A MULTIPROCESSOR SIMULATOR TO TEST FAULT DETECTION AND RECONFIGURATION ALGORITHMS

UNMESH BHATHIJA

A MULTIPROCESSOR SIMULATOR TO TEST FAULT DETECTION AND RECONFIGURATION ALGORITHMS

by
Unmesh Bhathija

A Thesis Submitted to the Faculty of the
College of Engineering
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Florida Atlantic University
Boca Raton, Florida
August 1990

A MULTIPROCESSOR SIMULATOR TO TEST

FAULT DETECTION AND RECONFIGURATION ALGORITHMS

by

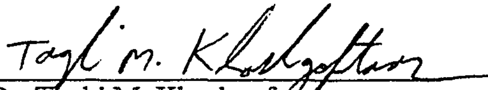
Unmesh Bhathija

This thesis was prepared under the direction of the candidate's thesis advisor, Dr. Eduardo B. Fernandez, Department of Computer Engineering and has been approved by the members of his supervisory committee. It was submitted to the faculty of the College of Engineering and was accepted in partial fulfillment of the requirements for the degree of Master of Science in Computer Engineering.

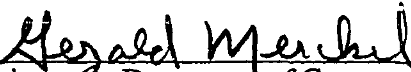
SUPERVISORY COMMITTEE:

Thesis Chairman
Dr. Eduardo B. Fernandez

Dr. Ravi Shankar



Dr. Taghi M. Khoshgoftaar



Chairperson, Department of Computer Engineering

Dean, College of Engineering

Dean of Graduate Studies

6/14/90
Date

ACKNOWLEGEMENTS

I would like to express my sincere gratitude to Dr. Eduardo B. Fernandez for his valuable guidance and most patient help during the period of this work.

I would also like to extend my thanks to Dr. Ravi Shankar and Dr. Taghi Khoshgoftaar for their suggestions.

I am extremely thankful to my parents, for making lots of sacrifices in their lives for my overall upbringing and education without which the entire course of my life would have been different. Finally, I would like to thank my wife, Shubhada (Yamu) without whose support it would have been almost impossible to finish this thesis.

ABSTRACT

Author:	Unmesh Bhathija
Title:	A Multiprocessor Simulator to test Fault Detection and Reconfiguration Algorithms
Institution:	Florida Atlantic University
Thesis Advisor	Dr. Eduardo B. Fernandez
Degree:	Master of Science in Computer Engineering
Year:	1990

In recent years multiprocessor systems are becoming increasingly important in critical applications. In particular, their fault tolerance properties are of great importance for their ability to be used in these type of applications. We have developed a multiprocessor simulator that can be used to test different fault detection algorithms. The processors must have four communication links. This simulator operates by passing messages between processors. An algorithm was developed for routing the messages among the processors. The simulator can also be used to try different reconfiguration strategies. In particular we have tested Malek's comparison algorithm using different multiprocessor configurations. We also developed a program which determines the configuration of an unknown network of transputers.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION AND OVERVIEW 1

1.1 Motivation and Objectives 1

1.2 Contributions of the Thesis....................2

1.2.1 Simulation of the architectural configuration....................2

1.2.2 Network simulation and routing algorithm....................2

1.2.3 Reconfiguration 4

1.2.4 Configuration scheme 4

1.2.5 Detection of faults....................4

1.2.6 Analysis of Malek's algorithm on different configurations 5

1.3 Overview of the Thesis....................6

CHAPTER 2 BACKGROUND....................8

2.1 The Transputer....................8

2.2 The Occam Programming Language....................10

2.2.1 Occam processes....................10

2.2.2 Occam channels 11

2.3 Fault Tolerance....................12

2.4 Reconfiguration....................13

2.4.1 Construction of an initial configuration....................14

2.4.2 Achieving a given dimension....................15

2.5 Malek's Comparison Model....................17

2.6 Summary....................20

CHAPTER 3 A SIMULATOR FOR MESSAGE PASSING MULTIPROCESSORS....21

3.1 Organization of the Simulator....21

3.2 Link Task....24

3.3 Application Task....24

3.4 Communicator Task....25

3.5 Implementation of the Tasks....25

3.5.1 Link task....25

3.5.2 Application task....25

3.5.3 Communicator task....26

3.6 Communication Operation....27

3.7 Node Database....29

3.8 Parameters governing the Network Architecture....31

3.9 Summary....34

CHAPTER 4 RECONFIGURATION IN A MULTIPROCESSOR ENVIRONMENT....36

4.1 Reconfiguration of a Network of Nine Processors....36

4.1.1 Transforming a toroid into a three dimensional prism....38

4.1.2 Transforming a three dimensional prism to a two dimensional mesh....39

4.1.3 Transforming a two dimensional mesh to a star configuration and a linear array....40

4.2 Summary....42

CHAPTER 5 CONFIGURATION OF A GENERAL NETWORK........43
5.1 The Structure of a Tracing Program under the TDS................................43
5.2 The Host Transputer..46
5.3 The Exploratory Trace PROGRAM..48
5.3.1 Introduction...48
5.3.2 Searching a neighboring transputer..49
5.3.3 Booting a neighboring transputer..50
5.4 Exploring a Tree of Transputers...52
5.5 Exploring a General Network of Transputers...................................54
5.6 Summary...56

CHAPTER 6 TESTING AND EVALUATION OF MALEK'S FAULT DETECTION ALGORITHM..57
6.1 Diagnostic Table...57
6.2 Implementation of the Algorithm..58
6.3 Extension of the Simulator to Test any Fault Detection Algorithm.................64
6.4 Evaluation of Malek's Algorithm..64
6.5 Analysis of Different Configurations..65
6.6 Conclusions..72

CHAPTER 7 CONCLUSIONS AND FUTURE WORK.....................73

REFERENCES...76

APPENDICES

Appendix A ADA SOURCE CODE FOR THE SIMULATOR..............79

Example 1 Message passing in a toroidal configuration 101

Example 2 Message passing in a mesh configuration 113

Appendix B RESULTS OF MALEK'S ALGORITHM........................121

Appendix C CONFIGURATION CODE..146

(I) Searching a tree of transputers...146

(II) Searching a general network of transputers 149

LIST OF TABLES

Table 3.1 Link interconnection map 33

Table 6.1 Diagnostic table 58

Table 6.2 Table for diagnosis of healthy nodes 63

Table 6.3 Test cycle detecting the faulty node 63

Table 6.4 Comparison parameters for case 2 67

Table 6.5 Comparison parameters for case 3 69

Table 6.6 Comparison parameters for case 4 70

Table 6.7 Comparison parameters for case 5 71

LIST OF ILLUSTRATIONS

Figures

1.1 Toroidal configuration 3

1.2 A comparator and two compared units....6

2.1 Transputer with four bidirectional links....9

2.2 Generalized switch lattice....14

2.3 Linear array from a 3-cube 16

2.4 A comparator and two units....19

2.5 Comparison model graphs....19

3.1 Toroidal configuration of nine nodes....23

3.2 Node database....30

3.3 Mesh connected network 35

4.1 A toroidal mesh of nine processors....37

4.2 A toroid with switches inserted....38

4.3 Transformation of a toroid into a prism....39

4.4 Transformation of a prism to a two dimensional mesh....40

4.5 Transformation of a mesh to a star and linear configuration....43

5.1 Transputer development system 44

5.2 A tree of transputers....52

5.3 A closed loop connection 55

6.1 Set of three processors....59

6.2 Pictorial view of the diagnosis of nine nodes 62

6.3 Nine nodes in a toroidal configuration 66

6.4 Star shaped configuration 67
6.5 Lattice structure with nine nodes 68
6.6 Lattice configuration for ten nodes 69
6.7 Five nodes connected in a pentagonal shape 71

CHAPTER 1

INTRODUCTION AND OVERVIEW

1.1 Motivation and Objectives

Recent years have brought the need for fault tolerant multicomputer systems that are capable of supporting continuous service over long periods. Typical applications are air traffic control and national defense message switching systems where clearly human life is at stake, while for banking systems, ticket reservations, telephone exchange control, and other forms of message switching, investment and revenue are more important. In fact, most real time computer control systems require fault tolerance of a measure beyond that readily obtainable from conventional computer systems.

The basic objective of fault tolerance is to provide systems with the capability of performing their intended work in the presence of faults. Fault tolerance is provided to a system through fault masking (using enough redundancy to hide the effect of a fault) or through a set of functions including fault detection, fault confinement, and system reconfiguration.

The most convenient way to obtain a high level of fault tolerance is by using multiprocessors. Other reasons why multiprocessor systems are given much importance are better performance and a lower growth cost and serviceability.

Multiprocessor architectures can be classified in many different ways:

* Based on granularity of unit of concurrency: job level, process level, instruction level, microinstruction level.
* Based on coupling between processors: loosely coupled or tightly coupled multiprocessor computer networks.
* Based on data and instruction flow: SISD, SIMD, MIMD, etc.

A further classification of multiprocessors is based on whether they fall under the category of shared memory systems or message passing systems.

This thesis describes a simulator to model some fault tolerance aspects of a message-passing multiprocessor system. Specifically we apply this simulator to study the operation of a comparison algorithm to detect faults and to analyze reconfiguration and configuration aspects of complex interconnections of processors.

1.2 Contributions of the Thesis

1.2.1 Simulation of the architectural configuration

The multiprocessor architectures which we consider are interconnections of computer nodes. The computer nodes can be of any reasonable type, provided they have at the most four serial links to form the connection edges for building a message passing network. The simulator can be applied to any interconnection structure for any number of processors by changing certain input parameters.

1.2.2 Network simulation and routing algorithm

To show an application of the simulator we simulate a network of transputers using an architecture such as the one shown in Figure 1.1. Every node has four links

which are used for message passing. Thus, while passing messages even if one or two links fail, the message can still be rerouted via another path.

A transputer is a special type of microprocessor which has a few registers and some local memory. It supports concurrent processing and has a built-in byte oriented protocol. We have selected a transputer for our simulation because it is a promising building block to construct modular multiprocessors. Its point-to-point communications architecture made up of four links makes it particularly appropriate for high-bandwidth interconnection structures.

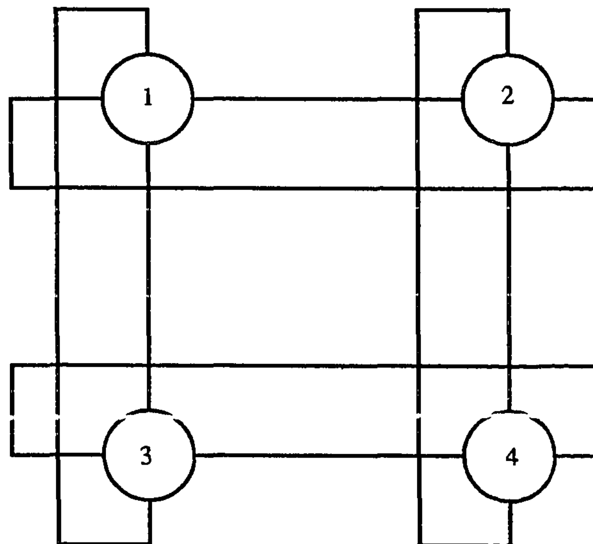


Figure 1.1 Toroidal configurartion

We have developed an algorithm for routing messages between different processors in the network. The algorithm uses a store and forward scheme for the

delivery of messages. The messages are sent and received by processors with the help of a special communicator task.

1.2.3 Reconfiguration

We apply a switched lattice method proposed by Snyder [Snyd82] to achieve reconfigurability in the multiprocessor environment. We show how a transputer based multiprocessor can move from one configuration to another with the aid of a reconfiguration controller.

1.2.4 Configuration scheme

We have developed a configuration program in Occam which traces an unknown network of transputers and determines its configuration. This program is useful when a large number of transputers are connected to form networks in multiprocessor arrays. These arrays can become quite large and complex. The program can also be used to load code segments into a network whose configuration is not known in advance.

1.2.5 Detection of faults

We extend the simulator to implement a specific fault detection algorithm. In particular we implement a comparison algorithm to detect a faulty unit in a network of transputers. However, the simulator can be applied for various other fault detection algorithms.

This comparison algorithm is based on a model introduced by Malek, and can be explained as follows:

As shown in Figure 1.2, let us consider a set of three processors. We want to find which of the three units (1 or 2) is faulty. We proceed by assigning unit 3 the tasks of a comparator. Unit 3 must be a healthy unit. Unit 3 assigns some tasks to nodes 1 and 2 and then compares their outputs. If it detects a mismatch it can determine that there is a faulty unit. If the outputs match, we conclude that nodes 1 and 2 are fault-free. We prove unit 3 to be healthy in a similar fashion.

We show how other fault detection algorithms can be implemented by changing some input parameters of the simulator.

1.2.6 Analysis of Malek's algorithm on different configurations

We find out the comparison parameters, i.e. the number of comparison cycles and the comparison edges required to detect a single faulty processor in different environments. We consider five different cases, which describe different multiprocessor architectures. Malek's algorithm is applied to each of these structures and the comparison parameters are calculated for all these different configurations. These results can be used to verify the operation of the simulator.

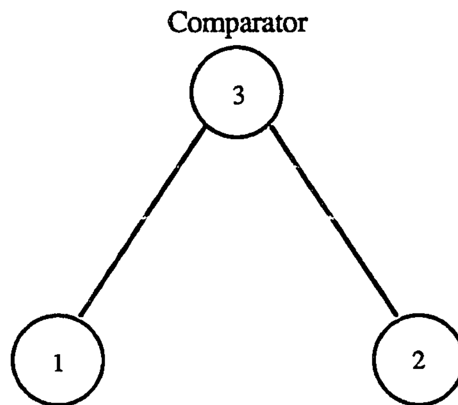


Figure 1.2 A comparator and two compared units

1.3 Overview of the Thesis

Chapter 2 elaborates some background information on the transputer architecture and Malek's algorithm. It also explains the switched lattice method proposed by Snyder.

Chapter 3 describes the general structure of the simulator and the various data structures used in it. It also describes how communication is implemented.

Chapter 4 describes a reconfiguration scheme proposed by Snyder to restructure the network of processors to adapt itself to different topological structures.

Chapter 5 explains a configuration scheme which explores an unknown network of transputers. This is useful in confirming that the transputers have been connected in a particular configuration as required for some particular task and that they are all working properly.

Chapter 6 describes the simulation of Malek's comparison algorithm [Male80]. We also discuss how other algorithms can be implemented in this simulator. We further present some results of applying Malek's algorithm to a selected set of multiprocessor structures.

Chapter 7 presents our conclusions and future work.

Appendix A shows the source code for the simulator along with the input and output files.

Appendix B shows the results of applying the simulator to detect a fault in a network of transputers. We also show how the simulator can be applied to an alternate configuration of multiprocessors.

Appendix C illustrates the configuration program which is used to trace an unknown network of transputers.

CHAPTER 2

BACKGROUND

We present in this section some background information on the transputer architecture and the model of our example network. The transputer fits well the requirements of this simulator and will be used as an example processor.

A reconfiguration scheme based on a generalized switched lattice is also briefly discussed to show how a transputer based multiprocessor can move from one configuration to another. This is a type of function which would be of value for this simulator.

A comparison method introduced by Malek for fault diagnosis of multiprocessor systems using a graph theoretical model is also discussed. Given a system of 'n' units modeled by a linear graph, one can locate the faulty unit using this algorithm. The minimum number of comparison edges and test cycles required for fault detection is given by two of Malek's theorems and can be used for the efficient application of this algorithm in a complex network.

2.1 The Transputer

A transputer is a microprocessor designed for efficient concurrent execution. This high performance is obtained by reducing the overhead involved in task switching and by high bandwidth interconnections. The transputer has only a few registers and

two predefined queues for processes with two priority levels. In addition, the transputer instruction set is small, which also accounts for its high performance.

The transputer implements the model of interprocess communication defined by the Occam language [Poun86], which is based on the CSP notation [Hoar78]. A unique feature of the transputer is that its I/O hardware links function as communication channels, i.e. the four serial links provide a path for message passing. Each transputer has four serial bidirectional links (as shown in Figure 2.1) with a byte-oriented protocol. This allows to use them as building blocks by interconnecting them in a regular structure. This simple interconnection scheme is provided by a simple link hardware protocol which is common to all members of the transputer family. The transputer manufacturer, INMOS, provides an off the shelf line of link adapters which allow to interconnect transputers with other devices.

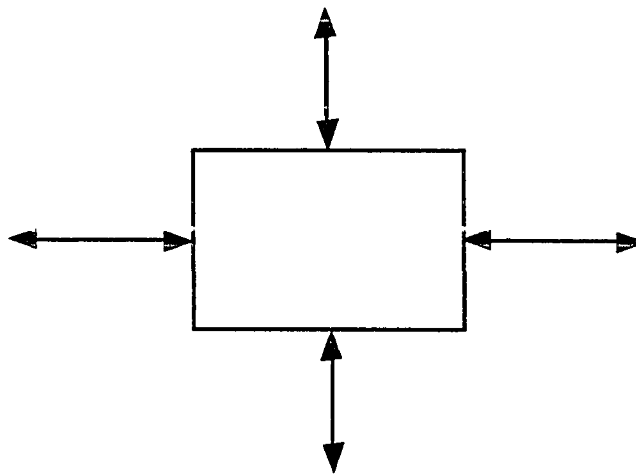


Figure 2.1 Transputer with four bidirectional links

2.2 The Occam Programming Language

Occam is the transputer's programming language. The choice of the features in Occam has been motivated by the need to support many communicating processes to perform a common task. Occam enables a system to be described as a collection of concurrent processes, which communicate with each other and with peripheral devices through logical communication channels.

2.2.1 Occam processes

Writing an Occam application model begins by describing some problem as a collection of tasks or events. A task or process is a program component that is executed asynchronously. Occam programs are built from three primitive processes:

v := e	Assign expression e to variable v.
c ! e	Output expression e to channel c.
c ? v	Input from channel c to variable v.

These primitive processes combine to form constructs:

1. Sequential (SEQ)

The statements following this construct are executed one after another.

2. Parallel (PAR)

All components in the scope of this construct are performed concurrently. The construct terminates when all constituent components are executed.

3. Alternative (ALT)

In Occam programming, it is sometimes necessary for a process to receive an input from any one of the several other component processes. For this purpose Occam includes an ALT construct. Each component of an ALT construct starts with a guard.

The guard is an input possibly with a boolean expression. The earliest process which satisfies its guard condition is executed first. If two or more processes satisfy their guard condition then either process is executed first. The choice in this case is arbitrary.

4. Conditional (IF)

This construct is followed by a condition. If the condition is true, the primitives encompassed by the construct are executed.

5. Repetition (WHILE)

A condition follows the WHILE and the primitives encompassed by the construct are executed until the condition is false.

2.2.2 Occam channels

Message passing has been adopted in Occam for process communication through the use of channels. Communication in Occam occurs when one process names another as destination for output and the second process names the first as source for input. When this happens the output values are copied from the first process to the second. The transfer of information occurs only when both the source and destination processes have invoked the input and output commands respectively. This implies that either source or destination process may be suspended until the other process is ready with a corresponding input or output. Thus the communication facility of Occam serves as a synchronization mechanism. At the execution level the transputer reflects the structure of the Occam language. The transputer is used to model Occam processes and the interconnecting links are used to model Occam channels and vice versa. One can have an arbitrary number of logical channels but for communication between processes

on different transputers, the maximum number of channels is four since there are only four physical channels.

2.3 Fault Tolerance

A basic requirement of fault tolerance is redundancy. In fault-tolerant designs redundancy is used to provide the information needed to mask out the effects of failures. Redundancy is achieved through additional time, information or components. One form of time redundancy involves extra executions of the same calculation, perhaps by different methods. Comparisons or other operations on the multiple results (identical when no errors are present) provide the basis for subsequent action. Time redundancy is usually provided by software. Component redundancy is aimed at providing continued service even when some component units fail and also constitutes the basis for certain forms of fault detection, e.g. comparisons. Component replication can occur at many levels in a system, e.g. circuit level, gate level, logic unit level and even at higher levels such as buses, memory subsystem, processors, etc.

Hardware redundancy usually takes the form of dual-duplex configurations, triple modular redundancy (TMR), or N-modular redundancy (NMR) voting schemes. In addition, other schemes are also available, e.g. reconfigurable NMR. These schemes provide a fine granularity for fault detection and isolation.

Multiprocessors are designed with various degrees of coupling, so we find for instance tightly coupled systems in which interprocessor communication takes place over a common global memory area and loosely coupled systems in which processors

communicate among themselves by sending and receiving messages. One aspect of fault tolerance is easy to achieve in a loosely coupled system because messages are an explicit way of communicating among processors and processes. In addition, loosely coupled systems are not subjected to certain single point of failures such as failure of a global bus, global memory states, etc. For these reasons, there is interest on fault tolerant systems based on multiple processors and loosely coupled schemes. In addition a large amount of current research is dedicated to this field for the following reasons:

* A message is an explicit form of communication, and makes it easier to provide error detection, confinement, and recovery.
* Synchronization is not dependent on low level features (e.g. locks, indivisible operations which are costly and difficult in multiprocessor environments)
* Message passing is the choice of synchronization and communication between objects and object oriented programming is a methodology which is becoming more and more popular.

2.4 Reconfiguration

There has been a considerable amount of research going on in the area of reconfigurable systems ([Kart78], [Kung84], [Yala85]). Reconfiguration may be needed in a system for various reasons. It could either be for reallocation of processors, for efficient processor utilization, for generation of a new topology that matches a certain algorithm, or to achieve fault tolerance. We consider the problem of reconfiguring a multi-microprocessor system in order to adapt to a new topology. We have chosen a transputer as the microprocessor for the reasons mentioned in Section 2.1.

The switch lattice approach proposed by Snyder [Snyd82] is two-dimensional. In practice, however, higher dimensional configurations are often desired.

2.4.1 Construction of an initial configuration

An initial configuration must be easily reconfigurable. Change of the dimension should not be very complicated. Further initialization must be easy to perform. From all these considerations, an n-cube is considered to be a good choice for the starting configuration of a network of processors. Thus, the initial switch lattice is an n-cube as shown in Figure 2.2. There is a transputer on each node and there is a switch between any two adjacent nodes. The switch itself has memory to store the connection information which is a pattern of 1's and 0's to indicate the switch's on and off states.

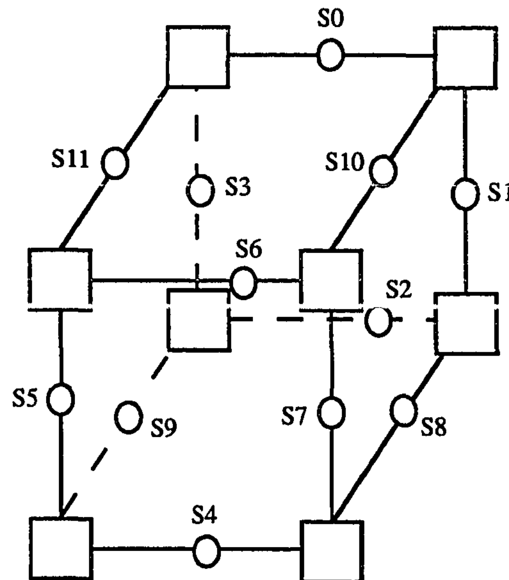


Figure 2.2 Generalized switch lattice

The n-cube can be reconfigured either into a one dimensional structure or into a variety of higher dimensional structures depending on the application requirement. All this can be done by turning the right switches on and off, which, in turn, can be done through loading the switches memory by an external controller. Since the switches work collectively towards an objective with certain configuration, they should be syncronized.

