1993

High Performance Software Reconfiguration in the Context of Distributed Systems and Interconnection Networks.

Vinayak Gajanan Hegde
Louisiana State University and Agricultural & Mechanical College

Order Number 9401531

High performance software reconfiguration in the context of distributed systems and interconnection networks

Hegde, Vinayak Gajanan, Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1993

U·M·I

300 N. Zeeb Rd.
Ann Arbor, MI 48106

HIGH PERFORMANCE SOFTWARE RECONFIGURATION IN THE CONTEXT OF DISTRIBUTED SYSTEMS AND INTERCONNECTION NETWORKS

A Dissertation

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

in

The Department of Computer Science

by
Vinayak Gajanan Hegde
B.E., University of Mysore, India, 1984
M.S., Indian Institute of Technology, Madras, India, 1988
May 1993

ACKNOWLEDGEMENTS

My sincere gratitude and appreciation goes to Prof.S. Sitharama Iyengar, who gave me a chance to pursue my doctoral program. His encouragement and support helped me through out my stay at LSU.

I would like to thank my committee members: Prof. Donald Kraft, Prof. Bush Jones, Prof.Doris Carver, and Prof Suresh Rai for all their encouragement and support. My association with Prof. Hoppe was extremely fruitful to latch on to my research topic: distributed systems and operating systems. I will remember him, also for his sense of humor, patience, involvement, and gentleness.

Without friends like Phatak, Mohan, Krishnakumar, Raj, Phil, Sridhar, Sankar, Rahgu, and Daryl, my memories with Robotics lab could not be so dear to me. My roommates Subba Rao, Ananth Prasad, Venky and Vijayan, and my friends at Varsity Village including Sai, Gautam, and Harish have made many occasions very special to me. My association with Larry will be green in my memory for a long time to come. I am lucky to have met wonderful families including Bhat's, Vani's, Rama's, and Natraj's. Main sources of my moral strength and of my desire to be worthwhile are my family as well as so many wonderful people from my home town. If my achievements make all these people proud, my work will mean much more to me.

Table of Contents

ABSTRACT

Designed algorithms that are useful for developing protocols and supporting tools for fault tolerance, dynamic load balancing, and distributing monitoring in loosely coupled multi-processor systems. Four efficient algorithms are developed to learn network topology and reconfigure distributed application programs in execution using the available tools for replication and process migration.

The first algorithm provides techniques for transparent software reconfiguration based on process migration in the context of quadtree embeddings in Hypercubes. Our novel approach provides efficient reconfiguration for some classes of faults that may be identified easily. We provide a theoretical characterization to use graph matching, quadratic assignment, and a variety of branch and bound techniques to recover from general faults at run-time and maintain load balance.

The second algorithm provides distributed recognition of articulation points, biconnected components, and bridges. Since the removal of an articulation point disconnects the network, knowledge about it may be used for selective replication. We have obtained the most efficient distributed algorithms with linear message complexity for the recognition of these properties.

The third algorithm is an optimal linear message complexity distributed solution for recognizing graph planarity which is one of the most celebrated problems in graph theory and algorithm design. Recently, efficient shortest path algorithms are developed for planar graphs whose efficient recognition itself was left open. Our algorithm also leads to designing efficient distributed algorithm to recognize outerplanar graphs with applications in Hamiltonian path, shortest path routing and graph coloring. It is shown that efficient routing of information and distributing the stack needed for for planarity testing permit local computations leading to an efficient distributed algorithm.

The fourth algorithm provides software redundancy techniques to provide fault tolerance to program structures. We consider the problem of mapping replicated program structures to provide efficient communication between modules in multiple replicas. We have obtained an optimal mapping of 2-replicated binary trees into hypercubes. For replication numbers greater than two, we provide efficient heuristic simulation results to provide efficient support for both 'N-version programming' and 'Recovery block' approaches for software replication.

CHAPTER 1

INTRODUCTION

We are willing to bet that it is a mistake, at least on the long term, to allow our thinking to be cramped by preconceptions about what is and is not implementable. If we are mistaken in our optimism, then at least we will have examined some interesting problems in a fresh light. We insist ... that languages do not become interesting because we can implement them. Implementation problems become interesting when new languages seem worthy of study

-- David Gelernter, ACM TOPLAS, Vol.7(1), January 1985, pp. 80-112.

1.1. Overview

In the present age of advanced technology, use of computers in home, in work place, and in industry is very common. Introduction of computers for such wide spread applications has increased the demand for the computer to be more and more user friendly, reliable, efficient and cost effective. Computers have been used in critical and sensitive applications by providing multiple resources and ensuring the certainity of the environment at great cost. Attempts are necessary to develop computers which operate in realistic uncertain environments and perform computationally intensive real tasks. Research works about computers operating in faulty environments are no more only of theoretical interest. Due to the recent improvements in the understanding about the dynamic needs of realistic applications coupled with the availability of basic supporting mechanisms for cooperative computing, research in the direction of improving performance of fault tolerant parallel and distributed computing is encouraged more than ever before.

Apart from the need for making our dependence on computers more secure, other reasons for the successful application of co-operating solutions to computing include the recent trends of reducing costs of electronic hardware, possible increase

in performance due to the partitioning effect on seemingly difficult large problems, advantages of increased resource sharing, and recent stagnation of technology to increase the clock rates of individual computers. Instead, improvements in performance is achieved by aggregating the cost effective multiple machines on a single problem. Unlike the stand alone computer which can work in a fixed environment, multiple computers may be used in different combinations to interact in a dynamic environment with much more versatility. Significantly advanced fault tolerance, resource sharing, and communication supports can help multi-computers to behave like a single, powerful virtual machine. With increasing degree of difficulty of advanced systems, it is necessary to pursue software based system reliability rather than the current focus of hardware redundancy based approaches.

The research with multiple computers have been mainly focussed on tightly coupled parallel machines called interconnection networks and the loosely coupled asynchronous machines called distributed systems. Interconnection networks are designed emphasising the need for intensive computation that may be divided and executed in parallel achieving high performance. These networks use the models using parallel algorithms which optimize the overall computation time and the required space. Communication and synchronization is of secondary importance since many of these networks are based on shared memory and thus it is assumed that any processor can access any shared memory location in constant time after resolving contention. On the other hand, the distributed systems are designed with emphasis on the communication oriented computation where in messages are used for communication and synchronization. Hence distributed algorithms optimize the number of messages that are needed to be exchanged to solve a distributed problem cooperatively without using any shared memory, but only exchanging messages. The specific characteristics of one of these models may be better suited for some applications, but each of them have their merits for different applications.

Recent advances in network based models have introduced islands of sites connected over a geographically distributed configurations, wherein each site may consist of nodes including interconnection networks and workstations. Such a configuration allows exploiting the merits of both parallel and distributed computers to execute tasks with varied complexity which were hitherto too large to solve. It is also possible to exploit special purpose abilities of multi-computers in such a system so that specific functions of a given application may use the properties of both parallel and distributed systems to solve the overall problem. Even distributed shared memory is implemented in loosely coupled message systems using communication system for simulating shared address space [1]. Such advancing technologies have made the programming of distributed applications more attractive than ever.

As the technology of networking computers grow, the interconnections among them become more complex. A fully expanded network topology can consist of dozens of Local Area Networks interconnected by numerous network routers and bridges. Based on Open Systems Interconnect (OSI) model [2] , for example, services for message passing route communication packets of the higher layers through intermediate points over long distance in a typical internetwork. With all the communication protocols in place, application programs may be distributed and remotely executed making resource sharing possible.

As the complexity of systems increase, so does the need for the reliability of the system to be more than the reliability of its components. While building sophisticated systems for any critical applications, it is possible to capture the nature of the specialized domain in to a more flexible computational framework which can be reconfigured dynamically to tolerate partial failures. Such a system will improve the ability to respond to varying needs of computation and failure. However, further research is necessary to understand the mechanics of fault tolerant programs and to make the process of reconfiguration more efficient. Since the general problem of

program fault tolerance is still in early phase of research, providing efficient solutions for specific applications and system re-configuration is currently pursued.

This research is mainly concerned with message passing models used in interconnection networks such as hypercubes and also in distributed systems connected over a computer network. We refer to such an environment of cooperating multicomputers as the distributed systems. Principal concern of this research is to provide standard software interface support in order to develop distributed application programming services to improve the performance and the reliability of applications in execution in the face of failures of components in the system.

1.2 Fault Tolerance in Distributed Systems

Distributed system is a set of physically distinct computational devices working on a common problem, while their operation is coordinated via communication channels connecting some or all of these devices. Each computing device has certain processing and memory capabilities, and is preprogrammed to perform its part of computation, as well as send and receive control or data messages over the communication links. Since the system is loosely coupled via a communication network, and delays occur because of congestion and traffic , besides the propagation delay, the fundamental characteristics of distributed systems is that some of its components may fail while others are still working, and also that the messages in transit are subject to variable delays and failures, unlike in the tightly coupled systems.

Generic problems and architectural issues related fault tolerant distributed systems are studied by Cristian[3]. Developing strategies and methodologies for fault tolerant applications using coordinated computing spans a broad range of issues: distributed control, task decomposition, communication, synchronization, performance monitoring, and distributed system design. Since the main concern of this dissertation is to design high performance algorithms in the domain of distributed systems,

our main focus of fault tolerance in distributed systems in the context of this research is to develop a co-operative computing system that is resilient to failures as well as changing needs of demand. This research addresses *only* the issues related to software approaches to fault tolerance.

We concentrate on the design of distributed network protocols for the software reconfiguration of applications in execution. Some of the functions of these protocols include collecting performance parameters using graph traversals, constructing distributed depth first trees or spanning trees, and obtaining topological configurations of a distributed system which may be used for reconfiguring the applications. Two classes of problems, network traversal and global state determination, are considered fundamental, as discussed by Raynal [4]. Distributed algorithms for a variety of problems such as distributed mutual exclusion, leader election, termination, failure detection, synchronization, and graph algorithms including those for finding shortest path, minimum spanning trees, connected components, graph recognition etc., have been extensively studied and reported in literature [5,6,7,8].

1.2.1 Objectives of software fault tolerance

Very little theory is developed to optimize the cost, dependability, and functional correctness parameters of redundancy and design diversity techniques for fault tolerance. Software techniques are being actively pursued since there is a need for building systems with higher reliability than the reliability of its hardware components. Main objectives of software fault tolerance is to make the system more *reliable* and *available*[9,10]. A system is reliable if it does not corrupt or loose any data. Availability on the other hand is about the working status of being ready whenever it is needed to be used. Because of the multiplicity of components in a distributed system, it can be made more reliable as well as available using hardware as well as software techniques. Hardware failures are due to the failure of its components, while

software failures may be either due to implementation error or specification error. In this research, we restrict fault tolerance to software solutions to provide high-level architecture independent facilities for a distributed program in execution to continue operating even in the face of the failure of one or more of the hardware components of the system. Some of the systems which provide this kind of software solutions may be found in CLOUDS[11], ARGUS[12], and CIRCUS[13].

1.2.2 Difficulties of designing fault tolerance

It is difficult to design a unified framework for a fault tolerant distributed systems. Some of the tasks which are considered to be trivial in sequential computing can be surprisingly difficult in concurrent multi-processing environment because of the interleaving of events and the nature of interaction between the processes. The tasks involving synchronization and communication between processes are especially difficult to automate. Consider, for example, the task of moving a process from a given machine to a specified target machine on a distributed network. Given that the multiprocessing systems save the process context for multi-programming, the problem looks simple at the first sight to move the context saved from one machine to the other. However, from a point of view of the correctness of program, this raises sub-problems of messages in transit, varying clock speeds, and system specific variables such as an access to a file, which are difficult to be updated.

It is also necessary to gather the dynamic information such as changing topology of the network for routing information. Even if the information about the configuration is completely known, it is necessary to derive a solution based on the present situation to achieve an optimal utilization of the resources. If the information available is only partial, it is necessary to generate a plan to collect the necessary information using as minimum effort as possible. All these tasks must be carried out in real time with acceptable efficiency and reliability.

Some of the difficulties in the design of cooperative fault tolerant distributed systems are

1) To represent the real working environment as a model which may be used by the system to recognize a match between the given application and the currently available system in which some of the components have failed.

2) To describe the distributed tasks so that system parameters may be synthesised about a program in execution from a set of primitive functional routines which are compiled and stored in a library which will be used at run time to collect the information of interest needed for reconfiguration.

3) To develop a system which can assess suitability of planned degree of fault tolerance and modify actions when relevant to the real task environment, i.e. to detect errors about the planned faults and recover from them automatically using reasoning about the performance measures.

4) To develop efficient selective fault tolerant algorithms which make a critical process of applications more resilient when necessary, rather than providing a less efficient, and probably incomplete, general solution applicable to the whole problem.

5) To develop a computer system for controlling and for providing the intelligence and support necessary for executing the given tasks to completion despite the uncertainities involved.

1.2.3 Models of software fault tolerance

Software fault tolerance models provide software support for making use of redundant resources so that programs may continue to operate in the case of partial failures of some of the system components. Basic models of software fault tolerance include state machine model,[14,15], nested atomic transaction model,[16], remote

procedure call model[17], and fail-stop processor model[18]. Various approaches for software fault tolerance may use a mixture of these basic models to obtain design diversity. These models are explored in more detail in chapter 2.

First approach to classify the models is to look at the kinds of supports provided: distributed programming language level, distributed operating system level, or run-time support level. Second approach to classify the models is based on the redundancy being used: replication, or process migration. Third approach to classify is based on the basic methodology: transparent support, or fault tolerant tools approach.

a) Language support:

Important variations supports for nested atomic transactions are provided in Guardians and Actions as in ARGUS[19], optimistic recovery in[20], Communicating Sequential Processes with and without voting[21,22] using synchronous message passing and watchdog processor based general rollback techniques, communication at the end of replicated processes as in SIFT[23]. Also object based fault tolerance is provided in EMERALD[24] where a pessimistic synchronization with roll-back using check-points is presented. TABS/ CAMELOT[25] supports nested transaction scheme.

b) Operating System Support

Preemptable remote execution facilities are provided in V-System[26]. Client-Server based time-out approach with orphan invocation is provided in AMOEBA[27] along with large-grain object support. CHORUS[28] also uses time-out for detecting invocation failures. Fault tolerant distributed file service. based on replicated processes is supported in[13].

c) Run-time Support

This approach provides raising exceptions for predetermined types of faults so that recovery routines for exception handling may be carried out. This technique is also implemented in complete replication of ADA programs with remote tasking [29].

d) Replication based Fault Tolerance

As mentioned above, many object based systems [24] as well as process based systems as in Fault Tolerant Concurrent C[30], and in ADA [29] use replication to provide redundancy in order to tolerate faults.

e) Process Migration

An excellent survey of process migration support mechanisms and approaches may be found in [31]. Transferring sufficient amount of a process's state from one machine to the other for execution is called process migration. This technique along with checkpointing is used for fault tolerance in DEMOS/MP[32] and in LOCUS[33].

f) Transparent Approach

This technique refers to the ability of the systems to provide fault tolerance without any special effort from the user's side. Process replication, many variations of checkpointing and recovery supports are provided in Sender Based Message Logging [34] and in Optimistic recovery[35].

g) Tools approach

This approach provides programming facilities so that application programmer or the system on its own, may provide selective fault tolerance. This approach needs more effort on the part of the programmer to specify which part of application is

critical. But offers greater efficiency. This approach is taken to design selective replication in Fault Tolerant Concurrent C[30], applications management in ISIS[36]. A very interesting variation of application directed approach where in the processes are aware of the underlying network and its topology is pursued in WORMS[37].

However, the real challenge of such future attempt will be the development of the hardware and software system for the integration of information regarding the distributed processes, executing multiple tasks concurrently with replication, and controlling the system in real time. The challenge can be handled possibly by gaining better understanding about the overall problem from perspectives of system level control program which characterizes the behavior of the processes involved in specific applications, and takes in to account the interaction of processes in the given applications. In order to capture the above said requirements, we recognize the need to introduce a specialized coordinating module in to the classical layered protocol structure which acts as an intermediary between the application layer and the kernel. Such a module will collect the necessary information, triggers necessary actions in response to failures or changing requirements to achieve the best possible performance in the given system state.

1.3 Scope for Software Reconfiguration of Applications

The development of software reconfiguration algorithms address some of the state of the art issues in the field of distributed computing including some of the issues concerning the integration of the design diversity for fault tolerance and computational issues of correctness. In this research, the feasibility of some algorithms for improving the performance of software reconfiguration is pursued. This research attempts to use the knowledge gained from a good understanding of the applications as well as the nature of the system topology to incorporate the efficiency aspects of program execution whenever the support for reconfiguration exists. Research takes

up these issues for optimization to improve the functionality of the application at the system level.

Dynamic reconfiguration of applications requires large amounts of computational capabilities with stringent requirements on reliability. These capabilities have to be coupled with an ability to operate in response to the locally computed performance parameters as well as the information collected from remote sites at real-time in dynamic environments. Furthermore, these machines have to be able to adapt themselves in accomplishing specified tasks by managing their resources and maintaining their integrity in an optimal way based on the multi-objectives of computation, communication, and storage space.

The information available to a process that is monitoring the distributed program is different at different stages of execution and hence it has to make use of several supporting modules which will be initiated non-deterministically whenever needed. Current research provides some algorithms which may be used for such occasions for collecting topological parameters and for using such information for reconfiguring a process-in-execution that is affected by failures. Our approach is suitable for reconfiguration to meet the load balancing considerations also.

Implementation of a practical system based on the tools currently available, provides the guidelines for using the results of theoretical research and also provides the necessary focus for further research. Research from the view of implementation provides sufficient operational knowledge, if not the goal of optimum operation of theoretical research.

Our approach is to take the readily available tools for fault tolerant support and take up the study of several specific problems whose solutions would improve the performance of problems in wide areas of applications, especially those that are coarse grained and computationally intensive. We will first examine the practical

problem of reconfiguration of an application in execution in order to make use of idle capacity of hardware, and then describe the theoretical issues to be addressed. We will also provide a problem solving framework with special emphasis on the trade-off between the cost of communication at run time compared to the cost of reconfiguration and continued computation.

1.4 Contributions of the Dissertation

This research pursues both exploratory as well as implementation based research in fault tolerant distributed computing. Focus however is to extend some of the design issues of contemporary distributed application management tools for high performance software reconfiguration.

The field of Software Failure Reconfiguration in distributed systems and interconnection networks is still in its beginning, though many of the necessary tools to build such systems are becoming widely available. Insights gained during this research are related to dynamic re-allocation issues and about how to collect topological information of interest for the reconfiguration problem.

The first contribution of this dissertation is to analyze system level fault tolerant aspects of distributed applications and to recognize some of the issues related to software reconfiguration. Motivation for this work is the wide spread use of workstations and multi-computers connected to networks. It remains a challenge to use idle machines to provide fault tolerance for large grained computation-intensive applications. Using the results of checkpointing and recovery at process level and on the process migration support provided by many contemporary systems such as Demos/MP, V-System, etc., this research recognizes the possibility of introducing a control program residing on top of the Kernel and below the application layer so that it can determine system parameters needed to reconfigure an application when needed.

The second contribution is to provide transparent software reconfiguration strategies based on process migration in the context of embeddings in interconnection networks. Previous works have approached the problem of fault intolerant embeddings or have given solutions which accept degradation of performance because of the need for simulating faulty nodes or by degrading the performance parameters. This research is the first attempt to analyze the effect of the cost of migration in the process of software reconfiguration. Efficient algorithms for reconfiguring applications are presented for applications based on tree structures such as binary trees and quadtrees for special classes of faults. Further, the theoretical characterizations are provided to deal with arbitrary faults in distributed models using graph matching, and branch-and-bound, and quadratic assignment techniques

The third contribution is towards efficient fault diagnosis and fault tolerance in distributed systems. Our contributions include the distributed recognition of Articulation Points, Biconnected Components, and Bridges. Since the articulation point is a critical node in the network, it is a good candidate for selective replication. In this dissertation, the most efficient distributed algorithms for the recognition of all these properties of the networks are provided. The complexities obtained are $O(n)$ time, $O(n)$ number of messages, using message of utmost $O(n)$ length. Most efficient previously known algorithms have $O(e)$ messages, $O(d)$ time, and $O(logn)$ message length. Clearly, the new algorithms save $O(logn)$ message bits in the worst case, where n is the number of nodes in the graph corresponding to the underlying network, e is the number of messages, and d is the diameter.

The fourth contribution is an optimal distributed algorithm for recognition of graph planarity which is one of the most celebrated problems in graph theory and algorithm design. The distributed algorithm for Planarity Testing is based on the biconnected component solution given by Hopcroft and Tarjan[38]. The algorithm presented in this dissertation uses $O(n)$ time, O(n) message, and O(n + log n) message

size. This result for the celebrated problem on planarity testing is not only the first distributed algorithm, but is also optimal in both time and number of messages.

The fifth contribution of this study is an evaluation of the software redundancy techniques to provide fault tolerance to program structures. An approach of configuration management to reduce communication overhead if the failures are not frequent is taken in the context of embedded applications in interconnection networks. Hence for 'N-version programming', where results of different versions of components are used in each step, higher priority is given to keep all the program components near by while allocating the resources. On the other hand, for the technique based on recovery blocks which uses a back up component only in the case of failure, less priority is given to assign redundant components, thus keeping the primary components together. This research shows that such a problem may be reduced to a labeling problem and the solution is based on quadratic programming. A sub-optimal algorithm and simulation results based on heuristics are provided.

The main contribution of our work is to characterize the potential of a control program for software reconfiguration which can make use of the available tools for replication and process migration. Efficient algorithms are provided to achieve high performance in reconfiguration and to learn network topology.

1.5 Organization and Overview of the Dissertation

We review generic research problems related to distributed computing and fault tolerance in Chapter 2. Basic concepts of distributed fault tolerant computing are developed in chapter 2 in addition to summarizing some interesting previous results.

In chapter 3, transparent software reconfiguration strategies for embeddings based on tree structures in hypercube interconnection networks are identified based on process migration. Theoretical characterization uses graph matching, quadratic

assignment, and a variety of branch-and-bound techniques to deal with arbitrary faults in distributed models. Specifically, we have illustrated the reconfiguration of a complete quadtree into a hypercube whenever some of the processors used in the initial embedding are determined to be faulty. We provide a new 2-dilation embedding of a complete quadtree into its nearest hypercube and show that it is reconfigurable in case of homogeneous faults to yield an optimal 3-dilation reconfiguration. It is shown that a new class of reconfigurable embedding strategy may be used to recover from faults at run-time as well as maintain the quality of the mapping within a constant.

In chapter 4, we build on the well known principle that *a priori* knowledge of network topology leads to designing efficient solutions to many problems in distributed systems. We design new algorithms for finding important topological parameters such as biconnected components, articulation points, and bridges. Since such knowledge is very useful for selective replication, it may be used to achieve high performance reconfiguration. The most efficient algorithms for these problems known to date is presented in Chapter 4.

It is known in sequential computing that knowledge of planarity of graphs may be used for solving many problems efficiently which are computationally interactive in a general network. Distributed shortest path algorithms which are very basic to many distributed computations and data exchange, is shown to be efficient on planar graphs[39]. However, determining whether a graph is planar using a distributed algorithm is more complex because of the lack of global control over the system to monitor the algorithm behavior, compared to the sequential or parallel computing. In distributed systems, no global information is maintained at each node since the system is dynamic, and the global information might require prohibitive space to store the information. However, each node is equipped with algorithms that will enable these nodes to determine certain topological parameters, whenever necessary. These

algorithms are called *Learning Algorithms.* Building on the learning algorithms given in chapter 4, for finding connectivity and biconnected components, we extend distributed depth-first-search based algorithms for testing whether the global topology of an asynchronous network is planar or not in Chapter 5.

In Chapter 6, we present algorithms to study the performance issues related to the reliability techniques based on N-version programming and Recovery Block methods in the context of the embedding of program structures in interconnection networks. These software redundancy techniques are largely used in practice to provide both design fault tolerance as well as component failures. We take the approach of configuration management to reduce communication overhead if the failures are not frequent. Hence for 'N-version programming', we give higher priority to the program components while allocating the resources. On the other hand, we prefer to assign redundant components first for 'Recovery Block' technique.

Concluding remarks about the main contributions of our work and a discussion of open problems are presented in Chapter 7.

CHAPTER 2

PRELIMINARIES

To learn something, you should almost know it
-- Anonymous

2.1. Overview

Parallelism in many applications may be exploited to overcome the limitations of intensive computations. It is well known that many computational problems may be solved efficiently on interconnection networks and distributed systems. In this research, we are mainly concerned with the efficient and reliable distributed execution of application programs. Specifically, we investigate the problem of using several machines to co-operate on the solution of a single problem. Such an attempt requires us to gain insights about the nature of the given problem, understanding how machines co-operate and what are the factors that affect the efficiency of the solution. Existing facilities for execution of programs on remote machines have to be explored, evaluated, and new designs have to be implemented.

Novel approaches for these problems are currently the focus of much research interest. Attempts to build more powerful machines as well as to use the idle power of distributed systems are rigorously pursued. By a systematic comparison of different systems and applications, we can gain better understanding of the important issues involved. Distributed program execution have to make use of a large amount of computing power available on different machines. However, the potential for making use of a collection of machines and resources also offer significant challenges about sharing and protection, allocation and fault tolerance, synchronization and communication.

Many contemporary systems provide facilities for remote initiation, execution, and support for partial failures. Much is desired to identify idle machines that are suitable for robust execution to tolerate the failure of components and hardware crashes, and facilities for monitoring the performance.

2.2 Objectives of Robust Execution of Distributed Programs

Consider the possibilities of distributing an application program at various computing sites. Co-ordinated execution will have to address the issues related to allocation of tasks, interaction between processes, consistent access to non-local data, recovering and re-allocating tasks in case of failures, etc. These in turn have to be matched against the existing facilities in order to achieve optimal usage of resources and efficient solutions. Some of the issues that must be considered are described below:

2.2.1 Distribution

Many computationally intensive real world applications may be divided in to sub-tasks so that intelligent algorithms may be used to execute each sub-task in parallel on distributed machines and interconnection networks. Two most important considerations in selecting one of many available multi-computers are *processor utilization* and *communication cost.* The communication needs of the computations are modeled by a graph, called the *guest graph.* This graph depicts the required interaction between the data elements of the computation. The interconnection topology of the available system may be depicted by a graph, called the *host graph.* The guest graph may be mapped into the host graph for execution. The quality of a mapping is often determined based on two performance parameters of desired approximation to a measure of processor utilization and inter-processor communication cost.

Often, the mapping is guided by some constraints which may be different from application to application. The most commonly used constraint is the minimization of the communication overhead in the case of interconnection networks. Lee and Aggarwal [1] have given assignment optimization procedures for mapping schemes using various objective functions.

