Achieving Parallel Performance in Scientific Computations

Lyndon J. Clarke

Submitted for the degree of
Doctor of Philosophy

University of Edinburgh
September 1990

Declaration

This thesis has been composed entirely by myself.

The work reported in Section 2.4 (Deadlock Avoidance) was performed in collaboration with Michael G. Norman and with some assistance from Dominic M.N. Prior. All other work reported herein is my own, except where otherwise stated.

Lyndon J. Clarke

Acknowledgements

I should like to express my gratitude to Professor D. J. Wallace, Professor R. A. Cowley, Doctor R. D. Kenway and Doctor Alastair D. Bruce for academic and practical guidance given during my postgraduate study. Thanks are also due to all of my colleagues, past and present, in the University for helpful discussions, assistance in the use of various computer systems and advice freely given in the preparation of this thesis.

I should like to acknowledge the award of a Science and Engineering Research Council grant and the financial support of the Edinburgh Parallel Computing Centre, formerly the Edinburgh Concurrent Supercomputer Project.

Finally I am indebted to my parents and grandmother for financial assistance, and to Rachael for patience and moral support.

Abstract

The exploitation of high performance computing will be a major factor in the future advancement of science as computational methods are increasingly becoming a third discipline alongside theory and experiment. Despite advances which are being made in VLSI technology, enabling the construction of faster uniprocessor machines, it is now widely recognised that the future of high performance computing will be dominated by parallel architectures. It is of prime importance that the scientific community is able to effectively program such machines.

In the case of distributed memory MIMD architectures support is required for arbitrary communications between processing entities located at different processors, and operating on data stored in different memory units. If a message routing facility is not available in hardware it is necessary to provide a software implementation. Where this facility is available in hardware then a layer of software is required which presents the programmer with an interface to this hardware.

This thesis discusses a number of issues which arise in the implementation of message routing systems and the application interface. We have constructed such a system for use on arrays of INMOS transputers, called TINY, and the methods used in the implementation of this software are described. The system shares processor time with the application and we demonstrate that the processor bandwidth required by TINY is very small.

We have selected a concrete, but simple, application which utilises the services provided by this system. The implementation of this application was considerably simplified by the use of TINY and we show that the overheads induced by this software layer are insignificant. The application selected performs rendering of space filling molecular models, reflecting the growing importance of visualisation in science.

Contents

List of Figures

List of Tables

Chapter 1

Prologue

Parallel computers have now been in use by the scientific computing community in Edinburgh University for approximately a decade. During the past five years I have personally undertaken a number of applications in both SIMD (represented by the ICL-DAP) and distributed memory MIMD (represented by the MEIKO Computing Surface) architectures. The motivation for use of such machines is, of course, the possibility of solving physical problems which were previously intractable due to excessive execution times on conventional sequential machines, by exploiting parallel performance.

Calculations performed using the ICL-DAP have mainly featured the investigation of phase transition behaviour in simple spin models. In two specific calculations non-universal critical behaviour over a line of continuous transitions in the spin–$\frac{1}{2}$ Ising model [LB85] with first and second neighbour interactions, and tricritical behaviour in the spin–1 model with a quadratic spin self energy term [LS86], were studied extensively. The object of these studies was to utilise calculations of the free energy surface in coupling space in order to extract information about the relevant and marginal scaling fields. The computational core of these calculations is the Monte Carlo update of spin values for which the DAP is exceptionally well suited due to the large array of bit serial processors. In each case it proved very difficult to obtain data of sufficient statistical quality for the analysis proposed.

The MEIKO Computing Surface was used to investigate the behaviour of the two di-

mension Hubbard model which was thought to be relevant to the existence of high temperature superconductivity in ceramic oxides. The simulation technique employed was the Hybrid Monte Carlo algorithm [DKPR87]. The computational core of this method is the inversion of a large sparse matrix which we choose to perform using the conjugate gradients method; a suitable preconditioning matrix for this calculation is known. It was found to be very difficult to develop a program which executed rapidly enough to be of use in production calculations. Each conjugate gradient iteration requires (at least) two global summations to be performed and the associated matrix vector multiplication requires a relatively straightforward nearest neighbour exchange of edge data in the regular domain decomposition. The performance barrier was the fact that the communication overheads and corresponding performance degradation associated with these operations, particularly the global summation, precluded the use of large processor arrays.

It is uniformly observed that the effective programming of parallel computers has been considerably more difficult than conventional sequential programming. This has been especially true in the MIMD case where there is far greater freedom in machine architecture and programming methodology, and generally poorer support for the application programmer.

These machines have almost exclusively been programmed in the so-called "explicit message passing" paradigm, in which the programmer is responsible for coding data transfer between concurrently executing processing entities. The implementation of communication has often been the most difficult problem experienced by scientific programmers faced with an algorithm and such a computer.

Where the communication structure of the algorithm requires communication between entities which cannot be located on the same or neighbouring processors then it has been necessary to write message routing code for each application. Such software is notoriously difficult to code correctly, worse to debug, and was often very inefficient. This is unfortunate since the communication overheads associated with the parallel adaption of algorithms play a large part in controlling the parallel performance which may be achieved [Fox89].

I was convinced that this scenario was unnecessary since general purpose message

routing software could be written. Such an approach has the added advantage that time can be taken to ensure that the general purpose software is both correct and efficient. The bulk of this thesis describes work I have performed in expediting this approach.

In Chapter 2 issues arising in the design of a general purpose communication system are discussed. The chapter divides the problem into two major components; the application interface and message through routing. The division is facilitated by decomposing the system into two logical layers by analogy with the treatment of communication systems in computer networks.

This work has lead to the construction of a communication system for multiprocessors composed of INMOS T800 transputers. The implementation of this system, which uses ideas discussed in the previous chapter, is described in Chapter 3. This system shares CPU with the application program and it is therefore of particular importance that the processing bandwidth consumed should be minimal. The techniques developed to meet this requirement are discussed in detail.

The communication system has been incorporated into a large number of applications. In Chapter 4 one of these applications, implemented by myself, in the field of molecular graphics is described. The effect of communication overheads on the parallel adaption of the algorithm are analysed, and based on data obtained in the previous chapter they are found to be negligible. The system also provides a platform on which software presenting the programmer with a higher level model of parallel programming can be implemented. Examples of such models are LINDA [Gel85] and FORTNET [AH89].

Chapter 2

Design

2.1 Introduction

The construction of computer networks has generated a great deal of expertise in communication systems [Tan89] which the multicomputer community could well hope to benefit from. Indeed there is superficially a close similarity between the requirements of the communication system in a computer network and a multicomputer; each must provide end-to-end connections between several users on many machines in a fashion independent of the hardware in use. There is, however, a fundamental difference between computer networks and multicomputers; in a computer network the users are human beings who typically run independent programs and communicate with one another rather infrequently whereas in a multicomputer the "users" are elements of a single program which probably communicate with one another very frequently.

This difference between a computer network and a multicomputer has a number of consequences for the behaviour and requirements of the users, and for the hardware involved, for example:

- The users of a computer network require access to mail facilities, text editors, sophisticated document preparation systems; the users of a multicomputer require access to a high performance processor.

- The users of a computer network will tolerate failure in parts of the network, they go for lunch; the users of a multicomputer cannot tolerate failure in parts of the computer, the program fails.
- The number, and location, of users in a computer network changes frequently; the placement of application processes in a multicomputer is, almost invariably, static.
- The hardware of a computer network can be found in different buildings, cities and even countries; the hardware of a computer is usually found in a small number of boxes, often just one, in the same room.
- The topology of a computer network is variable; multicomputers often have fixed topology, and where the machine is reconfigurable this is (almost) never performed during program execution.
- The interprocessor communication links of a computer network are inherently noisy; the links inside a multicomputer are generally error free.

These observations naturally lead to different requirements for the properties and interfaces to the two types of communication systems. We would in principle like the multicomputer to be able to access facilities of the network, and users of the network should have access to the facilities of the multicomputer. It is clear that there should be a well-defined boundary and interface between the two systems.

The following assumptions about the requirements of the user and the underlying hardware are usually made:

1. Application processes generally require the illusion of complete connectivity.
2. The mapping of application processes onto processors is arbitrary but static.
3. The intercommunication links are completely error free and allow bidirectional traffic.
4. The interconnection topology of the processor array is arbitrary but static.

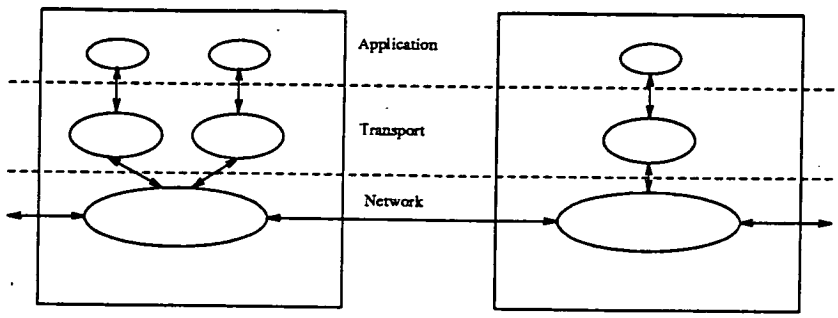

Figure 2.1: A three layer model.

It is convenient to divide a communication system into a number of *layers* each of which address separate issues arising in the design and implementation of these systems. For example the problem of efficient and correct routing of messages between processors is distinct from the problem of providing a suitable interface to the programmer.

In Section 2.2 we present an overview of the system decomposition we consider and the interfaces between the component layers. Sections 2.3 and 2.4 discuss the design of system components in detail.

2.2 System Overview

We consider a conceptual three layer structure as depicted in Fig 2.1. The components, *network*, *transport* and *application*, are named after layers of the OSI model [DZ83]; they perform approximately similar functions.

Application This layer contains all user processes. Application entities enjoy the illusion of reliable communications between all pairs. This layer approximately corresponds to layer 7 of the OSI model. Each application entity uses the services of a single transport entity.

Transport This layer is the interface between application and network. Transport is responsible for maintaining the security of communications between application entities. This layer approximately corresponds to layers 4 to 6 of the

OSI model. Each transport entity provides services to a single application entity and uses the services of a single network entity.

Network This layer is the actual message carrier. Network is responsible for passing message data between transport entities. It should avoid internal deadlock states without discarding messages. This layer approximately corresponds to layers 1 to 3 of the OSI model. Each network entity provides services to one or more transport identities.

The boundary between transport and application is very real. This is also the boundary between the communication system and the user and as such defines the user model of communication. This interface is discussed at length. The boundary between network and transport may be somewhat artificial and the interface is dealt with rather briefly.

2.2.1 Communication Models

The manner in which messages are communicated, the *communication model*, plays an important role in the implementation of all programs. We consider two models of communication that could be implemented and used, and indeed have been, *asynchronous* and *synchronous* communication.

In the asynchronous models messages appear to be sent from one process to another via an intervening medium which has a bounded capacity to store messages and is shared by all processes. The source process can send a message, for some destination, only when the medium is able to accept a message and may not wait for the destination to receive the message. The destination process can receive a message only when the medium contains a message for it, which the medium is able to deliver. The medium may be unable to deliver messages accepted from some sources because some destinations have not accepted messages which are deliverable. The medium may or may not guarantee that when a destination D receives a message from a source S there are no earlier messages for D from S within the medium. If such a guarantee is made then the medium is ordered, we say it preserves message

sequentiality, else it is unordered. Since the medium has a bounded storage capacity the communication of arbitrarily large messages must be effectively synchronous and therefore also sequential whether or not the medium is ordered.

In the synchronous model messages appear to be sent directly from one process to another without being stored by an intervening medium. The source process can send a message, for some destination, only when the destination waits to receive the message; there is always a point at which the two processes are both waiting for transfer of the message to terminate although they may not resume simultaneously as measured in an external frame of reference. Message sequentiality is implicit. The synchronous model is implemented within an underlying asynchronous model by the use of a synchronisation protocol.

The question of which model to provide is not trivial. The asynchronous model necessarily offers more communication bandwidth and lower message latency than the synchronous model since no protocol messages are required. The synchronous model offers a precise definition of communication and its implementation can ensure that messages will always be delivered.

2.2.2 Transport Interface

The transport interface defines the services provided to application. It should be intuitive enough for general purpose users and at the same time powerful enough for sophisticated utility packages to be written. We discuss the issues of process identification, message addressing and selection, communication models, and present an example procedural interface.

Process Identification

In order to use a multicomputer there must be a *partition* of the processing and data of a program into a number of processes, which have distinct address spaces and exchange information by communication of messages, and a *placement* of pro-

cesses onto physical processors, in which at least one process is mapped onto each processor. The partitioning problem is determined mostly by details of the program whereas the placement problem is determined mostly by details of the multicomputer. In order that the partition can be defined without reference to the, perhaps as yet undetermined, placement we require that the identification of these *internal* processes should be defined by the partition. It is intuitive to identify such processes by a sequence of natural numbers, for example when there are N internal processes one might label them by integers in the interval $(0, N]$.

Every program must have a means of collecting input and reporting output, typically via the utilities of an interactive terminal, file system, graphical device ... Whether a device exists internally within the multicomputer or externally within the computer network there must be a process within the multicomputer which interfaces between the user and the device providing a defined utility. Such interface processes are not considered to be part of the program but rather part of the environment within which the program executes and they are properly accessed by a procedural interface provided by the environment; the identification of these *external* processes should be defined by the environment. It is intuitive to identify such utilities by a textual string, for example one might label the terminal by the name "tty". The environment can provide a mapping of such names into integers, outwith $(0, N]$, which identify the external process. Were all utilities *localised*, in the sense that they are provided by a single process, as the examples given above tend to be, then this method would be quite adequate. However we cannot discount the possibility of utilities which are *distributed*, in the sense that they are provided by many (communicating) processes, examples of which could be concurrent file systems, linear algebra packages, graphical rendering packages ...

There is a sense in which each utility, whether distributed or localised, is a program which co-operates with the user program and as such the partition of each program defines a set of labels for the processes it contains. The case of such a collection of programs can be homogeneously handled by introducing the concept of a *group*, a set of processes performing some named function. One such group is the user program, others are the utilities which the environment of the user program provides. In this method each group is identified by a textual string, its name, and the

environment provides a mapping of group names into unique integer group labels. The processes internal to a group are defined by the group and can be labelled by a set of natural numbers as we previously labelled the processes internal to the user program; the system wide unique identifier of a process becomes a (group, process) label pair. The group mechanism need only be visible to programmers implementing utilities on the part of whom we can assume some sophistication; of course this does not prevent the mechanism being available to general purpose users.

We have frequently referred to the idea of a process as a processing entity with a private address space. In general we would like to avoid the constraint that such entities are purely sequential and allow them to be composed of a number of concurrent *threads* which share a common address space. This allows programmers to write programs which overlap communication with calculation within a process, which may in turn lead to a more effective overlap of calculation with calculation over a set of processes, and thereby enhance the performance of the program running on the multicomputer; as physical scientists we do, after all, have only one reason for using such a machine.

Addressing, Selection and Models

Processes exchange information by communication of messages. It seems then that a fundamental property of a message, aside from the data it may contain, is the pair of identifiers defining the source and destination processes; it is natural to think that processes send and receive messages. When the source wishes to send a message it specifies the identifier of the destination process, or perhaps the identifiers of permissible destination processes; an act of *addressing*. When the destination wishes to receive a message it specifies the identifier of the source process, or perhaps the identifiers of permissible source processes; an act of *selection*. Another property we could assign to a message is a *type* which in some way represents the meaning of the message; either or both of addressing and selection can be expressed in terms of, or in combination with, types. It is convenient to specify a type label as an integer in the interval $(0, T]$ where T is a statically bounded number of message types.

Selection Above we hinted at the possibility of conditional output, in addressing, and conditional input, in selection. It is not necessary to provide both of these, however it is necessary to provide one and a system which offers conditional input as opposed to conditional output will be more intuitive to use. Another reason for choosing to provide conditional input lies in the fact that we can completely and sensibly define a conditional input without reference to the communication model whereas this is not true of conditional output. The question of how conditional input is presented is important. It is generally sufficient to allow a process to specify a number of (group, process) labels as alternatives however the time taken to determine which alternatives can be satisfied and which should be chosen necessarily increases as the number of alternatives specified; this is a real performance problem. A less formal method, which can be implemented reasonably efficiently and which experience suggests will nevertheless be very powerful, is to allow selection to be specified as a (group, process, type) label triplet, in which either process or group is allowed to be a *wildcard* indicating all, from which the goal is determined by the following scheme.

```
if       group label is wildcard
         any message with given type
else if  process label is wildcard
         any message from given group with given type
else
         any message from given (group, process) with given type
```

Addressing We have discussed only point-to-point messages, in which there is exactly one source and one destination, thus far. In practice it is frequently necessary to send the same message to a significant number, i.e. close to N, of destinations. Of course this can always be achieved by sending the message to each in turn but it is often possible for the communication subsytem to perform the operation more efficiently when it provides for broadcast message routing. We take advantage of the structure implicit in the group mechanism to define a broadcast as a message sent to all members of a group and a global broadcast as a message sent to all members of all groups, both subject to the exception that the message is not delivered

to its source. We could allow addressing to be specified as a (group, process, type) label pair triplet, similar to the specification of selection, from which the action is determined by the following scheme.

if	group label is wildcard
	global broadcast with given type
else if	process label is wildcard
	broadcast to given group with given type
else	
	send to given (group, process) with given type

Models We would like transport to provide for both synchronous and asynchronous models of communication. A message is labelled by a (source, destination, type) triple, where source and destination are process identifiers; it makes no sense for messages with the same label to be communicated within different models at different times. The communication model of each message should be statically determined by its label. It is convenient to allow this to be determined by the source process group label alone. We anticipate that a group will statically specify the types and associated models for internal communication, and will similarly specify the types and models of external messages which will be used to provide the services offered by the group. The user group does not provide services, it only uses those of other groups, and need only specify the types and models for its internal communication since any external communication is determined by the specifications given by other groups.

Procedural Interface

Just as utilities provided by the environment are properly accessed by a procedural interface the services offered by the communication subsystem should also be accessed by a procedural interface. This readily allows such services to be accessible from any number of languages and on any number of multicomputers provided that the necessary procedures can be written.

We give an example functional interface, in ANSI C, which exploits the ideas discussed above. Groups, processes and types will be labelled by integers. Message selection and addressing will be performed with a (group, process, type) label triplet as discussed and provision for hiding the group mechanism from general purpose users is explicitly made.

Symbolic Constants The following symbolic constants are declared:

`WildLabel` The wildcard label.

`BadGroup` Indicates a bad group parameter.

`BadProcess` Indicates a bad process parameter.

`BadType` Indicates a bad type parameter.

They have distinct negative values.

Names, Labels and Sizes Functions are provided for mapping a group name into a group label, and vice versa, determining the size of a group with a given label and determining the process label and enclosing group label of the calling process.

.....

```
int groupLabelFromName(char *groupName)
```

Determine a mapping of group name to group label.

Return the label of the group with name `groupName`, else `BadGroup` if there is no group with the given name.

.....

`char *groupNameFromLabel(int groupLabel)`

Determine a mapping of group label to group name.

Return a reference to the name of the group with label `groupLabel`, else a null reference if there is no group with the given label.

.....

`int groupSizeFromLabel(int groupLabel)`

Determine the size of a group.

Return the size of the group with label `groupLabel`, else `BadGroup` if there is no group with the given label.

.....

`int getGroupLabel()`

Return the label of the group containing the calling process.

.....

`int getGroupSize()`

Return the size of the group containing the calling process.

.....

```
int getProcessLabel()
```

Return the process label of the calling process.

.....

Group Internal Messages Two functions provide for sending and receiving messages between processes within the same group and do not require a group label parameter.

.....

```
int isend(int *process, int type, void *data, int size)
```

Send the message in the buffer referenced by `data` of length `size` bytes and of type given by `type`. The message is a broadcast to the group if the integer referenced by `process` has the value `WildLabel`, else a single send to the given process.

Return a non-negative integer, whose value is undefined, if the send was successful, else return one of:

`BadProcess` The process label was invalid,

`BadType` The type label was invalid.

Note:

1. The function blocks until the message can be sent.

.....

```
int irecv(int *process, int type, void *data, int size)
```

Receive a message into the buffer referenced by `data` of length `size` bytes and of type given by `type`. The message will be received from any process in the group if the integer referenced by `process` has the value `WildLabel`, else from the given process.

Return a non-negative integer, whose value is the message size, if the receive was successful, else return one of:

`BadProcess` The process label was invalid,

`BadType` The type label was invalid.

Note:

1. If the message was longer than `size` then only the first `size` bytes are received and the remainder are lost. This is strictly an error and may be detected because the return value is greater than `size`.
2. The process label from which the message was received is stored in the integer referenced by `process`; this only affects that value if it was previously `WildLabel`.
3. The function blocks until a suitable message can be received.

.....

Group External Messages Two functions provide for sending and receiving messages between processes in different, or the same, groups and take a group label parameter.

.....

```
int esend(int *group, *process, int type, void *data, int size)
```

Send the message in the buffer referenced by `data` of length `size` bytes and of type given by `type`. The message is a global broadcast if the integer referenced by `group` has the value `WildLabel`, else a broadcast to the given group if the integer referenced by `process` has the value `WildLabel`, else a single send to the given process in the given group.

Return a non-negative integer, whose value is undefined, if the send was successful, else return one of:

`BadGroup` The group label was invalid,

`BadProcess` The process label was invalid,

`BadType` The type label was invalid.

Note:

1. The function blocks until the message can be sent.

.....

```
int erecv(int *group, *process, int type, void *data, int size)
```

Receive a message into the buffer referenced by `data` of length `size` bytes and of type given by `type`. The message will be received from any process in any group if the integer referenced by `group` has the value `WildLabel`, else any process in the given group if the integer referenced by `process` has the value `WildLabel`, else from the given process in the given group.

