View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Lehigh University: Lehigh Preserve

Lehigh University
Lehigh Preserve

Theses and Dissertations

1995

Design and implementation of a modular deadlock-free routing for a mesh network of transputers

Syed H. Kirmani
Lehigh University

Follow this and additional works at: http://preserve.lehigh.edu/etd

Recommended Citation
Kirmani, Syed H., "Design and implementation of a modular deadlock-free routing for a mesh network of transputers" (1995). *Theses and Dissertations.* Paper 332.

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.

AUTHOR:

Kirmani, Syed H.

TITLE:

Design and Implementation of a Modular Deadlock-Free Routing for a Mesh Network of Transputers

DATE: May 28, 1995

Design and Implementation of a Modular Deadlock-free Routing for a Mesh network of transputers

by

Syed H. Kirmani

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

December 1995

This thesis is accepted and approved in partial fulfillment of the degree of Master of Science in Computer Science.

12/6/94

Date

[illegible]

Thesis Advisor

EECS Department Chairperson

Table Of Contents

List Of Figures

Abstract

The availability of a reliable routing algorithm for a transputer based parallel system is of elementary importance. The difficulty in designing such an algorithm lies in the occurance of deadlocks in blocking message passing processes. The situation of several processes waiting indefinitly is called *deadlock*. Deadlock is usually discussed in the multi-programming environment where several processes may compete for a finite number of resources. One way to break this deadlock is for the system to take extreme action and preempt the resources held by the processes. In general, detecting and recovering from deadlocks is difficult, and the best strategy is to have policies and strategies that avoid deadlock.

The topic of this thesis is the design and implementation of a modular deadlock_free routing algorithm for a two-dimensional mesh, on a network of transputers of any size. Various routing techniques and topologies are first discussed to get a better understanding of the nature of deadlocking. This thesis also describes the difficulties in avoiding deadlock situations. The routing algorithm is implemented on a network of transputers using the OCCAM language.

Chapter 1: Introduction

Message routing in interconnection networks has been the subject of constant interest in recent years. Various techniques have been used to route a message through a network. Of even a greater interest has been the problem of deadlocking, which can effect communication latency. A deadlock is a situation, in which a set of processes are allowed to hold some resources while requesting others [1].

In order to address the issue of designing routing algorithms, several factors have to be considered such as routing techniques, type of architecture and topologies and flow control strategies.

Two major techniques are used in routing: source and distributed routing. Each of the techniques has its advantage over the other and is discussed in detail in this thesis. Another classification of routing is deterministic or adaptive routing.

A network consists of many channels and buffers. A flow control strategy deals with the allocation of channels and buffers to a packet as it travels along a path through the network. What action is to be taken when a packet is blocked, depends on the flow control policy.

In this chapter, concurrent processing is first introduced. A brief description of various interconnection topologies and switching techniques are then presented.

1.1 Concurrent Processing

Concurrent processing is an ancient concept and has been an important factor in the formation and development of societies on earth. With the advancement in computer technologies, as the problems become more complex, computers have been gaining both in speed and data storage capabilities. In the near future, there seems to be no realistic approach to substantially increasing the performance of individual computers ; technology is already nearing limits set by the speed of light and quantum physics effects. The general agreement is that the only route to significantly increase performance is through concurrent-computation i.e the use of many computers together to solve the same problem [2].

Concurrent Processing is defined as the "use of several working entities, working together toward a common goal" [2]. In concurrent computation, the entities are computers and the goal might be a complex scientific computation.

1.2 Interconnection Topologies

Since in Concurrent Processing more than one processor is used, an immediate question is how to connect the processors to achieve the desired speed-up in a way that is both economically and physically feasible, since the network can easily dominate the hardware cost and program execution time [3]. The interconnection network in a parallel computer (a computer with multiple processors) specifies how the processors are connected together. The processors communicate by sending information through the network.

1.2.1 Shared vs Distributed memory

Parallel Computers are characterized by shared or distributed memory organizations. The shared memory machines feature a common memory that can be accessed by all the processing elements. The simplest design is shown in Figure 1.1 [2]. This type of design uses a common bus or a communication channel to allow the individual processors to access the shared memory. Such a design is particularly appropriate if N, the number of processing elements, is small [2].

In distributed memory machines, the basic processing element includes local memory to the exclusion of shared or global memory. "Distributed memory machines go hand-in-hand with the message passing model for concurrent computation" [2]. In distributed memory machines, messages can only be exchanged directly by two nodes (a node is a processor with local memory) that have a direct connection of some sort with each other. Messages from other nodes can also be passed through intermediate nodes, which then forward the message to their neighboring nodes. The distributed memory system has an advantage over the shared memory system in that the bottleneck or memory conflict caused by the access of one memory location by multiple processes, as in shared memory, is not encountered.

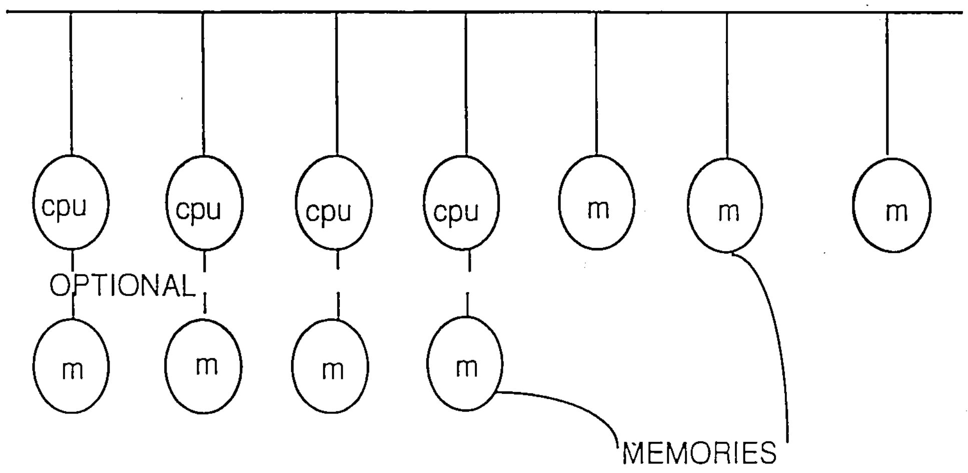


Figure 1.1 A Shared Memory Computer With A Common Bus

1.2.2 Interconnection and Node Structures

Each node of a network contains a switching element (SE) responsible for interprocess communications. Other processors are then attached either to this SE or to the I/O ports of the processor. If a processor wants to communicate, or send some information to another processor, it sends the request to it's SE, and from there the information is passed on to the neighboring processor's SE, until it reaches its destination. The destination processor or memory module finally removes this information from the network [3].

Indirect and Direct networks

There are basically two types of networks, indirect and direct [3]. In indirect networks, processors and memory modules are attached only to the I/O ports of the SE. There are two major configurations:

a) processor-to-memory: In this method, the network is put in between the arrays of processors and memory modules, and communications between processes are achieved by processors sending messages through the network to shared variables in the memory modules [3]. The main advantage of such an architecture is the ability to share large blocks of data and to vary the amount of memory used by each processor. This is depicted in Figure 1.2, where processors on the left send messages through the network, to the memory modules on the right side of the network.

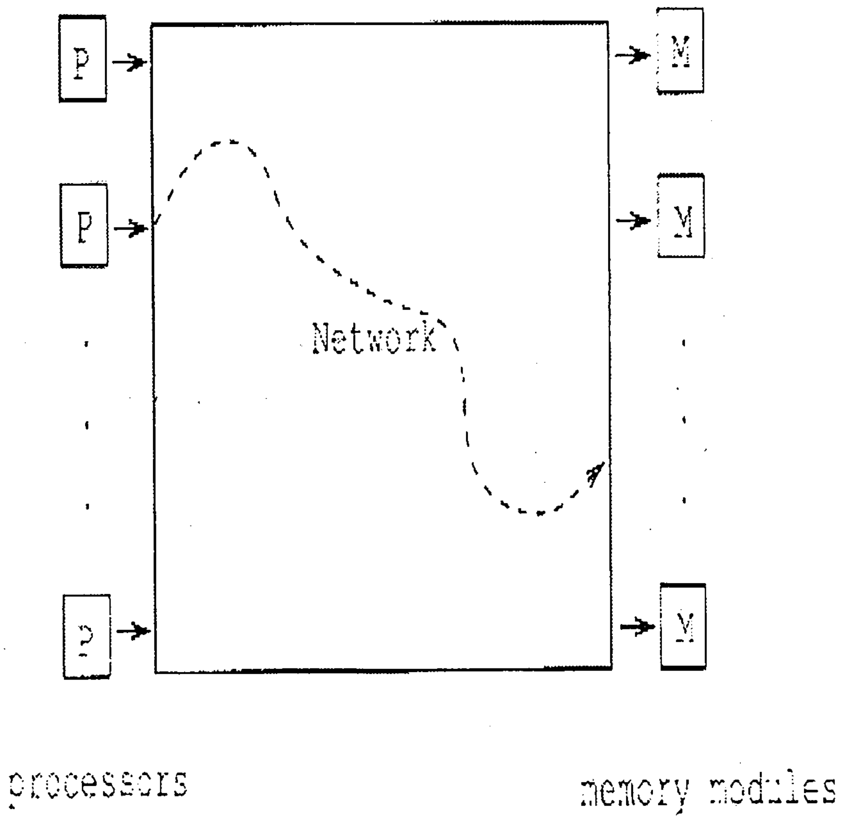


Figure 1.2 Process To Memory Organization [3]

b) PE-to-PE: In this configuration, each processor is paired with a memory module forming a processing element (PE), which is then attached to one of the input ports and one of the output ports of the network. The main advantage of such a structure is fast local memory reference. An example of PE-to-PE configuration is shown in Figure 1.3. A memory module and a processor are paired together to form a processing element (PE), which is then connected to the network.

In direct networks, each node is attached to by a PE . Hence each PE is connected directly to a number of PE's via neighboring SE's. Figure 1.4 shows an example of such a structure on a hypercube. Each SE is connected to its neighboring SE's via links.

Indirect interconnection schemes are used for shared memory models, while the direct structures have been used mainly for message-passing architectures. The indirect approach is preffered for systems with a few number of processors, while the direct approach is preferred for systems with hundreds or thousands of processors [3].

1.2.3 Switching Techniques

Switching refers to the way an SE switches data from one link to another, as it is routed to a destination node. There are four main approaches to switching: circuit switching, store and forward, virtual cut-through and wormhole routing.

