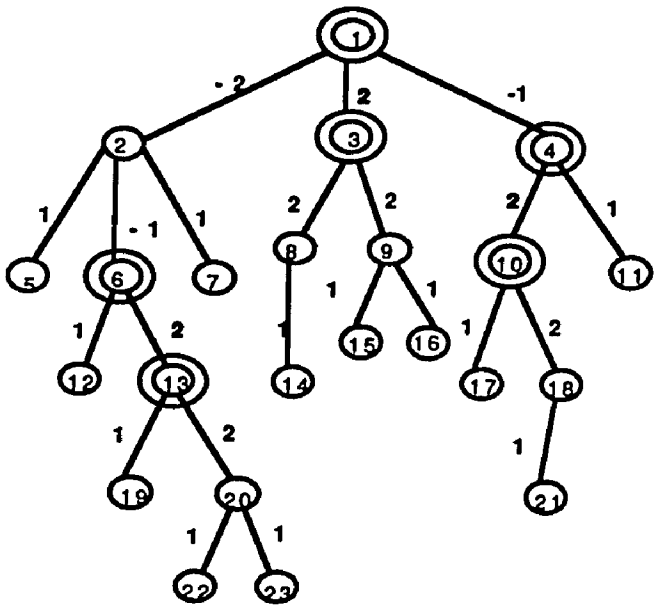
( $ID$, State_Flag, Message Number, Destination, $ID_{near}$ )

The fields of the message for this problem are identical to that of the $D_r$ problem, except for the field *state_flag*. The state_flag of a node can be one of the following types: { Looking, In, Seen, Saturated, Special }

- Every node has a state_flag of *Looking* until it receives a message upon which it can change its state_flag.
- A node changes the state_flag to *In* if it receives a message which is *critical Positive*.
- A node changes the state_flag to *Seen* if it receives a critical positive message from a child whose state_flag is *In*.
- A node is said to be *Saturated* if it has received messages from all its children.
- The root node sends a message with the state_flag Special to one of its children, to include it in the dominating set. This specific case is explained in

the following section.

In the description of the algorithm that follows, we do not indicate the sending of the *ID*, as it is implicitly understood.




The vertices in the reliable 2-dominating set marked by double circles are { 1, 3, 4, 6, 10, 13 }

## 7.2.2 Algorithm RRDS

Procedure *Leaf* (i);

/* Any leaf node $i$ executes the following code */
Send (i, "Looking", 1, $P(i)$, 0 );
$D_{near} = 0$;
$ID_{near} = 0$;

---

When an internal node receives messages from all its children, the following code is executed. The node first computes $G_p$, $I_p$, $G_n$, and $I_n$, $ID_{near}$ and $D_{near}$.

**Procedure** *Internal* (i);

```
    /*    Φ(i) refers to the message received from node i    */

begin
      if ( G_p = r ) And ( Φ(I_p).state_flag = "In" ) then
      begin
            RD_r = RD_r ∪ {i};
            Send (i, "Seen", -1, P(i), ID_near );
      end
      else if ( G_p = r ) And ( Φ(I_p).state_flag = "Looking" ) then
      begin
            RD_r = RD_r ∪ {i};
            Send (i, "In", r, P(i), ID_near );   { A message number of 'r' is sent to force }
      end                                        { the parent to be in RD_r }
      else if ( G_n = 0) then
            Send (i, "Looking", G_p + 1, P(i), 0 )
      else if ( G_p = 0 ) And ( G_n <> -r ) then
            Send (i, "Seen", G_n - 1, P(i), ID_near )
      else if ( G_p = 0 ) And ( G_n = -r ) then
            Send (i, "Seen", 0, P(i), 0 )
      else if ( G_p - G_n ) ≤ r then
            Send (i, "Seen", G_n - 1, P(i), ID_near )
      else
            Send (i, "Looking", G_p + 1, P(i), 0 );
end;
```

---

The root node upon entering the SATURATED state executes the following code.

**Procedure** *Root*

```
      begin
       if (( G_n = 0) And ( G_p ≤ r) And ( Φ(I_p).state_flag = "In" )) then
            RD_r = { RD_r ∪ i };
       else if (( G_n = 0) And ( G_p ≤ r) And ( Φ(I_p).state_flag ≠ "In" )) then
       begin
            Send (i, "Special", r, I_p , i );
            RD_r = { RD_r ∪ i };
       end
```

```
        else if ((( G_p - G_n ) > r) And ( Φ(I_p).state_flag ≠ "In" ) )then
        begin
              Send (i, "Special", r, I_p , i );
              RD_r = { RD_r ∪ i };
        end
        Terminate;
end;
```

The reliable $r$-dominating Algorithm differs from the Algorithm RDS described in the previous chapter, in the following aspects.

1. Since every node in the set $RD_r$ needs a neighbor, a node which receives a message number = r, sends a message with message number = r, and state_flag = "In", to its parent, if no child of the node in consideration belongs to $RD_r$.

2. The root node also has to decide whether it is to be included in the $RD_r$, depending on the message numbers received from its children, but it also has to send a special message to one of its children to force it to be in the dominating set, if no child of the root is already in $RD_r$, since it also needs a neighbor in $RD_r$.

### 7.2.3 Termination of Algorithm RRDS

The terminate phase of this algorithm is exactly the same as that of the $r$-dominating set problem.

### 7.2.4 Message Complexity Analysis of Algorithm RRDS

The overall message complexity of the algorithm RRDS is $O(n)$, where $n=|V|$.

The argument for this complexity is the same as that provided for the r_dominating set algorithm.

## 7.3 Total $r$-Dominating Set

The total $r$_Dominating set referred to as $TD_r$, is a subset of the set $V$, such that for every vertex $a \in V$ (including the vertices in the dominating set itself), there exists a vertex $b \in TD_r$, where $b <> a$, and the distance between $a$ and $b$ in $G$, denoted by $D(a,b)$, is less than or equal to $r$. Therefore the cardinality of the reliable and total dominating sets is at least 2.

The algorithm $TD_r$ is different from the reliable dominating set algorithm, in that a dominating node itself is also to be dominated by some other node in the tree. Therefore, the calculation of $G_p$ and $G_n$ is modified.

The format of the message remains the same as in $RD_r$, but, one more type is included in the state_flag, namely DOM, indicating that the message number was initiated by a dominating node. When a node receives a message number equal to $r$, it is immediately included in the dominating set. This node then sends a message number 1, with type "DOM" to its parent, since this node is also looking for some other node to dominate it. Once a "DOM" message has been received by a node, it is subsequently propagated up the tree.

When an internal node receives messages from all its children, it proceeds to compute the values of the local variables. The initialization of the variables is the same as in the previous algorithms, except that $G_n$ = MAXINT. The algorithm to compute the value of the local variables is different from that used in the previous problems.

---

**Procedure** *Total_Compute_Local* **(i);**

```
/*   Φ(i) refers to the message received from vertex i.        */
/*   G_p , I_p , ID_near , D_near                              */
/*   Modified computation of G_n and I_n                       */
if Φ(j).mesg_num > G_p then
 begin
   G_p = Φ(j).mesg_num;
   I_p = Φ(j).ID;
 end;
 if (Φ(j).state_flag = "DOM") and (Φ(j).mesg_num < G_n) then
 begin
   G_n = Φ(j).mesg_num;
   I_n = Φ(j).ID;
   ID_near = Φ(j).ID_near;
   D_near = G_n;
 end
```

---

After an internal node enters the SATURATED state, it changes

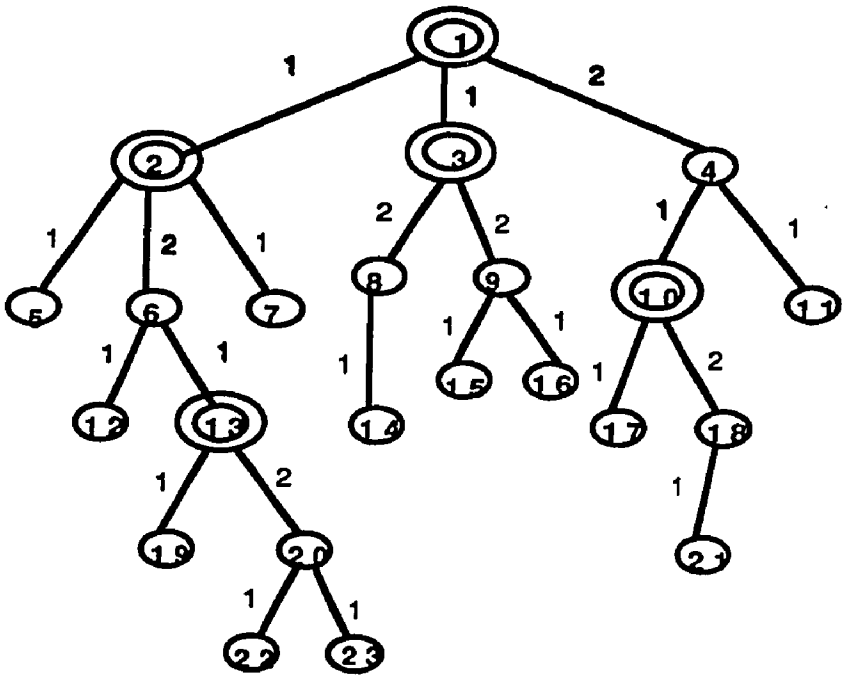
$$G_n = -1 * G_n;$$

* Let a message with a message type "DOM" and message number $p$ be received by an internal node. For the purpose of computation of $G_n$ the negated value $-p$ is used, since this indicates the distance from the nearest dominating node in the subtree containing the node from which this message was received. If a positive message number is received, but the state is not "DOM" then $G_n$ is made 0.

* When the root node receives no message with type "DOM", it is indicative of the fact that no node in the entire tree is a dominating node, thus forcing the root to be in included in the dominating set. In such a case, the root sends a special

message down the tree along any one of its subtrees, to force that child to be in included in the dominating set.

It is interesting to note that no message with a negative message number is ever propagated by any node during the execution of this algorithm.



The vertices in the total 2-dominating set marked by double circles are { 1, 2, 3, 10, 13 }

### 7.3.1 Algorithm TRDS

An internal node in the SATURATED state, after receiving messages from all its children, executes the following code:

**Procedure** *Intern* **(i)**

**begin**
**if** ( $G_p$ = r ) **then**

```
    begin
        TD_r = { TD_r ∪ i };
        Send (i, "DOM", 1, P(i), i );
    end
    else if (G_n <> 0) and ( G_p - G_n ) ≤ r then
        Send (i, "DOM", G_n * -1, P(i) )
    else
        Send (i, Φ(I_p).state_flag, G_p + 1, P(i) );
end;
```

The root node after receiving messages from all its children executes the following code:

**Procedure** *Root*

```
    begin
     if ( G_n = 0) And ( G_p ≤ r ) then
     begin
        TD_r = { TD_r ∪ i };
        send (i, "special", I_p, i);
     end
     else if ( G_p - G_n ) > r then
        TD_r = { TD_r ∪ i };
     Terminate;      /* same as r-dominating set Algorithm RDS */
end;
```

---

### 7.3.2 Termination of Algorithm TRDS

The terminate phase of this algorithm is exactly the same as that of the $r$-dominating set problem.