In the domain of distributed systems, however, most studies have left the problem of program distribution to the systems based on *allocator* services. If a process wants to be executed by a remote process, it requests for collecting the information about all the processors which have idle capacity to meet the requirements[2]. In some cases, group communication capabilities are used to send requests of availability information to all program servers on different machines and interested servers will reply to the requesting process[3]. In the later model, a program server is assumed to be present on each machine so that program execution may be started by sending a message to the remote machine.

2.2.2 Load balancing and resource sharing

The objective of distribution based research is to make use of the free processors and still co-schedule a group of processors on nearby machines to reduce communication. On the other hand, load balancing is also of primary concern so that some processors are not overloaded while while others are idle. It may be noted thus that the objectives of maximizing throughput, minimizing response time, and distributing a load uniformly are conflicting goals which need compromises and tradeoffs. This could be difficult since the the problem of finding an optimal solution to most of these concerns is NP-complete. Hence heuristic load balancing algorithms are used to communicate the load information in a distributed system periodically and to bid for creating or migrating the existing processes on to a lightly loaded machine[4,5,6].

2.2.3 Software fault tolerance and reconfiguration

The ability of a distributed system to continue to operate in spite of the failure of a subsystem, possibly with degraded performance, is very important. The primary goal of our research is to provide fault tolerance at the system level so that dynamic reconfiguration of the program in execution may be carried out in interconnection networks and distributed systems.

The approaches taken to provide fault tolerance in interconnection networks fall into the following three categories : *architecture-based, algorithm-based* and *reconfiguration-based.* The motivation to develop schemes based on the first approach is that the computational graph structure is fixed and hosts may be built in a such a way that, in case of faults, it can maintain the topology of the computational graph. Spare nodes and edges may be added to the basic host architecture. The resultant architecture will facilitate fault recovery by providing subgraphs isomorphic to the original graph [7]. In algorithm-based techniques, on the other hand, the underlying graph structure is made robust using redundant paths without altering the architecture of the host [8,9]. In reconfiguration schemes, there are two stages: fault detection and relocation of faulty nodes to free nodes. These schemes provide intelligent mapping of the computational graph into the interconnection network so that reconfiguration becomes easy. There is a growing need for designing reconfiguration algorithms for commonly used interconnection networks such as hypercubes. Chen, et.al. [10], have provided efficient reconfiguration algorithms for grids into hypercubes. This algorithm enables the system to recover from a single failure in one-step by locating a free node in the hypercube. Time taken by the system to reconfigure itself after the detection of faults is the primary concern of these algorithms. The previous techniques do not address the problem of maintaining the quality of initial mapping after reconfiguration. We investigate methodologies to develop reconfigurable mapping strategies that maintain certain important qualities of the initial mapping.

The main goal of research in software fault tolerance in distributed systems is to identify software replaceable units of failure analogous to hardware replaceable units which enable reconfiguration in case of failures or upgrades, without disrupting the activities of programs in execution[11]. Specific objectives of such research may be stated as follows:

1 Develop system level capabilities to mask hardware failures so that application program can continue to run without any interruption.

2 Study efficient methods to tolerate failures at the operating system level using redundant program components which run on distinct hardware processor hosts and maintain redundant information about the state of the program in execution. In case of failures of a program component, the information in the surviving processors should be sufficient to recover from the fault and continue program execution.

Reconfiguration in distributed systems is a very difficult problem because of the dynamic nature of the underlying network. The details of the related works may be found in [2,12]. Local and global properties in networks of processors are formally treated in [13]. Very little work is pursued, in the related areas of topology-update problem in dynamic networks and may be found in [14,15,16,17]. In a distributed system based on client-server model, servers that provide a single logical service may be replicated on different nodes to make the system robust and by providing token based protocols for maintaining consistency among replicated servers[18].

2.3 General Techniques for Software Fault Tolerance

Several fault-tolerant mechanisms have been proposed in research and many commercial systems have been built in software. Since many of these systems overlap the basic models of fault tolerance, we classify them into five categories: State machine approach and replication, Nested atomic actions, Rollback and

recovery, Functional programming, and Application specific tool development. These generic models of software fault tolerance provide design guide lines to cope with the operational failures in hardware or in supporting software. Most of these models use fail-stop processor approach for designing fault tolerant systems in which a processor is assumed to identify all failures and further that failures do not cause any state transformations[19]. The fail-stop processor model is appealing since it may be approximated by real hardware. Representative approaches in each of these categories are described in the following sections.

2.3.1 State machine approach and replication

The seminal work of Lamport proposes a model of distributed programs where in, a program is viewed as a state machine which receives input, carries out appropriate actions and maintains internal state, and generates output[20,21]. A reliable system may be constructed based on this model using replicated state machines that execute in parallel. Such a model is proven to receive consistent input despite failures even under Byzantine Generals Protocol[19].

a) N-version Programming

This software approach is based on the hardware mechanism for fault tolerance called N-modular redundancy, in which a given component is replicated for a given number of times, and a special voting circuit determines the correct answer based on the voting of majority of the answers supplied by redundant components. A special case is triple-modular redundancy which has the replication factor of three. This approach is extended for software systems so that software generated solutions are chosen based on a voting scheme in software. This technique, called N-version programming, is used extensively in systems including Software Implemented Fault Tolerance, SIFT[22]. This scheme uses replicated execution of software modules with intermediate synchronization points where the consistency of computations are

voted upon using software techniques. This method does not require the Fail-stop model, and can be used for software design faults as well.

b) Recovery Block Technique

This software technique uses different implementations of each task, where the first implementation of a task is called primary module and other implementations are called alternate modules. Each module uses an acceptance test to check if the results of computations are acceptable. If primary block succeeds in this test, it continues to the next task. If not, failure has occurred and thus alternate module is executed in turn till the acceptance tests succeed. This method makes use of the knowledge of the domain of computation and further details may be found in [23].

c) Active Backup Process

This method uses multiple concurrent execution of each process similar to N-version programming, but instead of voting, the first available solution is chosen as the correct answer. Large-grain object support in CHORUS[24] uses this method along with time-out for detecting invocation failures. Fault tolerant distributed file service based on replicated processes is supported using a similar approach in[25], using one-to-many and many-to-one communication between the processes for synchronization and communication of results.

2.3.2 Nested atomic actions

This approach ensures data resiliency in a quiescent environment by nested atomic actions which provide *indivisibility* and *recoverability* [26]. Indivisibility refers to non-overlapping execution of an activity. Recoverability refers to a guarantee that either all objects involved in an action remain in the initial state if the action is committed or all of them reach final state if the actions are committed. Nested actions may be implemented using the two-phase commit protocol derived from the

atomic transaction concepts in database systems where in the write actions need locks from remaining objects, resolving conflicts using a total order on writing processes. This general mechanism is extended for general purpose computations instead of just read and write actions as in atomic transactions. Important variations supports provided are nested atomic transactions in Guardians and Actions as in ARGUS[27], Communicating Sequential Processes with and without voting[28,29], and TABS/ CAMELOT[30] support for nested transaction scheme.

2.3.3 Rollback and recovery

In the fail-stop model, when an execution of a program is stopped because of a fault, if the internal state of the failed process is available, the program may be restarted on another working processor. Some facilities for rolling back the program to the state that was in progress when a failure occurred. Routine that is needed to complete the state transformation, called a recovery protocol, is also assumed to be available. Since it is necessary to rollback to any intermediate state after a failure, it is necessary to store the state informations as checkpoints in stable storage [19].

a) Checkpointing

A checkpoint is the static record of the state of a process at some instance of time. In the context of concurrent processes communicating with each other, state of the process change after every communication. Thus, fault tolerance may be provided by creating checkpoints and storing an inactive process on a separate processor, and may be activated in cases of failure of the primary process. To avoid the domino effect needing to restart from the beginning, check points must record all communication with other processes[3].

b) Message Logging

The effort to keep the state of checkpoints current is mitigated by reducing the frequency of checkpointing and logging all messages between processes. Recorded messages are used to update the last checkpoint. The idea of message logging is used in many modern systems including ISIS[31,32], in which remote procedure call invocations are kept in passive processes called cohorts.

c) Optimistic recovery

This technique allows asynchronous message logging together with its own check pointing so that normal execution of a process does not have to synchronize with any other, but in cases of failure the processes have to be rolled back using checkpoints and logged messages. NIL project uses this technique[33].

d) Sender based message logging

This approach achieves low overhead for fault tolerance using the message log in the senders process. In case of failure, the messages since the failures will be re-sent to the process after restoring the internal state using checkpoints. Frequency of checkpointing may be tuned to achieve the time time taken for recovery[3].

2.3.4 Functional programming

All computations in functional programming are evaluated such that no side effects are caused except the resulting return values. Hence this approach may be used in fault tolerant distributed programming by re-invoking a failed process with the same parameters as before. Since there is no state information that might have been lost, this is an effective method. This method is not general since the functions in many applications do yield a neat separation requiring access to outside data which are not explicitly passed. The master-slave model designed for compute-intensive applications use purely functional robust sub-task execution[34].

2.3.5 Application specific tool development

In order to achieve efficiency, these approaches do not try to provide general purpose mechanisms for transparent use in any programs. Instead, knowledge about the applications may be exploited to express the types of errors expected, and specific corrective actions may be pre-programmed. The facilities for recovering from faults and control are given to the application process. Such a power to the process, though it removes transparency, enables the application program to decide the most appropriate processor for program execution at run time. Exception handling in ADA as well as process based systems in Fault Tolerant Concurrent C[35], use this approach to provide selective redundancy in order to tolerate faults.

a) Process Migration

Transferring sufficient amount of a process's state from one machine to other for execution of a process is called process migration. This technique along with checkpointing is used for fault tolerance in DEMOS/MP[36] and in LOCUS[37]. Preemptable remote execution facilities are provided in V-System[34]. An excellent survey of process migration support mechanisms and approaches may be found in [38].

This approach provides programming facilities so that application programmer or the system on its own, may provide selective fault tolerance. This approach needs more effort on the part of the programmer to specify which part of application is critical. However, the benefit is greater efficiency. This approach is taken to design selective replication in Fault Tolerant Concurrent C[35], applications management in ISIS[31]. A very interesting variation of the application directed approach where in the processes are aware of the underlying network and its topology is pursued in WORMS[39].

Various approaches for software fault tolerance may use a mixture of these basic models to obtain design diversity. In this research we focus on high performance algorithm development for application specific tools approach.

2.4. Concepts and Techniques for Distributed Programming

Distributed systems are characterized by message passing models and it is assumed shared memory is absent. Hence, concurrent execution in the context of distributed systems introduces a set of issues involving synchronization and communication mechanisms. Algorithms on distributed systems use the concept of *message* and hence the primitives are defined for sending and receiving messages. All distributed systems achieve synchronization and communication for transferring data as well as control using messages only.

Different nodes of a distributed system cooperate by exchanging local control information. It is assumed that no single node has complete control information. The order of computation and hence operational results become unpredictable during the execution of distributed algorithms due to unpredictable delays in communication. Event control and execution at different sites take place in a distributed fashion. Processing time is considered negligible while designing distributed algorithms since the communication costs are far higher, and hence the number of messages required to execute an algorithm is a measure of the efficiency of a distributed algorithm.

a) Diffusing Computation Model:

The distributed algorithms make use of techniques such as using messages for acknowledgement, broadcasting results and local data to some selected neighbor processors, etc., irrespective of the model being employed to solve a problem. Dijkstra and Scholten proposed this technique mainly for termination detection of a

distributed process or mutual blocking of two or more processes. [40]. This principle may be used to construct a spanning tree of the graph representing the network. The *root* of the tree plays a different role compared to other nodes in the network since it has to send the information to al other nodes and receives replies from others in the following sequence:

i. An outside initiator process sends a message to the root to start the spanning tree of computation

ii. The root sends a message to its neighbors

iii. On receiving the message, each node propagates the message to its unvisited neighbors and waits for a reply. Upon getting the reply, it sends a reply to its parent. Note that if a node is a leaf, then it does not have any unvisited neighbors and thus it sends the reply to its parent immediately.

iv. the root terminates the algorithm by sending a message to the initiator when it receives replies from all its neighbors.

b) Algorithmic Complexity

A small number of messages lead to less traffic in a distributed network, and hence the overall delay may be reduced. In a distributed algorithm, it is desirable to use as few messages as possible to achieve the desired goal at the end of the computation. Efficiency of a distributed algorithms is measured by the total number of messages exchanged during the execution of the algorithm.

c) Computing global states:

The absence of a system-wide clock inhibits any node having the global knowledge of a distributed system. However, there are occasions when an algorithm has to be devised to compute the global state of a distributed system. Technique for computing global states has many applications in distributed computing to detect the

invariant properties in the system, depending upon the application. Since the system states keep changing, it is only possible to take *snapshot* of the system and compute the invariants such as, "the total number of tokens in the system is n" or "the system is deadlocked" is an invariant property[41].

d. Agreement and Commitment:

The problem of designing a protocol by which all processors agree upon a value is called establishing agreement in a distributed system. This value may be a data, or the results of a computation. The agreement problem is fairly simple when the processors do not fail. However, in presence of faulty processors, there is ambiguity about the *integrity* of the data that is sent from it. In such an environment, the agreement problem may be solved using protocols in which a known value at a designated processor, called the *transmitter* is sent to other processors so that:

1 At the termination, all non-faulty processors agree on some value.

2 If the transmitter a is non-faulty, then all non-faulty processors use its value as the one they agree upon.

There are a number of variants of the problem, depending on whether processor failures are detectable, whether processor speeds are bounded, and whether messages can be authenticated. Broadcast protocols describe how a message being broadcast is disseminated to the non-faulty processors in the network. Byzantine agreement deals with the agreement problem when no assertion can be made about the behavior of faulty processors. Several protocols have been devised and reported in literature on these topics [19].

In a distributed system, it is necessary to coordinate a set of actions at different processors so that the whole set of actions occur or none do. For example, when a database is replicated at different processors, it is necessary to see that all copies of the database remain consistent. If one copy is changed, it must be reflected in all or

none. A *commit protocol* is an agreement protocol in which the value agreed upon is computed by applying a *decision function* to the values corresponding to the participanting nodes[27].

e) Elections

Election technique is used by a set of processes to choose one outcome from many possible candidates, any of which is acceptable. The decisions could be to choose a coordinator, referred to as 'break symmetry', in the system, or to resolve a conflict of two or more contending processes regarding succeeding actions being attempted.

Elections are very important in distributed systems. Depending upon several topological configurations for the system, the assumptions regarding the reliabilities of the nodes as well as links, there are a variety of variations of models for elections, such as weighted voting, random selection, etc. An extensive treatment on election algorithms could be found in [42].

CHAPTER 3

SOFTWARE RECONFIGURATION

"Would you tell me, which way I ought to go from here" Alice said
"That depends a good deal on where you want to get to", said the Cat
"I don't know where ...", said Alice
"Then it does not matter which way you go", said the Cat.

-- From Lewis Carrol's Alice's Adventures in Wonderland

3.1 Overview

In recent years we have witnessed a tremendous surge in the availability of very fast and inexpensive hardware. These are made possible partly by novel architectural features such as pipelining, vector processing etc., and partly by using novel interconnections between processors and memories such as Hypercubes, Orthogonal Tree Networks and others. As the complexity of these networks increased, reliability and thus the ability to reconfigure the system in case of faults became more important. Further, increased computational power has also made it possible to attempt those problems which were impractical to solve until recently. Recovery of such computationally intensive applications from hardware failures and their ability to continue computation without a need to start all over has become feasible due to the recent advances in operating system support for check pointing and process migration. However, such a given problem can be solved more efficiently if an intelligent mapping of the structure of computation into the network of processors may be obtained before and after fault recovery.

The approaches taken to provide fault tolerance in interconnection networks fall into the following three categories : *architecture-based, algorithm-based* and *reconfiguration-based.* The motivation to develop schemes based on the first approach is

that the computational graph structure is fixed and hosts may be built in a such a way that, in case of faults, it can maintain the topology of the computational graph. Spare nodes and edges may be added to the basic host architecture. The resultant architecture will facilitate fault recovery by providing subgraphs isomorphic to the original graph [1 In algorithm-based techniques, on the other hand, the underlying graph structure is made robust using redundant paths without altering the architecture of the host[2,3]. In reconfiguration schemes, there are two stages: fault detection and relocation of faulty nodes to free nodes. These schemes provide intelligent mapping of the computational graph into the interconnection network so that reconfiguration becomes easy. There is a growing need for designing reconfiguration algorithms for commonly used interconnection networks such as hypercubes. Chen, et.al., [4] have provided efficient reconfiguration algorithms for grids into hypercubes which enables the system to recover from a single failure in one-step by locating a free node in the hypercube. Recently, a distributed algorithm for embedding rings into faulty hypercube using only local knowledge of the faults is presented [5]. Time taken by the system to reconfigure itself after the detection of faults is the primary concern of these algorithms. Apart from this, reconfiguration in faulty hypercubes has been approached by simulating the faulty nodes by nodes within constant distance [6]. Reconfiguration parameters in these algorithms depend on the probability of failure since more node failure loads the simulating nodes more.

Parallel architectures based on binary hypercube topology have gained widespread acceptance in parallel computing [7]. It is known that embedding trees into hypercubes is NP-complete. [8]. Embedding trees have been studied by many researchers [9,10,11,12,13,14,15]. Some researchers have approached this problem as graph isomorphism [16] , where as some others have made use of quadratic assignment [17]. Throughout our discussion, the terms *mapping* and *embedding* mean the same, and are used interchangeably. Often, the mapping is guided by

application dependent constraints such as minimization of the communication overhead. Lee and Aggarwal [18] have given assignment optimization procedures for mapping schemes using various objective functions.

Quadtrees have received a lot of attention in recent years as an efficient data structure for a variety of image processing applications. In the quadtree representation of an image, the root represents an entire square field, and each node has either four children, each representing a quadrant of its parent, or is a leaf representing a quadrant. Klinger and Dyer [19] provide a good bibliography of the history of quadtrees. The reader is referred to the comprehensive paper on quadtrees by Samet [20]. Jones and Iyengar [21] have studied efficient ways of storing quadtrees. The issues in the related works are the representations of different features of images, space efficient quadtrees [22,21,23] , and developing time efficient algorithms to perform some common operations on images [24,25,26]. Realizing the importance of quadtrees, we use it as an underlying data structure to illustrate our technique of reconfigurable embedding. The method described is not restricted to one set of data structures but can be extended to many others also.

The previous techniques do not address the problem of maintaining the quality of initial mapping after reconfiguration since they allow degraded performance. In this chapter, we investigate methodologies to develop reconfigurable mapping strategies that maintain certain important qualities of the initial mapping. As an example, we provide an embedding of a complete quadtree into a hypercube which can be reconfigured for any instance of homogeneous faults. The characteristic feature of our technique is that the dilation in the reconfigured system is a constant. Further, performance is not degraded at run-time and the overhead of reconfiguration is only a one-time cost.

The remainder of the chapter is organized as follows. In section 3.2 we describe the problem statement and discuss the issues related to reconfiguration. We introduce preliminaries and basic notations used in our technique in section 3.3. We provide a new embedding of quadtree onto hypercubes in section 3.4. and provide the characterization of free node distribution. We provide a process of specific reconfiguration in section 3.5. Reconfiguring faults in complete quadtree embedding into hypercube in the case of general faults is discussed in section 3.6. We summarize the results and discuss the scope for future research in this area in section 3.7

3.2 Problem Statement

It is well known that many computational problems may be solved efficiently on interconnection networks by providing intelligent mapping of the structure of the computation onto the network of processors. The software reliability of application programs running on these networks is of great importance in faulty environments. Hence it is necessary to reconfigure the embedding in case of faults, so that the computation may be continued. The purpose of reconfiguration is two fold: (a) keep the network operational, and (b) maintain the same quality. Even though many researchers have addressed the operational aspect of reconfiguration, all of these approaches result in degraded performance at run-time either due to increased dilation or due to overloading of some nodes.

It is important to characterize the quality aspect of reconfiguration. We investigate the existence of reconfiguration algorithms that satisfy both of the above demands using process migration and maintaining the dilation of the initial embedding after reconfiguration. Note that an arbitrary assignment of a process at a faulty node to a free node in a hypercube, may result in a maximum dilation of $O(\log n)$. A reconfigurable embedding technique should maintain a constant dilation while tolerating faults up to the total number of *free nodes*.

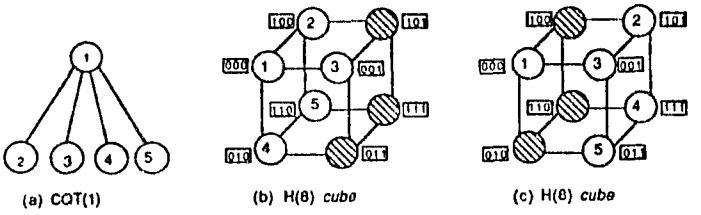

Figure 3.1 Embedding $CQT(1)$ into $H(8)$

Consider an embedding of a complete quad tree into a nearest hypercube. In case of faults in the leaves, the smallest constant of dilation that is obtained by assigning the processes at the faulty nodes to the *free nodes* is 3. This can be verified by considering the smallest complete quadtree (of height 1) and its nearest hypercube (of dimension 3), a *cube*. A $CQT(1)$ is shown in figure 3.1(a). A 2-dilation embedding of $CQT(1)$ into $H(8)$ is given in figure 3.1(b). Note that one of the *free nodes* is at a distance of 3 from the node to which the root of the tree is assigned. Figure 3.1(c) depicts an alternate embedding with dilation 3. This motivates us to investigate the existence of reconfigurable embedding algorithms that maintains a dilation of 3 in an optimal expansion hypercube for a complete quadtree of arbitrarily large height.

3.3 Notations and Preliminaries

We introduce notations and definitions used to describe the quadtree, the hypercube, and their associated terms. We use the terms *nodes/processors* and *links* for the hypercube and *vertices/processes* and *edges* for the quadtree.

3.3.1 Terminologies for a complete quadtree

We denote a Complete Quadtree of height h, as $CQT(h)$. The root of $CQT(h)$ is at level 0, and the leaves are at level h. The number of vertices at level i is 4^i. We now introduce some terminologies pertaining to $CQT(h)$. The number of leaves, l is given by, $l = 4^h$. The total number of vertices at level $h-1$ (i.e., the parents of leaves) denoted by, p, is given as: $p = 4^{h-1}$. The total number of vertices in $CQT(h)$ denoted by, m, is given as: $m = \frac{4^{h+1} - 1}{3}$.

We perform a breadth-first-search on *CQT(h)* and assign to each vertex, its BFS-Number, called *BFSN*. For example, the *BFSN* of root is 1, its children are 2,3,4 and 5, and so on. We refer to every vertex by its *BFSN*.

3.3.2 Terminologies for a binary hypercube

We denote a hypercube with n nodes, as $H(n)$. We represent every node in $H(n)$ by a unique binary number (k-bit address) $b_k, b_{k-1} \cdots, b_1$, where k is the dimension of $H(n)$. Note that a node is adjacent to k nodes in $H(n)$. Also two nodes in $H(n)$ are adjacent when their addresses differ in exactly one of k bits.

$H(8)$ is called a *cube*. The addresses of the eight nodes in a *cube* have identical bits in their $k-3$ most significant bit(m.s.b.) position. We call the binary number formed by taking these $k-3$ bits as the *address* of the *cube*. All *cubes* that have even number of identical m.s.b.'s are said to be in a *quadrant*, and these m.s.b.'s are referred to as *quadrant bits*. Further, every *quadrant* is recursively divided into four *sub-quadrants* by making use of the permutations of higher 2 least significant bits (l.s.b.'s) following the quadrant bits until no such division is possible. We now define a *similar* cube and a *square*.

* Two *cubes* are said to be *similar*, if the bits in their addresses are identical, except two successive bits starting at the same odd bit position. Note that these two differing bits will identify a *quadrant* or a *sub-quadrant*. For example, let the addresses of *cube1* and *cube2* be 01 01 00 00 01 and 01 11 00 00 01. These two cubes are *similar* and both of them belong to quadrant **01**.

* For any cube in a *quadrant* or *sub-quadrant*, there are three more *similar cubes*. In each of these *cubes*, all the eight nodes are identified using three l.s.b.'s. The nodes having three identical l.s.b.s in these four *cubes* are said be *similar nodes*. A *square* is formed among *similar nodes*.

Following are the conventions followed in the rest of chapter regarding the figures involving hypercubes. The addresses of the nodes in a *cube* are given in rectangular boxes. The address of the cube itself is denoted as the quadrant bits. For example, $H(32)$ is divided into four quadrants each having a *cube*. The quadrant numbers are **00, 01, 11, 10** and are marked in their appropriate quadrants. For the sake of clarity, the interconnections of only certain nodes are shown. The circles represent the nodes (processors) of a hypercube. The actual address of any node in the diagram is obtained by concatenating the quadrant bits as the prefix to the address shown in the rectangular box. Again, the circles with numbers in them (numbers correspond to a vertex in the quadtree) are used in the embedding process, while the shaded circles denote free nodes or faulty nodes.

We define the number of faults at an internal node,u as the sum of the faults in the subtree rooted at u. We refer to the sequence of the number of faults at the nodes in the order of increasing BFSN at level $h-1$ as the fault sequence,*FTS*, while that of free nodes as the free node sequence, *FNS*. Further, we refer to the sequence of the number of faults at the nodes at a level of the tree as the *faults at the level of the tree*.

3.3.3 Embedding terminologies

The embedding function f maps each node in the guest graph $G = (V_G , E_G)$ into a *unique* node in the host graph $H = (V_H , E_H)$. V_G and V_H denote the node sets of the guest graph and the host graph respectively, and E_G and E_H the edge sets. Let v_1 and v_2 be nodes in G. Since the embedding is a 1-1 mapping of V_G onto V_H, let $f(v_1)$ and $f(v_2)$ be the nodes in H that are images of v_1 and v_2 respectively. In the following we define distance, dilation, and expansion.

* Let the distance, $d(v_i , v_j)$ between v_i and v_j be defined as the number of edges in the shortest path connecting v_i and v_j in either G or H.

* The dilation d_f is defined as

$$d_f = \max \frac{d(f(v_1), f(v_2))}{d(v_1, v_2)}$$

* The expansion e_f is defined as

$$e_f = \frac{|V_H|}{|V_G|}$$

Our embedding algorithm builds an array, $B[1..m]$, where $B[i]$ contains the address of a unique node in $H(n)$ for the vertex of $CQT(h)$ whose *BFSN* is i. We now state a simple lemma that is useful for the following discussions.

Lemma 3.1: The height of $CQT(h)$ and the dimension, k of $H(n)$ are related as: $k = 2h + 1$.

Proof: We prove by induction on h.

Basis: The claim is true for $h = 1$. We need a *cube* ($k = 3$) to embed $CQT(1)$.