Circuit Switching

In this approach, a physical circuit is established between the source and

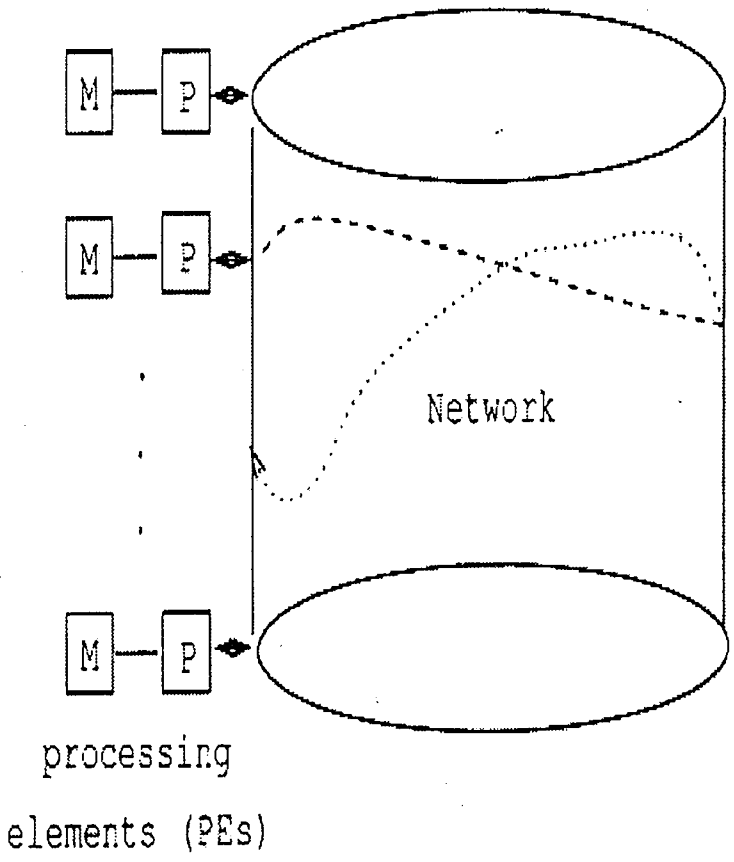


Figure 1.3 PE-to-PE Organization [3]

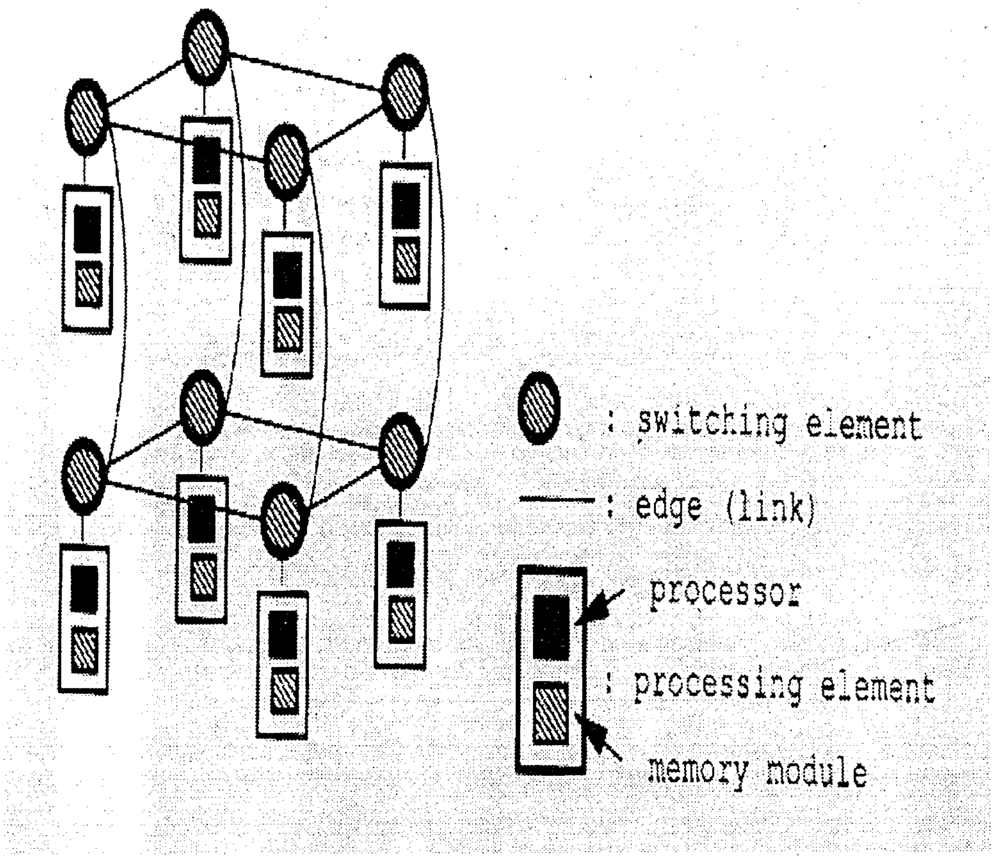


Figure 1.4 An Example of a Direct Network Architecture [3]

destination nodes during the circuit establishment phase [1]. Once the path is established, the SE's on the path remain dedicated until the path is released. The Intel PSC/2 uses circuit switching for data.

Store and Forward

In this approach, a logical unit of data called message makes its way from SE to SE, releasing links and SE's immediately after using them. The advantages of such a technique is greater line efficiency and no need for simultaneous availability of the sender and receiver. This method is used in iPSC-1, Ncube 1, and FPS T-series. A major drawback is that each node has to buffer every incoming packet consuming memory space [9].

Virtual Cut-through

In this approach, a packet is stored at an intermediate node only if the next required channel is busy [12].

Wormhole routing

In this approach, a packet is divided into a number of flits(flow control digits) for transmission. The header flit governs the route. As the header flit advances along the route, the remaining flits follow in a pipelined manner. If the header flit encounters a busy channel, it is blocked until the channel becomes available, and the rest of the flits remain in the flit buffers along the established route [10].

Figure 1.4 [1] compares the communication latency of wormhole routing with that of store and forward and circuit switching in a contention-free network. The

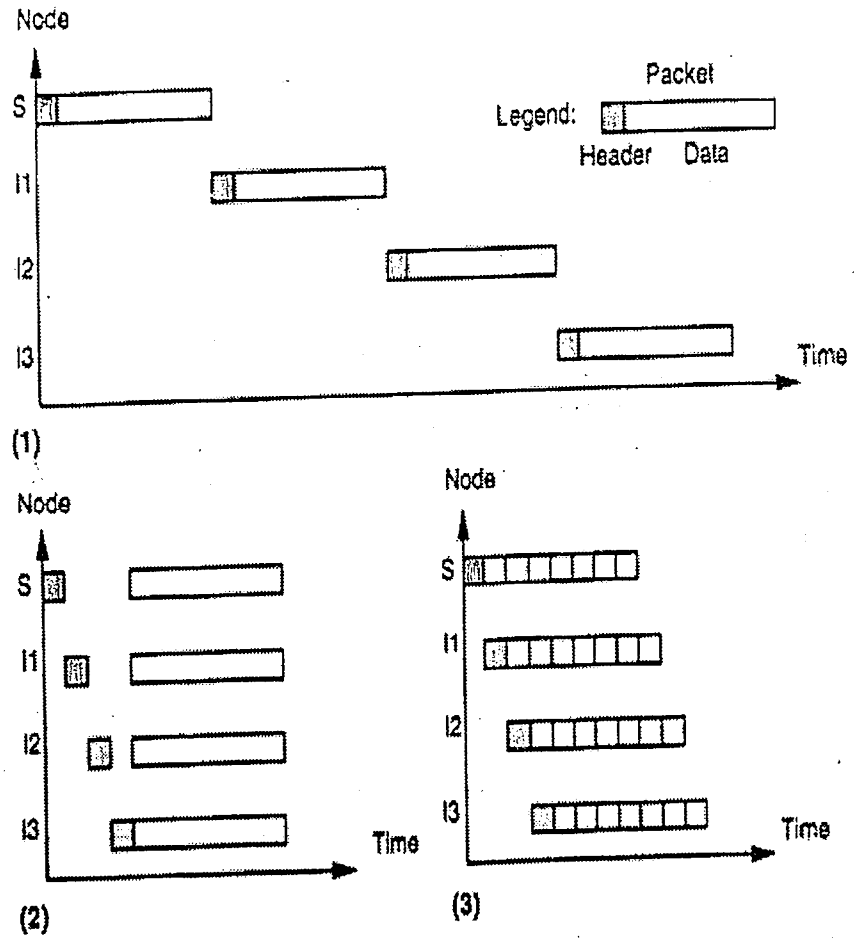


Figure 1.5 Comparison of Different Switching Techniques [1]

1) Store-and-forward
2) Circuit Switching
3) Wormhole routing

figure shows the activities of each node over time when a packet is transmitted from a source node S to the destination node D through three intermediate nodes, I1, I2 and I3. Unlike store and forward switching, both circuit switching and wormhole routing have communication latencies that are nearly independent of the distance between the source and destination nodes [11].

1.2.4 Topologies of Direct Networks

The topology of a direct network is usually modeled as a graph. It defines how nodes are interconnected by channels. If every node is connected to every other node in the network, it is called a fully-connected network. Such a network is very expensive and impractical to use because of limited amount of VLSI area available for communication related hardware. Many direct networks use a fixed, multiple-hop topology such as the hypercube network shown in Figure 1.4. Most popular direct networks fall in the general category of either n-dimensional meshes or k-ary n-cubes because their regular topologies simplify routing.

> " Formally, an n-dimensional mesh has $k_0 * k_1 * \ldots k_{n-2} * k_{n-1}$ nodes, k_i nodes along each dimension i, where $k_i >= 2$. Each node x is identified by n coordinates, $\sigma_{n-1}(x)$, $\sigma_{n-2}(x), \ldots, \sigma_1(x)$, $\sigma_0(x)$, where $0 <= k_i - 1$ for $0 <= i <= n-1$. Two nodes x and y are neighbors if and only if $\sigma_i(x) - \sigma_i(y)$ for all i, $0 <= i <= n-1$, except one, j, where $\sigma_j(y) = \sigma_j(x) +- 1$. Thus , nodes have from n to 2n neighbors, depending on their location in the mesh.
>
> In a k-ary n-cube, all nodes have the same number of neighbors. The definition of a k-ary n-cube differs from that of an n-dimensional mesh in that all of the k_i's are equal to k and two nodes x and y are neighbors if and only if $\sigma_i(x) = \sigma_i(y)$ for all i , $0 <= i <= n-1$, except one , j, where $\sigma_j(y) = (\sigma_j(x) +- 1) \bmod k$. The use of modular arithmetic in the definition results in wraparound channels in the k-ary

n-cube, which are not present in the n-dimensional mesh. A k-ary n-cube contains k^n nodes. If $k=2$, then every node has n neighbors, one in each dimension. If $k > 2$, then every node has 2n neighbors, two in each dimension." [1]

A general description of some commonly used topologies follow:

Binary Trees

In the basic binary tree, interior nodes have degree 3 with two children and one parent [6]. Leaves have degree 2 and the root has degree 1. The binary tree network has many variations, a tree with double roots being one. In the fat-tree network, the PEs are attached to the leaves of the tree while the trees internal nodes are SEs; furthermore, the closer an integral node is to the root, the more the number of edges is between it and its father. The Connection Machine CM-5 adopts this topology. The X-tree network is a binary tree with additional edges between neighboring nodes on the same level. Figure 1.6 (c) and (d) shows an example of a binary tree, and an X-tree topology.