Return a non-negative integer, whose value is the message size, if the receive was successful, else return one of:

`BadGroup` The group label was invalid,

`BadProcess` The process label was invalid,

`BadType` The type label was invalid.

Note:

1. If the message was longer than `size` then only the first `size` bytes are received and the remainder are lost. This is strictly an error and may be detected because the return value is greater than `size`.
2. The group label from which the message was received is stored in the integer referenced by `group`; this only affects that value if it was previously `WildLabel`.
3. The process label from which the message was received is stored in the integer referenced by `process`; this only affects that value if it was previously `WildLabel`.
4. The function blocks until a suitable message can be received.

.....

2.2.3 Network Interface

The network interface defines the services provided to transport. If transport implements synchronous communication between application entities then there is little point in network providing a synchronous model between transport entities. It makes most sense for network to provide an asynchronous model leaving flow control and synchronisation to the higher layers.

Each network entity has a number of conceptual ports, one connected to each transport entity, over which the network and transport entities communicate. Transport

entities are addressed by a (processor, port) number pair, which can be packed into an integer; the abstraction from such placement details enjoyed by application can be provided entirely by transport which maps application process indentifiers into transport addresses.

We could conceive of the service being provided by a set of functions, which provide for directed asynchronous messages which are either ordered, in the sense that an ordered message will not arrive at its destination before an earlier ordered message, or unordered, and for asynchronous broadcast messages.

.....

```
void unordered(int transport, void *data, int size)
```

Send an unordered message of length `size` bytes buffer referenced by `data` to the transport entity identified by `transport`.

.....

```
void ordered(int transport, void *data, int size)
```

Send an ordered message of length `size` bytes buffer referenced by `data` to the transport entity identified by `transport`.

.....

```
void broadcast(void *data, int size)
```

Broadcast message of length `size` bytes in the buffer referenced by `data` to all other transport entities.

.....

```
void receive(void *data, int *size)
```

Receive a message into the buffer referenced by `data` returning the size in `size`.

.....

Of course, in practice transport would like to avoid taking a copy of data which must later be copied again to application, if this is possible, and these functions may not be realistic. In fact such a set of functions will probably not even exist; when network and transport share the same CPU and memory then the boundary between them should be artificial and when network comprises specialised routing hardware then the interface would probably be interrupt driven.

2.3 Transport Layer

In this section we discuss issues involved in the design of the transport layer. This layer provides service to the application layer using the service provided to it by network.

Our major requirements of transport are:

1. Transport supports the illusion of full connectivity and abstraction from process placement.
2. Transport fairly and efficiently implements the addressing and selection methods of the user model.
3. Transport maintains the reliability of communications between application entities, provided that no application entity indefinitely refuses to accept any deliverable asynchronous message.

Requirements 1 and 2 are self explanatory. The last point implies that in the synchronous model transport maintains the reliability of all communications whereas

in an asychronous, or mixed, model application is responsible for implementing flow control in asynchronous messages.

2.3.1 Asynchronous Models

Experience in using the asynchronous model has shown that it is often possible for relatively inexperienced users to write programs with very simple communication patterns, and that sophisticated users can succesfully create programs with rather complicated communication patterns.

The absence of imposed flow control affords greater performance but can lead to a deadlock situation where cyclical dependencies in the communication of user processes fatally interact with the uncontrolled sharing of network storage resources, a problem for which we have borrowed the term *hangup* since the network itself is not in a deadlock state. Some simple examples of hangup are:

- A number of application entities attempt to send messages to one another and will only attempt to receive messages after completion of sending. The storage capacity of the system is used up and it cannot accept further messages until application accepts messages from it; application waits forever to send messages to it. Although this is a simple problem to solve it may be difficult for the less sophisticated programmer to diagnose.
- An application entity is waiting for a message of a particular type which it never receives despite the fact that it has been accepted by the system. The storage capacity of the system which would be required to deliver the message is taken up by messages for this entity but of a different type and thus the entity must accept some of these before the system can deliver the required message; application waits forever to receive a particular message. This problem may be more difficult to solve and diagnose.

Hangup may be avoided when the amount of message data which can be outstanding for any destination which might indefinitely refuse to receive said data, because

it is waiting to receive other data or to send data, can be bounded either due to the structure of communications natural to the application or by explicit flow control. If this bound is not zero then the destination is assuming it may use the storage capacity of network; the destination must reverse this assumption by installing sufficient buffering threads to reduce the bound to zero. The fact that a large number of programs have now been written within the asynchronous model is testimony to the fact that these bounds are often readily calculable.

2.3.2 Synchronous Models

In order to implement the synchronous model a protocol is required between transport entities; this protocol determines when the source and destination application processes can be resumed, and when message data is transferred. The implementation need not suffer from the hangup problem since transport can trivially be arranged to behave as a sink.

Consider a very simple protocol, which requires a constant amount of memory at each transport. When a process wishes to source a message it sends the data immediately and waits for an acknowledge; when a process wishes to sink a message it waits to receive the message and then sends an acknowledge to the source. This achieves synchronisation between the source and sink but fails to avoid hangup, because messages are being stored in network. This problem can be removed by defining a retry message which transport sends back to the source whenever a message arrived but was discarded because the destination user process was not ready to receive the message; the source sends the message again whenever the retry message is received rather than the acknowledge. This busy wait technique is clearly capable of consuming enormous amounts of communication bandwidth.

We describe three further protocols, each of which require memory proportional to the number of processes, which use the communications bandwidth rather more efficiently.

Request-Transfer (RT) The source waits for a *request for transfer* message before sending the data transfer message. The destination sends the request message and waits for the transfer message. Transport requires a bit, or more realistically a word, at every source for every destination which may wish to receive from that source, with which to record the arrival of requests. This protocol has the disadvantage that it forces the destination to specify which source it wishes to receive from; conditional input is not possible.

Transfer-Acknowledge (TA) The source sends the data transfer message and waits for an *acknowledgement of transfer* message. The destination receives the data transfer then sends the acknowledgement. Transport requires a buffer at every destination for every source which may wish to send to that destination, with which to store the arriving data transfers. This protocol has the disadvantage that unless message buffers are very small an inordinate amount of memory may be required; it has the advantage of offering functionality well aligned to the requirements of applications.

Request-Acknowledge-Transfer (RAT) The source sends a *request to transfer* message and waits for an acknowledgement message before sending the transfer message. The destination chooses any request, sends the acknowledgement and waits for the transfer. Transport requires a bit, or more realistically a word, at every destination for every source which may wish to send to that destination, with which to record the arrival of requests. This protocol has the advantages of providing the right functionality and requiring relatively little memory; it has the disadvantage of being more expensive in communication of protocol. The time ordering of the request and acknowledge can be relaxed when the destination is performing an unconditional input if message sources also record the arrival of these "acknowledgement" messages.

2.4 Network Layer

In this section we discuss issues involved in design of the network layer. This layer provides services to the transport layer using the communication hardware of the multicomputer.

Our requirements of network are:

1. Network is capable of passing a message from any transport entity to any other transport entity, and from any transport entity to all other transport entities.
2. Network will not corrupt or discard messages, and any deliverable message which waits to be received by a transport entity will hold network resources indefinitely if required.
3. Network will not indefinitely fail to accept a message from any transport entity or indefinitely fail to deliver any accepted message, provided that no transport entity indefinitely fails to accept any deliverable message.

We demand 1 so that transport can always support the illusion of full connectivity and the abstraction from placement to application, and efficiently implement broadcast messages. In order for transport to securely implement synchronous messages 2 is essential. Deadlock and starvation in network are eliminated by 3 which we must ask in order to implement any reliable communications.

2.4.1 Switching Techniques

Two important, and very different, switching techniques used in communication systems are circuit and packet switching. Two more recent techniques are virtual cut-through and wormhole switching.

Circuit Switching

The circuit switching technique is easily understood by analogy with the telephone system. When one subscriber dials the number of another the telephone system attempts to find a line between the two; a line is a sequence of available *wires* from the caller to the receiver. If the receiver is engaged or a line could not be found then the call fails and any wires used are retrieved. If, on the other hand, the receiver was available then the line is held until the two parties have finished their conversations and replaced their handsets, or until the caller becomes bored of waiting for the receiver to pick up the phone! Key features of circuit switching are:

- The initial establishment of the end-to-end connection.
- The use of a dedicated set of wires during the connection.

These features have a number of immediate consequences for the point-to-point and aggregate bandwidth of the system.

After call connection the point-to-point bandwidth is very high since there is a private set of wires; there is no need to store the message at an intermediate node while it waits for the outgoing wire to become available. The transfer time depends only on the message length.

The time taken to establish a connection can be very large, as our experiences with the telephone system will confirm. The number of wires waited for increases as the path length. The probability of a wire being free decreases as the network loading increases. The waiting time increases with the length of message waited for. A cross over occurs from a set up time depending only on the path length to one depending on both the path length and average message length.

Since a connection which is waiting for a free wire is itself holding wires the achieved aggregate bandwidth falls off heavily with loading and is rather smaller than the maximum available.

Circuit switching has been applied to multicomputers, in hardware, in the Direct Connect Module of the Intel iPSC/2 [Nug88].

Packet Switching

If circuit switching is compared to the telephone system then packet switching should be compared to the postal system. When one correspondent wishes to communicate with another they write down the contents of the message, package them up in an envelope, put the address of the recipient on the envelope, post it and never hear of it again unless the recipient chooses to reply explicitly. The mail will only handle letters below a certain weight; assume there is no parcel post service, so a really large message will have to be sent as several separately addressed letters. During transit each letter shares resources with a number of other letters; there are no dedicated lines. Key features of packet switching are:

- Messages may be fragmented into one or more separate packets, each of which is separately addressed.
- Packets do not reserve private lines, each packet is completely received before being forwarded.

The performance properties of a packet switched system are quite the inverse of a circuit switched system.

When a message packet arrives at an intermediate node the output link it will require may, in general, be busy. Whether or not this is the case the packet is completely received into a buffer before any of it is forwarded; these are commonly known as *store-and-forward* systems. It follows that the transfer time depends on both the path length and message length. This is unimportant for sufficiently small messages but the latency will probably be unacceptable for large messages.

It is easy to see that, as in the analysis of circuit switching, one finds that the effect of network loading is to add a term dependent on both the path length and average message length.

The achievable aggregate bandwidth of packet switched systems is generally better than that of circuit switched systems since, given some number of buffers at each communication link, a message waiting to propagate to the next node does not prevent use of the link which is used to arrive at the current node.

Packet switching has been applied to multicomputers, in software, in the Cosmic Cube [Sei85], the Intel iPSC/1 [Int86], and a number of routing systems for transputer arrays.

Virtual Cut-through

The virtual cut-through technique is a direct hybridisation of the circuit switching and packet switching methods and is based on the observation that in a packet switching system it is not necessary to store the whole message at an intermediate node if the outgoing link is not busy. This requires hardware which allows direct link to link and link to memory copies.

Key features of virtual cut-through are:

- Messages may be fragmented into one or more separate packets, each of which is separately addressed.
- Packets are stored in intermediate nodes only when the chosen output link is busy.

The performance properties of virtual cut-through systems are similar to circuit switched systems, without the requirement of initial path set up, in very lightly loaded networks; they are optimal for point-to-point bandwidth. In heavily loaded networks the probability of the next link being free is very small and there is a cross over to the properties of packet switched system.

Virtual cut-through has been applied to an experimental transputer communication system [Mcn] but is not used in any commercially available systems for multicomputers.

Wormhole

This technique splits a message into a number of, typically rather small, units known as *flits* of which only the first few contain addressing. An intermediate node accepting the message completely receives the first flit, and maybe a small number of following flits, before forwarding these flits in a manner similar to store-and-forward systems. Remaining flits are not received until the first flit has been forwarded and are always forwarded in the same way, if the message must wait for a blocked output channel then all channels behind the head are also temporarily blocked. Key features of wormhole switching are:

- Messages are fragmented into a number of small flits, only the first few of which contain addressing.
- All flits of a message follow the same path and only a small number are received before forwarding the message.

The performance properties of wormhole switching are also optimal for point-to-point bandwidth in lightly loaded networks. The achievable aggregate bandwidth suffers in a similar manner to circuit switched systems.

Wormhole switching has been applied to multicomputers, in hardware, in [DS88] and in the C104 routing chip [Pou90].

Connectionless Switching

We define a switching model, *connectionless* switching, which embraces the store-and-forward, virtual cut-through and wormhole techniques. The name is intended to imply that there is no dedicated connection set up before message transfer takes place, unlike circuit switching which can be described as *connection-oriented.*

The network consists of *network elements*, or simply *elements*, which are network processes controlling the use of some network resource. A message consists of a

Feature	Store-and-forward	Virtual Cut-through	Wormhole
Element	Buffer set	Buffer set	Channel
Unit	Buffer	Buffer	Flit
u	$= 0$	> 0	$= 0$
v	> 0	> 0	> 0
$w(m)$	$= 0$	$\leq u$	> 0

Table 2.1: Feature Identification in Connectionless Switching

finite number of *message units*, or simply *units*. A unit is the largest piece of message data that may be forwarded to or by message elements. Elements have a finite capacity for storage of units. The first unit, or first few units, alone carry addressing information for the complete message.

1. An element Q accepts the first v units of a message m, from an element or transport P, which establishes the portion of the message path leaving P.

2. The v units accepted by Q contain all addressing information and the routing function is applied to this information. These units may be modified to produce a set of $v' \leq v$ units which allows for removal of redundant addressing etc.

3. The v' resulting units are forwarded to an element or transport R which establishes the portion of the message path leaving Q.

4. There may be a finite number, $w(m) \geq 0$, of remaining units which Q accepts from P and forwards to R.

5. Q has the capacity to store at most $u \geq 0$ of the remaining units and uses this storage capability to concurrently accept units whilst waiting to establish R in 3 or for which R is not ready in 4.

This model is a description of the three switching techniques when the feature identifications of Table 2.1 are made.

2.4.2 Routing Functions

Routing functions deal with the business of moving messages between processors which are not directly connected. We consider functions which perform *directed* routing, from one processor to another, and *broadcast* routing, from one processor to all others. These kinds of functions can be further divided into *static* functions which always use the same routes and *adaptive* functions which vary the routes, in an attempt to alleviate the effects of localised congestion. We consider static and adaptive directed functions, and static broadcast functions.

Directed Functions

When a message at some processor must be delivered to some other processor there are generally a number of routes it could take; we need some criteria of goodness to determine which route to use. A simple and often used criteria is the *shortest path*, i.e. one that crosses the minimum number of links. A slightly more general criteria associates a cost with transfer over each link and we choose a route which minimises this cost; the shortest path approach associates equal cost with all links. Whenever the processor graph contains cycles we are liable to find that a number of routes have equal cost and the static routing function chooses a single one of these possibilities for all directed messages. We will also, and more frequently, refer to this routing function as *sequential*.

In order to effectively utilise network bandwidth under heavy load it is necessary to attempt to avoid highly localised areas of congestion by either routing messages around these areas or preventing the occurence of such areas. The problem is to decide which communication link should be used to forward a given message so that it will reach its destination in the minimum time. Of course this problem cannot be solved without quite considerable knowledge of the network current and future states; a good guess is the best that can realistically be hoped for and congestion control algorithms are concerned with making these guesses.

When the network shares CPU with the application we require that it be purely

reactive which implies that a local, or isolated [Tan89], algorithm is employed; each node makes a routing decision based on information it has gleaned about the network without specific exchange of information between nodes. A simple example of such an algorithm is the *hot potato* method; each node simply forwards the message on whichever link it thinks will first becomes ready without regard as to whether the chosen link actually takes the message any closer to its destination. Another is the *diffusive* method in which each node forwards the message on any randomly chosen link.

These approaches can be combined with a static knowledge of which links are on minimal cost paths and the decision is then which of these links to use. A number of possible methods for choosing the next link in the route are now described. In the absence of realistic and well understood models of a network under heavy load it is not possible to compare quantitatively the performance of these methods; a qualitative discussion is given.

Least Used: The message is queued to the link which has been used the least. This method requires a counter for each link which is updated whenever the link is used. It is not able to adapt to local congestion and simply tries to use the links as evenly as possible.

Random Link: The message is queued to a randomly chosen link. This method is easy to implement, requiring only the added complexity of using a random number generator. It is not able to adapt to local congestion; the hope is that by partially randomising message paths the probability of localised areas of congestion is decreased.

Quiet Link: The message is queued to the link which has not been queued to for the longest period of time. This method requires the time of the last queueing event for each link to be recorded and the time for each suitable link to be inspected. The decision is based on the history of events which may not be a good guide to future events.

Short Queue: The message is queued to the link which has the shortest queue of pending messages. This method requires the queue of waiting messages for

each suitable link to be inspected. The decision is based on the current state of the node which is a good guide to future events if areas of congestion are sparse.

First Ready: The message is queued to the link which will first become ready. This method directly addresses the question of future events, and is certainly the best of the algorithms we describe here. It can be implemented, for example, by placing the message in all suitable queues; the first queue to become ready removes the message from other queues so that it is forwarded only once as required. Unfortunately this requires twice as many queueing events as, for example, the short queue method and more complicated queueing structures which will consume additional CPU.

Notice that the last of these methods differs fundamentally from the others in that the message is not committed to being forwarded by an unready link if another possible link is ready. The method can be useful in designing deadlock free routing functions, as we shall later see, where the set of output links are not necessarily on a minimal cost path. Whether this is the case or not we shall use the term *quasiadaptive* to describe all of these methods since they contain a definite static component.

The quasiadaptive and static functions are easy to implement in general topologies when we introduce a routing table at each processor which typically maps the destination processor, and perhaps the arrival link, into a set of possible output links. In this way every message contains the number of the destination node and a table look-up determines the set of output links; the static function uses the first of the alternatives whereas the quasiadpative function chooses between them. In geometrically regular topologies the routing table can often be dispensed with since a simple rule determines the set of output links.

Broadcast Functions

We could always effect a broadcast routing function by generating a directed copy of the message to every other processor, and we could do this in such a way that

every copy followed a minimal cost path. Observe that the same message, although a different copy, will pass over certain links a large number of times; clearly we can broadcast a message much more efficiently by avoiding this. We would ideally like a broadcast function to have the property that every broadcast message arrives at each processor, excluding the source, exactly once and preferably along minimal cost paths.

We can readily construct a static broadcast function which has these properties but we know of no adaptive broadcast functions which have both of these properties. We therefore limit ourselves to considering static broadcast functions.

When a minimal cost path from A to C passes through B then the portion of this path from B to C is necessarily a minimal cost path from B to C. We can exploit this trivial fact to build a directed tree with A at its root which contains every node and in which the path from A to any node is one of minimal cost. Consider the simple case where unit cost is associated with every link, shortest path routing. All processors neighbouring A become the children of A and A becomes their parent. These procesors then, in turn, examine their neighbours and only when a neighbour has no parent they take that neighbour as a child. This process is repeated until all processors have parents or equivalently it is not possible for any processor to take a child. Because our multicomputer is conjoint and its links are undirected we can perform this calculation for every processor.

The (static) broadcast function is also easy to implement in general topologies when we introduce a routing table at every processor which maps the source processor into a set of output links. Every broadcast message contains the number of the source processor and a table look-up determines the set of output links; the message is replicated (it need not be copied) and forwarded by each of these links, simultaneously if they are ready. In geometrically regular topologies it is again often possible to replace the table with a simple rule.

2.4.3 Deadlock Avoidance

Deadlock arises within network when there exists a set of network elements each waiting to transfer a message unit to another element within the set, such that none can proceed before any other. Deadlock can also occur involving transport elements, in which the resources of network are overloaded and it is the responsibility of transport to avoid overloading network.

Elegant solutions to the network deadlock problem in a small number of topologies are known and it is useful to introduce these methods before discussing the general problem. We then present a method for avoidance of deadlock in arbitrary toplogies based on these ideas.

Trees

It has long been known how deadlock may be avoided in a network with a tree shaped topology [MS80]; a message originating in processor s destined to processor d is simply forwarded along the unique shortest path through the tree, exploiting the fact that the tree is an undirected acyclic graph.

One associates a single network element with each directed link. Each network element will accept units from any adjacent element or a source. The routing function may forward a message to an element at any adjacent link, or a sink.

The dependency of network elements resolves into a directed acyclic graph, as depicted in Figure 2.2, and it is therefore not possible for a cycle of waiting network elements to exist.