Hypothesis: Let the claim be true for $CQT(h-1)$. Assume that we need a hypercube of dimension k'. Then we have $k' = 2(h-1) + 1$....(i)

Step: Now consider embedding $CQT(h)$ into a hypercube of dimension k. $CQT(h)$ is obtained by taking 4 copies of $CQT(h-1)$ and having a new root that is connected to all the 4 roots of $CQT(h-1)$'s. Hence the number of nodes in the hypercube to embed $CQT(h)$ is 4 times that is used to embed $CQT(h-1)$. That is, $k = k' + 2$....(ii)

From (i) and (ii) we get $k = 2h + 1$, and hence the proof.□

We now state a corollary which follows from lemma 3.1.

Corollary 3.1: The number of *cubes*, p, is equal to *one-fourth* of l (the number of leaves).

3.3.4 Fault model

In this section we state the model used for fault tolerance. We explain the assumptions regarding faults, during the execution of our reconfiguration algorithms.

1 We assume that expansion of the embedding of G into H is *optimal*, i.e., expansion < 2, and the number of nodes in H is greater than the number of vertices in G. This implies that some nodes in H are *unused* and we refer to those nodes as *free nodes*, FN. Hence the number of free nodes is greater than 0, and is given by $FN = |H| - |G|$.

2 Maximum number of faults tolerated is equal to the number of available *free nodes* after the initial embedding. The set of faulty nodes in H is denoted as FT. Hence it implies that $|FT| \leq |FN|$.

3 Reconfiguration is the assignment of the processes at the faulty nodes to unique free nodes. Hence the process migration is implied from a faulty node to a newly assigned free node.

4 The host graph, H remains *connected* after the faults, i.e., H - FT is connected. This implies that there exists at least one *fault-free path* from every node in H to any other non-faulty node.

5 We assume that reliable fault diagnosis mechanisms are available.

6 Reconfiguration is carried out after the faults are detected.

7 Any node that detects a fault may initiate the reconfiguration algorithm.

8 We also assume that the links are fault-free.

In order to keep discussions simple in the rest of the chapter, with respect to the assumption 3 regarding reconfiguration, we refer to the assignment of a process at any faulty node to a free node as simply assigning faulty node to a free node. Thus, the assignment of a faulty node implies the assignment of the process at the faulty node.

3.4 A New Embedding of Quadtree into Hypercube

We give an algorithm to obtain a dilation 2 embedding which has the unique characteristics of free node distribution that will help reconfiguring faults.

The embedding algorithm works in a top-down fashion on the quadtree. In the beginning, the root vertex is assigned to a node in the hypercube and further, its children are assigned to unique nodes such that dilation requirement is satisfied. In order to provide a unique assignment, the bit address which is used for assignment once is never used again for the embedding. After embedding successfully all the internal vertices at any level of the $CQT(h)$, the vertices in the next level are considered for assignment in their *BFSN* order. Bit address for any internal node is assigned by considering their level in the quad tree and the index in the level. The main idea of the algorithm is to assign at each step a unique address such that the assignments for all the nodes of the quad tree maintain a 2-dilation.

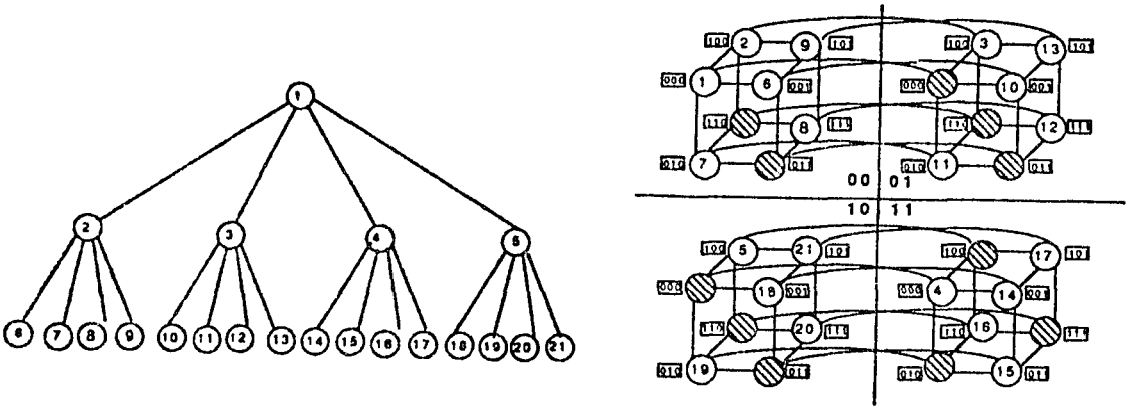

Figure 3.2 Illustration of 2-dilation Initial Embedding

For illustration purpose, consider the embedding of $CQT(2)$ into $H(32)$ given in figure 4.4.1.1. As a first step, the root (i.e. the vertex 1) is assigned to the node in the quadrant, **00** at address 00000. Note that the root of the quadtree is at level 0. As we explained in the previous section, the vertex (2) is placed in the quadrant **00**, where

its parent is placed at address 00001, while (3),(4) and (5) are placed at quadrants **11,10** and **01** at addresses 11000,10000 and 01000 respectively. Finally, the children of these subroots are assigned to unique nodes in the same quadrant of their respective parents. It may be easily verified that the fault sequence given in figure 4.4.0 may be reconfigured with dilation 2 in the embedding we have provided above.

Theorem 3.1: A complete quadtree of height h, $CQT(h)$ may be embedded into its nearest hypercube, $H(n)$ with at most dilation 2.

Proof: Our embedding function, f_h, builds an array, $B[1..m]$, where $B[i]$ contains the address of a unique node in $H(n)$ for the vertex of $CQT(h)$ whose *BFSN* is i, and m refers to the total number of vertices in $CQT(h)$. Formally, f_h is defined as follows:

$f_h : i \rightarrow B[i]$

where

$B[i] = b_k b_{k-1} \cdots b_1$

h is the height of $CQT(h)$;

i is the BFSN of any vertex in $CQT(h)$;

$k = log(n)$ is the dimension of $H(n)$.

We obtain f_h by a recursive construction on h and prove the theorem by induction on the height of $CQT(h)$.

Basis: The basis of induction is given for $h = 1$. The nearest hypercube needed to embed $CQT(1)$ is $H(8)$ as given by lemma 2.3.1. The binary representation of any node in $H(8)$ will have 3-bit address, say b_3, b_2, b_1. We give a 2-dilation constructive embedding of $CQT(1)$ into $H(8)$ as follows:

Let the root of $CQT(1)$ be assigned to the node with address 000 in $H(8)$, i.e, $b_3 = b_2 = b_1 = 0$. The four children of the root of $CQT(1)$ have to be assigned to nodes with unique addresses which is different from that of the root. This is achieved by using the permutations of the bits b_3 and b_2 and assigning a 1 for b_1 when both b_3 and

b_2 are complements of the corresponding bits of the root, while assigning a 0 for b_1 otherwise. Note that such an assignment is unique and maintains a dilation of 2. More specifically, the four permutations of the bits b_3 and b_2 are: 00, 01, 11, 10. The permutation in which both the bits are complements of that of root is 11. Hence the b_1 will be a 0 for the child with $b_3\, b_2 = 11$, and the b_1 will be a 1 for all other children. For $h = 1$ the actual embedding of $CQT(1)$ into $H(8)$ is given in figure 4.4.1.2

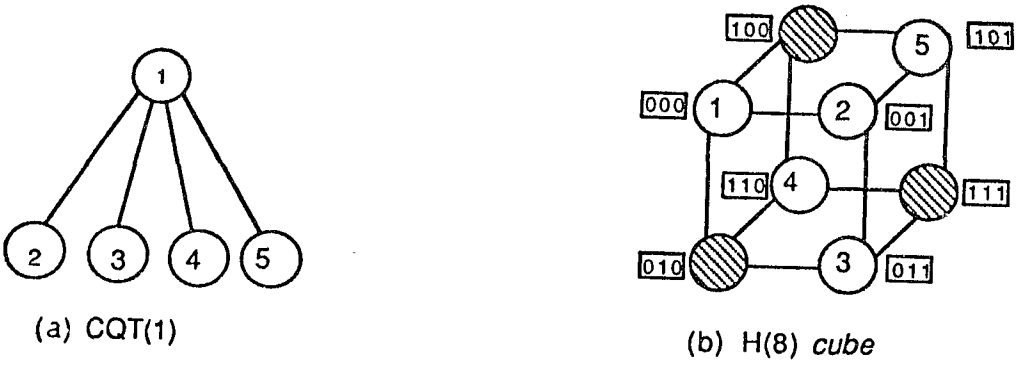

Figure 3.3 Embedding of CQT(1)

The array, B obtained using the function $f_h : i \rightarrow B[i]$ is given below for $h = 1$:

$B[1] = 000;\ B[2] = 001;\ B[3] = 011;\ B[4] = 110;\ B[5] = 101;$

The Theorem 3.1 clearly holds good for our basis.

Hypothesis: For any height $h' < h$, a $CQT(h')$ can be embedded into a hypercube $H(n')$ with dilation 2, where n' is the number of nodes in the nearest hypercube needed to embed $CQT(h')$. Note that the number of bits needed to address any node in $H(n')$ is $k' = 2h' + 1$ from lemma 2.3.1 and $k' = log(n')$.

Induction Step: We shall prove that $CQT(h)$ can be embedded into $H(n)$ with dilation 2 by obtaining f_h defined before. Assume that there exists an embedding function $f_{h'}$ which satisfies the induction hypothesis. Let $h' = h - 1$. We make the following observations needed to prove the theorem.

Observation 3.4.1: The $CQT(h)$ can be obtained from $CQT(h-1)$ by adding four children to each of the leaves in $CQT(h-1)$. In other words, all the newly added nodes become the leaves of $CQT(h)$.

Observation 3.4.2: From Lemma 3.1, the height of a $CQT(h)$, and the dimension, k of the hypercube $H(n)$ are related as $h = 2k + 1$. Since $k = \log(n)$ refers to the number of bits in the binary representation of the nodes of the hypercube, $k = k' + 2$ where k' is the dimension of the nearest hypercube needed to embed $CQT(h-1)$.

We provide the embedding function, f_h for the vertices of $CQT(h)$ in two steps as follows:

Step 1: Embed internal vertices of $CQT(h)$

Let v_i be an internal vertex in $CQT(h)$. Let $v_{i'}$ be the vertex in $CQT(h-1)$ with the same BFSN as that of v_i. Let X' be the binary representation of the node assigned for $v_{i'}$ using $f_{h'}$. We obtain the binary representation, X of v_i by concatenating X' with two 0's, i.e. $X = X'\,00$. Formally, f_h for all the internal vertices of $CQT(h)$ may be given as:

$f_h : i \rightarrow B[i]$

where

$B[i] = CONCAT(B'[i]\ , 00)$;

CONCAT is a concatenation function;

$B'[i]$ is obtained using f_{h-1} as $f_{h-1} : i \rightarrow B'[i]$.

Claim 3.4.1: Step 1 embeds all the internal vertices of $CQT(h)$ into unique nodes in $H(n)$ and maintains a dilation of 2.

Proof: Since the array B generated by f_h has a 1-1 correspondence with the addresses of the nodes in $H(n)$, it is sufficient to prove that the elements in B are distinct. From the hypothesis it follows that the elements in B' are distinct and the embedding has a dilation of 2. Since $B[i]$ is obtained by concatenating $B'[i]$ with 00, every element in B is distinct and maintains the dilation 2. Hence the claim.□

Step 2: Embed leaves of CQT(h)

Consider a set of four leaves in $CQT(h)$ which have a common parent. Let p be the BFSN of the parent, and let $CH[i], \forall i = 1,4$ refer to these four leaves ordered by their BFSN's. We obtain $B'[p]$ using f_{h-1}. We obtain the node assignments for the children depending on whether the l.s.b of $B'[p]$ is a 0 or a 1.

Case I: l.s.b of $B'[p]$ is a 0

The f_h for these four leaves may be given as:

$f_h : BFSN_i \rightarrow B[BFSN_i], \forall\ i = 1,4$

where

$BFSN_i$ is the BFSN of the $CH[i]$;

$B[BFSN_1] = CONCAT(EXPLOD(B'[p], 1), 001)$;

$B[BFSN_2] = CONCAT(EXPLOD(B'[p], 1), 011)$;

$B[BFSN_3] = CONCAT(EXPLOD(B'[p], 1), 110)$;

$B[BFSN_4] = CONCAT(EXPLOD(B'[p], 1), 101)$;

CONCAT is a concatenation function;

EXPLOD(S,i) is a function which takes two arguments, an array, S, and an index, $i < |S|$, and returns a new array of size $|S| - i$ in which all the i l.s.b's of S are removed.

Case II: l.s.b of $B'[p]$ is a 1

The f_h for these four leaves may be given as:

$f_h : BFSN_i \rightarrow B[BFSN_i], \forall\ i = 1,4$

where

$BFSN_i$ is the BFSN of the $CH[i]$;

$B[BFSN_1] = CONCAT(EXPLOD(B'[p], 1), 001)$;

$B[BFSN_2] = CONCAT(EXPLOD(B'[p], 1), 010)$;

$B[BFSN_3] = CONCAT(EXPLOD(B'[p], 1), 111)$;

$B[BFSN_4] = CONCAT(EXPLOD(B'[p], 1), 101)$;

CONCAT is a concatenation function;

EXPLOD(S,i) is a function which takes two arguments, an array, S, and an index, $i < |S|$, and returns a new array of size $|S| - i$ in which all the i l.s.b's of S are removed.

The procedure given for a set of four leaves may be repeated for every other set of four leaves.

Claim 3.4.2: Step 2 embeds all the leaves of $CQT(h)$ into unique nodes in $H(n)$ and maintains a dilation of 2.

Proof: Since the array B generated by f_h in step 2 has a 1-1 correspondence with the addresses of the nodes in $H(n)$, it is sufficient to prove that the elements in B are distinct. From the hypothesis it follows that the elements in B' are distinct and the embedding has a dilation of 2. Consider any set of four leaves as mentioned above. The addresses of these four leaves will be different in their $(k-3)$ m.s.b's, from those of leaves in the other sets since the address of the parents of the latter would be different in B'. The addresses of these four leaves are obviously different in their three l.s.b's. Hence every element in B is distinct.

Since the parent, p of these four leaves is an internal vertex in $CQT(h)$, the three l.s.b's of $B[p]$ can be either 000 or 100, which are considered in Case I and Case II respectively. In both cases it may be verified that the addresses $B[p]$ and $B[BFSN_i]$, $\forall\, i = 1,4$ differ in at most two bit positions. Hence step 2 maintains a dilation of 2. Hence the claim.□

From *Claim 3.4.1 and Claim 3.4.2* the proof of Theorem 3.1 follows.□

Corollary 3.2: Consider embedding $CQT(h)$ into $H(n)$ using theorem 3.1. Let any vertex, u, at the level $(h-1)$ in the $CQT(h)$ be assigned to the node, v in $H(n)$. There exists at least one node in $H(n)$ which is adjacent to v and not used in the embedding.

Proof: Note that u is an internal vertex. Let X be the binary address of v. From the Step 1 given in the proof of theorem 4.4.1, it follows that the three l.s.b's of X are either 000 or 100. These three bits are used to assign the children of u. Let $X = Y000$ or $X = Y100$ where Y is a binary vector of size $k-3$. If $X = Y000$ the nodes with binary addresses $Y010$ and $Y100$ are adjacent to X and are not used in the embedding. On the other hand, if $X = Y100$ the node with binary address $Y110$ is adjacent to X and is not used in the embedding. Hence the corollary.□

Consider embedding $CQT(h)$ into $H(n)$ using theorem 3.1. Let k be the dimension of $H(n)$, i.e, every node in $H(n)$ has a unique k-bit binary address. We define a *cube* as a set of eight nodes in $H(n)$ which have $(k-3)$ identical m.s.b's in their binary address. We refer to these $(k-3)$ m.s.b's as the address of the *cube.*

Corollary 3.3 The theorem 3.1 embeds a $CQT(h)$ into $H(n)$ such that in every cube of $H(n)$, only one node is used to map a vertex at level $(h-1)$ of $CQT(h)$.

Proof: Let u be a vertex at level $(h-1)$ of $CQT(h)$, and X be the binary address of the node, $v = f(u)$ in $H(n)$. From the Step 1 given in the proof of theorem 3.1, it follows that the three l.s.b's of X are either 000 or 100. Since only one of these bit addresses is used for mapping u and all other nodes in the cube are either used to map the children of u or are free nodes, the proof follows.□

Corollary 3.4: The number of free nodes in any cube of $H(n)$ after the embedding is either two or three.

Proof: Let Y be the address of any cube in $H(n)$. A node, v with address either $Y000$ or $Y100$, is used for the assignment of an internal vertex, u in $CQT(h)$. If the address of v is $Y000$ then the nodes with addresses $Y010$, $Y100$, and $Y111$ are free nodes. Hence the cube has three free nodes. On the other hand, if the address is $Y100$ then the nodes with addresses $Y011$ and $Y110$ are free nodes. Note that $Y000$ is either used in the embedding or is a free node. Hence the corollary.□

3.4.1 Characterization of free node distribution

Initial embedding should have features that makes the reconfiguration simple for all instances of the homogeneous faults. Though the fault instances differ from one to other, the even distribution imposes some common characteristics among the variabilities of the number of faults in successive levels of the tree. Our aim is to provide an embedding which obtains a free node distribution which is close to the homogeneous fault distribution on the average. The embedding should enable reconfiguration for any fault sequence with minimal changes to the initial embedding. The embedding algorithms available in the literature such as [27] do not meet the above requirements. The following subsections analyze the properties our embedding algorithms which may be used for reconfiguring faults.

The generalization of free node distribution is characterized in the following two lemmas. Note that the number of free nodes at the root of the tree is of the form $4x-1$, for some integer x. If the numbers in the fault distribution at any level consists of two different numbers with a difference of 1 between them, we show that the larger of the two is always of the form 4X-1.

Lemma 3.2: The number of free nodes at the root of the $CQT(h)$ will be an integer of the form $(4X-1)$, where X is recursively defined as an integer of the form $(4Y-1)$ for all $Y \geq 3$.

Proof: The maximum number of free nodes, F, is equal to the number of free nodes in $H(n)$ after the initial embedding. We determine F, the difference between the number of nodes in $H(n)$ and the number of vertices in $CQT(h)$, as follows.

$$F = n - m;$$

$$= 2^k - \frac{4^{h+1}-1}{3};$$

$$= 2^k - \left[4\left(\frac{4^h-1}{3}\right)+1 \right];$$

(since $CQT(h)$ may be obtained from 4 copies of $CQT(h-1)$ with the addition of a new root)

$$= 2^{2h+1} - 4\left(\frac{4^h - 1}{3}\right) - 1;$$

(from Lemma 2.3.1, $k = 2h + 1$)

$$= 2\,4^h - 4\left(\frac{4^h - 1}{3}\right) - 1;$$

$$= 4\left[\frac{4^h + 2}{6}\right] - 1;$$ which is of the form $F = 4X - 1$, where $X = \frac{4^h + 2}{6}$.

By similar derivation it may be shown that $X = 4Y - 1$ where $Y = \frac{4^{h-1} + 2}{6}$.

Further, we show that X is an integer as follows.

Claim 3.4.2.1: $X = \frac{4^h + 2}{6}$ is an integer.

Proof: We prove the claim by induction on h.

Basis: When $h = 1$, $X = 1$ and clearly the claim holds good.

Hypothesis: Assume that the claim holds good for all integers less than h.

Induction Step: Let $y = \frac{4^{h-1} + 2}{6}$, and from the hypothesis y is an integer. Note that the terms in the above expression for y may be rearranged as $4^{h-1} = 6y - 2$.

We prove that $X = \frac{4^h + 2}{6}$ is an integer as follows.

$$X = \frac{4^h + 2}{6}$$

$$= \frac{4\,4^{h-1} + 2}{6}$$

$$= \frac{4\,(6y - 2) + 2}{6}$$

(substituting the terms for 4^{h-1})

$= (4y - 1)$, which is an integer.

Hence the claim 3.4.2.1. It may be seen that for all $X = 4Y - 1$ which is greater than or equal to 3, Y will be an integer. Hence the Lemma 4.2.□

Definition 3.1: If the number of free nodes at any non-terminal vertex in $CQT(h)$ is of the form $(4X - 1)$ the free nodes may be homogeneously distributed among its children as the permutations of four integers $X - 1$, X, X, and X.

Definition 3.2: If the number of free nodes at any non-terminal vertex in $CQT(h)$ is of the form $(4X - 2)$ the free nodes may be homogeneously distributed among its children as the permutations of four integers $X - 1$, $X - 1$, X, and X.

Note that in both of the resulting free nodes distributions among the children, the difference in the number of free nodes between any two children is at most 1.

Lemma 3.3: The number of free nodes at any non-terminal node of the $CQT(h)$ will be an integer of the form $(4X - 1)$ or $(4X - 2)$, where X is recursively defined as an integer of the form $(4Y - 1)$ for all $Y \geq 3$.

Proof: We prove the lemma by induction on the level, l of the non-terminal node.

Basis: For $l = 0$ the non-terminal vertex is the root itself, and the number of free nodes at the root is of the form $(4X - 1)$ from the lemma 3.2. Hence the basis is true.

Hypothesis: Assume that the number of free nodes at all non-terminal nodes at level less than l is of the form $(4X - 1)$ or $(4X - 2)$ where X is recursively defined as an integer of the form $(4Y - 1)$ for all $Y \geq 3$. Note that the number of free nodes at the root is of the form $(4X - 1)$ (from the basis), which may be distributed using definition 3.4.2.1, among its children as a permutation of $X - 1$, X, X, and X Further, number of free nodes at the child with $X - 1$ free nodes, will be of the form $(4Y - 2)$ since X may be written as $(4Y - 1)$ for some integer Y. Also note that the number of free nodes of the form $(4Y - 2)$ may be distributed as given in definition 3.4.2.2.

Induction Step: Consider a non-terminal vertex, u, at level l in $CQT(h)$. Let v at level $(l-1)$ be its parent. From the hypothesis the number of free nodes at v may be of the form $(4X-1)$ or $(4X-2)$.

Case I: The number of free nodes at v is of the form $(4X-1)$.

The number of free nodes at u may be either $X-1$ or X as can be obtained from the fault distribution using definition 3.4.2.1. Since X is of the form $(4Y-1)$, the number of free nodes would be of the form $(4Y-2)$ or $(4Y-1)$ respectively.

Case II: The number of free nodes at v is of the form $(4X-2)$.

The number of free nodes at u may be either $X-1$ or X as can be obtained from the fault distribution using definition 3.4.2.2. Since X is of the form $(4Y-1)$, the number of free nodes would be of the form of $(4Y-2)$ or $(4y-1)$ respectively. Hence the proof.□

3.4.2 Homogeneous faults in quadtrees

In many of the computational models based on tree structures, (such as binary trees and quad trees) the processors at the leaves perform the actual computation while the processors at the internal nodes of the tree do minimal computation such as merging and communication between processors. Hence any failure of the processors at the leaves is critical since these failures could become major bottleneck for the whole computation. The focus of our present research is to develop reconfiguration algorithms that consider the faults at the leaves of the tree.

3.4.3 Characterization of homogeneous faults

We assume that the failure at any leaf is equally likely. We refer to a fault distribution as an even distribution if the number of faults at any internal vertex is distributed among its four children such that the difference between the faults for any two children is at most 1. We identify a class of faults, called *homogeneous faults* in

which the faults occur only at the leaves of $CQT(h)$ such that the total number of faults at any level of the tree adds up to an *even distribution.* The number of faults for any vertex at level $(h-1)$ is the number of children(leaves) of the vertex which are faulty. The number of faults for any vertex at level $l < (h-1)$ is the sum of the number of faults for its four children. Thus the total number of faults would be maximum for the root of $CQT(h)$ at level 0, and it would at most be equal to the number of free nodes in $H(n)$.

Based on the description of the homogeneous fault distribution and the nature of the number of free nodes available in $H(n)$ after embedding a $CQT(h)$, we make the following observations:

1. The number of free nodes is the difference between the number of vertices in $CQT(h)$ and the number of nodes in the nearest hypercube, $H(n)$. Homogeneous fault distribution is the collection of all possible fault sequences satisfying the properties of even distribution of faults among the children of any internal node. It may be noted that the numbers in a valid homogeneous fault sequence may be ordered in more than one way such that the resulting sequence will still be a valid homogeneous sequence. Hence it is not trivial to provide an initial embedding such that reconfiguration may be provided for any fault sequence.

2 It is counter-intuitive that a constant dilation transformation of *FTS* to *FNS* exists especially when *FTS* and *FNS* are not identical. In order to obtain a conflict free reassignment for any faulty node to a free node with a 3-dilation, the free nodes should be distributed in such a way that the difference between *FTS* and *FTN* may be reconfigured in case of all instances of homogeneous faults.

After the embedding, the number of faults at the root of the $CQT(h)$ distributed homogeneously among the quadrants of the four children recursively. The total number of free nodes and hence that of faults in case of the embedding $CQT(h)$ into $H(n)$ is of the form $4X - 1$ as given by Lemma 3.2. In homogeneous fault distribution allows a fair and equitable distribution of faults at any internal node such that the difference between the faults in any two of its children would be at the most 1. However, the feasible distribution of a given number of faults of the form of $4X - 1$ might allocate $X - 1$ faults to any of the child while X faults would be allocated to each of the other three children. By allowing the flexibility of choosing the index of the child to which $X - 1$ faults would be allocated at random at each stage of recursive distribution and still maintain the condition of homogeneity could clearly give rise to more than one fault sequence. Hence an instance of homogeneous faults, *FT* can be any one of the feasible sequence from the homogeneous distribution.

Theorem 3.1 states that the distribution of free nodes in $H(n)$ after embedding $CQT(h)$ follows a homogeneous sequence. If the fault sequence,*FT*, is identical to the free-node sequence, the reconfiguration involves reassigning free nodes in the same quadrant as those of the faulty nodes. However, if a valid homogeneous fault sequence *FTS* which is not identical to *FNS* occurs, it is necessary to redistribute the free nodes to satisfy the reconfiguration requirements. Such a redistribution will locate the quadrants in which free nodes exceed th number of faults, and find a migration path to meet the dilation requirement for each of the fault recovered.

Reconfiguration scheme uses the process migration and local faults. We show that our algorithm creates minimum number of migrations equal to number of faults. It also maintains an optimal dilation of 3. Hence our algorithm is optimal with respect to both migration and communication cost.

3.4.4 Reconfiguration process

The algorithm to reconfigure the embedding is executed by any node in the hypercube acting as a controller after the faults are detected. It is a recursive algorithm called at each level of the quad tree. There are three steps in reconfiguration process- determine the free node sequence, determine the fault node sequence, and reconfigure. In order to establish the concept of reconfiguration, we provide a schematic diagram of the reconfiguration process in figure 3.4.