Hypercubes

In the n-dimensional cube, each node is represented by a n bit number, and each edge connects two nodes that differ by exactly one bit position [6]. The hypercube has been used for a number of machines namely CM-2, Intel PSC-2 and iPSC/860. The number of processors in a hypercube is always an exact power of 2.

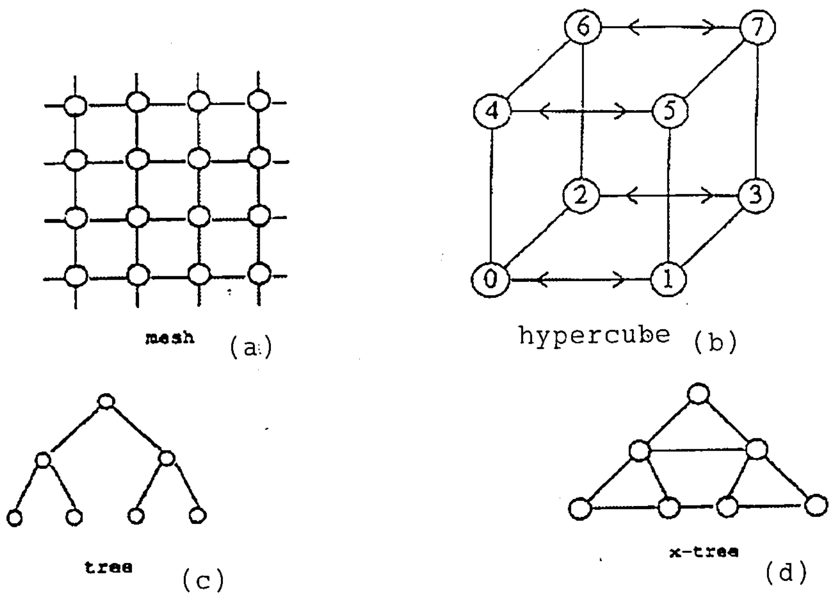


Figure 1.6 Sample Direct Networks

If the number of processors is 2^d , then d is called the dimension of the hypercube. Figure 1.6 (b) shows the structure of a hypercube of dimension 3.

Mesh

The simplest mesh has N^2 processors, with N rows and N columns. The connectivity of a mesh is 2 to 4 based on the location of the nodes. Other variations of meshes are a P * Q mesh, a 3-D mesh, or a wraparound mesh. An example of a mesh topology is shown in Figure 1.6 (a).

This chapter introduced parallel architecture and presented an insight of various interconnection topologies. The next chapter deals with Deadlocks in message passing systems and discusses methods to avoid deadlocking.

Chapter 2: Deadlock

Deadlock is usually discussed in the context of a multiprogramming environment, where a number of processes may compete for a finite number of resources, such as printers, processors etc. A process here is defined as an instant of a program in execution. As an example, consider a system with four tape drives. Say a process holds two tape drives and needs a third tape drives. It requests for the third drive and goes into a wait state. If another process is holding the other two drives, and it too needs a third drive, then each process will wait for the other to release its tape drives. This situation, where a number of processes wait for resources to get free is called a *deadlock* [22]. A way to break such a deadlock would be for the system to take some action, or preempt the resources held by the two processes, i.e. force them to release the tape drives. In a message-passing programming environment, deadlocking is also very common, and detecting and recovering from deadlocks is difficult. Thus the most favorable strategy to combat deadlocking is to have policies and mechanisms that avoid deadlock.

This chapter describes the various forms of deadlocks in blocking message passing. Techniques to combat or avoid deadlocking are also discussed.

2.1 Deadlock in Message Passing Systems

In multiprogramming environment, deadlocking occurs from events related to resource acquisition and release. However in message passing systems, the major cause of deadlocking is improper communication when blocking communication is used. Blocking is a term used to define synchronous communication where the sender and the receiver only communicate whenever they are both ready.

A set of processes is in a deadlock state when every process in the set is waiting for an event that can only be caused by another process in the set [4]. In concurrent programming languages, such as parallel C and Occam, deadlock occurs when a set of parallel processes is unable to progress any further. In such a condition, the system just hangs, and furthermore, debugging for the cause is extremely hard to do.

There are two major types of deadlocks that are encountered in a message passing environment where blocking communication is used between processes: Algorithmic deadlocks and cyclic deadlocks.

Algorithmic Deadlocks

These types of deadlocks arise when there is a mismatch in an algorithm for pairs of sends and receives (in Occam the '!' and '?' statements). That is for every send there must be a receive on the other end of that channel to accept it [4]. If there is no corresponding receive, a process waiting for its pair can not progress any further, thus blocking that process from continuing. In worst cases, the whole program can hang if the algorithm depends on that particular process to proceed.

Cyclic Deadlocks

These types of deadlocks occur if two processes are trying to send data to each other at the same time, thus forming a cycle. If each process is sending to another process, then each will wait indefinitely for the other to receive the message before progressing further which results in a cyclic deadlock [4]. This kind of deadlocks can occur between two or more concurrent processes. Any chain of sends that creates a cycle can cause a cyclic deadlock.

Cyclic deadlocks can prove really hard to debug, since the deadlock situation can depend on the ordering of events to happen. A slight change in the code may alter the way the execution of the processes interleaves and produce a situation that no longer deadlocks [5].

An example of a cyclic deadlock is shown in Figure 2.1, which involves four routers and four packets. Each packet is holding a flit buffer while requesting the flit buffer being held by another packet.

2.2 Avoiding Communication Deadlocks

There are numerous ways to avoid deadlocking when working in a parallel programming environment, some important techniques are discussed here:

2.2.1 Misuse of Channels

In parallel languages, it has to be made sure that all connections of channels are point to point. If the channel is not point to point, the program may hang and appear to be a deadlock, e.g.

```
ch:     Channel
z:      integer
SEQ {begin in sequence}
  receive z from ch
  send 2 on ch
```

In such an OCCAM construct, the program will hang because communications on channels are synchronous, the receive will wait for a send to occur. The send will never occur since the execution of the program is done in sequential order. The following rules should be practiced to avoid deadlocking

- If channels must be point to point, then if two processes share a channel, they must be the only processes that use that channel.
- If Channels are unidirectional, then once a direction has been established by a first use, all messages must pass in that same direction from then on.

2.2.2 Algorithmic Synchronization

If certain events of one process must be performed before the events of another process, then synchronization of processes is needed. In order for processes to synchronize, a synchronization message is sent to processes when needed. Thus a process that has to wait upon another process, will continue processing and will wait

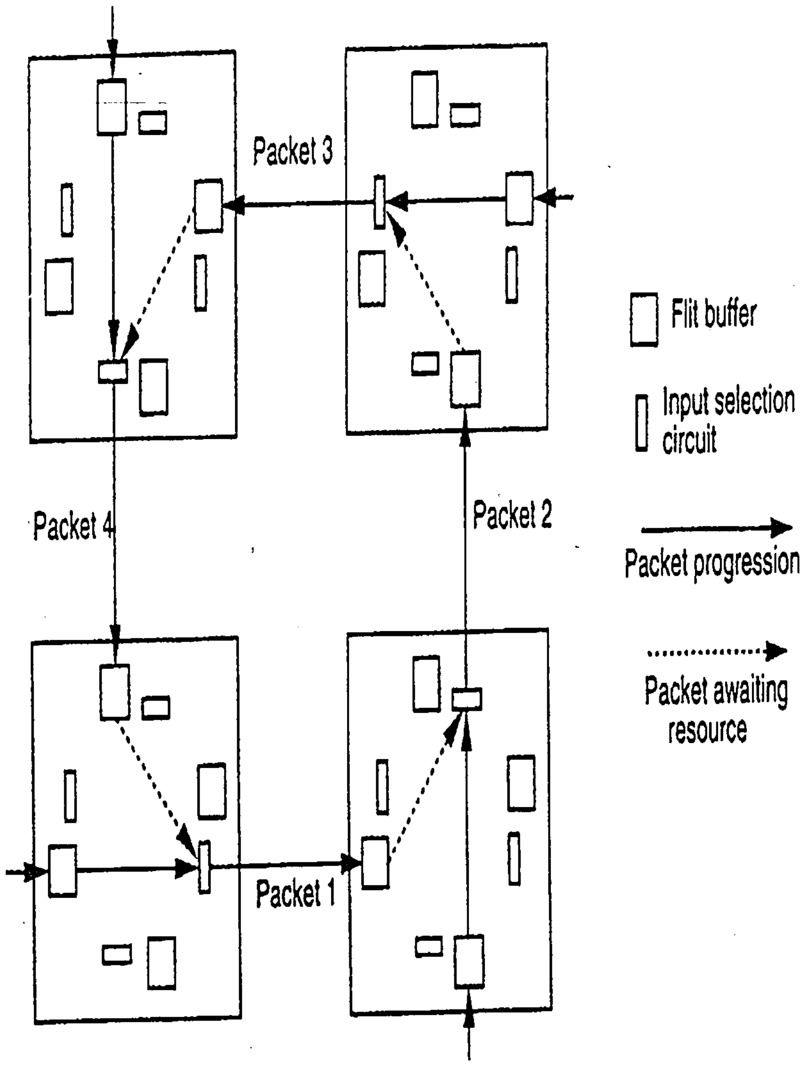


Figure 2.1 Example of a Cyclic Deadlock [1]

once it reaches the synchronization message for the other process. A perfect example is that of sorting in a hypercube, where each node sorts a certain number of items, and then items from neighboring nodes are merged to form a sorted list. If the synchronization is off during merging, then the list would end up to be wrongly sorted.

2.2.3 Shared Variables

When dealing with shared variables, where variables that are written by one process and read by another, one has to really be careful. Care must be taken to guarantee that the two processes are coordinated properly with a synchronization mechanism such as a semaphore [4].

2.3 Avoiding Cyclic Deadlocks

A cyclic deadlock, as discussed earlier, is one which may occur when two processes send data to each other at the same time. Any chain of sends or receives that creates a cycle can also cause a deadlock. Cyclic deadlocks occur due to the order of events which is time dependent, the buffers fill up and no sends or receives are possible. Cyclic deadlock avoidance requires careful planning to make sure cycles are not created. In order to prove that the program does not have a cycle of sends is to introduce a Hoare-style monitor to break the cycle [4]. A second way is to prove that the cycle never occurs by carefully introducing enough buffer space in a ring

[4]. A couple of these techniques are discussed below:

2.3.1 Buffering

If a cyclic deadlock occurs, buffering may reduce the possibility of deadlock. This technique lowers the chances of having a deadlock, but does not guarantee the complete elimination of a deadlock. One way to buffer is to use a concurrent process that waits for a message to arrive and stores it into a buffer. Inserting such a process between two processes to decouple the two processes communicating with each other allows more slack in communication.