Grids

The e-cube routing algorithm [DS88] solves the deadlock problem for multidimensional grids in such a way that all messages are delivered along shortest paths; in

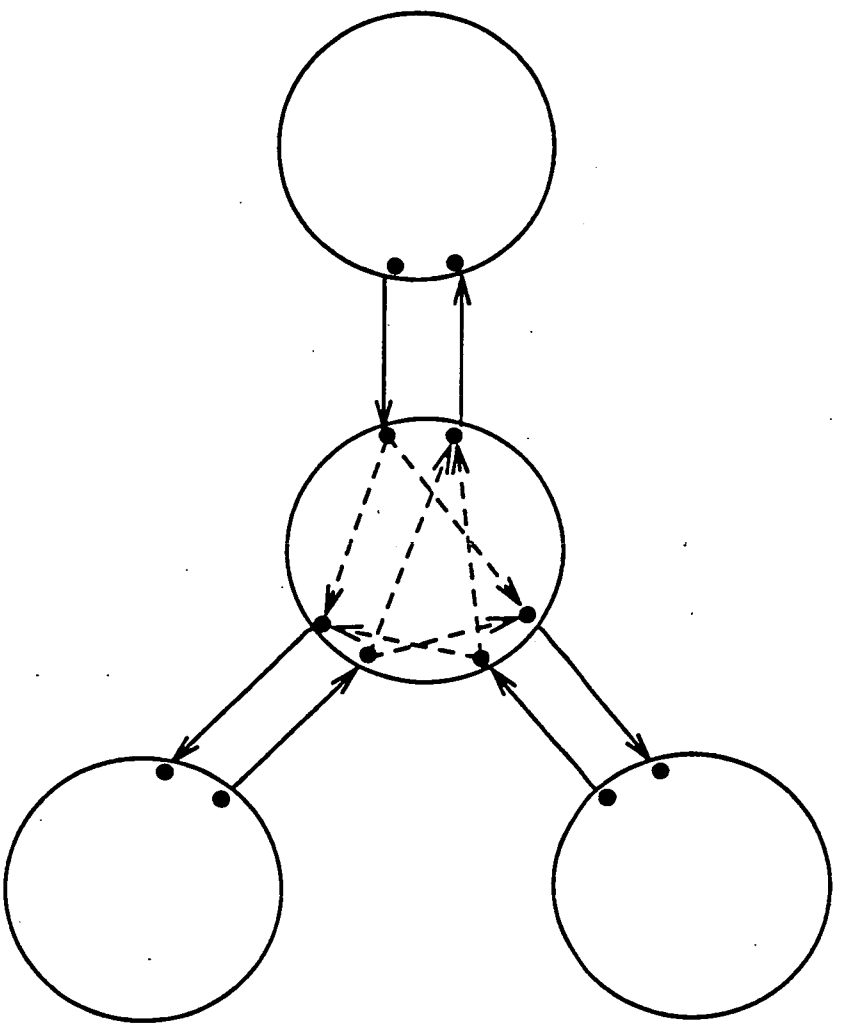

Figure 2.2: Routing in a binary tree.

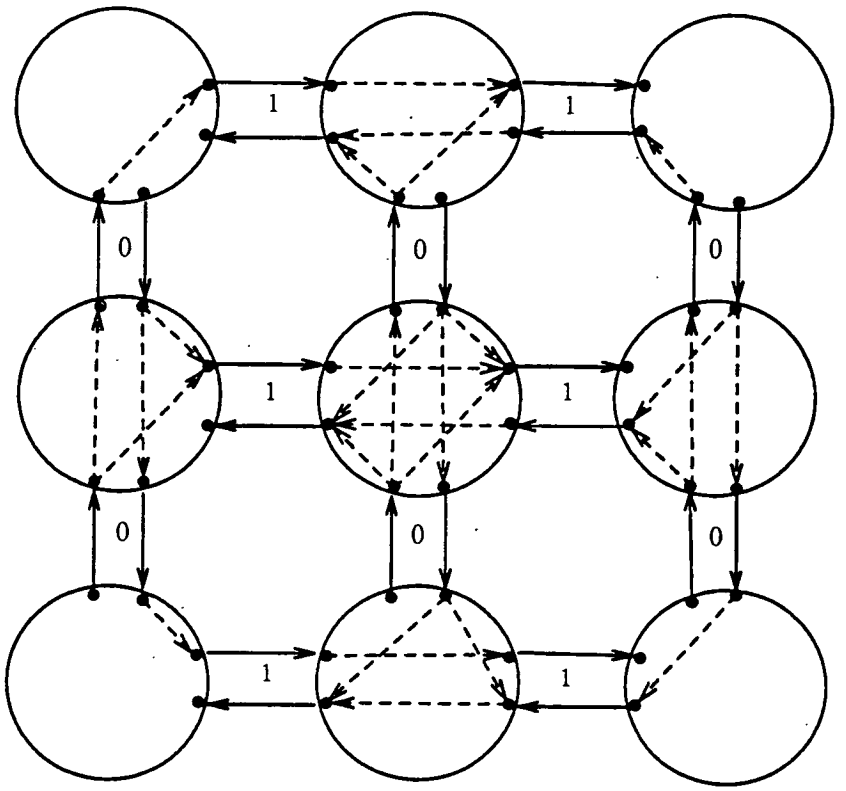

Figure 2.3: Routing in a 2 dimensional grid.

hypercubic grids the number of messages using any link is the same for all links. This algorithm exploits the fact that there are many shortest paths between nodes in these graphs and restricts the paths actually taken such that the graph of element, or link, dependencies is acyclic, as in the case of the tree graph.

One associates a single network element with each directed link. In a d dimensional grid one numbers the dimensions by natural numbers in $(0, d]$ and labels each link by the number of its dimension. Each network element will accept units from any sink and any adjacent element. The routing function may forward a message to an element with an equal or larger label, or a sink.

The graph of element dependencies again resolves into a directed acyclic graph, as depicted in Figure 2.3, and there is a path from every node to every other node. We shall refer to this technique as *acyclic* routing.

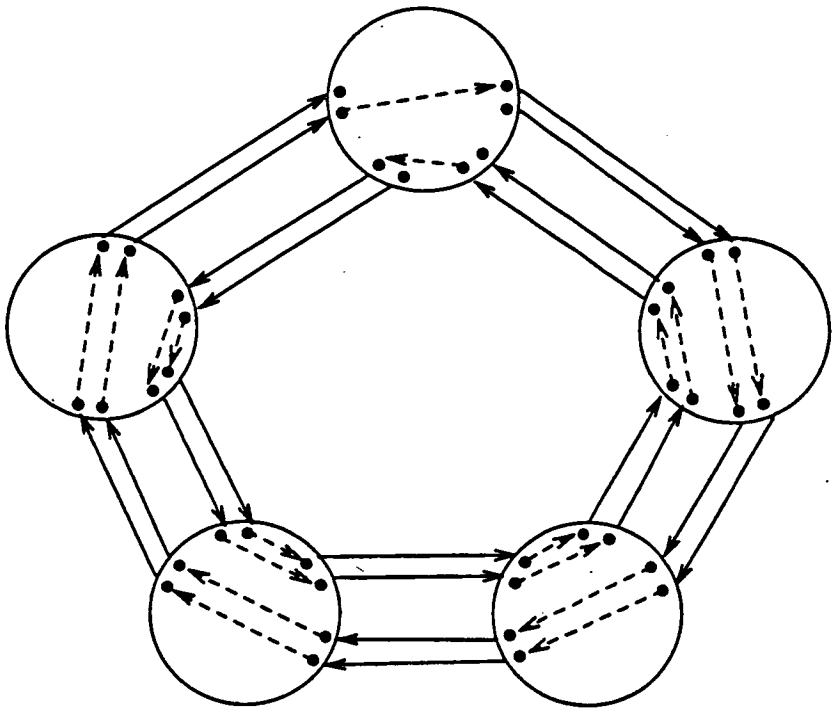

Figure 2.4: Routing in a Ring.

Rings

It is not possible to construct a routing function which leads to an acyclic dependency of network elements, as in the grid above, and preserves shortest message paths for all node pairs in rings of order greater than four.

The deadlock problem can be solved by use of acyclic routing when *multilinks*, an undirected link which is arranged to appear as a number of independent links, are introduced. A network element is associated with each directed component of the link. In the case of a ring we create two virtual links at each physical link, it is then possible to thread a chain through the graph which utilises all of the virtual links. Messages can be routed through this graph, shown in Figure 2.4, in a manner similar to the tree graphs above.

This is the *loop free buffer graph* technique of [MS80] ($w(\alpha) \leq u$), and the *virtual channel* technique of [DS88] ($w(\alpha) > u$).

In the regime $w(\alpha) \leq u$, there is another class of solutions in which one avoids

deadlock by ensuring that there is always a free buffer in each directed ring [Ros87]. This can be achieved in practice in a number of ways; for example a network element could simply not accept a packet from a transport element unless one of the buffer set associated with the network element is free. We shall refer to these methods as *circuit* routing.

General

We have discussed two routing techniques for avoidance of deadlock in network applied to specific topologies; we now describe how these methods can be applied to general topologies.

Acyclic Routing The method can be generalised to arbitrary topologies when one solves the problem of constructing an acyclic element dependency graph which nevertheless affords routes between all pairs of nodes.

One could trivially achieve this by exploiting the fact that any conjoint undirected graph contains a spanning tree; this would be a poor solution since very few of the links are utilised. We present below a method for calculation of link dependency which permits utilisation of all links. It is not generally possible to utilise shortest paths for all messages in this approach, consider for a moment the ring topology, however the method yields shortest paths in the tree and grid geometries as we should expect.

In the case of a multilink approach we can either analyse the multigraph directly or alternatively analyse the underlying physical graph and impose a linear ordering on components of multilinks. We take the latter approach in applying the results of our link dependency calculation to multigraphs for sake of simplicity in the algebra of the dependency calculation.

In the regime $w(\alpha) \leq u$, it is always possible for a message to utilise any outgoing link which is ready since this cannot cause a cycle of waiting elements. This has

been exploited in the *resource ordering* approach of [Gun81] and is included in our directed routing function.

Circuit Routing The circuit routing technique described above for a ring can be utilised in arbitrary topologies when one observes that because the links are bidirectional we can construct a single directed ring that utilises all of the directed links, and route around this ring [Ros87]. Alternatively one can exploit the fact that in even valency graphs one can construct an Eulerian cycle [NLW88] and route this cycle as the ring.

These approaches lead to message path lengths which increase linearly with the order of the graph. We do not pursue this approach.

Again it is possible to route a message to any ready output link and this has been exploited in [Sur90]. This is capable of yielding short routes in lightly loaded networks since necessary short cuts will almost always be free. The probability of the short cut being free decreases with increasing network loading; message path lengths are extended which in turn exacerbates loading.

2.4.4 Acyclic Routing

Definitions

The description of the interconnection network will frequently refer to properties of graphs; it is useful to define the terms used at this early point. In this context a *graph* is a directed graph, i.e. there is a direction associated with edge, and we treat undirected graphs by constructing two opposing directed edges from each undirected edge.

> A graph (V, E) is *strictly undirected* if for every edge E_{ij} there is an opposing edge E_{ji}, i.e.
>
> $$E_{ij} \Rightarrow E_{ji} \; \forall i, j.$$

A graph (V, E) is *strictly directed* if for every edge E_{ij} there is no opposing edge E_{ji}, i.e.

$$E_{ij} \Rightarrow \neg E_{ji} \; \forall \, i, j.$$

A *path* within a graph (V, E) is subset of E leading continuously from one of V to another. The predicate $\mathcal{P}_{i \to j}^{(E,V)}$ is true iff there is a path in E from V_i to V_j, i.e.

$$\mathcal{P}_{i \to j}^{(V,E)} \Rightarrow E_{ij} \vee (E_{ik} \wedge \mathcal{P}_{k \to j}^{(V,E)}).$$

A graph (V, E) is *conjoint* iff there is path in E from every vertex to every other vertex, i.e.

$$V_i \wedge V_j \wedge i \neq j \Rightarrow \mathcal{P}_{i \to j}^{(V,E)} \; \forall \, i, j.$$

A graph (V, E) is *acyclic* iff it contains no paths from any vertex back to itself,

$$V_i \Rightarrow \neg \mathcal{P}_{i \to i}^{(V,E)} \; \forall \, i$$

A graph (V, E) is *loosely acyclic* iff every strictly directed subgraph (V', E') of (V, E) is acyclic.

Components

Familiar features of the interconnection network are the processors and communication links. We denote the set of processors by P and refer to an individual processor with label i as P_i. We denote the directed communication links by L, each element of L is a processor pair ij, $i \neq j$, indicating that processor P_i is physically connected to processors P_j.

We demand that the physical links are bidirectional so that the graph (P, L) is strictly undirected; each physical link is a pair of directed communication links L_{ij}, L_{ji}. We also demand that self links, i.e. a physical link which connects a processor to itself, be absent. Multiplet links, i.e. a set of more than one physical links connecting the same pair of processors, are formally merged into a single physical link.

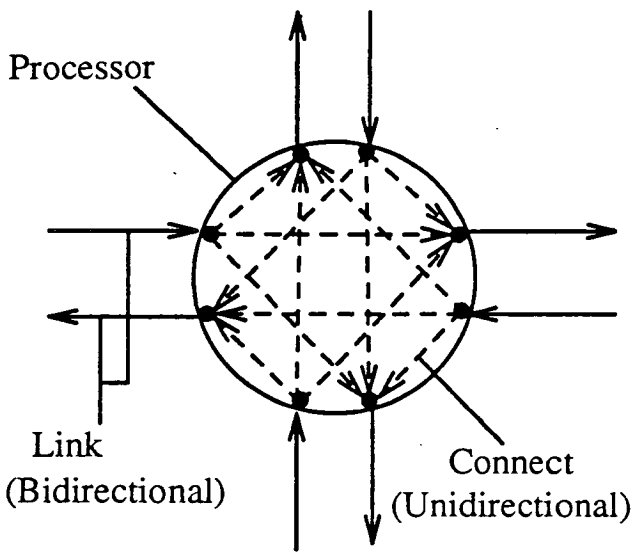

Figure 2.5: Processors, Links and Connects.

We express communication link dependency by the *connect matrix*, denoted by C. Each element of C is a processor triple C_{ijk}, $i \neq j \neq k$, which determines the way in which L_{ij} may, or may not, forward data to L_{jk}. Note that the graph (L, C), in which C_{ijk} is shorthand for $C_{ij\,jk}$, is strictly directed. The relationship of processors, links and connects is depicted in Figure 2.5. We represent an interconnection network by (P, L, C), the combination of these three elements.

We define some useful quantities with which to describe an interconnection network.

A *route* within an interconnection network (P, L, C) is a subgraph of (L, C) which leads continuously from one processor to another. The predicate $\mathcal{R}_{i\to l}^{(P,L,C)}$ is true iff there is a route in (L, C) from P_i to P_l, i.e.

$$\mathcal{R}_{i\to l}^{(P,L,C)} \Rightarrow L_{il} \vee \neg(\neg(L_{ij} \wedge \mathcal{P}_{ij\to kl}^{(L,C)} \wedge L_{kl})\ \forall j, k).$$

An interconnection network (P, L, C) is *fully connected* iff there is a route from every processor to every other processor, i.e.

$$P_i \wedge P_j \wedge i \neq j \Rightarrow \mathcal{R}_{i\to j}^{(P,L,C)}\ \forall i, j.$$

An interconnection network (P, L, C) is *cycle free* iff (L, C) acyclic, i.e.

$$L_{ij} \Rightarrow \neg\mathcal{P}_{ij\to ij}^{(L,C)}\ \forall i, j.$$

Simple Graphs

Routing We associate a network element with each link L_{jk}. Each element will accept the first unit of a message from an element at any adjacent link L_{ij} and any source at P_j by performing a conditional input from all such elements and sources.

An element at link L_{ij} will establish the next link for a message α, destined for a sink at processor $d(\alpha) \neq j$, by performing a conditional output of the first unit of α to all links L_{jk} satisfying

$$(\neg(\neg \mathcal{P}^{(L,C)}_{jk \to k' d(\alpha)} \forall k') \wedge (C_{ijk} \vee w(\alpha) \leq u).$$

A source at processor P_j will similarly establish the first step of a message by conditional output to all elements L_{jk} satisfying

$$\neg(\neg \mathcal{P}^{(L,C)}_{jk \to k' d(\alpha)} \forall k')$$

Note that these rules do not prevent messages $w(\alpha) \leq u$ from *looping*; i.e. repeatedly following the same path. It is therefore necessary to restrict the links which may be used in through routing in order to avoid this phenomenon and any restriction which contains at least one of the links for which C_{ijk} is true will be permissible.

Multigraphs

We extend the notation for an interconnection network to explicitly handle multilinks. An extended network is constructed from an underlying cycle free and fully connected physical network, without multilinks, by imposing a linear order on the components of multilinks.

Extended Interconnection Network Let each communication link L_{ij} be composed of M_{ij} multilinks. Denote components of the multilink by L'_{inj}, $n \in (0, M_{ij}]$ with unique labels n.

Denote connects between multilink elements by C'_{imjnk}, $m \in (0, M_{ij}]$, $n \in (0, M_{jk}]$. Construct the connect matrix by applying:

$$C'_{imjnk} \Rightarrow L'_{imj} \wedge L'_{jnk} \wedge i \neq j \wedge j \neq k \wedge k \neq i \wedge ((n > m) \vee (n = m \wedge C_{ijk}))$$

Cycle Free There are no n-cycles for $n = 1, 2$ by virtue of

$$C'_{imjnk} \Rightarrow i \neq j \wedge j \neq k \wedge k \neq i$$

Expanding an n-cycle, $n > 2$, in (L', C') one readily obtains

$$\mathcal{P}^{(L',C')}_{i_0 m_0 i_1 \rightarrow i_0 m_0 i_1} \Rightarrow (L_{i_0 i_1} \wedge \ldots L_{i_n i_0}) \wedge (\mathcal{P}^{(L,C)}_{i_0 i_1 \rightarrow i_0 i_1} \vee (m_0 < m_1 \wedge \ldots m_n < m_0))$$

which cannot be true because (L, C) is acyclic.

Fully Connected The construction of C_{imjnk} yields

$$C_{ijk} \Rightarrow C_{imjnk} \forall i, j, k, m, n$$

It follows that there is a route from every processor to every other using links with constant component index and the network is fully connected. Notice that where the natural path between a pair of processors contains $l \leq M$ points at which C_{ijk} is false then there is a route covering that path.

Routing We associate a network element with each component L_{jnk}. Each element will accept the first unit of a message from an element at any adjacent component L_{imj} and any source at P_j by performing a conditional input from all such elements and sources.

The rule for determining the set of elements waited for by L_{imj} in determining the next step of the directed routing function for a message α becomes all L_{jnk} satisfying

$$(\neg(\neg\mathcal{P}^{(L,C)}_{jnk \rightarrow k' n' d(\alpha)} \forall k', n') \wedge (C_{imjnk} \vee w(\alpha) \leq u).$$

A source at processor P_j will similarly establish the first step of a message by waiting for all elements L_{jmk} satisfying

$$\neg(\neg\mathcal{P}^{(L,C)}_{jnk \rightarrow k' n' d(\alpha)} \forall k', n')$$

As in the case of simple graphs the through routing rule permits messages for which $w(\alpha) \leq u$ to loop. A restriction may be applied containing at least one communication link for which C_{imjnk} is true.

Dependency Analysis

Colour the graph (P, L) such that each communication link has a single colour and each processor is allowed to have as many colours as there are links attached to it.

We refer to a graph comprising all links and processors having the same colour as a *colour graph* and denote the m^{th} colour graph by (P^m, L^m).

We say that colour graph m is *below* colour graph n if (P^m, L^m) and (P^n, L^n) satisfy the predicate $B_{m \to n}$ defined by:

$$B_{m \to n} \Rightarrow \neg(\neg(P_i^m \wedge P_i^n \wedge m < n)\, \forall\, i) \vee \neg(\neg(P_i^m \wedge P_i^l \wedge m < l \wedge B_{l \to n})\, \forall\, i, l)$$

This means that $m < n$, *and* either colour graph m and colour graph n share a processor *or* there is a way from colour graph m to colour graph n, via at least one intermediate colour graph l such that $m < l < n$, using a common processor whenever moving from one colour graph to a higher numbered colour graph.

Colouring Colour (P, L) to generate colour graphs (P^m, L^m), $m \in (0, M]$ such that the set of colour graphs satisifies G1, G2;

Axiom G1: $\{(P^m, L^m)\}$ is *complete* in the space of links, i.e. the union of all colour graphs contains all links:

$$L_{ij} \Rightarrow \neg(\neg L_{ij}^m\, \forall\, m)\, \forall\, i, j \ldots \text{complete}$$

$\{(P^m, L^m)\}$ is *orthgonal* in the space of links, i.e. each link has only one colour:

$$L_{ij}^m \Rightarrow (\neg L_{ij}^n \vee m = n\, \forall\, m, n)\, \forall\, i, j \ldots \text{orthogonal}$$

Axiom G2: $\{(P^m, L^m)\}$ is *complete* in the space of processors, i.e. the union of all colour graphs contains all processors:

$$P_i \Rightarrow \neg(\neg P_i^m \ \forall m)\ \forall i \ldots \text{complete}$$

$\{(P^m, L^m)\}$ is *overlapped* in the space of processors, i.e. for every ordered processor pair i, j which do not have a common colour i has a colour m and j has a colour n and colour graph m is below colour graph n:

$$(P_i^m \wedge \neg P_j^m)\ \forall m \Rightarrow \neg(\neg(P_i^l \wedge B_{l \to m} \wedge P_j^m)\ \forall l, m)$$

and each subgraph satisfies G3, G4;

Axiom G3: (P^m, L^m) is conjoint.

Axiom G4: (P^m, L^m) is loosely acyclic.

Satisfaction If we construct C by allowing connects only between links in the same subgraph then by virtue of G4 (L, C) is acyclic. We can allow connects between different colour graphs by applying partial linear ordering ideas; in this case we also allow a connect to a link in any higher numbered subgraph. We therefore construct the connect matrix according to C1:

C1: $C_{ijk} \Rightarrow \neg(\neg(L_{ij}^m \wedge L_{jk}^n \wedge m \leq n)\ \forall m, n) \wedge i \neq j \wedge j \neq k \wedge k \neq i)\ \forall i, j, k.$

Cycle Free We can easily reason that (L, C) is acyclic; each colour (P^m, L^m) is acyclic and the linear ordering of colours ensures that there are no cycles in the transitions between colours. A more formal, and tedious, treatment is now given.