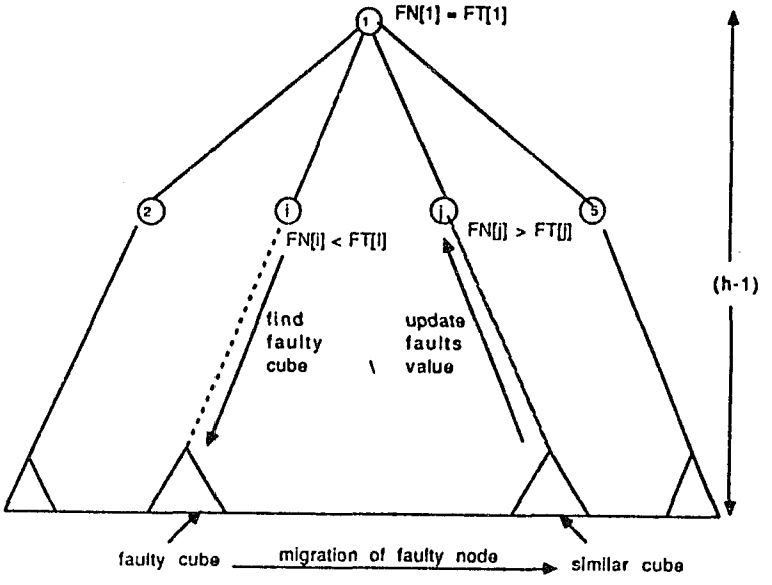

Figure 3.4 Schematic Diagram of Reconfiguration Process

3.4.4.1 When FT is identical to FN

Since *FT* is identical to *FN*, the fault sequence at the level $h-1$ is identical to the free node distribution at the individual cubes. The reconfiguration algorithms provide the local adjustments in the cube level. Since the number of faults is equal to the number of free nodes in any cube, each faulty node may be reassigned to a unique free node. Since the maximum dilation between any two nodes in a cube is 3, such reassignment will maintain 3-dilation. The actual assignment of specific free nodes in a cube to the faulty nodes may be systematically obtained in the order of the

BFSN of the faulty vertices to ordered free nodes. These steps are referred to as *cube level reconfiguration* in figure 3.4.

3.4.4.2 When FT is not identical to FN

The idea here is to transform the fault distribution to the free node distribution. These two distributions are stored in arrays, and the indices are the BFSN of the quadtree vertices. Hence, we can think of these distributions arranged in the form a quadtree of height $h-1$. Using the convention mentioned in section 4.1, we use all the terminologies applicable to the tree structure such as level, children, etc. We use figure 3.4 and figure 3.5 to discuss the reconfiguration process.

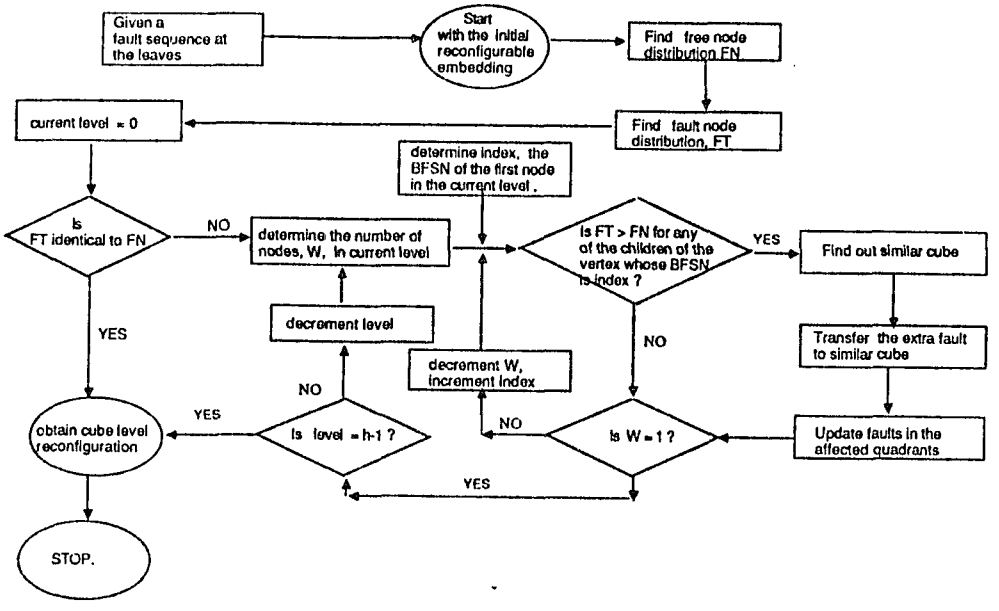

Figure 3.5 Steps in Reconfiguration

Reconfiguration process redistributes the excess faults in a child of an internal node to any one of three other similar nodes which contains extra free nodes so that homogeneous fault distribution may be achieved in each level. The algorithm follows a top down approach. For each internal node in every level, the algorithm checks if the number of faults in any of the children is greater than the number of free nodes. If the number of faults is less than the free nodes, reconfiguration will be

a simple instance of the worst case maximum failure. In such a case the reconfiguration is achieved when the algorithm proceeds to lower levels. Hence, we consider the worst case of failure involving maximum number of faults in all our illustrations. If number of faults is greater than the number of free nodes in any of the children of the current node, the steps necessary to reconfigure are shown in figure 3.5 in which the numbers kept inside the circles are the BFSNs, while those placed near the circles are the free node or fault value.

The distribution lists *FT* and *FN* are traversed in a breadth-first fashion, and at each level the contents of these lists with the same index, are compared. If the number of free nodes is greater or equal to the number of fault nodes in all four children of any internal node, the faults are either distributed homogeneously in that quadrant or the distribution will be adjusted in the lower levels. We explain the three steps for reconfiguring the faults at one internal node. It may be noted that the same procedure is applied to all the internal nodes in each level. We consider the case when the number of faults in one of the children is greater than the number of free nodes. We now discuss how the *FT* is transformed to *FN* using the procedure *homogeneous root.* The transformation is explained in three steps - find similar cube; transfer the extra fault to similar cube; and update the number of faults in the affected quadrants. Refer to figure 3.4 and figure 3.5 for the schematic outline.

(1) *find out similar cube*

Let i be the index of the child for which it is found that $FN[i] < FT[i]$. This situation implies that there is a *cube* in the quadrant where the subtree rooted at i is placed, and has more faults than the free nodes. Then there must exist another child with index j such that the number of free nodes in that quadrant is more than the number of faults which follows from the properties of the homogeneous fault sequence. We now traverse down the subtree rooted at i and locate the *faulty cube* in

which the number of faults is greater than the number of free nodes. The procedure *Fault_cubes_1* or the procedure *Fault_cubes_2* is used for this purpose. Locate the *similar cube* in the subquadrant where the subtree rooted at j is placed. Note that there is at least an *extra free node* in this *cube.*

(2) *transfer the extra fault to similar cube*

Transfer the extra fault in the *faulty cube* to the *similar cube* located in the previous step. This is done by the initial part of the procedure *Find_migration_path.* The vertex in the leaf which caused the extra fault is assigned to the free node in the similar cube which at dilation-3 from the parent of the faulty leaf. Hence the number of faults in the faulty cube is reduced by one while the number of free nodes is reduced by one in the similar cube. Note that after this transfer, the invariant that the number of free nodes in any quadrant is greater than or equal to the number of faulty nodes still holds.

(3) *update faults in the affected quadrants.*

The fault node distribution and free node distribution are updated in both the subtrees rooted at i and j. The procedure *update_faults* carries this work in the bottom up fashion. When this procedure is completed, the faults are homogeneously redistributed among the children of the internal node which started this procedure.

Theorem 3.2: The reconfiguration algorithm maintains an optimum dilation of 3 and the number of migrations is equal to the number of faults which is also optimum.

Proof: The cube level reconfiguration uses the free nodes in the same cube as the parent of the faulty vertex and thus maintains a dilation 3.

The migration of *extra* fault to an *extra free node* does not result in more dilation since the migration takes place among *similar cubes.* From the definition of *similar cubes,* it can be seen that the parents of the *faulty leaf* and the *extra free node* are at the most at dilation-2 in the hypercube. Recall that in the initial embedding

obtained using theorem 4.4.1, the parent of all the leaves have at least one free node adjacent as given in Lemma 4.1.3. Suppose a leaf node v is an additional faulty node. Let the parent of v be u. There are three other similar nodes(similar to u) in the other three sub-quadrants at a distance of at most 2. One of these quadrants will have a vertex, w which is a *similar node* with respect to u in this quadrant. Since we guarantee that a free node adjacent to w always exists in the embedding from Theorem 3.1, u may be reconfigured in to the similar node w.

After the fault propagation to w, the number of faults and number of free nodes in each quadrant at the level of u is made identical. However, the cube containing w should be able to reassign the fault to the free nodes in the same quadrant. This is possible since the resulting cube can have at the most two extra faulty nodes to be reassigned - one extra fault because of the property of the homogeneous fault sequence; and the other extra fault because of the migrated fault from the similar cube. Since there will be two similar nodes in the same quadrant and the sum of faults in the quadrant is equal to the free nodes, both extra faults may be reassigned to the free node adjacent to the similar nodes at dilation 3. Since reconfiguration at any level involves the similar steps, 3 dilation is guaranteed in all the steps explained in section 4.5.1.2.

Hence the proof.□

3.5 Reconfiguration in case General Faults

Let us consider a binary hypercube,$H(n)$, as the host graph and a complete quadtree, $CQT(h)$, as the guest graph. The strategy of the reconfiguration technique may be expressed in two steps as follows:

(1) Embed $CQT(h)$ into $H(n)$ to obtain a free node sequence, *FNS*. The reconfiguration technique tolerates maximum number of faults which is equal to the number of available free nodes in the hypercube after the initial embedding.

(2) Develop a generalized algorithm which can transform *FTS* to make it identical to *FNS* if the given instance of fault sequence *FTS* is different from the free node sequence,*FNS*.

We implement the following three approaches for obtaining the initial embedding and further reconfiguring the embedding for a given fault sequence: greedy embedding with reconfiguration using migration of faulty as well as non-faulty nodes, greedy embedding with reconfiguration using migration based on local search for free nodes, and reconfigurable embedding.

3.5.1 Greedy embedding with migration

Three important steps in this approach are: (a) enumerate a new embedding which does not make use of any of the faulty nodes, (b) minimize migrating processes to new processors if possible and, (c) migrate processes to new processors.

Embedding a guest graph, G onto a host graph, H, is a bijective function, f. It may be possible to have several different embeddings that do not make use of the faulty processors. In case of faults in any of the nodes used in the initial embedding, it may be possible to choose another embedding in which the faulty node is not used. Let the initial assignment of any process, j be the processor p_i, and the assignment in the latter embedding be p_f. If p_i and p_f are different for any process j then the process needs to be migrated from p_i to p_f.

Note, however, that the greedy process involves a combinatorial search which is computationally expensive. We have studied some variations of this technique to obtain sub-optimal efficient solutions. Advantage of this technique however is that it is applicable to many variety of faults.

3.5.2 Greedy migration using local search

This approach is similar to greedy technique above, but neighborhood relation of the nodes because of the architectural design may be incorporated. For example, we have considered a variation of greedy technique in which the next node considered for greedy enumeration are based on the grey code sequence to reflect the local search effect.

Remaining steps are similar to the technique described above. Note, however, that the process of generating neighborhood information can be achieved efficiently compared to a combinatorial search, however, the quality of solutions are not as good as exhaustive greedy techniques which search larger solution space. We have studied some variations of this technique also to obtain sub-optimal efficient solutions.

3.5.3 Reconfigurable embedding scheme

Our approach to reconfiguration of the initial embedding is to modify those portions of the embedding that are affected by faults. This approach may make use of the knowledge of the initial embedding and the nature of the faults. The motivation is to provide a reconfiguration which minimizes the number of process migrations. Accordingly, we provide two schemes: (1) *one-to-one* scheme, and (2) *Fault Propagation* scheme. The first scheme may be used when there exists a direct assignment of faulty nodes to free nodes which meets the requirements regarding quality. The second approach goes about finding one such assignment allowing the migration of minimal number of non-faulty nodes.

Figure 3.6 gives the diagrammatic representation of some of these definitions and data structures.

Definition 3.3:

Let f be the embedding function, given as: $f : v_G \rightarrow v_H$, where v_G, and v_H are the nodes of the *guest graph, G*, and the *host graph, H* respectively.

Definition 3.4:

Let f^{-1} be the *inverse* of f, and given as: $f^{-1} : v_H \rightarrow v_G$.

Definition 3.5:

For any node u in the guest graph, let $ADJ[u]$ be the set of all nodes in the guest graph that are *adjacent* to u. i.e.,

$ADJ[u] = \{ v: v \in G \text{ and } (u, v) \in E(G) \}$.

Definition 3.6:

For any node u in the host graph, we define *d-neighborhood*, $NBR[u, d]$ as the set of all vertices in the guest graph which are mapped to the nodes in the host graph as well as the free nodes that are at a distance of d or less from u. i.e.,

$NBR[u, d] = \{ f^{-1}(v) \cup r_i : u, v \in H\ r_i \in FN\ ,\ 1 \leq i \leq k\ ,\ dist(u, v) \leq d\ ,$

$\text{and } dist(u, r) \leq d \}$.

Definition 3.7:

For any faulty node, $u \in V(G)$, we define a data structure, $DLIST[u]$, which may be obtained as follows:

1. Determine image, $v \in V(G)$, of u, namely, $f^{-1}(u)$.
2. Determine $ADJ[v]$.
3. Obtain a set $U \subseteq V(H)$, by applying f on each element of $ADJ[v]$.
4. For any node, $x \in V(H)$ if $U \subseteq NBR[x, d]$ then insert x in $DLIST[u]$.

In other words, $DLIST[u]$ of any faulty node u, contains all nodes in the host graph whose d-neighborhood has U as a subset. i.e., $DLIST[u]$ is defined as follows:

$DLIST[u] = \{ x : x \in V(H), \text{and } U \subseteq NBR[x, d] \}$, where

$U = \{ w : \text{for } all\ s, w = f(s), s \in ADJ[v], \text{and } v = f^{-1}(u) \}$.

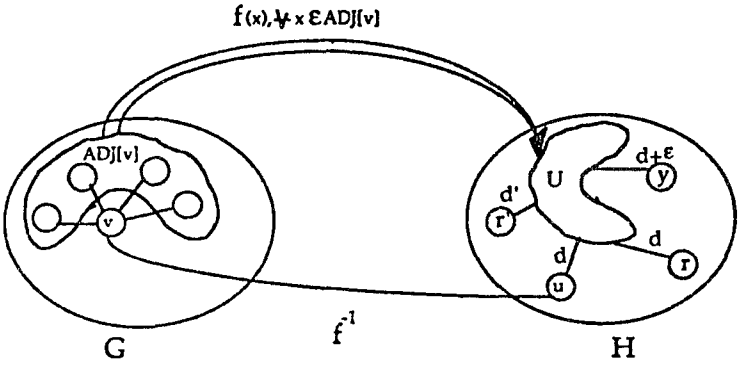

Figure 3.6 Schematic Representation of Reconfiguration Process

Definition 3.8:

For any free node r in the host graph, let $RLIST[r]$ be a set obtained as follows:

For each faulty node, $u \in FT$ do steps (1-2):

1. Determine $DLIST[u]$ using the definition 3.7.
2. If $r \in DLIST[u]$ then insert u into $RLIST[r]$.

In other words, $RLIST[r]$ of any free node, r, contains those faulty nodes whose adjacent nodes are in the d-neighborhood of r. More formally we define $RLIST[r]$ as follows:

$RLIST[r] = \{ u : u \in FT, \text{and } r \in DLIST[u] \}$

Before we give the details on the two schemes mentioned above, we describe certain steps which will allow us to choose one of these schemes :

1. For each node, $x \in V(H)$, determine its *d-neighborhood*, $NBR[x, d]$ where d is the maximum dilation in the initial embedding.
2. For each faulty node, $u \in FT$, obtain $DLIST[u]$.
3. For each free node, $r \in V(H)$, obtain $RLIST[r]$.

The $DLIST[u]$ contains all those nodes in the host graph that may replace the faulty node, u and maintain the same dilation, d. Note that such a replacement provides the mapping of the image of the faulty node, and its adjacent nodes with in the dilation criteria. An element in the $DLIST[u]$ may either be used in the initial embedding or may be free.

The $RLIST[r]$ contains all those faulty nodes in the host graph that may be replaced by this free node r and maintain the same dilation.

We use *RLIST* to develop a graph structure called *Assignment Graph(AG)*, which is used to determine whether there exists a direct assignment of faulty nodes to free nodes while maintaining the same dilation. Based on the necessary conditions given in Theorem 4.1, we determine the existence of a direct assignment, which leads to the one-to-one scheme. If a direct assignment is not possible we use the fault-propagation scheme in which a node in the *DLIST* of a faulty node is selected for replacement. If the selected node was used in the initial embedding we call that replacement as a local fault, which may be propagated until a free node is found for replacement.

Definition 3.9 :

We define a bipartite graph called *Assignment Graph*, $AG = (V_1 , V_2 , E)$ as follows:

V_1 is a set of vertices such that every vertex in V_1 is equivalent to a unique node in the faulty set, *FT*. Formally we define V_1 using a bijective function $f : FT \rightarrow V_1$ as:

$$V_1 = \{\ x : x = f(y), \textit{for each } y \in FT\}.$$

V_2 is a set of vertices such that every vertex in V_2 is equivalent to a unique node in the free node set, *FN*. Formally we define V_2 using a bijective function $g : FN \rightarrow V_2$ as:

$$V_2 = \{\ x : x = g(y), \textit{for each } y \in FN\}.$$

E is the set of undirected edges defined over V_1, and V_2 as obtained from the relation defined in *RLIST* for all the free nodes as follows:

$$E = \{(u, v) : \forall\ u \in V_2\ ,\ f^{-1}\ (v) \in RLIST[\ g^{-1}\ (u)\ ]\}.$$

It may be seen that for every vertex in V_1 there is an equivalent, and unique faulty node. Similarly for every vertex in V_2 there is an equivalent, and unique free node. An edge between a vertex in V_1 and a vertex in V_2 implies that the corresponding faulty node may be replaced by the corresponding free node.

We now state a theorem to determine the existence of a *direct assignment.*

Theorem 4.1: For the existence of a *direct assignment,*

the necessary conditions are:

(1) The union of *RLIST* for all the free nodes must be the faulty set, *FT*.

(2) *RLIST* for at least k of the free nodes must be non empty in order to reconfigure k faults.

and the sufficient condition is:

(3) a complete matching of V_1 into V_2 must exist in the *Assignment Graph, AG.*

Proof: When the union of *RLIST* for all the free nodes is not equal to the faulty set, *FT*, clearly there exists at least one faulty node which can not be directly assigned to a free node. Hence the condition (1) follows.

Let the number of faulty nodes be k. Let the the number of free nodes whose *RLIST* is non-empty be l. When $l < k$, clearly there are $k - l$ faulty nodes which can not be directly assigned to free nodes. Thus the condition (2) follows.

Assignment Graph, *AG* is a bipartite graph by definition. The existence of a complete matching in *AG* implies that there exists a *direct assignment* of every vertex in V_1 to a unique vertex in V_2. From the definition of the bijective functions f and g, and *RLIST* , it follows that every faulty node may be replaced by a unique free node if and only if there exits a complete matching.

Hence the theorem.□

One-to-one Scheme

This scheme may be used if it is possible to make direct assignments. Following steps are necessary for reconfiguration.

1. Obtain *RLIST* for all the free nodes.
2. Construct the Assignment Graph and test whether there exists a direct assignment using theorem 4.1.
3. If a direct assignment is possible, use the corresponding free nodes in the complete matching to replace the faulty nodes using process migration.

Fault Propagation Scheme

When a *direct assignment* is not possible we use this scheme for reconfiguration. Here we select a node in the *DLIST* of a faulty node for replacement. If the selected node was used in the initial embedding we call this replacement as a *local fault*, which may be propagated until a free node is found for replacement.

Definition 3.10:

Let u be a faulty node in the host graph, and $u' = f^{-1}(u)$ be the corresponding vertex in the guest graph. Let $v \in DLIST[u]$ be a node chosen for reconfiguring the

fault at u. This means that the vertex u' will be mapped onto the node v. If the node v was used in the previous embedding, let $v' = f^{-1}(v)$ be the corresponding vertex in the guest graph. After the reconfiguration v' needs to be mapped on to some other node in the host graph, which we refer to as *local fault.*

Definition 3.11:

We define *Fault Propagation* as the recursive process of reconfiguring a faulty node by creating a local fault, and reconfigure this fault by another local fault or by a free node.

Definition 3.12:

We define a *Fault Propagation List*, $FPL[u]$ as a set of all the intermediate nodes used to propagate the fault at u until a free node is found by the fault propagation process.

We now describe a procedure which enumerates a $FPL[u]$ for a faulty node, u to any of the free node as follows:

1. For each node, x in the host graph, determine $NBR[x, d]$ where d is the maximum dilation of the initial embedding.
2. For each vertex, y in the guest graph, determine $ADJ[y]$.
3. Let u be the faulty node in the host graph. Determine $DLIST[u]$ by selecting all nodes x whose neighborhood $NBR[x, d]$ contains $ADJ[f^{-1}(u)]$, where f is the mapping function.
4. Choose a node, $v \in DLIST[u]$.
5. If v is a free node, u' is mapped onto v, output $FPL[u]$, and stop. If v is not a free node, u' is mapped onto v, and a local fault is created at v to reconfigure v'.
6. If a *local fault* is created at step (5), then we update all the $NBR[x, d]$ by substituting each occurrence of v in any of the $NBR[x, d]$ by u.

7. Add v to the *Fault Propagation List*, $FPL[u]$.

8. Repeat the steps (3-8) recursively, for the local fault at v till a free node is located in step 5.

The above procedure must be used to find $FPL[u]$ and update assignments for every u which does not appear in at least one of the $RLIST[r]$, and to reach every free node r which does not appear in any of the $DLIST[u]$. It is necessary to find alternate $FPL[u]$ if a complete matching is not obtained in the updated Assignment Graph.

Note that in step (4), there may be more than one choice for v, and each selection of v yields a different $FPL[u]$. For a given pair of a faulty node and a free node, we may find more than one $FPL[u]$. These iterations may be carried out in several different ways based on various optimizing criteria such as : (a) matching which needs minimum number of local faults, and (b) matching which results in the best average dilation. We are currently investigating efficient solutions to characterize some of these optimization criteria, and their complexity issues [28]. We limit our discussion only to establish the usefulness of reconfiguration strategy since the details of the analysis of fault propagation scheme is beyond the scope of this chapter.

3.6 Performance Analysis of Various Algorithms

The performance of greedy algorithms is compared with our technique. We have used some of the important parameters for comparison such as pre-fault dilation, post-fault dilation, and number of faults. The table indicates that our technique is unique in the sense that this is the first reconfigurable embedding technique with superior performance.

We recognize two types of overheads(costs) in reconfiguring an interconnection network. They are: (1) cost due to the migration of faulty processes to free processors called *migration-cost, MC*, and (2) increase in run-time cost called *dilation-cost,*

DC. The migration-cost,MC, of any reconfiguration algorithm would be minimum if the processes migrated are only those at the faulty nodes. The dilation-cost,DC, will be kept to a minimum if the dilation of the reconfigured system remains the same or as close to that of the initial mapping as possible. Reconfiguration algorithms based only on the operational aspect will have the minimum-MC overhead, and may not have the minimum-DC. For example, an arbitrary assignment of faulty processes to free processors, though make the network operational, may not maintain the quality of the initial mapping. These algorithms are not suitable for applications that require minimum run-time cost. On the other hand, algorithms which focus on the quality aspect must have the minimum-DC. We have developed reconfiguration approaches which obtain a trade-off in increase of both dilation-cost and migration-cost such that the overall run time cost may be minimized.

3.7 Conclusions

We have introduced the problem of how to continue the computation whenever some of the nodes used in the embedding are diagnosed as faulty. We have examined reconfigurable embedding techniques which tolerate faults as well as maintain the qualities of embedding. The idea of the reconfiguration while maintaining the characteristics of the initial embedding is new. Realizing the importance of exploiting the inherent parallelism and the need for fault tolerance in computationally intensive applications such as image processing and computer vision, reconfigurable embedding algorithms for complete quadtrees have been presented. Since architectures based on binary hypercube topology have gained widespread acceptance in these applications, we have used hypercube architecture as the interconnection network in our research.

Reconfiguring an embedding in case of faults could be difficult since the faults might make the topology of the architecture arbitrary in which case the problem of

finding a new embedding could be NP-complete. Hence we studied the certain types of faults that would allow a tractable reconfiguration. We have identified a class of homogeneous faults which yield efficient reconfiguration.

The main contribution of this chapter is the identification of a new class of reconfigurable embedding strategies which may be used to recover from faults as well as maintain the quality of the mapping. Our research has focused on three issues: (i) to make use of the free processors, which are not used in the embedding, for reconfiguration ; (ii) to tolerate as many faults as the number of free processors; and (iii) to maintain the quality of embedding in the resulting reconfigured system. In the case of reconfiguring a complete quadtree into a hypercube, we have provided a 2-dilation reconfigurable embedding which yields an optimal 3-dilation reconfiguration. Our approach uses process migration paradigm which uses minimum number of migrations.

We compared our results with existing reconfiguration and embedding algorithms and have shown the superior performance of our algorithm. It is possible to extend our methodology for a variety of computational graph structures onto various interconnection networks such as star graphs, and generalized boolean n-cubes.

CHAPTER 4

DISTRIBUTED BICONNECTIVITY TESTING

Alternatives do not usually have the courtesy to parade themselves in rank order on the drill ground of the imagination

-- Kenneth E. Boulding

4.1. Overview

The problem of determining whether a given graph G is biconnected or not is well known in the area of algorithm design and graph theory [1,2]. Testing biconnectivity is both of theoretical and of practical importance since biconnected components offer a natural decomposition of graphs yielding efficient solutions to many problems, such as Planarity Testing, based on each component. Fault tolerant applications in distributed systems will benefit from biconnectivity since the failure of a node will not disconnect a biconnected network. Knowledge of biconnectivity may also be used for selective process replication for fault tolerant distributed systems.

Distributed Biconnectivity Testing (*DBT*) problem provides a local code for each process to be executed so as to find out whether the graph G, corresponding to an underlying communication network of a distributed system is biconnected or not. Given that all computations at any node have to be based only on the data available at a local site, the data from other nodes which are of global interest must be obtained by communicating through neighbors alone. Most important concern in such a distributed computation is to solve the *DBT* problem using a minimum number of messages to route any non-local data when necessary. Assuming that the time taken by individual nodes for computation is negligible, we investigate the message complexity of *DBT* problem. Computation should terminate in finite time and at the

end of the computation each node should correctly know whether it is an articulation point, whether an edge incident on the node is a bridge; and each node should be labeled so that all the members of a biconnected component may be determined.