2.3.2 Monitors

Monitors offer a solution in the sense that they guarantee that a cycle of processes cannot send at the same time; that there is a free buffer space. The monitor contains a data structure that keeps a record of which process is sending at a given moment. Before any process sends a message, it has to take permission from the monitor if it can do so. The monitor checks if the introduction of this message will result in a cycle. If it does, it does not allow the message to be sent. If a send is completed between other processes, the monitor updates its data structure, once it receives a message from the completed send [23]. Implementing a Monitor, though guarantees no deadlocks, involves a great cost in extra communication.

2.3.3 The Roscoe Approach

In a paper in 1988, Roscoe [13] developed a message passing protocol for a ring network that avoided deadlock.

In Roscoe's approach, at each node there is a two-slot buffer which can hold two messages, and there is a computation section . A node of such a structure is shown in Figure 2.2 If a message from the ring is for the computation process, it is removed from the first slot and forwarded to the computation section. The buffer can only accept a message from the computation section when it is empty. Also, only after a send of a message along the ring does the node receive from the ring. If this rule is observed, the following example shows that a deadlock is not possible : If there exists a cycle of deadlocked processes, P_0, P_1,..P_n. Each P_i must have received from P_{i-1} after its last send along the ring. Therefore, a message was passed from P_i to P_{i+1} after the last message from P_{i+1} to P_{i+2}. If this is continued around the entire ring, a contradiction is reached. Therefore, no cycles of deadlocked processes can exist if this protocol is followed [13].

Thus from this result, it is deduced that if the number of messages in the ring is smaller than the number of message slots, a ring will not cyclic deadlock. There is always a free slot, that a message can be moved to.

This chapter discussed deadlocks in detail and presented various techniques to avoid such situations. The next chapter introduces routing algorithms in direct networks.

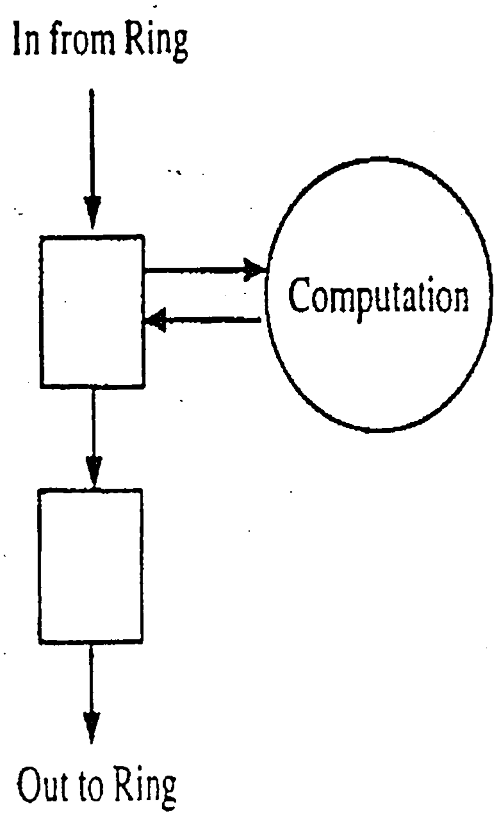


Figure 2.2 Example of a node in the Roscoe Approach [13]

Chapter 3. Routing Algorithms in Direct Networks

One of the most important issue in parallel computers is interprocessor communication. Since processors are linked together by a relatively sparse network, due to cost considerations, packets have to traverse many links and even be delayed by other packets due to conflicts or full buffers before reaching their destinations. Communication time therefore can easily exceed execution time, with the result that efficient routing becomes a primary concern. It is also desirable to have a constant buffer size for scalability purposes.

This chapter presents various routing techniques that are commonly used. Flow control strategies are also discussed and adaptive routing techniques are examined.

3.1 Classification of Routing

There are certain classifications of routing in a direct network topology that are discussed here. In direct networks, every node must be able to send packets to every other node. In the absence of a fully connected topology, routing determines the path selected by a packet to reach its destination. Efficient routing is critical to the performance of direct networks [1].

Routing is classified under the following methods:

Source Routing vs Distributed Routing

In source routing, the source node selects the entire path before sending the packet. Each packet that is being sent out must carry with it the routing information, thus increasing the packet size. Also once the packet is sent, the route cannot be altered. This type of routing has a few drawbacks. First the packet size is increased reducing scalability. Second if malfunctioned links are encountered, the packet will remain undelivered since it will not be re-routed.

Distributed routing is the most commonly used routing in direct networks. In this approach each router, upon receiving the packet, decides if the packet has to be sent to the local processor or to the router for further transmission. In the latter case, the routing algorithm determines which neighboring node is the packet to be forwarded to. In a practical router design, the routing decision process must be as fast as possible to reduce the network latency [1]. The decision process does not require global state information of the network. If this has to be incorporated, additional storage space in each router is needed, and also results in additional traffic.

Deterministic Routing vs Adaptive Routing

In deterministic routing, the path is totally determined by the source and the destination addresses. On the other hand in adaptive routing the path taken by the packet depends on the dynamic network conditions such as the presence of faulty or congested channels [8].

Minimal Routing vs Nonminimal Routing

A routing approach is said to be minimal if the route taken by the packet is the shortest. In this type of approach, every channel visited will bring the packet closer to the destination.

In the nonminimal routing approach, the packet is allowed to follow a longer path, depending on the network condition such as faulty links. Special care should be taken in this type of approach, that the packet does not end up being undelivered forever.

3.2 Flow Control Strategies

A network consists of many channels and buffers. Flow control deals with the allocation of channels and buffers to a packet as it travels along a path through the network without causing congestion or deadlock situation. If two processes collide or compete for a resource, and one of them cannot proceed, it is called a resource collision. Whether the packet is dropped, blocked in place, buffered, or rerouted through another channel depends on the flow control policy. A good flow control policy is one which avoids channel congestion while reduces the network latency [1].

The allocation of channels and their associated buffers to packets can be viewed from two perspectives. In the *output selection policy*, the routing algorithm determines which output channel should the packet be sent to that is arriving on an input channel. In the *input selection policy*, it has to be determined which packet may use the output channel since many incoming packets on the input channels may request a particular output channel.

In order to resolve conflicts, certain strategies have been developed, a few of the deterministic and adaptive routing algorithms are discussed below.

3.2.1 Packet Collision Resolution

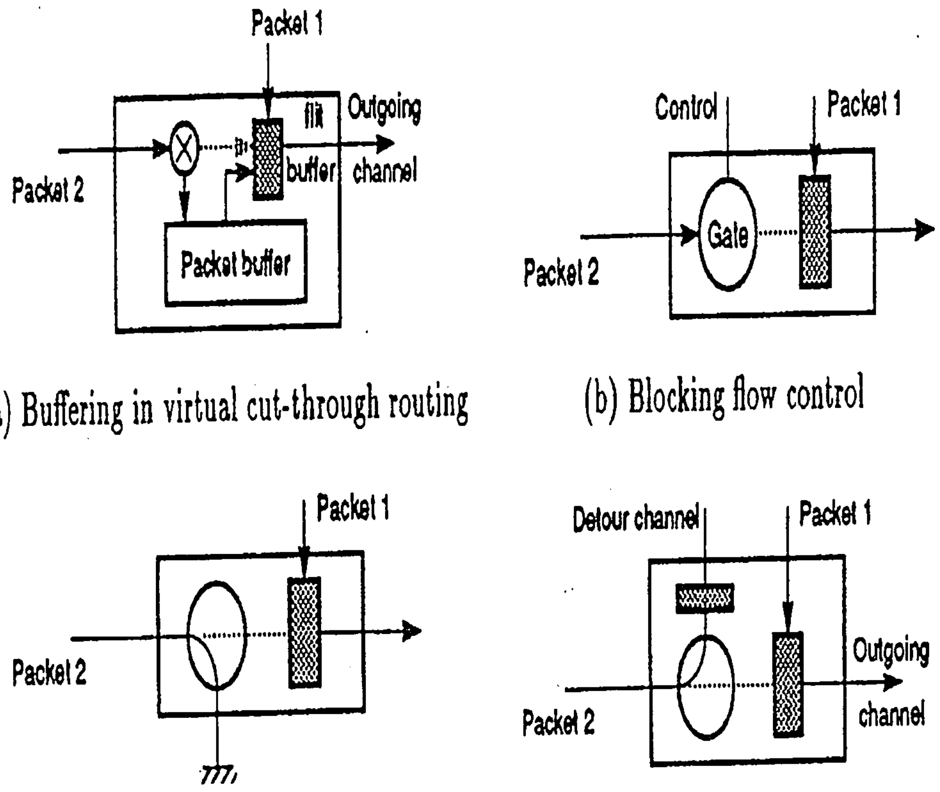
When packets are moved from one node to another, the following three elements must be present: a) The source buffer holding the packet, b) The channel being allocated, and c) the receiver buffer accepting the packet.

When two packets reach the same node, they may request the same receiver buffer or the same outgoing channel. Two major decisions have to be made at this point: i) Which packet will be allocated the channel?, and ii) what will be done with the packet that has been denied the channel?

There are four basic techniques for resolving the conflict between packets competing for the same outgoing channel. In the first approach as illustrated in Figure 3.1 (a), Packet 2 is temporarily stored in a packet buffer. When the channel becomes available, it will then be transmitted. This buffering approach has the advantage of not wasting the resources already allocated. However, it requires the use of a large buffer to hold the entire packet [5]. This virtual cut-through method combines both wormhole and store-and-forward schemes.

The second technique, illustrated in Figure 3.1 (b) uses a blocking policy in case of packet collision. Packet 2 is blocked from advancing and once packet 1 is transmitted, Packet 2 is let through. This approach is one used by wormhole routing.

The third technique illustrated in Figure 3.1 (c) uses the discard policy that



(a) Buffering in virtual cut-through routing

(b) Blocking flow control

(c) Discard and retransmission

(d) Detour after being blocked

Figure 3.1 Flow control methods for resolving collisions [5]

just drops the packet being blocked from passing through. The discard policy may result in severe waste of resources and it demands packet retransmission and acknowledgement. Otherwise, a packet may be lost after discarding it [5]. This policy is rarely used due to its unstable nature.

The fourth policy illustrated in Figure 3.1 (d) is called detour. In this policy, the blocked packet is misrouted to a detour channel. This technique offers more flexibility in packet routing. However, it may waste more channel resources than necessary to reach the destination.

These are some of the techniques that are used by various interconnection networks. In practice, most networks use hybrid policies which may combine the advantages of some of the above flow control policies [5].