### 7.3.3 Message Complexity Analysis of Algorithm TRDS

The overall message complexity of the algorithm $TD_r$ is $O(n)$, where $n=|V|$. The

argument for this complexity is the same as that provided for the algorithm $D_r$.

## 7.4 Optimality Criterion

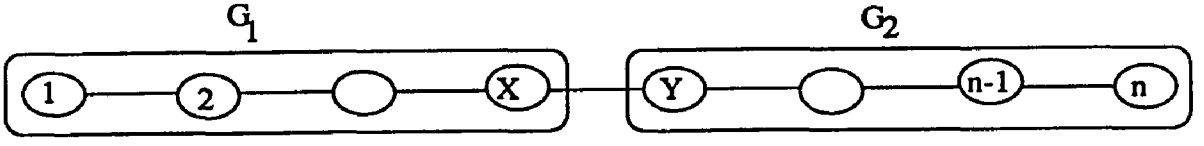
**Theorem 7.1.**

An algorithm that solves the minimum r-domination problem on every tree network uses at least $e$ messages where $e = |E|$

*Proof:*

Suppose, to the contrary, that an algorithm solves the minimum $r$-domination problem using fewer than $e$ messages, then it follows that for every terminating execution of the algorithm, there is a edge $(x, y)$ on which no messages are transmitted.

Now consider a graph $G = (V, E)$, where $G$ is a line graph, as shown in the following figure.




A Graph 'G'

Fig 7.3. Line Graph

Since, there is no global knowledge available to any node, a node cannot distinguish whether a neighboring node is a leaf or an internal node. Therefore, in the specific case in question, node $x$ has no knowledge whatsoever about the nodes that are connected to node $y$ and vice versa. Now the graph $G$ in figure 7.3. is broken into two subgraphs, $G1$ and $G2$, where $G1$ consists of all nodes

from node 1 up to and including $x$, and $G2$ contains all nodes from $y$ to node $n$. Since the nodes in $G1$ have no knowledge of the existence of $G2$ ($G$ is a tree), no dominating node in $G1$ dominates all the nodes in $G2$, and vice versa. Therefore, there should exist at least one dominating node in each subgraph $G1$ and $G2$. This is a contradictory, to the case where $r >= |V|/2$, wherein it suffices to have only one node that dominates all the nodes in the graph. Since $e = |E| = |V| - 1 = n - 1$, so we can say that any algorithm that solves the minimum $r$-domination problem, uses at least $\Omega(n)$ messages. ■

## 7.5 Summary

The algorithms discussed in chapter 6 and 7 are optimal in the number of messages used during their execution. The time complexity of the algorithms is $O(h)$, where $h$ is the height of the tree in consideration. The value of $h$ is obviously bounded by $n$, which is the number of vertices in the tree. It is interesting to note that both the time and message complexities are independent of the value of $r$. This fact is reflected from the nature of the algorithms, which are simple and elegant. Many variations of the $r$-domination problems have been studied earlier. We introduce the problem of reliable $r$-domination in this dissertation. A reliable $r$-dominating set ensures that the failure of any single facility at a dominating vertex/node still ensures that the distance of any node from any facility is increased by at most one.

In this chapter, we considered three variations of the $r$-domination problem. We introduce the problem of reliable $r$-domination. We have presented an optimal unified distributed algorithms for the total, and reliable r-Domination on tree structures with a message and time complexity of $O(n)$, where $|V| = n$. A dominating set $D$ of a graph $G$ is *Independent* if the nodes in $D$ are pairwise non-adjacent, *connected* if the subgraph $G[D]$ induced by $D$ is connected, *total* if $G[D]$ has no isolated node. In this chapter, we introduce the problem of *reliable* $r$-domination. The reliable $r$_Dominating set referred to as $RD_r$, is a subset of the set $V$, such that for every vertex $a \in V - RD_r$, there exists a vertex $b \in RD_r$ and the distance between $a$ and $b$ in $G$, denoted by $D(a,b)$, is less than or equal to $r$, and no vertex in $RD_r$ is isolated. A $r$-reliable dominating set is important, since failure of any single facility at a dominating vertex/node still ensures that the distance of any node from any facility is increased by at most one. We have also developed optimum distributed algorithms for $r$-domination, reliable $r$-domination, and total $r$-domination.

# Chapter Eight

## PARALLEL ALGORITHMS FOR $r$-DOMINATION IN TREES

### 8.1 Overview of Parallel Computation

The subject of parallel computing has moved from the exotic to mainstream computer science within a decade. Many types of parallel computers are in operation, ranging from 2 to 65,536 processors. There are major differences between the existing parallel machines, even as far as the naive user is concerned. There is no one "generic" model of computation that can applied to all parallel computers. The design of parallel algorithms and their analysis is much more difficult than the corresponding steps for sequential algorithms. In this chapter we shall look briefly into the PRAM (Parallel Random Access Machines) model of computation.

The main measures of complexity for sequential algorithms are running time and space utilization. These measures are important in parallel algorithms as well, but we must also worry about about other resources, namely, processor utilization. A comprehensive view of parallel machine models is beyond the scope of this dissertation. Hence we shall limit ourselves to the discussion of only PRAM models. The PRAM model is analogous the sequential RAM model. It is also a version of the shared memory model described in [Manb89]. A shared memory computer system consists of several processors and a shared memory. The computation is assumed to consist of steps. In each step, each processor performs an operation on the data it possesses, reads from the shared memory, or writes into the shared memory. In practice, each processor may also have its own local memory. The shared memory models

differ in the way they handle memory conflicts. The EREW (Exclusive Read Exclusive Write) model does not allow more than one processor to access the same memory location at the same time. The CREW (Concurrent Read Exclusive Write) model allows several processors too read from the same memory location at the same time, but only one processor can write. The CRCW (Concurrent Read Concurrent Write) model poses no restrictions on memory accesses.

## 8.2 Introduction

A set of nodes $D$ is a $r$-dominating set for a graph $G = (V, E)$ if every node in $V - D$ is at distance $r$ or less from a node in $D$. The domination problem is to determine the cardinality of a minimum dominating set for $G$. The domination problem is used in facility location problems, where each node represents a customer or a potential site for a facility. A feasible solution corresponds to a set of facilities located at $D \subseteq V$ such that every customer is at a distance at most $r$ to some facility. An optimal solution is a minimum cardinality set of facilities with this property.

The domination problem is NP-complete for undirected path graphs [Lask84], split graphs [Lask83], bipartite graphs [Lask83] and 2-CUBS [Corn86]. Polynomial time sequential algorithms for the domination problem are available for trees [Lask84], series-parallel graphs [Lask84], permutation graphs [Corn86], interval graphs [Keil86] and strongly chordal graphs [Chan89].

Recently, He and Yesha [He90] proposed an efficient parallel algorithm for solving the $r$-domination problem on trees. The parallel algorithm of He and Yesha for the $r$-domination problem runs in $O(lognloglogn)$ time with $n$ processors on an CREW-

PRAM. The algorithm in [He90] cannot be improved by the use of optimal tree contraction algorithm [Gazi87] since, the operation "SHUNK" performed at every phase of the tree contraction requires a time and processor complexity which is not a constant. In this paper we present an optimal parallel algorithm for the $r$-domination problem on trees which runs in $O(logn)$ time and uses $O(n/logn)$ processors on an EREW-PRAM. Our algorithms use the optimal parallel tree contraction algorithm proposed by Gazit et. al [Gazi87].

## 8.3. Sequential Algorithm for $r$-Dominating Set

In this section we present our sequential algorithm for solving the $r$-dominating set problem and is similar in spirit to the algorithm of He and Yesha [He90]. Our "labeling technique" is simple and yields an efficient parallel algorithm.

The algorithm performs a "bottom up" search on the tree $T$, starting from the leaves and moving towards the root $t$ of $T$. During the "bottom up" search for the $r$-dominating set, information is sent from the child nodes to the parent, the parent then determines whether it has to be in the dominating set $D$ and sends information to its parent. The information that is sent by a node $v$ to its parent ($p(v)$) is a'flag' indicating whether $v$ is *looking for* or *covered by* a member in the dominating set and a distance $k$. A negative or a positive $k$ would indicate that $v$ is *covered by* or *looking for*, respectively.

If a node $v$ sends a distance $k$, then it means that there is no node $w$ for $v$ which is in the dominating set such that the distance between $v$ and $w$ ($d(v,w)$) is less than or equal to $k$ (i.e. no node covering $v$ and node $v$ is *looking for* a member in the dominat-

ing set). If a node $v$ sends a distance $-k$ which is negative, then it means that there is a node $w$ in the dominating set for $v$ which is at distance $k$ (node $v$ is *covered by* $w$). Let us say that a node $v$ **receives** distances $k(c_1), ..., k(c_p)$ from its children $c_1, ..., c_p$. The node $v$ is in the dominating set if any of the following conditions are satisfied.

(1) node $v$ is not the root and receives a number $k(c_i) = r$. This means that the child node $c_i$ of $v$ has to be covered by node $v$.

(2) node $v$ is the root and no child $c_i$ of $v$ *covers* all the other children of $v$. A child $c_i$ covers all the other children if $k(c_j) - k(c_i) \leq r$ for all children $c_j$ of $v$. For example let $r = 2$, $k(c_i) = -1$ (which means that $c_i$ is in the dominating set; see (1) below) and $k(c_j) = 1$ ($c_j$ is looking for a member in the dominating set), then since distance between $c_j$ and $c_i$ is 2, node $c_j$ is covered by $c_i$.

The node $v$ which is not the root sends the following information to its parent.

(1) If node $v$ is in the dominating set, then it sends a -1 to its parent.

(2) If all $k(c_i)$'s are positive, then $v$ sends *maximum*$(k(c_1), ..., k(c_p)) + 1$ to its parent, where $c_i$ is a child of $v$ and $k(c_i)$ is a number sent by $c_i$ to $v$.

(3) If all the children $c_j$'s of $v$ are covered by some child $c_i$ of $v$, then $v$ sends the maximum of the negative values among $k(c_i)$'s plus a minus one to its parent, assuming some $k(c_i)$ is positive. Also, if the $k(c_i)$'s are negative then $v$ sends the maximum of the negative values among $k(c_i)$'s plus a minus one to its parent. If the maximum of all negative values is equal to $-r$, then $v$ sends a 0 to its parent.

(4) If not all the children $c_j$'s of $v$ are covered by some child of $v$, then $v$ sends the maximum positive value among $k(c_i)$'s plus one to its parent.

Now, we present the procedure Rake_dom which when called, repeatedly removes leaves of the tree until the root is reached and during this bottom up removal the dominating set is determined. In the procedure Rake_dom, each node $v$ is associated with an array $d(v)$ whose size is equal to the number of children of $v$. $d(v).l$ corresponds to the array element $l$, that is the location for storing the information sent by the $l$th child (from left) of $v$. Also, $\#v$ denotes that node $v$ is the $\#v$th child of the parent of $v$, $p(v)$.

**Procedure** Rake_dom ($v$);

**Begin**

(* Initially for each leaf node $v$, $|d(v)| = 1$ and $d(v.1) = 0$ *)

1. $L \leftarrow$ Maximum negative value among all $d(v)$'s, set it to $-(r+1)$ if none.

2. $M \leftarrow$ Maximum value in the array $d(v)$.