Usefulness of depth first search(DFS) paradigm to solve biconnectivity testing problem is well established [1]. Algorithms for distributed depth-first-search (DDFS) for graphs representing communication networks may be found in [3,4,5,6]. The most efficient algorithm known for DDFS uses exactly $2n-2$ messages and terminates after $2n-2$ units of time [6]. Even though many efficient $O(n)$ message and time complexity distributed algorithm for DDFS are currently known, a linear message complexity distributed algorithm is not reported in literature for determining biconnected components.

There has been a considerable amount of research on distributed biconnectivity testing and its various characterizations [7,8,9]. The most efficient algorithms known for distributed recognition of biconnected components uses $O(m)$ messages and $O(d)$ time with a bound of $O(\log n)$ on message size [7]. We take the approach of designing a distributed algorithm by adapting a sequential algorithm to a distributed environment. We specifically use the techniques in [6,10,11] to provide an algorithm with linear complexity in both the number of messages and the time.

In Table 4.1, we summarize the performance of algorithms for *DBT* considering unbounded message lengths for message complexities. It may be noted that the message size in our algorithm is $O(n)$ bits while it is $O(\log n)$ for the others in the table and still a saving of $O(\log n)$ bits is obtained in the worst case.

4.2 Preliminaries

We adapt a sequential biconnectivity testing algorithms of [1,2] for our distributed implementation. We re-state some of the essential concepts that are needed for improving readability. We refer to [1,2,12,13] for further details.

Table 4.1. Summary of Related Recent Results

Author	Year	Time Complexity	Comm. Complexity	message length
E.J.H. Chang [10]	1982	$O(d)$	$O(m)$	$O(n\log n)$
S. T. Huang [8]	1989	$O(d+kn^2/m)$	$O(kn^2)$	$O(\log n)$
Ahuja & Zhu [9]	1989	$O(n)$	$O(m)$	$O(\log n)$
Lin et. al.[8]	1990	$O(d+k)$	$O(m+kn)$	$O(\log n)$
Hegde & Iyengar [proposed here]	1992	$O(n)$	$O(n)$	$O(n)$

A vertex u in an undirected graph $G_0 = (V_0, E_0)$ is an *articulation point* if the removal of u splits G_0 in to two or more parts. An edge $e = [u, v]$ in G_0 is a *bridge* if every spanning tree in G_0 contains e. Further, G_0 is *biconnected* if and only if it does not have any articulation points.

A depth-first-search (*DFS*) on an undirected graph $G_0 = (V_0, E_0)$ gives a directed graph $G = (V, T, B)$ where in the vertices in V are those of V_0 labeled with depth-first-number $DFN(v), v \in V_0$, and the edges in E_0 are divided in to two mutually exclusive sets of directed edges: Tree edges, T and Backedges, B. We refer to the union of T and B as E. Let $n = |V|$, and $m = |E|$. If $e = [u, v] \in E$ then u is called the *father* of v and v is a son of u. If there is a *path* from u to v, then u is an *ancestor* of v and v is a *descendant* of u, and further, if $u \neq v$ then u is a *proper ancestor* of v and v is a *proper descendant* of u. We define *successors, descendents,* and *ancestors* for *edges* similarly. Back edges have no successors. Define *out(v)* as the set of edges, in which each edge e is incident on v such that *tail(e)* is v. For any vertex v, the subgraph, $G' = (V', E')$ where V' and E' are the vertices and edges incident on $out(v)$ and the descendants of $out(v)$, are collectively called as the *segment(v)* which is said to be rooted at v. We say that a vertex v' is reachable from the $segment(v)$ if v' is a successor to v in segment(v).

Define $Low(v)$ for any vertex v as the lowest $DFN(u)$ of any vertex u that is reachable from the *segment*(v). In other words, for any vertex v, define

$$Low(v) = minimum(\{ DFN(v) \} \cup \{ Low(s) \ni s = son(v) \} \cup \{ DFN(w) \ni (v,w) \in B \})$$

Previous distributed algorithms for *DBT* make use of the following properties: If a vertex v is is an articulation point then v is either the root with at least two sons or v is not the root and it has a son s such that $Low(s) \geq DFN(v)$. A tree edge $e = [u, v]$ is a bridge if and only if $Low(v) = DFN(v)$.

However, our algorithm is based on the observation that in a distributed environment computing $Low(v)$ which needs explicit values of DFN of neighbors is an unnecessary restriction for obtaining the articulation points. Instead, we use the following Lemma 2.1 which only requires determining whether there exists any back edge from a descendant of a vertex v to any of the proper ancestors of v. We later show that it is indeed possible to do so in a distributed fashion using only linear number of messages.

We call a back edge $e' = [a, b]$ in *segment(v)* as an *attachment*(v), and further if $DFN(b) < DFN(v)$ then we call e' as a *proper attachment(v)*. Let the set of all e'', where e'' is a *proper attachment(v)*, be called a *patts*(v).

In Figure 4.1, *segment*(d) = $G' = (V', E')$ where $V' = \{d, h, g, a, j, e, f\}$

and $E' = \{ [d,h], [h,g], [g,d], [d,e], [d,a], [a,j], [j,f] \}$. Also, *patts*$(d) = \{ [d,e], [j,f] \}$

Lemma 2.1 : For any vertex v that is not the root, v is an articulation point if and only if no *proper attachment*(v) is in *patts*(s) for some son s of v. If v is the root, then v is an articulation point if and only if v has more than one son.

Lemma 2.2 : For any vertex v, the edge $e = (v,s) \in T$ is a bridge if and only if *patts*(s) empty.

The proof of Lemmas 2.1 and 2.2 may be followed easily from [1].

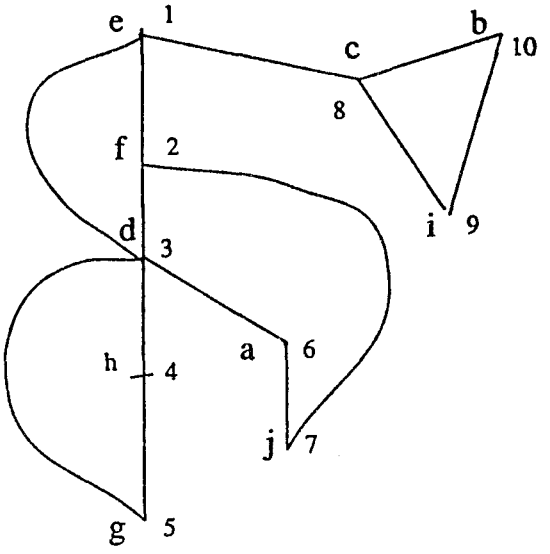

Figure 4.1. Illustration of Terms Used

4.3 The Model of Computation

We use a standard asynchronous model which is closely consistent with [6,11]. The message communication is *asynchronous* to the effect that a sender process i does not have to synchronize with the receiver process j before i can send a message to j. However, the distributed control synchronizes process interactions. Consider a network of n processes with bidirectional communication channels among some of the processes. Let $G = (V_0, E_0)$ be the undirected graph which is topologically isomorphic to the above network. A processor knows its distinct identity (ID) as well as the IDs of its neighbors in the graph. No two processors share memory, they communicate by message passing only. Computation should start when a designated node receives an outside signal and should terminate within finite time. The computation should terminate gracefully, and at that time, each process should possess the required result, should know locally that it has completed its part in the computation, and there should be no messages in transit or waiting in some input port.

We assume that the computation time is small compared to the message transmission time and thus may be neglected. Each processor has sufficiently large memory, thus memory buffering will not cause any problems. Arbitrarily long delays may

be encountered in processing a message at a node, however, no message is lost, communication is error free, and messages are not necessarily handled in FIFO order.

The *communication complexity* is the total number of messages used during the entire execution of the algorithm. *Time complexity* is the maximum time spent from the start to the end of the algorithm assuming that the time taken for delivering a message is at most one unit.

Our model is developed so that it is not necessary for each node to know the total number of nodes in the network. Instead, each node should know the details about the ID of its neighbors with whom it has links for communication. Whenever a node learns the existence of a node with higher ID than it previously had knowledge about, it takes into account the new information and will send that in all future communication with its neighbors. Hence it is easy to observe that the root node will obtain the value of the maximum ID, when the computation is terminated, that may be broadcast to all other nodes if it is necessary.

4.4 The Proposed Algorithm

We take the approach that distributed algorithm development is equivalent to providing specification of the desired correctness properties of a concurrent program in which processes communicate by message passing. In this paper, we start with a specification of the correctness properties of the *DBT* problem and proceed backward in a series of steps to a complete solution and further show that the solution is correct.

Similar to the well established technique, in sequential algorithm development, of using DFS for finding biconnected components, we use the DDFS algorithm developed in [6] to solve the *DBT* problem. The computation proceeds as in the *DDFS* algorithm using a scheme similar to Probe/Echo algorithms [10,11]. Each process carries out local computations as well as interacts with its neighbors to solve

the problem of *DBT*. During the probe-phase, each process collects the information about the attachments incident on the vertex, then proceeds to collect information from its descendents, and during the echo-phase it communicates the result of local computation to its parent. Our approach is motivated from the results of Reif [14] that the DFS problem is inherently sequential, and similar to the approach in [6], we try to minimize the number of messages by eliminating any parallelism.

$$
\begin{aligned}
&DBT:\ (\forall\, i,j : 1 \le i \le n, 1 \le j \le n : \\
&\quad (\, patts_i[j] \Leftrightarrow (\exists\, k : descendent_i[k])\,([k, j] \in B \wedge DFN(j) < DFN(i))\,) \wedge \\
&\quad (artq\_pt_i \Leftrightarrow (i \neq root \wedge \exists\, j \in son_i \ni \forall\,([k, m] \in patts_j) \wedge [k, m] \in B)\ DFN(m) \le DFN(i))\,) \vee \\
&\qquad (i = root \wedge \text{cardinality}(\, son_i\,) > 1\,)\,) \wedge \\
&\quad (\, bridge_i\,[j] \Leftrightarrow (\, son_i\,[j] \wedge patts_j = \varnothing\,) \vee (\, father_i\,[j] \wedge patts_i = \varnothing\,) \wedge \\
&\quad (bicon\_ptr_i\,[j] \Leftrightarrow (\exists\, k : 1 \le k \le n : bicon\_compt_k \neq \varnothing)\, i \in bicon\_compt_k \wedge j \in bicon\_compt_k\,))
\end{aligned}
$$

Figure 4.2. Specification of Correctness of Biconnectivity Testing

4.4.1 Program notation

We follow closely the programming conventions in [11]. A program consists of the global declarations followed by the process declaration. A process is declared with a capitalized name followed by a double colon, and then the local variable declaration, and statements. Each line of comment is preceded by a sharp character (#), and assertions are enclosed between braces.

A process communicates over a shared channel, which is an abstraction of a physical communication network to provide communication path between processes. Channels are accessed using **send** and **receive** primitives. Since we use asynchronous message passing model, we use different channels for different kinds of messages. Each type of channel is declared globally to share them globally.

In our model, processes communicate with known neighbors only. Hence we declare those bi-directional channels which are locally accessible in *links* array. A direct path is provided between two declared processes if the corresponding bit in the links array is set respecting the neighbor relation.

A channel is declared as **chan** *ch-name* ($f_1{:}t_1, f_2{:}t_2, \ldots, f_n{:}t_n$), where each optional data field in the message is described using field name f_i corresponding to field type t_i. The accessing primitives are **send** *ch-name*($expr_1, expr_2, \ldots, expr_n$) which is non-blocking and **receive** *ch-name*($var_1, var_2, \ldots, var_n$) which is blocking.

Statements, including guarded commands, have the well known syntax and semantics.

4.4.2 Description of constants and variables

The constants declared are self-explanatory. We describe only those variables and their meanings that may not be clear from the context easily :

chan *probe_echo*[1:*n*](*kind,sender*; *bit_len :* **int**, *bit_array*[1:*bit_len*] : **bool**)

is a global variable that is used for sending messages of the type START, DISCOVER, and RETURN. The messages sent are queued at the specific process addressed with an index in the range of 1 and *n*. Since we use variable length array in the message, bit-len is used to extend variable array length before the variable is read in the receiving process. We assume that *n* is not known a priori. Each process knows the maximum of the IDs of its neighbors and of itself.

chan *bicmp_echo*[1:*n*] (*biptr* : **int**)

is used to send the biconnected component pointers. Since artq_ptr may be shared in more than one biconnected component, we use the first son of the articulation point as the *biptr*.

chan *final_echo* (ECHO)

is used by the root node to inform the initiator process that its computation is terminated.

s is the set of sons of a vertex v in the tree of a particular execution of Depth First Search.

lower is the set of ancestors which are visited prior to a vertex v. The value of this set is not changed after it is read in.

patts[i] is the set of proper attachments, it is initialized to ϕ, updated first when control arrives to the node by checking if any of its neighbors are already visited even before the control arrived at v the first time. It is also updated every time control returns from one of its sons.

artq_set contains the value of the biptr in which an articulation pointer is a member of a biconnected component.

links[1:m] represents the knowledge of neighbors. The process gets the capability to access only those channels whose index has a corresponding value of TRUE in links.

bridge_set is the set of vertices representing bridges incident on the node p.

bcflag is an auxiliary variable which is set to true if *patts*[i] is not empty for a son i meaning that p is not an articulation point.

visited is a variable length boolean array carrying the knowledge of a process about the nodes previously visited at a given instant of computation. It is updated first time when the control arrives at a node from its ancestor, each node updates its own bit, and later using the comb_vector array when the control comes back from its sons.

Table 4.2 Illustration of Terms derived for the graph in Figure 4.1.

DFN	*node*	*ID*	*patts, q*	*comb_vector*	*artq_set*	*bridge_set*	*biptr_set*
1	e	5	∅, { 5,6,3 }	{ 5 }	{ 3 }	{ 3 }	{ 6 }
2	f	6	{ 5 }, { 6, 4 }	{ 5 }	∅	∅	{ 6 }
3	d	4	{ 5, 6 }, { 4,8,1 }	{ 4,5,6 }	{ 8 }	∅	{ 6 }
4	h	8	{ 4 }, { 8,7 }	{ 4 }	∅	∅	{ 4 }
5	g	7	{ 4 }, { 7 }	{ 4 }	∅	∅	{ 4 }
6	a	1	{ 6 }, { 1,10 }	{ 6 }	∅	∅	{ 6 }
7	j	10	{ 6 }, { 10 }	{ 6 }	∅	∅	{ 6 }
8	c	3	∅, { 3,9 }	{ 3 }	{ 9 }	{ 5 }	{ 9 }
9	i	9	{ 3 }, { 9,2 }	{ 3 }	∅	∅	{ 9 }
10	b	2	{ 3 }, { 2 }	{ 3 }	∅	∅	{ 9 }

comb_vector is also a variable length boolean array. Some of its bits corresponding to the indices of vertices in *lower* array are reset to FALSE to indicate that there exists some proper attachments from some of its son. It also carries information about the newly visited nodes in segment(v).

Remaining variables are used so that their meaning is self-explanatory in the context of use.

We have used italic fonts for the variables. Some functions are used with their name represented in Roman fonts, such as 'max' for the maximum, etc., and we assume that their meaning is self explanatory. We have assumed that functions are in-built to extend the variable length arrays whenever the index used is greater than the currently known values. These additional bits extended are assumed to be initialized to FALSE.

4.4.3 Specification of correctness properties

We have pursued programming as a goal oriented activity so that the following predicate is true when the program terminates. Our specification relates the output variables with the input variables. Figure 4.2 provides the specification of correctness for distributed biconnectivity problem. Input variables in our specification may be recursively specified for correctness. We have not provided specification for terms such as *descendent* since they are well established in literature.

For notational convenience, we associate a predicate with every variable *var* in process *i*. For example, $descendent_i$ [*j*] implies that the process *i* has a variable *descendent* with value *j*. Also note that we assume that the predicates for variables *var* of type *set* are accessed as an array so that the index in the array is TRUE if the element with value of the index is in the *var*.

For illustration, when the program *DBT* terminates, for a particular execution of *DDFS* shown on the graph given in Figure 4.1, it may be verified that the desired values of the local variables are as given in Table 4.2. The entries in the table corresponding to array variables refer to the indices of interest of *ID*s generated in the lexicographic ordering of nodes. Note that $patts_{,s}$ is shown as the union of the $patts[p][q]$, $\forall\, q \in s \cup p$, for convenience of display.

4.4.4 Formal Description of the Algorithm

We present formal description of the proposed algorithm in Figure 4.3, which is executed at each node p. The algorithm is described in a language whose syntax and semantics follow closely those given in [11].

4.5 Complexity Analysis and Proof of Correctness.

We presented an algorithm for *DBT* problem in Figure 4.3, and in this section we prove that time and message complexities of the algorithm are both $O(n)$. We also

show that the algorithm is optimal in message complexity and further demonstrate that *DBT* has a message complexity lower bound of $O(n)$. We also establish that the order of communication bit complexity of our algorithm is less than previously reported algorithms whenever $m = O(n^2)$ as in complete graphs.

Theorem 5.1. For a given connected graph, G, the algorithm for *DBT* problem given in Figure 4.3 computes *patts*(v) for every vertex v in G, and marks articulation points, bridges and biconnected component pointer correctly.

Proof: The algorithm given in Figure 4.3. is a straight forward implementation of the specification *DBT*. It may, however, be noted that the visited information about the ancestors of a node is correctly restored using the *lower* array for those bits that are reset in the *comb_vector* whenever the *patts* is non-empty for a descendent of any vertex. Also, note that the messages are routed only through the tree edges and further that a node does not send *bicmp_echo* messages on the bridges. Correctness is easy to follow otherwise. □

Theorem 5.2. *DBT* has the message complexity lower bound of $O(n)$ for n > 1.

Proof Since information that is not local but is of global topological interest have to be exchanged using communication alone, it is clear that at least one message must be sent from any node i to any node j in the network, $i, j \in V_0$. Hence at least $n-1$ messages must be sent over the links so as to communicate to all the n nodes at least once. Hence *DBT* has an O(n) lower bound in message complexity. □

Theorem 5.3 The proposed algorithm is optimal in communication complexity and uses less than $3n-3$ messages.

Proof Every node in the network except the root node receives message of type *probe_echo* once with kind DISCOVER message from its father (forward path) and sends back RETURN kind message to its father (return path) . Also it is clear from the algorithm that DISCOVER kind message is not sent to an already visited node in

```
const source = h ; TRUE = 1 ; FALSE = 0
      # 'h' is the index of the node that initiates the algorithm
type kind = enum(START,DISCOVER,RETURN,ECHO)
chan probe_echo[1:n](kind,sender, bit_len : int, bit_array[1:bit_len] : bool)
chan bicmp_echo[1:n] ( bicon_ptr : int )
chan final_echo ( ECHO )

Node[p:1..n]:: var k : kind, father, bi_ptr : int
               var links[1:m], bcflag = FALSE : bool
                     # m is equal to the maximum of all the ID of the neighbors of p
                     # links[q] is initialized to TRUE if q is a neighbor of Node[p];
               var s = lower = patts[1 : r] = artq_set = bridge_set = biptr_set = ∅ : set of int
                     # r is equal to the maximum of ( p , m )
               var visited[1:*], comb_vector[1:*] : bool
               receive probe_echo[p] (k,f,len, visited[1:len])
               if k = START → father := 0
               [] k = DISCOVER → father := f fi
                     # father of p is zero if p = source
                indx := max (len,m, p)
               fa i := 1 to indx st visited[i] → lower := lower ∪ { i }
                     if links[i] ∧ i ≠ father → patts[p] := patts[p] ∪ { i } fi af
               visited[p] := TRUE
               fa i := 1 to m st links[i] ∧ ¬visited[i] → s := s ∪ { i }
                     send probe_echo[i] ( DISCOVER,p, indx, visited[1:indx] )
                     receive probe_echo[p](k, j, r, comb_vector[1:r] )
                           { for a given execution of DDFS, k = RETURN and j = i }
                     indx := max(r,indx)
                     fa q := 1 to indx st ( q ∈ lower ) ∧ ¬comb_vector[q] → patts[i] := patts[i] ∪ q
                           bcflag := TRUE af
                     if p ≠ source ∧ ¬bcflag → artq_set := artq_set ∪ { i } fi
                     if ¬bcflag ∧ comb_vector[p] → bridge_set := bridge_set ∪ { i }
                     [] ¬bcflag ∧ ¬comb_vector[p] → biptr_set := biptr_set ∪ {i}
                           send bc_echo[i](i) fi
                     bcflag := FALSE
                     visited[1:indx] := visited[1:indx] ∨ comb_vector[1:indx] af
               if p = source → ns := size ( s )
                     if ns > 1 → artq_set := { s } fi
                     send final_echo (ECHO)
               [] p ≠ source → comb_vector[1:indx] := visited[1:indx]
                     fa i := 1 to indx st i ∈ patts[p] → comb_vector[i] := FALSE
                           np := np + 1 af
                     fa i := 1 to m st links[i] ∧ patts[i] ≠ ∅ →
                           fa j := 1 to indx st j ∈ patts[i] → comb_vector[j] := FALSE
                                 np := np + 1 af af
                     send probe_echo[father](RETURN,p, indx, comb_vector[1:indx])
                     if np = 0 -> bridge_set := bridge_set ∪ { father }
                     [] np > 1 → receive bicmp_echo[father](biptr)
                           biptr_set := biptr_set ∪ {biptr}
                           fa i := 1 to m st i ∈ s ∧ i ∉ artq_set ∧ i ∉ bridge_set → send bicomp_echo[i] (biptr) af
                     fi
               fi

Initiator::    send probe_echo[source](START,Initiator_ID,0,φ);
               receive final_echo (ECHO);
```

Figure 4.3. Algorithm for *DBT* Problem

the network. Thus, each of the $n-1$ nodes (excluding the root) exchanges *probe_echo* messages with its father exactly twice. Further, the information about biconnected component pointer is sent from the articulation point to its descendents

in a biconnected component using *bicmp_echo* type message on the tree edges alone. Since *bicmp_echo* message is not sent over a tree edge if it is a bridge, total number of messages sent of type *bicmp_echo* is less than $n-1$. Hence the total number of messages used in the algorithm *DBT* is less than $3n-3$. Hence, the message complexity of the algorithm is O(n) and is optimal within a constant. □

Theorem 5.4 The algorithm terminates after less than $3n-3$ units of time, if all messages are delivered in one unit of time, and at that time the root node will have the knowledge about the largest ID of the nodes in the network.

Proof: Total time needed for the algorithm to construct DDFS is the time required to transmit the messages over the links. From theorem 5.3 , total messages needed to transmit is less than $3n-3$. Hence the total time needed is also less than $3n-3$ units if all messages are delivered in at most one unit of time. It may be easily observed that each node extends the *visited* array any time it learns the existence of a higher ID than it is aware of. Thus the cardinality of the *visited* array in the root node when the algorithm terminates provides the largest ID of any node in the network. □

Lemma 5.4 The total number of bits used for communication in the proposed algorithm is less than that for the previous algorithms for a fully connected network.

Proof: In our algorithm, the number of bits transmitted for message is at most $O(n)$ bits due to the bit array that is appended to the messages. Since the message complexity is $O(n)$, total number of bits used is $O(n^2)$. However, all previous algorithms use $O(m)$ messages with $O(logn)$ bits each. Hence, for a fully connected network, total number of bits used for communication is $O(n^2 \log n)$ since $m = O(n^2)$ for fully connected graphs. Hence the total number of bits used for communication is reduced by $O(logn)$ in the proposed algorithm compared to the previously known algorithms and hence the proof. □

6.0 Conclusions

We have presented a simple to understand *DBT* algorithm that is optimal in both time and message. Taking into account the result [14], that *DFS* is inherently sequential, we have pursued minimizing the number of messages used to solve *DBT* by eliminating any parallelism. The algorithm for *DBT* is shown to use less than $3n-3$ messages with the time complexity of less than $3n-3$ units. We have shown that the extended message format reduces overall message complexity in terms of total number of bits used for communication under the assumed model whenever $m = O(n^2)$.

It is interesting to see that the proposed solution is valid for both synchronous and asynchronous communication models. We will explore further the possibilities of extending our results to reliable communication models.

CHAPTER 5

DISTRIBUTED PLANARITY TESTING

You have to make judgements; you can not wait until all evidence is in.

-- Hubert Humphrey

5.1. Overview

The problem of determining whether a given graph G is planar or not is well known in the areas of algorithm design and graph theory [1,2]. Testing planarity is of both theoretical and of practical importance since planar graphs offer efficient solutions to many problems, such as graph colorings, which are in general very difficult for arbitrary graphs. Knowledge of planarity has also been used recently to design efficient distributed algorithms for finding shortest paths in a graph [3,4].

Distributed Planarity Testing (*DPT*) is a problem to devise a distributed algorithm that provides a local code for each process to be executed so as to find out whether a graph G, corresponding to an underlying communication network of a distributed system, is planar or not. Given that all computations at any node have to be based only on the data available at the local site, the data from other nodes which are of global interest must be obtained by communicating through neighbors alone. The most important concern in such a distributed computation is to solve the *DPT* problem using a minimum number of messages to route any non-local data when necessary. Assuming that the time taken by individual nodes for computation is negligible, we investigate the message complexity of the *DPT* problem. Computations should terminate in finite time and at the end of the computations the designated root node should correctly know whether the graph is planar or not.

We take the approach of designing a distributed algorithm by adapting a sequential algorithm to a distributed environment. We specifically use the technique of Probe/Echo algorithms[5,6]. We give an implementation of a path based embedding algorithm of Williamson[7] which itself is a refinement of Hopcroft and Tarjan's algorithm[8]. We show that $O(n)$ messages are sufficient for *DPT*. We also reduce the message size to $O(logn)$ instead of $O(n \log n)$ which is required for the stacks used in the sequential algorithms of [7,8].

If each node has knowledge about its neighbors, the graph planarity problem could be solved by routing all information to a particular node using the *flooding technique* given in[9] and then applying any of the well-known efficient sequential algorithms for planarity testing[8,10,7]. However, this flooding approach could use $O(mn)$ messages, where n is the number of nodes, and where the number of edges, m, could be $O(n^2)$. In this paper, we study the problem of *DPT* and present a distributed planarity testing algorithm that uses $O(n)$ message and $O(n)$ time, assuming that the measure of time will correspond to the time used by the algorithm if every message transmission is completed in unit time.

The usefulness of a depth first search(DFS) paradigm to determine biconnectivity as well as to determine planarity is well established in sequential computing [1]. Algorithms for a distributed depth-first-search (DDFS) for graphs representing communication networks may be found in [11,12,13,14]. The most efficient algorithm known for DDFS uses exactly $2n$-2 messages and terminates after $2n$-2 units of time[14]. Despite an efficient $O(n)$ message and time complexity distributed algorithm for DDFS, currently no distributed algorithm is known in the literature for determining whether an underlying graph is planar or not. We present the first distributed algorithm which efficiently determines, using linear number of messages, whether an underlying graph is planar.