3.2.2 Dimension Order Routing

Dimension-ordering routing requires the selection of successive channels to follow a specific order based on the dimensions of a multidimensional network. Each packet is routed in one dimension at a time, arriving at the proper coordinate in each dimension before proceeding to the next dimension. By enforcing a strictly monotonic order on the dimensions traversed, deadlock free routing is guaranteed. Both hypercubes and 2-D meshes each use a deadlock free minimal deterministic routing algorithm. In the case of the two-dimensional mesh network, the scheme is called X-Y routing because a routing along the X-dimension is decided first before choosing a path along the Y-dimension [5]. This approach is discussed in detail in the next chapter on Mesh.

3.3 Adaptive Routing Techniques

The previous approaches that were discussed were deterministic techniques where the communication path is completely determined by the source and destination nodes. The main disadvantage of deterministic routing is that it cannot react to dynamic network conditions such as congestion [1]. An adaptive routing technique must also address the deadlock issue. To do so requires the use of additional channels, in particular, some adjacent nodes must be connected by multiple pairs of opposite unidirectional channels. These pairs of channels may share one or more physical channels. An adaptive routing technique on the 2-D mesh is discussed in the next chapter.

Chapter 4. Mesh Topology

The mesh topology consists of processors arranged in a two-dimensional array. A given processor at row i and column j is connected to its four immediate neighboring processors to the left, right, above and below: at locations (i-1,j), (i+1,j), (i,j-1), and (i,j+1). All connections are horizontal between adjacent columns, or vertical between adjacent rows see (Figure 4.1). There are no diagonal connections. Boundary processors have only two or three immediate neighbors. The 2-D meshes in some form are used in the ILLIAC IV, MPP, DAP, and WRM [6]. As an example in the ILLIAC, all nodes have degree 4, bottom nodes are connected to the top node, in the same column, and rightmost nodes are connected to the leftmost nodes in the next row.

4.1 Characteristics of a Mesh

A mesh multicomputer topology consists of processors arranged in a two-dimensional array. The mesh has the following characteristics.

* All connections are horizontal between adjacent columns, or vertical between adjacent rows.
* There are no diagonal connections between processors.
* The processors are numbered sequentially by rows, with processor 0 in

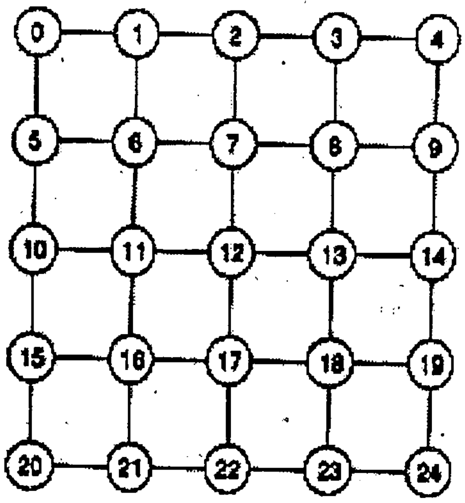


Figure 4.1 A Simple two dimensional square mesh

the upper left corner.

* Every pair of processors have a minimum path length between them measured by the sum of the row distance and the column distance.
* For a mesh of N processors, where N=n*n and n is the number of processors on one side of the mesh, the diameter of the network is the path length between processors at opposite corners of the mesh, which is always 2*(n-1).
* The Mesh topology can be improved by adding end-around connection.
* The minimum latency of a mesh is $O(n)=O(\sqrt{N})$.
* The Connectivity of the Mesh is 4-8 nearest neighbors.
* The wire cost for a Mesh Topology is $O(N)=O(n^2)$

Figure 4.2 shows Latency, bandwidth, connectivity, wire and switching costs of some commonly used direct networks as compared to the mesh network.

4.3 X-Y Routing on the 2-D Mesh

In a deterministic X-Y routing, an approach called Dimension ordered routing is used [1]. In this method, each packet is routed in one dimension at a time, arriving at the proper coordinate in each dimension before proceeding to the next dimension. Deadlock free routing is guaranteed by enforcing a strictly monotonic order on the dimension traversed [1].

In a 2-D mesh, each node is represented by its position (x,y) in the mesh. In

Network	Minimum Latency	Maximum Bandwidth per PE	connecti-vity	wire cost	users
hypercube	log N	const.	log N nearest neighbors	N log N	Cosmic Cube, CHOPP, Connection Machines
cube con-nected cycles	log N	const.	3 nearest neighbors	N	
tree	log N	const.	3 nearest neighbors	N	Non Vonn, DADO, Cellular
mesh	N	const.	4-8 near-est neig-hbors	N	ILLIAC IV, DAP, MPP, WRM,

Figure 4.2 Tradeoffs among commonly used direct networks

the X-Y routing method, packets are first sent in the X dimension and then in the Y dimension, depending on the destination of the message. This approach allows only one turn in the routing and that turn is from the X dimension to the Y dimension.

As an example, let (s_x,s_y) and (d_x,d_y) denote the address of a source and destination node, respectively, and let $(g_x,g_y) = (d_x - s_x, d_y - s_y)$. XY routing can be implemented by placing g_x and g_y in the first two flits of the packet, respectively. When the first flit of a packet arrives at a router, it is decremented or incremented depending on whether it is greater than 0 or lesser than 0. If the result is not equal to 0, the packet is forwarded in the same dimension and direction it arrived in. If the result is 0 and the packet arrived on the Y dimension, the packet is delivered to its local processor. If the result is 0 and the packet arrived on the X dimension, the flit is discarded and the next flit is examined . If that flit is zero too, then the packet is delivered to the local processor, otherwise it is forwarded in the Y dimension [1]. An example of X-Y routing is shown in Figure 4.3

This routing method is deadlock free and ensures that the path taken is the minimal route and the shortest distance between any source and destination processors.

4.4 Adaptive Routing on the 2-D Mesh

The problem with a deterministic routing such as the one discussed above is that it cannot conform to dynamic network conditions. For example, in a deterministic

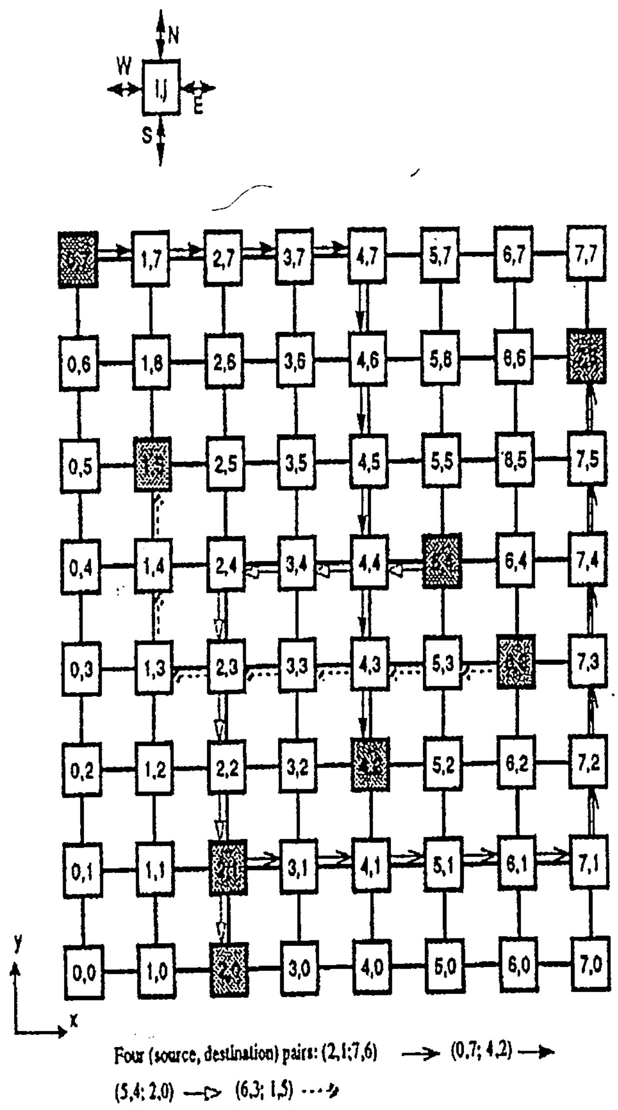


Figure 4.3 Example of X-Y routing on a mesh [5]

approach, if a packet encounters a bad link or a busy channel all it will do is wait there for the channel to get fixed or get free whatever the situation may be. If the problem is major, the packet will wait there indefinitly and will not be rerouted.

In the adaptive routing algorithm, the messages are re-routed based on the network conditions. To ensure deadlock free adaptive routing, the use of additional channels is required however in particular, some adjacent nodes must be connected by multiple pairs of opposite unidirectional channels [1].

One approach for a minimal adaptive routing is to partition the channels into disjoint subsets. Each subset constitutes a corresponding subnetwork and packets are routed through different subnetworks depending on the location of destination nodes.

Figure 4.4 shows the application of this approach on a 2-D mesh. Additional pair of channels are added to the Y-dimension, with the result that the network can now be partitioned into two subnetworks labelled +X and -X, each having a pair of channels in the Y dimension., and a unidirectional channel in the X dimension. If the destination node is to the right of the source, i.e $d_x > s_x$, the packet will be routed through the +X subnetwork, and if $d_x < s_x$, the -X subnetwork will be used. If $d_x = s_x$, then either of the subnetworks can be used.

This double Y-Channel routing algorithm is minimal and fully adaptive; that is, a packet can be delivered through any of the shortest paths. The algorithm can be proved to be deadlock-free by ordering the channels appropriately. An example of such an ordering of the channels in the +X subnetwork is shown in Figure 4.4 (b).

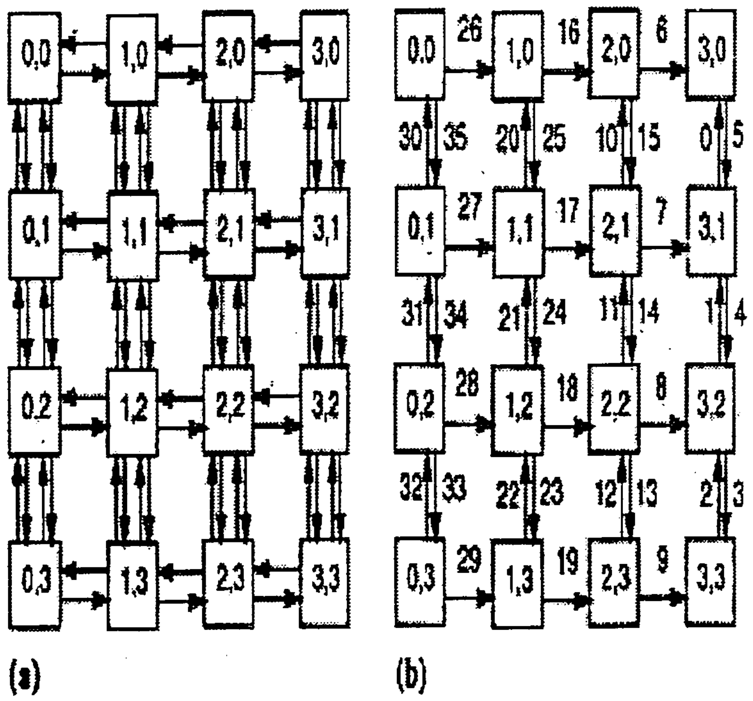


Figure 4.4 Adaptive Y-Channel routing for a 2D mesh [1]
(a) Double Y Channel 2D mesh
(b) +X subnetwork and labelling

For any pair of source and destination nodes, the channels will be traversed in descending order, no matter which shortest paths are taken. Hence a deadlock cannot occur [1].