(* If $v$ is the root, it is in the dominating set if it has a child node which is not covered by the other child nodes. *)

3. **If** (($v$ is the root) **and** $((M - L) > r)$) **or** $(M = r)$ **Then**

   **Begin**

4. $D \leftarrow D \cup \{v\}$;

5. **If** ($v$ is not the root) **Then**

6. $d(p(v).\#v) \leftarrow -1$ (* $p(v)$ is covered by $v$ *)

   **End**

7. **Else If** ($v$ is not the root) **Then**

8. **If** $(M = -r)$ **Then**

9. $d(p(v).\#v) \leftarrow 0$ (* $p(v)$ is looking for a member in $D$ *)

10. **Else If** $(L \neq -(r+1))$ **Then**

11. **If** $(L = M)$ **or** $((M - L) \leq r)$ **Then**

12. $d(p(v).\#v) \leftarrow L - 1$

13. **Else** $d(p(v).\#v) \leftarrow M + 1$

14. **Else** $d(p(v).\#v) \leftarrow M + 1$

**End.**

The dominating set of a tree is obtained by repeatedly calling the Rake_dom procedure until the root is reached. The algorithm has a sequential time complexity of $O(n)$, where $n$ is the number of nodes of the tree. The correctness of the procedure Rake_dom can be proved formally using arguments similar to [He88, He90, Kari79, Slat76].

## 8.4 Parallel Algorithm for $r$-Dominating Set

The parallel algorithm for $r$-dominating set presented in this section uses the optimal tree contraction algorithm of Gazit, Miller, and Teng [Gazi87]. The parallel tree contraction algorithm consists of two operations called *rake* and *compress*. The *rake* operation removes all leaves from a tree $T$. Let a *chain* be a maximal sequence of nodes $v_1, ..., v_k$ in $T$, such that $v_{i+1}$ is the only child of $v_i$ for $1 \leq i < k$, and $v_k$ has exactly one child. The *compress* operation replaces each chain of length $k$ by one of $\lceil k/2 \rceil$. Let contract be the simultaneous application of rake and compress to the entire tree. Miller and Reif [Mill85] show that contraction operation need be executed only $O(\log n)$ times to reduce $T$ to its root. Gazit et. al [Gazi87] presented an optimal parallel algorithm for the tree contraction algorithm which runs on an EREW-PRAM

model. It was shown that even with the requirement that it takes $O(logk)$ time to rake $k$ leaves in the tree at any step, the contraction algorithm can be executed in $O(logn)$ time.

**Theorem 3.1** [Gazi87] Tree contraction can be performed deterministically in $O(logn)$ time using $O(n/logn)$ processors on an EREW-PRAM.■

We can use the above tree contraction algorithm to solve the $r$-domination problem in the following way. The RAKE operation of the algorithm Tree_contraction is replaced with the following step

**if** $v$ is a leaf, call Rake_dom ($v$) and remove it

The Rake_dom procedure on a node $v$ described in the previous section requires the calculation of minimum and maximum of the numbers supplied by its $k$ children. This can be trivially done in $O(logk)$ time using $O(k/logk)$ processors on an EREW-PRAM model.

We now have to present a method to perform the COMPRESS in parallel on a chain $C = v_1, ..., v_k$. During such a process it should determine the members in the chain to be placed in the minimum dominating set and make sure that $v_k$ sends the right value to its parent as required in the procedure Rake_dom. Let $C = \{v_1, ..., v_k\}$ be a chain during the execution of tree contraction algorithm. We will denote the position of a node $v_i$ in the chain $C$ to be its subscript $i$. During an early process let $T_i$ $(1 \le i \le k)$ be the tree attached at $v_i$ that has been deleted from $T$. Let $n_i$ be the label for node $v_i$ obtained after processing $T_i$. The label $n_i$ can be one of the following.

(i) If $v_i$ is in the dominating set then $n_i = r$ and $v_i$ is said to be *covered*.

(ii) If $T_i = v_i$, then $n_i = 0$.

(iii) Node $v_i$ is said to be *looking-for* if $n_i \geq 0$ and $n_i \neq r$.

(iv) Node $v_i$ is said to be *covered* if $n_i < 0$.

A node $v_j$ which is looking-for can be covered either by a node which is already covered, or by a node $v_i$ which should be placed in the dominating set to cover $v_j$. A node $v_i$ which is already covered can cover $v_j$ if either of the following two conditions is true.

$$ABS(i-j) + n_j + ABS(n_i) \leq r, \text{ or}$$

$$ABS(i-j) + n_j \leq r \text{ if } v_i \text{ is in the dominating set.}$$

Let $v_i$ be a node in the chain $C$ which does not satisfy the above two conditions. The node $v_i$ will be called as *critical* if satisfies the condition $U = i + r - n_i \leq k$ otherwise it will called as *non-critical.* That is a critical node is one which cannot be covered by a node which is already covered and requires a node $v$ in the chain $C$ to be placed in the dominating set to cover it. The node $v$ should be selected as close as possible towards the end of the chain since it would not only cover some nodes in the chain $C$, but also cover some of the nodes in the other chains of the parent of $v_k$. A non-critical node $n_j$ can be covered by nodes which are selected from $C$ to cover critical nodes if possible, otherwise the decision is left to the root of $v_k$ as to which node to select to cover $n_j$.

Based on the observations made above, we now present an algorithm to compress a chain $C = v_1, ..., v_k$ in $O(logk)$ time with $O(k/logk)$ processors using the EREW-PRAM model.

**Algorithm Compress;**

**Begin**

1. Identify nodes which can be covered by nodes which have already been covered i.e., by nodes $v_i$ with $n_i < 0$ or $n_i = r$.
2. Determine critical nodes and select a minimum number of nodes in $C$ to be placed in the dominating set $D$ to cover critical nodes.
3. Determine non-critical nodes which can now be covered based on new members in the dominating set.
4. Based on covered and uncovered nodes in $C$ determine the appropriate label that has to be sent to the parent of node $v_k$.

**End.**

We will now present methods to implement each of the above steps optimally in parallel. Many of the steps involve the use of the standard parallel techniques for which optimal parallel algorithms are known, like recursive doubling, duplication of a data item, maximum and minimum of numbers, integer sorting etc. A thorough description of some of the standard parallel techniques are presented in [Karp90].

**Implementation of Step 1.**

Mark the vertex $v_i$ *covered* which was previously uncovered if the interval $l_i = (i-(r-n_i),\ i+(r-n_i))$ contains the interval $l_j = (i-ABS(n_i),\ i+ABS(n_i))$ with $n_i < 0$ or $n_j = r$. Given a set of $k$-intervals, Attalah and Chen [Atta89] present an optimal parallel algorithm to identify intervals which contain other intervals which takes $O(logk)$ time and uses $O(k/logk)$ processors in the EREW-PRAM model. Therefore, Step (1) can be

implemented using the same time and processor bounds.

**Implementation of Step 2.**

Let $L$ be a set of nodes that are critical in the chain $C$. These can be identified optimally in parallel. In order to execute step (2) we transform the problem in step (2) to a problem which has a geometric flavor. We will formulate the *stabbing problem* as follows: for each $n_i$ of $v_i \in L$ draw a interval $l_i$ on the $X$-axis with dimensions $S_i = i - (r - n_i)$ and $E_i = i + (r - n_i)$, where $S_i$ and $E_i$ are the starting point and ending point of the interval $l_i$. Now, given a set of intervals $H$ ($|H| = k$) on the $X$-axis determine a *minimum* set of points $Y$ on the $X$-axis which satisfies the following two conditions:

(a) Every intervals in $H$ is intersected by some vertical intervals drawn through points in $Y$.

(b) The maximum point $y$ in $Y$ be as high as possible.

Clearly, a solution to the above stabbing problem is a solution to our problem. Before we proceed to discuss the optimal parallel algorithm we define some graph theoretic terminologies. A clique is a maximal set of intervals which share a common point. A minimum clique cover is a minimum set of cliques such that each interval is in at least one clique. A maximum independent set is a maximum set of intervals with the property that no two intervals in the set overlap. The stabbing problem can be solved by finding a minimum clique cover of the set of intervals in $H$ and taking the lowest end point from each clique. Also, the stabbing problem can be solved by determining the maximum independent set $I$ of a set of intervals in $H$ and choosing the end point of each interval in $I$. In [Moit88] an optimal parallel algorithm is presented to solve the

maximum independent set problem in $O(\log k)$ time with $O(k)$ processors using the CREW-PRAM model. We now present an optimal parallel algorithm which solves the maximum independent set problem given a set of intervals in $H$ using the EREW-PRAM model.

**Algorithm** MAX-IND-SET;

**Begin**

(i). Remove intervals from $H$ which contains another interval.

(ii). Let the remaining intervals (none of which contains another) be sorted along with their starting and end points together. Form groups of starting points which occur adjacent in the sorted list. Similarly from groups with ending points.

(iii). Let the sequence after step (ii). be $\{s_1, e_1, s_2, e_2, ..., s_k, e_k\}$ where $s_i$ ($e_i$) consists of starting (ending) points of intervals which occur together the sorted list determined in 2. Let the SUCCESSOR of an interval $i$, call it SUCC($i$) be the nearest interval to the right of $E_i$ (the end point of interval $i$). SUCC($i$) is the interval $j$ if $E_i$ is in $e_a$ and the first point in $s_b$ is $S_j$ with $a < b$. This is based on the crucial observation that since no is completely contained in some other interval, a interval that starts first ends first.

(iv). Let $S$ be the sequence of intervals traced by starting at interval 1 and following the SUCC($i$) repeatedly. The set $S$ is the maximum independent set of the intervals in $H$.

**End.**

The correctness proof of the above procedure is based on the proof of correctness of

the earlier algorithm using a CREW-PRAM model of [Bert87, Moit88]. We will now estimate the complexity of the algorithm MAX-IND-SET. Step (i). can be executed in $O(logk)$ time with $O(k/logk)$ processors in the EREW-PRAM using the algorithm in [Atta89]. Step (ii). can be done optimally in parallel in the EREW-PRAM by using an optimal sorting algorithm (see [Karp90]). The groups can be formed trivially in optimal parallel time. Step (iii). can be done optimally in parallel by first linking each $e$ group to its immediate $s$ group to its right and selecting the first point $S_j$ from the $s$ group and setting SUCC($i$) to be $j$ for each $E_i$ in the $e$ group. Step (iv). can be executed by using the optimal list ranking algorithm. We now state the following lemma.

**Lemma 3.2:** Given a set of $k$ intervals, the maximum independent set can be determined in $O(logk)$ time with $O(k/logk)$ processors in the EREW-PRAM model.■

The result in Lemma 3.2 improves the result in [Moit88] where the CREW-PRAM model is used to achieve the above time and processor bound. The solution to our stabbing problem is the end points of the intervals in the set $S$ obtained by the algorithm MAX-IND-SET.

**Implementation of Step 3.**

For each of the non-critical nodes draw a intervals as described earlier. Let $E_{last}$ be the end point of the last intervals which is in the set $S$ determined by the algorithm MAX-IND-SET. A non-critical node $v_i$ whose interval is $(S_i, E_i)$ can be covered by new members of the dominating set if $S_i \leq E_{last}$. Trivially, all such $v_i$'s that can be covered can be determined in optimally in parallel.