2.4.2 Achieving a given dimension

One of the powerful properties of the generalized switch lattice is the ease with which the dimension can be changed. A user can get the configurations of dimension ranging from 1 to n. As an example a 3-cube can be easily changed to a two dimensional topology by cutting down some edges, that is, by turning all the corresponding switches off. Similarly, one can construct configurations of any dimension. The parameters which a user should provide are the dimension and number of processors needed. A condition has to be satisfied regarding the dimension and the number of processors, i.e.

$$n \geq 2^d, \text{ where}$$

d denotes the dimension and n is the number of processors.

To generalize the condition, a single processor is defined as 0-dimensional,

(a) If $n < 2^d$, the controller issues an error message

(b) If $n = 2^d$, no dimensional change is needed

(c) If $n > 2^d$, the initial state needs to be reconfigured.

An example to illustrate the above conditions is explained with the help of Figure 2.3. By turning switches S1 and S3 off, a 3-cube (a three dimensional structure) can be changed to a two dimensional array. Similarly if switches S2 and S6 are turned off a linear array can be formed as shown. The parameters chosen in this example are n = 8 and d = 3. In general terms if $n = 2^d$ and we have a m-cube machine, m > d, one can have two or more task sets in operation concurrently. This is applicable to any subset of processors.

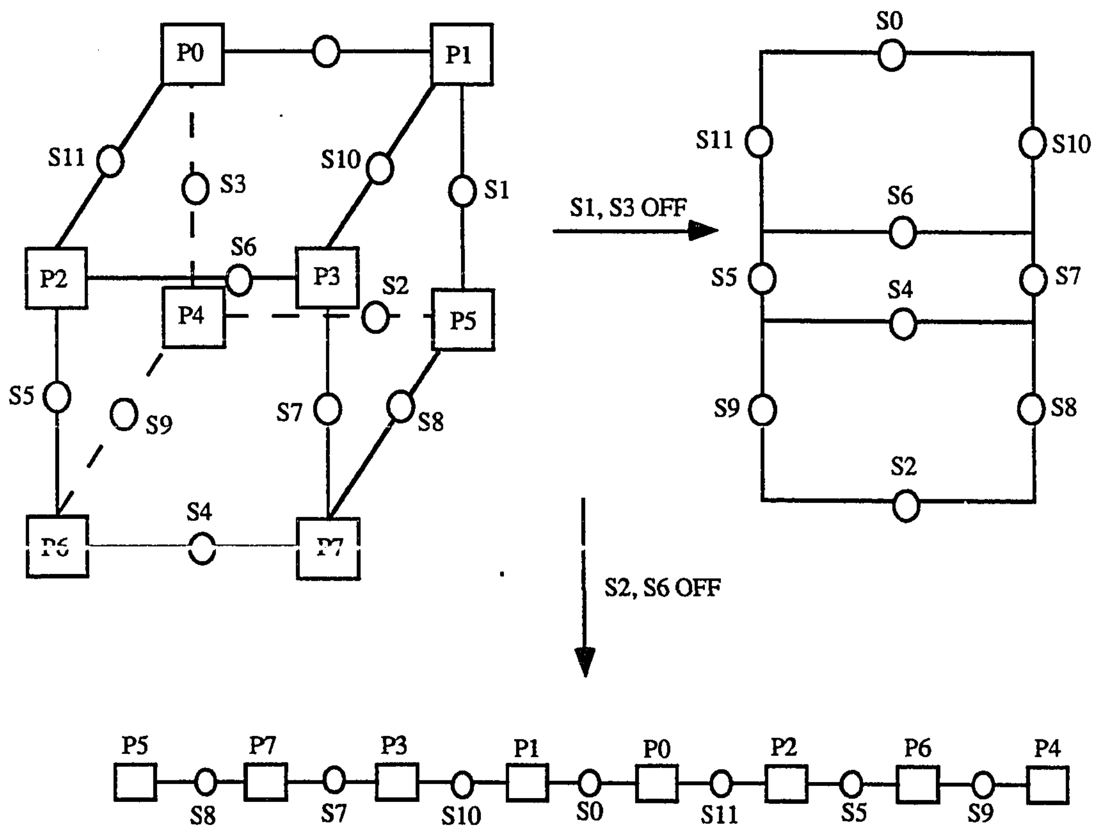


Figure 2.3 Linear array from a 3-cube

2.5 Malek's Comparison Model

Various approaches have been proposed for self-diagnosis of multiprocessor systems [Liu80], where the Preparata, Metze and Chien (PMC) model [Prep67] is the most classical one. Examples of the comparison method were shown by Toy [Toy78] and DeGonia [Degon78]. Malek [Male80] expanded the idea and introduced a comparison connection assignment for fault diagnosis in multiprocessor systems, where a pair of units is assumed to be compared by another unit.

Malek's method takes advantage of the homogeneity of multiprocessor systems in which comparisons can be made easily. A comparison is performed such that a processor, chosen to be a comparator, monitors a pair of processors executing the same test input and compares their responses. Any mismatch during the comparison period indicates some failure in the set of three processors.

A multiprocessor system is modeled by an undirected graph G(V,E) where V is a set of vertices that correspond to processing units and E is a set of bidirectionsl communication links. Each pair of processors (v_j, v_k) is tested by a processor v_i, by comparing their outputs. A comparator is any unit v_i in the system which compares a pair of units v_j and v_k during the test cycle and forms a comparison edge $C^i_{j,k}$ through two communication edges e_{ij} and e_{ik} as shown in Figure 2.4. A mismatch indicates a fault in either v_j or v_k. A set of tests can be described by a graph G(V,C) where C is a set of comparison edges $C^i_{j,k}$ defined as

$C^{i}_{j,k} = 0$ if v_i, v_j and v_k are fault free

$= 1$ if v_i is fault free and either v_j or v_k is faulty

$= X$ (don't care) if v_i is faulty

A comparison model is illustrated in Figure 2.5 which shows a graph G(V,E) and its corresponding graph G(V,C).

A basic assumption in Malek's model is that a faulty processor performs all of its assigned tasks incorrectly and faults are permanent.

Malek's algorithm is appropriate for single fault diagnosis. We can also find the number of comparison edges required and the number of comparison cycles to detect a fault in a network of processors. These two parameters are respectively denoted by q_l and c_l.

The case of fault detection is straightforward. In order to detect whether there is a faulty unit in the system, every unit should be compared to some other one. In an ideal case, if there is an even number of units and there are connections sufficient to cover every pair separately by a single link, the number of required comparison edges is equal to n/2. If the number of units is odd, then the obtainable minimum equals the upper approximation of n/2.

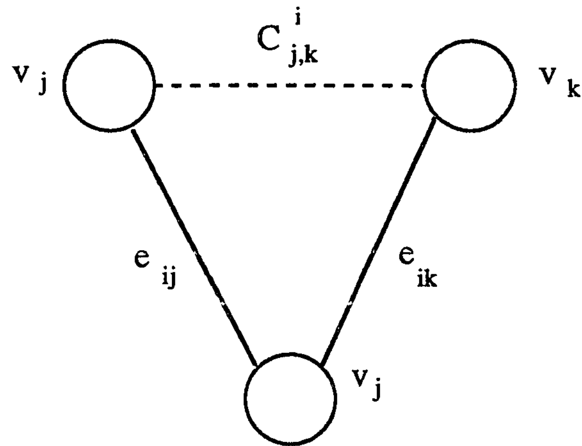


Figure 2.4 A Comparator and two compared units

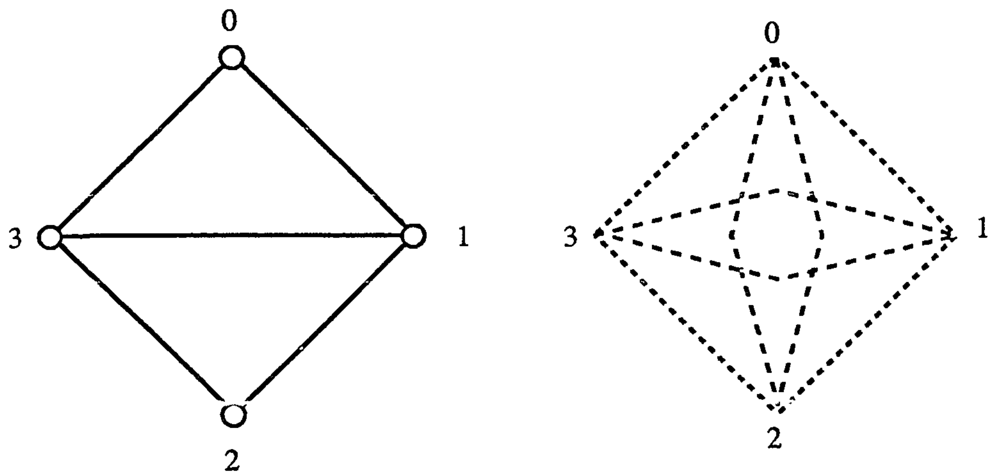


Figure 2.5 Comparison model graphs

The upper and lower bounds for the number of comparison edges (q_l) in order to locate any fault in a multiprocessor system are [Male80] :

$$n - \left\lfloor \frac{n}{3} \right\rfloor \leq q_l \leq n-1$$

Also the bounds for the number of comparison cycles (c_l) needed to detect a faulty unit are

$$\left\lceil \frac{n+1}{2} \right\rceil \leq c_l \leq n-1$$

where n is the number of units in the system being analyzed.

2.6 Summary

This chapter explains the transputer architecture and some fault tolerance aspects of a multiprocessor system. We explain briefly the Occam language and its relevance to the transputer. A reconfiguration scheme for a multiprocessor system to adapt itself to any topological structure is also discussed. The switched lattice approach is used to reconfigure our network of multiprocessors into two different topologies. We also discuss Malek's comparison model to detect a faulty unit in a network of processors.

CHAPTER 3

A SIMULATOR FOR MESSAGE-PASSING MULTIPROCESSORS

In this section we present a description of a simulator for message-passing multiprocessors. The processors are restricted to have at the most four interconnection links. The interconnection network can be of any type and can be expanded to accomodate any number of processors. We can simulate faults and use algorithms to detect a faulty node in the network. In later chapters we discuss additional functions not yet implemented.

3.1 Organization of the Simulator

The network simulator provides a functional simulation environment which considers the communication aspects of a multiprocessor system. Specifically, the simulator is illustrated using a 3 X 3 network of nine nodes connected in a toroidal configuration. This arrangement can describe many interconnections of computer nodes by changing the link configurations through some input parameters for the program.

A processor is simulated as a collection of independent cooperating tasks. For the programming of the simulator, the Ada language was considered to be a good choice because of its concurrent facilities and structured approach [Booc83], [Buhr84]. The major relevant features of Ada are:

* concurrent programming : to express the concurrency that exists in a transputer

network (each node is able of independent execution) and to express the concurrency that exists within a transputer (the processors and the links work concurrently). In this way each component able of independent execution can be simulated by a separate task.

* communication : to express the information transfers between the transputers of a network, between the memory and the processor in a node.
* error handling mechanisms in order to deal with predefined system error (arithmetic errors) and simulation errors(deadlocks, etc.)
* visibility features: Ada allows several ways to combine packages, including packages at the same level that can see each other, packages that depend on other packages, and nesting of packages. The required structuring depends on the needs about type and function visibility, as well as other considerations such as sharing of objects , reusability, performance, and readability.

Originally we intended to design all the component tasks separately (links, communicators and applications) and then instantiate them under a common dummy task; however, Ada made this approach difficult because of visibility constraints. For this reason all component tasks of the processor are implemented separately within the same procedure at the same level so that they can see each other. Thus a processor is described by a collection of tasks.

Each processor task is simulated by the following tasks (Figure 3.1):

1. Four link tasks.
2. An application task.
3. A communicator task.

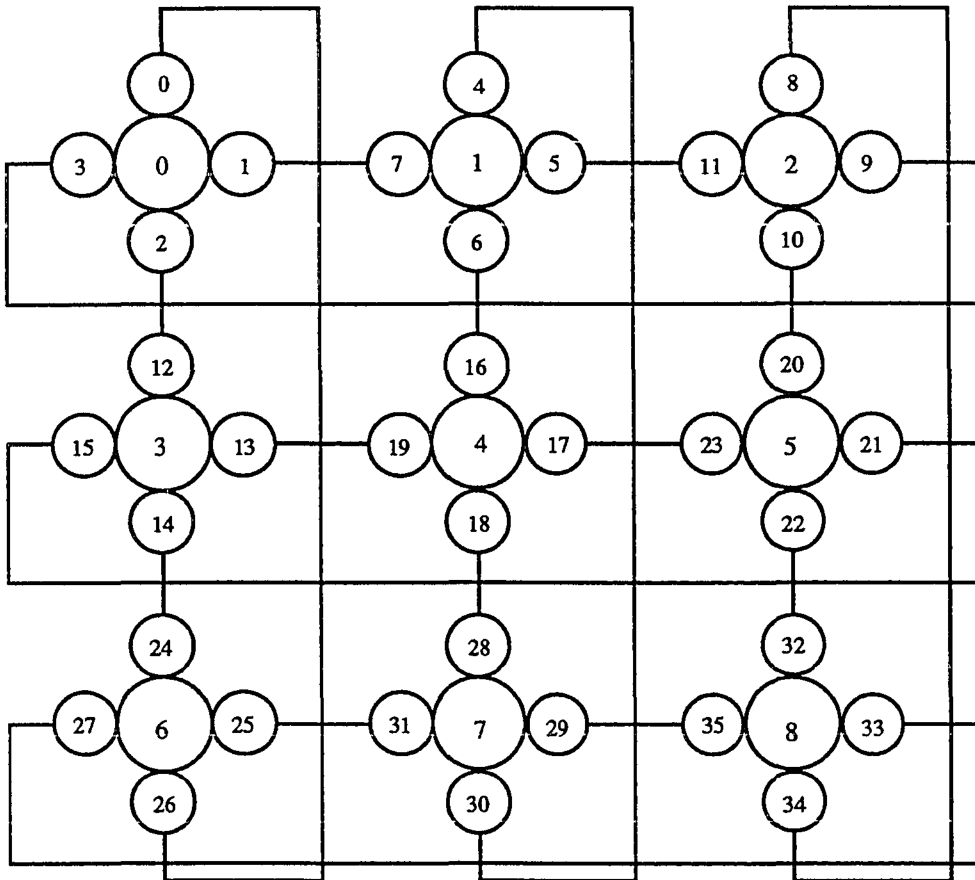


Figure 3.1 Toroidal configuration of nine nodes

The link task simulates the communication characteristics of a link. The basic link operation is to transmit or receive requests. The application task is an abstraction of a processor's computational activity. The communicator task provides the basic task type to support the protocol communication and error detection primitives.

These three tasks correspond to components that are able of independent execution and that are of significance for the objectives of this simulation.

3.2 Link Task

This type of task consists of three entries: a CONFIGURE_LINK entry which is required by Ada to be able to identify the task and two entries SEND and RECEIVE which allow the bidirectional sending and receiving of packets that represent messages. The messages that we pass between different nodes are in the form of packets. Since we are not studying the effect of data types each packet just contains integers in our case. The link configuration of the grid is defined in a separate input file which defines the link interconnections among the different nodes.

3.3 Application Task

The application task receives messages from other nodes to transmit to other application tasks residing in different nodes. We have made the simulation user friendly so that as the message travels from a source node to a destination node the user gets informed at every stage about the status of the message and the path via which it traverses.

3.4 Communicator Task

The communicator task performs various operations on the packets. It provides the basic task type to support the protocol communication in the simulation environment. It builds its own database to calculate the offsets to be used in routing the packets. The offsets determine which adjacent links are to be used by the packets for their routing. The communicator task contains three entries. The first one is similar to the one explained above in the link task, i.e. a CONFIGURE_LINK. Two other entries are a TRANSMIT entry and a RECEIVE entry. The TRANSMIT entry is used by the application task to request transmission of a packet to another node in the grid. The RECEIVE entry is used by the corresponding link tasks to pass a received packet to the communicator task residing in another node.

3.5 Implementation of the Tasks

3.5.1 Link task

The link task is a generic processor link. This task identifies the node to which a particular link is connected. It also recognizes the links of the nodes to which a particular link is connected. Functionally, it sends a packet to the communicator task of another node.

3.5.2 Application task

The application task represents a user defined computational activity. The application part within each node in the multiprocessor system simulates a computational activity. We create an input file in which requests for the various nodes can be inserted by the user. In this case the request is in particular a user defined

activity which happens to be the SEND_TO operation. The SEND_TO defines the source node and the destination node identity for a message to pass between any two processors. A job is an intended action requested by the user. Requests consist of an originating node number, a destination node number and the type of job (currently only send_to is implemented). Jobs are then read from the input file and stored in the form of a FIFO queue for each node. There could be various requests in the input file for different nodes.

When an application task decides that the job obtained consists of a request to send a message, it partially formats a packet and sends it to the communicator task. On the other hand, if the application task receives a packet, it displays a message on the terminal indicating that it has received a packet and prints its contents. This allows to verify that a packet has reached its final destination. The packet usually has to traverse intermediate nodes unless the destination node is adjacent to the source node. When it moves through the intermediate nodes the message displays the node information and the contents of it in a similar fashion as it would do for a destination node.

3.5.3 Communicator task

When a packet is passed to the communicator task through its TRANSMIT entry, this task applies some routing functions (described in the next section) to determine through which of its attached links the packet should be sent to another node. When a packet is passed to the communicator task through its RECEIVE entry, the communicator task finds out if the packet is for the current node in which case it passes it to the application task. If the packet is not for the current node, i.e. the destination

node identifier is different from the current node identifier, it then performs additional routing functions to determine to which of its attached links the packet should be forwarded.

3.6 Communication Operation

A packet is described by a record specifying the header and data portion of the packet. When a packet to be sent to a destination node is originated in a node, the communicator task attempts to find a route for it. For this purpose it invokes a procedure, which determines through which of its links the packet should be sent. This packet is then received at one of the adjacent nodes through the link attached to the originating node.

When a packet received by a node is not directed to this node, i.e. the destination field in the packet is different from the current node_id, it is forwarded to another node using a local routing function. This routing function provides a check to confirm that the packet has indeed reached the destination node.

The following events take place in the simulator when a message passes from one node to another :

* A packet is received by a node and is stored in a buffer.
* The link of the receiving node notifies the task in charge of getting the received packet that a packet has been received. A direct call to the task communicator is used for this purpose.
* The communicator task stores the packet, performs some transformations on it,

i.e. it makes the necessary changes (described in Section 3.4) to continue the required processing.

* If the destination_id field within the packet is the current node, the packet is then passed into the application or any other specified destination task.
* If the packet is not for itself, i.e. the node_id of the current node is different from the final destination_id of the packet field, a forwarding function is invoked.

If the destination node is in the same row of the network as the current node (the one which just received the packet), a function is invoked to determine if the destination node is to the right of the current node (a higher column number) in which case the packet is forwarded to the right through the right link. If the packet is for a node located to the left of the current node (a lower column number), the packet is forwarded to the left using the left link of the node. If the current node and the destination node are not aligned (neither in the same row nor in the same column), a predefined routing policy is used to forward the packet. This could have been any random direction. In our case we send the packet to the bottom.

Thus we design an algorithm for the routing of packets between intermediate nodes. This, however, is a centralized algorithm for the whole network and cannot be applied locally to a specific node.

The pseudo code for the routing algorithm is as follows:

Find destination for the packet -- i.e. read destination field

if in the same row of the grid

```
    then
    if to the right
        then forward_to_right
        else forward_to_left
if in the same column of the grid
    then
    if above
        then forward_to_top
        else forward_to_bottom
if not in the same row or column
    then
    if in row above
        then forward_to_top
        else forward_to_bottom
```

3.7 Node Database

It has been mentioned in the previous sections that routing takes place in the communicator task. For this purpose some routing procedures can be applied. However the final link assignment for routing purposes is achieved by using the node internal database.

A node database is shown in Figure 3.2. It consists of a list of the link connections (the hardware connection is determined by the input parameters) and a routing function.

Application task
Node identifier
Task communicator entry

Communicator task
Node identifier
Link up identifier
Link right identifier
Link down identifier
Link left identifier
Application entry
Link entry

Task link up
Link identifier
Partner link identifier
Communicator task
Link entry
Communicator entry

Task link left
Link identifier
Partner link identifier
Communicator task identifier
Link entry
Communicator entry

Figure 3.2 Node database

3.8 Parameters Governing the Network Architecture

The parameter which decides the link connection map is the file oriented user interface. In other words, this input file describes the link interconnection map for a multiprocessor network.

The link connection map for the example network is illustrated in Table 3.1. Every node has four links and the links are numbered in a cyclic fashion. The first column lists the nodes to which the links are attached. The second and third columns describe the link interconnection structure for the network. The following example will help understand the map in a more precise way. The first entry in the first row describes that link number 0 (second column) of node number 0 (first column) is attached to link number 26 (third column). Link number 26 happens to be a serial link for node number 6 as one can see from row number 27.

Row_no.	Node_no.	Link_no.	Link_no.
1	0	0	26
2	0	1	7
3	0	2	12
4	0	3	9
5	1	4	30
6	1	5	11
7	1	6	16
8	1	7	1

9	2	8	34
10	2	9	3
11	2	10	20
12	2	11	5
13	3	12	2
14	3	13	19
15	3	14	24
16	3	15	21
17	4	16	6
18	4	17	23
19	4	18	28
20	4	19	13
21	5	20	10
22	5	21	15
23	5	22	32
24	5	23	17
25	6	24	14
26	6	25	31
27	6	26	0
28	6	27	33
29	7	28	18
30	7	29	35
31	7	30	4

33	8	32	22
34	8	33	27
35	8	34	8
36	8	35	29

Table 3.1 Link connection map.

To implement a different architecture, this link map will have to be modified according to the specifications of the network. The user would know the link interconnection information, i.e. the individual link identifications of a pair of nodes to be connected. The user would need to edit the link interconnection map for a different architecture.

Furthermore, this simulator can be extended to implement any number of nodes. For the example network, we declare two arrays which define the number of nodes, viz.

```
application : array (0..n) of application_task;
communicator : array (0..n) of communicator_task;
```

The elements in these arrays define the number of nodes. The user can change these values to accomodate any number of nodes.

The source code for the simulator is illustrated in Appendix A. Our simulation can be applied to any configuration. Two examples are tested and the results are shown in Appendix A. In example 1 we consider a toroid of nine processors and in example 2 we consider a mesh connected structure as the one shown in Figure 3.3.

3.11 Summary

This network simulator can be used to test fault detection algorithms. The network simulator is basically a tool which can be applied to any configuration, provided some of the variables and functions are altered. The simulator bases its operation on a message passing system. The simulator can be extended to include functions such as reconfiguration so that it achieves some fault tolerance aspects.