(L, C) contains no n-cycles, $n = 1, 2$, on account of

$$C_{ijk} \Rightarrow i \neq j \wedge j \neq k \wedge k \neq i$$

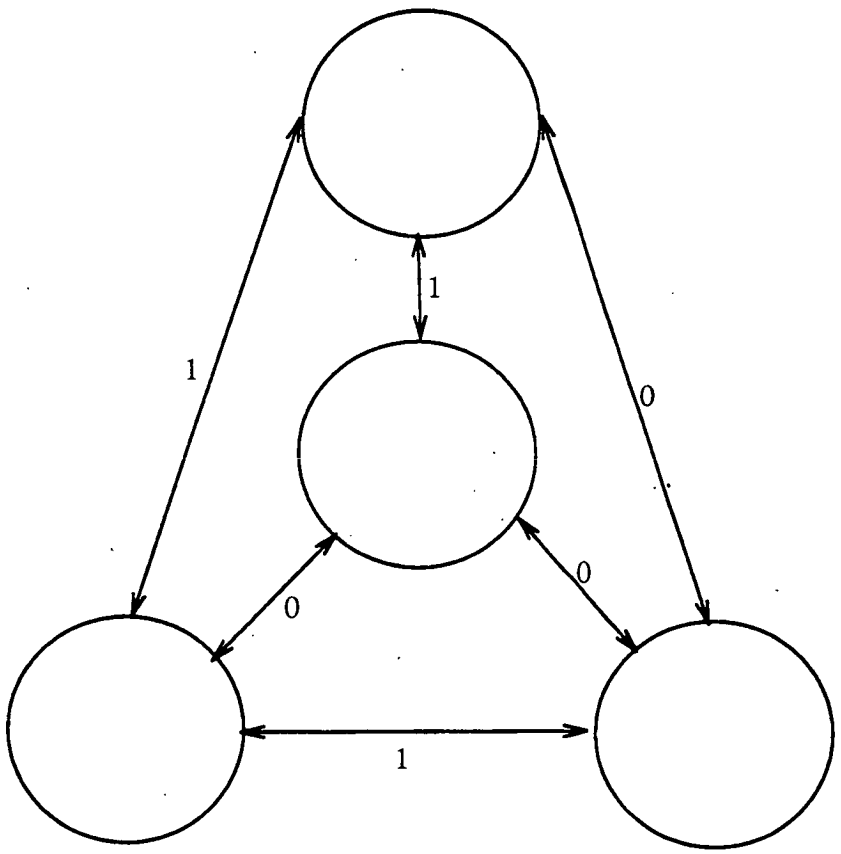

Figure 2.6: Valid Colouring of the Tetrahedron

Expanding the path for an n-cycle, $n > 2$ then using C1, G1 then G3 one obtains

$$\begin{aligned}
\mathcal{P}^{(L,C)}_{i_n i_0 \to i_n i_0} \Rightarrow\ & \neg(\neg(L_{i_n i_0} \wedge C_{i_n i_0 i_1} \wedge L_{i_0 i_1} \wedge \ldots \\
& \ldots \wedge L_{i_{n-1} i_n} \wedge C_{i_{n-1} i_n i_0} \wedge L_{i_n i_0}) \\
& \qquad \forall i_1, \ldots, i_{n-1}) \\
\Rightarrow\ & \neg((\neg(\neg(L^{m_n}_{i_n i_0} \wedge L^{m_0}_{i_0 i_1} \wedge m_n \leq m_0)\, \forall m_n, m_0) \wedge \\
& i_n \neq i_0 \wedge i_0 \neq i_1 \wedge i_0 \neq i_n \wedge \ldots \\
& \ldots \wedge \neg(\neg(L^{m_{n-1}}_{i_{n-1} i_n} \wedge L^{m_n}_{i_n i_0} \wedge m_{n-1} \leq m_n)\, \forall m_{n-1}, m_n) \wedge \\
& i_{n-1} \neq i_n \wedge i_n \neq i_0 \wedge i_0 \neq i_{n-1}) \\
& \qquad \forall i_1, \ldots, i_{n-1}) \\
\Rightarrow\ & \neg(\neg(i_n \neq i_0 \wedge L^m_{i_n i_0} \wedge i_0 \neq i_1 \wedge L^m_{i_0 i_1} \wedge i_1 \neq i_n \wedge \ldots \\
& \ldots \wedge i_{n-1} \neq i_n \wedge L^m_{i_{n-1} i_n} \wedge i_n \neq i_0 \wedge L^m_{i_n i_0} \wedge i_0 \neq i_{n-1}) \\
& \qquad \forall i_1, \ldots, i_{n-1},\ m) \\
\Rightarrow\ & \neg \mathcal{P}^{(L,C)}_{i_n i_0 \to i_n i_0}
\end{aligned}$$

Fully Connected By definition there is a route from every processor to each of its neighbours.

C1 has the property that every connect between links of the same colour is asserted; since every colour subgraph was chosen to be conjoint by G3 then there is a route from every processor to every processor having the same colour.

C1 has the property that every connect between a link in subgraph m and a link in $n > m$ is asserted. The overlap condition was chosen such that if P_i and P_j are not in the same colour graph then there is a path through common processors of ascending colour labels from P_i to P_j. Since there is always a route between processors having the same colour, and connects to higher colour labels are asserted by C1, there is a route to every processor not having the same colour.

Finally, since the union of colour graphs comprises all processors there is a route from every processor to every other processor.

Improvement We can improve the quality of (P, L, C) by noticing that there may be a number of C_{ijk} which are false as a result of C1 but if asserted would not

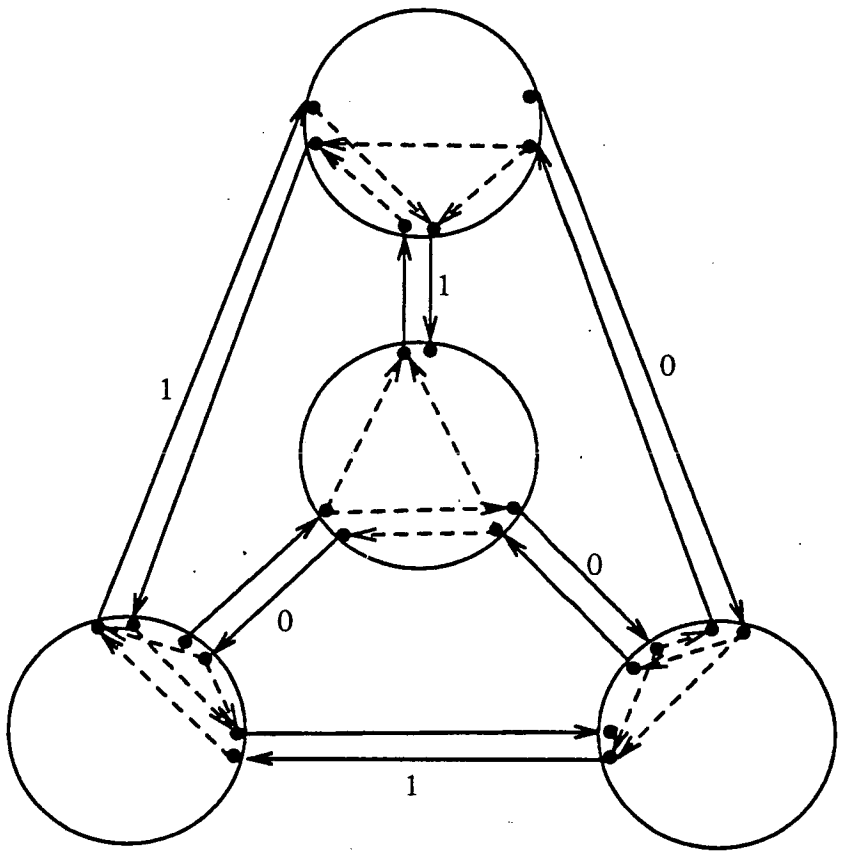

Figure 2.7: Satisfied Connects of the Tetrahedron

generate a cycle in (L, C). This occurs, for example, when there is a point at which we have the opportunity to descend the set of colour graphs and do not thereafter have the opportunity to ascend the set. We improve each C by applying C2.

C2: $C_{ijk} \Rightarrow \neg \mathcal{P}^{(L,C)}_{jk \rightarrow ij} \wedge L_{ij} \wedge L_{ji} \wedge k \neq i \; \forall i, j, k.$ [1]

Correctness: It is trivially true that the interconnection network remains cycle free and fully connected.

By definition C2 cannot generate a cycle in (L, C), therefore (L, C) remains acyclic. Because (L, C) is acyclic it is not possible for C2 to retract any C_{ijk} and the network must remain fully connected. In practice we can exploit this fact by only testing those connects which are false after C1.

Existence We have not yet shown that it is possible to find a colouring satisfying G1 through to G4; in fact there is always a set of solutions for a conjoint undirected graph. We shall call these the *trivial* solutions since the set of colour graphs satisfy the colouring in a trivial sense.

1. We can choose all colour graphs to be undirected trees.
   This satisfies both G3 and G4.
2. We can construct a spanning tree in any conjoint undirected graph.
   This satisfies G2.
3. We can repeatedly construct trees with unused links, until all links are used by some tree.
   This satisfies G1.

The algorithm for generating a trivial solution, shown in Figure 2.8, is very simple.

[1] This predicate is schematic in that it is not meaningful to simultaneously apply it to all of C_{ijk}; it should therefore be understood that we test and conditionally assert each of C_{ijk} in a sequential manner.

```
Set all link labels to NoLabel
Set TreeNumber to 0
Choose Root as an arbitrary processor
WHILE Root is found
  Label with TreeNumber the links of the largest tree having processor
                Root at its root and containing only unlabelled links
  Increment TreeNumber
  Find Root with at least one link labelled NoLabel
```

Figure 2.8: Algorithm for Trivial Solution of Colouring

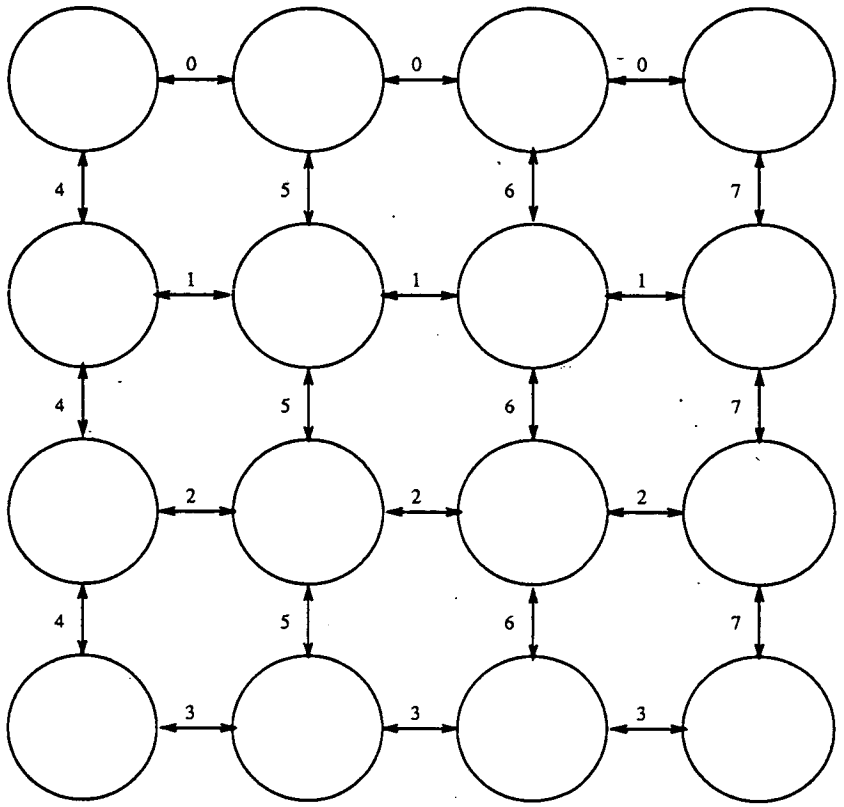

Figure 2.9: Non-trivial Colouring of a Grid

Of course we can expect to find non-trivial solutions to the colouring problem. Indeed the grid routing function discussed above is obtained by considering the non-trivial colouring shown in Figure 2.9.

Evaluation Routing along shortest, natural, paths is incompatible with acyclic routing in general topologies if a multilink approach is not used. We have therefore evaluated the trivial solutions of the colouring problem in a number of, no more than four valent, topologies including those discussed above.

In these calculations we have determined the maximum message path length D and average μ for messages α having $w(\alpha) > u$ and compared these with the same quantities in the natural paths. We denote the acyclic routes without improvement by a single prime and those with improvement with a double prime.

Trees: The question of evaluating the trivial solution algorithm within tree graphs is null since the spanning tree will be the complete graph and natural message paths are always used.

Grids: The grid routing function discussed above allows one to route messages between any pair of processors by choosing one of the shortest natural paths. We should hope that the method is capable of reproducing this result and we find that trivial solutions have this property. This result is expected since we can easily envisage a trivial colouring which permits natural routes in this topology; see Figure 2.10.

Rings: We should expect the ring topology to appear as the worst case for these quantities. In Figure 2.11 we see that in each case the trivial solution yields approximately twice the natural value. The improvement calculation cannot make any difference to these routes since the only connects "broken" are the two which remove the cycles running in each direction around the ring. Utilisation of the multilink, approach with two communication links mapped onto each physical link, according to the routing rules above yields the natural values of D and μ as we should expect.

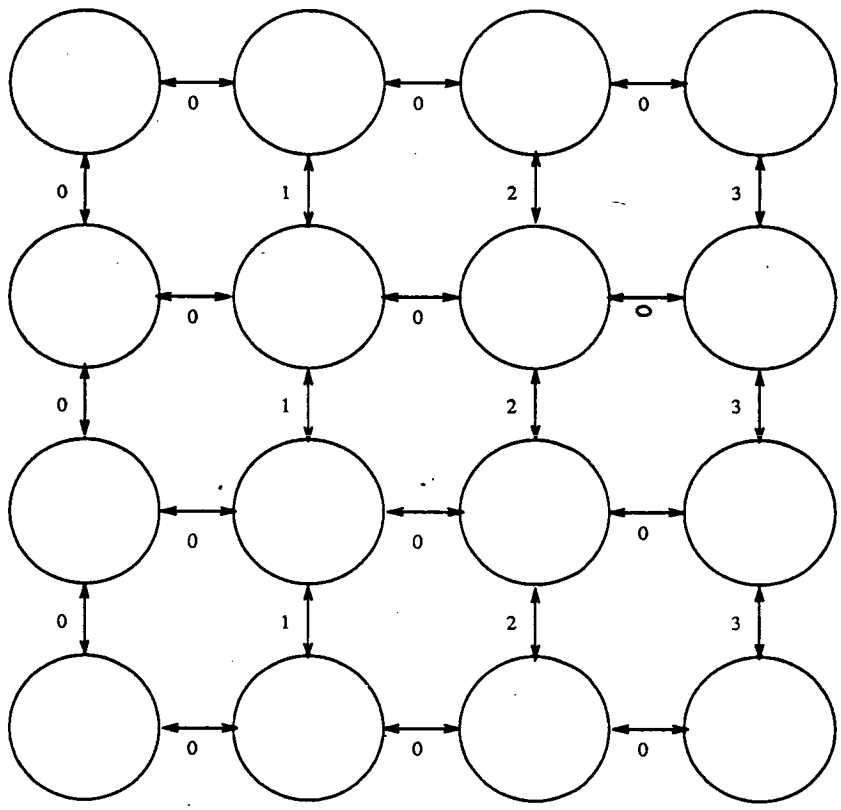

Figure 2.10: Trivial Colouring of a Grid

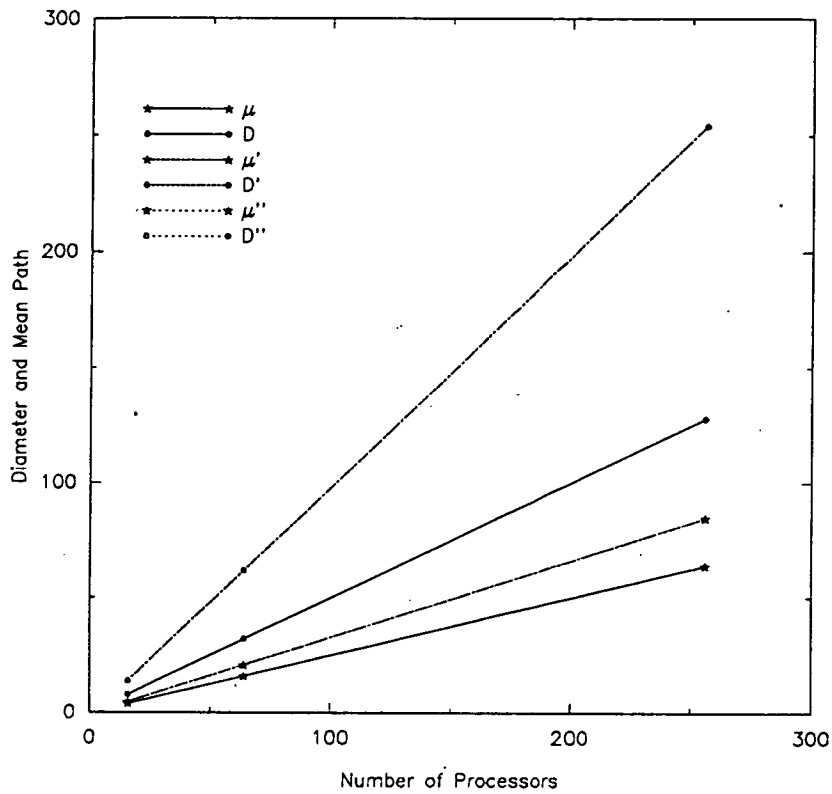

Figure 2.11: Evaluation of Trivial Colouring of Rings

General: The torus, a grid with wrap around links, is an interesting topology for many scientific problems where a natural cubic symmetry exists. There is no shortest route solution for general four valent tori; an exception is the familiar $d = 4$ hypercube which maps onto a 4×4 torus. It is therefore interesting to investigate the trivial solutions in these graphs and we present the results of these calculations in Figure 2.12.

The trivial colouring yields shortest message paths directly for the 4×4 torus. This is not surprising given that shortest paths are obtained for grids. In the larger tori the average message path length is extended by approximately 10–15% and is decreased very little by application of improvement. The increase in diameter is similar to that observed in the simple rings and improvement had no effect on this metric. The multilink approach, as in the simple rings, yields the natural of values of D and μ for all tori.

Where there is no locality which can be exploited in a problem then a useful criterion

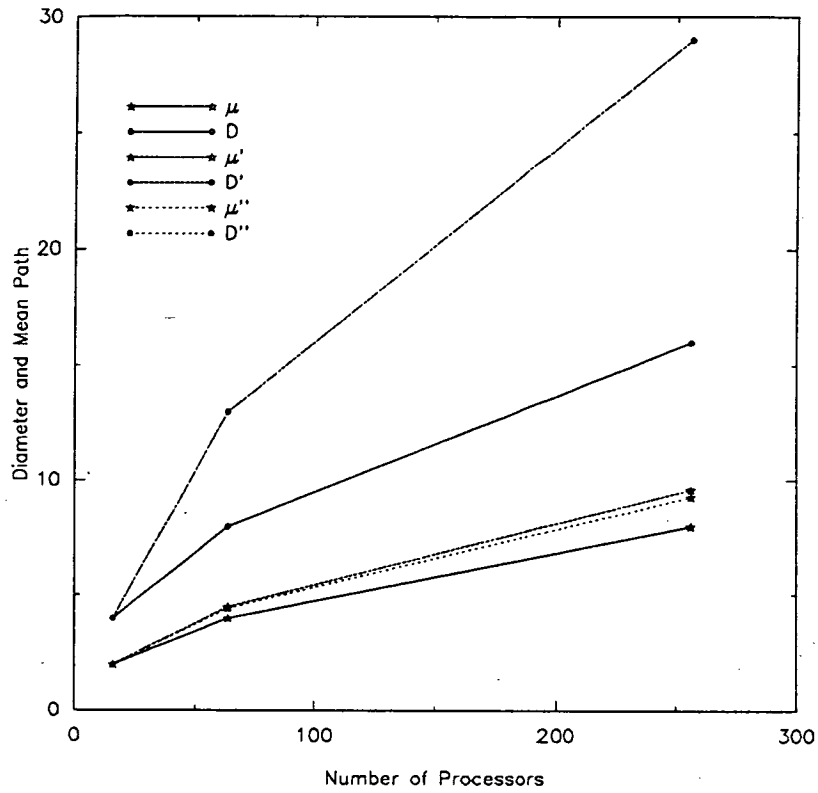

Figure 2.12: Evaluation of Trivial Colouring of Tori

for choosing a topology is that each processor should be as close to every other processor as possible. In [PNRC90] it was shown that irregular, even random, graphs approach optimality in this respect. We have chosen to perform calculations for random hamiltonian graphs, i.e. a ring with the remaining pair of links at each processor connected together randomly, which we generated with the constraints that there be no self or multiplet links.

We should anticipate that the effect on D and μ will be larger in dense graphs than in the tori. This arises because there are fewer instances of shortest routes between processor pairs, due to the larger girth of the graph, and thus a larger number of routes must be extended in order to eliminate cycles. These observations are borne out by the calculations shown in Figure 2.13. The increase in μ for a graph of $P = 256$ is approximately 25% which we consider to be acceptable considering the observation that in order to obtain the natural values of D and μ in a multilink approach it was necessary to increase the number of multilinks as P increased.