**Implementation of Step 4.**

The label that has to be sent to the parent of $v_k$ is determined as follows.

**If** there exist at least one node $v_i$ in $C$ which is not covered

**then** label sent to the parent of $v_k$ is $\underset{i}{Maximum}\,(k - i + n_i) + 1$

**else begin**

Let $v_x$ be the largest $x$ in the chain with $v_x$ in the dominating set.

Let $A = Minimum\ (\underset{i}{Minimum}\,(ABS(n_i) + k - i), (k - x))$.

**If** $A = r$

**then** send a 0 to the parent of $v_k$

**else** send $(A + 1)^* -1$ to the parent of $v_k$.

**end.**

The above steps can be executed optimally in parallel which involves finding maximum and minimum of certain numbers.

**Lemma 3.3:** Let $C = \{v_1, ..., v_k\}$ be the chain. The algorithm Compress correctly determines the members of the chain $C$ that has to be in the dominating set and computes the label that has to be sent to the parent of $v_k$ in $O(logk)$ time with $O(k/logk)$ processors using the EREW-PRAM model.■

**Theorem 3.4:** The $r$-dominating set problem on a tree with $n$ nodes can be deterministically solved in $O(logn)$ time with $O(n/logn)$ processors using the EREW-PRAM.■

## 8.5 Parallel Algorithm for the $p$-Center Problem

The $p$-center problem on trees asks for a set of nodes $X$ with $|X| = p$ belonging to the tree $T$ which minimizes the maximum distance from any node in $T$ to the nodes in

$X$. The node set $X$ constitutes the $p$-center and the maximum distance obtained is called the $p$-radius. For general graphs this problem is NP-complete [Kari79]. In [He90] a parallel algorithm which runs in $O(\log^2 n \log\log n)$ time with $n$ processors on the CREW-PRAM model is presented. We will present a faster algorithm which runs on a weaker model namely the EREW-PRAM model.

First we observe that the $p$-radius cannot be greater than $n$, where $n$ is the number of nodes in the tree $T$. Thus, the candidates for the $p$-radius are all $i$ with $1 \le i \le n$, call this set $R$. For a fixed tree $T$, the cardinality of the minimum $r$-dominating set is a nonincreasing function of $r$. So the $p$-radius $r_p$ of $T$ can be computed as follows. Select the median $r_1$ and compute minimum $r_1$-dominating set, call the resulting set $D_1$. If $|D_1| \le p$, then $r_p \le r_1$. Hence the members in $R$ greater than $r_1$ can be discarded. If $|D_1| > p$, then $r_p > r_1$ and the members in $R$ that are less than $r_1$ can be discarded. The algorithm continues in a binary fashion until only one member, which is $r_p$, in $R$ remains. The $p$-center is then the minimum $r_p$-dominating set. Using our parallel algorithm for computing the minimum $r$-dominating set the above $p$-center problem can be solved in $O(\log^2 n)$ time with $O(n)$ processors on the EREW-PRAM model.

**Theorem 4.1**: The $p$-center problem can be solved in $O(\log^2 n)$ time with $O(n)$ processors using the EREW-PRAM model.■

## 8.6 Summary

In this chapter we presented an optimal parallel algorithm for $r$-domination problem and an efficient parallel algorithm for the $p$-center problem on trees. An optimal

algorithm for computing the maximum independent set on intervals in the EREW-PRAM model was also presented and such a set is required by our algorithms on trees. We obtained the following results.

(1) The $r$-domination problem on trees can now be solved in $O(logn)$ time with $O(n/logn)$ processors using the EREW-PRAM model. Previous algorithm for the same problem runs using the CREW-PRAM model in $O(\log^2 n)$ time with $O(n)$ processors.

(2) The $p$-Center problem on trees can now be solved in $O(\log^2 n)$ time with $O(n)$ processors using the EREW-PRAM model. Previous algorithm for the same problem runs using the CREW-PRAM model in $O(\log^2 n \log logn)$ time with $O(n)$ processors.

# Chapter Nine

## PARALLEL ALGORITHMS FOR MULTI-DIMENSIONAL RANGE SEARCH

## 9.1 Introduction

Range Search has important applications in the areas of databases and computational geometry. The range search problem is to obtain a set of data points (tuples, records) satisfying a query which specifies a range of values on each dimension (attribute) of the data. We use the range tree proposed by Bentley as our data structure to be distributed. We show that $O(logn)$ search time can be effected for a range search on $n$ 3-dimensional points using $(2.\log^2 n - 14.logn + 8)$ processors and this is optimal for the range tree distribution. The results presented in this chapter appear in [Radh90].

## 9.2 Range Search

Let $S$ be a set of $n$ $d$-dimensional points in $R^d$. A range query $q$ is a $d$-range which is the cartesian product of $d$ intervals. The output of the query is all points in $S$ that lie within $q$. In the case of two dimensions the 2-range is a rectangle and for more than three dimensions the $d$-range is a hyperrectangle. Thus, the answer to the query $q$ is a set of all points in $S$ that are inside the rectangle or hyperrectangle as the case may be. Range search has several applications including databases and computational geometry [Prep85].

Bentley [Bent80a] gives a thorough overview of various multi-dimensional and range searching problems. Several data structures and algorithms for range searching

have been proposed and each has trade-offs between storage and time complexity. Bentley and Maurer [Bent80b] have shown the lower bound on the time complexity of range search on a set of $n$ $d$-dimensional data to be $(d.logn)$. With the overlapping-ranges data structure [Bent80b] the time bound of $O(d.logn)$ can be obtained at the expense of very high storage cost which is $O(n^d)$. Most practical algorithms use a storage cost of $O(n\log^{d-1}n)$ to obtain a time bound of $O(\log^{d-1}n)$ [Prep85]. Layered Range tree data structure [Prep85] a variant of range tree has the above storage and time complexity; a reduction of $O(logn)$ factor in storage and time complexity of the range tree. Chazelle [Chaz86] using the concept of filtering search reduced the storage cost to $O(n.\log^{d-1}n/loglogn)$ while retaining the time complexity.

More recently, Katz and Volper [Katz88] developed a parallel algorithm for retrieving the sum of values in a region on a two dimensional grid in $O(logn)$ time with $O(n^{1/3})$ processors. We will show that the range search on a two dimensional plane can be effected in $O(logn)$ time with $3/2.logn - 1$ processors. The retrieval of the sum of the values in a two dimensional region can also be done with the above time and processor bound.

One of the keys to efficient parallel searching is the distribution of the data points to be searched. A *distributed* data structure is typically a large data structure, such as a B-Tree, K-d tree, Range tree and others, that is logically a single entity but that has been distributed over several independent processor stores. This concept is not new and frequently arises in the area of distributed data bases. Ellis [Elli85] developed a distributed version of Extendible Hashing for database searching. Distributed data

structures of scientific calculation and processing of sets were introduced in [Scot87, Mu86] respectively. One of the fundamental advantages of the concept of distributed data structure is that processors are assigned to data statically and overheads due to dynamic allocation is avoided.

We will assume that the set $S$ of $n$ $d$-dimensional points is stored in a range tree (see Section 9.2). We will present a simple range-tree data distribution scheme and show that the search algorithm is optimal for this data structure (Section 9.3). Let DS($t$, 1) denote the best sequential time taken to search a data structure DS with 1 processor. A parallel algorithm with $p$ processors is optimal for a data structure DS if the search time is DS($t$, 1)/$p$. A non-trivial range search implementation technique on a Hypercube parallel architecture is presented in Section 9.4. Section 9.5. presents an argument for the reduction on the number of processors used for range searching. Summary is provided in Section 9.6.

## 9.3 The Range Tree data structure

We will first introduce the 2-dimensional range tree. The generalization to $d$ dimensions can be easily visualized. Let $S$ be the set of $n$ 2-dimensional points. First sort the $n$ points based on the value of the $x$-coordinate. Imagine each point $p$ as an interval $[x_i, x_i]$, where the first and second components are $B[p]$ (begin point) and $E[p]$ (end point). Now, the range-tree corresponding to the first dimension is a rooted binary-tree whose leaves contain the $n$ points sorted and placed from left to right as intervals. An interior node $v$ and its left ($v_1$) and right ($v_2$) children has an associated

interval with $B[v] = B[v_1]$ and $E[v] = E[v_2]$. Now the second dimensional coordinates i.e., the y-coordinates are stored in the tree as follows. For each interval $I = (B[v], E[v])$ belonging to the node $v$ in the tree, the y-coordinates of the points which project onto the interval $I$ are stored as a binary-tree and the node $v$ points to the root of the binary tree. Figures 9.2.a and 2.b show a set of points in the plane and its corresponding range tree, respectively.

| X | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... | 16 |
|---|---|---|---|---|---|---|---|---|---|---|
| Y | 9 | 13 | 12 | 17 | 14 | 6 | 10 | 16 | ... | 2 |

Figure 2.a - A set of points in the plane.

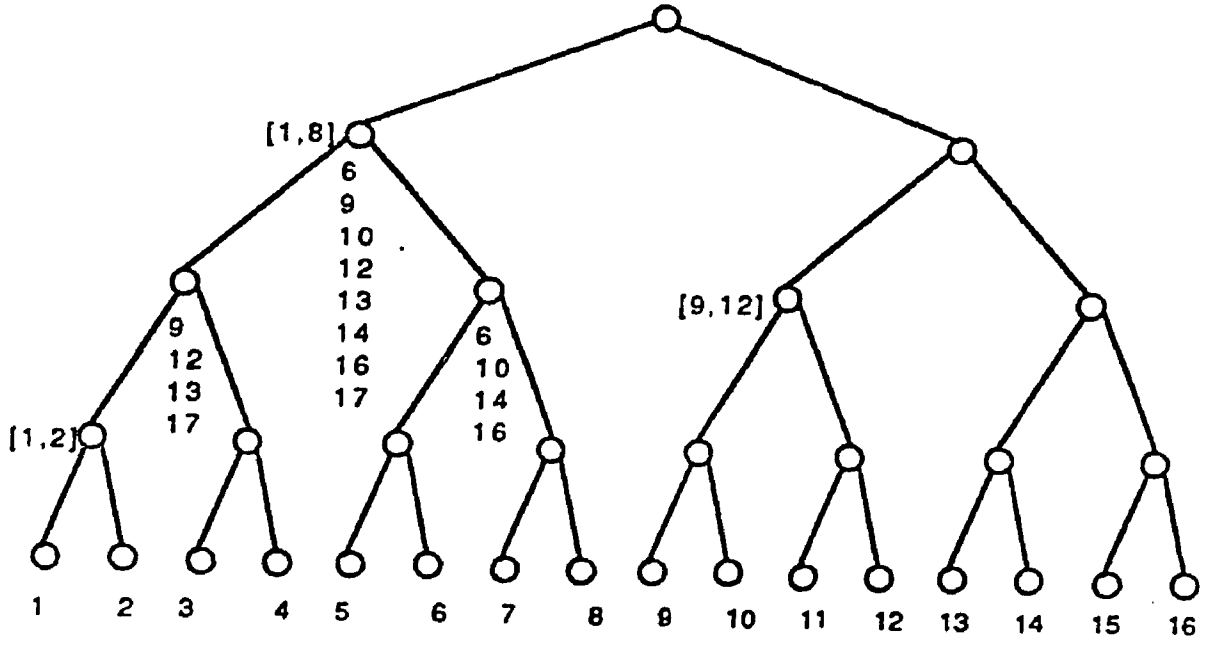



Figure 2.b - The range tree corresponding to points in Figure 2.a.