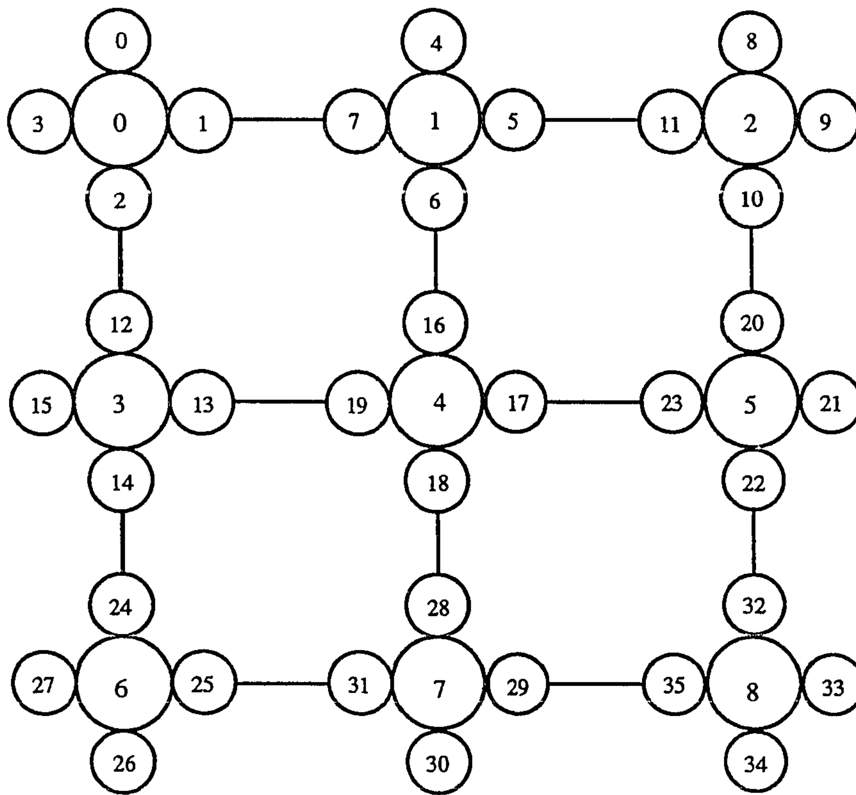


Figure 3.3 Mesh connected network

CHAPTER 4

RECONFIGURATION IN A MULTIPROCESSOR ENVIRONMENT

To enhance the flexibility of the simulator the next step would be to consider a reconfiguration scheme to generate a new topology. By reconfiguring a network of processors one can achieve an efficient way of executing a particular algorithm which needs a particular architecture. Also if a node fails in a network, then the network has to be reconfigured for it to still function properly. One could isolate the faulty node and keep the network running with the remaining nodes. The current network of processors can be reconfigured into different architectures by a method suggested by Snyder [Snyd82].

In this Chapter we explain a methodology to transform a multiprocessor configuration to adapt to four different topological structures. It bases its operation on the switched lattice discussed in Chapter 2 (Section 2.3).

4.1 Reconfiguration of a Network of Nine Processors

The simulation which we have developed can be applied to a specific configuration as shown in Figure 4.1. This is basically a toroidal mesh of nine transputers. Every processor in the system has four serial bidirectional links which are used to connect to other processors in the network. We apply Snyder's switch lattice method to adapt this configuration to four different topologies. We can implement this

approach in our simulator by defining a separate switching task which would control the switching action for the various switches between any pair of nodes. In other words the switching task would execute the functions of the external switching controller.

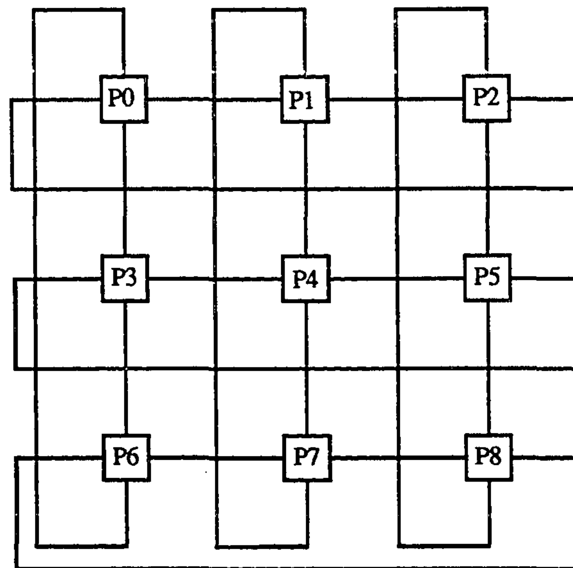


Figure 4.1 A toroidal mesh of nine processors

A switch is inserted in each of the links connecting the transputers in the toroid. The function of the switches is to turn the links on and off. By doing so various configurations can be formed. The switches are controlled by an external controller which decides when they are to be turned on and off. Each switch has memory to store the connection information which is a pattern of 1's and 0's to indicate the switch's on and off states. Figure 4.2 shows the toroid with the switches.

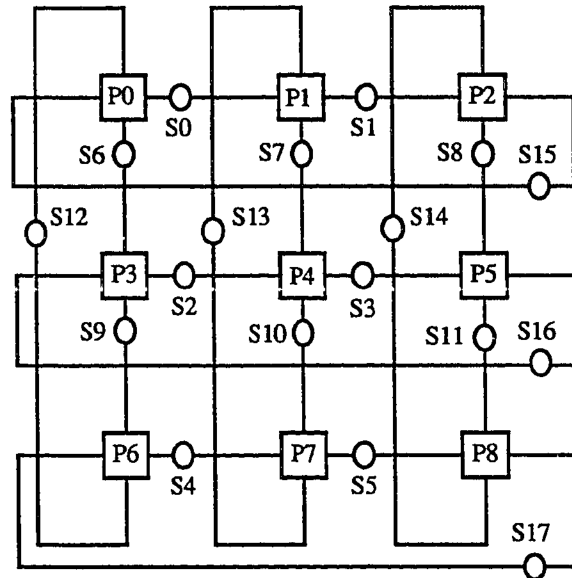


Figure 4.2 A toroid with switches inserted

4.1.1 Transforming a toroid into a three dimensional prism

As explained previously we have nine processors in a two dimensional array. So, n = 9 and d = 2, where n and d are the number of processors and the dimension respectively. Thus we encounter the condition $n > 2^d$ (Section 2.3.2). Thus the initial configuration, a toroid in our case, can be reconfigured. By turning off switches S12, S13 and S14 the toroid gets transformed to a three dimensional prism. The transformation is shown in Figure 4.3.

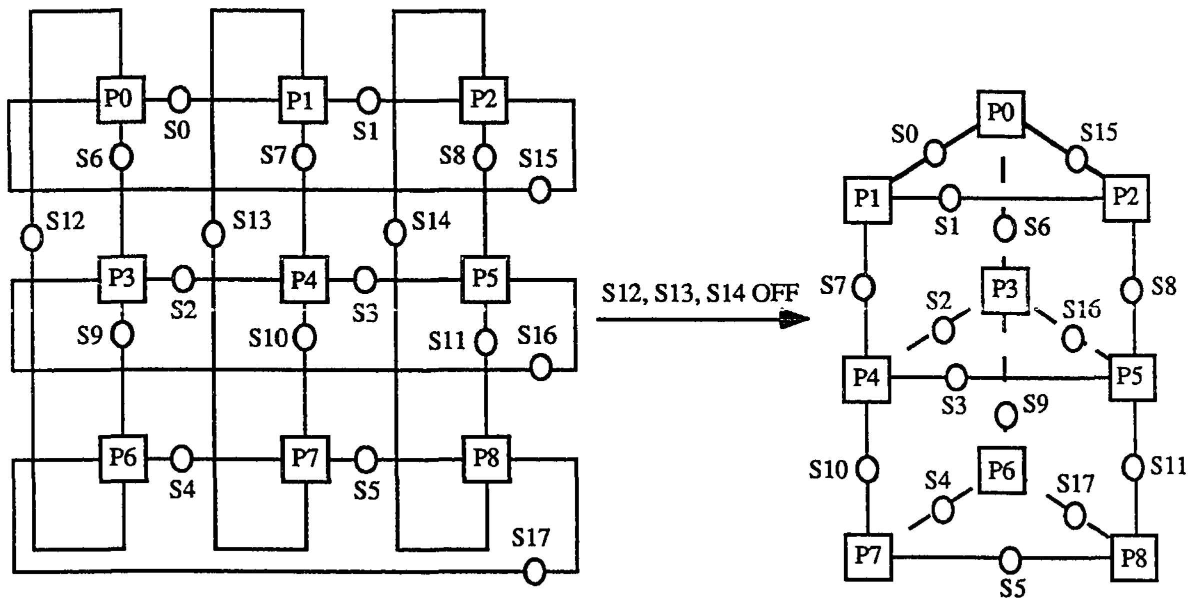


Figure 4.3 Transformation of a toroid into a prism

4.1.2 Transforming a three dimensional prism to a two dimensional mesh

In the prism we have nine processors but the dimension has changed. The parameters n and d will hence change. Now n = 9 and d = 3. Thus now our initial configuration which is a prism can be reconfigured according to the condition $n > 2^d$. We turn off the switches S0, S2 and S4. This converts the prism to another two dimensional structure which is an ordinary mesh as shown in Figure 4.4.

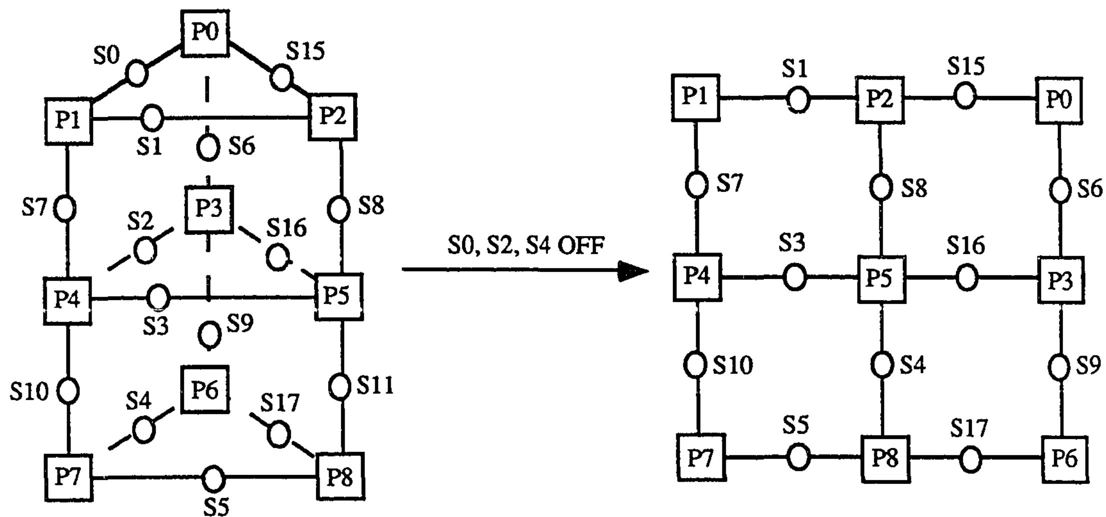


Figure 4.4 Transformatuion of a prism to a two dimensional mesh

4.1.3 Transforming a two dimensional mesh to a star configuration and a linear array

The two dimensional mesh has the same parameters as those of a toroid. Now we treat this new configuration as our initial state. We come up with two different topologies by turning of a certain combination of switches. When switches S3, S6 and S16 are turned off , a star configuration and a linear array of processors is formed as shown in Figure 4.5.

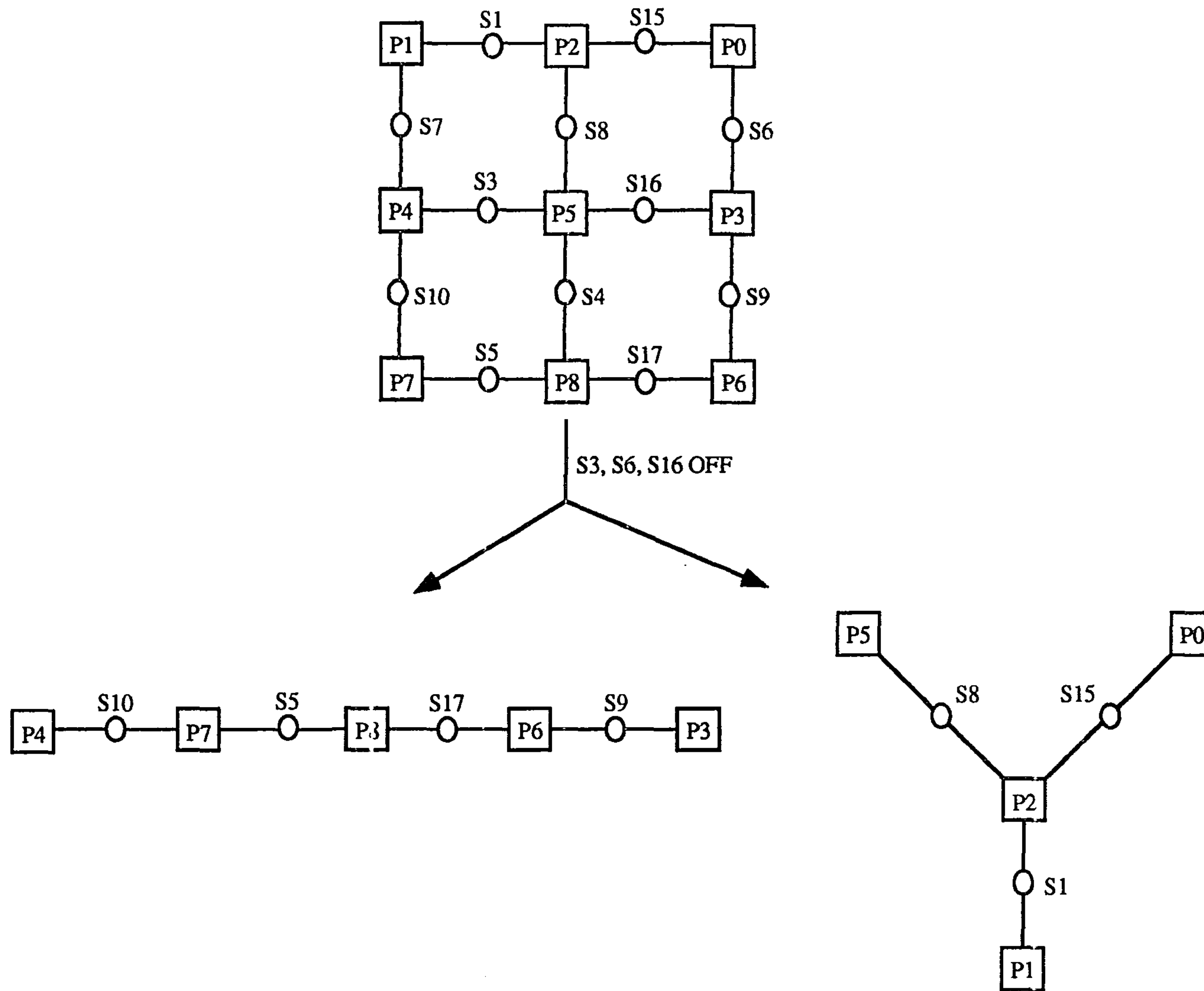


Figure 4.5 Transformation of a mesh to a star and linear configuration

4.2 Summary

We describe a way of reconfiguring our network by the switch lattice approach. However, in this method the most complicated part is the time complexity of changing from an initial state to another topology. The controller has to find out which switches are to be turned on and off at any instant of time. There should be certain rules to be followed in order to achieve the syncronization of the switches. We can implement this reconfiguration scheme in the simulator by representing suitably the switches and the controller.

CHAPTER 5

CONFIGURATION OF A GENERAL NETWORK

A transputer is a microprocessor which can be easily connected to form networks in multiprocessor arrays. These arrays can be large and complex. We develop a program which explores an unknown network of transputers and determines its configuration. This is useful in confirming that the transputers have been connected in a particular configuration, as required for some particular task, and that they are all working properly. The exploration is achieved by a program which will find its way around the network, exploring all the links on all the transputers to determine the interconnections. The program can also be used to load code into a network whose configuration is not known in advance. These functions could be included in the simulator to increase its functionality and usefulness.

5.1 The Structure of a Tracing Program under the TDS

The transputer development system (TDS) recognizes two different types of programs, known as **EXE** and as **PROGRAM**. An **EXE** program runs on the host transputer, and may access the keyboard, the screen and the filing system of the host machine. A **PROGRAM** runs on a network of one or more transputers and is loaded from the host transputer via a transputer link. An example of such a system is shown in Figure 5.1

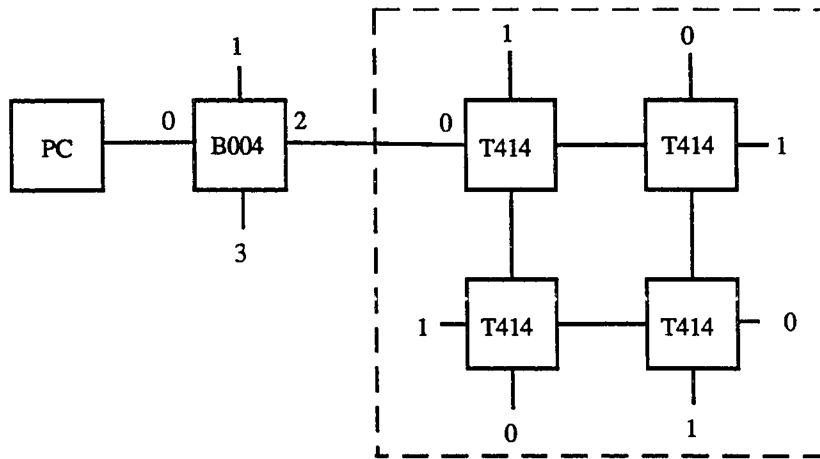


Figure 5.1 Transputer development system

An IBM PC-AT is connected to a B004 evaluation board which has a single transputer on it. This transputer acts as the host processor for developing programs and for loading multiple transputer networks. Link 2 of this evaluation board is connected to an INMOS B003 evaluation board which has four transputers on it. When a program is loaded onto a multiple transputer network an **EXE** program will be run on the host transputer which monitors the output transmitted back from the **PROGRAM** and then interacts with the PC to display the results. An example of a **PROGRAM** running on a single transputer is as follows:

```
{{{  PROGRAM  Example
{{{F
...  SC  Example
PROCESSOR  0  T4
    Example  ()
}}}
```

```
}}}
```

When the above lines are compiled and extracted, a new fold is created:

```
...F  CODE  PROGRAM  Example
```

"..." denotes the Occam folds. These folds can be used to accomodate a hierarchical set of program elaborations. Each fold can contain program statements or other folds. The open and close folds are denoted by "{{{" and "}}}" respectively.

This `CODE PROGRAM` fold will initialize and load a single transputer and run the `SC Example`. Now if an occam byte array `Program` contains the contents of `CODE PROGRAM` fold, then the effect of

```
ToLink ! Program
```

is to load and run the program on a transputer connected to link `ToLink`. The exact way in which a transputer loads the code is described in [Inmo88].

A tracing program searches a network of transputers as follows:
Suppose a transputer is already executing a trace program, and that it is connected to another transputer which is not yet been loaded. The first transputer, which will be called the 'parent' loads the second ('child') by outputting the code `Program` as explained above. It then sends `Program` a second time, which the child stores as a byte array in memory. The child is now in a position to load other transputers until the entire network is loaded.

To achieve this, the trace program is made of two parts:

```
... EXE Host       - Runs on the host transputer
```

```
... PROGRAM trace - Searches the network
```

The Host **EXE** reads the **CODE PROGRAM** trace fold, and stores a byte array **Program**. After initializing the network, it loads the program onto the first tranputer in the network by outputting **Program** on a suitable link. As the trace program searches the network, the program running on the host transputer processes any data returned to it from the trace program, interpreting and displaying the results.

5.2 The Host Transputer

The program (**EXE**) which runs on the host transputer looks like this:

```
SEQ
    code.fold.reader  (Screen,from.user.filer[0],
                       to.user.filer[0],programTable,
                       programLength,errorFlag)
    IF
      errorFlag
          SKIP
      TRUE
        SEQ
          ... Determine which link to examine
          ... Reset subsystem, links

          -- Main Section
          VAL Program IS [programTable FROM 0 FOR
```

```
                    programLength]  :
        PAR
            Tracer(LinkIn[linkNumber],
              LinkOut[linkNumber],ToInterface,
              linkNumber,Delay,Program)
             Interface(ToInterface,Screen,Heading,
            linkNumber)
          ...  Display  and  file  output  using  std.
procs
    write.full.string  (Screen,  "*C*NType  <any>  to
continue")
    Keyboard ? word
```

The process **code.fold.reader** provided in the trace program attempts to read a **CODE PROGRAM** fold which is already compiled. If an error occurs, the boolean **errorFlag** is set to **TRUE** and the cause of the error is displayed on the PC. It is assumed that the reset pins of the subsystem network are chained together, and controlled by the host transputer. In order to reset the transputers correctly, the reset pin must be held high for a sufficient amount of time.

The program asks the user which link of the host transputer is to be examined (**linkNumber**). The link which is connected to the subsystem must be specified. None of the other links will be tried during the course of the program. If two or more links are connected to the same subsystem, then only one can be tried. The other link(s)

will receive data from the subsystem, as the trace program searches. To keep the host transputer from getting interrupted, all the links are reset on completion of the program.

The channels `LinkIn`, `LinkOut` perform the functions of the transputer's serial links. This process attempts to load a transputer connected to link `linkNumber` with the trace program. However, there may be nothing connected at all, or the transputer connected may not have been reset, in which case the output will fail. If the output of the code `Program` is not completed within a certain period of time, then it is terminated and the link reset. If the code `Program` is successfully output from the link, booting a transputer, then `PROC Tracer` sends more data as described in section 5.3.3. The new transputer is given an identity number '0'. As the search proceeds, `PROC Tracer` relays data back from the network to `PROC Interface`. The `Interface` process has data which it receives from the `Tracer`.

5.3 The Exploratory Trace PROGRAM

5.3.1 Introduction

As described earlier the exploratory trace program is constructed as a `PROGRAM` fold which consists of a separately compiled process `SC Trace`. This is then extracted to produce a `CODE PROGRAM Trace` fold, which contains code to boot a transputer and run `SC Trace` on that transputer. The trace is structured as follows:

```
SEQ
        ... Read in copy of program, identify boot link
        ... Initialize
        SEQ I = 0 FOR Nlinks
            ... Try each link in turn
        ... Return control to parent
        ... Feed back final link information to parent
```

When **SC Trace** starts to run on a transputer, it first identifies which link is connected to its parent and inputs a copy of the program code, so that it too may boot other transputers.

After initializing various flags (which keep track of which links have been searched), the program selects a link and tries to send a search down the link, which may (or may not) be connected to another transputer. If the program does not receive any response, it will timeout and look elsewhere. Section 5.3.2 describes the way in which a transputer searches a link to test whether a neighboring transputer is attached. Section 5.3.3 explains this and shows how the program is loaded and run on the neighbor.