In Figure 4.4 (b) for example, any of the minimal paths (25,24,18), (25,17,14), and (16,15,14) can be taken, to go from node (1,0) to node (2,2). All the paths mentioned valid and are all minimal. Thus this approach is adaptive, but there is a cost added to it, in the sense that additional channels are to be added to the network, and the number of additional channels increase rapidly with N, which is the number of processors, and thus makes it impractical with large N's.

4.5 Network of Transputers

A transputer is a microcomputer with local memory and communication links that can connect it to four other transputers. The SuperSetPlus64 [7] contains 64 T805-20 transputers connected by C004 software programmable link switches. The T805-20 is a 32-bit microcomputer running at 20MHz. It has a 64-bit floating point unit, a 4 bi-directional 10Mb/sec communication links, and 4K of on-board RAM. In addition, each transputer has access to either 4MB or 1MB of external memory. The SuperSetPlus64 does not have a defined or fixed topology, and thus a number of topologies such as a Mesh, Hypercube, a Tree or a user-defined topology can be implemented on it.

The SuperSetPlus64 transputer array is partitioned into groups of four and each group is called a cluster. Nodes are allocated on a cluster-by-cluster basis. So processors are allocated in units of four at a time. Figure 4.5 shows a cluster of four transputers of a SuperSetPlus64.

4.5.1 Setting up a Mesh on the Transputer

A number of different topologies can be set up on the SuperSetPlus64. By default, a four-processor network is provided with each processor connected to every other processor, and the host processor is connected to the PC. This standard topology can be modified by using a MAP file.

A sample MAP file follows:

```
; Sample .MAP file to create a Two by Two mesh
;
; Processor #          Connects To  Position
        1      :       2,3          ;Upper Left Corner
        2      :       1,4          ;Upper Right Corner
        3      :       1,4          ;Lower Left Corner
        4      :       3,2          ;Lower Right Corner
```

```
;The overall processor arrangement is:

;        1--------2
;        |        |
;        3--------4
```

Once the MAP file has been created, a network information file (NIF) is created by using the node_map program. This generates the NIF file. This file contains the information of how each processor is connected to another. An example of setting up a two by two mesh follows:

#	[Link0	Link1	Link2	Link3	]
0	[HOST	1:2	2:1	...	]
1	[...	3:2	0:1	...	]
2	[...	0:2	...	3:1	]
3	[...	2:3	1:1	...	]

The first line in this file (processor # 0) shows that link #0 connects it to the HOST (PC), and link #1 connects it to 1:2, i.e processor, link 2. In other words it says that processor # 0 is connected to link # 2 of processor #1 , by link #1, and so on.

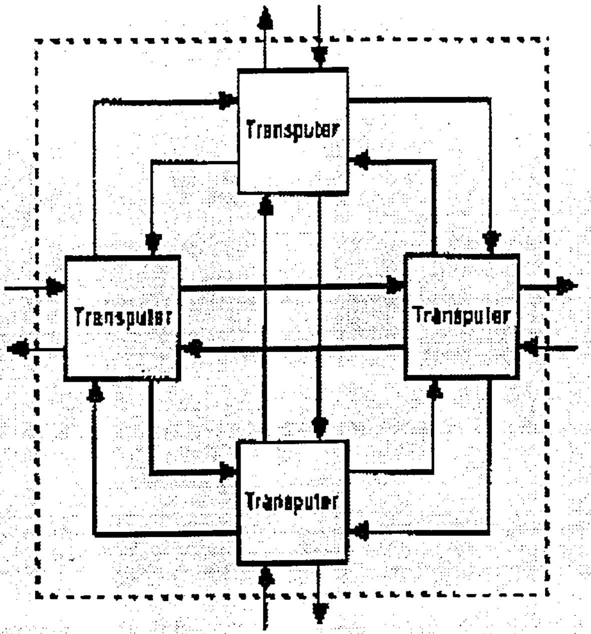


Figure 4.5 A cluster of four transputers of a SuperSet64 [7]

This chapter detailed the characteristics of a mesh, and the various routing techniques that are commonly used. Adaptive routing, though may look more general than the deterministic one on the mesh, at its worst it can result in more problems than are bargained for such as the message can be lost forever in attempting to find a clear path. The deterministic method is more easier to implement and it is the most commonly used among the two.

The next chapter details the design and implementation of a deadlock-free routing algorithm on a 2-D mesh using the Superset Plus 64 machine. Various techniques that were discussed earlier in this chapter and that will be discussed in the following chapter have been incorporated to achieve the goal.

Chapter 5. Design And Implementation Of Deadlock-free Routing

The previous chapters discussed various issues regarding routing, deadlocks, and meshes in general. This chapter presents the actual implementation of a deadlock-free routing algorithm on a network of transputers, using some of the techniques that were discussed in the preceeding chapters. The structure of each process and the routing technique that is implemented for a deadlock-free routing algorithm are outlined.

5.1 Previous Work

The problem of developing a deadlock-free routing support has dragged on for years without any really satisfactory solution being found. Only recently has the routing systems for networks based on transputers been discussed in literature, and there are currently quite a few techniques for setting up deadlock-free routing protocols.

Ugo De Carlini and Umberto Villano [14] discussed various approaches to eliminate deadlocks in transputer systems. The one they emphasized on was the use of a simple graphical representation of the data exchanged in a program, the I/O graph, which makes it possible to detect potential program deadlock. They illustrated that Communcation deadlocks can be detected by means of the communication state graph, which is a direct graph that dynamically models the ungranted I/O requests issued by a set of communicating processes. In this graph the processes waiting for an I/O exchange and their partners are represented by vertices, and the ungranted I/O request by arcs. The arcs are directed by convention from the process issuing a request to the one requesting the I/O exchange. A deadlock is detected, if there is a *cycle* in the communication state diagram [14].

Another approach that they discussed was the use of buffers. An example of the buffering technique is given by the following code that outlines the outputting process

```
-- outputting process

PRI ALT
 out.chan ! message
  SKIP
 NOT(buffer.full) & SKIP
 ...store message into buffer
...
```

Whenever a message is available to be transmitted, the above process tries to output it to the out.chan, but if the channel is busy, the message is temporarily stored in a buffer. In this example, the buffers reduces the likelihood of there being a deadlock but it does not completely solve the problem, because the buffer might be full, and thus it can result in a deadlock.

When discussing buffers, there is also a method called the 'structured buffer pool' [15-17]. In this technique, message buffers are partitioned into d(max) classes, where d(max) is the maximum path length in the network, and a packet that has already covered i hops is allowed to use only the buffers of the classes $<= i$. Under normal traffic conditions, the messages are stored only in class 0 buffers, but when the load of the messages increases, buffers from level 0 to level d(max) are progressively used. This means that if the buffers of the classes $<= k$ are full, only messages coming from a node distant at least k hops from their sources are accepted. If this strategy is adopted, an ungranted request for a buffer can be directed only from a buffer of class i to a buffer of class $(i + 1)$. Hence a possible chain of ungranted requests cannot form a circuit, because it passes through buffers of progressively higher classes, and no deadlock can occur.

N.T Son and Y. Parker [18] used an adaptive deadlock-free packet routing approach in transputer-based multiprocessor interconnection networks. Their research presented an adaptive routing algorithm which controlled the transmission of messages through the network. As the messages build up in the network, the algorithm tries to reduce the blocking of traffic by using adaptively the existing idle buffers in the network. The method is called adaptive deadlock-free routing (ADR). This ADR algorithm is based on a combination of deterministic and adaptive routing. The algorithm is deterministic for light traffic, and becomes adaptive for heavy traffic conditions.

A simplified algorithm of the ADR approach is given below, where d.a represents the destination address of a packet, the link l is obtained from the routing function R{n} where n is the destination, and q(i) stands for the state of the queue i, i.e full or empty.

```
WHILE running
  SEQ
    ...Input a packet
    ...Find link $l$:=R{d.a}
        IF
          q($l$) NOT full
            ...Send the packet to queue $l$
          TRUE
            SEQ
                ...Find the q($i$) that is NOT full
                ... Send the packet to queue $i$
```

The messages thus try to first find the shortest path, i.e deterministic. The adaptive strategy is used only when a shortest path starts getting overloaded or a destination address is temporarily unable to consume incoming messages.

G.D. Piffare, L. Gravano, S.A. Felperin and J.L.C. Sanz [19], developed a fully adaptive minimal deadlock-free packet routing algorithm on hypercubes, Meshes and other networks. For a Mesh, a partially adaptive algorithm is developed that is based on the idea of "Hanging" the mesh from the (0,0) and (n-1,n-1) nodes and consists of two phases. In phase A the messages move toward their destination by visiting nodes in such a way that if a message passes from (x,y) to (x', y') in one routing step, then $x < x'$ or $y < y'$. In phase B, the messages visit nodes with lower numbers. In other words, in phase A the mesh is hung from node (0,0) and the messages visit nodes with higher levels where the level of (x,y) is x+y. In phase B , the mesh is hung from node (n-1,n-1) and the nodes are visited in decreasing level order. Once all the steps that could be taken in step A have been completed, the message enters phase B. This scheme can be implemented with two queues,, q[a] for phase A, and q[b] for phase B messages. The routing scheme is deadlock free, because the queue dependency grapgh is acyclic [19]. This scheme can also be extended into a fully adaptive one, that is still deadlock-free and uses the same number of queues. This is done by allowing messages that have not completed their phase A to take phase-B steps (but still visiting q[a] queues. In phase B, the messages still have to go through ascending paths. The resulting algorithm is such that every message always has a chance of taking a static transition, as messages keep visiting q[a] queues while taking dynamic transitions [19].

These are some of the techniques that are available and that have been designed and developed. The algorithm designed in this research uses some of the existing techniques and at the same time introduce some modifications to develop a deadlock-free routing algorithm for a square mesh.

5.2 Choice of Language - "OCCAM"

Transputers can be programmed in most high level languages and are designed to ensure that compiled programs will be efficient. Where it is required to exploit concurrency, but still to use standard languages, OCCAM is the most reasanable choice.

To gain most benefit from the transputer architecture, the whole system can be programmed in OCCAM. This not only provides the advantages of a high level language but also provides maximum program efficiency and the ability to use the special features of the transputer.