For the case where each point in the plane represents a value and the range query is to sum the values in a specified region we need to store the values $S_v$ at each node $v$ of the range tree as follows. Let $t_v$ be the binary tree corresponding to the y-coordinates at node $v$. Let $t_v^b$ and $t_v^e$ represent the left-most and the right-most leaves of the binary tree $t_v$. The value $S_v$ stored at node $v$ is the sum of the values of the points whose x-coordinates and y-coordinates lie in the interval $(B[v], E[v])$ and $(t_v^b, t_v^e)$, respectively.

We will now state some properties of the range tree from [Prep85].

**Proposition 9.1:** The number of nodes selected in the range tree during the range search on any dimension is at most $2.logn - 2$ and there are not more than two nodes selected from each level of the range tree.■

**Proposition 9.2:** Range searching of an $n$-point $d$-dimensional file can be effected by an algorithm in time $O((log\ n)^d)$ using the range-tree technique.■

## 9.4 Range tree distribution and parallel algorithm

The key to the success of any parallel algorithm for range searching is determined by the type of data distribution. With $O(n)$ processors effective searches can be made, but, having such large number of processors is highly impractical. In this section, with range tree as the data structure, we present a simple data distribution scheme with which $O(logn)$ search time using $(2\log^2 n - 14.logn + 8)$ processors is effected for the case of 3-dimensional data points. The technique we describe can

easily be extended to the case of $d$-dimensions. We will assume that the root and the leaves of the tree are at heights $h$ $(n = 2^h)$ and 1, respectively.

*Estimation of processor and time-complexity*

We now estimate the number of processors required to search in parallel for the case when $d = 2$ and $d = 3$.

In Proposition 9.1 we note that at most $2.logn - 2$ range tree nodes are selected for any range query on a single dimension. This tells us that with $2.logn - 2$ processors we can search the next dimension in parallel. Now, the time-complexity is given by the following simple equation:

$$Q(1, n) = O(logn)$$

$$Q(2, n) = Q(1, n) + O(logn) = O(logn)$$

Here $Q(1, n)$ is the time taken to search the range tree in dimension 1. Let us say that another $2.logn - 2$ processors are available at each of the selected nodes during the processing of the dimension $i$. The next dimension $i+1$ can also be processed in parallel. Generalizing this scheme to $d$-dimensions we can see that the time-complexity is now given by the equation:

$$Q(1, n) = O(logn)$$

$$Q(d, n) = Q(d-1, n) + O(logn) = O(d.logn)$$

The total number of processors ($P(d, n)$) required to search a range tree storing $n$ $d$-dimensional points and achieve the above time-complexity is given by the equation:

$$P(1, n) = 1$$

$$P(2, n) = 2logn - 2$$

$$P(d, n) = P(d-1, n)\,(2logn - 2) = O(\log^{d-1} n)$$

A simple observation that at most 2 nodes are selected at each of the heights from $h-2$ to 1 (Proposition 9.1) helps to reduce the above loose processor bound to a great extent. The number of data points belonging to dimension $i$ stored at node $v$ at height $r$ in the $i-1$-dimension range tree is $2^r$. Let $t(v)$ be the range tree corresponding to these points. The number of processors required to do a parallel search on $i$+1-dimensional points stored in $t(v)$ is $2.log(2^r) - 2$. We now present the estimation on the number of processors for $d = 3$. >From arguments above we have,

$$P(3, n) = 2.[2.log(2^{h-2}) - 2 + 2.log(2^{h-3}) - 2 + \cdots + 2.log(2^{h-(h-1)}) - 2]$$

$$P(3, n) = 2.\log^2 n - 14.logn + 8$$

In the above processor estimation we have not included processors needed to search a tree stored in the node $v$ at height $h-1$. It is not necessary to have additional processors for node $v$, since, if node $v$ is selected during the search none of the nodes in the subtree rooted at $v$ will be selected. Note that with at most $2.logn - 4$ processors the node $v$ can be processed. There are $\log^2 n - 7.logn + 4$ processors assigned to the nodes of the subtree rooted at $v$ and they are sufficient to process the tree belonging to node $v$. We now give a set of equations with which we can estimate the number of

processors needed to search $d$-dimensional data.

Let $T_0$ denote a complete binary tree on $n$-nodes with height $h$ $(2^h = n)$. The root of $T_0$ be at height $h$ and leaves are at height 1. Let $T_i$ denote a complete binary subtree on $2^{h-i}$ nodes and $a$, $a_1$, $a_2$, etc. are integer constants greater than zero. We define a function $E$ as follows:

$$E(m, [a_1.T_i + a_2.T_j + ...]) = E(m, a_1.T_i) + E(m, a_2.T_j) + \cdots$$

$$\text{For } i > 0,\ E(m, a.T_i) = \begin{cases} a.T_i & \text{if } i < 2.m - 4 \\ a.T_i + 2.T_{i+2} + 2.T_{i+3} + \cdots + 2.T_{h-1} & \textit{otherwise} \end{cases}$$

$$E(1, T_0) = 1$$

$$E(2, T_0) = T_0$$

$$E(d, T_0) = E(d, E(d-1, T_0))$$

The number of processors required is obtained by applying the following function $F$ to every term $a.T_i$ in the resultant equation.

$$F(a.T_i) = a.(2.log(2^{h-i}) - 2) = 2.a.(h - i - 1)$$

*3.2 Distribution of data among processors*

In the case of shared memory model data contained in the range-tree need not be distributed among processors and idle processors are allocated dynamically to the selected nodes of the tree. The dynamic assignment of processors to nodes is an over-

head to the system as it has to maintain a list of idle processors. Assuming that the selected nodes during the processing of dimension $i$ is $I$, the time taken to assign idle processors to the selected nodes is $O(I)$. Other obvious benefits of data distribution which include recovery and data reconstruction motivates the need for static assignment of processors to the nodes of the tree. In the previous subsection we have determined the upper bound on the number of processors required to do a parallel range search in $O(logn)$ time for the case of 2- and 3-dimensional data points. We now show how the processors are actually assigned and give the search strategy for the above cases. The case of $d$-dimension is a natural extension of the approach presented here.

We will call an assignment of processors to the nodes of the range-tree *proper* if the number of processors used in the assignment is less than or equal to the number of processors estimated in Section 3.1 to achieve a time-complexity of $O(d.logn)$, for a range search in $d$-dimensions. We will now present a proper assignment scheme for the case of 2-dimensions first. Let $T$ be a 1-dimensional range tree of height $h$. Starting with the leaves at height 1 to height $h-3$, we will allocate 2 processors to each of the heights, since, at most two nodes are selected from each of those heights by Proposition 9.1. If processors $p_i$ and $p_j$ are allocated at height $r$ $(1 \leq r \leq h-3)$, then starting from the left assign nodes at height $r$ $p_i$ and $p_j$, alternatively. This assignment would guarantee that the two selected nodes would be in different processors. Now, let $p_i$ and $p_j$ be allocated to height $h-2$. The first and the second pairs of nodes at height $h-2$ from the left are assigned $p_i$ and $p_j$, respectively. The two nodes at height $h-1$ are assigned the same processor that are assigned to their children. The root of $T$ is

assigned any of the processor assigned to its immediate child at height $h - 1$. The total number of processors used in the assignment is $(2.logn - 2)$. To search the third dimension, the tree corresponding to a node $v$ is assigned new set of processors the same way as described in the case of 1-dimensional range tree. For two nodes $v_1$ and $v_2$, their trees are assigned with the same set of processors if processor assigned to $v_1$ is the same as the processor assigned to $v_2$. Thus, the above assignment scheme uses exactly the same number of processors as estimated in Section 3.1. Figure 3.a gives the assignment of processors for the tree in Figure 2.b.

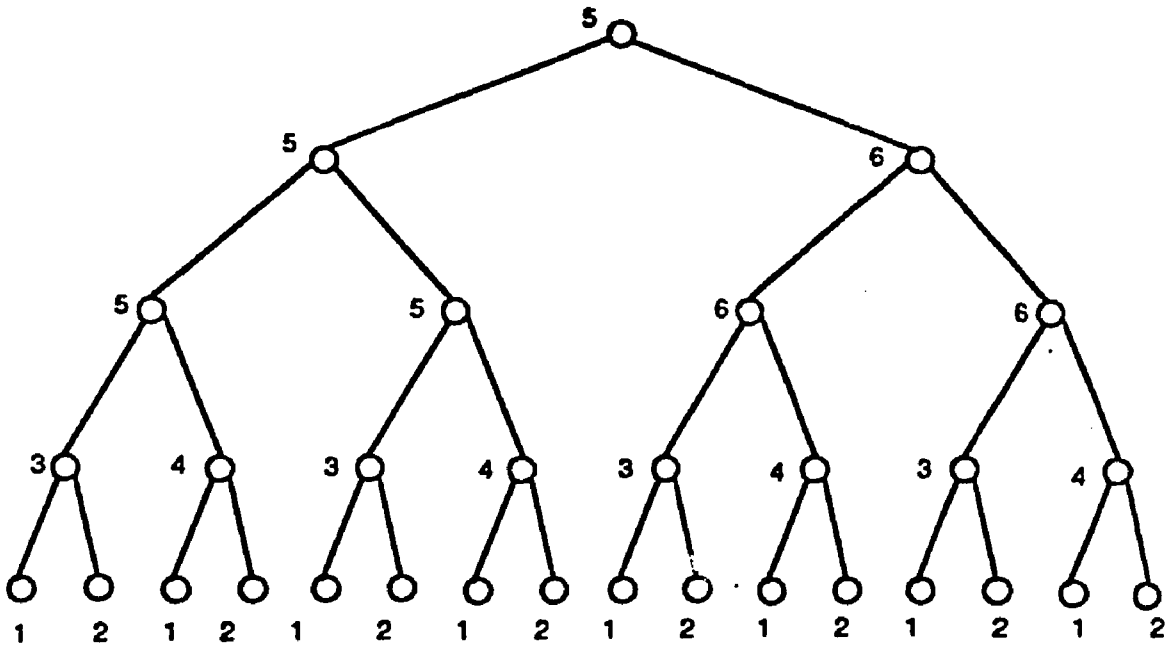



Figure 3.a - A proper assignment of processors for the tree in Figure 2.b.

The search strategy is very simple. Each processor assigned to a node $v$ at height $r$ is responsible for giving the search message to the appropriate processor at height $r-1$. The search message is sent from a processor $v$ at height $r$ to a processor at height $r-1$ if the query interval does not completely contain the interval $(B[v], E[v])$. If the interval containment is satisfied no more search message is issued from $v$ and the tree at $v$ is searched next. We know that each processor is assigned more than one tree node. The node interval to be chosen for comparison with the query interval and the processor to which the search message has to be sent are all done by the processor with the help of simple array indexes. We skip the details here.

**Theorem 9.1:** The range search on a 2-dimensional and 3-dimensional sets of $n$ points can be done in $O(logn)$ time with $(2.logn - 2)$ and $(2.\log^2 n - 14.logn + 8)$ processors, respectively. The sum of values in the range can also be done for the case of 2-dimension and 3-dimension in $O(logn)$ time with the above processor bounds.