5.3.2 Searching a neighboring transputer

A transputer can check whether link **I** is attached to an unbooted neighboring transputer by using the write and read features of occam. A transputer may load a word of data at an address and then read it back as follows:

```
[4] CHAN OF ANY LinkIn, LinkOut :
PLACE LinkIn  AT 4 :
PLACE LinkOut AT 0 :
SEQ
   LinkOut[I] ! 0 (BYTE); Address; Data -- Write Data
   LinkOut[I] ! 1 (BYTE); Address       -- Read Data
   LinkIn[I]  ? word                    -- Data is returned
```

The Read and Write features are equivalent to a write and read constructs of a high level language. If the address specified exists in memory, then the word returned should match the data sent. A convenient address could be **MinInt**, the minimum 32-bit integer of a transputer.

5.3.3 Booting a neighboring transputer

After having determined that a link is connected to an unbooted transputer, a transputer loads a neighboring unbooted transputer by outputting the code **Program** as mentioned in section 5.1. The newly booted neighbor will first read in a copy of the program, and identify the boot link:

```
SEQ
  ALT I = 0 FOR 4    --  Determine  which  link  is
connected
                  -- to my parent
      LinkIn[I] ? programLength
      parentLink := I
```

```
    LinkIn[parentLink]  ?  [programTable  FROM  0  FOR
programLength]
    LinkIn[parentLink]  ? token;  loadingData
    loadingData[3]  := parentLink
    LinkOut[parentLink]  !  LoadingData.t;  loadingData
      LinkIn[parentLink] ? token  --  Synchronize.t  from
the  host
```

The parent sends the length of the program, which enables the child to determine which link is connected to the parent. The code **Program** is sent again, and stored by the child as a byte array for future use. The parent also sends a set of data which includes the parent identity number, the link attached to the child, and the number of transputers found so far, **nTransputers**. The child returns the data, with the link on which the child was booted.

The data returned by the child is referred to as **loadingData**. **loadingData** contains information useful to follow the path of the trace. Its four elements are, the identity number of the parent, the link which the parent used to boot the child, the identity number of the child and the link on which the child was booted. This array is transmitted back to the host transputer for display. The **Tracer** process, running on the host, acknowledges receipt of the **loadingData** with a **Synchronize.t** token, transmitted back to the new child.

5.4 Exploring a Tree of Transputers

We describe in this section an example which is traced by the algorithm. We specifically explore a tree of transputers. The algorithm can also be extended to search a network which has closed loops. This case is explained in section 5.5. An example of a tree of transputers is shown in Figure 5.2.

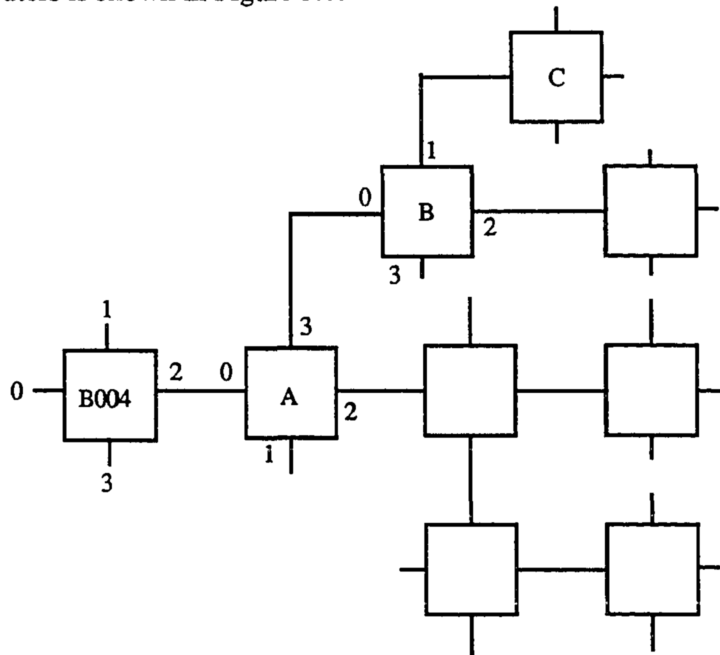


Figure 5.2 A tree of transputers

The trace program searches the branches of the tree sequentially. Excluding the host transputer, each transputer in the tree will be in one of the following states:

(R) reset but unbooted

(0) booted, but not yet searching its links

(1) searching a link, to see if there is another transputer connected

(2) Booting a neighboring transputer

(3) relaying **loadingData** to the host

(4) all links have been explored

The network is explored as follows:

Suppose that link 3 of transputer A has booted transputer B by link 0, and B has input a copy of the program from A. A enters stage 3, in which it will wait to transmit further data. Transputer B starts stage 1, searching one of its links to see if any other transputer is connected. Since link 0 is known to be connected to transputer A, link 1 is the first link to be searched. As described in section 5.3.2 the transputer attempts to write and read data to any transputer which may be attached to that link. The processor then waits for a word (**MinInt**), to be returned on input link 0, for a period of time, **Delay**, before timing out. If nothing is returned, the program assumes this link is unattached, and sets a boolean variable **downLoad[0]** to **FALSE**. The next link, link 2 is searched in a similar manner.

Let us assume that a transputer is attached to link 1, and that it has returned the value **MinInt** in response to the search. Transputer B now attempts to load the neighbor with code (stage 2), as described in the previous section. Let us call this new child 'C'. C determines its **parentLink**, the code **Program**, and **loadingData** (stage 0). It takes its identity number to be **nTransputers**, and increments **nTransputers** by one, where **nTransputers** is the number of transputers found so far (the third element of **loadingData**).

At this point, transputer B enters stage 3 of the program, and acts simply to pass on messages from C, even though it has not yet checked links 2 or 3. While transputer C explores its environment, B does not attempt to timeout link 1. Let us suppose that C is not connected to any other transputers. Having failed to find any neighbors, transputer C returns control to B, by sending the token **ReturnControl.t**, together with the latest number of transputers found so far. Transputer C then enters stage 4, and since it has tried all of its links, takes no further part in the exploration. B sets **downLoad[1]** to **TRUE**, to note that a transputer has been loaded from this link. Transputer B now returns to stage 1 of the program, and similarly tries link 2, and finally link 3. When all links have been tried, B returns control to A, together with the number of transputers found so far. The code for the algorithm is illustrated in Appendix C.

5.5 Exploring a General Network of Transputers

The algorithm described in the previous section is valid for a tree of transputers. In a real time system, however, the networks are more complicated than the tree structure. There could be closed loops of connections involving more than one transputer. An example of such a network is shown in Figure 5.3.

The basic algorithm is as before, but in addition there is a situation where a link is connected back to a transputer which has already been booted. This is solved by arranging for every transputer to look for all the links which have not yet been tried, (using an **ALT** construct).

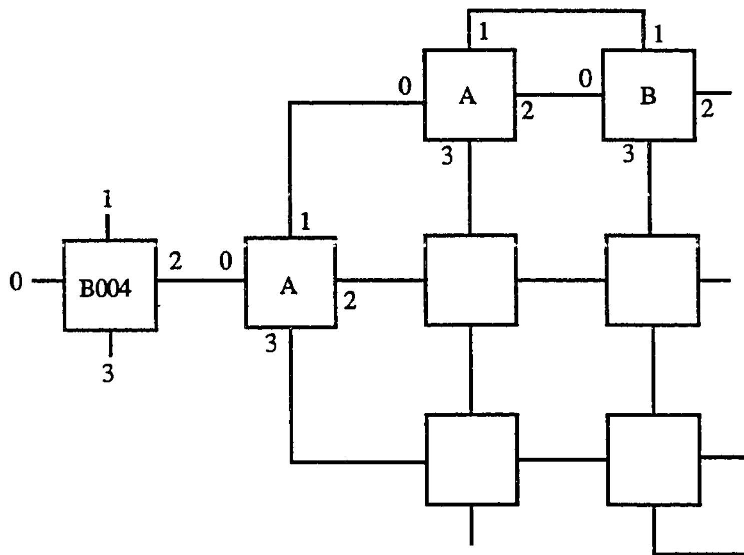


Figure 5.3 A closed loop connection

Suppose that link 2 of transputer A has booted transputer B on link 0, and is waiting while B explores further. B outputs the write and read sequence on link 1 which arrives back at link 1 of transputer A. It must now be arranged that A will recognize this sequence, even though it comes in on a different link to the one on which child B was booted. So A inputs the whole message and returns a token **AlreadyLoaded.t** which has a value different from **MinInt** in order to be recognized by B. In order that A does not try link 1 again later, a boolean **tryLink[I]** is maintained (initialized to be true). In our example, **tryLink[1]** is set to **FALSE**.

We can also build a link connection map which illustrates which links are connected to whom. A table, **`INT linkArray`**, is assigned for each transputer, in which each link has a corresponding entry giving the identity of the neighbor attached to that link (if any), and that neighbor's link, e.g.

```
linkArray[3]  :=  [6,0]
```

would be set to indicate that link 3 is connected to link 0 of transputer 6. When a parent boots a child, this information is transmitted in the **`loadingData`**. The source code for this example is shown in Appendix C.

5.6 Summary

We show in this chapter how a large array of transputers can be configured. A program determines the interconnection structure of a network of processors. Two different interconnection structures are considered: a tree and a generalized closed connection structure. A similar function can be included in the simulator. This would be implemented by tracing the path of the rendezvous between the communicator tasks at different nodes.

CHAPTER 6

TESTING AND EVALUATION OF MALEK'S FAULT DETECTION ALGORITHM

We have already introduced the comparison model designed by Malek. A pair of units is assumed to be compared by a matcher. In this section we show how our simulator can be extended to implement Malek's algorithm and detect a faulty unit in the grid. It also explains how the grid can be used as a tool to test other fault detection algorithms.

6.1 Diagnostic Table

Three assumptions are made by Malek in the comparison model :

1. No unit compares itself with others.
2. A comparator compares a pair of adjacent units, i.e. there is no other unit on the path from the comparator to the compared units.
3. A single comparator compares only two units at a time.

The basic diagnostic table is illustrated in Table 6.1.

We can see from the table that if the comparator is faulty, then no matter what the status of the compared units the test outcome is always a 'don't care'.

Comparator	Compared unit # 1	Compared unit # 2	Comparison outcome
fault-free	fault-free	fault-free	0
fault-free	fault-free	faulty	1
fault-free	faulty	fault-free	1
fault-free	faulty	faulty	1
faulty	fault-free	fault-free	X
faulty	fault-free	faulty	X
faulty	faulty	fault-free	X
faulty	faulty	faulty	X

Table 6.1 Diagnostic Table

6.2 Implementation of the Algorithm

There are certain assumptions to be made before we apply the algorithm to our network. We first assume that a maximum of two processors can fail in the network. The probability of more than two units failing out of nine processors at any instant of time is very low. However, we analyze the network based on a single fault assumption. The other assumption is that while analyzing the faulty processor the rest of the processors in the network remain healthy. The simulator can be extended to detect multiple faults but the number of test cycles required increases proportionally.

The general principle behind detecting a faulty unit is explained as follows:

Let us assume that there are three sets of processors connected as shown in Figure 6.1. We assume that unit 3, which acts as a comparator, is fault-free. The comparator assigns some tasks to units 1 and 2. The comparator then compares the outcomes of units 1 and 2. If there is a mismatch, we know that there is a fault in one of the units.

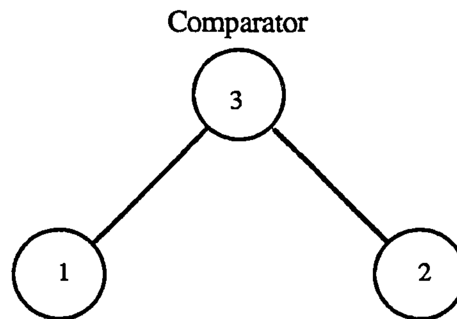


Figure 6.1 Set of three processors.

We assign these tasks to processors using a file oriented user interface. In our simulator, the assigning of tasks to different units by the comparator is equivalent to sending a message to any of the units. The next step is to compare the outcome from the two units under consideration. The simulator can introduce a fault by sending an incorrect result back to the comparator. Once the comparator has received the messages from both the units it can determine whether the results are the same. The messages that are sent among the different processors are saved in a separate file. The simulator reads the input file and determines the route via which the messages would be passed among the processors. The file format is as follows:

Source	Command	Data
0	send_to 1	20
0	send_to 3	30
.	.	.
.	.	.

The entry in the first column lists the name of the source node from where a message gets transmitted. The entry in the second column shows the destination node number. It also shows the task to be performed. In our case it is sending messages, thus the type variable declaration send_to. The last entry is the actual data sent. The data in our case is an integer. We show how to detect a single fault by means of an example.

Let us assume that node 2 is faulty. Now the simulator's task is to show that it is faulty. First of all, we have to prove that the rest of the nodes in the grid are healthy. There could always be a possibility that one of the other nodes is faulty. There are in all four comparison test cycles that are required to prove that the other nodes are healthy.

First we compare the outputs of nodes 1 and 3. The comparator in this case is node 0. Node 0 sends some messages to nodes 1 and 3. Nodes 1 and 3 in return send back the received messages to node 0. This could be considered as an acknowledgement procedure. If the messages received by node 0 are the same as those that had been transmitted, then we can conclude that nodes 1 and 3 are healthy nodes. Table 6.2 shows the four test cycles required to prove that all the nodes other than node 2 are fault-free. The pictorial view of the nodes under consideration is also shown in

Figure 6.2. We can see from each of the test cycles in Table 6.2 that the comparator receives the same messages that it had transmitted. Thus we can conclude that all the compared units are fault-free.

Notice that only four test cycles are required to prove that node 2 is faulty. Out of the eighteen connections required to form the grid only eight connections are used for comparisons as one can see from Figure 6.2.

To detect the faulty node 2, the program simulates a fault in node 2 by assigning it a task that sends a wrong message to the comparator. The test cycle required to detect it is shown in Table 6.3. The results of executing the algorithm are shown in Appendix B.

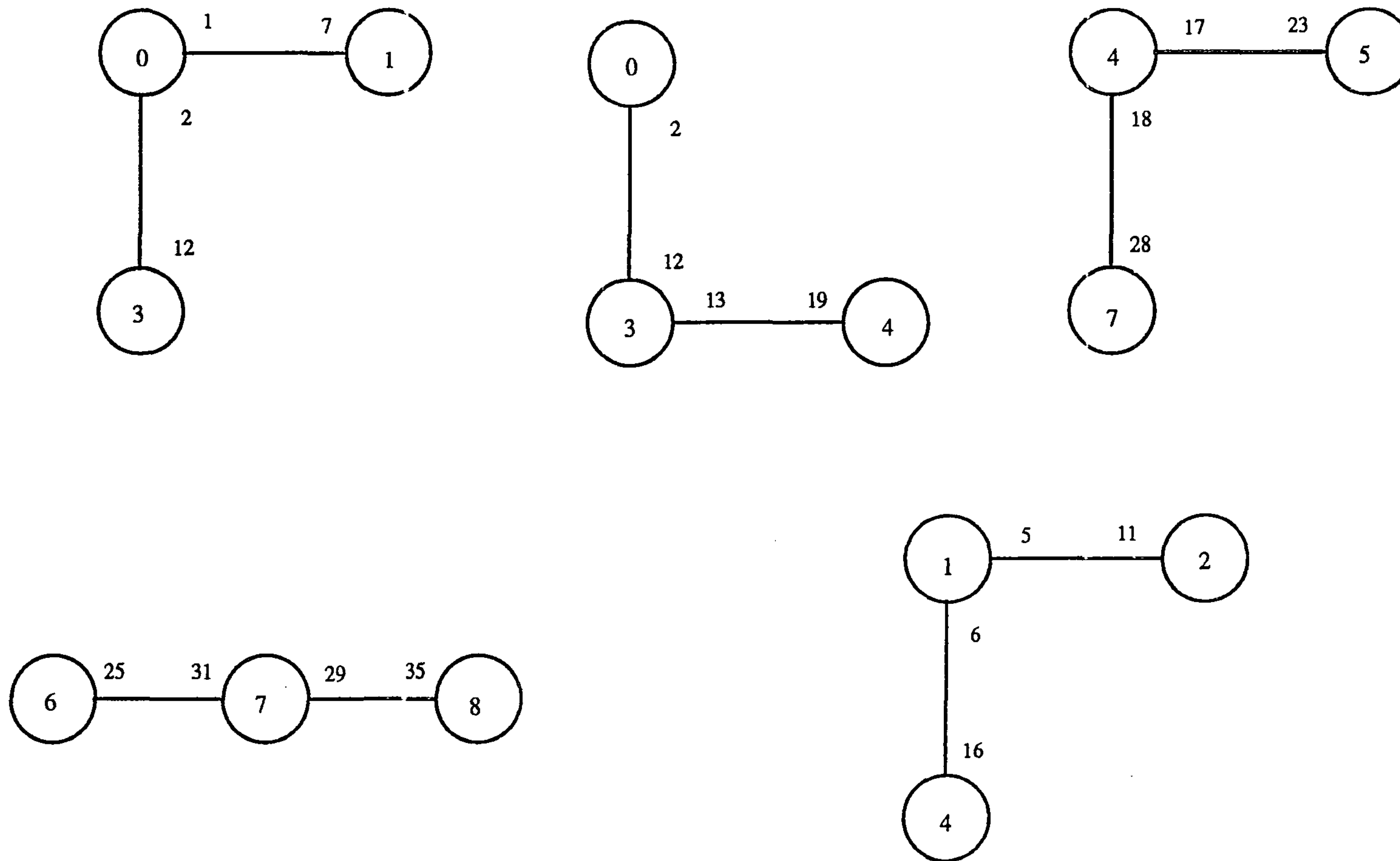


Figure 6.2 Pictorial view of the diagnosis of nine nodes

Comparator C	Compared Unit A	Compared Unit B	Message sent to A	Message sent to B	Message received by C from A	Message received by C from B	Test Outcome
0	1	3	10	10	10	10	0
3	0	4	20	20	20	20	0
4	5	7	30	30	30	30	0
7	6	8	40	40	40	40	0

Table 6.2 Table for diagnosis of healthy nodes

Comparator C	Compared Unit A	Compared Unit B	Message sent to A	Message sent to B	Message received by C from A	Message received by C from B	Test Outcome
1	2	4	50	50	52	50	1

Table 6.3 Test cycle detecting the faulty node

6.3 Extension of the Simulator to Test any Fault Detection Algorithm

The simulator can be used to test any other fault detection algorithm. To apply it to other algorithms, there are certain things that need to be changed. First and foremost, the fault detection algorithm should be based on a message passing system because our simulator's operation is based on message passing. Since any diagnostic algorithm can be expressed in terms of message-passing processes this not a limitation of the simulator.

The current simulator supports only 'send_to' operations. If the algorithm needs some calculations to be performed, then the user would have to implement some extra subroutines to support the calculations. Furthermore the file oriented user interface would also need to be altered. If the algorithm needs any mapping strategies, then the interface which decides the input for the program also needs to be modified. As explained in Section 3.6, the efficient execution of the algorithms would also depend on the interconnection structure of the network.

6.4 Evaluation of Malek's Algorithm

In this section we present some results of executing Malek's algorithm on different multiprocessor structures. We consider five different cases including the specific multiprocessor structure which we have simulated. Essentially we find out the number of comparisons and comparison edges required to detect a single faulty unit in a network of processors.

As explained in Chapter 2, the upper and lower bounds for determining the comparison cycles and the comparison edges are given by the following equations.

$$\left\lceil \frac{n+1}{2} \right\rceil \leq c_1 \leq n-1 \quad \text{......... (a)}$$

$$n - \left\lfloor \frac{n}{3} \right\rfloor \leq q_1 \leq n-1 \quad \text{.......... (b)}$$

where c_1 = the number of comparisons required in order to locate any fault in the system, and

q_1 = the number of comparison edges required in order to locate any fault in the system.

We consider five different configurations and apply Malek's comparison algorithm to them. The first three configurations have nine nodes. The fourth one has ten nodes and the last one has five nodes. The parameters c_1 and q_1 are found for each configuration and they are found to lie within the bounds given by equations (a) and (b).

6.5 Analysis of Different Configurations

We analyze in detail the first case which we have actually implemented with the help of our simulator. The other four cases are analyzed similarly and their results are shown in the tables that follow.

Case 1: Configuration with nine nodes connected in a toroidal fashion.

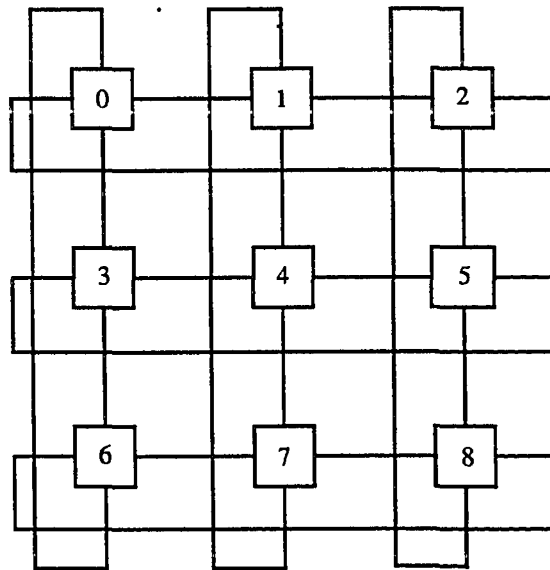


Figure 6.3 Nine nodes in a toroidal configuration.

The configuration which our simulator analyzes is shown in Figure 6.3. The comparisons required to detect a faulty unit in this case are shown in Figure 6.2. We assume that node 2 is faulty. Before analyzing node 2 we have to confirm that the remaining nodes in the grid other than node 2 are healthy. Thus there are in all five comparisons which need to be done to detect a faulty node. The equations for a configuration where nine nodes are involved are as follows:

$$5 \leq c_1 \leq 8 \text{ and } 6 \leq q_1 \leq 8 \ldots\ldots \text{(c)}$$

The bounds for these parameters are calculated from equations (a) and (b) given in section 6.1. Thus from Figure 6.2 we find out that $c_1 = 5$ and $q_1 = 8$.

Case 2 : Star shaped configuration with nine nodes

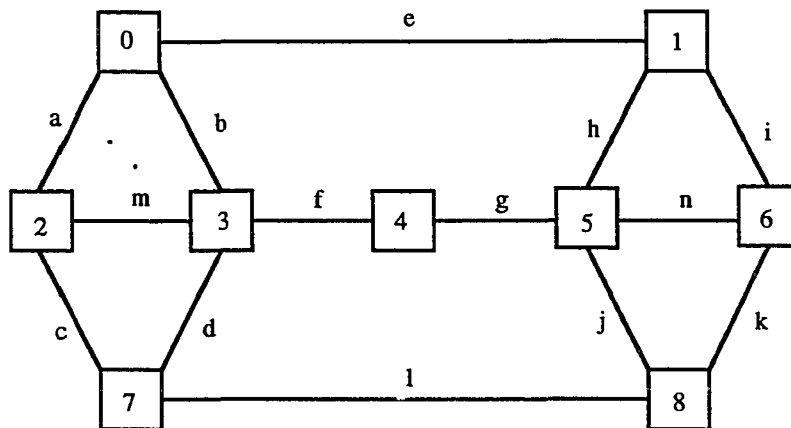


Figure 6.4 Star shaped configuration

Comparator	Compared unit # 1	Compared unit # 2	Comparison edges
2	0	7	a nd c
7	2	8	c and l
2	0	3	a and m
0	1	2	e and a
5	1	4	h and g
1	5	6	h and i

Table 6.4 Comparison parameters for case 2

The comparison cycles and the comparison edges required to detect a single faulty unit (in this case unit 6) for the configuration shown in Figure 6.4 are shown in Table 6.4. We find out that there are six cycles required to detect a faulty unit and the number of comparison edges required are eight, viz. a, c, e, g, h, i, l and m. Thus $c_1 = 6$ and $q_1 = 7$. These values are within the bounds given by equation (c) in case 1.