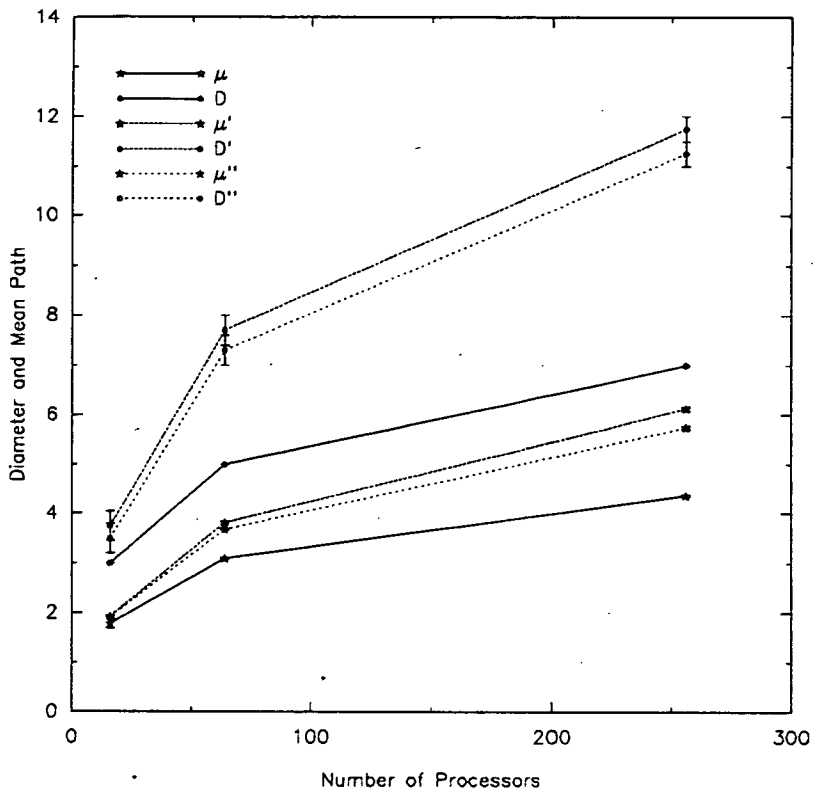

Figure 2.13: Evaluation of Trivial Colouring of Random Graphs

2.5 Summary

In this chapter we have discussed a number of issues pertaining to the design of communication systems for reconfigurable multiprocessor machines. The division of such systems into two logical layers, network and transport, has allowed us to describe the largely separate problems of message through routing and the provision of an application interface.

The application interface, provided by the transport layer, was discussed at some length and a concrete example provided. The guiding principles in arriving at this example were that the facilities offered should satisfy the following criteria.

- The procedural interface should be intuitive enough for the interested scientific programmer to write parallel programs with a minimum of specialist training.

- The facilities offered should be sufficiently powerful for the systems programmer to write sophisticated utilities which provide some level of abstraction from the message passing architecture.
- It should be possible to efficiently implement the interface on a large variety of processors without building special purpose hardware.

Issues involved in the network layer were described extensively. The variety of message switching models was discussed and a number of these were expressed as connectionless. The problem of deadlock avoidance within this model of switching was considered and the application of acyclic routing to this model brought together supposedly separate peices of work by other groups. We went on to give a specific methodology for construction of an acyclic interconnection network expressed as a graph colouring problem. The direct utilisation of acyclic routing is not compatible with shortest path routing in general graphs and we evaluated the results of a very simple instance of the method described.

In Chapter 3 we go on to describe the implementation of a communication system for networks of INMOS transputers which exploits the ideas of this chapter. This system uses a simplification of the application interface applicable to the assumed configuration environment. A restriction of the connectionless switching model is employed, the simple packet switching regime, and an adaption of the acyclic deadlock avoidance technique is implemented.

Chapter 3

Implementation

3.1 Introduction

In this chapter we describe the algorithms and techniques used in a software implementation of a communication system for transputer based machines called TINY. The development environment was the MEiKO Computing Surface although the system has now been used on a number of other transputer machines. The design of the system exploits ideas discussed in the previous chapter, with restrictions and specialisations applicable to the target hardware and configuration environment. It would not be appropriate to present here a complete and accurate description of the implementation of TINY. We choose a simplified subset of the functionality which captures the key algorithms and techniques developed in the system.

The fact that the target machine is reconfigurable, and in principle can be configured arbitrarily, indicates that we require a system which correctly routes messages in any topology. Where the topology chosen by the user is rather irregular, as has been shown to be of general utility [PNRC90], and moreover where the topology obtained is not identical to that requested, due to shortcomings in the configuration software or hardware of the machine, it is most convenient if the communication system determines the topology of the machine within which it is expected to route. The system implemented performs such a calculation which is described in Section 3.3.

Given a description of the interconnection network topology we must address the problem of how to formulate a routing function for that topology, which should be free from deadlock. Since the topology is arbitrary we store the routing function in a set of routing tables; the calculation of these tables for sequential, quasiadaptive and broadcast routing functions is described. The methods for deadlock avoidance are an example of those discussed in the previous chapter. The calculation of the routing tables is described in Section 3.4.

The system shares CPU and memory with the user; the target machine typically consists of T800 processors with upwards of 1 Mbyte of memory. The system should therefore use as little CPU as possible and should not require excessive amounts of memory. In order to minimise the CPU consumption compromises between functionality and performance must be made. A guiding principle was the observation that the granularity, and therefore parallel speedup, which can be obtained in applications is bounded by the notional zero length message latency. The methods used in implementing the message passing kernel are described in Section 3.5.

3.2 Overview

In this section we present an overview of the implementation of TINY and restrictions we shall make in order to simplify discussion in the following sections.

It was anticipated that applications would be written in occam, C or FORTRAN within an occam harness using the Transputer Development System, or one of its proprietary reincarnations[1]. In this system the programmer explicitly specifies all details of the execution environment and must insert code into the harness which instantiates the the message routing system.

The message routing system, as opposed to the application procedural interface, is

[1] This system has subsequently been ported to the 3L Parallel Languages configuration environment where it would arguably be preferable it utilise the worm algorithm

referred to as the *kernel.* The kernel is partitioned into three components; *configuration exploration*, *routing generation* and *message passing.*

Configuration Exploration

The main purpose of configuration exploration is to determine a correct description of the topology within which the system is running in order that a routing function can be calculated. Other services such as distributing information about the configuration of application processes are also performed within this component.

In a system with P processors we label these processors by natural numbers in $(0, P]$. We must determine, for every link on every processor, the label of the processor connected by that link, and this is performed by sending and receiving trial messages over links. This information can be collated at every processor or at a single processor.

The labelling of processors can either be specified within the configuration environment or determined by the topology exploration algorithm itself. We have considered a *worm* method in which exploration begins at a single processor and controls the allocation of labels to processors, and a *flood* algorithm in which exploration begins at all processors and processor labelling is performed within the configuration environment.

In practice we used the flood algorithm. The occam configuration code requires that the user statically declare the number of processors in use and assign unique numeric labels to each processor; given P processors it is not inconvenient both to choose labels in $(0, P]$ and to pass this information through to the routing system as formal parameters. It is unfortunate that the algorithm simply will not terminate if the processor labelling is in error; however it is not difficult for the programmer to ensure that this does not occur.

It is possible for the programmer to perform arbitrary computations at each processor before invoking the routing system; it follows that there can be arbitrary

time differences between the commencement of the algorithm on one processor and another. The same considerations apply when the programmer terminates the routing system and later invokes it again, possibly to reconfigure the machine. Under these conditions it is not possible to specify a time out as required by the worm algorithm without a knowledge of these delays, which may well be undetermined.

The flood algorithm requires more memory at each processor than the worm algorithm. Given attainable transputer arrays of up to 256 processors, each with ≥ 1 Mbyte of memory, this was not seen to be a serious consideration.

Routing Generation

The programmer is allowed to configure the routing function generator to calculate straighforward shortest routes, the *cyclic* routing functions, or a deadlock free set of routes, the *acyclic* routing functions. The topology exploration component deposits a copy of the configuration description at every processor and it is convenient to allow each processor to calculate the portion of the routing function it requires. The routing function data at each processor is stored in a set of routing tables which are carried through to the message passing component of the kernel.

The cyclic routing function is provided because there are a large number of problems where the possibility of deadlock within the network layer simply does not arise, due to the communication patterns of the application (of course this depends also on the configuration) or simply because there is sufficient buffering within the network to "accidentally" avoid deadlock. The programmer is also allowed to configure the number, and size, of through routing buffers available to each instance of the kernel. In Chapter 3 we shall see an example of a program having a simple communication pattern which can safely use the cycle routing function.

The methods used to obtain the acyclic routing functions are a concrete application of those discussed in the previous chapter. The deadlock system utilises a single message plane; this was chosen for a number of reasons:

- The system was originally implemented without solving the network deadlock problem and network deadlock was infrequently observed despite substantial use of the sytem. Choosing a single message plane allowed the message routing processes to be used with a simple change of routing tables.
- The management of a multiple plane approach requires considerably more CPU than the single plane approach. Minimisation of the CPU impact in through routing events, and demonstration that this could be achieved in sub $40\mu s$ timescales, was a primary objective of the development exercise.
- We demand that the number of message buffers at each processor required to implement the deadlock free routing function be independent of the size of the processor array. By restricting ourselves to a solutions with a single message buffer at each link we meet that demand.

Message Passing

Within the anticipated configuration environment the message routing system and system services are viewed as part of the application program as opposed to distinct entities. This fact renders the group mechanism described in the previous chapter, which was largely designed to shield the programmer from details of the execution environment, less relevant. Only the process labelling, in abstraction from placement, and message type labelling features are provided. The programmer is required to allocate a unique numeric process identifier within the interval $(0, N]$ to each of the N application processes.

A brief inspection of transputer applications using a communication system with a related interface [NW88] revealed that programmers had generally used only a small number of message types. We can exploit this observation by explicitly assigning lightweight processes to the maintenance of separate message queues for each type.

The configuration environment provides a channel between each of these transport processes and the application. A typed message send is implemented by indexing into an array of channels to transport source processes and communicating with

the selected process. Similarly a typed message receive is implemented by indexing into an array of channels to transport sink processes and communicating with the selected output process.

Message receive with an unspecified type would require either a search through message queues or a two dimensional queueing system within transport. This has not been implemented and such a message receive is not provided. Selection on the identifier of the source application task would also require more sophisticated, and costly, queueing techniques and neither was this provided.

TINY allows the programmer to configure these transport processes to provide asynchronous or synchronous messages, by passing an array of message type descriptions to the kernel. It is also possible to create static broadcast groups by declaring that messages of a particular type are required only by application processes within the broadcast group.

These features are not included in the transport layer of the simplified system we describe; they add little to the discussion of techniques employed but complicate the description of the implementation considerably. A simple FORTRAN transport layer interface for the primitive system we describe in this section is given in Figure 3.1.

The network layer uses a simple packet switching technique and upper bounds the size of messages which can be delivered, although the programmer is allowed to configure this bound. This decision was made as the system was intended to optimise small messages and minimize CPU usage during routing. The ability of the transputer to perform input and output on all of its links concurrently with execution of the main processor is an important feature of the design. We allocate a pair of processes to each transputer link, one of which handles message input and the other performing message output.

The passage of each data block through an intermediate processor consumes the CPU required to initialise two link communications and to synchronise a pair of input and output processes. Experience suggests that if these processes have workspaces in external memory, with a cycle time of approximately 200ns, then this will amount to around 10 μs of CPU usage. The byte transfer time on the development hardware

```
FUNCTION SEND(TYPE, DESTINATION, DATA, LENGTH)
INTEGER TYPE, DESTINATION, DATA(*), LENGTH
```

This function sends a directed message held in DATA of LENGTH bytes to the task with identifier DESTINATION, or a broadcast message if DESTINATION has the value NONE, of the type TYPE. The length of the message sent, or NONE if DESTINATION was invalid, is returned. The message sent will be smaller than the LENGTH when LENGTH is larger than a network buffer.

```
FUNCTION SSEQ(TYPE, DESTINATION, DATA, LENGTH)
INTEGER TYPE, DESTINATION, DATA(*), LENGTH
```

This function sends a quasiadaptive directed message held in DATA of LENGTH bytes to the task with identifier DESTINATION, or a broadcast message if DESTINATION has the value NONE, of the type TYPE. The length of the message sent, or NONE if DESTINATION was invalid, is returned. The message sent will be smaller than the LENGTH when LENGTH is larger than a network buffer.

```
FUNCTION RECV(TYPE, SOURCE, DATA, LENGTH)
INTEGER TYPE, SOURCE, DATA(*), LENGTH
```

This function receives a message into DATA, of the type TYPE, storing identifier of the source task in SOURCE. The length of the message received, in bytes, is returned. When the message sent was larger than LENGTH then the first LENGTH bytes are received and the remainder discarded; this can be detected from the return value.

Figure 3.1: A Simple Transport Layer Interface

running links at 20Mbits^{-1} is approximately 0.7μs indicating that when all links are being used a block size of less than approximately 50 bytes will saturate the CPU.

3.3 Topology Exploration

In this section we describe the topology exploration algorithms which we have considered, the *flood* and *worm*. We are able to assume that the machine is connected since the routing system and application codes will have been succesfully booted in which case a boot path, a tree, exists.

Once cannot discount the possibility that there will be certain links which are required for other uses; a familiar example is the link from the first processor to the host machine which often runs a terminal served file system protocol. The topology exploration algorithm must therefore accept as input an indication of which transputer links may be used by the communication system and which have been reserved.

One cannot either discount the possibility that certain links are not connected to any other transputer; this can occur due to machine configuration failure or simply because the configuration does not utilise all links. This must be detected; any trial messages on such a link will fail and the implementation must be able to recover from this.

Worm

The configuration environment distinguishes a single processor, which we shall call the *root* from all other processors, which we shall call the *nodes*. It does not specify the number of nodes or a labelling scheme for the nodes.

The idea behind this algorithm is that the root can allocate a processor label to itself, say 0, and probe its own links allocating labels to any processors discovered.

```
If this is not the root
  Execute Probee Algorithm

ELSE
 Set own identifier to 0, number of processors to 1

Execute Prober Algorithm

IF this is not the root
  Output number of processors to parent
```

Figure 3.2: Worm Algorithm

Of course the nodes being probed must be running a complementary part of the algorithm which responds to the probing messages issued by root. When a processor is discovered we call the discovered processor the *child* of the discoverer and likewise the discoverer is the *parent* of the discovered; these parent-child relationships will form a tree.

After a processor, either root or a node, has probed each of its free links and labelled its children it arranges for each child, in turn, to probe its link and label its children. The processor passes to each child, in turn, the number of processors thus far discovered and waits for the child to return this number, possibly updated. The number of processors is finally returned to the parent, excepting at the root where the completion of the last child terminates the worm. When the worm terminates we would like each processor to have recorded which link leads to its parent, which links lead to its children, and the labels of its neighbouring processors. We will additionally need to know the number of the link at the neighbouring processor on which it is connected to ourself. The worm algorithm is summarized in Figure 3.2.

After this information has been generated we gather it at the root processor for analysis and possible distribution to the nodes. This is easy to achieve since we have already defined a tree within the graph. Each node forwards its own information to its parent then, in turn, receives information from its children, again forwarding

```
WHILE number of processors not recorded
  Wait for a free link to become ready
  Input a number from ready link

  IF this is first number input
    Record number as own identifier
    Record link as parent link

  ELSE IF link was parent link
    Record number as number of processors

  IF number of processors not recorded
    Input and record neighbour identifier and link number from link
    Output own processor identifier and link number to link
```

Figure 3.3: Probee Algorithm

to the parent, until some *NoMoreInformation* token is received; finally it forwards *NoMoreInformation* to the parent. The root simply receives information on each link until *NoMoreInformation* has been received on all links. If we wish to send some information from the root to the nodes then we simply pass the information down this tree, sending a replica to each child.

The probing of links is central to this method and merits further consideration. It will be simplest to introduce the probee in the first instance; note that root is never a probee. The probee is described in Figure 3.3. The prober sends messages to the probee and has to recover from the possibility that the first message will fail due to a disconnected link; the first message is output with some associated timeout. The prober is described in Figure 3.4.

We point out here that the above implementation of the worm algorithm is slightly flawed. The problem arises when an output is aborted due to timeout while communication is in progress; the process on the neighbouring processor can never complete its communication and the algorithm does not terminate.

```
FOR all links
  IF link is free and not probed by self or neighbour
    Output number of processors to link with timeout

    IF output succeeded
      Output own processor identifier and link number to link
      Input and record neighbour identifier and link number from link

      IF neighbour identifier equals number of processors
        Record that link is a child link
        Increment number of processors

    ELSE
      Record link as disconnected

FOR all links
  IF child which has not executed prober
    Output number of processors to child
    Input  updated number of processors
```

Figure 3.4: Prober Algorithm

We can arrange for the algorithm to terminate and produce a consistent, although incomplete, description of the topology by replacing `Input number from chosen link` in the probee algorithm by an input with timeout; we can sensibly upper bound the time it will take to transfer the number across a link which is already waiting to transfer. If the input failed due to timeout then we know that the prober must have aborted the output; we can simply record the link as not connected which is consistent with the prober. This fine detail was not shown for sake of clarity.

Flood

The configuration environment specifies at each processor the total number of processors P and a unique label in $(1, P]$. The algorithm fails if P is not the same at every processor or there are duplicate, or missing, processor labels.

The idea behind this algorithm is that if each processor outputs its own identifier on each free link and listens for identifiers of other processors on all free links, recording the arrival and forwarding on all free links if this was the first arrival of the identifier, then each processor must receive, and send, the identifier of each processor, including itself, exactly once on each connected link.

In order to implement this algorithm one requires a buffer process to handle output at each link whilst a control process handles input on all links, deals with arriving identifiers and passes identifiers to the buffers for forwarding when requested. The exchange of link numbers can be achieved by arranging for the buffers to output initially the number of the link which they are using; the buffer process is shown in Figure 3.5. The control process manages a record of up to P pending outputs for each buffer process as shown in Figure 3.6.

This algorithm terminates when for each free link communications have either completed, P identifiers plus one link number have been output by the buffer process and P identifiers plus one link number have been input by the control process, or are not yet commenced, the link number has not been output by the buffer process or a link number has not been received by the control process.

```
Output buffer link number
Output token on buffer request channel
Input data item on buffer data channel
WHILE data item is not null item
  Output token on buffer link
  Output token on buffer request channel
  Input data item on buffer data channel
```

Figure 3.5: Buffer Process

This evaluation is not quite correct since it is possible for a buffer process to have completed an output while the condition has evaluated to false. In order to handle this situation we attempt to abort the link number output of the buffer processes which have not yet made a request to their queue of pending messages. If that output had succeeded then we continue flooding since the link has commenced activity; if on the other hand the output was aborted then we restart and terminate the process immediately. This termination condition is shown in Figure 3.7.

At this stage each processor knows only the processor identifiers and link numbers of its immediate neighbours and it is necessary for this information to be distributed amongst all processors. It is not difficult to see that the above algorithm, with slight modification, can be used to achieve this.

In this case the connected links are used as opposed to the free links, some of which may not be connected, and the initial exchange of a link number is omitted. Each communication of a processor identifier is accompanied by the neighbour processor identifiers and link numbers of the identified processor. The control process records these data items in some suitable data structure as they arrive on the first occasion and discards them thereafter. Of course any other information can also be distributed among the processors in this way.

Termination is considerably simpler in this case since it is known that none of the communications can fail. The algorithm completes when exactly one message from every processor has been output and input on every connected link.

```
Initialise empty buffer pending queues
Record no messages from any processor
Enqueue own identifier to buffer queues
Record message arrived from self
Set Finished to False
WHILE NOT Finished
  Wait for any free input link which has not completed communications
       or any buffer request where queue not empty to become ready
  IF link became ready
    IF first time this link became ready
      Input and record neighbour link number from link
      Input identifier from link
      Record identifier as neighbour identifier
    ELSE
      Input identifier from link

    IF first time this identifier received
      Enqueue identifier to buffer queues
      Record this identifier received

  ELSE
    Input token from buffer request channel
    Dequeue identifier from buffer queue
    Output identifier to buffer data channel

  Set running False if ready to terminate

Terminate buffer processes
```

Figure 3.6: Control Process

```
IF there is any link or buffer which has received some processor
                  identifiers but not all processor identifiers
  Set Finished False
ELSE
  Set Finished True

IF Finished is True
  FOR all link
    IF link is free and buffer process has received
                          no processor identifiers
      Reset buffer link
      IF buffer process was waiting on link
        Restart and Terminate buffer process
       ELSE
         Set Finished false
```

Figure 3.7: Flood Termination Condition

We again take the opportunity to point out that the implementation of the algorithm is flawed, in the same way as the implementation given of the worm algorithm. The solution is similar; the input of the link number within the control process, the first message on a link, might be aborted at the other end and should be input with an appropriate timeout. In the case of the input failing to complete within the period then the control process can recover from this and record the link as not connected, as has the control process which aborted the matching buffer process output. Again this was not shown for sake of clarity.

3.4 Routing Function

Routing table data indicates which way a message should be forwarded, whether this should be to another transputer link or to a transport if this is the destination processor. The two possibilities are handled homogeneously by introducing the concept of a *pseudolink*, which can be either a transputer link or a transport. If

a particular processor has l links and n local users tasks then there are $n + l$ pseudolinks at the processor; we might choose, for convenience, to allocate the first n pseudolink indices to the actual links and the remainder to transports.

The routing tables can be held in a processor indexed form or a task indexed form. If held in a processor indexed form then it is also necessary to maintain a task indexed table holding the processor and pseudolink number of each task in order; the accumulated size of these tables is nevertheless smaller than the task indexed routing table. At the time of development the largest transputer array attainable was of order 256 processor with each processor having upwards of 1 Mbyte of memory; we decided to use task indexed routing tables for convenience.

In either the cyclic or acyclic case the routing table is stored in the same manner. The header of the routing table segment contains $L + 1$ base indices, one for each network and one shared by all transports. Each of these indices points to the base of an index table within the routing segment. The directed entry for task i within an index table is held at word offset $2i$ from the index table base and the broadcast entry is held at offset $2i + 1$. The reason for interleaving the two tables will become apparent when the message passing kernel is described.

Each entry in the index table is a further index into the routing segment pointing to a row of pseudolink numbers, terminated by the null index -1. The entries in a directed table are the pseudolinks which may be used to forward the given message; the quasiadaptive function can use any of these links and the sequential function will always use the first of them. Where the pseudolink points to a transport then there will be a single entry. The entries in a broadcast table are the pseudo links which must be used to forward the given message in order for it to arrive at every transport. The extra level of indirection is used in order to handle the variation in routing entry lengths. The consequent table look-up is expensive where array indexing is used, however we anticipate transformation of the tables to an efficient pointer based form in the message passing component.