*Proof*: The sum of values in a range can be retrieved using the values $S_v$ stored at each node of the range tree (see Section 2.). The rest of the result follows from the discussion above.■

We would like to end this section with the note that there can be more than one proper assignment scheme. Figure 9.3.b gives another proper processor assignment scheme for the tree in Figure 9.2.b.

## 9.5 Range searching on the Hypercube machine

We will now proceed to give details on how the nodes of the tree can be mapped on to the hypercube for efficient searching. First we will present the 2-dimensional case. A good mapping is one which minimizes the communication time in the hypercube. Consider the processor assignment discussed in the previous section. A mapping which takes the $i$th processor and maps it to the $i$th hypercube node would require a total communication time of $O(logn.loglogn)$, since we require $O(logn)$ processors for range searching in 2-dimensions and $O(logn)$ is the height of the range-tree corresponding to the first dimension. Hence the total search time for range-searching in two dimensions using a range-tree on a hypercube would be $O(logn.loglogn)$. We now present a mapping which would reduce the total search time to $O(logn)$ on the hypercube.

Consider the assignment of processors as discussed in the previous section to the nodes of the range tree corresponding to dimension 1. A processor at height $r$ ($p_i$) after checking its range will send the search message to another processor at height $r-1$ ($p_j$). If $p_i$ and $p_j$ are adjacent to each other in the hypercube the communication time is a constant, otherwise, it can be as high as $O(loglogn)$ the diameter of the hypercube. We now present a mapping (embedding) technique which gives constant-time communication time between processors in adjacent levels of the range tree.

It can be seen that a processor $p_i$ at height $r$ is adjacent to two processors at heights $r-1$ and $r+1$. Based on the processor assignment discussed earlier and the adjacency relationship between processors we form a graph $G$ called the *processor assignment graph.*

A processor assignment graph $G$ consists of $(2.logn - 2)$ nodes and is connected as follows. The graph $G$ consists of 4 chains $c_1$, $c_2$, $c_3$, and $c_4$. The chains $c_1$ and $c_2$ contains odd and even numbered processors respectively (will be referred as odd and even numbered nodes). Two odd or even numbered nodes are adjacent in their respective chains iff they belong to adjacent levels of the range tree. The chain $c_3$ ($c_4$) formed when an edge is drawn from every node $a$ in $c_1$ ($c_2$) to every node $b$ in $c_2$ ($c_1$), whenever $a$ and $b$ are in adjacent tree levels (see Figure 4.a).

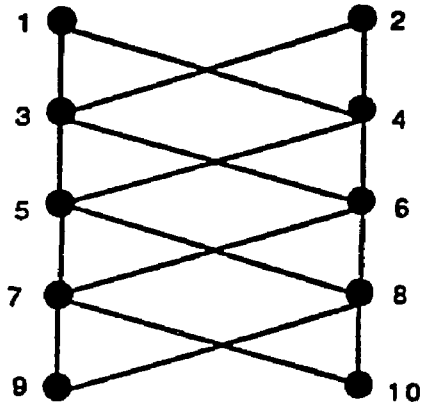



Figure 4.a - A processor assignment graph $G$.

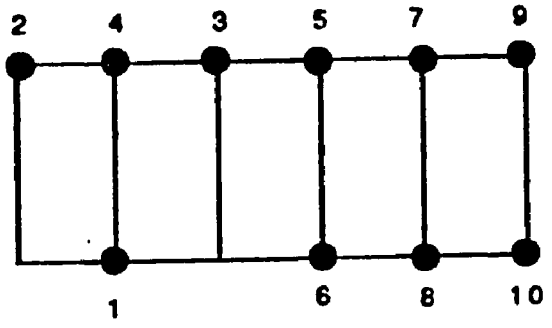



Figure 4.b - The embedding of the graph $G$ in Figure 4.a onto a grid.

We will show in Proposition 9.3 that the graph $G$ cannot be embedded in the hypercube with dilation 1 (i.e., all adjacent nodes in $G$ will not be adjacent when embedded in the hypercube). For dilation two embedding we require that the dimension of the cube be $O(logn)$, i.e., with a cube containing $2^{O(logn)}$ nodes. In this case the expansion, (i.e., the ratio of the number of hypercube nodes to the number of graph nodes) is exponential. The processor assignment graph can be embedded onto a binary tree with dilation 3. The binary tree can then be embedded onto an hypercube with an expansion of one and a dilation of 3 [Moni88]. Thus, the processor assignment graph can be embedded onto an optimal hypercube with dilation 6.

**Proposition 9.3**: The processor adjacency graph $G$ cannot be embedded on a hypercube with dilation one.

*Proof*: In a hypercube 2 nodes $a$ and $b$ can be adjacent at most to the two same set of two nodes $c$ and $d$. In $G$ two nodes $a$ and $b$ can be adjacent to the same 4 set of nodes. This implies either $a$ should be adjacent to $b$ or vice versa. Now, the dilation is 2.■

**Lemma 4.2**: The processor adjacency graph $G$ can be embedded on to an optimal hypercube with dilation 6.

*Proof*: First we will show the processor adjacency graph $G$ can be embedded onto a binary tree with dilation 3. Let $c_j(i)$ refer to the $i$th element in the $j$th chain. Make $c_1(2)$ the root $r$ of a binary-tree $T$. Make $c_2(2)$ the right child of $r$ and $c_1(1)$ and $c_2(1)$ the left and right children of $c_2(2)$. We use the following segment to construct the rest of the tree $T$.

1. $i = 3$;
2. Make $c_1(i)$ the left child of $r$;
3. $p = c_1(i)$;
4. Make $c_2(i)$ the right child of $p$;
5. Make $c_1(i+1)$ the left child of $p$;
6. If not all nodes in $c_1$ has been processed increment $i$

   and GOTO step 3;

Clearly, the above construction would obtain an embedding of $G$ on to a binary tree with dilation three. Figure 9.4. gives the graph $G$ and its corresponding binary-tree representation. The above segment of code guarantees that two adjacent nodes in $G$ are at most three distance apart in $T$. Now, using known algorithms [Moni88] we can embed $T$ onto an optimal hypercube with dilation 3. Thus, $G$ can be embedded onto an optimal hypercube with dilation 6.■

**Theorem 9.2**: Theorem 9.1. holds in the case when the processors are arranged as an hypercube architecture.

*Proof*: From Lemma 4.2 it is clear that the communication time for the search message to travel from one level to the adjacent one in the range tree is a constant. In the case of 2-dimensions after embedding $G$ in a hypercube with $(2.logn - 2)$ nodes, we can easily see that the range search can be done in $O(logn)$ time. In the case of 3-dimensional range search the processor bound can be achieved with several hypercube machines of different sizes as follows. With the availability of two cubes of size $(2.logn - 4)$, two cubes of size $(2.logn - 5)$ and so on, the 3-dimensional range search can

be done using a total of $(2.\log^2 n - 10.\log n + 12)$ processors. This is done by embedding each of one of the subtrees optimally onto their respective hypercubes.■

It would be interesting to see if given the availability of single hypercube can the search be carried efficiently. It turns out that it is possible and for a 3-dimensional range search using an hypercube with $O(\log^2 n)$ processors.

## 9.6 Processor reduction

In this section we will show that $(3/2.\log n - 1)$ processors are sufficient to effect a range search in $O(\log n)$ time for a set of points in the plane and thus saving $((\log n)/2 - 1)$ processors. The approach can be generalized to $d$-dimensions easily. The processor reduction is illustrated in the following example. Let $T_{256}$ be a range tree with 256 leaves. Time taken by a single processor to process the tree $T_{256}$ is in the worst case 8 units of time. Two $T_{16}$ trees can be processed sequentially by a single processor in 8 units of time. This means that with two processors, the tree $T_{256}$, and two $T_{16}$ trees can be processed in 8 units of time instead of using three processors and still requiring 8 units of time. We will generalize the above idea and estimate for a two dimensional range tree of height $h$. The range tree $T_{2^{h-2}}$ requires $h-2$ units of time. Since there are two such trees at height $h-2$, we will allocate two processors. For similar reasons we have to allocate two processors for each of the heights from $h-2$ to $(h-2)/2-1$. For heights from $(h-2)/2$ to 1 we allocate a single processor. The total number of processors allocated to the entire tree is now $(3/2.\log n - 1)$. Finding a proper processor assignment scheme with reduced number of processors is easy.

The estimation on the number of processors mentioned in Section 3. can further be reduced as the processors allocated during the processing of dimension $i$ can be used to process dimension $i+1$.

## 9.7. Summary

The problem of range search was solved in parallel using the range tree data structure. The nodes of the range tree were distributed among the processors in such a way that the search can be carried efficiently in parallel. It can be easily shown that our our algorithm is optimal for the chosen data structure in the case of arbitrary dimension $d = O(1)$. Based on the assignment of processors to the nodes of the range tree a processor assignment graph was created. The processor assignment graph was embedded onto an optimal hypercube for the execution of the range search without any communication overhead.

# Chapter Ten

## CONCLUSIONS

The contribution of the dissertation is two fold. In part I, we design algorithms for three generic problems in distributed systems: set manipulation, network structure recognition and facility placement. In part II, we design parallel algorithms for facility placement and $p$-center problems in trees using an EREW-PRAM model of computation. We also design efficient parallel algorithms for range searching using distributed data structures.

We present optimal distributed algorithms for recognizing rectangular-mesh networks. The time and message complexity of our algorithm is linear in the number of nodes in the network. We also lay the foundation for the recognition of 2-reducible, outer-planar and cactus graphs. These algorithms have a message complexity of $O(kn)$, where, $k$ is the number of isolated two degree nodes in the network.

Facility placement or $r$-domination is NP-complete on general networks. Many variations of the domination problem have been studied on restricted graph structures; e.g; trees, chordal, and interval graphs. We introduce the problem of reliable $r$-domination and design unified optimal distributed algorithms for the total, reliable and independent $r$-domination on trees. The time and message complexity of our algorithm is $O(n)$, where $n$ is the number of nodes in the tree.

In the domain of set manipulation we design optimal algorithms for determining the intersection of sets in a distributed environment, where each processor is assumed to have its own set. In many situations, the intersection would be null, where we

propose optimal algorithms for determining the mode (element occuring maximum number of times). The time and message complexity of our set intersection algorithm is $O(mn)$, where $m$ is the cardinality of the smallest set.

In part II of our research we design optimal algorithms for $r$-domination and efficient parallel algorithms for the $p$-center problems on trees. We also present an optimal algorithm for computing the maximum independent set on intervals in the Exclusive Read Exclusive Write - Parallel Random Access Memory (EREW-PRAM) model and such a set is required by our algorithms on trees. The $r$-domination problem on trees can now be solved in $O(logn)$ time with $O(n/logn)$ processors using the EREW-PRAM model.

A parallel algorithm for range searching is developed here using the concept of distributed data structures. We use the range tree proposed by Bentley as our data structure to be distributed. We show that $O(logn)$ search time can be effected for a range search on $n$ 3-dimensional points using $(2.\log^2 n - 14.logn + 8)$ processors, and this is optimal for the range tree distribution. We present a non-trivial implementation technique on the hypercube parallel architecture with which the above time and processor bound can be achieved without any communication overhead. Our algorithm can easily be generalized for the case of $d$-dimensional range search.