Our algorithm tests planarity by testing each of the bicomponents of a graph. There has been a considerable amount of research on distributed biconnectivity testing and its various characterizations [15,16]. The most efficient algorithms known for the distributed recognition of biconnected components uses $O(m)$ messages and $O(d)$ time with a bound of $O(logn)$ on message size[15]. We have recently taken the approach of designing a distributed algorithm for biconnectivity by adapting a depth first search based sequential algorithm to a distributed environment, using the techniques in [5,14,6]. We provided an algorithm for distributed biconnectivity testing with linear complexity in both the number of messages and the time, and with a bound of $O(n)$ on message length [17].

Though many well-known results of optimal $O(n)$ time sequential algorithms [8,10,18,7,19] as well as an optimal $O(logn)$ time parallel algorithm [20] for planarity testing are available, surprisingly there is no distributed algorithm available for planarity testing. The problem of efficient distributed planarity testing was left as an open problem in [21]. The author of [21] conjectured at that time that a distributed algorithm for Planarity Testing, assuming that a minimum spanning tree is available, using $O(n)$ messages, $O(d)$ time, and $O(logn)$ message size may not exist. (Here n is the number of nodes, m is the number of edges, and d is the diameter of the graph corresponding to the underlying asynchronous network.) We use a *DDFS* algorithm[14] to construct a spanning tree using $O(n)$ messages and $O(n)$ time, and using $O(n)$ message size. Furthermore, based on this tree constructed by the *DDFS* algorithm, we provide a distributed algorithm for planarity testing using an additional $O(n)$ messages with $O(logn)$ message size.

In Table 5.1, we summarize the performance of the algorithm for *DPT* and related problems in sequential, parallel, and distributed environments. We again note that the message size in our distributed algorithms is $O(n)$ bits for the initial phase to find out whether $m \leq 3n-6$.

Table 5.1 Comparison of Related Works

Author	Year	Type	Problem	Time Complexity	Comm. Complexity
Hopcroft & Tarjan	1974	Sequential	Planarity	$O(n)$	-
Cai, Han, & Tarjan	1992	Sequential	Max. Planarity	$O(n)$	-
Ramachandran & Reif	1989	Parallel	Planarity	$O(logn)$	-
E.J.H. Chang	1982	distributed	*DBT*	$O(d)$	$O(m)$
Huang et. al.	1989	distributed	*DBT*	$O(d)$	$O(m)$
Sharma & Iyengar	1989	distributed	*DDFS*	$O(n)$	$O(n)$
Hegde & Iyengar	1992	distributed	*DBT*	$O(n)$	$O(n)$
Proposed here	1992	distributed	*DPT.*	$O(n)$	$O(n)$

5.2 Preliminaries

In this section we present the terminology and notation used in this paper. We refer the reader to [1,22,23,24] for any additional graph theoretic terminologies, and to [2,10,7] for details about planarity related concepts and terminologies.

A depth-first-search (*DFS*) on an undirected graph $G_0 = (V_0, E_0)$ gives a directed graph $G = (V, T, B)$ where the vertices in V are those of V_0 labeled with depth-first-number $DFN(v), v \in V_0$, and the edges in E_0 are divided into two mutually exclusive sets of directed edges: Tree edges, T and Backedges, B. We refer to the union of T and B as E. Let $n = |V|$, and $m = |E|$. If $e = [u, v] \in E$, then u is called the *parent* of v and v is a son of u. If there is a *path* from u to v, then u is an *ancestor* of v and v is a *descendant* of u, and further, if $u \neq v$ then u is a *proper ancestor* of v and v is a *proper descendant* of u. We define *successors, descendents,* and *ancestors* for edges similarly. Back edges have no successors.

We define $Ret(e)$ to be the set of vertices such that for some s, $[s,t]$ is a backedge in G and also a descendent of e. If $Ret(e)$ is not empty, we define $low_1(e)$ to be the smallest integer in $Ret(e)$, and $low_2(e)$ to be the second smallest integer in $Ret(e)$. Otherwise, we set $low_1(e) = low_2(e) = n + 1$.

For every edge $e = [u, v] \varepsilon E$, define the function ϕ as follows.

$$\phi(e) = \begin{cases} 2low_1(e) & \text{if } low_2(e) \geq u \\ 2low_1(e) + 1 & otherwise \end{cases}$$

The function ϕ arranges the successors of each edge incident out of a node. An adjacency list of a vertex v arranged in the increasing order of ϕ is called a *properly arranged adjacency list of* v. We denote this list by $PAL(v)$.

For an edge $e = [u_1, u_2]$ we define $path(e) = u_1 u_2 \ldots u_k$ where u_{i+1} is a successor of u_i in $PAL(u_i)$ and where $[u_{k-1}, u_k] \in B$. We denote the set of vertices $\{ u_1, u_2, \ldots u_k \}$ by $PLIST(e)$. If we add the unique sequence of tree edges from u_k to u_1, we obtain $Cycle(e)$.

We define a directed ordered rooted tree, $PATR(G, T)$, as follows. Each vertex in $PATR(G, T)$ corresponds to a path in G. The root of $PATR(G, T)$ corresponds to the path $\{ u_1, u_2, \ldots u_k \}$ in G starting at the root of the *DFS* tree, and where u_i is the first successor of u_{i-1} in $PAL(u_{i-1})$. For each additional vertex v in $PATR(G, T)$ corresponding to the path $\{ u_1, u_2, \ldots u_k \}$ in G, each successor of v corresponds to $path(e_i)$, where $e_i = (u_i, w_i)$ with $w_i \in PAL(u_i)$ and $w_i \notin PLIST(v)$.

The *segment* of e, $Seg(e)$, is the subgraph of G obtained from $path(e)$ and all of its descendents in $PATR(G, T)$. The *segment list of* e, $SEGLIST(e)$, is the ordered set $Seg(f_i)$, $1 \le i \le k$, where f_i is a son of $Path(e)$ in $PATR(G, T)$. We will often denote $Seg(e)$ by $\hat{e}$.

If $e = [u, v]$, then $Tail(e) = u$ and $Head(e) = v$. We define $Range(e) = Ret(e) \cup Tail(e)$. The $Span(e)$ is the restriction of $Cycle(e)$ starting from $low_1(e)$ to $Tail(e)$, and the *open span* of $Seg(e)$, $OSPAN(e)$, is $Span(e) - \{ low_1(e), Tail(e) \}$. If $Range(e) = v_1 < v_2 < \ldots < v_r$, then the directed path in $Cycle(e)$ from v_i to v_{i+1} are the *proper gaps* of $\hat{e}$. The directed path from v_r to v_1 in $Cycle(e)$ is the $cospan(e)$.

If two segments A and B must be embedded on opposite sides of $Cycle(e)$ in any planar embedding of G, then A and B are said to be *directly linked.*

Define $SEGLIST(e, X)$ to be the set of segments in $SEGLIST(e)$ that are less than or equal to X in the linear order in $PATR(G, T)$. The graph $SEGGR(e, X)$ is a graph whose vertex set corresponds to the set $SEGLIST(e, X)$. Two vertices in $SEGGR(e, X)$ are

connected by an edge if they are directly linked in G. We denote a connected component of $SEGGR(e, X)$ by C.

An $\hat{e} \in C$ is *internal* if there is an edge $(s,t) \in \hat{e}$ such that $low_1(\hat{e}) < t < Tail(\hat{e})$. The status of a component C relative to $\hat{e}$, $Status(C, \hat{e})$, is internal if there is an $\hat{e} \in C$ such that $\hat{e}$ is internal. We set $g^-(C,\hat{e})$ = Minimum ($Ret(C, \hat{e})$) and $g^+(C, \hat{e})$ = Maximum ($Ret(C, \hat{e})$). The set of edges (s,t) such that $t \in Ret(C, \hat{e})$ and (s,t) is embedded on the inside of Cycle(e) is denoted by $I(C, \hat{e})$ and the set of edges embedded on the outside of Cycle(e) is denoted by $O(C, \hat{e})$. We will refer to the values ($g^+(C, \hat{e})$, $g^-(C, \hat{e})$, $I(C, \hat{e})$, $O(C, \hat{e})$, , $Status(C, \hat{e})$) as the *component parameters of C* relative to $\hat{e}$.

5.3 The Model of Computation

We use the standard asynchronous model which is closely consistent with [14,6]. The message communication is *asynchronous* to the effect that a sender process i does not have to synchronize with the receiver process j before i can send a message to j. However, the distributed control synchronizes process interactions. Consider a network of n processes with bidirectional communication channels among some of the processes. Let $G = (V_0, E_0)$ be the undirected graph which is topologically isomorphic to the above network. A processor knows its distinct identity(ID) as well as the IDs of its neighbors in the graph. No two processors share memory, they communicate by message passing only. Computation starts when a designated root node receives an outside signal and should terminate within finite time. The computation should terminate gracefully, and at that time, the root process should possess the required result, each node should know locally that it has completed its part in the computation, and there should be no messages in transit or waiting in some input port.

We assume that the computation time is small compared to the message transmission time and thus may be neglected. Each processor has sufficiently large memory, thus memory buffering will not cause any problems. Arbitrarily long delays may be encountered in processing a message by a node, however, no message is lost, communication is error free, and messages are not necessarily handled in FIFO order.

The *communication complexity* is the total number of messages used during the entire execution of the algorithm. *Time complexity* is the maximum time spent from the start to the end of the algorithm assuming that the time taken for delivering a message is at most one unit.

Our model is developed so that it is not necessary for each node to know the total number of nodes in the network. Instead, each node should know the details about the ID of its neighbors with whom it has links for communication. Whenever a node learns the existence of a node with higher ID than it previously had knowledge about, it takes into account the new information and will send that in all future communication with its neighbors. Hence it is easy to observe that the root node will obtain the value of the maximum ID when the computation is terminated, which may then be broadcast to all other nodes if it is necessary.

5.4 Program Notations and Conventions

We take the well established approach of distributed algorithm development starting from an efficient sequential algorithm for the given graph problem and determine a distribution of local computations to minimize communication. The computation proceeds as in the *DDFS* algorithm using a scheme similar to Probe/Echo algorithms[5,6]. Each process carries out local computations as well as interacts with its neighbors to solve the problem of interest. During each probe-phase, a node collects the information from the nodes incident out of the node and

from its descendents, and during the echo-phase it communicates the result of local computation to its parent. Our approach is motivated from the result of Reif[25] that the *DFS* problem is inherently sequential, and similar to the approach in [14], we try to minimize the number of messages even by reducing parallelism if necessary.

We present a distributed algorithm which closely follows the programming conventions in [6]. A program consists of the global declarations followed by the process declaration. A process communicates over a shared channel, which is an abstraction of a physical communication network, to provide a communication path between processes. Channels are accessed using **send** and **receive** primitives. Since we use an asynchronous message passing model, we use different channels for different kinds of messages. Each type of channel is declared globally so that it may be shared by processes.

In our model, nodes communicate with known neighbors only. We declare those bi-directional channels which are locally accessible in *links* array. A direct path is provided between two declared processes if the corresponding bit in the links array is set respecting the neighbor relation. We describe only those variables and functions whose meanings may not be clear from the context. We have assumed that functions are in built to extend the variable length arrays whenever the index used is greater than the currently known values. These additional bits extended are assumed to be initialized to FALSE.

5.5 Description of the Sequential Algorithm

We present the first distributed algorithm for determining whether or not a given graph is planar. Our algorithm is a distributed implementation of a sequential algorithm for planarity testing given by Williamson [7]. We show that $O(n)$ messages are sufficient for distributed planarity testing. We also show that a message size of $O(logn)$ is sufficient for the distributed implementation of the stacks needed in the

algorithm of [7]. Our distributed algorithm presents not only an efficient routing of local information to all nodes in the network but also a different ordering and delaying of certain computations required in [7]. This provides a distributed algorithm with the total number of messages limited to $O(n)$ with proposed $O(logn)$ bound on message size.

The sequential algorithm given by Williamson [7] is a refinement of an algorithm given by Hopcroft and Tarjan [8]. In Williamson's algorithm, the inherent tree structure of the cycles in a graph G, generated during the recursive computations performed in [8], is exploited to examine the bi-colorability of the segment graphs of this tree. It is shown in [7] that determining bi-colorability is equivalent to planarity testing. In what follows, we briefly outline the necessary steps of [7].

Begin by traversing $PATR(G, T)$ in depth first order. At each vertex path(e), assume that a post-order traversal has been carried out for each $X \in SEGLIST(e)$ and, subsequently, the list $Ret(X)$ is available. We assume a bi-colored spanning forest for $SEGGR(e, X)$ is constructed by processing each $X \in SEGLIST(e)$ in the linear order of $PATR(G, T)$. For each component C of this spanning forest, we maintain the *component parameters* ($g^+(C, e)$, $g^-(C, e)$, $I(C, e)$, $O(C, e)$, , $Status(C, e)$).

Let Y be the next segment in $SEGLIST(e)$. We begin processing Y by deleting those values t in $Ret(X)$ with $t > Tail(Y)$. The components C_i of $SEGGR(e, X)$ with $low_1(Y) < g^-(C_i)$ are called the *critical components* of $SEGGR(e, X)$ relative to Y, denoted $CRITCOM(e, X, Y)$. In order to determine the relation between Y and the $Ret(C_i, X)$ of each component C_i in $SEGGR(e, X)$, we define four sets of attachments which we denote by A, B, A', B'. Let A' equal the return set of $CRITCOM(e, X, Y)$. Let $B' = Ret(Y)$. For a component C_i for which $g^-(C_i, X) < low_1(Y) < g^+(C_i, X)$, let $A = Ret(I(C_i, X))$ and $B = Ret(O(C_i, X))$.

The following lemma is one of the main basis for testing planarity in the algorithm of [7]:

Lemma 4.1: If $\exists$ $f_i, f_j \in SEGLIST(e)$ s.t. $i < j$ and f_i, f_j proper and $f_i \; dl \; f_j$ then G is non-planar.

Proof: $Cospan(e)$ is a directed path from $Tail(e)$ to $low_1(e)$. Since $f_i \; dl \; f_j$ and are proper, $Cospan(e)$ interlaces some proper attachment in f_i and some proper attachment in f_j. Hence the proof. □

The algorithm of [7] tests planarity by examining whether or not the critical components are one sided, whether or not A', B', A, and B may be concatenated satisfying specified conditions relative to a partial order, and whether or not the status of the components are collectively valid. For the details of these conditions, see Fig. 5 in [7]. If this examination is successful, Y is merged with components of $SEGGR(e, X)$ and is colored appropriately to get a spanning forest for $SEGGR(e, Y)$. If Y is the last entry in $SEGLIST(e)$, then $Seg(e)$ is obtained from $path(e)$ and $SEGLIST(e)$ after deleting those backedges (s,t) with $t > Tail(e)$.

5.6 Novel Distributed Algorithm

Let G be the underlying graph. If G is not biconnected, the biconnected components of G may be obtained using the distributed algorithm in [17]. The following distributed algorithm for planarity testing may be applied to each of the biconnected components of G. Since G is planar if and only if each of its bicomponents is planar, successful testing of each of the bicomponents is sufficient to determine the planarity of G.

We give a distributed implementation of the path tree based planar embedding algorithm outlined in section 4.0. We show that $O(n)$ messages are sufficient for distributed planarity testing. We achieve this by showing that the computations needed for embedding the segments of a path may be localized. Note that these localizations

are achieved by exploiting the following observations facilitating the distribution of the stack needed in the computation:

1) The Range of each segment in a path is linearly ordered.

2) The segments of a given path are also linearly ordered and the Range of any segment in a given path is restricted to the Range the path.

Hence, during the processing of a segment computations may be localized to the range of its paths. Furthermore, we show that we can limit the message size for planarity testing to $O(logn)$. We observe that a straight forward implementation of the algorithm in Section 4.0 requires a message size of $O(n \log n)$ because of the need to carry the return set of a component as a stack. We provide a distributed implementation of these stacks avoiding a need to explicitly carry entire stack parameters in the message. This requires reordering of certain computations because of the unavailability of all the distributed data at each node. We show in section 6.0 that these computations may be delayed until the data becomes available and that these delayed computations maintain the correctness of the algorithm in Section 4.0.

We execute the following three rounds of a probe/echo algorithm for our distributed algorithm:

Round 1

A DDFS is carried out with some additional local computations at each node to label the type of edges incident on the node. When the root node is prompted from an external source, the probe phase for this round is initialized. As in [14], a message is created by the root node with the first two fields to be used for the following: the first corresponds to a bit vector field to mark all the visited nodes and the second field is an ID-field for the sender information. The size of the bit vector is initialized as the largest of the ID's known to the root. The root marks its ID as visited in the bit-vector and then sends the control/message to an unvisited neighbor.

When the control is received for the first time, each node determines its parent from the sender information in the message. Corresponding to the remaining neighbors which are marked as visited in the bit vector, it determines the set of backedges incident on it. The node then updates the bit-vector by marking the bit corresponding to its own ID as visited. It stores this bit-vector in a local variable. It then labels an unvisited neighbor as a son and sends the control/message to this son. When the control/message is returned to the node and if an unvisited neighbor exist in the bit-vector received for the son, it repeatedly selects the next son and sends and receives the control/message until all its neighbors are marked as visited. When all the sons are labeled and the control is returned, it marks as IN-attachments those edges corresponding to the neighbors which are not sons and which are also listed as unvisited in the bit-vector stored earlier as a local variable. At this time, the node has partitioned the set of edges incident on it as the tree edge from its parent, the tree edges to its sons, the backedges to its ancestors, and the backedges from its descendents. The node then returns the control to its parent along with the updated bit-vector signaling the Echo phase for this round.

When the root receives the Echo signal from its last son and labels its IN-attachments, the first round of computations is complete.

We remark that in the Echo phase, information about the total number of edges E is obtained so that at the root the graph may be declared non-planar if $|E| > 3V - 6$.

Round 2

If $|E| \leq 3V - 6$, then the root initializes the Probe phase for the second round. This round is used to calculate low_1, low_2, ϕ, and PAL values for each node.

In the probe phase, each node sends its depth first number (*dfn*) to those nodes corresponding to the set of IN-attachments incident on it. The control is then concurrently passed to its sons. When the control reaches a leaf node of the DFS tree,

the echo phase for this round is initialized. At this time, each leaf has the *dfn* numbers of its neighbors. The leaf uses these numbers to calculate low_1, low_2, and $\phi-$, values for edges corresponding to its neighbors, and orders its neighbors to obtain its *PAL*.

At each internal node similar computations are performed. When the control is returned from all of its descendents, observe that the *dfn*'s corresponding to its backedges have already been received. When the control is returned to the root from all of its sons, the second round is complete.

Round 3

The root once again initializes the probe phase for this round. In the probe phase, the path tree $PATR(G,T)$ is constructed. In the echo phase, at each vertex $path(e)$ of $PATR(G,T)$, $Seg(e)$ is determined by successively merging $SEGLIST(e, X_i)$ in the linear order of X_i in $PATR(G,T)$.

The root starts the process of constructing the path tree $PATR(G,T)$. In what follows, we describe how the paths corresponding to the vertices of $PATR(G,T)$ are constructed, and how the vertices of $PATR(G,T)$ are linearly ordered relative to this tree structure.

Let $path(e) = v_1 v_2 \ldots v_k$ correspond to a vertex v in $PATR(G,T)$. Let $PAL(v_i) = \{ v_1^i, v_2^i, \ldots\ldots, v_r^i \}$. Then $v_{i+1} = v_1^i$. For those v_j^i corresponding to backedges, a segment $Seg(e_j)$ is constructed. For the remaining v_j^i, $j > 1$, corresponding to edges which are not backedges a Seg-Message is sent. When a vertex v_j^i receives a Seg-Message from v_i, it initiates a new path, $path(e_j)$, by constructing a Path-Message with $Tail(e_j) = v_i$ and $low_1(e_j) = low_1(v_j^i)$. The vertex v_i then sends *Path Message* to its successor v_{i+1} on the path and waits until it receives back the control from all $v_j^i \in PAL(v_i)$ which do not correspond to backedges. We observe that for the case when v is the root of the $PATR(G,T)$, v_1 is the root of the *DFS* tree in G. The probe phase of this round ends at the *last* vertex v_{k-1} of each path.

Beginning at the last vertex v_{k-1} of $path(e) = v_1 v_2 \ldots v_k$, the Echo phase is initialized. v_{k-1} maintains $low_1(e) = v_k$ as the bound for $Seg(e)$ to be sent on the attachment corresponding to $low_1(e_i)$, where e_i is the first segment with a proper attachment. Unlike the sequential algorithm in which $Seg(e)$ is obtained after processing $SEGLIST(e)$, we send this information before beginning any processing of $SEGLIST(e)$. This allows us to gain efficiency by avoiding a backward traversal from v_1 to v_k in order to merge $path(e)$.

The following is a recursive procedure executed at each node. We explain the steps involved in this procedure.

Step 1:

For each vertex v_i on path(e) with segments $\hat{e}_i$ consisting of a single backedge, $Ret(\hat{e}_i)$ is set equal to $Tail(e_i)$.

Step 2:

For those non-singular segments $\hat{e}_i$, $Ret(\hat{e}_i)$ is assumed to be available by the recursive procedure applied to $\hat{e}_i$). In this case, $Ret(e_i)$ is maintained as a distributed stack. This distribution is achieved by sending an element in the ordered set $Range(\hat{e}_i) - Tail(e)$ over the attachment corresponding to its successor element in this set.

Step 3:

For any $IN - attachment$ incident on v_i the information received is used to update the current component parameters received from v_{i+1}. This involves analyzing which of the following three cases apply.

case 1 : $g^-(C, X) < v_i < g^+(C, X)$ for some component C of $SEGGR(e, X)$. In this case, the information received corresponds to the next element (relative to v_i) in the Range of C.

case 2 : $v_i = g^-(C, X)$ and the set of accumulated attachments is empty. The current component parameters are set to those received over the $IN - attachment$.

case 3 : $v_i = g^{-}(C, X)$ and the set of accumulated attachments is non-empty. One of the following decisions is made to resolve relaxed calculations.

(i) The graph is non-planar because the set of accumulated attachments, *SAC*, is directly linked to segments in C embedded on both sides. This decision can be made based on component parameters received over the $IN-attachment$ and the $low_1(SAC)$ in the current component.

(ii) Merge the set of accumulated attachments with a segment for which the set of accumulated attachments is not directly linked.

(iii) Form a new component consisting of the set of accumulated attachments and also maintain the component parameters relating to the old pair received.

Step 4 :

If v_{k-1}, the last vertex of $path(e)$ has a segment $\hat{e}_1$ incident on it, it initializes the first component of $SEGGR(e, \hat{e}_1)$ using the component parameters due to $\hat{e}_1$. If v_{k-1} has no such segment, then the control is passed from v_{k-1} to v_i, where $\hat{e}_1$ is incident on v_i; v_i then performs this initialization.

Step 5 :

The value $low_1(e)$, corresponding to the lowest element in $Range(\hat{e})$, is sent over the attachment corresponding to $low_1(\hat{e}_i)$, where $\hat{e}_i$ is the first segment of $path(e)$ with *Status* internal.

Step 6

The segments $\hat{e}_i$, $i \geq 2$, incident on v_i, are merged in the linear order with the current component. This involves the following computations.

(i) To begin, the current component is either initialized due to $\hat{e}_1$ or is received from v_{i+1}. The current component parameters are updated as described in *Steps* 1 – 3.

(ii) Let $Y = \hat{e}_i$. From the component parameters of Y, check to see if the proper attachments due to Y are embedded on one side. Note that only the *IN – attachments* incident on v_i can be embedded on the opposite side of the proper attachments. If this is not the case, then the graph is non-planar. The proper attachments have been determined to be one-sided using the following recursive procedure in Y: $low_1(Y)$ is sent over the first proper attachment in Y. The subsequent proper attachments are used to update the bounds of the single-sided component. This is equivalent to maintaining the component as a distributed stack. The details of the required calculations follow easily from *Lemma*1. We remark that those attachments which are not proper are not necessarily one-sided. The calculations required to embed these attachments are described in what follows.

(iii) Let C = current component. If C is one-sided and $low_1(Y) < g^-(C, e_i)$, then it is necessary to relax computations needed to check that the critical components, $CRITCOM(e, e_i, Y)$, are one sided. These relaxed computations are required because only the information about the current component is available locally. We carry the information that computations are relaxed by setting a *RELAXFLAG* = TRUE. If $Status(A')$ = $Status(B')$ = INTERNAL, then G is non-planar. If not, then we form a component, called a *psudo_component*, as (A', B') if $Status(B')$ = FREE or as (B', A') otherwise.

(iv) If $v_i = g^-(C, e_i)$ and if *RELAXFLAG* = TRUE, then the relation between the old component C received over an *IN – attachment* at v_i and the pseudo-component remains relaxed when the condition $low_1(Y) < g^-(C, e_i)$ holds. If this condition does not hold, then the relaxed calculations have to be resolved as described in Case (iii) of Step 3.

(v) Let C = current component. If $g^+(C, e_i) < low_1(Y)$, then the new current component is set with the component parameters of Y, and C becomes an internal component.

Since only the current component is required for localized computations, C has to be routed appropriately as described in Lemma 5.2. This routing ensures that each tree edge is traversed exactly once by at most two messages containing component parameters. Also, because this is the second type of calculations which may require relaxations, we set *NESTEDFLAG* = TRUE.

(vi) Let C = current component. If *NESTEDFLAG* = TRUE, and if $v_i = low_1(I(C, e_i))(low_1(O(C, e_i)))$ with $v_i < low_1(O(C, e_i)($ $v_i < low_1(I(C, e_i)$, we call the attachments in the segment ê $_i$ *accumulated attachments*. In this case, the current component is updated to obtain a single sided component consisting entirely of accumulated attachments.

(vii) If *NESTEDFLAG* = TRUE and if the current component C is two sided, then the segments ê $_i$) are merged as described in Step 3. If Case 3 (i) holds, ê $_i$ is said to be *incompatible*. In this case, G is non-planar.

(viii) If *NESTEDFLAG* = TRUE and $v_i = low_1(I(C, e_i))$, where C is a single sided component consisting entirely of accumulated attachments, then the relation between the accumulated attachments and the old pair received must be resolved. If the current component remains nested with respect to the old pair received, the current component is maintained unaltered. The decision is made prior to sending the control to v_{i-1} To route the old pair either over the tree edge along with the current pair or over the backedge if additional nesting is needed while processing ê $_i$ at v_i. How this decision is made follows from Lemma 5.2.

(ix) When *NESTEDFLAG* = TRUE, the cases when no further nestings are needed at v_i and when the accumulated attachments are directly linked to the old pair are processed in a fashion similar to the calculations described in Step 3.