Case 3: Lattice configuration with nine nodes

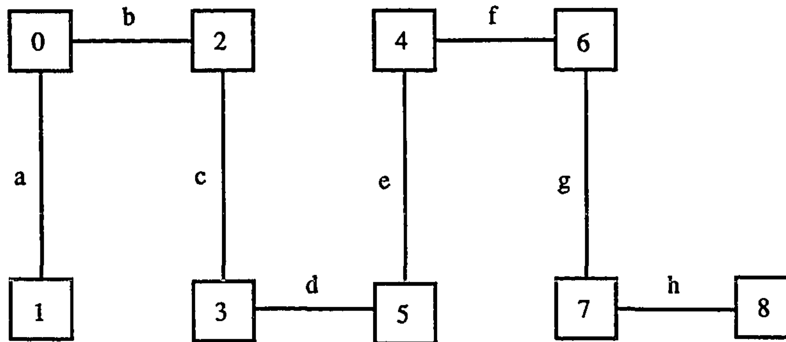


Figure 6.5 Lattice structure with nine nodes

This configuration is shown in Figure 6.5. We require a total number of seven cycles to detect a faulty node (in this case node 8). The number of comparison edges required are eight. The results can be seen in Table 6.5. The comparison edges are namely a, b, c, d, e, f, g, h. All the edges are utilized in this case. Thus $c_1 = 7$ and $q_1 = 8$.

Comparator	Compared unit # 1	Compared unit # 2	Comparison edges
0	1	2	a and b
2	0	3	b and c
3	2	5	c and d
5	3	4	d and e
4	5	6	e and f
6	4	7	f and g
7	6	8	g and h

Table 6.5 Comparison parameters for case 3

Case 4: Lattice structure with ten nodes

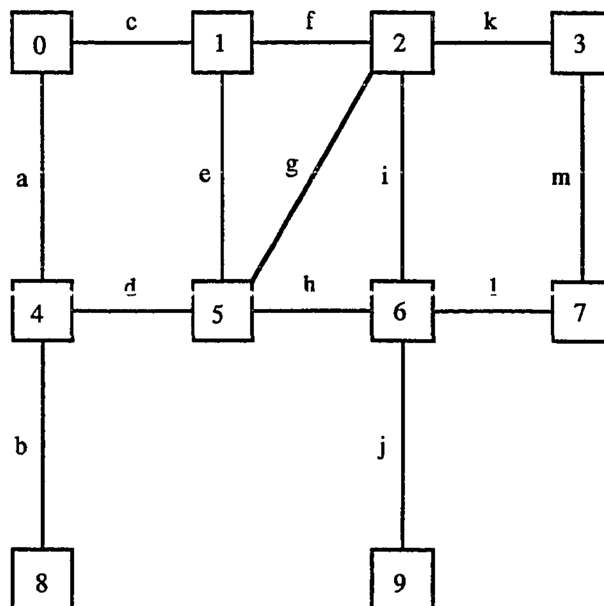


Figure 6.6 Lattice configuration for ten nodes.

Comparator	Compared unit # 1	Compared unit # 2	Comparison edges
4	0	8	a and b
4	0	5	a and d
5	2	6	g and h
6	2	9	i and j
6	2	7	i and l
7	3	6	m and l
0	1	4	c and a

Table 6.6 Comparison parameters for case 4

For the configuration shown in Figure 6.6 we can see from Table 6.6 that we require seven comparison cycles to determine a single fault (in this case node 1). The number of comparison edges required are nine, viz. a, b, c, d, g, h, i, j, and l. Thus $c_l = 7$ and $q_l = 9$.

The bounds for this network however are different as there are ten nodes. They are:

$$6 \leq c_l \leq 9 \text{ and } 7 \leq q_l \leq 9.$$

Case 5: Pentagonal configuration

The bounds for this configuration (Figure 6.7) will also change as there are only five nodes in the network. The bounds would be as follows:

$$3 \leq c_l \leq 4 \text{ and } 2 \leq q_l \leq 4.$$

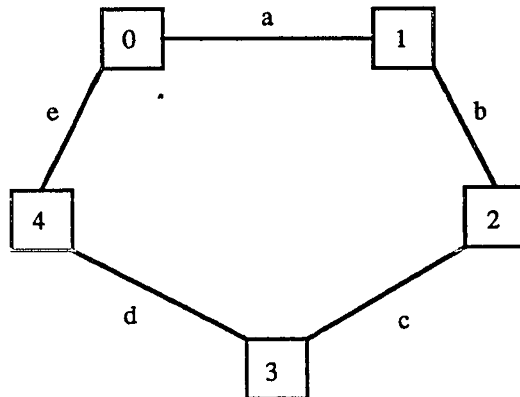


Figure 6.7 Five nodes connected in a pentagonal shape

Comparator	Compared unit # 1	Compared unit # 2	Comparison edges
0	1	4	a and e
4	0	3	e and d
1	0	2	a and b

Table 6.7 Comparison parameters for case 5.

As we can see from Table 6.7, there are only three comparison cycles required to detect a faulty unit in the network (in this case node 2). The number of comparison edges required to detect this faulty unit are four. Thus in this case $c_l = 3$ and $q_l = 4$. The comparison edges are a, b, d and e.

6.6 Conclusions

We have show in this Chapter, how a particular fault detection algorithm can be tested with the help of our simulator. We also explain the flexibility of our network and the different parameters that need to be changed in order for our network to test different algorithms. Specifically we test Malek's algorithm to detect a faulty unit in a network of transputers. We selected Malek's algorithm because it seems interesting due to its simple way of detecting faults.

Appendix B shows the different results obtained in testing Malek's algorithm. We test the algorithm to detect a single faulty unit in the network of transputers.

The number of comparison cycles required to locate a faulty processor and the number of comparison edges required vary according to the complexity of the network. When there are less number of processors, one requires less number of comparison edges and comparison cycles to detect a faulty unit. This is obvious in case 5, where we have a network of five processors. The parameters also depend on how tightly the network is connected. For the cases where we consider nine nodes, we see that in case 1, the processors are fully connected to each other. In cases 2 and 3 not all the links of the individual processors are utilized as the architectures do not require some of the links. Thus for case 1, the number of comparison cycles and the comparison edges required for fault detection are lower than those of cases 2 and 3.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

Multiprocessors are gaining importance nowadays in real time applications. We have designed a simulator for multiprocessor systems of transputers or similar processors. Essentially, we have developed a tool with the help of which we can test various fault detection algorithms.The simulator is made flexible to some degree, in the sense that it can adapt itself to any configuration and any number of processors by changing only a few parameters. The simulator bases its operation on message passing. and uses a store and forward scheme to pass messages between the processors. We have analyzed Malek's comparison algorithm on different configurations of multiprocessors.

We have limited the study of the network simulator only to the test of fault detection algorithms. We only considered the case of a total node failure and assumed that if there is a path to a healthy node, the interprocessor communication scheme is operational. We did not concern ourselves with the effect of multiple node failures on the distribution of the interprocessor traffic. However, in a real time system, node failures may affect the overall system reliability. For example, with more nodes going down, the burden of communications in certain nodes will vary according to the various communication patterns and intensity, with an increase in delay among the nodes of a system. This adds another dimension to the problem due to the fact that in real time systems, it is not enough to deliver a correct result, but correctness is also

bound by the time limitation factors. Correctness implies the delivery of a correct result within a limited time period. However, timing aspects cannot be measured with a simulator of the type discussed here.

The existence of hardware replication allows to replicate a task to have a backup unit, or having multiple functional copies of the same task engaged in some sort of decision making scheme, e.g. byzantine voting. This software replication truly takes advantage of the hardware replication present in the architecture. In these cases, the analysis becomes more complicated because we cannot address the problem of node or communication failures generically, i.e. the distribution or configuration of the specific tasks across the replicated hardware, together with the criticality of each particular task define the overall system reliability. This problem could be explored in greater detail and depends in general, on the allocation schemes used, types of faults, etc.

The reconfiguration strategy discussed here is an interesting way to restructure multiprocessor architectures using the switch lattice approach. By designing a switching task in Ada to control the switches the simulator can be made to achieve fault tolerance in a multiprocessor environment. As such, it could be useful to test different fault tolerance mechanisms.

The configuration program is a useful vehicle for testing transputer based applications. Tests for memory and for the links may be included in the basic program, for example. If a hardware fault occurs, the program may report the location and nature of the problem, while continuing to check other components in the network. By testing

the network repeatedly with a configuration program, any failure may be detected and logged, while the rest of the network continues to be tested. This concept could be implemented in the future to make the configuration program a complete diagnostic package to detect any faults in a multiprocessor system. This is an offline method for fault detection. It can be implemented in the simulator so that it can be used as a tool to test various algorithms as well as to configure an unknown network of processors.

The simulator can be improved with respect to its usability. A user-friendly interface could allow a user to define different configurations, to insert faults, to collect statistics, to control displays, and to use different fault detection or reconfiguration algorithms.

REFERENCES

[Booc83] G. Booch, Software Engineering with Ada, The Benjamin/Cummings Publishing Company, Inc., 1983.

[Buhr84] R. Buhr, System Design with Ada, Prentice Hall Intl., 1984.

[Degon78] P. K. DeGonia, R. C. Witt, D. R. Lampe, and E. L. Cole, Jr., "Micronet - A self-healing network for signal processing", Digest of Papers-Government Microcircuit Application Conf., Monterey, California, Nov. 1978, 370-377.

[Fern88] E. B Fernandez, Class notes for Fault Tolerant Computer Systems, F.A.U. 1988.

[Hoar78] C.A.R. Hoare, "Communicating Sequential Processes", C. ACM, 21, 8, 1978, 666-677.

[Kart78] S. I. Kartashev and S. P. Kartashev, "Dynamic Architectures: Problems and Solutions", Computer, Vol. 11, July 1978, 26-40.

[Kerr84] J. M. Kerridge and D. Simpson, "Three Solutions for a Robot Arm Contrioller using Pascal-plus, Occam and Edison", Software Practice and Experience, 14, 1984, 3 - 15.

[Kuhl86] J.G. Kuhl and S. M. Reddy, "Fault-Tolerance Considerations in Large, Multiple-Processor Systems", IEEE Transactions on Computers, 1986, 56 - 67.

[Kung84] H. T. Kung and M. S. Lam, "Wafer Scaled Integration and two Dimensional Pipelined Implementations of Systolic Arrays", Proc. Conf. Advanced Research in VLSI, MIT, Cambridge, Mass., Jan. 1984.

[Liu80] K. Y. Liu and Malek,"Graph theory models in fault diagnosis and fault tolerance", Journal of Design Automation and Fault-Tolerant Computing, Vol. III, Issue 3/4, 1980, 155-169.

[Maen81] J. Maeng, M. Malek, "A Comparison Connection Assignment for Self-Diagnosis of Multiprocessor Systems", Proc. Intl. Conf. on Fault Tolerant Computing Systems, 1981, 173 - 175.

[Male80] M. Malek, "A Comparison Connection Assignment for Diagnosis of Multiprocessor Systems", Proc. 7th Symp. on Computer Architecture., 1980, 31 - 36.

[Poun86] D. Pountain, "A Tutorial Introduction to Occam Programming", INMOS 1986.

[Prep67] F. P. Preparata, G. Metze and R. T. Chien, "On the connection assignment problem of diagnosable systems", IEEE Trans. on Elect. Comp., Vol. EC-16, No. 6, Dec. 1967, 848-854.

[Snyd82] L. Snyder , "Introduction to the Configurable Highly Parallel Computer", Computer, January 1982, 47-56

[Toy78] W. N. Toy, "Fault tolerant design of local ESS processors", Proc. of IEEE, vol. 66, No. 10, Oct. 1978, 1126-1145.

[Yala85] S. Yalamanchalli, J. K. Agarwal, "Reconfiguration strategies for Parallel Architectures", Computer, Dec. 1985, 44-61