Occam is the first language to be based upon the concept of parallel, in addition to sequential, execution and to provide automatic communication and synchronization between concurrent processes.

The transputer and OCCAM were designed together and all transputers include special instructions and hardware to provide maximum performance and optimal implementations of the OCCAM model of concurrency and communications.

In OCCAM, processes are connected to form concurrent systems. Each process can communicate with other processes using point to point communication channels.

Basic Overview

In its most general form, OCCAM programs are built from three primitive processes [20]: *Assignment, Input and Output.*

An assignment computes the value of an expression and sets a variable to the value. An example would be

```
c := v                    -- assign the value of v to c
```

Input and Output primitives are used for communication between processes. Two processes communicate with each other by means of a one-way channel. One process outputs or sends the data on the channel, while the other process inputs or receives the data from the channel. Communication in OCCAM takes place when both the sending and receiving processes are ready. Thus in order for communication to take place, the processes have to be synchronized [21].

An example of communication between two processes follows:

```
PAR
  c ! e          -- output variable e to channel c
  c ? v          -- input from channel c and assign the value to v
```

The other major component of OCCAM are constructs. A construct itself is a process and is composed of a combination of several processes. Each component process of a construct is written two spaces further from the left hand margin to indicate that it is a part of a construct. The following three constructs are the most important in OCCAM

SEQuential	--	components executed one after the other.
PARallel	--	components executed together
ALTernative	--	component first ready is executed.

An example of several constructs running in parallel follows:

```
PAR
  SEQ        -- process1
    p1
    p2
  SEQ        -- process 2
    p3
```

```
p4
PAR
  p5      -- process3
  p6      -- process4
```

In this example four processes are running in parallel. Firstly process 1 and 2 are running in parallel, while process 3 and 4 are running in parallel as part of process 2.

OCCAM can be used to program an individual transputer as well as a network of transputers. On a single transputer, channels are used to communicate data between processes. On a network of transputers, communication is implemented directly by transputer links.

Thus considering the versatile nature of OCCAM and its close link to the transputer, OCCAM was the obvious choice to implement the routing algorithm on the transputer.

5.3 Software Overview

The basic goal of the thesis is the design and implementation of a modular deadlock_free routing algorithm for a two-dimensional mesh on a network of any size. In this thesis, for discussion and explanation purpose only, a two by two mesh will be used as an example. This will help understand the structure of the approach and give an insight of how communication is performed.

In chapters 3 and 4 routing on a mesh topology were discussed, and it was found that the most suitable approach for routing on the mesh is the X-Y routing. Both deterministic and adaptive techniques can be used, but since adaptive routing can cause loss of messages in the worst case, and also since it is more tedious to implement, minimal deterministic routing was chosen . What this means is that the path taken by the message as it travels from the source to the destination will always be the minimal path, i.e requiring the least number of hops. Also the store and forward technique is used for switching.

Since for discussion purposes, only four nodes/processors will be used, only one cluster from the Superset Plus 64 is required.

5.3.1 Process Structure

Each node has been designed to have a number of processes running on it. Basically six major processes are running, all in parallel. Firstly there is the input process that waits in parallel to receive on any of the channels that a node might have. Since there are more than one input channels, the ALT component is used. The moment any channel recieves an input, all other channels are blocked and the message is read and stored in a buffer. Figure 5.1 outlins all the six processes and how they interact with each other. A pseudo code of the input process follows:

```
PAR
  WHILE TRUE
    ALT
      channel1x ? source; destination; direction; message; status
                  -- read values from channel1x
        buffer1 ! source; destination; direction; message; status
                  -- write the values to a buffer
    channel2x ? source; destination; direction; message; status
              -- read values from channel2x
      buffer2 ! source; destination; direction; message; status
              -- write the values to a buffer
    channel3x ? source; destination; direction; message; status
              -- read values from channel3x
      buffer3 ! source; destination; direction; message; status
              -- write the values to a buffer
```

The second process running is the one that reads from the buffer and calls the routing procedure. Since the routing process has already been discussed, an example of the buffer process follows:

```
WHILE TRUE
  ALT
    buffer1 ? source; destination; message; status
              -- read from buffer1
      SEQ
        route(bufferout, source, destination, direction, message, status)
              -- call the router
        IF
          direction = 1        -- if direction is 1, i.e source is reached call route again
                                  to determine the destination
            route(bufferout, source, destination, direction, message, status)
          TRUE
            SKIP
    buffer2 ? source; destination; message; status
      SEQ
        route(bufferout, source, destination, direction, message, status)
    buffer3 ? source; destination; message; status
      SEQ
        route(bufferout, source, destination, direction, message, status)
```

Next come the output processes, the first one gets the data from the router and puts it in the output buffer, while the other reads from the output buffer, and sends the message to the respective output channel.

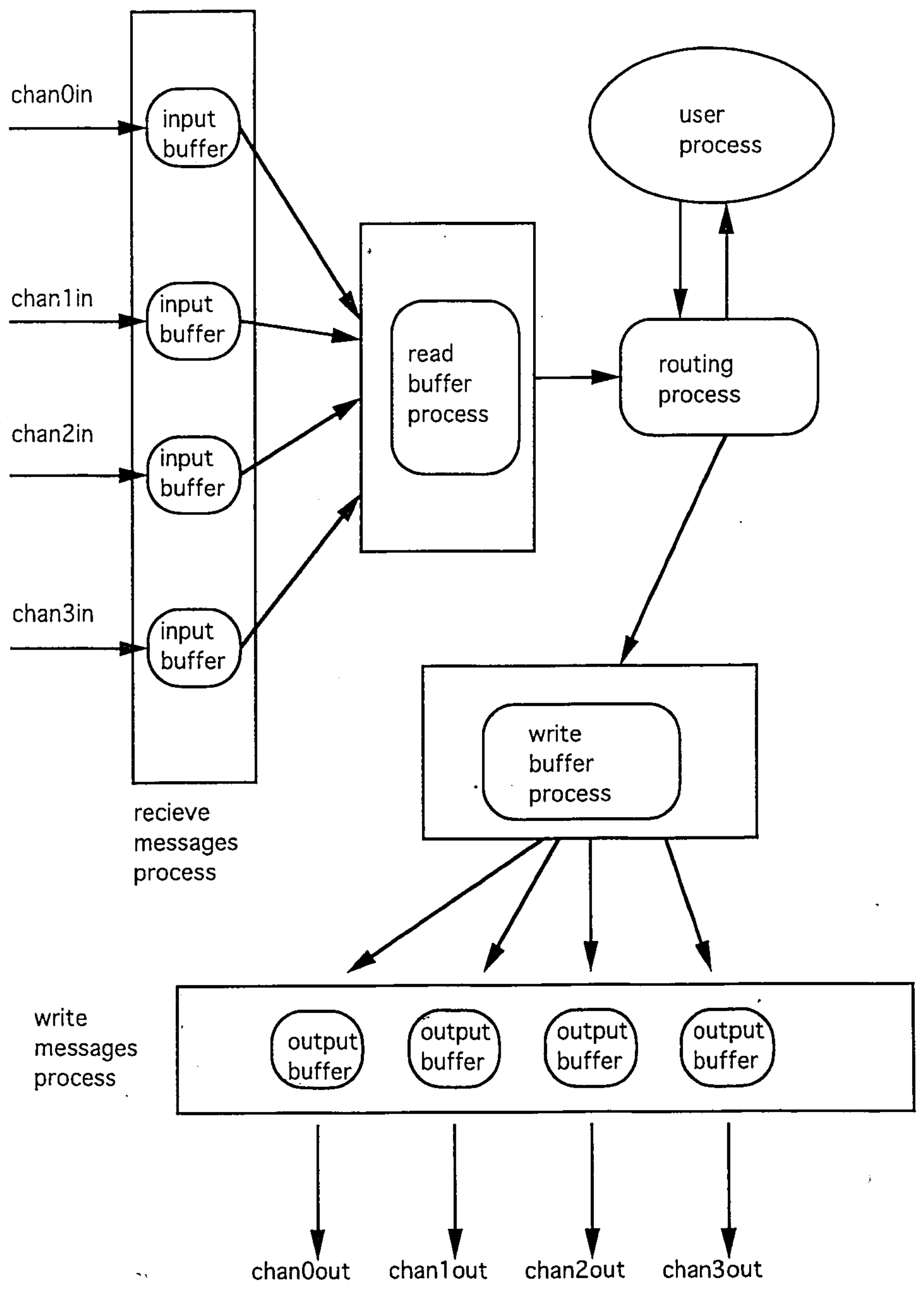


Figure 5.1 Structure of the processes and their interaction

```
WHILE TRUE
  ALT
    outbuffer1 ? source; destination; direction; message; status
      SEQ
        channel1y ! ource; destination; direction; message; status
    outbuffer2 ? source; destination; direction; message; status
      SEQ
        channel2y ! ource; destination; direction; message; status
```

The last process is the local process, that can recieve message that is sent to it, do computation on it, and also send it to other processors.

5.3.2 Message Routing

In its most general form, processors are labelled from 0 to 4. Since processor 0 will be the root processor, it is attached to a HOST which is used for I/O. The HOST requests for user input and asks for the source and the destination node. The data type of source and destination is kept as a structure, thus each x and y co-ordinates are read for each processor.

The HOST also requests for the message that must be sent by the processor. Once the message is read, the HOST sets the direction of the message to 0, sets the status to 0, and writes to the channel (channel0x in this example) that connects it to Processor 0, the source address, the destination address, the message, direction and status.

At the same time, in Processor 0, there is a process that is waiting in parallel for inputs from all its channels. Since in this example, channel0x is the channel that connects Processor 0 to the HOST for inputing message, the moment the HOST writes something to this channel, it is read by Processor 0 and stored in a buffer. At the same time, another process is running in Processor 0 that reads from the buffer and sends it to the routing process.

The routing process is an independent process that gets the value from the buffer and attempts to route the message to its appropriate destination. The direction of the message is kept at 0 until the source of the message is determined. Once the source is reached, the direction variable is set to 1 and passed on to the destination. Once the destination is reached, it is set to 2. This is done in case the message has to be re-routed to the HOST, and a direction of 2 would mean that this message has reached its destination and is on its way back to the HOST for monitoring purposes.

The routing process uses the deterministic X-Y method. Each Processor knows its x and y coordinates. Once a message is recieved, the routing process compares the x value of its own with the incomming message. If it is the same, it compares the y value of the message with itself. If that is the same too, it knows that it is the source and direction is set to 1. If the x coordinates are not the same, then it checks to see if it is greater or less than its own x value. Depending on the result, it writes to the respective output channel buffer where it is read by another process and written out to the output channel. The same is true for the y- coordinates. Thus each message is routed in one direction at a time until it arrives at the proper coordinate in each dimension, before proceeding to the next dimension. Instead of incrementing or decrementing the difference of the source and destination coordinates, each processor just compares the source and destination address with its own coordinates depending on which direction the message is comming from and directs the message depending on the results of the comparison.