(x) When the current component $C = (X, Y)$ is two-sided, and if $g^-(C, e_i) < low_1(SAC) < g^+(C, e_i)$, where *SAC* denotes the set of accumulated attachments,

and if *RELAXFLAG* = TRUE, it is necessary to merge *SAC* with either X or Y. Because of the localized computations, a *relaxed merging* has to be carried out at the vertex v_i, where $Tail(SAC) < v_i = g^-(C, e_i)$. This relaxed merging is described in Lemma 5.2.

(xi) All other cases for merging (A, B), (A', B') and the remaining cases when G is non-planar may be carried out easily using the component parameters available at v_i. For example, (A, B) and (A', B') are merged to form a new pair $(A\#A', B\#B')$, where "#" denotes concatenation, when $Status(A') = Status(B')$ = FREE, and when the conditions $g^+(A) \leq g^-(A')$ and $g^+(B) \leq g^-(B')$ holds. Since the information to determine if these conditions hold is available locally, such a merging can be carried out. Similar arguments hold for the remaining cases.

Step 7 :

If the control is returned to the root from all its sons, and if all computations are successful, the graph is planar.

5.7 Complexity Analysis and Proof of Correctness.

In the previous section, we presented the *DPT* algorithm employing two messages with the above said message format. In this section we prove that time and message complexities of the algorithm are both O(|V|). We also show that the algorithm is optimal in message complexity and further demonstrate that DDFS has message complexity lower bound of O(|V|). We also establish that the communication complexity of our algorithm is less than previously reported algorithms in terms of total number of bits used for communication, whenever $e = O(n^2)$.

Lemma 5.1 The status of any component of $path(e) = v_1 v_2 \ldots v_k$ may be determined by maintaining at each vertex v_i the information $low_1(e)$, $teail(e)$, the set of back edges incident out of v_i, and the values ($low_1(\hat{e}_i)$, $low_2(\hat{e}_i)$) for each segment $\hat{e}_i$) incident on v_i.

Proof: The status of a component is *Internal* if there is an attachment (s,t) with $low_1(e)$ with $low_1(e) < t$ $Tail(e)$ where (s,t) is either a backedge incident on some v_i in $path(e)$ or an attachment in some $\hat{e}_i$. If the former case does not apply, $low_1(\hat{e}_i)$ can be used to determine the status except for $low_1(\hat{e}_i) = low_1(e)$, in which case $low_2(\hat{e}_i)$ may be used. □

Because of the distribution of stack, the information about the components other than the current component are not available at any vertex v_i. Hence it may be necessary to carry out relaxation for checking the relationship of *accumulatedpair*, B' with the $old\_pair[X, Y]$ received from a backedge incident at v_i. Result of a relaxed computation may be either to declare the graph is non-planar or may be to carry out relaxed_merging to form a new pair $[X', Y']$. Note that the new pair $[X', Y']$ is formed when $g_X^+ > g_{B'}^- > g_Y^+$ where X *dl* Y and $last(X) > last(Y)$. In such a case, X' is set to X where as Y' is obtained by concatenating Y with B' represented as $Y\#B'$ where $g_{Y'}^- = g_Y^-$, $g_{Y'}^+ = g_{B'}^+$, $Next(g_{B'}^- > g_Y^+$. Note that the first 2 equality may be set using the available information from the accumulated_pair and the old_pair. The third equality must be relaxed till the control reaches the vertex corresponding to $g_{B'}^-$. The need to send the next bound over $Tail(g_{B'}^-)$ may be relaxed by carrying the following 3 accumulated fields in the message using only a constant space: *RXFlag_T*, $g_{B'}^-$, and g_Y^+. The observation that any nested relaxation of such computations will not arise which would have violated the constant space requirement in the message fields is expressed by the following Lemma:

Lemma 5.2: The values $g_{B'}^-$, and g_Y^+ once initialized with *RXFlag_T* as *TRUE* remain constant until the distributed control reaches the node corresponding to $g_{B'}^-$.

Proof: It suffices to check the following cases:

Case (1): Condition - $low_1(B) > low_1(X)$

Note that an additional need for relaxed merging arises only if *current_pair* is single sided. If the condition of case 1 is satisfied then the current pair remains 2-sided until the control reaches node at g_B^-.

Case (2): Condition - $low_1(X_2) > low_1(B)$

It may then be necessary to process an attachment (s,t) where $low_1(X) > s > low_1(B)$. The following two situations will have to be considered:

(i) If $t < g_Y^-$, it may be necessary to carry out an additional relaxed merging after the control reaches $low_1(Y_2)$. But, by this point, the relaxation initialized for g_B^- is complete and the values g_B^- and g_Y^+ may be reinitialized. (ii) If $t \geq g_Y^-$, the current_pair information is completely available and there is no need for relaxed merging.

Hence the lemma. □

Theorem 5.1. DBT has the message complexity lower bound of O(n) for n > 1.

Proof: Since information that is not local but is of global topological interest have be exchanged using communication alone, it is clear that atleast one message must be sent from any node i to any node j in the network, i,j ∈ {V}. Hence atleast |V|-1 messages must be sent over the links so as to communicate to all the $O(n)$ nodes atleast once. Hence *DBT* has an O(n) lower bound in message complexity. □

Theorem 5.2 The proposed algorithm is optimal in communication complexity and uses exactly $3n-3$ messages.

Proof: Every node in the network except the root node receives only one DISCOVER message from its father (forward path) and sends one DISCOVER message to its father (return path) . Also it is clear from the algorithm that DISCOVER message is not sent to an already VISITED node in the network. Thus, each of the

|V| - 1 nodes (excluding the root) exchanges DISCOVER messages with its father exactly twice. Further, the information about biconnected component pointer is sent from the articulation point to its descendents using the tree edges alone. Hence, the total number of messages used in the algorithm is exactly 3|V| - 3. Clearly, the message complexity of the algorithm is O(n) and is optimal within a constant. □

Theorem 5.3 The algorithm terminates after $3n-2$ units of time, if all messages are delivered in one unit of time.

Proof: Total time needed for the algorithm to construct DDFS is the time required to transmit the messages over the links. From theorem 4.2 , total time needed to transmit a total of $3n-3$ messages is $3n-3$ units of time if all messages are delivered in atmost one unit of time. □

We now compare our algorithm with the previous algorithms for bit-wise message complexity. Let n be the total number of nodes in the network and m be the number of bits transmitted for each message.

Lemma 5.4 The total number of bits used for communication in our algorithm is less than that for the previous algorithms, for n > 1 and m > 4 bits, for a fully connected network.

Proof In our algorithm, the number of bits transmitted for message is atmost $O(n)$ bits due to the bit array that is appended to the messages. Since the message complexity is $O(n)$, total number of bits used is $O(n^2)$. However, all previous algorithms use $O(e)$ messages with $O(\log n)$ bits each. Hence, for a fully connected network, total number of bits used for communication is $O(n^2 \log n)$ since $e = O(n^2)$ for fully connected graphs. Hence the total number of bits used for communication is reduced by $O(\log n)$ and hence the proof.□

5.8 Conclusions

We have presented a simple to understand *DBT* algorithm that is optimal in both time and message. The algorithm is shown to use exactly 3|n| - 3 messages with the time complexity of 3|n|-3 units. We have shown that the extended message format reduces overall message complexity in terms of total number of bits used for communication under the assumed model whenever $|E| = O(|n|^2)$.

As in[14], we use the result by Reif [25] that the DFS problem is inherently sequential while developing proposed algorithm. It is interesting to see that the proposed solution is valid for both synchronous and asynchronous communication models. We will explore the further possibilities of extending our results to reliable communication models.

CHAPTER 6

SOFTWARE REDUNDANCY TECHNIQUES

It is wise to look ahead, but difficult to look further than you can see

-- Winston Churchil

6.1 Overview

In recent years, parallel computation has gained importance in all areas of computer science. This has been motivated by the fact that the scope for sequential computing is limited by the speed of the hardware available today and by the efficiency of the software written on it. The design of a parallel algorithm involves the following three steps: (1) divide the problem into smaller subproblems, (2) solve these subproblems on parallel machines, (3) integrate solution of the subproblems as the final solution. This is achieved by creating a process for each of these subproblems, and executing these processes on different processors of the parallel machine. The task of decomposing a problem into smaller subproblems, without the constraint of data dependency between them, is not easy. Processes need to communicate with other processes to solve the problem. Other problems in parallel computing includes assigning of processes to processors, synchronizing the processes, and controlling the communication among the processes. Moitra and Iyengar [1] present an extensive survey on the issues and current research in the area of parallel algorithms.

Another field that has emerged as an important area of research is that of distributed computing. This emergence has been because of technological advancements made in the area of computer communications and VLSI. While such advancing technologies have made the programming of robust distributed applications more and efficient than ever before. Because of increasing importance of reliability in

distributed applications, it has become necessary to use software-based reliability techniques instead of the currently used approach, which are based on replicating hardware-modules. Replication is used in distributed computing systems to achieve significant increase in fault tolerance, resource sharing, and availability.

Currently, the method of providing fault tolerance in interconnection networks has been based on hardware oriented approaches. However, for many applications involving intensive computation, and which have to meet real-time constraints, multiplicity and replication may be used together to provide fault tolerance as well as high performance. Communication and synchronization is also of primary importance in such systems. The former reduced the cost of communications, the latter has emerged the use of a group of specialized computers instead of a general-purpose computer. The approaches for developing distributed an application should try to optimize the number of messages needed to be exchanged to solve the problem. It would be interesting to study the problem of replicating a distributed application on interconnection network with an objective of reducing the overall inter-process communication.

Replication based fault tolerance uses replication to provide redundancy in order to tolerate faults. Many researchers have used replication as the basis for providing software fault tolerance in distributed systems. SIFT uses synchronous message passing and watchdog processor based general rollback techniques, along with communication between replicated processes [2]. Fault tolerant distributed file service based on replicated processes is supported in[3]. Replication ahs also been used in ADA programs with remote tasking [4]. Process replication along with many variations of checkpointing and recovery supports is used in Sender Based Message Logging [5].

These approaches provide programming facilities so that an application programmer or the system on its own, may provide generic fault tolerance. An interesting variation of application directed approach where the processes are aware of the underlying network and its topology has been used in WORMS[6]. Application directed approach needs more effort on the part of the programmer to specify which part of application is critical. However, it offers greater efficiency and may also be used to provide selective fault tolerance.

Further research to understand the mechanics of application directed replication of the programs is needed to make the task of distribution more efficient. In this thesis, we are mainly concerned with the efficient and reliable distributed replication of application programs. Specifically, we investigate the problem of improving the performance and the reliability of *binary tree-based applications* running on hypercubes when some of the nodes can fail. Replication may be used in this context for tolerating design faults in the software also, by comparing the results of multiple copies may be compared to obtain an agreement about a correct solution.

6.2 Replication in Distributed Applications

A distributed application consists of many independent processes running concurrently on different processors. In order to provide fault tolerance, critical process are duplicated and each such copy is called a replica. Even though these copies are identical, it is possible to implement them using different algorithms if "design diversity" is the goal to be achieved. Advantage of replication is that the program can continue to operate as long as at least one of the replicas of the processes are alive.

Software fault tolerance provide support for making use of redundant resources so that system reliability may be made more than the reliability of its hardware components. Various approaches for software fault tolerance was been discussed in Chapter 2. We recall the following approaches used in replication based techniques.

6.2.1 Approaches for replication

Distributed program execution uses a large amount of computing power available on different machines. Potential for exploiting such a collection of resources offers significant challenges. Basis of the following models is different from the state machine model which allows replicated state machines to execute in parallel[7,8].

6.2.2 N-Version programming

The idea for this approach derived from the hardware technique for fault tolerance called N-modular redundancy. The main idea is to execute the programs in multiple copies and to choose the results based on majority voting. Triple-modular redundancy is a special case in which the number of copies is three. In N-version programming, voting is also achieved using software, unlike the N-modular redundancy technique.

Note that the results of each of the copy is needed before a solution is chosen in this model.

6.2.3 Recovery block technique

This approach is also based on redundancy, but it allows "design diversity" so that each replica may be programmed using different algorithms.

One of the replicas is called a primary module while other replicas are called alternate modules. Each replica uses an acceptance test to check if the results of computation is acceptable. If the primary replica succeeds its test, it continues onto the next stage. If not, the alternate models are executed in turn till the acceptance tests succeed. Note that in this approach alternate modules in replica are not used if the primary block does not fail.

6.2.4 Active backup process

This method uses multiple concurrent execution of each process similar to N-version programming, but instead of voting the first available solution is chosen as the correct answer. Large-grain object support in CHORUS[9] uses this method. Fault tolerant distributed file service based on replicated processes is supported using a similar approach as in [3], using one-to-many and many-to-one communication between the processes for synchronization and communication of results.

Note that this approach may be used in conjunction with recovery block technique to improve efficiency since the primary copy can continue execution asynchronously.

6.3 Objectives of Software Redundancy

It is well known that many real world computational problems may be divided into subtasks so that intelligent algorithms may be used to execute each sub-task, in parallel. Such divide-and-Conquer paradigms when applied recursively follow a binary tree structure. We study how to provide replication based fault tolerance to binary tree based problems executing in hypercube environment.

Our primary concern is to optimize the cost of communication (the number of messages) while solving the problem. The computation needs of a problem are modeled by a graph called guest graph, in this case, a binary tree. The edges in this graph represent communication needs between nodes of the tree. The interconnection topology of the computer system is modeled by a graph called the host graph, in this case, a hypercube. The problem of mapping the guest graph into a host graph is generally refined to as embedding. Such mapping is usually guided by some constraints, which change from problem to problem.

We observe that it is possible to guide the process of embedding from the knowledge of the kind of replication approach being used to increase the average

performance. In case of N-version programming, since the results of all replicas are needed to be compared at each step of the computation, it is necessary to cluster them together in the guest graph. However, in case of recovery-block based approaches, the replicas are used only in the case of failure. Hence in the later approach, the primary objective would be to keep the modules in the primary copy as near as possible when there is a conflict between a primary module and an alternate module.

In the domain of replicated distributed programs, we study the problem of how to allocate and replicate processes to various processors so that the cost of communication in the overall program distributed over different machines is minimized. The objectives of the research is to map replicated binary trees in to hypercube such that:

1 Efficient solutions should be derived for all possible cases of binary trees and hypercubes, or say at least for replication number of 3 or less.

2 A general solution frame work should be derived for any given replication number.

In the following section, we pursue a deterministic solution for 2-replicated binary trees and later provide a "double quadratic programming framework" for general cases.

6.4 Optimal Solution for 2-Replicated Binary Trees

We first consider the problem of providing two replicas of the modules of the complete binary tree, called a 2-replicated binary tree. In this section, different replicas of the same module are mapped to different nodes in the hypercube.

6.4.1 Preliminaries

The communication needs of the computations are modeled by a graph, called the *guest graph*. This graph depicts the required interaction between the data

elements of the computation. The topology of a standard interconnection network is depicted by a graph, called the *host graph*. The guest graph is embedded in the host graph for execution. The quality of an embedding is often measured by two parameters: (a) dilation and (b) expansion. The expansion is a measure of processor utilization where as the dilation affects the communication cost.

The embedding function f maps each node in the guest graph $G = (V_G , E_G)$ into a *unique* node in the host graph $H = (V_H , E_H)$. V_G and V_H denote the node sets of the guest graph and the host graph respectively, and E_G and E_H denote the edge sets respectively. Let v_1 and v_2 be nodes in G. Since the embedding is a 1-1 mapping of V_G onto V_H, let $f(v_1)$ and $f(v_2)$ be the nodes in H that are images of v_1 and v_2 respectively. Let the distance, $d(v_i , v_j)$ between v_i and v_j be defined as the number of edges in the shortest path connecting v_i and v_j in either G or H.

Definition 6.1: The dilation d_f is defined as

$$d_f = \max \frac{d(f(v_1) , f(v_2))}{d(v_1 , v_2)}$$

Definition 6.2: The expansion e_f is defined as

$$e_f = \frac{|V_H|}{|V_G|}$$

6.4.2 Embedding augmented 2-replicated binary tree

Our technique first derives an Augmented 2-Replicated Binary Tree ($A2RBT(k)$) of height k, by augmenting two replicas of binary tree with communication edges and then maps this A2RBT into hypercube. When the processes of a $A2RBT(k)$ are mapped onto processors of a hypercube, we place the processes which need to communicate in neighboring processors, thus communication overhead is minimized.

Definition 6.4.3.1: An *Augmented 2-Replicated Binary Tree*, $A2RBT(k) = G(V, E)$, where V is the set of vertices of both the replicas of binary trees and $E = E' \cup E''$, where E' is the set of union of edges of both the binary trees, and E'' is the set of *cross edges* defined over the neighbors of a vertex in another copy of the replicated tree.

We classify the set of edges in the $A2RBT(k)$, of height k, into two classes:

(1) the set of edges in the underlying binary tree;

(2) the union of the set of cross edges in the hypercube of dimension i at level i of the complete binary tree $CBT(k)$ of height k;

(3) the edge between the *spare* and the *root* in each of the binary tree with additional spare obtained after embedding the binary tree into the hypercube. (Note that the hypercube of nearest dimension to a given binary tree always has one more node, called spare node, than the number of vertices in the binary tree. For example, a $CBT(3)$ has seven vertices, while $H(3)$ has eight nodes.)

Spare node is adjacent to the root because of the nature of embedding of binary tree. We provide an algorithm that embeds the underlying $A2RBT(k)$ such that the *root*, the spare vertex, and the vertices of the *sub tree* rooted at the *root* in each replica into two similar disjoint sub cubes of dimension $n-1$, in $H(n)$. Our algorithm recursively embeds the $A2RBT(k)$ into $H(n)$ and is given in theorem 6.1. We use induction on the height k of the $A2RBT(k)$ to prove theorem 6.1. Our basis of induction, unlike given in [10] , is given for $k = 2$, though our theorem holds good for $k = 0, 1$ trivially. The basis is so chosen in order to clarify the embedding of cross edges mentioned above in the characterization of $A2RBT(k)$. As stated in [10] , this embedding technique is similar to those used by Bhatt and Leiserson [11] and Bhatt and Ipsen [12]. We now state theorem 6.1.

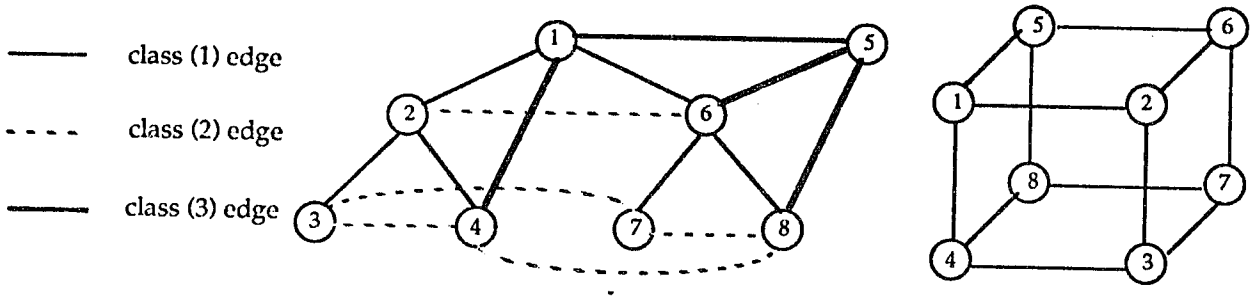

Figure 6.1. Embedding of ABG(3) into H(3)

Theorem 6.1: An *Augmented 2-Replicated Binary Tree*, *A2RBT(k)* can be embedded into a hypercube, $H(n)$ with dilation 3.

Proof: Let f_k be the function that maps an *A2RBT(k)* into an $H(n)$ with dilation 2. We define f_k by a recursive construction on k, and prove the theorem by induction.

Basis: For $k = 2$ the actual embedding of *A2RBT*(2) into $H(3)$ is given in Figure 6.1. The Theorem 6.1 clearly holds good for our basis as the function f_k is given below for $k = 2$.

$f_2(1_1, c_1) = 000$; $f_2(1_2, c_1) = 100$; $f_2(2_1, c_1) = 010$; $f_2(3_2, c_1) = 010$; $f_2(2_2, c_1) = 110$; $f_2(3_1, c_1) = 110$;

$f_2(1_1, c_2) = 001$; $f_2(1_2, c_2) = 101$; $f_2(2_1, c_2) = 011$; $f_2(3_2, c_2) = 011$; $f_2(2_2, c_2) = 111$; $f_2(3_1, c_2) = 111$;

where, $f_2(v_i, c_j)$, $1 \le i \le 3$, and $1 \le j \le 2$ is the mapping function applied on the vertex i in replica number j of the *A2RBT*(2), and the right-hand-side of the above relations refer to the binary addresses of the nodes in $H(3)$.

Hypothesis: For $k \le n$, an *A2RBT(k)* can be embedded into an $H(n)$ with dilation three, and the real root and the additional root mapped to the spare are adjacent in the cube.

Induction Step: Assume that there exists an embedding function f_m which satisfies the induction hypothesis. In order to embed an $A2RBT(m+1)$ into $H(n+1)$, we make the following observations. The remainder of the proof uses the figure 6.2.

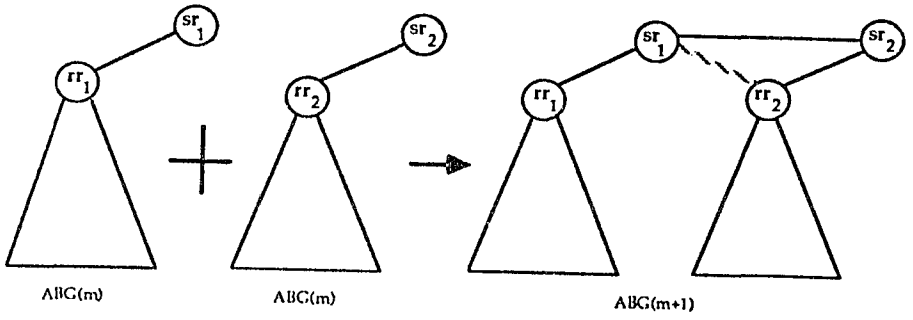

Figure 6.2 Induction Step for Theorem 6.1.

Observation 1: An Augmented 2-Replicated Binary Tree, $A2RBT(m+1)$ may be obtained from two identical copies of $A2RBT(m)$. Let rr_1 and sr_1 be the *real root* and the *spare root* of the first copy, while rr_2 and sr_2 refer to those of the second copy. In $A2RBT(m+1)$, sr_1 is the *real root*, and sr_2, the *spare root*. Note that rr_1 and rr_2 become the *left child* and *right child* of the $A2RBT(m+1)$ respectively.

Observation 2: A Hypercube, $H(n+1)$ may be obtained from two identical copies of $H(n)$. It follows from the recursive definition of hypercubes, that these two copies refer to the maximal sub cubes, and we refer to any two adjacent nodes that belong to different copies as similar nodes.

Claim: It is sufficient to show that the edge between sr_1 and rr_2 of $A2RBT(m+1)$ is dilated to 2, to prove the induction step.

Proof: Let us represent a vertex in $A2RBT(m+1)$ using three parameters: (1) copy number, c (1 or 2), (2) level of the vertex in the underlying bitonic tree of $A2RBT(m)$, l

$(0 \le l \le m-1)$, and (3) the index of the vertex in that level, i. For example, the vertex $v(1,l,i)$ is in the first copy, at level l, and its index is i. We refer to an edge, $e\,(u(c_1,l_1,i_1),u(c_2,l_2,i_2))$ as a *similar edge* where $c_1 \neq c_2$, l_2, and i_2 refer to the level and index of neighbors of u in replicated copies. Two $A2RBT(m)$'s of $A2RBT(m+1)$ can be embedded into $H(n)$ using the induction hypothesis and the Observation 1. The remaining edges of $A2RBT(m+1)$ that need to be embedded in $H(n+1)$ fall into the following two types.

Type I: The set of similar edges of $A2RBT(m+1)$.

Type II: The edge between sr_1 and rr_2. We refer to this edge as *prime cross edge.*

In our embedding algorithm, the *Type I* edges are dilated to 2 since the two end vertices of these edges are mapped onto nodes in one level higher or lower in the replicas, and thus are mapped on to two dilated nodes in the hypercube, $H(n)$. This follows from Observation 2. Now we are left with only the *Type II* edge and hence the claim.

We now consider the *Type II* edge. Since sr_1 and sr_2 are connected by a *similar edge* (dilated to 2), and sr_2 and rr_2 are atmost the corner of a sub-cube(from hypothesis), the *Type II* edge is dilated to 3. Hence the theorem.□

6.5 Heuristic Approach for General Replication Number

We consider the general case of mapping of replication of binary trees on to hypercubes in this section. We show that this problem is reducible to double quadratic programming problem[13]. Hence this problem is NP-hard for the general problem, because underlying this problem is a mapping problem which is NP-hard. We take up performance study of variations of many greedy heuristic algorithms.

6.5.1 Problem definition

We consider the problem of embedding replicated binary trees into hypercubes. Let the distance matrix D=[d_{ij}] refer to the distance between any two nodes i and j in a hypercube. Let there be n_i copies of the binary tree. Clearly two nodes x and y in a replica of a binary tree are neighbors if they are related by the neighborhood relation of parent or child in any of the replica. Let us denote the k th node in the l th by a_{kl}.

We are interested in the solution to the double quadratic assignment problem of minimizing G, where

$$G = weight(a,b) * dist((f(a), f(b)) + weight(a,b') * dist(f(a), f(b'))$$

$$+weight(a',b) * dist(f(aprime), f(b))$$

where, $\forall\ a'$ and $b'\ \in\ neighbor_j(a_i) \wedge neighbor_j(b_i)\ \ni\ j\ \in\ m \wedge i \neq j$

Note that a_i and a_j in two different replicas do not have to be adjacent through neighbors of a_i and a_j have to be nearer to both of them.
In figure 6.1, neither 2 and 2′ have to be adjacent nor 1 and 1′, but 2 has to be adjacent to 1 prime and 2 prime has to be adjacent to 1.

6.5.2 Generic greedy heuristic algorithms:

Since quadratic programming technique is NP-hard, we propose a generic greedy hueristic algorithm and study some variations so that we can gain insight into the performance of these algorithms. Many embedding problems have also been solved by similar techniques[14].

Most solutions to solve quadratic programming problems use greedy heuristics for an initial solution and then use iteratiion to improve the quality of solution[13]. Previous hypercube embedding algorithms have used heuristics as an effective technique[15]. A trade-off of the quality of solution and amount of computational effort expended is obtained by a gain function appropriate for the given problem.

In this thesis, we propose to use a selective gain-update scheme wherein the gain of the only neighbors of the most recently chosen vertex has to be updated. Further, we use the technique in [Ref..] to maintain a priority queue of lists of nodes updated, along with a record of the currently known best gain. At any stage, a vertex with the best gain is selected.

We define gain function as a measure of the advantage of choosing the most vertex q in relation to the previously chosen vertices.

$$gain(p, q) = \sum_{r \in Neighbor(p) \wedge r \notin Chosen\_set} (max - dilation - dil(q, f(r))) \qquad (6.1)$$

where p is a vertex in the binary tree in any of the m-replicas,

q is a node in the hypercube,

$Neighbor(p)$ is the neighbor set in all of the m copies

$Chosen - set$ is the set of vertices which are already mapped.