**Future research directions**

1. The problem of bipartite matching in a distributed environment is very important and there are no known algorithms to do that.

2. It is important to recognize planar graphs distributively, using $O(n)$ messages. This algorithm should be able to merge with the unified algorithms mentioned in this dissertation.
3. The outer planarity recognition algorithm presented in this dissertation can be improved to $O(n)$.
4. The algorithm for recognition of mesh connected networks can be extended to generalized boolean n-cubes.

**References**


[Atal89] Atallah, M.J., Cole, R., and Goodrich, M.T., "Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms," *Siam J. Comput.*, May 1989.

[Atta89] Attalah, M.J. and Chen, D.Z., "An Optimal Parallel Algorithm for the Minimum Circle-Cover Problem," *IPL*, vol. 32, pp. 159-165, 1989.

[Awer84] Awerbuch, B., "An Efficient Network Synchronization Protocol," *ACM Symposium on Theory of Computing*, April 1984.

[Awer85] Awerbuch, B. and Gallager, R., "Distributed BFS Algorithm," *IEEE*, 1985.

[Baru89] Baru, C. K. and Frieder, O., "Database operations on Cube-Connected Multicomputer System," *IEEE Trans. on Computers*, vol. 38, no. 6, pp. 920-927, June 1989.

[Bent80a] Bentley, J.L., "Multidimensional Divide-and-Conquer," *CACM*, vol. 23, no. 4, pp. 214-229, April 1980.

[Bent79] Bentley, J.L. and Friedman, J.H., "Data Structures for Range Searching," *ACM Surveys*, vol. 11, no. 4, December 1979.

[Bent80b] Bentley, J.L. and Maurer, H.A., "Efficient Worst-Case Data Structures for Range Searching," *Acta Informatica*, vol. 13, pp. 155-168, 1980.

[Bert87] Bertossi, A.A. and Bonuccelli, M.A., "Some Parallel Algorithms on Interval Graphs," *Discrete Applied Mathematics*, vol. 16, pp. 101-111, 1987.

[Chan85] Chandy, K.M. and Lamport, L., "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans Comput Systems*, vol. 3, no. 4, pp. 63-75, 1985.

[Chan82a] Chandy, K.M. and Misra, J., "Distributed Computation on graphs: Shortest Path Algorithms," *CACM*, vol. 25, 11, pp. 833-837, 1982.

[Chan83] Chandy, K.M., Misra, J., and L.Haas, "Distributed Deadlock Detection," *ACM TOCS*, vol. 1,2, pp. 144-156, 1983.

[Chan82b] Chang, E.J.H., "Echo Algorithms: Depth Parallel Operations on General Graphs," *IEEE Trans. on Software Eng.*, vol. SE-9 no. 4, pp. 391-401, July 1982.

[Chan89] Chang, G.J., "Labeling Algorithms For Dominating Problems In Sun-Free Chordal Graphs," *Disc. Appl. Math.*, vol. 22, pp. 21-34, 1988/89.

[Chaz86] Chazelle, B., "Filtering Search: A New Approach To Query-Answering," *Siam J. Comput.*, vol. 15, no. 3, pp. 703-724, August 1986.

[Chin85] Chin, F. and Ting, H.F., "An almost linear time and $O(n \log n + e)$ messages distributed algorithm for minimum-weight spanning trees," *Proc. 26th IEEE FOCS*, pp. 250-256, 1985.

[Cock75] Cockayne, E.J., Goodman, S., and Hedetneimi, S.T., "A Linear Algorithm For The Domination Number Of A Tree," *Information Processing Letters*, vol. 4, pp. 41-44, 1975.

[CORN85] CORNAFION, "Distributed Computing Systems," *Elsevier Science Publishers B.V.*, 1985.

[Corn86] Corneil, D.G. and Stewart, L.K., "Dominating Sets In Perfect Graphs," *Submitted for Publication, University of Toronto*, 1986.

[Dech87] Dechter, R. and Kleinrock, L., "Broadcast Communication and Distributed Algorithms," *IEEE Trans. on Computers*, vol. C35, 3, pp. 210-219, 1987.

[Dehn88] Dehne, F. and Stojmenovic, I., "An $O(\sqrt{n})$ Time Algorithm for the ECDF Searching Problem for Arbitrary Dimensions on a Mesh-Of-Processors," *IPL*, vol. 28, 1988.

[Dijk80] Dijkstra, E.W. and C.S.Scholten, "Termination Detection for Diffusing Computations," *Inform. Process. lett.*, vol. 11, 1, pp. 217-219, 1980.

[Ecks79] Eckstein, D.M., "Simultaneous Memory Access," *Dept. Comput. Sci., Iowa State Univ., Iowa City, Tech. Rep. TR-79-6*, 1979.

[Elli85] Ellis, C.S., "Distributed Data Strucutures: A Case Study," *Int'l Conf. on Parallel and Distributed Computing*, 1985.

[Film84] Filman and Friedman, "Coordinated Computing: Tools and Techniques for distributed software," *McGraw-Hill Book Company*, 1984.

[Fred85] Frederickson, G., "A Single Source Shortest Path Algorithm for a Planar Distributed Network," *Proc. Symp. Theoret. Aspects Comput. Sci.*, Jan 1985.

[Gall82] Gallager, R.G., "Distributed Minimum Hop Algorithms," *M.I.T. Technical Report, LIDS-P-1175*, Jan 1982.

[Gall83] Gallager, R., Humblet, P., and Spira, P., "A Distributed Algorithm for Minimum-Weight Spanning Trees," *ACM TOPLAS*, vol. 5, no. 1, pp. 66-77, 1983.

[Gare79] Garey, M.R. and Johnson, D.S., "Computers and Intractability: A Guide to the Theory of NP-Completeness," *W.H. Freeman and Company, San Fransisco*, 1979.

[Gazi87] Gazit, H., Miller, G.L., and Teng, S.H., "Optimal Tree Contraction in the EREW Model," *Proceedings of the Princeton Workshop on Algorithms, Architectures, and Technical Issues for Model of Concurrent Computation*, 1987.

[Hara69] Harary F., "Graph Theory," *Addison-Wesley, Reading, MA*, 1969.

[Hela88] Helary, J.M., Plouzeau, N., and Raynal, M., "A Distributed Algorithm for Mutual Exclusion in an Arbitrary Network," *The Computer Journal*, vol. 31, No.4, pp. 289-295, 1988.

[He88] He, Xin and Yesha, Yaacov, "Binary Tree Algebraic Computation and Parallel Algorithms for Simple Graphs," *J. of Algorithms*, vol. 9, pp. 92-113, 1988.

[He90] He, Xin and Yesha, Yaacov, "Efficient Parallel Algorithms for $r$-Dominating Set and $p$-Center Problems on Trees," *Algorithmica*, vol. 5, pp. 129-145, 1990.

[Hirs80] Hirschberg, O.S. and Sinclair, J.B., "Decentralized Extrema Finding in Circular Configurations of Processors," *CACM*, vol. 23, 11, pp. 627-628,

1980.

[Iyen88] Iyengar S.S., Rao N.S.V., Kashyap R.L., and Vaishvani V.K., "Multidimensional Data Structures: Review and Outlook," *Advances in Computers*, vol. 27, pp. 70-119, 1988.

[Iyen90] Iyengar, S.S. and Subbiah, R., "Optimal Distributed Algorithm for determining the Mode of a distribution.," *Tech Report, Robotics Research Laboratory, LSU.*, 1990.

[Kari79] Kariv, O. and Hakimi, S.L., "An Algorithmic Approach to Network Location Problems I: the $p$-centers," *SIAM J. Appl. Math*, vol. 37, no. 3, pp. 513-538, 1979.

[Karl88] Karlsson, R.G. and Overmars, M.H., "Normalized Divide-and-Conquer: A Scaling Technique for Solving Multi-Dimensional Problems," *IPL*, vol. 26, pp. 307-312, 1987/88.

[Karp72] Karp, R.M., "Reducibility Among Combinatorial Problems," *Complexity* of Computer Computations, 1972.

[Karp90] Karp, R.M. and Ramachandran, V., "Parallel Algorithms for Shared-Memory Machines," *Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity, edited by Jan Van Leeuwen, Elsevier and MIT Press.*, 1990.

[Katz88] Katz, M.D. and Volper, D.J., "Geometric Retrieval in Parallel," *Journal of Parallel and Distributed Computing*, vol. 5, pp. 92-102, 1988.

[Keil86] Keil, J.M., "Total Domination In Interval Graphs," *Information Process-*

*ing Letters*, vol. 22, pp. 171-174, 1986.

[Kora84] Korach, E., Rotem, D., and Santoro, N., "Distributed Algorithms for finding centers and Medians in Networks," *ACM Toplas*, vol. 6,3, pp. 380-401, 1984.

[Laks87] Lakshmanan, K.B., Meenakshi, N., and Thulasiraman, K., "A Time Optimal Mesage-Efficient Distributed Algorithm for Depth-first Search," *Inform. Process. Lett.*, vol. 25, (2), pp. 103-109, May 1987.

[Lamp78] Lamport, L., "Time, Clocks and Ordering of Events in a Distributed System," *CACM*, vol. 21(7), pp. 558-565, 1978.

[Lamp] Lamport, L. and Schnieder, F.B., "Paradigms for Distributed Programs in distributed Systems," *LNCS 190*, vol. Springer-Verlag 1985, pp. 431-480.

[Lask83] Laskar, R. and Pfaff, J., "Domination And Irredundance In Split Graphs," *Tech. Rept. 430, Clemson University, Clemson, SC*, 1983.

[Lask84] Laskar, R., Pfaff, J., Hedetniemi, S.M., and Hedetniemi, S.T., "On the Algorithmic Complexity of Total Domination," *SIAM J. Alg. Disc. Meth.*, vol. 5, pp. 420-425, 1984.

[Lehm81] Lehman, P. and Yao, S.B., "Efficient locking for concurrent operations on B-trees," *ACM TODS*, vol. 6, no. 5, pp. 650-670, December 1981.

[Less81] Lesser, V.R. and Corkill, D.D., "Functionally Accurate, Cooperative Distributed Systems," *Man and Cyber*, vol. SMC-11, No.1, pp. 81-89, Jan 1981.

[Lync85] Lynch, N.A. and Fischer, M.J., "On Describing The Behavior And Implementation of Distributed Systems," *ACM TOCS*, vol. 3, pp. 145-159, 1985.

[Manb89] Manber, Udi, "Introduction to Algorithms," *Addison-Wesley Publishing Company*, 1989.

[Mill85] Miller, G.L. and Reif, J., "Parallel Tree Contraction and Its Applications," *Proc. 26th IEEE Symposium on Foundations of Computer Science*, pp. 478-489, 1985.

[Mill87] Miller, R. and Stout, Q. F., "Mesh Computer Algorithms for Line Segments and Simple Polygons," *Int'l Conf. on Parallel Processing*, pp. 282-285, 1987.

[Misr82] Misra, J. and Chandy, K.M., "A Distributed Graph Algorithm: Knot Detection," *ACM TOPLS*, vol. vol. 4, no. 4, pp. 678-686, Oct 1982.