APPENDIX A

Ada Source code for the simulator

```
WITH text_io;
USE  text_io;
WITH calendar;
USE calendar;

PROCEDURE grid_simulation  IS

  PACKAGE int_io IS NEW integer_io(integer);
  USE int_io;


 --D E C L A R A T I O N S
                                      -- (*global_declarations*)--

   SUBTYPE bit IS INTEGER RANGE 0..1;
   -- For the transputer links

   TYPE header_type IS (control_hdr,diagnostic_hdr,data_hdr);
   SUBTYPE node_id_type IS INTEGER;

   TYPE protocol_class_type IS (transputer_protocol,unsuported_protocol,
                              protocol_class_error);
   TYPE protocol_reply_class_type IS(no_reply_expected,reply_expected);
   SUBTYPE command_type IS INTEGER;
   TYPE diag_info_type IS ARRAY(0..8) OF bit;

   TYPE
   buffer_header_type IS RECORD          -- Header of a transmission buffer
                header              : header_type;
                source_id           : node_id_type;
                destination_id      : node_id_type;
                router_source_id    : node_id_type;
                router_destination_id: node_id_type;
                protocol_class      : protocol_class_type;
                protocol_reply_class : protocol_reply_class_type;
                END RECORD;

   TYPE
   buffer_data_type IS RECORD           -- DATA CONTENTS OF BUFFER
               command_info : command_type;
               diag_info    : diag_info_type;
```

```
            END RECORD;
 TYPE
 packet_type IS RECORD

            buffer_header : buffer_header_type;
            buffer_data   : buffer_data_type;
        END RECORD;

PACKAGE  hdr_io IS NEW enumeration_io(header_type);
USE hdr_io;

PACKAGE  pro_io IS NEW enumeration_io(protocol_class_type);
USE pro_io;


 matted_array:ARRAY(0..35) OF INTEGER;  -- USED  FOR LINK MAPPING
     --** The matted array is part of the GRID hardware configuration
     --** database. Each entry element in this array contains an
     --** identifier for the link number connected to the ith link

 TYPE compass_array_type IS ARRAY (0..3) OF INTEGER;
      --( Used by a node to figure out to which of its link the message
      --  is to be routed to)..

 grid_size   : integer := 3  ; -- For a 3X3 grid ( 9 nodes )
 this_link   : integer ;
 st_id       : integer ;
 partner_link : integer ;



--------------------------------------------------------------- * * *
 TYPE job_kind IS (send_to,nop);
 TYPE job_card_type;
 TYPE job_ptr IS ACCESS job_card_type;

--------------------------------------------------------------- * * *
 TYPE job_card_type IS RECORD
      doer : integer  ;       -- Node doing the job
      class : job_kind ;       -- Job type  (WHAT )
      dest : integer  ;       -- Destination NODE
      dat  : integer  ;       -- DATA .....
      next : job_ptr  :=null; -- Pointer to next JOB
 END RECORD;


--------------------------------------------------------------- * * *
 PACKAGE  job_io IS NEW enumeration_io(job_kind);
```

```
  USE job_io;

---------------------------------------------------------------- * * *
                                          --(*file_declarations*)--


  linkmap:  file_type;     --(* LINK CONNECTION MAP  *)--
  injobs :  file_type;     --(* JOBS TO BE PERFORMED *)--


---------------------------------------------------------------- * * *
-- DEBUGG DECLARATIONS
 debugg : boolean:=false;

---------------------------------------------------------------- * * *

                                        --(*task_declarations*)--

-- Simulation of a transputer link. This task is a generic basic
-- Transputer link


 TASK TYPE link_task IS
    ENTRY configure_link(node_id: in integer;
                         link_id: in integer; partner_link: in integer );
    ENTRY transmit(f_packet : in packet_type);
    ENTRY receive( f_packet : in packet_type);
 END link_task;


-- Simulation of a communicator task.

  TASK TYPE communicator_task IS
        ENTRY configure_comm(id:in integer);
        ENTRY message_to_transmit(f_packet : in  packet_type );-- DATA TX
        ENTRY message_received(f_packet   : in  packet_type );-- DATA RX
  END communicator_task;


-- Simulation of an application task. This task contains the basic
-- definitions for the application dependent tasks.


 TASK TYPE application_task IS
    ENTRY configure_appl(id:in integer);
    ENTRY input(f_packet  : in  packet_type);
    ENTRY output(f_packet : out packet_type);
 END application_task;
```

```
                        --(*utilities_declarations*)--

  PROCEDURE message_checker(message_in        : in packet_type;
                            validation_result : out boolean);

  FUNCTION  find_direction_to_go(current_node_id: in integer;
                                 going_to_node  : in integer) return integer;

  FUNCTION  find_next_link_id(f_compass : in compass_array_type;
                           f_destination_link_offset : in integer)
                           RETURN integer;


  PROCEDURE read_initial_grid_configuration;

PROCEDURE  initialize_job_queue;
PROCEDURE  add_job_into_job_queue( f_node  : in  integer   ;
                                   f_do    : in  job_kind  ;
                                   f_dest  : in  integer   ;
                                   f_dat   : in  command_type) ;
PROCEDURE  read_input_jobs ;
PROCEDURE  get_job_from_job_queue( f_node  : in integer  ;
                                   f_job   : out job_card_type ;
                                   outcome: out boolean ) ;
PROCEDURE  print_job_card(f_job: in job_card_type );


-------  FORMATTING ROUTINES FOR OUTPUT

PROCEDURE  print_separation_line;

PROCEDURE  print_packet(f_packet : in packet_type);



                        --(* task_instantiations*)--

  link:array(0..35) of link_task;
  application:array(0..8) of application_task;
  communicator:array(0..8) of communicator_task;


   job_queue_status: ARRAY(0..8) OF BOOLEAN;
```

```
  job_queue_ptrs: ARRAY(0..8) OF job_ptr;   -- Pointer to individual queues

  a_job   : job_card_type;
  success : boolean;

                                              --(*task_implementation*)--
-- Body of transputer link

  task body link_task is
   node_number : integer;
   link_own_id : integer;
   adjacent_link : integer;
   tx_buffer : packet_type;
   tx_buffer_empty : boolean := true;
   rx_buffer : packet_type;
   rx_buffer_empty : boolean := true;

BEGIN

   ACCEPT configure_link(node_id: in integer;
                         link_id: in integer;
                         partner_link :in integer) DO

      node_number := node_id;        -- This link belongs to this node
      link_own_id := link_id;        -- This is the link own id
      adjacent_link := partner_link; -- To which link is connected
      new_line;
      put("  LINK TASK INSTANTIATED node number is");
      put(node_number,4);
      put("  THE LINK IDENTIFICATION NUMBER IS ");
      put(link_own_id,4);
      put("  ADYACENT  node number is");
      put(adjacent_link,4);
      new_line;

    end configure_link;

link_loop: LOOP   -- Infinite loop

  SELECT

-- Accept transmit operation  (configuration dependent)

   ACCEPT transmit(f_packet:IN packet_type)DO
```

```
        new_line;
        put(" *** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS");
        put(link_own_id,2);
        new_line;

        tx_buffer  := f_packet;

        link(adjacent_link).receive(tx_buffer);

      END transmit;

    OR

-- Accept RECEIVE operation
      ACCEPT receive(f_packet : IN  packet_type)DO
       new_line;
       put(" *** LINK RECEIVED PACKET **** LINK # IS ");
       put(link_own_id,2);
       new_line;

       rx_buffer := f_packet;

       communicator(node_number).message_received(rx_buffer);

       END receive;

    END  SELECT;

end loop link_loop;

end link_task;


-- Body of communicator task

   TASK BODY communicator_task IS

     -- Internal Declarations

     a_packet : packet_type;
     node_number: integer;
     my_compass : compass_array_type;
     link_base  : integer;   -- Base Number to calculate own links IDs.
     an_offset  : integer;   -- Displacement to compute task number
     for_link   : integer;   -- Variable to  identify receiving link

     BEGIN
```

```
    ACCEPT  configure_comm(id:in integer)
          DO
      NULL;
           node_number := id;  -- Node Identification Number
           NEW_LINE;
           put(" COMMUNICATOR TASK INSTANTIATED node number is");
           put(node_number,4);
           new_line;

        END  configure_comm;

link_base := node_number*4 ;


for i in 0..3 loop
     my_compass(i) := link_base + i ;  -- Base Number + offsets...
end loop;

new_line;
put("  ***TASK COMMUNICATOR FULLY CONFIGURED *** NODE IS");
put(node_number,4);
new_line;



loop    -- LOOP FOREVER....

SELECT
    ACCEPT  message_to_transmit(f_packet: in packet_type)
           DO
           NULL;
           new_line;
           put(" **>>>>>>Task communicator accepted message to
                transmit");
           put(" at node number --->");
           put(node_number,2);
           new_line;

           a_packet := f_packet;

           an_offset := find_direction_to_go(node_number,
                      a_packet.buffer_header.destination_id);

            for_link := find_next_link_id(my_compass,an_offset);

            link(for_link).transmit(a_packet);
```

```
            END  message_to_transmit;

    OR
        ACCEPT  message_received(f_packet : in packet_type )
                DO
                NULL;
                new_line;
                put(" **>>>>>>Task communicator accepted message received
                     ");
                put(" at node number --->");
                put(node_number,2);
                new_line;
                -- Copy the packet into temp. variable
                a_packet := f_packet;

                -- Check it this node is the terminating node for the packet
                -- Received, if so, pass it to the application, otherwise,
                -- pass it to the appropriate link....

                IF  a_packet.buffer_header.destination_id = node_number

                THEN

print_separation_line;
new_line;
put(" Packet received by task communicator reached final destination");
put(" at NODE number");
put(node_number,2);
new_line;
print_separation_line;
                    -- Pass packet to the application task
                    application(node_number).input(a_packet);

                ELSE
          --WHEN OTHERS      =>   -- For another node

print_separation_line;
new_line;
put(" Packet received by task communicator is to be forwarded ");
put(" at NODE number");
put(node_number,2);
new_line;
print_separation_line;

                -- 1. Calculate the offset to be used in routing this pkt.

                an_offset := find_direction_to_go(node_number,
                           a_packet.buffer_header.destination_id);
```

```
              -- 2. Find which of its links to be used...

                for_link := find_next_link_id(my_compass,an_offset);

                -- 3. Link to be used is known now, so pass-it-on !!!
if debugg then
              new_line;
              put(" **>>>>>>Task communicator about to forward packet ");
              put(" at node number --->");
              put(node_number,2);
              new_line;

end if; --debugg

                link(for_link).transmit(a_packet);

if debugg then
              new_line;
              put(" **>>>>>>Task communicator fowarded packet completed
            ");
              put(" at node number --->");
              put(node_number,2);
              new_line;
end if; --debugg

                --END CASE;
                END IF;

             END message_received;
   END SELECT;

   END LOOP ;  -- Loop forever

      END communicator_task;


-- Body of the Application Task

TASK BODY  application_task IS

        node_number:integer;
        a_packet : packet_type ; -- Packet to be sent or received
        a_job    : job_card_type ; -- What to do....
        success  : boolean       ; -- Outcome of request...

  -- (* Internal Procedures *)--
```

```
PROCEDURE  appl_build_packet(f_job      : IN  job_card_type ;
                                  f_packet : OUT packet_type) IS
--*  This procedure builds a packet originating in the APPLICATION TASK
--*  and sends it to the TASK COMMUNICATOR...

f_header     : buffer_header_type;
f_buffer_data : buffer_data_type;


BEGIN

  -- The packet is build from the job card by extracting
  -- the necessary information from it.
  -- This information is used to build the header first
  -- and then to build the packet data portion....


  -- BUILD THE BUFFER_HEADER
  f_header.header               := data_hdr  ;
  f_header.source_id            := node_number;
  f_header.destination_id       := f_job.dest ;
  f_header.router_source_id     := 0;           -- Filled by COMMUNICATOR
  f_header.router_destination_id:= 0;           --  "    "    "
  f_header.protocol_class       := transputer_protocol;
  f_header.protocol_reply_class := no_reply_expected ;

  -- BUILD BUFFER_DATA....
  f_buffer_data.command_info   := f_job.dat ;  -- Data to be send
  f_buffer_data.diag_info      := (0,0,0,0,0,0,0,0,0);  -- N/A

  -- AND PUT IT IN THE PACKET
  f_packet.buffer_header  := f_header ;
  f_packet.buffer_data    := f_buffer_data;

  END appl_build_packet;




  BEGIN      -- Application_body_begins.....

   accept configure_appl(id:in integer) do

      node_number:=id;     -- (* Accept the node identification number *)--
    new_line;
    put(" *** APPLICATION TASK  ACCEPTED ID  node number is");
```

```
      put(node_number);
      new_line;

  end configure_appl;

forever:  LOOP              --(* INFINITE LOOP   *)--

SELECT

   -- Accept Input Data

   ACCEPT input(f_packet: in packet_type ) DO
      new_line;
         put(" *** APPLICATION TASK ACCEPTED INPUT ***");
         put(node_number,2);
         new_line;

         -- The application task received a packet, display a
         -- message and the packet contents....

         new_line;
         print_separation_line;
         put("  PACKET RECEIVED BY THE APPLICATION TASK AT NODE
              ");
         put(node_number,2);
         new_line;
         print_separation_line;
         print_packet(f_packet);
         print_separation_line;

    end input;

 OR


   -- Accept Output Data (To be sent to another place)

   ACCEPT output(f_packet : out packet_type )  DO
      new_line;
         put(" *** APPLICATION  TASK ACCEPTED OUTPUT ***");
         put(node_number,2);
         new_line;
        NULL;

    end output;

  OR  delay 1.1  ;
```

```
  END SELECT;

  IF job_queue_status(node_number)  then

    get_job_from_job_queue(node_number,a_job,success);

    IF success then

new_line;
put(" JOB OBTAINED ");
put(node_number,2);
new_line;

    CASE  a_job.class  IS

    WHEN  send_to =>

new_line;
put(" Application about to attempt packet build ");
put(node_number,2);
new_line;

            appl_build_packet(a_job,a_packet);

new_line;
put(" Application built  packet  ");
put(node_number,2);
new_line;

            communicator(node_number).message_to_transmit(a_packet);

if debugg then

new_line;
put(" Application sent packet to COMMUNICATOR ");
put(node_number,2);
new_line;
end if;  --debugg

    WHEN OTHERS => NULL;
                    put_line(" UNRECOGNIZED JOB TYPE FOR APPLICATION
TASK");

    END CASE;

    END IF;    -- IF SUCCESS
```

```
  ELSE
    delay 0.2;

if debugg then
    new_line;
    put(" --__--__--__--__--__--__--__--__--__--__--__--__--__--__--__");
    put_line(" *** Queue tested and found empty for task ");
    put(node_number,2);
    put(" --__--__--__--__--__--__--__--__--__--__--__--__--__--__--__--__");
end if; --debugg

   END IF;      -- QUEUE NOT EMPTY

 END LOOP forever;   --(* INFINITE LOOP FOR APPLICATION TASK *)--


END application_task;


                                    --(*utilities_implementation*)--



   PROCEDURE message_checker (message_in: in packet_type;
                              validation_result: out boolean) IS

         BEGIN
         NULL;
         new_line;
         put("  message checker has been called");
         new_line;

         end message_checker;


  FUNCTION  find_direction_to_go(current_node_id: in integer;
                                 going_to_node  : in integer) return integer is


   destination_link_offset : integer;   -- Indicates to which direction
                                         -- the packet should be routed

   type relative_position_type is (in_the_same_row,
                                   in_the_same_column,
                                   not_aligned);
   relative_position  : relative_position_type;

  ----------------------------------------------------------------------
```

```
function find_relative_position(current_node:in integer;
                                destination: in integer) return
                                relative_position_type IS
  row_a, row_b      : integer;
    column_a, column_b : integer;
    begin

      -- Find row number identifier
      row_a := current_node / grid_size;
      row_b := destination / grid_size;

      -- Find column number identifier
      column_a := current_node mod grid_size;
      column_b := destination  mod grid_size;

      if row_a = row_b then return in_the_same_row; end if;
      if column_a = column_b then return in_the_same_column; end if;
    if not ( (row_a = row_b) or (column_a = column_b) ) then
                                    return not_aligned;
    end if;

      end find_relative_position;
 ------------------------------------------------------------------

FUNCTION  is_to_the_top(a:in integer;b:in integer )return boolean is

    begin

    if a>b then return true ;  -- Destination is above source
      else return false;  -- Destination is below source
         end if;
    end is_to_the_top;
 ------------------------------------------------------------------

FUNCTION  is_to_the_right(a:in integer;b:in integer)return boolean  is

    begin
    if a<b then return true ;  -- Destination is to the right
      else return false;  -- Destination is to the lefth
         end if;

    end is_to_the_right;

 BEGIN
     NULL;
            relative_position:=find_relative_position(current_node_id,
                                                 going_to_node);
```

```
-- NOW FIND WHICH CASE APPLIES

CASE relative_position IS

   when in_the_same_row      =>

               if is_to_the_right(current_node_id, going_to_node)
                  then destination_link_offset:= 1; --GO RIGTH
                  else destination_link_offset:= 3; --GO LEFT
                  end if;

   when in_the_same_column   =>

               if is_to_the_top(current_node_id, going_to_node)
                  then destination_link_offset:= 0; --GO UP
                  else destination_link_offset:= 2; --GO DOWN
                  end if;

   when not_aligned          =>

               destination_link_offset:= 2; -- GO DOWN ALWAYS

END CASE;

  RETURN destination_link_offset;

 END find_direction_to_go;


FUNCTION find_next_link_id (f_compass: in compass_array_type;
                            f_destination_link_offset: in integer)
                            RETURN integer IS

  a_link_id : integer;

  BEGIN
 a_link_id := f_compass(f_destination_link_offset);
  return a_link_id;
  END find_next_link_id;

PROCEDURE read_initial_grid_configuration IS
     begin
     NULL;
     end read_initial_grid_configuration;

------------------------------------------------------------- * * *
PROCEDURE initialize_job_queue IS
```

```
        BEGIN
        -- SET JOB QUEUE STATUS TO EMPTY
        FOR index IN 0..8 LOOP
        job_queue_status(index):= FALSE ;  -- NO JOBS
        job_queue_ptrs(index)  := NULL  ;  -- NO JOBS
        END LOOP;

END  initialize_job_queue;


-------------------------------------------------------------- * * *
PROCEDURE   add_job_into_job_queue(
                              f_node   : in  integer   ;
                              f_do     : in  job_kind  ;
                              f_dest   : in  integer   ;
                              f_dat    : in  command_type) IS
        new_job : job_ptr;  -- Job Card pointer
        temp_ptr: job_ptr;  -- Job Card pointer

      BEGIN

        -- Create job card
        new_job := NEW job_card_type;      -- Creates new job card

        -- Copy input job information into the new job card
        new_job.doer     := f_node   ; -- Job is for this node task
        new_job.class    := f_do     ; -- Type of job to do
        new_job.dest     := f_dest   ; -- Destination Node
        new_job.dat      := f_dat    ; -- Data associated with job
        new_job.next     := NULL     ; -- Pointer to next JOB


        IF job_queue_status(f_node) = FALSE then
          job_queue_status(f_node) := TRUE;
          job_queue_ptrs(f_node)  := new_job;  -- LINK JOB CARD

          ELSE
          temp_ptr := job_queue_ptrs(f_node);  -- First JOB for task

          find_last_job: LOOP
          EXIT  find_last_job WHEN temp_ptr.next = NULL;

                temp_ptr := temp_ptr.next;  -- Advance to next job
          END LOOP find_last_job;

          temp_ptr.next := new_job;

          END IF;
```

```
      END add_job_into_job_queue;
------------------------------------------------------------- * * *
PROCEDURE  read_input_jobs  IS

   for_node : integer;
   do_this  : job_kind;
   to_node  : integer;
   some_data: integer;
   BEGIN

   -- OPEN INPUT JOB FILE
   open(injobs,in_file,"appl_jobs.file");


   while  not end_of_file(injobs) LOOP

   get(injobs,for_node );         -- JOB FOR NODE ID
   get(injobs,do_this  );           -- JOB TYPE
   get(injobs,to_node  );           -- TO WHERE ?
   get(injobs,some_data);          -- DATA....

   add_job_into_job_queue(
                         for_node  ,
                         do_this   ,
                         to_node   ,
                         some_data);


   END LOOP;

   END  read_input_jobs;
------------------------------------------------------------- * * *

PROCEDURE  get_job_from_job_queue(
                        f_node : in integer  ;
                        f_job  : out job_card_type ;
                        outcome: out boolean ) is

    no_job : job_card_type :=(
                          doer  => 0   ,
                          dest  => 0   ,
                          class => nop ,
                          dat   => 0   ,
                          next  => NULL);  -- Dummy JOB CARD

    BEGIN
```

```
new_line;
put(" ***** Getting a job for node ");
put(f_node);
new_line;

IF not job_queue_status(f_node) THEN
    f_job := no_job;
    outcome:= FALSE ;  -- Failure, no job in the queue
    RETURN;
    END IF;

f_job := job_queue_ptrs(f_node).all; -- Copy all fields...
outcome := TRUE; -- JOB transfer successful

job_queue_ptrs(f_node) := job_queue_ptrs(f_node).next ;
IF job_queue_ptrs(f_node)= NULL then
    job_queue_status(f_node):= FALSE; -- No more jobs

    put_line(" last job taken ... queue is empty now");

    END IF;


END get_job_from_job_queue;
--------------------------------------------------------------- * * *

      PROCEDURE print_job_card(f_job: in job_card_type )IS

      BEGIN
      new_line;
      put_line("    ********* PRINTING JOB CARD **********");
      put(" Node originating job is "); put(f_job.doer);
      new_line;
      put(" TYPE of JOB is "); put(f_job.class);
      new_line;
      put(" Destination node is "); put(f_job.dest);
      new_line;
      put(" The data associated with this job is");put(f_job.dat);

      END print_job_card;

--------------------------------------------------------------- * * *
PROCEDURE print_separation_line IS
    BEGIN
    new_line;
    put("----------------------------------------------------------------------");
    new_line;
    END print_separation_line;
```

```
------------------------------------------------------------- * * *
PROCEDURE print_packet(f_packet : in packet_type) is
    BEGIN
    new_line;
    put_line(" ******** PACKET HEADER ********");

    put(" Header type ");
    put(f_packet.buffer_header.header);
    new_line;

    put(" Source Node ");
    put(f_packet.buffer_header.source_id);
    new_line;

    put(" Destination Node ");
    put(f_packet.buffer_header.destination_id);
    new_line;

    put(" Protocol Class ");
    put(f_packet.buffer_header.protocol_class);
    new_line;

    put_line(" ******** PACKET DATA    ********");

    put(" Data value contained  ");
    put(f_packet.buffer_data.command_info);
    new_line;

    END print_packet;


------------------------------------------------------------- * * *

 FUNCTION find_matted_link(id_a:in integer) return integer is

    begin
    return matted_array(id_a);
    end find_matted_link;

                                        --(*MAIN_BEGINS*)--

 BEGIN  -- BEGINS GRID PROCEDURE

 read_file_link_map:
   declare
```

```
matted_link_id  :integer;
link_index      :integer;
a_node_id       :integer; -- dummies
begin
open(linkmap, in_file, "link_map.file");

      WHILE NOT END_OF_FILE(linkmap) LOOP
      get(linkmap ,a_node_id );
      get(linkmap ,link_index );
      get(linkmap ,matted_link_id);
      -- Store value in the table
      matted_array(link_index) := matted_link_id ;

      END LOOP;

      put_line("  * * *  LINK DISTRIBUTION CONFIGURED * * * ");
end read_file_link_map;

-- 1. Initialize the job queue
initialize_job_queue;
put_line(" Job queue initialized ");

-- 2. read all the jobs ...
read_input_jobs;

put_line("  All jobs inputted ");

--3. Verify jobs....
new_line;
put_line(" printing status of the jobs_queue");
for i in 0..8 loop
new_line; put(" status for node");put(i);put(" is ");
if job_queue_status(i) then put(" true ...some jobs");
                        else put(" false ... no jobs");
                        end if;
end loop;

new_line;
put_line(" PRINTING STATUS OF POINTERS      ");
for i in 0..8 loop
new_line; put(" POINTER for node");put(i);put(" is ");
if job_queue_ptrs(i)/=null then put(" NOT NULL ....some jobs");
                        else put(" NULL  ... no jobs");
                        end if;
end loop;


for I in 0..8 loop
```

```
   -- check if there is a job
   if job_queue_status(I) then   -- there are some jobs for this node
        new_line;
        put(" There are some jobs for this node "); put(i);
        --exaust: loop
        --exit  exaust when  not job_queue_status(i);
        --get_job_from_job_queue(i,a_job,success);

        -- And print job card
        --print_job_card(a_job);
        --end loop exaust;
        end if;

        end loop;    -- for I in 0..8
new_line;
put(" * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ *");
new_line;
put(" * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ *");
new_line;

---------------------------------------------------------------- * * *


 -- INITIALIZE THE LINK CONFIGURATION CHOOSEN....
 -- (*  INITIALIZE THE GRID *)--


    FOR node_id IN 0..8            --(* For each node in the grid  *)--


  LOOP


     application(node_id).configure_appl(node_id);

        communicator(node_id).configure_comm(node_id);


      st_id := 4 * (node_id );   -- Calculate starting number for link
                                    -- ( base link identification number )

         -- SINCE each node has four (4) link give each of the links of this
         -- node an identification number following previously defined express.

      FOR link_range IN 0 .. 3 LOOP --(* For each link in a node   *)--

         -- SEARCH CONNECTION MAP FOR MATTED LINK                        --
IDENTIFICATION
```

```
        this_link:=link_range+st_id;
        partner_link:= find_matted_link(this_link);
        link(this_link).configure_link(node_id,this_link,partner_link);
END LOOP;    --(* For each link in a node    *)--
END LOOP;    --(* For each node in the grid  *)--
END grid_simulation ;
```

EXAMPLE 1

Message passing in a toroidal configuration

Input File for the Simulator

Source Node	Operation	Destination Node	Data
0	send_to	8	08
8	send_to	4	84
3	send_to	5	35
7	send_to	1	71

Link Map for the Simulation

Node no.	Link no.	Link no.
0	0	26
0	1	7
0	2	12
0	3	9
1	4	30
1	5	11
1	6	16
1	7	1
2	8	34
2	9	3
2	10	20
2	11	5
3	12	2
3	13	19
3	14	24
3	15	21
4	16	6
4	17	23
4	18	28
4	19	13
5	20	10
5	21	15
5	22	32
5	23	17
6	24	14
6	25	31
6	26	0
6	27	33
7	28	18
7	29	35
7	30	4
7	31	25
8	32	22
8	33	27
8	34	8
8	35	29

Results of the Simulation

```
 * * * LINK DISTRIBUTION CONFIGURED * * *
Job queue initialized
 All jobs inputted

printing status of the jobs_queue

status for node      0 is  true ...some jobs
status for node      1 is  false ... no jobs
status for node      2 is  false ... no jobs
status for node      3 is  true ...some jobs
status for node      4 is  false ... no jobs
status for node      5 is  false ... no jobs
status for node      6 is  false ... no jobs
status for node      7 is  true ...some jobs
status for node      8 is  true ...some jobs
PRINTING STATUS OF POINTERS

POINTER for node      0 is  NOT NULL ....some jobs
POINTER for node      1 is  NULL  ... no jobs
POINTER for node      2 is  NULL  ... no jobs
POINTER for node      3 is  NOT NULL ....some jobs
POINTER for node      4 is  NULL  ... no jobs
POINTER for node      5 is  NULL  ... no jobs
POINTER for node      6 is  NULL  ... no jobs
POINTER for node      7 is  NOT NULL ....some jobs
POINTER for node      8 is  NOT NULL ....some jobs
There are some jobs for this node      0
There are some jobs for this node      3
There are some jobs for this node      7
There are some jobs for this node      8

*** APPLICATION TASK  ACCEPTED ID  node number is       0

COMMUNICATOR TASK INSTANTIATED node number is  0

 ***** TASK COMMUNICATOR FULLY CONFIGURED *** NODE IS  0

 LINK TASK INSTANTIATED node number is  0  THE LINK IDENTIFICATION
NUMBER IS    0  ADJACENT  node number is  26

 LINK TASK INSTANTIATED node number is  0  THE LINK IDENTIFICATION
NUMBER IS    1  ADJACENT  node number is  7
```

LINK TASK INSTANTIATED node number is 0 THE LINK IDENTIFICATION NUMBER IS 2 ADJACENT node number is 12

LINK TASK INSTANTIATED node number is 0 THE LINK IDENTIFICATION NUMBER IS 3 ADJACENT node number is 9

*** APPLICATION TASK ACCEPTED ID node number is 1

COMMUNICATOR TASK INSTANTIATED node number is 1

***** TASK COMMUNICATOR FULLY CONFIGURED *** NODE IS 1

LINK TASK INSTANTIATED node number is 1 THE LINK IDENTIFICATION NUMBER IS 4 ADJACENT node number is 30

LINK TASK INSTANTIATED node number is 1 THE LINK IDENTIFICATION NUMBER IS 5 ADJACENT node number is 11

LINK TASK INSTANTIATED node number is 1 THE LINK IDENTIFICATION NUMBER IS 6 ADJACENT node number is 16

LINK TASK INSTANTIATED node number is 1 THE LINK IDENTIFICATION NUMBER IS 7 ADJACENT node number is 1

*** APPLICATION TASK ACCEPTED ID node number is 2

COMMUNICATOR TASK INSTANTIATED node number is 2

***** TASK COMMUNICATOR FULLY CONFIGURED *** NODE IS 2

LINK TASK INSTANTIATED node number is 2 THE LINK IDENTIFICATION NUMBER IS 8 ADJACENT node number is 34

LINK TASK INSTANTIATED node number is 2 THE LINK IDENTIFICATION NUMBER IS 9 ADJACENT node number is 3

LINK TASK INSTANTIATED node number is 2 THE LINK IDENTIFICATION NUMBER IS 10 ADJACENT node number is 20

LINK TASK INSTANTIATED node number is 2 THE LINK IDENTIFICATION NUMBER IS 11 ADJACENT node number is 5

*** APPLICATION TASK ACCEPTED ID node number is 3

COMMUNICATOR TASK INSTANTIATED node number is 3

***** TASK COMMUNICATOR FULLY CONFIGURED *** NODE IS 3

LINK TASK INSTANTIATED node number is 3 THE LINK IDENTIFICATION NUMBER IS 12 ADJACENT node number is 2

LINK TASK INSTANTIATED node number is 3 THE LINK IDENTIFICATION NUMBER IS 13 ADJACENT node number is 19

LINK TASK INSTANTIATED node number is 3 THE LINK IDENTIFICATION NUMBER IS 14 ADJACENT node number is 24

LINK TASK INSTANTIATED node number is 3 THE LINK IDENTIFICATION NUMBER IS 15 ADJACENT node number is 21

*** APPLICATION TASK ACCEPTED ID node number is 4

COMMUNICATOR TASK INSTANTIATED node number is 4

***** TASK COMMUNICATOR FULLY CONFIGURED *** NODE IS 4

LINK TASK INSTANTIATED node number is 4 THE LINK IDENTIFICATION NUMBER IS 16 ADJACENT node number is 6

LINK TASK INSTANTIATED node number is 4 THE LINK IDENTIFICATION NUMBER IS 17 ADJACENT node number is 23

LINK TASK INSTANTIATED node number is 4 THE LINK IDENTIFICATION NUMBER IS 18 ADJACENT node number is 28

LINK TASK INSTANTIATED node number is 4 THE LINK IDENTIFICATION NUMBER IS 19 ADJACENT node number is 13

*** APPLICATION TASK ACCEPTED ID node number is 5

COMMUNICATOR TASK INSTANTIATED node number is 5

***** TASK COMMUNICATOR FULLY CONFIGURED *** NODE IS 5

LINK TASK INSTANTIATED node number is 5 THE LINK IDENTIFICATION NUMBER IS 20 ADJACENT node number is 10

LINK TASK INSTANTIATED node number is 5 THE LINK IDENTIFICATION NUMBER IS 21 ADJACENT node number is 15

LINK TASK INSTANTIATED node number is 5 THE LINK IDENTIFICATION NUMBER IS 22 ADJACENT node number is 32

LINK TASK INSTANTIATED node number is 5 THE LINK IDENTIFICATION NUMBER IS 23 ADJACENT node number is 17

*** APPLICATION TASK ACCEPTED ID node number is 6

COMMUNICATOR TASK INSTANTIATED node number is 6

***** TASK COMMUNICATOR FULLY CONFIGURED *** NODE IS 6

LINK TASK INSTANTIATED node number is 6 THE LINK IDENTIFICATION NUMBER IS 24 ADJACENT node number is 14

LINK TASK INSTANTIATED node number is 6 THE LINK IDENTIFICATION NUMBER IS 25 ADJACENT node number is 31

LINK TASK INSTANTIATED node number is 6 THE LINK IDENTIFICATION NUMBER IS 26 ADJACENT node number is 0

LINK TASK INSTANTIATED node number is 6 THE LINK IDENTIFICATION NUMBER IS 27 ADJACENT node number is 33

*** APPLICATION TASK ACCEPTED ID node number is 7

COMMUNICATOR TASK INSTANTIATED node number is 7

***** TASK COMMUNICATOR FULLY CONFIGURED *** NODE IS 7

LINK TASK INSTANTIATED node number is 7 THE LINK IDENTIFICATION NUMBER IS 28 ADJACENT node number is 18

LINK TASK INSTANTIATED node number is 7 THE LINK IDENTIFICATION NUMBER IS 29 ADJACENT node number is 35

LINK TASK INSTANTIATED node number is 7 THE LINK IDENTIFICATION NUMBER IS 30 ADJACENT node number is 4

LINK TASK INSTANTIATED node number is 7 THE LINK IDENTIFICATION NUMBER IS 31 ADJACENT node number is 25

*** APPLICATION TASK ACCEPTED ID node number is 8

COMMUNICATOR TASK INSTANTIATED node number is 8

***** TASK COMMUNICATOR FULLY CONFIGURED *** NODE IS 8

LINK TASK INSTANTIATED node number is 8 THE LINK IDENTIFICATION NUMBER IS 32 ADJACENT node number is 22