In the cyclic routing function each base index is identical; in order that transformations on the index table and its entries can be correctly performed by the message routing kernel a status word is held at offset -1 from the index table

base, indicating the condition of the table.

In the acyclic routing function the $L + 1$ base indices are distinct. The directed table is a $P \times P \times L \mapsto \{L\}$ function however there are some mappings which produce an empty set, i.e. there is no route from A to B if the message arrived at A on link l; in these cases the index is null. The broadcast table is a $P \times P \mapsto \{L\}$ function and for each source task there is a null index in all but one of the index tables.

3.4.1 Cyclic Routing

These tables are calculated by demanding that the directed table contains all shortest routes for each destination and the broadcast table contains a minimal spanning tree for each source. In order to perform these calculations we require an algorithm which generates the distances between pairs of processors; we exploit the fact that if X is the shortest path from A to B and X passes through C, then X includes the shortest path from C to B.

Distances

The interprocessor distances are calculated using a simple graph traversal algorithm. This method has much in common with the worm algorithm described in the topology exploration preamble except that in this case we use a breadth first search to ensure that shortest distances are obtained.

Directed

In order to determine the directed routing to processor d from processor s, $d \neq s$, we calculate the distances from s to all processors. We exploit the fact that in an undirected graph if X is a shortest path from A to B then $-X$ is a shortest path from B to A.

The minimum of these distances, over all neighbours of s, is evaluated and all links connecting to neighbours which are at this distance may be used. This is guaranteed to deliver messages correctly to d since at each step a message becomes closer to d than at the previous step.

The routing information is expanded over tasks located on d with possible reordering of the pseudolink numbers in different task entries.

Broadcast

In order to determine the broadcast routing for a source s through a processor d, $s \neq$d, we calculate the distances from s to all processors. This has already been performed in the calculation of the directed routing function.

The minimum distance from s of the neighbours to d is evaluated and an arbitrary decision is made that the broadcast message will be received by d on one of the links whose neighbour is at that distance. Each processor then swops the result of its decision with its neighbours and routes broadcast messages to all neighbours which elected to receive broadcast messages from it. This correctly routes messages from s to all other processors since the routing tables describe a spanning tree of the graph with s as its root.

The routing information is expanded over tasks located on s and the pseudolink numbers of tasks placed on d are appended to each entry.

3.4.2 Acyclic Routing

In order to calculate the acyclic routing functions we exploit the method given in the previous chapter, introducing a concrete implementation. The computational techniques used are closely related to the graph traversal performed in the distance calculation of the cyclic routing function.

```
/* initialise */
Create an empty queue of processor numbers
Add Root processor number to tail of queue
Set Parent of each processor to NONE
Set Parent of Root processor to Root

/* traverse */
WHILE queue not empty
  Remove processor number from head of queue
  LOOP over all links of processor
    IF link is unlabelled
      IF Parent of neighbour is not processor
        Label both directions of link with TreeNumber
        Set Parent of neighbour to processor
        Add neighbour processor number to tail of queue
```

Figure 3.8: Tree Labelling Algorithm

Connects

Colouring We shall use a trivial solution of the colouring problem given in the previous chapter. The algorithm to label a tree, required in Figure 2.8, is adapted to correctly handle multigraphs, as shown in Figure 3.8.

Satisfaction The graph colouring is performed and we now construct a connect matrix which satisfies C1. The rule for the connect from link l to link m at processor p is quite simple.

If the neighbour at link m is the same as the neighbour at link l then the connect is marked BAD; this handles multiplet links correctly when they have not been merged. Otherwise, if the label of link l is not less than the label of link m the connect is marked GOOD; C1 tells us that this connect cannot be in a cycle. If the connect has not already been marked then mark it UGLY; improvement will later mark this connect either GOOD or BAD.

Improvement We can apply improvement to the connects, restricting ourselves to the connects marked as UGLY, provided we have an algorithm for determining whether a particular connect is in a cycle containing itself and GOOD connects.

One method for doing this consists of determining a maximal tree in the (L, C) graph whose root is L_{jk}; if this tree does not include the link L_{ij} then the connect C_{ijk} is not in a cycle; this technique tests all the i simultaneously. The algorithm for determining distances used in the directed routing function performs this calculation.

Directed

We calculate the directed routing tables using the same ideas as those applied in calculating the cyclic directed table; in this case we simply calculate distances in the acyclic (L, C) graph.

The minimum distance to any link arriving at d from any link leaving s is then evaluated. In the case of messages originating at s we allow the message to be routed on any link which has the minimum distance. In the case of a message arriving at some link l we allow the message to be routed through any link m for which the connect from l to m is GOOD and m has the minimum distance. It can happen that there is no link on which such a message can be forwarded, consider the leaves of a tree; in this case the routing table entry is null and no messages for that destination will arrive on the link in question. This function correctly routes messages for the same reason as the cyclic routing function.

Again the routing obtained is expanded over the tasks located on processor d.

Broadcast

In the acyclic routes the broadcast function is still a mapping of $P \times P \mapsto L$. We have to search the graph again to determine the broadcast function for sources on a processor p since it is no longer the case that the route from B to A is the reverse of

the route from A to B. We can find a broadcast function because of the algorithm we have chosen for determining the graph colouring.

The search required for this function is very similar to the search performed in the simple procedure distances. The only modifications are that when the search finds a processor then the link on which that processor is found, the *parent* link, is recorded and in order for the search to propagate through that processor to a potential child the connect from the parent link to the potential child link must be GOOD. When a child is found then this fact is also recorded; the set of parent-child relations then defines the tree we require for the routing decision. The only network index table containing an entry which is not null is that corresponding to the parent. The transport index table entry is null except where the transport is local.

The broadcast routing thus obtained is similarly replicated over the tasks placed on the source processor and the pseudolink numbers of tasks placed on the local processor are appended to each entry.

3.4.3 Simulations

The program written to compare the cyclic and acyclic routing functions simulates a problem in which each processor sends a single message, of fixed length, to every other processor. We are interested in measuring from this the average message distance μ, the maximum message distance, D, the maximum number of messages passing through any link, or worst link load, L_l, and the maximum number of messages passing through any processor, or worst processor load, L_p. Since the T800 transputer has four links we limit our discussion to the results of simulations in fully connected four valent topologies.

The simulation program assumes that there is an essentially unlimited buffering capacity at each node; in actual fact the amout of buffering required in order to avoid deadlock in the case of cyclic routing functions. No account is taken of the time spent calculating the routing function or queueing messages to links etc; we

assume that any such time is small compared to the time taken to transfer the message data.

In the first instance the program queues a message send from each processor to every other processor in a random order; we assume that all messages are of the same length. Thereafter the program executes a number of time steps which advance messages toward their destinations until all messages are delivered or a deadlock is detected. A single time step consists of visiting each output in random order to determine whether a link transfer is ready to take place, and if so transferring the message to the relevant input, followed by visiting each input in random order to determine if a link transfer has taken place, and if so routing and queueing the message to the relevant output.

Rings The tree labelling algorithm for the trivial colouring algorithm given above is intended to handle multigraphs and we have evaluated the performance of the acyclic routing function in doubly linked rings. The results of these simulations are shown in Table 3.1.

We observe, as anticipated, that the values of D and μ are identical for both functions since the algorithm finds an acyclic routing function closely related to that shown in Figure 2.4. The worst link load is, in the limit of very large processor arrays, doubled by the acyclic routing function due to the fact that a small number of links are not used for through routing which forces message paths onto the corresponding parallel links.

General We have again performed calculations for square tori and random hamiltonian graphs; the results of these simulations are shown in Table 3.2 and Table 3.3.

We observe in these results that whereas the average distance and diameter are extended by less than a factor of two the worst link loading and worst processor loading become very much larger as the number of processors is increased; *hot spots* are generated. This arises due to the possibility of free message travel within the undirected tree colour graphs which ultimately places an unfair load on the links

P	μ	D	L_p	L_l
16	4	8	52.0 ± 0.4	18.0
64	16	32	965 ± 2	260 ± 1
256	28	64	16138 ± 1	4100 ± 1

P	μ'	D'	L'_p	L'_l
16	4	8	53.8 ± 0.2	31.0
64	16	32	979 ± 2	507 ± 5
256	28	64	16197 ± 2	8166 ± 2

Table 3.1: Evaluation of Acyclic Routing in Double Rings

P	μ	D	L_p	L_l
16	2	4	18.5 ± 0.2	10.8 ± 0.2
64	4	8	200.5 ± 0.8	71.0 ± 0.6
256	8	16	1806 ± 1	523 ± 1

P	μ'	D'	L'_p	L'_l
16	2	4	33.3 ± 0.5	18.3 ± 0.5
64	4.44	13	783 ± 2	330 ± 2
256	9.27	29	13636 ± 3	6684 ± 2

Table 3.2: Evaluation of Acyclic Routing in Square Tori

near the root of the tree.

This appears alarming, however many applications will be able to avoid placing heavy load on such hotspots by simple configuration changes. The performance of algorithms whose key communication structures are similar to those simulated will always be impeded; one such algorithm is the distributed multi-dimensional fast fourier transform which utilises the "all-to-all" data exchange when reordering the transform set.

P	μ	D	L_p	L_l
16	1.85 ± 0.01	3.25 ± 0.05	19.5 ± 1.2	12.5 ± 0.7
64	3.13 ± 0.02	5.75 ± 0.25	194 ± 7	83 ± 5
256	4.38 ± 0.01	7.0 ± 0.13	1110 ± 20	393 ± 4

P	μ'	D'	L'_p	L'_l
16	1.97 ± 0.02	4.3 ± 0.3	34 ± 2	19 ± 1
64	3.73 ± 0.03	8.3 ± 0.3	660 ± 40	240 ± 10
256	5.71 ± 0.01	11.0 ± 0.3	11700 ± 200	3640 ± 100

Table 3.3: Evaluation of Acyclic Routing in Random Hamiltonian Graphs

3.5 Message Passing

In this section we describe the techniques used in the implementation of the message passing kernel. We begin with with a very clear, and clean, occam prototype implementation which demonstrates the basics of the system. There are a number of performance problems associated with this prototype which we consider and for which we briefly explain a solution. These solutions are implemented in an improved prototype, written on this occasion in ANSI C on account of the necessary pointer handling and reliance on rich data structures.

The message passing kernel of TINY performs better than either of the prototypes described since it is implemented in highly optimised transputer assembly code; we did not think it appropriate to describe these optimisations here. We conclude the section with a discussion of the performance analysis of these systems and present a characterisation of the performance of TINY.

3.5.1 Simple Prototype

A very simple router can be written in clean occam providing a subset of the overall functionality; in particular the quasiadaptive routing function is omitted.

This system contains two network processes, which we shall call `input` and `output`, and two transport processes, which we shall call `source` and `sink`.

These processes are assigned to external (connecting with a neighbour processor or a local application process) channels thus:

input The input channel mapped onto a transputer link.

output The output channel mapped onto a transputer link.

source A channel used for message send by a local application process.

sink A channel used for message receive by a local application process.

A channel is provided from each input to each output and each sink. A channel is also provided from each source to each output and each sink of the same type index.

Each message will consist of a header accompanied by the message data. The header contains the message length, destination task identifier, source task identifier and the message type. A broadcast message will be indicated by the destination task identifier having the value `NONE`. The application will supply transport with the message destination, length and data at a source; transport will supply the message source, length and data to application at a destination.

Processes

Output and Sink The `output` and `sink` processes are the simplest since they are essentially just simple message multiplexors. These are shown in Figure 3.9 and Figure 3.10.

The alternation over inputs is the counterpart of the CI state in the previous chapter. Unfortunately the occam `ALT` construct does not satisfy the fairness assumptions made regarding conditional input. It should perhaps be replaced by a "round robin" alternation.

```
PROC output(CHAN OF ANY out, []BYTE buffer
            []CHAN OF ANY from.input,
            [][]CHAN OF ANY from.source)

  WHILE TRUE
    INT length, source, destination, type :
    SEQ
      ALT
        ALT i = 0 FOR SIZE from.input
          from.input[i] ? destination; source; type;
                          length :: buffer

        ALT i = 0 FOR SIZE from.source
          ALT j = 0 FOR SIZE from.source[i]
            from.source[i][j] ? destination; source; type;
                                length :: buffer

      out ! destination; source; type ; length :: buffer
:
```

Figure 3.9: Simple Router: Output process

```
PROC sink(CHAN OF ANY out, []BYTE buffer
          []CHAN OF ANY from.input,
          []CHAN OF ANY from.source,
          VAL INT taskId, typeId)
  WHILE TRUE
    INT length, source, destination, type :
    SEQ
      ALT
        ALT i = 0 FOR SIZE from.input
          from.input[i] ? destination; source; type;
                          length :: buffer

        ALT i = 0 FOR SIZE from.source
          from.source[i] ? destination; source; type;
                           length :: buffer

      out ! source; length :: buffer
:
```

Figure 3.10: Simple Router: Sink process

```
PROC input(CHAN OF ANY in, []BYTE buffer,
           []CHAN OF ANY to.output,
           [][]CHAN OF ANY to.sink,
           VAL []INT routing, VAL INT routing.base)

  WHILE TRUE
    INT length, source, destination, type :
    SEQ
      in ? destination; source; type; length :: buffer

      IF
        destination <> NONE
          ... route directed message

        TRUE
          ... route broadcast message
:
```

Figure 3.11: Simple Router: Input process

Input and Source The input and source processes are more complicated since they implement the directed and broadcast routing functions. These are shown in Figure 3.11 and Figure 3.12.

Since these processes will not implement the quasiadaptive routing function the routing of a directed message is simply the sequential function which uses the first element in the directed routing table entry for the task `destination`.

Routing a broadcast message involves forwarding the message to all pseudolinks contained in the broadcast routing table entry for the source task. This simply involves a different routing table lookup and a loop around the code to forward the message.

```
PROC source(CHAN OF ANY in, []BYTE buffer,
            []CHAN OF ANY to.output,
            []CHAN OF ANY to.sink,
            VAL []INT routing, VAL INT routing.base,
            VAL INT taskId, typeId)

  WHILE TRUE
    INT length, source, destination, type :
    SEQ
      in ? destination; length :: buffer
      type   := typeId
      source := taskId

      IF
        destination <> NONE
          ...  route directed message

        TRUE
          ... route broadcast message
:
```

Figure 3.12: Simple Router: Source process

```
{{{  route directed message
VAL routing.entry IS routing[routing.base + (2 * destination)] :

VAL pseudo.link   IS routing[routing.entry] :
IF
  pseudo.link < (SIZE to.output)
    VAL index IS pseudo.link :
    to.output[index] ! destination; source; type;
                       length :: buffer
  TRUE
    VAL index IS pseudo.link - (SIZE to.output) :
    -- input uses next line
    to.sink[index][type] ! destination; source; type;
                           length :: buffer
    -- source uses next line
    -- to.sink[index] ! destination; source; type;
    --                  length :: buffer

}}}
```

Figure 3.13: Simple Router: Directed routing

```
{{{  route broadcast message
VAL routing.entry IS routing[routing.base + ((2 * source) + 1)] :
INT i :
SEQ
  i := routing.entry
  WHILE routing[i] <> NONE
    SEQ

      VAL pseudo.link IS routing[i] :
      IF
        pseudo.link < (SIZE to.output)
          VAL index IS pseudo.link :
          to.output[index] ! destination; source; type;
                           length :: buffer
        TRUE
          VAL index IS pseudo.link - (SIZE to.output) :
          -- input uses next line
          to.sink[index][type] ! destination; source; type;
                               length :: buffer
          -- source uses next line
          -- to.sink[index] ! destination; source; type;
          --                  length :: buffer
      i := i + 1
}}}
```

Figure 3.14: Simple Router: Broadcast routing

Performance Considerations

There are a number of reasons why this appealingly simple prototype will perform badly. These are discussed and some solutions briefly described.

Message Header Format The components of the message are communicated as separate entities. Each communication event must first synchronize the communicating processes and then move the message data as a block; there is a substantial overhead involved in this synchronisation compared with the transfer of a short message.

The header data should be packed into a short vector; the message data must still be transferred separately since it is of variable length. The header fields should be packed as bit fields where appropriate to minimise header length.

Internal Copies The header and message data are copied between input and output processes. The input and output processes all run within the same processor and communication across an internal channel is implemented by a block move; unlike external communication the processor is not free for computations during the block move, which take a time proportional to the amount of data moved.

Since these processes necessarily share the same processor and memory it is sensible for them to share a common pool of buffers and associated headers. The communication of messages internally is replaced by swopping buffer indices. In this way there are the same number of synchronisation events but smaller block moves. Some care will have to be taken with the broadcast function since the `input` process cannot swop indices with more than one `output` process for a single message.

Conditional Input The execution of an alternation takes a time proportional to the number of guards. In order to perform an alternation the alternating process must enable each guard in turn then suspend itself. The process is resumed when a guard becomes ready and then disables each guard in turn. In a replicated

alternation there is the additional overhead of looping while enabling and disabling the guards.

The alternation and corresponding unconditional outputs look suspiciously like a queue. The alternation of `output` should be replaced by a dequeueing procedure operating on a queue of `input` process waiting for synchronisation with this `output`; the dequeue operation can swop buffer pointers and reschedule the dequeued `input` process, or deschedule if there is no queued process. The unconditional output of the `input` process should be replaced with an enqueue procedure; the enqueue operation deschedules if the `output` process is unready else it swops buffer pointers and reschedules the `output` process. Given such a queueing system it should be easy to implement the *short queue* quasiadaptive routing function.

Network Buffering An `input` process blocks while waiting to synchronise with the chosen `output` process; if the `input` process is handling a link then the link is unused which will cause the `output` process on the neighbour processor to block waiting for the link to become ready. The `output` process may be waiting for a link transfer to complete which can take a considerable time. This effect substantially degrades aggregate network bandwidth, as is discussed in [Hlu88].

The process queues should be replaced by buffer queues and more buffers should be associated with each link. It is possible to allocate an arbitrary positive number of buffers to each `input` process, and none to any `output` process, provided the `input` processes are also allocated queues, or stacks, of free buffers. Before message input an `input` process dequeues a free buffer, performs input and chooses an `output` process; the buffer is placed in the queue of the chosen `output` process. The `output` process dequeues a buffer from its queue and performs output; the freed buffer is returned to the queue of the `input` process which previously queued it to the `output` process.

External Copies Message data is always copied between the source application and its `source` process, and between a destination application and its `sink` process; i.e. each message is copied twice. We can always avoid the copy at the source,

and we can avoid the copy at the destination whenever the application is already waiting to receive the message (although in the case of a broadcast message it will usually be convenient to enforce the copy at the destination). We can achieve this within the above system at the expense of further distinguishing the input and output processes of network and transport. The method for achieving this will also obviate the need to allocate message buffers to transport processes which could prove costly in terms of memory usage.

Transport input processes should not input message data directly from application into a buffer. They can input the address of the application message buffer and arrange for network to output this directly on the transputer link. After this output has completed the application can be acknowledged and may reuse the message buffer.

Transport output processes should not output message data directly to application from a buffer. They also can input the address of the application message buffer and arrange for a network process later inputting a message destined for this transport output to detect that the transport output is ready and input the message to application space rather than the usual message buffer; alternatively one could arrange for the transport process to perform this input.

3.5.2 Improved Prototype

Implementation of the improvements to the simple occam prototype above will require a certain amount of pointer handling and it is therefore advantageous to move to a language which explicitly contains pointers; we shall use ANSI C for this implementation. The quasiadaptive routing function is introduced.

The process structure of this prototype is identical to the above; the major changes arise in the exchange of information between the input and output processes. In this case channels are replaced by pointers to shared structures; the implementation requires critical sections. These conditions can be met provided the group of processes all run at the same transputer priority level; in practice routing processes

should always be run at high priority for performance purposes and all computations performed by high priority transputer processes are an implicit critical region.

Data Structures

The prototype requires shared access to two structures; message buffers with the associated headers and buffer queues. It is as well to use structured data types for these, and other, purposes and we introduce the associated structures at this point. The ANSI C specifications are given in Figure 3.15.

Header : The `Header` structure contains the message header data.

destination For a directed message this is the task identifier of the destination task shifted left one place. For a broadcast message the source task identifer shifted left one place and the least significant bit set. The masking and testing of the lowest four bits are cheap operations on the transputer. The routing tables were interleaved so that the same indexing applies to both tables.

source The most significant byte holds the message type. The remaining bits hold the message source identifier shifted left one place. The least significant bit is set to indicate quasiadaptive routing. We will access the most significant byte with a large shift operation (although this can be more efficiently performed by loading the byte alone from store).

length The length of the message in bytes. This is assumed to be positive or zero by network processes.

Buffer : The `Buffer` structure contains the message header and a pointer to the actual memory area used to store the message data along with some book-keeping items.

hdr The message header as described above.

```
struct Header {
int destination;
int source;
int length;
};

struct Buffer {
struct Header hdr;
char          *data;
struct Queue  *owner;
int           special;
struct Buffer *link[];
};

struct Client {
union {
int destination;
int *source;
} task;
int length;
char *data;
};

struct Queue {
struct Buffer *head;
struct Buffer *tail;
int           count;
void          *pid;
int           pseudo;
CHAN          *channel;
struct Queue  *indirect[];
};
```

Figure 3.15: Improved Router: Data Structures

data A pointer to the message data. If the buffer belongs to a transport process then this will be a pointer into application space.

owner A pointer to the queue of the process which "owns" the buffer. The buffer will be returned to this queue after the message has been output.

special The normal action of an output process is to output the message held in `data` and queue the buffer back to `owner`. This field is used to indicate that some special action should be taken; see later.

link A number of links for queue lists. Note that there is a queue for each pseudolink; the reason for this is that during broadcast a message buffer can be queued to more than one, and distinct links are needed, however there can only be one queue reference for each pseudolink.