A pseudo code of the routing algorithm follows:

```
PROC route
 SEQ
  IF
   direction = 0
    IF
     source[0] > x        -- if the x coordinate of source is greater than x
      SEQ
       buffout1 ! source, destination, direction, message, status
                          -- write the message to the output buffer
     source[0] < x        -- if the x coordinate of source is less than x
      SEQ
       buffout2 ! source, destination, direction, message, status
                          -- write the message to the output buffer
     source[0] = x        -- if x coordinate of source is equal to x
    IF
     source[1] > y        -- the y coordinate of source is greater than y
      SEQ
       buffout3 ! source, destination, direction, message, status
                          -- write the message to the output buffer.
     source[1] < y        -- the y coordinate of source is less than y
      SEQ
       buffout4 ! source, destination, direction, message, status
                          -- write the message to the output buffer.
     source[1] = y        -- if y coordinate of source is also equal to y
      SEQ
       dir := 1  -- set direction to 1: the source has been reached
        TRUE
         SKIP
     TRUE
      SKIP
```

This gives an example of how the router works. A small variation is used for messages comming from other directions, but the concept is the same. This type of routing is minimal and always uses the shortest path to reach the destination.

5.4 Deadlock Avoidance Techniques

Since deadlock-free routing was one of the goals of the thesis, several features have been used in the algorithm to avoid deadlocking. A set of processes is in a deadlock state when every process in the set is waiting for an event that can only be caused by another process in the set. Since deadlocking is very much inherent to parallel computation, the proper choice of routing algorithm is required.

Considering this problem, the X-Y routing technique , which uses dimension order routing, enforces a strictly monotonic order on the dimensions traversed, and thus guarantees deadlock-free routing. [1]

The other technique that is used in this algorithm is buffering. Buffers do not guarantee deadlock-free situation, but very much reduces the chance of having a deadlock to occur. Keeping one buffer for each input and output channel, reduces the chance of a deadlock to occur. A concurrent process is used in this program as seen in the previous section, that buffers the input and output messages. Though it might create a slack in communication, but it helps in avoiding deadlocks.

If buffering was not used a deadlock would easily occur. As an example if there are four nodes arranged in a two-by-two mesh and node 1 is sending to node 2, node 2 is sending to node 3, and node 3 is sending to node 1 all at the same time, then a send at one time. Thus a deadlock will occur with only four messages in the network. However if a buffer is placed in each node, then the writing node can write to the buffer of the recieving nodes and thus will free a link.

Other causes of deadlocks such as "Misuse of Channels" are also considered, and the program is reviewed by introducing test data to check if a miscommunication is occuring. Also since all channels are uni-directional, care has been taken that the channels are being used in one direction only.

5.5 Flow Control Strategy

Flow control deals with the allocation of channels and buffers to a packet as it travels along a path through the network. If a problem is encountered in the normal path of a packet, a decision is to be made what to do with the packet such as rerouting, blocked or dropped altogether. In this program, the first come first serve strategy has been employed using the ALT construct. Along with the Flow control Strategy, the Store and Forward switching technique is used. So if a message is to arrive at a node and cannot continue because of a channel is in use, the whole message is stored in the buffer and forwarded once the channel is available again.

5.6 Modularity of the Algorithm

The program has been developed in such a manner that it can account for a square mesh of any size. In the configuration file, only the network part where the nodes are connected to each other with links, that a user has to hardcode the information. Other than that, the program computes addresses of each processor, based on the size of the mesh.

The processors are numbered from left to right in the order of 0,1,2,3 ..Width-1. The next issue is to determine what is the position of each node on the mesh. This is done by simple DIV's and MOD's functions with the width of the mesh, and the programs goes through an 'IF' construct to determine the position of each node.

As an example, the following code will determine where each node is located on the mesh and accordingly how many channels it is connected to.

```
PAR I=1 FOR width*width
  PROCESSOR p[I]
    IF top edge
      IF left edge
        PAR
          call procedure for Corner node (four channels plus host)
          call procedure for host
      IF right edge
        call procedure for Corner node (four channels)
      ELSE
        call procedure for intermediate edge node (six channels)
    IF bottom edge
      IF left edge

        call procedure for corner node
      ....
      ....
    ELSE
      call procedure for middle node (eight channels)
:
```

In this way, three major procedures are called each having four, six or eight channel declarations. The process number 0 is a special case, since it is a corner node but at the same time it is connected to the host with two extra channels.

This chapter presented the method utilized to implement a deadlock-free algorithm on a transputer machine. The process structure, the choice of the routing algorithm and various approaches to avoid deadlock were also discussed.

CHAPTER 6. Comments And Future Work

The topic of this thesis was the design and implementation of a modular deadlock_free routing algorithm for a two-dimensional mesh, on a network of transputers of any size. This paper presented the mesh topology and the different routing techniques that can be implemented on them. The X-Y routing was selected for implementation on the mesh since it incorporates minimal routing and is deterministic. Also the store and forward technique was used for switching. The advantage gained from this switching technique was that there was a greater link efficiency and there was no need for simultaneous availability of the sender and reciever.

For avoiding deadlocks, the X-Y routing in mesh enforces a strictly monotonic order on the dimensions traversed and thus guarantees deadlock-free routing.

For future work, the wormhole routing technique can be implemented instead of the store and forward switching technique one used in this research. Since wormhole routing works in a pipelined fashion, the absence of network contention makes the network latency relatively insensitive to path length [1]. This is also shown in Figure 1.1 where the store and forward approach, circuit switching and wormhole routing are compared in communication latency.

Another point that was neglected in this paper but would be of interest is the size of the buffer at each processor. The larger the size of the buffer, the lesser the possibility of a deadlock [4]. Instead of a buffering process, a process that manipulates a queue, that is, a data structure that stores and recieves messages can be used.

Adaptive routing is another issue that can be implemented on the network of transputers. Deterministic routing cannot respond to dynamic network conditions such as congestion. In order to do that requires additional channels which has a cost associated with it, and also in most applications the utilization of these extra channels is not high [1].

Lastly virtual channels were also used to implement deadlock-free routing in this research. The way the algorithm has been developed in this paper, the unidirectional virtual channels used provide the same result and were thus ineffective.

References

1. Lionel M. Ni, Philip K. McKinley, "A Survey of Wormhole Routing Techniques In Direct Networks", Computer. Feb 01 1993 v 26 n2.

2. G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker, Solving Problems On Concurrent Processors, Prentice Hall, Englewood Cliffs, N.J 1988.

3. Yuh Dauh Lyuu, Information Dispersal And Parallel Computation, Cambridge University Press, 1992.

4. C. Nevison, D. C. Hyde, G. Schneider, and P. Tymann, "Laboratories for Parallel Computing", Jones and Bartlett Pub. Boston, 1994.

5. K. Hwang, Advanced Computer Architecture, McGraw Hill, 1993.

6. F. Thomson Leighton, Introduction To Parallel Algorithms & Architectures, Morgan Kaufmann Publishers, CA 1992.

7. InMos The Transputer Databook, 2nd edition 1989, Berkely, California.

8. D. H. Linder and J. C. Harden, "An Adaptive and Fault-Tolerant Wormhole Routing Strategy for k-ary n-cubes," IEEE Trans. Computers, Vol. 40, No. 1, Jan. 1991, pp. 2-12.

9. Tanenbaum, A S Computer Networks, Prentice Hall, Englewood Cliffs, NJ, 1981.

10. Seitz, C "The Cosmic Cube" Comm. . ACM vol 28, No. 1 Jan. 1985 pp 22-

33.

11. Dally, W. J. "Fine-grain Message Passing Concurrent Computers", Proc. 3rd Conf. on Hypercube Concurrent Computers Vol. 1 1988 p. 2-12.

12. Kermani, P and Kleinrock, L. "Virtual Cut-through: a new communication switching technique", Computer Networks Vol. 3, 1979, pp. 267-286.

13. Roscoe, A. W., 1988 "Routing Messages Through Networks: An Exercise in Deadlock Avoidance", Parallel Programming of Transputer Based Machines, edited by Traian Muntean, Springfield, Va. : IOS (Proceedings of the 7th Occam User Group Technical Meeting, Grenoble, France), 55-79.

14. Ugo De Carlini and Umberto Villano, "The routing problem in Transputer-based parallel systems", Microprocessors and Microsystems, Vol 15. No. 1 Jan/Feb 1991 pp. 21-32.

15. Gerlernter, D "A DAG-based algorithm for prevention of store-and-forward deadlock in packet networks", IEEE Trans. Computers Vol C-30 No. 10, Oct 1981, pp. 709-715.

16. Gunther, K. D. "Prevention of deadlock in pacaket switched data transport systems", IEEE Trans. Communications Vol COM-29 No. 4 April 1981, pp. 512-524.

17. Annot, J. K and Van Twist, R.A.H. "A novel deadlock free packet switching communication processor", in de Bakker, J.W., Nijman, AJ and Treleaven, PC (eds) Lecture notes in Computer Science 258 Springer Verlag, NY 1987, pp. 68-

85.

18. N.T. Son and Y. Parker, "Adaptive deadlock-free packet routing in Transputer-based Multiprocessor Interconnection Networks," The computer Journal, Vol. 34, No. 6, 1991 pp. 493-502.

19. G. D. Pifarre, L. Gravano, S.A. Felperin, and Jorge L.C. Sanz, "Fully adaptive Minimal deadlock-free packet routing in hypercubes, meshes and other networks: Algorithms and simulations", IEEE transactions on parallel and distributed systems, Vol. 5, No. 3, March 1994 pp. 247-253.

20. Dick Pountain and David May, "A Tutorial Introduction to OCCAM programming", 1988, Hollen St. Press, U.K.

21. InMos, OCCAM2 toolset, User Manual part 1, InMos Ltd. 1991.

22. Peterson, James L., and Abraham Silberschatz, 1985. Operating Systems concepts, 2d edition, New York: Addison-Wesley.

23. Hoare, C. A. R., 1974. Monitors: An Operating System Structuring Concept, Communications of the ACM, 17, pp. 549-557.

Vita

Syed H. Kirmani was born in Lahore, Pakistan on May 25th, 1968. He recieved his Bachelor's degree in Computer Science from Rutgers University, New Jersey in 1991. He also has a Bachelor's degree in Computer Science from The Canadian School of Management, Lahore Pakistan. Presently he is working as a consultant for Lehigh University Computing Center. Syed has worked for Wild Orchid Ltd, New York NY, where he helped design an information system for the company. He has also worked as a system administrator for Dr. Mirza M.D where he was responsible for the computer operations and managed a medical package for the clinic. He expects to receive his M.S. in Computer Science from Lehigh University in 1995.

END
OF
TITLE