LINK TASK INSTANTIATED node number is 8 THE LINK IDENTIFICATION NUMBER IS 33 ADJACENT node number is 27

LINK TASK INSTANTIATED node number is 8 THE LINK IDENTIFICATION NUMBER IS 34 ADJACENT node number is 8

LINK TASK INSTANTIATED node number is 8 THE LINK IDENTIFICATION NUMBER IS 35 ADJACENT node number is 29

***** Getting a job for node 0
last job taken ... queue is empty now

JOB OBTAINED 0

Application about to attempt datagram build 0

Application built datagram 0

**>>>>Task communicator accepted message to transmit at node number ---> 0

*** LINK ACCEPTED DATAGRAM TO TRANSMIT *** LINK # IS 2

*** LINK RECEIVED DATAGRAM **** LINK # IS 12

**>>>>>Task communicator accepted message received at node number ---> 3

Datagram received by task communicator is to be forwarded at NODE number 3

*** LINK ACCEPTED DATAGRAM TO TRANSMIT *** LINK # IS14

*** LINK RECEIVED DATAGRAM **** LINK # IS 24

**>>>>>Task communicator accepted message received at node number ---> 6

Datagram received by task communicator is to be forwarded at NODE number 6

*** LINK ACCEPTED DATAGRAM TO TRANSMIT *** LINK # IS25

*** LINK RECEIVED DATAGRAM **** LINK # IS 31

**>>>>>Task communicator accepted message received at node number ---> 7

Datagram received by task communicator is to be forwarded at NODE number 7

*** LINK ACCEPTED DATAGRAM TO TRANSMIT *** LINK # IS29

*** LINK RECEIVED DATAGRAM **** LINK # IS 35

**>>>>>Task communicator accepted message received at node number ---> 8

Datagram received by task communicator reached final destination at NODE number 8

*** APPLICATION TASK ACCEPTED INPUT *** 8

DATAGRAM RECEIVED BY THE APPLICATION TASK AT NODE 8

******** DATAGRAM HEADER ********
Header type DATA_HDR
Source Node 0
Destination Node 8
Protocol Class TRANSPUTER_PROTOCOL
******** DATAGRAM DATA ********
Data value contained 8

***** Getting a job for node 8
last job taken ... queue is empty now

JOB OBTAINED 8

Application about to attempt datagram build 8

Application built datagram 8

**>>>>Task communicator accepted message to transmit at node number ---> 8

*** LINK ACCEPTED DATAGRAM TO TRANSMIT *** LINK # IS34

*** LINK RECEIVED DATAGRAM **** LINK # IS 8

**>>>>>Task communicator accepted message received at node number ---> 2

Datagram received by task communicator is to be forwarded at NODE number 2

*** LINK ACCEPTED DATAGRAM TO TRANSMIT *** LINK # IS10

*** LINK RECEIVED DATAGRAM **** LINK # IS 20

**>>>>>Task communicator accepted message received at node number ---> 5

Datagram received by task communicator is to be forwarded at NODE number 5

*** LINK ACCEPTED DATAGRAM TO TRANSMIT *** LINK # IS23

*** LINK RECEIVED DATAGRAM **** LINK # IS 17

**>>>>>Task communicator accepted message received at node number ---> 4

Datagram received by task communicator reached final destination at NODE number 4

***** Getting a job for node 3
last job taken ... queue is empty now

JOB OBTAINED 3

Application about to attempt datagram build 3

Application built datagram 3

**>>>>Task communicator accepted message to transmit at node number ---> 3

*** LINK ACCEPTED DATAGRAM TO TRANSMIT *** LINK # IS13

*** LINK RECEIVED DATAGRAM **** LINK # IS 19

***** Getting a job for node 7
last job taken ... queue is empty now

JOB OBTAINED 7

Application about to attempt datagram build 7

Application built datagram 7

**>>>>Task communicator accepted message to transmit at node number ---> 7

*** LINK ACCEPTED DATAGRAM TO TRANSMIT *** LINK # IS28

*** LINK RECEIVED DATAGRAM **** LINK # IS 18

*** APPLICATION TASK ACCEPTED INPUT *** 4

DATAGRAM RECEIVED BY THE APPLICATION TASK AT NODE 4

******** DATAGRAM HEADER ********
Header type DATA_HDR
Source Node 8
Destination Node 4
Protocol Class TRANSPUTER_PROTOCOL
******** DATAGRAM DATA ********
Data value contained 84

**>>>>>Task communicator accepted message received at node number ---> 4

Datagram received by task communicator is to be forwarded at NODE number 4

*** LINK ACCEPTED DATAGRAM TO TRANSMIT *** LINK # IS17

*** LINK RECEIVED DATAGRAM **** LINK # IS 23

**>>>>>Task communicator accepted message received at node number ---> 5

Datagram received by task communicator reached final destination at NODE number 5

*** APPLICATION TASK ACCEPTED INPUT *** 5

DATAGRAM RECEIVED BY THE APPLICATION TASK AT NODE 5

******** DATAGRAM HEADER ********
Header type DATA_HDR
Source Node 3
Destination Node 5
Protocol Class TRANSPUTER_PROTOCOL
******** DATAGRAM DATA ********
Data value contained 35

**>>>>>Task communicator accepted message received at node number ---> 4

Datagram received by task communicator is to be forwarded at NODE number 4

*** LINK ACCEPTED DATAGRAM TO TRANSMIT *** LINK # IS16

*** LINK RECEIVED DATAGRAM **** LINK # IS 6

**>>>>>Task communicator accepted message received at node number ---> 1

Datagram received by task communicator reached final destination at NODE number 1

*** APPLICATION TASK ACCEPTED INPUT *** 1

DATAGRAM RECEIVED BY THE APPLICATION TASK AT NODE 1

```
----------------------------------------------------------------------

******** DATAGRAM HEADER ********
Header type DATA_HDR
Source Node         7
Destination Node         1
Protocol Class TRANSPUTER_PROTOCOL
******** DATAGRAM DATA  ********
Data value contained         71

----------------------------------------------------------------------
```

EXAMPLE 2

Message passing in a mesh configuration

Input File for a Mesh Connection

Source Node	Operation	Destination Node	Data
0	send_to	8	08

Link Map for the Mesh

Node no.	Link No.	Link No.
0	1	7
0	2	12
1	5	11
1	6	16
1	7	1
2	10	20
2	11	5
3	12	2
3	13	19
3	14	24
4	16	6
4	17	23
4	18	28
4	19	13
5	20	10
5	22	32
5	23	17
6	24	14
6	25	31
7	28	18
7	29	35
7	31	25
8	32	22
8	35	29

Results of the Mesh Interconnection

```
 * * * LINK DISTRIBUTION CONFIGURED * * *
Job queue initialized
 All jobs inputted

printing status of the jobs_queue

status for node      0 is  true ...some jobs
status for node      1 is  false ... no jobs
status for node      2 is  false ... no jobs
status for node      3 is  false ... no jobs
status for node      4 is  false ... no jobs
status for node      5 is  false ... no jobs
status for node      6 is  false ... no jobs
status for node      7 is  false ... no jobs
                           status for node      8 is  false ... no jobs
PRINTING STATUS OF POINTERS

POINTER for node      0 is  NOT NULL ....some jobs
POINTER for node      1 is  NULL  ... no jobs
POINTER for node      2 is  NULL  ... no jobs
POINTER for node      3 is  NULL  ... no jobs
POINTER for node      4 is  NULL  ... no jobs
POINTER for node      5 is  NULL  ... no jobs
POINTER for node      6 is  NULL  ... no jobs
POINTER for node      7 is  NULL  ... no jobs
POINTER for node      8 is  NULL  ... no jobs
There are some jobs for this node      0
* _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ *
* _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ *

*** APPLICATION TASK  ACCEPTED ID  node number is       0

 COMMUNICATOR TASK INSTATIATED node number is  0

 ***** TASK COMMUNICATOR FULLY CONFIGURED *** NODE IS  0

 LINK TASK INSTANTIATED node number is  0  THE LINK IDENTIFICATION
NUMBER IS   0  ADYACENT  node number is NULL

 LINK TASK INSTANTIATED node number is  0  THE LINK IDENTIFICATION
NUMBER IS   1  ADYACENT  node number is  7

 LINK TASK INSTANTIATED node number is  0  THE LINK IDENTIFICATION
NUMBER IS   2  ADYACENT  node number is  12
```

LINK TASK INSTANTIATED node number is 0 THE LINK IDENTIFICATION NUMBER IS 3 ADYACENT node number is NULL

*** APPLICATION TASK ACCEPTED ID node number is 1

COMMUNICATOR TASK INSTANTIATED node number is 1

***** TASK COMMUNICATOR FULLY CONFIGURED *** NODE IS 1

LINK TASK INSTANTIATED node number is 1 THE LINK IDENTIFICATION NUMBER IS 4 ADYACENT node number is NULL

LINK TASK INSTANTIATED node number is 1 THE LINK IDENTIFICATION NUMBER IS 5 ADYACENT node number is 11

LINK TASK INSTANTIATED node number is 1 THE LINK IDENTIFICATION NUMBER IS 6 ADYACENT node number is 16

LINK TASK INSTANTIATED node number is 1 THE LINK IDENTIFICATION NUMBER IS 7 ADYACENT node number is 1

*** APPLICATION TASK ACCEPTED ID node number is 2

COMMUNICATOR TASK INSTANTIATED node number is 2

***** TASK COMMUNICATOR FULLY CONFIGURED *** NODE IS 2

LINK TASK INSTANTIATED node number is 2 THE LINK IDENTIFICATION NUMBER IS 8 ADYACENT node number is NULL

LINK TASK INSTANTIATED node number is 2 THE LINK IDENTIFICATION NUMBER IS 9 ADYACENT node number is NULL

LINK TASK INSTANTIATED node number is 2 THE LINK IDENTIFICATION NUMBER IS 10 ADYACENT node number is 20

LINK TASK INSTANTIATED node number is 2 THE LINK IDENTIFICATION NUMBER IS 11 ADYACENT node number is 5

*** APPLICATION TASK ACCEPTED ID node number is 3

COMMUNICATOR TASK INSTANTIATED node number is 3

***** TASK COMMUNICATOR FULLY CONFIGURED *** NODE IS 3

LINK TASK INSTANTIATED node number is 3 THE LINK IDENTIFICATION NUMBER IS 12 ADYACENT node number is 2

LINK TASK INSTANTIATED node number is 3 THE LINK IDENTIFICATION NUMBER IS 13 ADYACENT node number is 19

LINK TASK INSTANTIATED node number is 3 THE LINK IDENTIFICATION NUMBER IS 14 ADYACENT node number is 24

LINK TASK INSTANTIATED node number is 3 THE LINK IDENTIFICATION NUMBER IS 15 ADYACENT node number is 1

*** APPLICATION TASK ACCEPTED ID node number is 4

COMMUNICATOR TASK INSTANTIATED node number is 4

***** TASK COMMUNICATOR FULLY CONFIGURED *** NODE IS 4

LINK TASK INSTANTIATED node number is 4 THE LINK IDENTIFICATION NUMBER IS 16 ADYACENT node number is 6

LINK TASK INSTANTIATED node number is 4 THE LINK IDENTIFICATION NUMBER IS 17 ADYACENT node number is 23

LINK TASK INSTANTIATED node number is 4 THE LINK IDENTIFICATION NUMBER IS 18 ADYACENT node number is 28

LINK TASK INSTANTIATED node number is 4 THE LINK IDENTIFICATION NUMBER IS 19 ADYACENT node number is 13

*** APPLICATION TASK ACCEPTED ID node number is 5

COMMUNICATOR TASK INSTANTIATED node number is 5

***** TASK COMMUNICATOR FULLY CONFIGURED *** NODE IS 5

LINK TASK INSTANTIATED node number is 5 THE LINK IDENTIFICATION NUMBER IS 20 ADYACENT node number is 10

LINK TASK INSTANTIATED node number is 5 THE LINK IDENTIFICATION NUMBER IS 21 ADYACENT node number is 7

LINK TASK INSTANTIATED node number is 5 THE LINK IDENTIFICATION NUMBER IS 22 ADYACENT node number is 32

LINK TASK INSTANTIATED node number is 5 THE LINK IDENTIFICATION NUMBER IS 23 ADYACENT node number is 17

*** APPLICATION TASK ACCEPTED ID node number is 6

COMMUNICATOR TASK INSTANTIATED node number is 6

***** TASK COMMUNICATOR FULLY CONFIGURED *** NODE IS 6

LINK TASK INSTANTIATED node number is 6 THE LINK IDENTIFICATION NUMBER IS 24 ADYACENT node number is 14

LINK TASK INSTANTIATED node number is 6 THE LINK IDENTIFICATION NUMBER IS 25 ADYACENT node number is 31

LINK TASK INSTANTIATED node number is 6 THE LINK IDENTIFICATION NUMBER IS 26 ADYACENT node number is NULL

LINK TASK INSTANTIATED node number is 6 THE LINK IDENTIFICATION NUMBER IS 27 ADYACENT node number is NULL

*** APPLICATION TASK ACCEPTED ID node number is 7

COMMUNICATOR TASK INSTANTIATED node number is 7

***** TASK COMMUNICATOR FULLY CONFIGURED *** NODE IS 7

LINK TASK INSTANTIATED node number is 7 THE LINK IDENTIFICATION NUMBER IS 28 ADYACENT node number is 18

LINK TASK INSTANTIATED node number is 7 THE LINK IDENTIFICATION NUMBER IS 29 ADYACENT node number is 35

LINK TASK INSTANTIATED node number is 7 THE LINK IDENTIFICATION NUMBER IS 30 ADYACENT node number is NULL

LINK TASK INSTANTIATED node number is 7 THE LINK IDENTIFICATION NUMBER IS 31 ADYACENT node number is 25

*** APPLICATION TASK ACCEPTED ID node number is 8

COMMUNICATOR TASK INSTANTIATED node number is 8

***** TASK COMMUNICATOR FULLY CONFIGURED *** NODE IS 8

LINK TASK INSTANTIATED node number is 8 THE LINK IDENTIFICATION NUMBER IS 32 ADYACENT node number is 22

LINK TASK INSTANTIATED node number is 8 THE LINK IDENTIFICATION NUMBER IS 33 ADYACENT node number is 12

LINK TASK INSTANTIATED node number is 8 THE LINK IDENTIFICATION NUMBER IS 34 ADYACENT node number is 13

LINK TASK INSTANTIATED node number is 8 THE LINK IDENTIFICATION NUMBER IS 35 ADYACENT node number is 29

***** Getting a job for node 0
last job taken ... queue is empty now

JOB OBTAINED 0

Application about to attempt datagram build 0

Application built datagram 0

**>>>>>>Task communicator accepted message to transmit at node number ---> 0

*** LINK ACCEPTED DATAGRAM TO TRANSMIT *** LINK # IS 2

*** LINK RECEIVED DATAGRAM **** LINK # IS 12

**>>>>>>Task communicator accepted message received at node number ---> 3

Datagram received by task communicator is to be forwarded at NODE number 3

*** LINK ACCEPTED DATAGRAM TO TRANSMIT *** LINK # IS14

*** LINK RECEIVED DATAGRAM **** LINK # IS 24

**>>>>>>Task communicator accepted message received at node number ---> 6

Datagram received by task communicator is to be forwarded at NODE number 6

*** LINK ACCEPTED DATAGRAM TO TRANSMIT *** LINK # IS25

*** LINK RECEIVED DATAGRAM **** LINK # IS 31

**>>>>>>Task communicator accepted message received at node number ---> 7

Datagram received by task communicator is to be forwarded at NODE number 7

```
--------------------------------------------------------------------

*** LINK ACCEPTED DATAGRAM TO TRANSMIT *** LINK # IS29

*** LINK RECEIVED DATAGRAM **** LINK # IS 35

**>>>>>>Task communicator accepted message received  at node number ---> 8

--------------------------------------------------------------------

Datagram received by task communicator reached final destination at NODE number 8

--------------------------------------------------------------------

*** APPLICATION TASK ACCEPTED INPUT *** 8

--------------------------------------------------------------------
 DATAGRAM RECEIVED BY THE APPLICATION TASK AT NODE  8

--------------------------------------------------------------------

******** DATAGRAM  HEADER ********
Header  type DATA_HDR
Source Node        0
Destination Node        8
Protocol Class TRANSPUTER_PROTOCOL
******** DATAGRAM  DATA    ********
Data value contained          8

--------------------------------------------------------------------
```

APPENDIX B

Malek's Algorithm Results

* * * LINK DISTRIBUTION CONFIGURED * * *
Job queue initialized
All jobs inputted

printing status of the jobs_queue

status for node 0 is true ...some jobs
status for node 1 is true ...some jobs
status for node 2 is false ... no jobs
status for node 3 is true ...some jobs
status for node 4 is false ... no jobs
status for node 5 is false ... no jobs
status for node 6 is false ... no jobs
status for node 7 is false ... no jobs
status for node 8 is false ... no jobs
PRINTING STATUS OF POINTERS

POINTER for node 0 is NOT NULL ....some jobs
POINTER for node 1 is NOT NULL ....some jobs
POINTER for node 2 is NULL ... no jobs
POINTER for node 3 is NOT NULL ....some jobs
POINTER for node 4 is NULL ... no jobs
POINTER for node 5 is NULL ... no jobs
POINTER for node 6 is NULL ... no jobs
POINTER for node 7 is NULL ... no jobs
POINTER for node 8 is NULL ... no jobs
There are some jobs for this node 0
There are some jobs for this node 1
There are some jobs for this node 3
* _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ *
* _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ *

***** Getting a job for node 0

JOB OBTAINED 0

Application about to attempt packet build 0

Application built packet 0

**>>>>>Task communicator accepted message to transmit at node number ---> 0

*** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS 1

*** LINK RECEIVED PACKET **** LINK # IS 7

**>>>>>>Task communicator accepted message received at node number ---> 1

Packet received by task communicator reached final destination at NODE number 1

*** APPLICATION TASK ACCEPTED INPUT *** 1

PACKET RECEIVED BY THE APPLICATION TASK AT NODE 1

******** PACKET HEADER ********
Header type DATA_HDR
Source Node 0
Destination Node 1
Protocol Class TRANSPUTER_PROTOCOL
******** PACKET DATA ********
Data value contained 10

***** Getting a job for node 1
last job taken ... queue is empty now

JOB OBTAINED 1

Application about to attempt packet build 1

Application built packet 1

**>>>>>>Task communicator accepted message to transmit at node number ---> 1

*** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS 7

*** LINK RECEIVED PACKET **** LINK # IS 1

**>>>>>>Task communicator accepted message received at node number ---> 0

Packet received by task communicator reached final destination at NODE number 0

*** APPLICATION TASK ACCEPTED INPUT *** 0

PACKET RECEIVED BY THE APPLICATION TASK AT NODE 0

******** PACKET HEADER ********
Header type DATA_HDR
Source Node 1
Destination Node 0
Protocol Class TRANSPUTER_PROTOCOL
******** PACKET DATA ********
Data value contained 10

***** Getting a job for node 0
last job taken ... queue is empty now

JOB OBTAINED 0

Application about to attempt packet build 0

Application built packet 0

**>>>>>>Task communicator accepted message to transmit at node number ---> 0

*** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS 2

*** LINK RECEIVED PACKET **** LINK # IS 12

**>>>>>>Task communicator accepted message received at node number ---> 3

Packet received by task communicator reached final destination at NODE number 3

*** APPLICATION TASK ACCEPTED INPUT *** 3

PACKET RECEIVED BY THE APPLICATION TASK AT NODE 3

******** PACKET HEADER ********
Header type DATA_HDR
Source Node 0
Destination Node 3
Protocol Class TRANSPUTER_PROTOCOL
******** PACKET DATA ********
Data value contained 10

***** Getting a job for node 3
last job taken ... queue is empty now

JOB OBTAINED 3

Application about to attempt packet build 3

Application built packet 3

**>>>>>>Task communicator accepted message to transmit at node number ---> 3

*** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS12

*** LINK RECEIVED PACKET **** LINK # IS 2

**>>>>>>Task communicator accepted message received at node number ---> 0

Packet received by task communicator reached final destination at NODE number 0

*** APPLICATION TASK ACCEPTED INPUT *** 0

PACKET RECEIVED BY THE APPLICATION TASK AT NODE 0

******** PACKET HEADER ********
Header type DATA_HDR

Source Node 3
Destination Node 0
Protocol Class TRANSPUTER_PROTOCOL
******** PACKET DATA ********
Data value contained 10

```
 * * * LINK DISTRIBUTION CONFIGURED * * *
Job queue initialized
 All jobs inputted

printing status of the jobs_queue

status for node       0 is  true ...some jobs
status for node       1 is  false ... no jobs
status for node       2 is  false ... no jobs
status for node       3 is  true ...some jobs
status for node       4 is  true ...some jobs
status for node       5 is  false ... no jobs
status for node       6 is  false ... no jobs
status for node       7 is  false ... no jobs
status for node       8 is  false ... no jobs
PRINTING STATUS OF POINTERS

POINTER for node       0 is  NOT NULL ....some jobs
POINTER for node       1 is  NULL  ... no jobs
POINTER for node       2 is  NULL  ... no jobs
POINTER for node       3 is  NOT NULL ....some jobs
POINTER for node       4 is  NOT NULL ....some jobs
POINTER for node       5 is  NULL  ... no jobs
POINTER for node       6 is  NULL  ... no jobs
POINTER for node       7 is  NULL  ... no jobs
POINTER for node       8 is  NULL  ... no jobs
There are some jobs for this node       0
There are some jobs for this node       3
There are some jobs for this node       4
*  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *
* _* _* _* _* _* _* _* _* _* _* _* _* _* _* _* _* _* _*

***** Getting a job for node        3

JOB OBTAINED 3

Application about to attempt packet build  3

Application built  packet  3

**>>>>>>Task communicator accepted message to transmit at node number ---> 3

*** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS 12

*** LINK RECEIVED PACKET **** LINK # IS  2

**>>>>>>Task communicator accepted message received  at node number ---> 0
```

Packet received by task communicator reached final destination at NODE number 0

*** APPLICATION TASK ACCEPTED INPUT *** 0

PACKET RECEIVED BY THE APPLICATION TASK AT NODE 0

******** PACKET HEADER ********
Header type DATA_HDR
Source Node 3
Destination Node 0
Protocol Class TRANSPUTER_PROTOCOL
******** PACKET DATA ********
Data value contained 20

***** Getting a job for node 0
last job taken ... queue is empty now

JOB OBTAINED 0

Application about to attempt packet build 0

Application built packet 0

**>>>>>>Task communicator accepted message to transmit at node number ---> 0

*** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS 2

*** LINK RECEIVED PACKET **** LINK # IS 12

**>>>>>>Task communicator accepted message received at node number ---> 3

Packet received by task communicator reached final destination at NODE number 3

*** APPLICATION TASK ACCEPTED INPUT *** 3

```
----------------------------------------------------------------------
 PACKET RECEIVED BY THE APPLICATION TASK AT NODE 3

----------------------------------------------------------------------

******** PACKET HEADER ********
Header type DATA_HDR
Source Node        0
Destination Node        3
Protocol Class TRANSPUTER_PROTOCOL
******** PACKET DATA   ********
Data value contained        20

----------------------------------------------------------------------

***** Getting a job for node        3
last job taken ... queue is empty now

JOB OBTAINED 3

Application about to attempt packet build 3

Application built packet 3

**>>>>>>Task communicator accepted message to transmit at node number ---> 3

*** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS 13

*** LINK RECEIVED PACKET **** LINK # IS 19

**>>>>>>Task communicator accepted message received at node number ---> 4

----------------------------------------------------------------------

Packet received by task communicator reached final destination at NODE number 4

----------------------------------------------------------------------

*** APPLICATION TASK ACCEPTED INPUT *** 4


----------------------------------------------------------------------
 PACKET RECEIVED BY THE APPLICATION TASK AT NODE 4

----------------------------------------------------------------------
```