Client : The `Client` structure will be used for communication between a transport process and its client, an application process.

data A pointer to the application message buffer.

length The length of data to send or the length of the largest message which can be received.

task The type of this field depends on whether the request is for a message send or receive.

destination The destination task identifier during message send. This takes the value `NONE` for broadcast. In the case of a directed message it is a valid identifier shifted left one place; the least significant bit set indicates a quasiadaptive message.

source A pointer to a place for return of the source task identifier during message receive. A copy of the source field of the message header will be written to the address given.

All bit manipulation of the destination identifier required before message send and cleaning up of the source identifier after message receive is expected to be performed by the procedural interface.

Queue : The `Queue` structure holds a queue of buffers and information about the queue process.

head Front of buffer queue list; pointer to first buffer in queue.

tail Back of buffer queue list; pointer to last buffer in queue.

count The length of the buffer queue.

pid The process identifier of the queue process.

pseudo The pseudolink number of the queue process.

channel The channel used by the queue process.

indirect The routing tables are assumed to be transformed to a pointer form in which a routing entry element is a pointer to a queue rather than a pseudolink number. Where the pseudolink is typed, i.e. transport, a level of indirection is required which is provided by this field. Where the pseudolink is untyped, i.e. network, this field is `NULL`.

Macro Definitions

We assume a few macros which handle transputer operations such as communication and process control.

`Move(source, destination, length)` : `length` bytes are moved from the address `source` to the address `destination`. This is a null operation if length is zero.

`Input(channel, destination, length)` : `length` bytes are input to the address `destination` from `channel`. This is a null operation if length is zero.

`Output(channel, source, length)` : `length` bytes are output to `channel` from the address `source`. This is a null operation if length is zero.

`Stop(pid)` : the process is stopped, without being queued for later execution, storing the process identifier in `pid`.

`Run(pid)` : the process with identifier `pid` is restarted, i.e. it is queued for later execution.

Using these macros and the above data structures we construct a number of macros used by the network and transport processes.

`Enqueue(queue,buffer)` : the message buffer referenced by `buffer` is appended to the buffer queue referenced by `queue`. If the queue was empty and a process waiting on the queue then that process is restarted. Used by all processes.

`Dequeue(queue,buffer)` : a reference to a message buffer is removed into `buffer` from the buffer queue referenced by `queue`. If the queue was empty then the executing process stops and waits on the queue. Used by all processes.

`RouteDirected(buffer,entry)` : the message buffer referenced by `buffer` is routed as a directed message with the routing table entry referenced by `entry`. Used by input processes.

`RouteBroadcast(buffer,entry)` : the message buffer referenced by `buffer` is routed as a broadcast message with the routing table entry referenced by `entry`. Used by input processes.

`RouteExceptional(buffer,queue)` : the message buffer referenced by `buffer` is routed as an exceptional directed message before data input, to the queue referenced by `queue`. If the queue belongs to a ready transport process then message data is not input, else message data is input and queued normally. Used only by network input.

Processes

Output and Sink The output processes are simpler than the input processes since they do not have to deal with the routing function. The `output` process is effectively a message multiplexor; see Figure 3.16.

```
void output(struct Queue *me) {
  struct Buffer *buffer;

  for (;;) {
    Dequeue(me,buffer);

    Output(me->channel, &buffer->hdr, sizeof(struct Header));
    Output(me->channel, buffer->data, buffer->hdr.length);

    if (buffer->special != 0) buffer->special--;
    else                      Enqueue(buffer, buffer->owner);

  }
}
```

Figure 3.16: Improved Router: Output Process

The **sink** process is slightly more complicated than **output** since it has to handle the application and the possibility of special input from a network channel into user space. It is also responsible for avoiding overflow of the user buffer; in Figure 3.17 this has been omitted for clarity.

Input and Source The two input processes, which handle the routing functions, are rather more complicated. The **input** process handles special routing where the routing table entry contains a single element; see Figure 3.18.

The **source** input process does not perform special routing but is responsible for constructing a valid message header from the application send request; in Figure 3.17 we have omitted error checking for clarity.

```
void sink(struct Queue *me) {
  struct Client user;
  struct Buffer *buffer;

  for (;;) {
    Input(me->channel, &user, sizeof(struct Client));
    Dequeue(me,buffer);

    *user.task.source = buffer->hdr.source;
    user.length       = buffer->hdr.length;

    if (buffer->special == -1) {
      buffer->special = 0; /* reset special and input to user */
      Input(buffer->owner->channel, user.data, user.length);
      Enqueue(buffer, buffer->owner);
      Run(buffer->owner->pid); /* restart net input process */
    }
    else  {
      Move(buffer->data, user.data, user.length);
      if (buffer->special != 0) buffer->special--;
      else                      Enqueue(buffer, buffer->owner);
    }

    Output(me->channel, &user.length, sizeof(int)); /* ack user */
  }
}
```

Figure 3.17: Improved Router: Sink Process

```
void input(struct Queue *me, struct Queue **routing[]) {
  struct Buffer *buffer;
  struct Queue  **entry;

  for (;;) {
    Dequeue(me,buffer);

    Input(me->channel, &buffer->hdr, sizeof(struct Header));

    entry = routing[buffer->hdr.destination];
    if (entry[1] != (struct Queue *)NULL)  {

      Input(me->channel, buffer->data, buffer->hdr.length);
      if (buffer->destination & 1) RouteBroadcast(buffer, entry);
      else                         RouteDirected(buffer, entry);
    }
    else                           RouteExceptional(buffer, entry[0]);
  }
}
```

Figure 3.18: Improved Router: Input Process

```
void source(struct Queue *me, struct Queue **routing[],
                                    int taskId, typeId) {
  struct Client user;
  struct Buffer *buffer;

  taskId = taskId << 1;              /* realign taskId and */
  typeId = (typeId << 24) | taskId; /* typeId in advance */

  Dequeue(me,buffer);
  for (;;) {
    Input(me->channel, &user, sizeof(struct Client));

    buffer->hdr.length = user.length;
    buffer->hdr.data   = user.data;

    if (user.task.destination == NONE) {
      buffer->hdr.destination = taskId | 1;
      buffer->hdr.source      = typeId;
      RouteBroadcast(buffer, routing[buffer->hdr.destination]);
    }
    else {
      buffer->hdr.destination = user.task.destination & (-2);
      buffer->hdr.source      = (user.destination & 1) | typeId;
      RouteDirected(buffer, routing[buffer->hdr.destination]);
    }

    Dequeue(me,buffer);
    Output(me->channel, &user.length, sizeof(int));
  }
}
```

Figure 3.19: Improved Router: Source Process

```
Enqueue(queue,buffer) {
  queue->count++;
  if (queue->count == 0) {
    Run(queue->pid);
    queue->head = buffer;
  }
  else {
    queue->tail->link[queue->pseudo] = buffer;
  }
  queue->tail = buffer;
}

Dequeue(queue,buffer) {
  queue->count--;
  if (queue->count == -1) {
    Stop(queue->pid);
  }
  buffer = queue->head;
  queue->head = buffer->link[queue->pseudo];
}
```

Figure 3.20: Improved Router: Queue Handling

Derived Macros

Queue Handling In Figure 3.20 we show the queue handling primitives Enqueue and Dequeue. These maintain queues as simple singly linked lists with a count of the queue length. A negative queue length, -1, is used to signify that a process is waiting on an empty queue; this has the advantage that we can distinguish between an empty queue and an empty queue with a waiting process.

Directed Routing We show directed routing in Figure 3.21. A branch is made on the least significant bit of the destination to determine whether the sequential or quasiadaptive strategy is to be used. Note that the quasiadaptive strategy will

```
RouteDirected(buffer,entry) {
  struct Queue *out;
  int min;
  if (buffer->source & 1) {
    min = INFINITY;
    while (*entry) {
      if ((*entry)->count < min) {
         out = *entry;
         min = out->count;
      }
      entry++;
    }
  }
  else {
    out = *entry;
  }
  if (out->indirect) out = out->indirect[buffer->hdr.source >> 24];
  Enqueue(out,buffer);
}
```

Figure 3.21: Improved Router: Directed Routing

always choose an empty queue with a waiting output process in preference to an empty queue without a waiting process due to the use of the count described above.

Broadcast Routing Figure 3.22 shows broadcast routing. In this case the `special` field of the buffer is set to one less than the number of output processes queued to in order that they can correctly free the buffer once all have completed message output.

Exceptional Routing The exceptional routing of a directed message which occurs in `networkInput` before message data has been input is shown in Figure 3.23. If the target queue belongs to a waiting transport process, determined by the requirement for indirection and a queue count of -1, then the `special` field is set to

```
RouteBroadcast(buffer,entry) {
  struct Queue *out;
  int count;
  count = -1;
  while (*entry) {
    out = *entry++;
    if (out->indirect) out = out->indirect[buffer->hdr.source >> 24];
    Enqueue(out,buffer);
    count++;
  }
  buffer->special = count;                /* set special count */
}
```

Figure 3.22: Improved Router: Broadcast Rouitng

-1 and the input process stands off its channel. If on the other hand the transport was unready or the target was a network process then the message is input and queue as above.

3.5.3 Performance

Message Latency We can characterise the performance of the systems described above under conditions of light network loading where conflict for link usage does not arise. We consider the operations involved in passing a single directed asynchronous message of l bytes along a path comprising h links.

Message Send Events are initiated by application making a message send request to transport. The time taken for the application and transport processes to synchronize and exchange message data is expressed as $\alpha_1 + \beta_1 l$ where β_1 will be zero if message data is not copied between transport and application.

Source Routing The transport process servicing the send request performs a routing decision and enqueues the message output to a network process. This

```
RouteExceptional(buffer,out) {

  if (out->indirect &&
      (out = out->indirect[buffer->hdr.source >> 24])->count == -1) {
    /* ready transport */
    buffer->special = -1; /* set special */
    Enqueue(out,buffer);
    Stop(queue);          /* stand off channel */
  }
  else {
    /* network or unready transport */
    Input(in, buffer->data, buffer->hdr.length);
    Enqueue(out,buffer);
  }
}
```

Figure 3.23: Improved Router: Exceptional Routing

takes a characteristic time $\alpha_2 + \beta_2 l$ where β_2 is zero if message data is not copied between transport and network.

Link Transfer The network process dequeues the message and initialises the transfer of the message header, followed by the message data. The time taken for these operations is written as $\alpha_3 + \beta_3 l$ where β_3 is the hardware bandwidth of the transputer link.

Through Routing A network process which received the message from the transputer link performs a routing decision which enqueues message output to another network process. We characterise the time taken for through routing as $\alpha_4 + \beta_4 l$ where β_4 is zero if message data is not copied between network processes.

Destination Routing A network process which received the message from the transputer link performs a routing decision which enqueues message output to another network process. We characterise the time taken for through routing as $\alpha_5 + \beta_5 l$ where β_5 is zero if message data is not copied between

transport and network.

Message Receive The message passing events complete when the (waiting) destination application process receives the message from the final transport output process and is scheduled for execution. The time taken for the application and transport processes to synchronize and exchange message data is expressed as $\alpha_6 + \beta_6 l$ where β_6 will be zero if message data is not copied between transport and application.

If the path contains $h > 0$ links then there are $h - 1$ through routing events and h link transfer events, thus the message latency $T_{l,h}$ is expressed as

$$\begin{aligned} T_{l,h} &= a + bh + cl + dlh \\ a &= \alpha_1 + \alpha_2 - \alpha_4 + \alpha_5 + \alpha_6 \\ b &= \alpha_3 + \alpha_4 \\ c &= \beta_1 + \beta_2 - \beta_4 + \beta_5 + \beta_6 \\ d &= \beta_3 + \beta_4 \end{aligned}$$

We can estimate the constants in this expression by employing use of a simple *echo* technique in which we exploit the fact that the time taken to send a message from X to Y and immediately back to X from Y is very nearly twice the time for the message send from X to Y provided that the path from Y to X is the opposite of the path from X to Y. We can arrange for this to be true quite easily by using a simple linear chain configuration. In Figure 3.24 we present a few measurements of $T_{l,h}$ for TINY which demonstrate that the bilinear form is observed.

From similar data we obtain the results shown in Table 3.4 using INMOS T800 transputers, running at 20MHz, with 200ns external memory. In these experiments we arranged for the routing threads and critical shared data structures to be located in the fast, 50ns, internal memory.

We have performed a similar calculation on the Intel iPSC/2 hypercube multicomputer and obtain the results shown in Table 3.5. The figures for message length of less than 100 bytes and larger are due to the communication protocol between NX/2 node operating systems which sends messages of < 100 bytes asynchronously

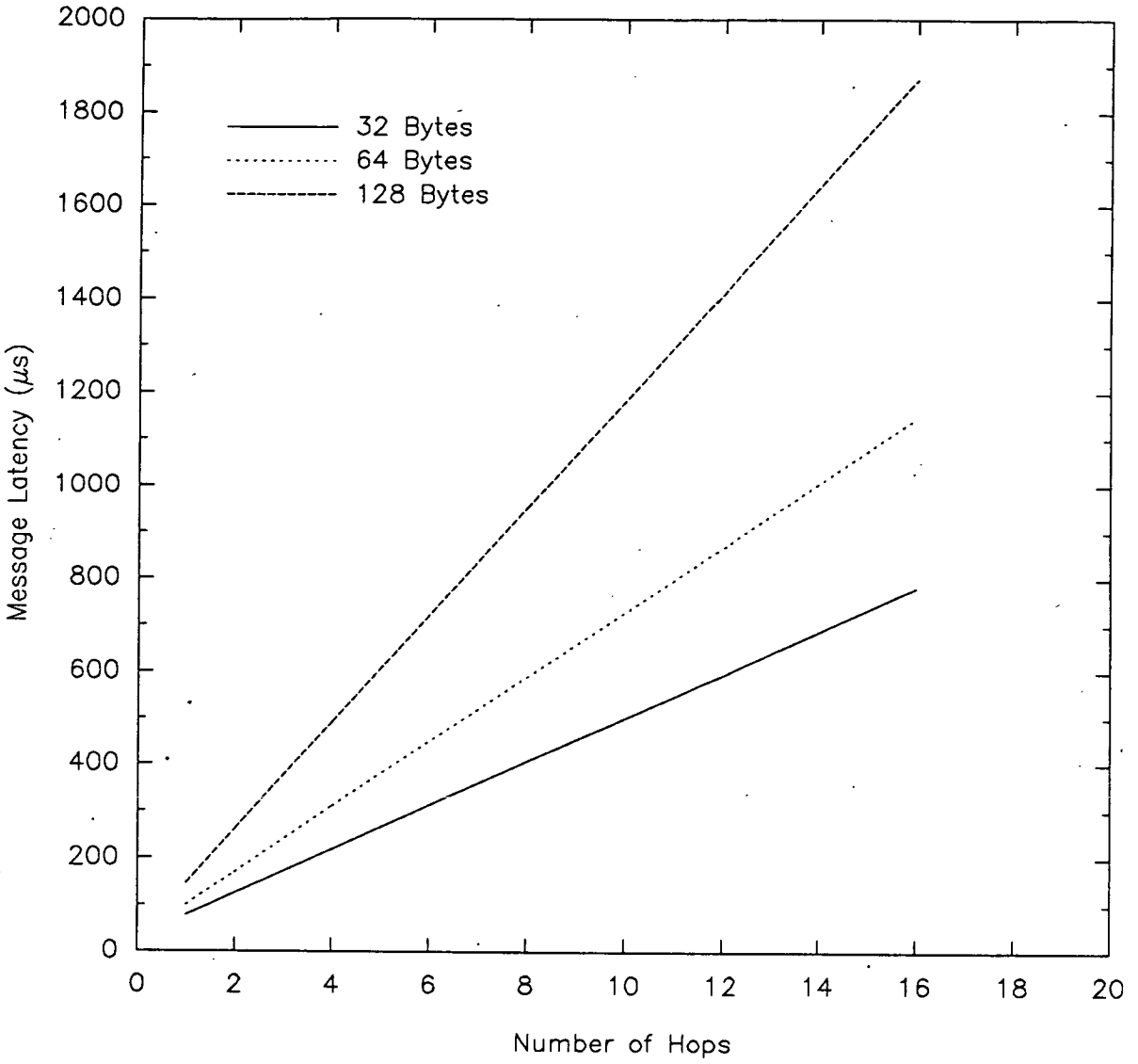

Figure 3.24: Quiet Network Message Latency

	20Mbits^{-1} Links	10Mbits^{-1} Links
a	$30.5\mu s$	$30.5\mu s$
b	$24.3\mu s$	$30.0\mu s$
c	$0.00\mu s$	$0.00\mu s$
d	$0.71\mu s$	$1.25\mu s$

Table 3.4: Quiet Network Latency in TINY

	$l < 100$ bytes	$l > 100bytes$
a	$367\mu s$	$670.60\mu s$
b	$10.2\mu s$	$30.0\mu s$
c	$0.05\mu s$	$0.09\mu s$
d	$0.00\mu s$	$0.00\mu s$

Table 3.5: Quiet Network Latency in the iPSC/2

but synchronises transfer for messages of > 100 bytes; it is then only reasonable to compare performance for messages of less than 100 bytes. These figures are in general agreement with those reported in [BR89].

There are two important differences between the figures obtained for these two systems. The d term for the iPSC/2 is identically zero due to the use of circuit switching whereas this term for TINY is equal to the asymptotic link bandwidth (where the links are switched by the MEIKO ES2 switching chips) due to the use of packet switching. The a term for TINY is approximately an order of magnitude less than that observed in the iPSC/2 which is crucially important in determining the notional zero length message latency.

It is of some interest to compare the message latency for neighbour communications alone, since such figures are directly comparable between these two systems. This time can be represented as $T_0 + lR_\infty$ where T_0 is the set up time, $T_0 = a + b$, and R_∞ is the asymptotic bandwidth, $R_\infty = c + d$. In [BR89] such figures were quoted for the iPSC/1, iPSC/2 and Ncube hypercubes, in Table 3.6 we have compared these figures (for messages of less than 100 bytes) with TINY.

CPU Impact In order to understand the impact of message events on an intermediate processor we have to consider the operations involved in message routing. Some of these have been dealt with above. However the events required to prepare for message input and recover from message output have not been discussed.

	T_0 (μs)	R_∞ (μs)
Ncube	446	2.5
iPSC/1	1000	1.0
iPSC/2	350	0.05
TINY	54.8	0.71

Table 3.6: Comparision of Neighbour Message Latencies

Message Input The network process prepares for input by allocating a buffer and initialising input of the message header; this preparation is not observed above. During header input the process is suspended and resumes only when the input has completed. After determining that the destination is not located on this processor the process initialises the input of message data. Again the process is suspended and resumes when input has completed. The DMA engine utilises memory cycles during message input which induces wait states in the processor. Process resumption involves interrupting an executing low priority process. We characterise this component of the through routing CPU requirement as $\alpha_7 + \beta_7 l$.

Through Routing The analysis of through routing above is entirely applicable to the discussion of CPU impact and we characterize this component as $\alpha_4 + \beta_4 l$ in the same way.

Message Output The network process prepares for output by dequeueing a buffer and initialising output of the message header. During header output the process is suspended and resumes only when the output has completed. Message data is then output in the same way. The process recovers from output by enqueueing the buffer to the original input process and suspending itself pending a further output. Again the DMA engine utilises memory cycles and process resumption involves an interrupt. We characterise this component of the through routing CPU requirement as $\alpha_8 + \beta_8 l$.

	20Mbits^{-1} Links	10Mbits^{-1} Links
a'	$32.4\mu s$	$32.3\mu s$
b'	$0.1\mu s$	$0.1\mu s$

Table 3.7: Parameters of Through Routing CPU Impact

The CPU requirement of through routing, T_l^r, for a message of length l is given by the above analysis as

$$\begin{aligned} T_l^r &= a' + b'l \\ a &= \alpha_7 + \alpha_4 + \alpha_8 \\ b &= \beta_7 + \beta_4 + \beta_8. \end{aligned}$$

We can estimate the quantities a' and b' by extending the above *echo* experiment so that all intermediate, through routing, processors are running a simple low priority *work* and comparing the time taken to execute the work in the presence of message passing events with the time taken in the absence of such events. The work loop chosen contained a mixture of floating point and integer calculations. The results shown in Table 3.7 were obtained for TINY.

The difference in a' between the two link speeds is within the resolution capability of the experiment for this quantity. The value of b' implies that in these experiments 100% of the DMA external memory requests caused the processor to wait.

3.6 Summary

In this chapter we have described the methods used in the implementation of a communication system for INMOS transputer arrays, called TINY. This system exploited many of the ideas in Chapter 2 and indeed experience with the implementation and utilisation of TINY has been a major stimulation for these ideas.

Since the configuration hardware and/or software of the development machine are occasionally found to be unreliable the system initially performs a topology explo-

ration. Two candidate algorithms for this calculation were described and a rationale given for preferring one of these in the development environment.

The methods used for construction of routing tables were discussed and the acyclic routing function was evaluated for a small number of topologies. We observed that the trivial colouring algorithm generates hot links which will impede performance in a number of applications bound by aggregate bandwidth e.g. distributed multidimensional fast fourier transforms. The search for a deterministic algorithm producing non trivial colouring solutions which perform better on worst loading metrics is an ongoing topic.

The techniques used in the development of the message passing processes were described in some detail. A simple occam prototype clearly displayed the basic process structure however could not perform well for a number of reasons. These issues were discussed and an improved prototype was presented. The performance of these systems was analysed and the characteristic parameters of TINY were obtained. Similar queueing techniques were independently arrived at by Mike Surridge [Sur] in the implementation of message routing software and similar performance characteristics were reported [Sur90].