Further, if there are more than one vertex in the priority queue with best-gain index, then we choose the following variation depending upon the kind of replication desired.

(a) N-version programming:

Since this approach needs access to all the replicas in each step of computation, there is no particular advantage in favoring any nodes. Hence we choose the "vertex in depth first" method.

(b) Recovery-Block approach:

Since alternate modules are used only when primary modules fail, on an average case it is advantageous to keep primary copies near to each other. Hence we try out variations of first-in-first-out expansion where in the priority is given to the primary copy.

Note that the order of nodes inserted and deleted may be controlled using different strategies.

Figure 6.3 gives a generic greedy heuristic algorithm in terms of the gain function defined above Equation 6.1.

```
Algorithm Generic-Heuristic
Begin
    For the height of the CBT, obtain the corresponding hypercube with appropriate
    dimension.
    Initialize gain-matrix and priority queue
    Obtain replication number and initialize adjacency list
    Assign root as the first-vertex
    Update gain-matrix and priority queue
    While (not all vertices are mapped) do
    Begin
        cur-vertex = delete-maximum ( priority queue, best gain )
        select next node and map the current vertex
        Update gain-matrix for neighbors
    End
End
```

Figure 6.3. A Greedy Algorithm

6.6 Implementation Results.

The above algorithms were implemented in the following three different schemes for both Depth-first insertion-deletion order as well as first-in-first-out order.

6.6.1 Distributing replicas strictly on different nodes.

In order to support partial hardware failures, different replicas have to be executed on different machines. This way, failure of a machine still permits continuation of the program using other replicas. The program implemented is called *nodup.*

6.6.2 Permitting repeating replicas on the same node.

If the goal of the design diversity is pursued, the different replicas are implemented using different algorithms. Thus, more than one replica may be executed on the same node. The program implemented is called *nver.*

6.6.3 Higher expansion of hypercube

If overloading the same node with more than one replica of even different vertex of the tree is not permitted, then higher expansion of the hypercube may be used. We have experimented the expansion of the hypercube by a factor of the replication number so that each node has to execute atmost one replica. This method however needs more number of nodes and is hence costly to implement. The program implemented is called *high_exp.*

6.6.3.1 With distribution:

Here the nodes selected by the greedy algorithm chooses the vertices from the higher dimension freely. The program implemented is called *fifov.*

6.6.3.2 Multiplicity of primary copy:

The algorithm embeds the primary modules using the nodes in the lowest dimension hypercube and repeats the same embedding in higher dimensions. The program implemented is called *static.*

6.7 Comparison of Results

For comparison, we provide the results of triple-modular equivalent of 3-version programming for binary trees for heights between two and five. We observe from actual simulation that results for 3-version programming as well as recovery block equivalent *fifo* approach is better than that of *highexp* approach. Hence it is encouraging to study deterministic approaches for higher versions also.

6.8 Conclusions

The problem of software based replication methods is promising for providing fault tolerance. In previous sections we obtained an algorithm for embedding with 3-dilation for the replication number of two. We are also desirous to investigate the feasibility of a deterministic algorithm for a generalized higher replication number.

It is encouraging to learn that even greedy heuristics perform very well for such complex double quadratic programming problems. Since this approach is very general, we will apply this technique to various other data structures and architectures.

CHAPTER 7

CONCLUSIONS

Focus of this dissertation has been mainly to design high performance algorithms that find applications in tool-based fault-tolerant distributed computing. Since the field of software fault tolerance is still in its beginning, it is important to exploit application specific features to provide efficient fault-tolerance. Motivation of our research is to provide a software reconfiguration scheme that help continuing computation using the process trace from faulty nodes, if possible, using deterministic process reallocation schemes.

In the context of embedding quad-tree based applications into hypercubes, we have approached a software based fault tolerance to make use of the idle nodes. An efficient reconfiguration solution is provided for some specific types of failures. Efficient heuristic algorithms are presented for general faults in this context. Simulation results look encouraging to pursue this technique for other data structures and architectures.

Unlike traditional approaches of embedding in a faulty network with the additional restrictions, we provided a software environment with fault tolerant capabilities to continue the execution of application programs. We introduced the problem of completing the computation whenever some nodes used in the embedding are diagnosed as faulty. The idea of reconfiguration while maintaining the characteristics of the initial embedding is new. In the case of reconfiguring a complete quadtree into a hypercube, we provided a 2-dilation reconfigurable embedding which yields an optimal 3-dilation reconfiguration. Our approach uses process migration with minimum number of migrations. We compared our results with existing algorithms and have shown the superior performance of our algorithm.

Towards providing selective replication and faults tolerance, knowledge of biconnectivity is very useful. We provide the most efficient distributed algorithm known to date to obtain articulation points, bridges and biconnected components of an asynchronous networks. Our algorithm provides specification of correctness for distributed biconnectivity. Solution is developed using incremental refinement. We have presented a new distributed algorithm for finding articulation points, bridges and biconnectivity components in an asynchronous networks. The communication and time complexities of our algorithm are both $O(n)$. The most efficient distributed algorithm currently known for these problems has a message complexity of $O(m)$ and time complexity of $O(d)$ where n is the number of nodes, m is the number of arcs, and d is the diameter of the graph corresponding to an asynchronous network. The proposed algorithm is derived as a step-wise refinement from a general specification of the problem to a concrete solution and is simple to understand compared to the existing algorithms. The reduction in message complexity in the proposed algorithm is achieved based on a reasoning from a knowledge about the previously visited nodes in a distributed depth first search. We have also compared our algorithm with the distributed algorithms presently known for these problems with respect to bit-wise message complexity. Though message bound is linear in our algorithm, in the worst case, total number of bits used is also reduced by $O(logn)$ compared to previous algorithms.

Planarity testing is one of the most celebrated problems in the area of graph theory and algorithm design. Though there are many sequential algorithms and parallel algorithm, no distributed algorithm is known to date. We provide an optimal distributed algorithm which is based on Hopcroft and Trajan's sequential algorithm. In spite of extensive and many well-known results of optimal linear time sequential algorithm as well as an optimal $O(logn)$ time parallel algorithm, currently, an algorithm for Distributed Planarity Testing (*DPT*) was not available in literature. The

problem of efficient *DPT* was left as an open problem in [1]. Motivated from the result in[2] that depth first search is inherently sequential, we pursue an algorithm for *DPT* based on Distributed Depth First Search (*DDFS*) to minimize the number of messages required by eliminating any parallelism. The communication complexity of our algorithm is $O(n)$ which is optimal within a constant. The time complexity of our algorithm is also $O(n)$. In case of our *DPT* algorithm, we also show that the bound on the message size may be reduced from $O(n \log n)$ to $O(\log n)$ bits, except for the initial phase to find out if $m \leq 3n-6$ using *DDFS* algorithm which uses n-bit message. Solution is simple to understand with the knowledge of the corresponding sequential algorithms.

Replication based techniques are used recently in distributed systems for hardware based fault tolerance. We explore the possibility of using replication to application based on binary tree. We provide an efficient 3-dilation reconfiguration for binary tree if there are only 2-replicas. Simulation studies look interesting to be applied to general cases of replication also. The insight gained about this technique is available tolerate hardware faults as well as design faults.

We have presented efficient algorithms for four problems to support fault tolerant applications. Performance of our algorithms are shown to be superior to the existing solution.

Scope of Future Research

1. Reconfiguration approach presented in chapter 3 may be applied to a number of problems and variety of architectures. We are currently examining a fault tolerant sorting application to be executed on orthogonal networks.
2. We are currently developing techniques to take care of different classes of faults. It is possible to extend our methodology for a variety of computational graph structures onto various interconnection networks such as star graphs, and

generalized boolean n-cubes We have characterized the computational aspects of software reconfiguration problem and have developed data structures that provide efficient reallocation of faulty processes.

3. Biconnected components solution may be extended to many communication protocols and distributed algorithms to improve robustness.

4 Distributed planarity has many application to solve problem which are different to solve on a general network. Since efficient planarity testing is possible, it is interesting to investigate efficient solutions to many such problems. Distributed Maximal Planarity Testing is another problem.

5 It would be interesting to see up to what replication number, it is possible to give deterministic replicated embedding of classical data structures on to practical interconnection networks.

REFERENCES

1. Michael Stumm and Songnian Zhou, "Algorithms Implementing Distributed Shared Memory," *Computer*, pp. 54-64.

2. H. Zimmermann, "OSI Reference Model - The ISO Model for Architecture for Open Systems Interconnection," *IEEE Trans. on Communications*, vol. 28(4), pp. 425-432.

3. Flavin Cristian, "Understanding Fault-Tolerant Distributed Systems," *Communications of the ACM*, vol. 34, pp. 56-78, February, 1991.

4. M. Raynal, "Networks and distributed computation: concepts, tools and techniques," *The MIT Press, Cambridge*, 1988.

5. Mohan Sharma, Sitharama Iyengar, and Narasimha Mandyam, "An Efficient Distributed Depth-First-Search Algorithm," *Information Processing Letters*, vol. 32, pp. 183-186, North Holland, 1989.

6. Gallager, R. G., Humblet, P. and Spira, P., "A distributed algorithm for minimum weight spanning trees," *ACM Trans. Programming Languages and Systems*, vol. 5, No. 1, pp. 66-77.

7. B.Awerbuch, "A new distributed depth-first-search algorithm," *Inform. Process. Lettr.*, vol. 20(3), pp. 147-150.

8. E.J.H. Chang, "Echo Algorithms : Depth Parallel Operations on General Graphs," *IEEE Trans. on Software Engineering*, vol. SE-8, NO. 4, pp. 391-401, 1982.

9. A. S. Tanenbaum and R. V. Renessee, "Distributed Operating Systems," *Computing Surveys*, vol. 17(4), pp. 419-470, ACM, December 1985.

10. H. E. Bal, J. G. Steiner, and A.S. Tanenbaum, "Programming Languages for Distributed Computing Systems," *Computing Surveys*, vol. 21(3), pp. 261-322, ACM, September 1989.

11. R. J. LeBlanc and T. Wilkes, "Systems Programming with Objects and Actions," *Proc. of the 5th Int. Conf. on Distributed Computing Systems*, pp. 132-139, IEEE.

12. B. Liskov, "Distributed Programming in Argus," *Commn. ACM*, vol. 31(3), pp. 381-404.

13. E. C. Cooper, "Replicated Distributed Programs," *Proc. of the 10th Symp. on Operating Sys. Principles*, pp. 63-78, ACM-SIGOPS.

14. L. Lamport, "Using Time instead of Timeout for fault-tolerant distributed systems," *Tech. Rep. 59*, SRI, June 1981.

15. F. B. Schneider, "Synchronization in Distributed Programs," *ACM TOPLAS*, vol. 4(2), pp. 125-148, April 1982.

16. B. Lampson, "Atomic Transactions," *Lecture notes in computer science - Distributed Systems, Springer-Verlag*, vol. 105, pp. 246-265, 1981.

17. A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Call," *ACM TOCS*, vol. 2(1), pp. 39-59.

18. R. D. Schlichting and F. B. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM TOCS*, vol. 1(3), pp. 222-238.

19. B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM TOPLAS*, vol. 5(3), pp. 381-404.

20. R.E.Strom and S. Yemini, "NIL: An Integrated Language and System for Distributed Programming," *ACM SIGPLAN '83 Symp. on Prog. Lang. Issues in Soft. Sys.*, pp. 73-82.

21. P. Jalote and S. K. Tripathi, "Fault Tolerant Computation in Synchronous Message Passing Models," *University of Maryland, College Park, MD.*

22. L. Mancini, "A Watchdog Processor based General Rollback Technique with Multiple Retries," *IEEE TSE*, vol. SE-12(1).

23. J. Wensely, et. al., "SIFT : Design and Analysis of a Fault Tolerant Computer for Aircraft Control," *Proc. of the IEEE*, vol. 60(10), pp. 1240-1254.

24. A. Black, et. al., "Distribution and Abstract Types in Emerald," *Tech. Rep. 86-02-04*, University of Washington, Seattle.

25. A. Z. Spector, et. al., "The Camelot Project," *Rep. CMU-CS-86-166*, Dept. of Comp. Sc., CMU, Pittsburgh, PA.

26. M. M. Theimer, K. A. Lantz, and D. R. Cheriton, "Preemptable Remote Execution Facilities for the V-System," *Proc. of the 10th Symp. on Operating Sys. Principles*, pp. 2-12, ACM-SIGOPS.

27. S. J. Mullender and A. S. Tanenbaum, "A Distributed File Service based on Optimistic Concurrency Control," *ACM 10th Symp. on Software Principles.*

28. J. S. Banino, et. al., "Some Fault Tolerant Aspects of the CHORUS distributed System," *Proc. of the 5th Int. Conf. on Distributed Computing Systems*, pp. 430-437, IEEE.

29. P. M. Melliar-Smith and R. L. Schwarz, "A Fault Tolerant Ada Architecture," *Proc. of the 4th. Jerusalem Conference on Information Technology.*

30. R. F. Cmelik, N. H. Gehani, W. D. Roome, "Fault Tolerant Concurrent C: A tool for Writing Fault Tolerant Distributed Programs," *IEEE Int. Symp. on Fault-Tolerant Computing*, pp. 56-61.

31. J. M. Smith, "A Survey of Process Migration Mechanisms," *ACM Operating Systems Review*, vol. 22(3), pp. 28-40.

32. M. L. Powell and B. P. Miller, "Process Migration in DEMOS/MP," *ACM Operating Systems Review*, vol. 17(5).

33. B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System," *ACM Operating Systems Review*, vol. 17(5).

34. D. B. Johnson and W. Zwaenepoel, "Sender based Message Logging," *FTCS 17 Digest of Papers*, Pittsburgh, PA.

35. R.E.Strom and S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM TOCS*, vol. 3(3), pp. 204-226.

36. K. Marzullo, R. Cooper, M. D. Wood, and K. P. Birman, "Tools for Distributed Application Management," *Computer*, pp. 42-51.

37. J. F. Shoch and J. A. Hupp, "The Worms Program - Early Experience with a Distributed Program," *Comm. of the ACM*, vol. 25(3).

38. John Hopcroft and Robert Tarjan, "Efficient Planarity Testing," *Journal of the ACM*, vol. Oct. 1974, pp. 549-568.

39. Greg N. Frederickson, "A Distributed Shortest Path Algorithm for a Planar Network," *Information and Computation*, vol. Vol. 86, pp. 140-159, 1990.

40. Lee S.Y, J.K. Aggarwal, "A Mapping Strategy for Parallel Processing," *IEEE Transactions on Computers*, vol. C-36(4), pp. 433-442, April 1987.

41. K. G. Shin and Y. Chang, "Load Sharing in Distributed Real-Time Systems with State-Change Broadcasts," *Trans. on Computers*, vol. 38(8), pp. 1124-1142, IEEE.

42. John A. Stankovic, "Decentralized Decision Making for Task Reallocation in a Hard Real-Time System," *Trans. on Computers*, vol. 38(3), pp. 341-355, IEEE.

43. J. F. Kurose and R. Simha, "A Microeconomic Approach to Optimal Resource Allocation in Distributed Computer Systems," *Trans. on Computers*, vol. 38(5), pp. 705-711, IEEE.

44. S. Dutt, J.P. Hayes, "Design and Reconfiguration Strategies for Near-Optimal k-Fault-Tolerant Tree Architectures," *IEEE-Conference on Fault Tolerance*, pp. 328-333, 1988.

45. E. Horowitz, A. Zorat, "The Binary Tree as an Interconnection Network: Application to Multiprocessor Systems and VLSI," *IEEE Transactions on Computers*, pp. 247-253, April 1981.

46. A. Despain, D. Patterson, "X-tree: A Structured Multiprocessor Computer Architecture," *5th Symposium on Computer Architecture*, pp. 144-151, April 1978.

47. S.K. Chen, C.T. Liang, W.T. Tsai, "An Efficient Multi-Dimensional Grids Reconfiguration Algorithm on Hypercubes," *International Conference on Parallel Processing*, pp. 368-373, 1988.

48. Flavin Christian, "Understanding Fault-Tolerant Distributed Systems," *Comm. of the ACM*, vol. 34(2), pp. 56-78, February 1991.

49. George A. Champine, et. al., "Project Athena as a Distributed Computer System," *Computer*, pp. 40-51, IEEE, September 1990.

50. Angluin Dana, "Local and Global Properties in Networks of Processors," *Proceedings of STOC*, pp. 82-93, ACM, 1980.

51. Eli Gafni, "Generalized Scheme for Topology-Update in Dynamic Networks," *Lecture Notes in Computer Science*, vol. 312, pp. 187-196, Springer-Verlag, 1987.

52. Yehuda Afek, Baruch Awerbuch, and Eli Gafni, "Local Fail-safe Network Reset Procedure," *Proceedings of FOCS*, vol. 28, pp. 199-211, IEEE, 1987.

53. Yehuda Afek, Baruch Awerbuch, and Eli Gafni, "Applying Static Network Protocols to Dynamic Networks," *Proceedings of FOCS*, pp. 358-370, IEEE, 1987.

54. Yehuda Afek, Baruch Awebuch, Serge A. Plotkin, and Micheal Saks, "Local Management of a global Resource in a Communication Network," *Proceedings of FOCS*, pp. 347-357, IEEE, 1987.

55. David Stevenson, *Token-based Consistency of Replicated Servers,* CH2686-4/89/0000/0179, pp. 179-183, IEEE, 1989.

56. T. Anderson and P. A. Lee, "Fault Tolerance : Principles and Practice," *Prentice-Hall, Englewood Cliffs, New Jersey.*

57. K. P. Birman and T. A. Joseph, "Reliable Communication in Presence of Failures," *ACM TOCS*, vol. 5(1), pp. 47-76.

58. K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining global states of distributed systems," *ACM TOCS*, vol. 3(1), pp. 63-75, January 1985.

59. H. Garcia-Molina, "Broadcast Algorithms," *Trans. on Computers*, IEEE.

60. Mee Yee Chan and Shiang-Jen Lee, "Distributed Fault-tolerant Embeddings of Rings in Hypercubes," *Journal of Parallel and Distributed Computing*, vol. 11, pp. 63-71, Academic Press, 1991.

61. Johan Hastad, Tom Leighton, and Mark Newman, "Fast Computation using Faulty Hypercubes," *Proccedings of STOC*, pp. 251-263, ACM, 1989.

62. Y. Saad, M.H. Schultz, "Some Topological Properties of the Hypercube Multiproceesor," *RR-389, Dept. Computer Science, Yale University*, 1984.

63. A. Wagner and D.G. Corneil, "Embedding Trees in a Hypercube is NP-complete," *Technical Report 197/87*, pp. 1-34, Dept. of Computer Science, University of Toronto, Toronto, Ontario, February 1987.

64. Wu A.Y, "Embedding of Tree Networks into Hypercubes," *Journal of Parallel and Distributed Computing*, vol. 2, pp. 238-249, 1985.

65. S.N. Bhatt, F. Chung, T. Leighton, A. Rosenberg, "Optimal Simulations of Tree Machines," *Proceedings of FOCS*, pp. 274-282, 1986.

66. L. Barasch, S. Lakshmivarahan,S.K. Dhall, "A New Algorithm for Embedding Complete Binary Trees in Binary Hypercubes," *TR-87-07, Parallel Processing Institute, University of Oklahoma, Norman*, 1987.

67. B. Monien, I.H. Sudborough, "Simulating Binary Trees on Hypercubes," *Proceedings of the Third AWOC*, 1988.

68. Tom Leighton, Mark Newman, Abhiram Ranade, and Eric Schwabe, "Dynamic Tree Embeddings in Butterflies and Hypercubes," *Proceedings of STOC*, pp. 224-234, ACM, 1989.

69. S.N. Bhatt, E.C.F. Ispen, "How to Embed Trees in Hypercubes," *RR-443, Dept. of Computer Science, Yale University*, 1985.

70. J.E. Bradenberg, D.S. Scott, "Embedding of Communication Trees and Grids in Hypercubes," *Technical Report, Intel Scientific Computers, Beaverton, Oregon*, 1985.

71. S.H. Bokhari, "On Mapping Problem," *IEEE Transactions on Computers*, vol. C-30(3), pp. 207-214, March 1981.

72. M. Hanan, J.M. Kurtzberg, "A Review of the Placement and Quadratic Assignment Problems," *SIAM Review*, vol. 14, pp. 324-342, April 1972.

73. A. Klinger, C.R. Dyer, "Experiments in Picture Representation Using Regular Decomposition," *Computer Graphics Image Processing*, vol. 5, pp. 68-105, 1976.

74. H. Samet, "The Quadtree and Related Hierarchical Data Structures," *ACM Computing Surveys*, vol. 16, pp. 187-260, 1984.

75. L. Jones, S.S. Iyengar, "Space and Time Efficient Virtual Quadtrees," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 6, pp. 244-247, 1984.

76. I. Gargantini, "An Efficient Way to Represent Quadtrees," *Communications of ACM*, vol. 25, pp. 905-910, 1982.

77. H. Samet, "A Quadtree Medial Axis Transform," *Communications of ACM*, vol. 27, pp. 680-693, 1983.

78. H. Samet, "Connected Component Labeling Using Quadtrees," *Journal of ACM*, vol. 28, pp. 487-501, 1981.

79. H. Samet, "Computing Perimeters of Images Represented by Quadtrees," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 3, pp. 683-687, 1981.

80. H. Samet, "Neighbor Finding Techniques for Images Represented by Quadtrees," *Computer Graphics Image Processing*, vol. 18, pp. 37-57, 1982.

81. C.T. Ho, S.L. Johnsson, "Dilation d Embedding of a Hyper-Pyramid into A Hypercube," *Proceedings of the Supercomputing '89*, pp. 294-303, November 1989.

82. Vinayak Hegde, Krishnakumar Narayanan, S. Sitharama Iyengar, "Optimization strategies for fault propogation schemes," *Technical Report*, vol. RRL-TR-91-010, pp. 1-90, Robotics Research Laboratory, Dept. of Computer Science, LSU, Baton Rouge, LA 70808., July 1991.

83. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1974.

84. R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. of Computing*, vol. 1(2), pp. 146-160.

85. T.Cheung, "Graph traversal techniques and the maximum flow problem in distributed computation," *IEEE Trans. Software Engineering*, vol. SE-9(4), pp. 504-512.

86. K.B.Lakshmanan, N.Meenakshi and K.Thulasiraman, "A time optimal message-efficient distributed algorithm for depth-first-search," *Inform. Process. Lett.*, vol. 25(2), pp. 103-109.

87. J. Lin, S. Yang, and M. Chern, *An Improved Distributed Algorithm for the Biconnectivity Problem*, pp. 405-409, 1990.

88. Shing-Tsaan Huang, "An New Distributed Algorithm for the Biconnectivity Problem," *Proc. International Conf. on Parallel Processing*, vol. III, pp. 106-113, 1989.

89. Mohan Ahuja and Yahui Zhu, "An Efficient Distributed Algorithm for finding Articulation points, Bridges, and Biconnected Components in Asynchronous Networks.," *Proc. Foundations of Software Technology and Theoretical Computer Science*, vol. 405, pp. 99-108, LNCS, Springer Verlag, 1989.

90. Gregory R. Andrews, "Paradigms for Processe Interaction in Distributed Systems," *ACM Computing Surveys*, vol. 23(1), pp. 49-90, March, 1987.

91. Narsingh Deo, *Graph Theory with Applications to Engineering and Computer Science,* Prentice Hall, 1990.

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

93. J.H.Reif, "Depth-first search is inherently sequential," *Inform. Process. Lett.*, vol. 20(5), pp. 229-234.

94. Janardan, R. and Cheng, S. W., "Efficient distributed algorithms for single-source shortest paths and related problems on plane networks," *4th Int. workshop on Distributed Algorithms*, vol. Springer-Verlag LNCS (486), pp. 133-150, 1990.

95. S. G. Williamson, "Embedding graphs in a Plane : Algorithmic aspects," *Ann. Discrete Mathematics*, vol. Vol. 6, pp. 349-384, 1980.

96. Tanenbaum, A. S., *Computer Networks,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

97. Jiazhen Cai, Xiaofeng Han, and Robert Tarjan, "A O(m log n)-Time Algorithm for the Maximal Planar Subgraph Problem," *Personal communication with Prof. Cai, to appear in SICOMM.*, 1992.

98. Vinayak Hegde and S. Sitharama Iyengar, "A new distributed algorithm to find articulation points, bridges, and biconnected components of a network," *To appear in the Proc. of ACM Symposium on Applied Computing*, 1993.

99. Di Battista and Roberto Tamassia, "Incremental Planarity Testing (Extended Abstract)," *Proc. 30th Annual IEEE Symposium on Foundations of Computer Science*, pp. 436-441, 1989.

100. T. Nishizeki and N. Chiba, "Planar Graphs: Theory and Algorithms," *Annals of Discrete Mathematics*, vol. 32, North-Holland, 1988.

101. Vijaya Ramachandran and John Reif, "An Optimal Parallel Algorithm for Graph Planarity (Extended Abstract)," *Proc. 30th Annual IEEE Symposium on Foundations of Computer Science*, pp. 282-287, 1989.

102. K.V.S. Ramarao, "Distributed Algorithms for Network Recognition Problems," *IEEE Trans. on Computers*, vol. Vol. 38, NO 9, pp. 1240-1248, 1989.

103. Bondy, J. A. and Murty, U. S. R., "Graph Theory with Applications," *North-Holland, New York*, 1976.

104. A. Moitra, S.S. Iyengar and A. Moitra, S.S. Iyengar, "Parallel Algorithms for a Class of Computational Problems - A Survey," *Advances in Computers*, vol. 26, pp. 94-153, 1987.

105. W. Chen, M. F. Stallmann, and E. F. Gehringer, "Hypercube Embedding Heuristics : AN Evaluation," *Int. Journal of Parallel Programming*, pp. 505-549.

106. J. Ramanujam, F. Ercal, and P. Sadayappan, "Task Allocation by Simulated Annealing," *Proc. of the Int. Conf. on Supercomputing.*

VITA

Vinayak Gajanan Hegde was born in Kumta, Karnataka, India on April 1, 1962. He obtained his undergraduate degree in mechanical engineering from Mysore University, India in 1984. He completed a research program leading to master's degree in Robotics at Indian Institute of Technology, Madras, India in 1988. He joined the Department of Computer Science at Louisiana State University, Baton Rouge, Louisiana for his Ph.D. program specializing in distributed application management. His research interests include distributed systems, operating systems, artificial intelligence, parallel processing, fault tolerance, and graph embedding.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Vinayak Gajanan Hegde

Major Field: Computer Science

Title of Dissertation: High Performance Software Reconfiguration in the Context of Distributed Systems and Interconnection Networks

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination:

September 29, 1992