******** PACKET HEADER ********
Header type DATA_HDR
Source Node 3
Destination Node 4
Protocol Class TRANSPUTER_PROTOCOL
******** PACKET DATA ********
Data value contained 20

***** Getting a job for node 4
last job taken ... queue is empty now

JOB OBTAINED 4

Application about to attempt packet build 4

Application built packet 4

**>>>>>>Task communicator accepted message to transmit at node number ---> 4

*** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS19

*** LINK RECEIVED PACKET **** LINK # IS 13

**>>>>>>Task communicator accepted message received at node number ---> 3

Packet received by task communicator reached final destination at NODE number 3

*** APPLICATION TASK ACCEPTED INPUT *** 3

PACKET RECEIVED BY THE APPLICATION TASK AT NODE 3

```
******** PACKET HEADER ********
Header type DATA_HDR
Source Node        4
Destination Node       3
Protocol Class TRANSPUTER_PROTOCOL
******** PACKET DATA   ********
Data value contained       20

------------------------------------------------------------------------
```

```
 * * * LINK DISTRIBUTION CONFIGURED * * *
Job queue initialized
 All jobs inputted

printing status of the jobs_queue

status for node      0 is  false ... no jobs
status for node      1 is  false ... no jobs
status for node      2 is  false ... no jobs
status for node      3 is  false ... no jobs
status for node      4 is  true ...some jobs
status for node      5 is  true ...some jobs
status for node      6 is  false ... no jobs
status for node      7 is  true ...some jobs
status for node      8 is  false ... no jobs
PRINTING STATUS OF POINTERS

POINTER for node      0 is  NULL  ... no jobs
POINTER for node      1 is  NULL  ... no jobs
POINTER for node      2 is  NULL  ... no jobs
POINTER for node      3 is  NULL  ... no jobs
POINTER for node      4 is  NOT NULL ....some jobs
POINTER for node      5 is  NOT NULL ....some jobs
POINTER for node      6 is  NULL  ... no jobs
POINTER for node      7 is  NOT NULL ....some jobs
POINTER for node      8 is  NULL  ... no jobs
There are some jobs for this node      4
There are some jobs for this node      5
There are some jobs for this node      7
*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*
*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*_*

***** Getting a job for node      4

JOB OBTAINED 4

Application about to attempt packet build  4

Application built  packet  4

**>>>>>>Task communicator accepted message to transmit at node number ---> 4

*** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS 17

*** LINK RECEIVED PACKET **** LINK # IS  23

**>>>>>>Task communicator accepted message received  at node number ---> 5
```

Packet received by task communicator reached final destination at NODE number 5

*** APPLICATION TASK ACCEPTED INPUT *** 5

PACKET RECEIVED BY THE APPLICATION TASK AT NODE 5

******** PACKET HEADER ********
Header type DATA_HDR
Source Node 4
Destination Node 5
Protocol Class TRANSPUTER_PROTOCOL
******** PACKET DATA ********
Data value contained 30

***** Getting a job for node 5
last job taken ... queue is empty now

JOB OBTAINED 5

Application about to attempt packet build 5

Application built packet 5

**>>>>>>Task communicator accepted message to transmit at node number ---> 5

*** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS23

*** LINK RECEIVED PACKET **** LINK # IS 17

**>>>>>>Task communicator accepted message received at node number ---> 4

Packet received by task communicator reached final destination at NODE number 4

*** APPLICATION TASK ACCEPTED INPUT *** 4

PACKET RECEIVED BY THE APPLICATION TASK AT NODE 4

******** PACKET HEADER ********
Header type DATA_HDR
Source Node 5
Destination Node 4
Protocol Class TRANSPUTER_PROTOCOL
******** PACKET DATA ********
Data value contained 30

***** Getting a job for node 4
last job taken ... queue is empty now

JOB OBTAINED 4

Application about to attempt packet build 4

Application built packet 4

**>>>>>>Task communicator accepted message to transmit at node number ---> 4

*** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS18

*** LINK RECEIVED PACKET **** LINK # IS 28

**>>>>>>Task communicator accepted message received at node number ---> 7

Packet received by task communicator reached final destination at NODE number 7

*** APPLICATION TASK ACCEPTED INPUT *** 7

PACKET RECEIVED BY THE APPLICATION TASK AT NODE 7

******** PACKET HEADER ********
Header type DATA_HDR
Source Node 4
Destination Node 7
Protocol Class TRANSPUTER_PROTOCOL
******** PACKET DATA ********
Data value contained 30

***** Getting a job for node 7
last job taken ... queue is empty now

JOB OBTAINED 7

Application about to attempt packet build 7

Application built packet 7

**>>>>>>Task communicator accepted message to transmit at node number ---> 7

*** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS28

*** LINK RECEIVED PACKET **** LINK # IS 18

**>>>>>>Task communicator accepted message received at node number ---> 4

Packet received by task communicator reached final destination at NODE number 4

*** APPLICATION TASK ACCEPTED INPUT *** 4

PACKET RECEIVED BY THE APPLICATION TASK AT NODE 4

```
******** PACKET HEADER ********
Header type DATA_HDR
Source Node        7
Destination Node        4
Protocol Class TRANSPUTER_PROTOCOL
******** PACKET DATA   ********
Data value contained        30

----------------------------------------------------------------------
```

```
 * * * LINK DISTRIBUTION CONFIGURED * * *
Job queue initialized
 All jobs inputted

printing status of the jobs_queue

status for node      0 is  false ... no jobs
status for node      1 is  false ... no jobs
status for node      2 is  false ... no jobs
status for node      3 is  false ... no jobs
status for node      4 is  false ... no jobs
status for node      5 is  false ... no jobs
status for node      6 is  true ...some jobs
status for node      7 is  true ...some jobs
status for node      8 is  true ...some jobs
PRINTING STATUS OF POINTERS

POINTER for node      0 is  NULL  ... no jobs
POINTER for node      1 is  NULL  ... no jobs
POINTER for node      2 is  NULL  ... no jobs
POINTER for node      3 is  NULL  ... no jobs
POINTER for node      4 is  NULL  ... no jobs
POINTER for node      5 is  NULL  ... no jobs
POINTER for node      6 is  NOT NULL ....some jobs
POINTER for node      7 is  NOT NULL ....some jobs
POINTER for node      8 is  NOT NULL ....some jobs
There are some jobs for this node       0
There are some jobs for this node       1
There are some jobs for this node       3
* _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ *
* _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ *

***** Getting a job for node       7

JOB OBTAINED  7

Application about to attempt packet build  7

Application built  packet  7

**>>>>>>Task communicator accepted message to transmit at node number ---> 7

*** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS29

*** LINK RECEIVED PACKET **** LINK # IS  35

**>>>>>>Task communicator accepted message received  at node number ---> 8
```

Packet received by task communicator reached final destination at NODE number 8

*** APPLICATION TASK ACCEPTED INPUT *** 8

PACKET RECEIVED BY THE APPLICATION TASK AT NODE 8

******** PACKET HEADER ********
Header type DATA_HDR
Source Node 7
Destination Node 8
Protocol Class TRANSPUTER_PROTOCOL
******** PACKET DATA ********
Data value contained 40

***** Getting a job for node 8
last job taken ... queue is empty now

JOB OBTAINED 8

Application about to attempt packet build 8

Application built packet 8

**>>>>>>Task communicator accepted message to transmit at node number ---> 8

*** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS35

*** LINK RECEIVED PACKET **** LINK # IS 29

**>>>>>>Task communicator accepted message received at node number ---> 7

Packet received by task communicator reached final destination at NODE number 7

*** APPLICATION TASK ACCEPTED INPUT *** 7

PACKET RECEIVED BY THE APPLICATION TASK AT NODE 7

******** PACKET HEADER ********
Header type DATA_HDR
Source Node 8
Destination Node 7
Protocol Class TRANSPUTER_PROTOCOL
******** PACKET DATA ********
Data value contained 40

***** Getting a job for node 7
last job taken ... queue is empty now

JOB OBTAINED 7

Application about to attempt packet build 7

Application built packet 7

**>>>>>>Task communicator accepted message to transmit at node number ---> 7

*** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS31

*** LINK RECEIVED PACKET **** LINK # IS 25

**>>>>>>Task communicator accepted message received at node number ---> 6

Packet received by task communicator reached final destination at NODE number 6

*** APPLICATION TASK ACCEPTED INPUT *** 6

PACKET RECEIVED BY THE APPLICATION TASK AT NODE 6

```
******** PACKET HEADER ********
Header type DATA_HDR
Source Node        7
Destination Node        6
Protocol Class TRANSPUTER_PROTOCOL
******** PACKET DATA   ********
Data value contained        40

----------------------------------------------------------------------

***** Getting a job for node        6
last job taken ... queue is empty now

JOB OBTAINED 6

Application about to attempt packet build 6

Application built packet 6

**>>>>>>Task communicator accepted message to transmit at node number ---> 6

*** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS25

*** LINK RECEIVED PACKET **** LINK # IS 31

**>>>>>>Task communicator accepted message received at node number ---> 7

----------------------------------------------------------------------

Packet received by task communicator reached final destination at NODE number 7

----------------------------------------------------------------------

*** APPLICATION TASK ACCEPTED INPUT *** 7

----------------------------------------------------------------------
 PACKET RECEIVED BY THE APPLICATION TASK AT NODE 7

----------------------------------------------------------------------
```

```
******** PACKET HEADER ********
Header type DATA_HDR
Source Node        6
Destination Node        7
Protocol Class TRANSPUTER_PROTOCOL
******** PACKET DATA   ********
Data value contained        40
```

```
 * * * LINK DISTRIBUTION CONFIGURED * * *
Job queue initialized
 All jobs inputted

printing status of the jobs_queue

status for node       0 is  false ... no jobs
status for node       1 is  true ...some jobs
status for node       2 is  true ...some jobs
status for node       3 is  false ... no jobs
status for node       4 is  true ...some jobs
status for node       5 is  false ... no jobs
status for node       6 is  false ... no jobs
status for node       7 is  false ... no jobs
status for node       8 is  false ... no jobs
PRINTING STATUS OF POINTERS

POINTER for node        0 is  NULL  ... no jobs
POINTER for node        1 is  NOT NULL ....some jobs
POINTER for node        2 is  NOT NULL ....some jobs
POINTER for node        3 is  NULL  ... no jobs
POINTER for node        4 is  NOT NULL ....some jobs
POINTER for node        5 is  NULL  ... no jobs
POINTER for node        6 is  NULL  ... no jobs
POINTER for node        7 is  NULL  ... no jobs
POINTER for node        8 is  NULL  ... no jobs
There are some jobs for this node        1
There are some jobs for this node        2
There are some jobs for this node        4
* _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ *
* _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ * _ *

***** Getting a job for node        1

JOB OBTAINED 1

Application about to attempt packet build  1

Application built  packet  1

**>>>>>>Task communicator accepted message to transmit at node number ---> 1

*** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS 5

*** LINK RECEIVED PACKET **** LINK # IS  11

**>>>>>>Task communicator accepted message received  at node number ---> 1
```

Packet received by task communicator reached final destination at NODE number 2

*** APPLICATION TASK ACCEPTED INPUT *** 2

PACKET RECEIVED BY THE APPLICATION TASK AT NODE 2

******** PACKET HEADER ********
Header type DATA_HDR
Source Node 1
Destination Node 2
Protocol Class TRANSPUTER_PROTOCOL
******** PACKET DATA ********
Data value contained 50

***** Getting a job for node 2
last job taken ... queue is empty now

JOB OBTAINED 2

Application about to attempt packet build 2

Application built packet 2

**>>>>>>Task communicator accepted message to transmit at node number ---> 2

*** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS11

*** LINK RECEIVED PACKET **** LINK # IS 5

**>>>>>>Task communicator accepted message received at node number ---> 1

Packet received by task communicator reached final destination at NODE number 1

*** APPLICATION TASK ACCEPTED INPUT *** 1

PACKET RECEIVED BY THE APPLICATION TASK AT NODE 1

******** PACKET HEADER ********
Header type DATA_HDR
Source Node 2
Destination Node 1
Protocol Class TRANSPUTER_PROTOCOL
******** PACKET DATA ********
Data value contained 52 -- Data received is wrong

***** Getting a job for node 1
last job taken ... queue is empty now

JOB OBTAINED 1

Application about to attempt packet build 1

Application built packet 1

**>>>>>>Task communicator accepted message to transmit at node number ---> 1

*** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS 6

*** LINK RECEIVED PACKET **** LINK # IS 16

**>>>>>>Task communicator accepted message received at node number ---> 4

Packet received by task communicator reached final destination at NODE number 4

*** APPLICATION TASK ACCEPTED INPUT *** 4

PACKET RECEIVED BY THE APPLICATION TASK AT NODE 4

******** PACKET HEADER ********
Header type DATA_HDR
Source Node 1
Destination Node 4
Protocol Class TRANSPUTER_PROTOCOL
******** PACKET DATA ********
Data value contained 50

***** Getting a job for node 4
last job taken ... queue is empty now

JOB OBTAINED 4

Application about to attempt packet build 4

Application built packet 4

**>>>>>>Task communicator accepted message to transmit at node number ---> 4

*** LINK ACCEPTED PACKET TO TRANSMIT *** LINK # IS16

*** LINK RECEIVED PACKET **** LINK # IS 6

**>>>>>>Task communicator accepted message received at node number ---> 1

Packet received by task communicator reached final destination at NODE number 1

*** APPLICATION TASK ACCEPTED INPUT *** 1

PACKET RECEIVED BY THE APPLICATION TASK AT NODE 1

```
******** PACKET HEADER ********
Header type DATA_HDR
Source Node        4
Destination Node        1
Protocol Class TRANSPUTER_PROTOCOL
******** PACKET DATA  ********
Data value contained        50

--------------------------------------------------------------------
```

APPENDIX C

Configuration Code

(I) Searching a tree of transputers

If a transputer is booted on link `parentlink`, then the algorithm discussed in Chapter 4 may be expressed as follows:

```
SEQ
  SEQ I = 0 FOR 4
    downLoad[I]:= FALSE
  nTransputers:=  LoadingData[2]
  id          :=  nTransputers
  nTransputers:=  nTransputers + 1
  SEQ I = 0 FOR 4 -- Try each link in turn
    IF
       I = parentLink
         SKIP
       TRUE
         SEQ
            stage     := 1
            waiting   := FALSE
            badOut    := FALSE
            ... Search neighboring transputer (set
                                       waiting)  (i)
            ... Boot neighbour,and wait while config
                                       explores (iii)

LinkOut[parentLink] ! ReturnControl.t; nTransputers
```

Note:

(i) Testing a neighbour:

```
SEQ
  OutputToken.t (LinkOut[I], 0 (BYTE), Delay, badOut) -- (ii)
  OutputInt.t   (LinkOut[I], MinInt,   Delay, badOut)
  OutputInt.t   (LinkOut[I], MinInt,   Delay, badOut)
  OutputToken.t (LinkOut[I], 1 (BYTE), Delay, badOut)
  OutputInt.t   (LinkOut[I], MinInt,   Delay, badOut)

  Clock ? time
  ALT
    LinkIn[I] ? token   -- Value returned
      SEQ
        stage := 2
        waiting := TRUE
    Clock ? AFTER time PLUS Delay
      SKIP
```

Note that the return of the value `MinInt` indicates that a successful write and read has taken place (the boolean `badOut` also indicates that this transputer has output the write and read). `waiting` is now set to true and the algorithm enters the nest loop.

(ii) The process `OutputToken.t`, `OutputInt.t`, `OutputString.t` are based on the output or fail routine. For example:

```
PROC OutputToken.t (CHAN OF ANY ToLink, VAL BYTE Token,
                              VAL INT Delay, BOOL stopping)
  INT time :
  TIMER Clock :
  VAL [1]  BYTE String RETYPES Token :
```

```
IF
  stopping
    SKIP
  TRUE
    SEQ
      Clock ? time
      time := time PLUS Delay
      OutputOrFail.t (ToLink,String,Clock,time,stopping) :
```

(iii) Given the success of (i) (`waiting` is set to TRUE), now try to boot the neighbouring transputer:

```
SEQ
  ... Try to boot neighbouring transputer
  WHILE waiting   -- config explores branch off neighbour
    LinkIn[I] ? token
      CASE token
         ... LoadingData.t
         ... ReturnControl.t
```

Booting is performed as follows:

```
VAL [] BYTE InitialData RETYPES [Id, I, nTransputers, 0] :
VAL Program IS [programTable FROM 0 FOR programLength] :
SEQ
  OutputString.t (LinkOut[I],          Program,  Delay, badOut)
  OutputInt.t    (LinkOut[I],     SIZE Program,  Delay, badOut)
  OutputString.t (LinkOut[I],          Program,  Delay, badOut)
  OutputInt.t    (LinkOut[I],     LoadingData.t, Delay, badOut)
  OutputString.t (LinkOut[I],  InitialData,      Delay, badOut)
```

(iv) The `loadingData` is returned to the host (for immediate display) and is acknowledged by the token `Syncronize.t`. On receipt of the data, the host process returns the token `Syncronize.t`. This synchronization is important, for

it guarantees that all transputers at stage 3 are ready to be probed on any link J, and are not still engaged in returning `loadingData`.

```
LoadingData.t
  [LoadingDataLength] INT passOnData :
  SEQ
    LinkIn[I]                  ? passOnData
    LinkOut[parentLink]        ! LoadingData.t; passOnData
    LinkIn[parentLink]         ? token      -- Synchronize.t
    LinkOut[I]                 ! Synchronize.t
    stage := 3
```

(v) The return of control indicates that the tree off link `I` has been completely explored. This process may now explore other links.

```
ReturnControl.t
  SEQ
    LinkIn[I] ? nTransputers
    downLoad[I] := TRUE
    waiting := FALSE
```

The searching procedure is initiated by `PROC Tracer` booting the first transputer in the tree, and telling it that `nTransputers = 0`. When that transputer finally returns control to `Tracer`, the total number of transputers in the network will be returned, and the network will have been completely searched.

(II) Searching a general network of transputers

The central part of the program looks like this:

```
SEQ
  ... Initialize downLoad, id, nTransputers as before
  ... Initialize tryLink, linkArray                         (i)
  SEQ I = 0 FOR 4
    IF
```

```
        NOT tryLink[I]
          SKIP
        TRUE
         SEQ
            stage  := 1
            waiting:= FALSE
            badOut := FALSE
           SEQ
            ...    Initialize as before
            ...    Search neighbour                      (ii)
            ...    Boot neighbour, and wait for reply    (iv)
            tryLink[I] := FALSE
   LinkOut[parentLink] ! ReturnControl.t; nTransputers
```

(i) Initialize `tryLink[I]` to `TRUE` for all links except the link back to the parent. The elements 0 and 1 of the array `loadingData` contain the identity and link of the parent transputer.

```
SEQ I = 0 FOR 4
  tryLink[I] := TRUE
tryLink[parentLink] := FALSE
linkArray[parentLink] := [loadingData FROM 0 FOR 2]
```

(ii) There is now the possibility that two links on the same transputer are connected. Hence, the read and write must be done in parallel to listening on all other links:

```
PAR
  ... Search neighbouring transputer
  SEQ
    Clock ? time
    ALT
      ALT J = 0 FOR Nlinks
        (J <> I) AND tryLink[J] & LinkIn[J] ? searchString
          SEQ
```

```
            linkArray[J] := [id, I]
            linkArray[I] := [id, J]
            tryLink[J]   := False
      LinkIn[I] ? token
        CASE token
          ...     MinInt as before
          ...     AlreadyLoaded
          ...     ELSE  -- error                  (iii)
      ... Time out as before                      (vi)
```

(iii) If there is a closed loop, we get the situation that one transputer probes another which replies `AlreadyLoaded.t`. The two ends then exchange the id and link.

```
PAR
  LinkOut[link] ! [id, link]
  LinkIn[link]  ? linkArray[link]
```

(iv) As before, waiting is only set to be true if a neighboring transputer has been found. The case when two links are connected on the same transputer need not be considered.

```
SEQ
  ... Try to boot neighbouring transputer as before
  WHILE waiting
    SEQ
      Clock ? time
      ALT
        ALT J = 0 FOR Nlinks
          (J <> I) AND tryLink[J] & LinkIn[J] ? searchString
            ...   Reply 'AlreadyLoaded.t'            (iii)
        LinkIn ? token
          CASE token
            ...   LoadingData.t                      (v)
            ...   ReturnControl.t   (as in the case for a tree)
```

```
        ...   ELSE  -- error                          (vi)
    ...   Time out                                    (vii)
```

(v) In addition to passing the loading data back, a track of the children id's boot link is also kept:

```
IF
  stage = 2
    linkArray[I] := [passOnData FROM 2 FOR 2]
  TRUE
    SKIP
```

(vi) A bad communication has taken place on this link by making a record in `linkArray`. We use a special token `TokenError.v` to indicate that an unexpected token has been returned.

```
SEQ
  waiting := FALSE
  linkArray [I] := [stage, TokenError.v]
```

(vii) A timeout at stage 1 implies that the link is unattached. However, if a timeout occurs at a later stage, assuming Delay is long enough to allow for the booting of a child, then the neighbor has not yet been successfully loaded. We report this as an error.

```
Clock ? AFTER time PLUS Delay
  SEQ
    linkArray[I] := [stage, TimeOutError.v]
    waiting := FALSE
```

Returning the Local Link Map

Having explored the local connections of each link on a transputer, and returned control to the parent, we send back the information `linkArray` back to the host transputer.

```
CHAN OF ANY ToParent IS LinkOut[parentLink] :
```

```
SEQ
  stage := 4
  ToParent ! NetworkData.t; id; linkArray

  SEQ I = 0 FOR 4
    IF
      NOT  download[I]
        SKIP
      download[I]--Pass on network info from daughter processes
        SEQ
          reading := TRUE
          WHILE reading
            SEQ
              LinkIn[I] ? token
              CASE token
                ...     NetworkData.t          (i)
                ...     NoMoreData.t           (ii)
                ...     ELSE                   (iii)
 ToParent ! NoMoreData.t
```

(i) Pass on the identity and link array.

```
NetworkData.t     -- pass on id and info
  INT passOnId :
  [4] [2] INT passOnLinkArray :
  SEQ
    LinkIn[I] ? passOnId; passOnLinkArray
    ToParent ! NetworkData.t; passOnId; passOnLinkArray
```

(ii) There is no more data to transmit from this branch

```
NoMoreData.t
  reading := FALSE
```

(iii) This is an error. Return a modified `linkArray` report.

```
ELSE
  SEQ
    reading := FALSE
    linkArray[I] := [stage, TokenError.v]
    ToParent ! NetworkData.t; id; linkArray
```