[Moha90] Mohan, S., "Efficient Distributed Algorithms for Network Facility Problems," *Ph.D. Dissertation*, May 1990.

[Moha89] Mohan, S.B., Iyengar, S.S., and Narasimha, M.K., "An Optimal Distributed Depth-first Search Algorithm," *Inform. Process. Lett.*, Nov 1989.

[Moit87] Moitra, Abha and Iyengar S.S., "Parallel Algorithms for Some Computational Problems," *Advances in Computers*, vol. 26, pp. 94-153, 1987.

[Moit88] Moitra, A. and Johnson, R., "PT-Optimal Algorithms for Interval Graphs," *In Proc. of the 26th Allerton Conference on communication, control, and computing, Urbana-Champaign, Illinois.*, pp. 274-282, Oct.

1988.

[Moni88] Monien, B. and Sudborough, I.H., "Simulating Binary Trees on Hypercubes," *3rd AWOC*, 1988.

[Mu86a] Mu, Z. and Chen, M.C., "Communication-Efficient Distributed Data Structures on Hypercube Machines," *Department of Computer Science, Yale University*, 1986.

[Mu86b] Mu, Z. and Chen, M.C., "Communication-Efficient Distributed Data Structures on Hypercube Machines," *Department of Computer Science, Yale University*, 1986.

[Omie89] Omiecinski, E. and Tien, E., "A Hash-Based Join Algorithm for a Cube-Connected Parallel Computer," *IPL*, vol. 30, pp. 269-275, March 1989.

[Park81] Parker, D.S. and Samadi, B., "Adaptive Distributed Minimal Spanning Tree Algorithms," *Proc 1st Symp. reliability Distrib. Software databases*, 1981.

[Pipp87] Pippenger, N., "On Simultaneous Resource Bounds," *Proc. STOC., New York*, May 1987.

[Prep85] Preparata, F.P. and Shamos, M.I., "Computational Geometry : An Introduction," *Springer-Verlag, New York*, 1985.

[Radh90a] Radhakrishnan, Sridhar, Iyengar, S.S., and Subbiah, Rajanarayanan, "Range Search In Parallel Using Distributed Data Structures," *Int'l PARBASE-90 Conf., Miami, Florida, To Appear in Journal of Parallel and Distributed Computing*, 1990.

[Radh90b] Radhakrishnan, S., Subbiah, R., Iyengar, S.S., and Narayan, C., "Optimal Algorithm for r-Dominating Set Problems On Trees In The EREW-PRAM Model," *in preparation*, 1990.

[Rama89] Ramarao, K.V.S., "Distributed Algorithms For Network Recognition Problems," *IEEE Trans. on Computers*, vol. 38 No.9., Sept 1989.

[Rama88] Ramarao, K.V.S., Daley, Robert, and Melhem, Rami, "Message Complexity of the Set Intersection Problem," *IPL*, vol. 27, pp. 169-174, 1988.

[Rama86] Ramarao, K.V.S. and Venkatesan, S., "On Finding and Updating Shortest Paths Distributively," *Proc. 24th Allerton Conf.*, 1986.

[Rayn85] Raynal, M., "Algorithms for Mutual Exclusion," *MIT Press*, 1985.

[Rayn88] Raynal, M., "Networks and distributed computation: Concepts, Tools and Algorithms," *MIT Press*, 1988.

[Scot87] Scott, L.R., Boyle, J.M., and Bagheri, B., "Distributed Data Structures for Scientific Computation," *Departments of Computer Science and Mathematics, Pennsylvania State University*, 1987.

[Sega83] Segall, A., "Distributed Network Protocols," *IEEE Trans. Inform. Theory*, vol. IT-29, pp. 23-25, 1983.

[Slat76] Slater, P.J., "*R*-Domination in Graphs," *J. Asso. Comp. Mach.*, vol. 23, pp. 446-450, 1976.

[Stoj87] Stojmenovic, I., "Computational Geometry on a Hypercube," *Int'l Conf. on Parallel Processing*, pp. 100-103, 1987.

[Subb90] Subbiah, R., Iyengar, S.S., and Sridhar, R., "Unified Message Optimal Distributed Algorithms for the Total, Reliable and Independent r-Dominating Set Problems," *Submitted to IEEE Trans. on. Comput.,*, 1990.

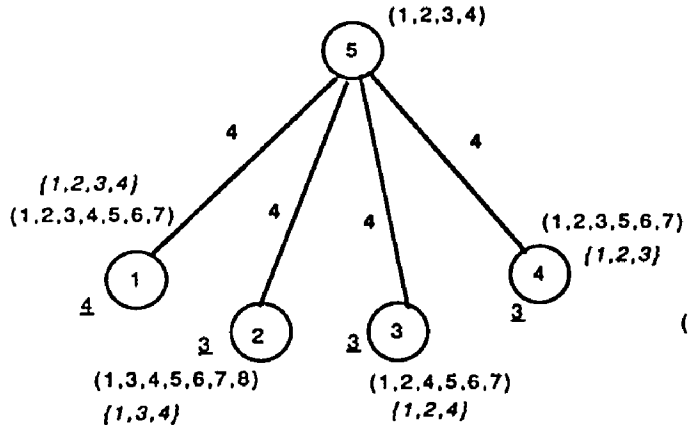
[Upfa84] Upfal, E. and Wigderson, A., "How to Share Memory in a Distributed Environment," *Proc. 25th Annu. IEEE Symp. Foundations Comput. Sci., West Palm Beach, FL*, pp. 171-180, October 1984.

[Wage] Wagers, Manfred, "Recognizing Outerplanar Graphs in Linear Time," *Graph-theoretic concepts in Comp. Science - International workshop*, 1986 .

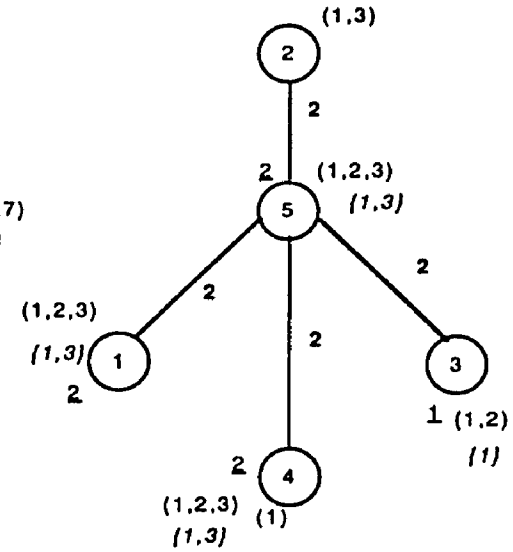
# Appendix A

## Algorithm of Ramarao et al. [4]

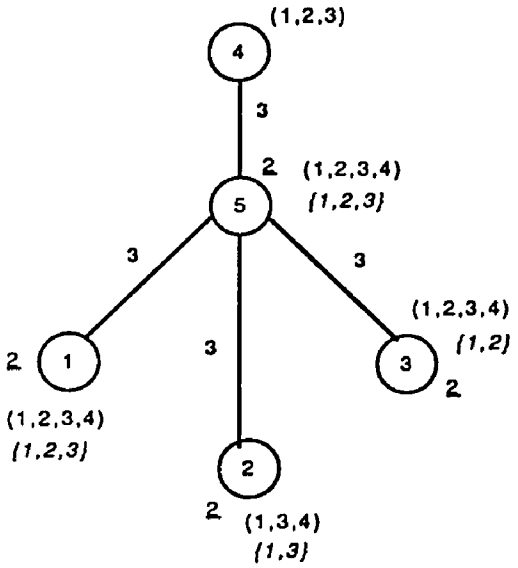
### PHASE I



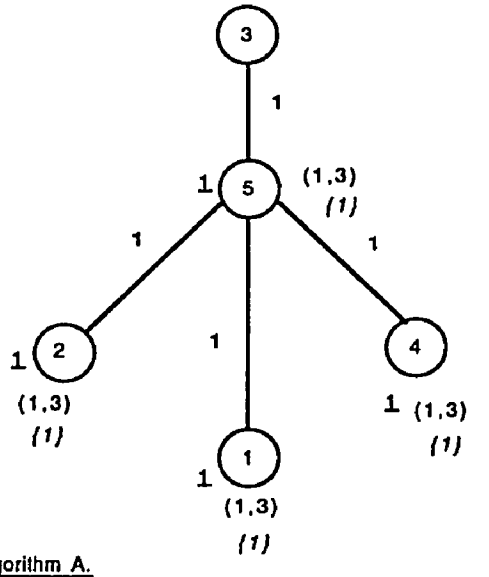

### PHASE III




### PHASE II




### PHASE IV



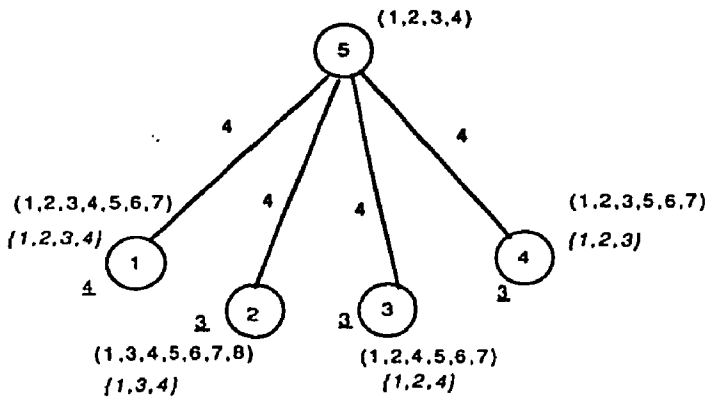

The total number of messages used by algorithm A.

| | | |
|---|---|---|
| Phase I | 4 + 4 + 4 + 4 + 4 | = 20 |
| Phase II | 3 + 3 + 3 + 3 + 4 | = 16 |
| Phase III | 2 + 2 + 2 + 2 + 4 | = 12 |
| Phase IV | 1 + 1 + 1 + 1 + 4 | = 8 |
| | Total | = 56 |

The total number of time units used by algorithm A

4 + 1 + 6 + 2 + 4 + 2 + 2 + 2 = 23 units.

## The Proposed Algorithm




**PHASE I**

The total number of messages used by the proposed algorithm is

4 + 4 + 4 + 4 + 4 + 3 + 3 + 3 = 29
Broadcast = 1 + 1 + 1 + 1 = 4
Total = 33

Total time units = 4 + 4 + 1 = 9 units

# VITA

Rajanarayanan Subbiah was born in Madras, Tamilnadu, India on September 11, 1963. He obtained his Master of Science (Tech) in computer science from Birla Institute of Technology and Science, Pilani, Rajasthan, India in 1985. He completed his Master's degree in Systems Science at Louisiana State University, Baton Rouge, Louisiana in 1989.

His research interests include parallel and distributed algorithm, fault tolerant computing, graph theory and operations research, computational geometry, computer vision and artificial intelligence.

**DOCTORAL EXAMINATION AND DISSERTATION REPORT**

**Candidate:** Rajanarayanan Subbiah

**Major Field:** Computer Science

Title of Dissertation: Parallel and Distributed Algorithms for a Class of Graph Related Computational Problems

**Approved:**

Major Professor and Chairman

Dean of the Graduate School

**EXAMINING COMMITTEE:**

**Date of Examination:**

July 30, 1991.