The communication services offered by TINY, through four stages of evolution, have been available for public use in Edinburgh for approximately 18 months and have been incorporated into a large number of application programs. In Chapter 4 we describe a simple application of TINY in the field of molecular graphics which we have implemented on the Edinburgh Concurrent Supercomputer [BKW88].

Chapter 4

Application

4.1 Introduction

Molecules are generally three dimensional objects; the human mind requires some assistance in the visualisation of such objects. This has become increasingly important in the study of biochemical processes where the shapes of molecules play a large part in the reactions which they undergo. There are two traditional approaches to the construction of molecular models; the so called "ball and stick" and "solid sphere" representations.

Where the molecule is small, containing no more than a few tens of atoms, it is reasonable to construct models with commercially available components; indeed I recall using a kit of such components when studying organic chemistry as an undergraduate. This approach fails when the modern biochemist is required to visualise the structure of molecules comprising several thousands of atoms, in such a regime the computer generation of three dimensional images is beneficial.

The solid sphere models are of particular interest when considering biochemical problems such as the reactions between enzymes and substrates where the shape of the substrate, particularly in the vicinity of the active site, and the shape of the enzyme may be of crucial importance; ball and stick models do not encode the relevant visual signals.

A typical solid sphere model contains a large number of spheres of varying radii, representing the covalent radii of the corresponding atoms, at different positions according to the atomic and molecular configuration to be visualised. The atoms are colour coded which conveys meaningful information about the type of atom and its position with respect to functional groups etc, usually according to one of a number of well established colour schemes.

This description naturally suggests the utilisation of the ray tracing technique for rendering the two dimensional image. There are two reasons for which different techniques have been sought after:

- The computational cost of rendering a scene containing several thousands of atoms by ray tracing is very large. On conventional modern workstations and even moderate supercomputers this is prohibitive.
- The quality of image produced by ray tracing is actually rather better than is required by the biochemist. In practice a photo-realistic image is less important than one which merely conveys the necessary information.

An algorithm which generates images of adequate quality and is computationally less demanding than ray tracing is reported in [BA88]. In Section 4.2 we describe this method. A sequential FORTRAN implementation was made available by the authors; we also characterise the performance of this code.

The execution time to render a scene consisting of a few thousands of atoms is too large for "interactive" use on current uniprocessor computers, in which cases the biochemist wishes to view one or more molecules from varying positions in order to compare structural features. In Section 4.3 we describe a simple approach to exploiting the parallelism inherent in the method and characterise the performance of the resulting parallel algorithm.

4.2 Sequential Algorithm

In this section we describe the sequential algorithm and characterise the performance of its components. This introduces the methods used and allows us to discuss later how the parallel algorithm is obtained from the sequential algorithm.

The *lighting model* contains two light sources. The *primary* light source is located at an arbitrary position (P_x, P_y, P_z), and the *secondary* source located at a position on the z-axis $(0, 0, S_z)$. A uniform background, or ambient light, is also used. The eye point is coincident with the secondary light source.

The algorithm takes account of three features which provide depth cueing:

- Perspective; distant lengths appear smaller than close lengths. A one-point perspective projection is used; the projection centre is the eye point and the projection plane is $z = 0$.
- Shading; when a light is shone upon an object having a reflective surface a *highlight* is caused by specular reflection, whereas light reflected by the remainder of the surface is caused by diffuse reflection. The shading model developed by Phong [FvD82] is utilised.
- Shadows; a point visible from the eyepoint may be in shadow from the primary light source; if it were in shadow from the secondary source then it would not be visible. An explicit shadow test is, optionally, applied to determine whether the primary light source contributes to the colour of each pixel.

The quality of the image produced by these processes is actually rather good. One deficit is an *aliasing* effect where the edges of spheres appear jagged due to the limitations of the pixel resolution. This effect is decreased by overcalculation and averaging, i.e. one calculates pixels on a finer mesh than the display and obtains each display pixel by averaging over a number of calculated pixels.

Basic Method

The description of the positions and sizes of the atoms will typically be given in some, fairly arbitrary, user coordinate system. In order to begin the rendering process these must be transformed into a suitable object coordinate system which also implements any rotations, translations or scaling operations requested by the user. This is achieved by representing the composed sequence of operations as a geometric transformation matrix in the 4×4 homogeneous coordinate system [FvD82]. The preprocessing transformation then requires, in principle, for each sphere the multiplication of a 4×4 matrix by a 4 vector. In practice this is implemented using fewer operations. The perspective transformation is also applied at this point.

The method for determining which portions of the image are in shadow is actually the same as the method for determining which portions of the image are visible, except of course that in the former case the eye is considered to be at the primary light source. A copy of the *object space* is therefore transformed into the *shadow space*; a rotation and scaling which maps the point (P_x, P_y, P_z) to the point $(0, 0, S_z)$. is applied.

If there are N_s spheres in the model then the preprocessing stage requires a time directly proportional to N_s. This transpires to be a small fraction of the total imaging time.

Each calculated pixel is represented as a point (x_p, y_p, S_z) in the eye plane. For every line (x_p, y_p) we must determine whether it intersects with the surface of a sphere in object space; if the line intersects no spheres then the pixel is assigned some background colour. The line is determined to intersect the surface of a sphere with centre at (x_i, y_i, z_i) and radius r_i straightforwardly by the test

$$r_i^2 - \Big((x_p, y_p) - (x_i, y_i)\Big)^2 > 0.$$

When one or more spheres are intersected we must determine the intersection point (x_p, y_p, z_p) which is closest to (x_p, y_p, S_z) and whether this intersection point is in shadow. The point is determined to be in shadow by rotating the point into shadow

space to (x_s, y_s, z_s) and determining whether the line (x_s, y_s) intersects the surface of a sphere in shadow space. If so then the intersection closest to the eye plane is again calculated and the point determined to be in shadow if this point lies between (x_s, y_s, z_s) and the eye plane.

The RGB components of the pixel colour are obtained from the colour of the object sphere intersected which is illuminated by the ambient light, the secondary source and the primary source if not in shadow.

After all pixels have been computed the averaging process which implements anti-aliasing is applied to produce the final image pixels which are sent to the display device.

Enhancements

The direct implementation of this method would be computationally inefficient since the search through space for intersections requires one to perform the intersection calculation for each pixel - sphere pair. If there are N_p calculated pixels then this search would require a time proportional to $N_s \times N_p$; This is reduced by two methods, *depth sorting* and *tile listing.*

Depth Sorting The spheres are sorted according to the distance of the closest point to the eye plane; this is performed in both object and shadow space. Depth sorting has the advantage that when performing the search through, say, object space for the closest intersection to the eye plane it will often be unnecessary to search the complete list of spheres. Rather one searches, beginning at the near end of the list, until an intersection is found. The point of intersection is recorded but the search continues until no remaining sphere can reach the point of intersection, i.e. its near point is behind the point of intersection. The sort can be performed in $O(N_s \log N_s)$ time.

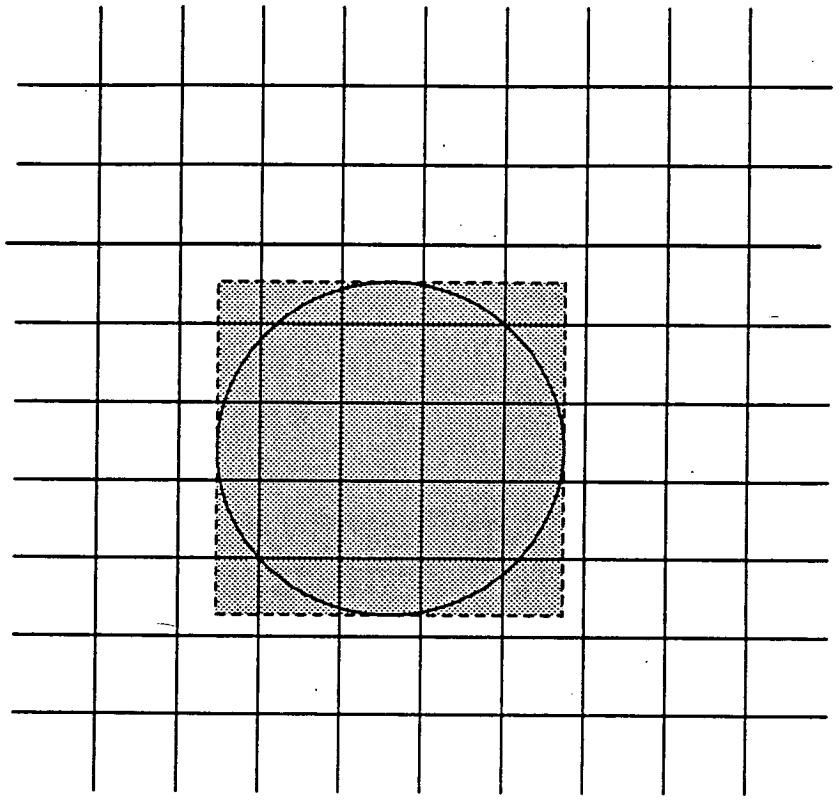

Figure 4.1: Calculation of Impingement

Tile Listing The mesh of calculated pixels is divided up by a regular decomposition into a mesh of rectangular tiles. For each tile a shortlist of spheres which *impinge* on the tile is calculated; a sphere centered at (x_i, y_i, z_i) and of radius r_i is determined to impinge on a tile if the region in the xy plane bounded by the lines

$$x = x_i + r_i \quad x = x_i - r_i$$
$$y = y_i + r_i \quad y = y_i - r_i$$

intersects with the region covered by the tile, as depicted in Figure 4.1 where each tile intersecting the shaded region is deemed to be impinged upon by the sphere. This calculation is implemented by determining the square enclosing the projection of the each sphere onto the eye plane as above, clipping this square to the visible area of the plane, and adding a pointer to the sphere to the list for each tile intersected.

The tile listing method has the advantage that we have reduced the extent of space which must be searched when determining the intersection of a line from the eye plane; there is trivially a large pay-off when there are no spheres impingeing on a

tile. If there are a total of N_i impingements then the preparation of the tile lists takes a time which increases linearly with N_s and N_i. Note that where there are N_t tiles then $N_s \leq N_i \leq N_s \times N_t$. The tile size is chosen to trade-off the time spent calculating the tile lists against the time saved searching space for intersections.

As an aside, the sequential implementation of the algorithm provided does not implement anti-alias averaging across tile boundaries.

Characterisation

The algorithm with depth sorting and tile listing enhancements is shown schematically in Figure 4.2. In order to determine a suitable strategy for parallelisation we have timed the computations in each stage for a variety of N_s, using a single INMOS T800 with a clock speed of 20 MHz and external memory cycle time of 200 ns. There was no actual display in the Tile Painting stage and anti-alias averaging was not performed. The molecules chosen were fragments of the penicillin peptide substrate. The target display device will provide a 768×512 pixel screen and a tile size of 16×16 pixels was used.

The results of these calculations are shown in Table 4.1. The first table shows the execution time of the preprocessing, sorting listing and painting stages. The second table shows a breakdown of the tile listing stage into the time spent projecting spheres onto the eye plane and adding spheres to the tile lists; N_i is also shown for reference.

4.3 Parallel Adaption

In this section we describe how the parallel version of the molecular imaging algorithm above is obtained. We shall use the properties of the algorithms involved and the performance characterisation of the sequential algorithm.

Display is effected using a MEIKO MK015 graphics board containing an INMOS T800

```
/* preprocessing */
LOOP over all spheres
  Transform sphere to object space
  Transform object sphere to shadow space

/* depth sorting */
Heap sort object space spheres according to near point depth
Heap sort shadow space spheres according to near point depth

/* tile listing */
LOOP over all spheres
  Project object sphere onto eye plane and clip projection
  LOOP over all impinged object tiles
    Add sphere to object tile list
  Project shadow sphere onto eye plane anc clip projection
  LOOP over all impinged shadow tiles
    Add sphere to shadow tile list

/* tile painting */
LOOP over all tiles
  IF object tile list not empty
    LOOP over all pixels in tile
      Determine nearest sphere intersection point to eye plane
      IF intersection found
        IF intersection in shadow
          Paint pixel without primary lighting
        ELSE
          Paint pixel with primary lighting
      ELSE
        Paint background colour pixel
    Perform anti-aliasing average
  ELSE
    Paint tile of background colour pixels
```

Figure 4.2: Schema of Algorithm

N_s	Processing	Sorting	Listing	Painting
500	0.08s	0.14s	0.19s	33.4s
1000	0.15s	0.31s	0.36s	42.0s
1500	0.22s	0.48s	0.54s	61.9s
2000	0.29s	0.67s	0.71s	79.2s

N_s	N_i	Sphere Projection	List Addition
500	8125	60ms	127ms
1000	16565	105ms	258ms
1500	24680	152ms	284ms
2000	33194	199ms	514ms

Table 4.1: Characterisation of Sequential Components

transputer and dedicated graphics hardware. This is used in 24-bit pixel mode on a 768 × 512 screen. A simple graphics task was written to run on this processor which provides two primitives:

Fill Colours each pixel within a rectangular area of the screen to the same specified RGB colour. This is used to paint a tile consisting entirely of background pixels.

Draw Colours each pixel within a rectangular area of the screen to different specified RGB colours. This is used to paint a tile containing one or more non-background pixels.

It is useful to consider the extent to which bandwidth into the graphics processor will determine the performance of the parallel algorithm. In order to render a single frame approximately 0.85 Mbytes of data have to be fed to the graphics processor, assuming that one quarter of the tiles are background and each tile consists of 16×16 pixels. With four transputer links operating at 20 Mbit/s, switched through the MEIKO ES2 switching chips, this cannot be achieved in less than approximately 0.16s. In practice we anticipate that memory contention between the DMA engines, the processor and the graphics hardware will increase this quantity to roughly 0.25s.

The transputer links operate concurrently with the processor and we attempt to exploit this fact by forwarding primitives to the graphics processor while calculating further primitives. It follows that we require a message routing process on each transputer. Despite the fact that the communication patterns of this problem transpire to be rather simple it is useful to exploit the availability of general purpose routing software provided that the overheads in using such tools are acceptable.

The TINY routing system uses a 12 byte header for each message which increases the data which must be fed into the graphics processor by 2%; this is quite insignificant. The processors neighbouring the graphics processor will route the largest number of messages since every graphics primitive must be routed via one of these processors. Under the same assumptions as above each neighbour to the graphics board routes approximately 380 messages. Using the through-routing CPU impact parameters given in Chapter 3 this corresponds to 0.036 seconds which represents about 14% of the estimated lower bound execution time obtained above.

User interaction and file system access are provided by a single *control* task which can be located in the host machine. We do not discuss the implementation of the graphics codes and restrict our discussion of the control code to those parts directly involved in the parallel algorithm. There are one or more *render* tasks, each of which runs on an INMOS T800 processor with 4 Mbytes of external memory. Each of the P render tasks is identified by a unique integer p in $(0, P]$. The render tasks execute the major part of the parallel algorithm.

Tile Painting

It is clear from Table 4.1 that we must primarily seek parallelism in the Tile Painting operation. Fortunately this will be easy to obtain since the tiles are entirely independent of one another. The only issue to be addressed in exploiting this parallelism is that of load balancing.

In order to paint a given object tile it will be necessary for a task to have available the list of object space spheres impingeing on the tile so that intersections of rays

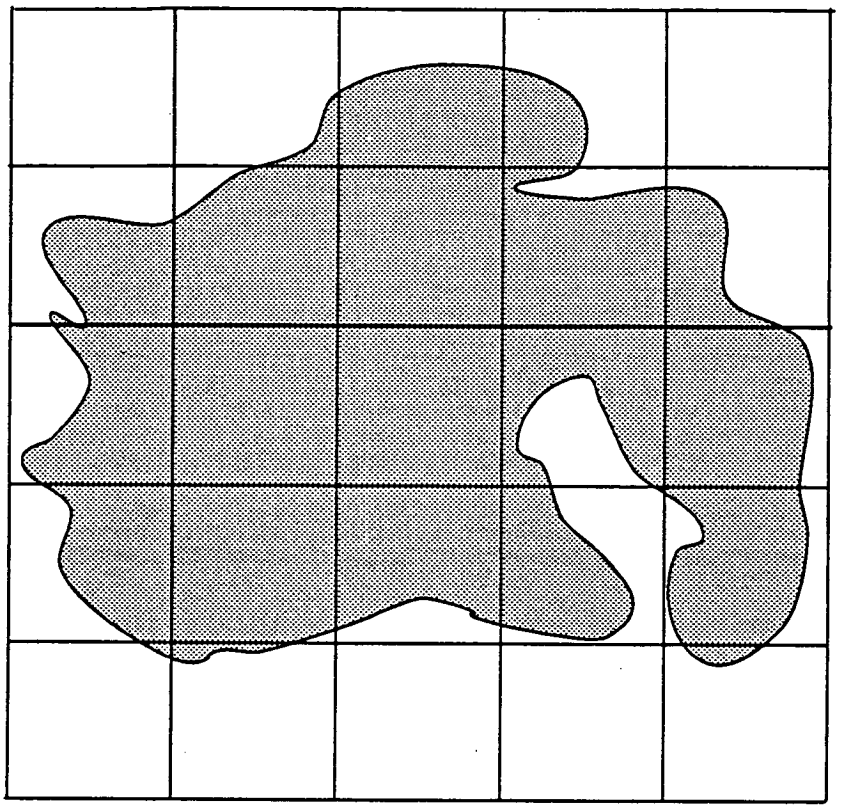

Figure 4.3: Regular Domain Decomposition

cast from the eye plane with these spheres can be calculated. It will also be necessary for the shadow tile lists of shadow space spheres which may render the pixels in shadow to be available. There is some advantage in having a copy of all object and shadow tile lists stored in every render task.

We can straightforwardly implement the scheme which these considerations suggest. In the first instance the control task gathers the positions, radii and colours of atoms from one or more files and broadcasts this data to the collection of render tasks along with control information such as the position and intensity of the primary light source. Having received this data each render task proceeds to perform preprocessing of all spheres followed by the depth sorts and calculation of tile short lists; no parallelism has yet been obtained from these parts of the calculation. After the scene has been rendered then transformations of the component molecules or atoms can be applied without again broadcasting the atomic data.

Load balancing can be achieved by using a scattered domain decomposition of

tiles across processors, or dynamic farming of tile numbers. A regular domain decomposition will not suffice due to the distribution of work across the viewing plane, as depicted in Figure 4.3. In this diagram the shaded region represents the pixel area covered by the molecule and it is easy to see that there is a very large variation in the workload allocated to each of the 25 render tasks. These load balancing techniques are general purpose methods for handling a space over which the computational effort is not evenly distributed and the effort associated with each point is not known in advance. The load balancing problem in which the effort associated with each point is known, or can be estimated, admits of other solutions.

Since each render task holds the short lists for each tile it can immediately obtain an estimate of the computational effort of painting the tile. We choose two ranks of effort; let E be the number of "easy" tiles upon which no spheres impinge and H be the number of "hard" tiles upon which one or more spheres impinge. We attempt to separately distribute tiles from E and from H across render tasks such that each render paints $\frac{E}{P}$ easy and $\frac{H}{P}$ hard tiles. The easy tiles all involve an identical computational effort whereas the variance of the effort painting hard tiles may be quite large.

Increasing $\frac{N_t}{P}$ decreases the sampling variation experienced at each processor, thus decreasing the probability of load imbalance. On the basis of this observation we would choose the number of tiles to be larger than that used in the sequential algorithm. There is however a competing effect induced by the message passing architecture; choosing small tiles requires more message send events to be scheduled and a higher *message header* to *message data* ratio, effectively reducing the communication bandwidth into the graphics processor. This is a manifestation of the general grain size optimisation problem which is a common feature of parallel applications.

The proposed decomposition of the problem naturally leads to an execution time which scales as $A + \frac{B}{P} + CP$, where A is the time taken for preprocessing, depth sorting and tile listing, B is the time taken for tile painting in the sequential program, and CP is the time taken to synchronise each render task with the control

task. Note that C is very small compared to A, B. A "law of diminishing returns" reduces the effectiveness of increasing the number of processors used; this is a general feature of applications in parallel processing. In order to extend the regime of effective processor array sizes we must minimize the sequential component of a calculation; in this case we should also address the problem of parallelising the first three components of the algorithm.

The communication structure of this program is exceptionally simple since the only communication events which occur are broadcasts from the control process to the render processes and sends from the render processes to the graphics process. In order for the control process to determine that the image has been rendered it will also be necessary for a communication structure in which the graphics process sends a message to the control process upon frame completion. Each of these three structures defines an acyclic graph of links and there is no temporal overlap between the structures, it follows that the cyclic routing function can be used without any possibility of deadlock.

Depth Sorting

The efficient parallelisation of sort algorithms is an ongoing research topic and we shall not attempt to solve this problem here. The sequential algorithm sorts all spheres before calculating tile lists; in this way the tile lists are automatically ordered as required. We observe that one can equally sort each tile list individually without performing a global sort beforehand. This has the advantage that each render then sorts only the lists for tiles which it is about to paint and the workload of sorting is distributed in the same way as the workload of painting.

We can perform an approximate analysis of the difference between these two approaches. The average number of spheres in a tile list is $\frac{N_i}{N_t}$, the sorting of which takes a time proportional to $\frac{N_i}{N_t} \log \frac{N_i}{N_t}$. In a sequential program the total time spent sorting is therefore approximately $N_i \log \frac{N_i}{N_t}$ which is larger than the global sort taking a time proportional to $N_s \log N_s$. In a parallel program the average time a render spends sorting is $\frac{N_i}{P} \log \frac{N_i}{N_t}$ which is always smaller than $N_s \log N_s$ provided

N_s	Process + List	Paint + Sort
500	0.22s	33.9s
1000	0.43s	43.2s
1500	0.64s	63.7s
2000	0.85s	81.7s

Table 4.2: Performance of Modifed Sequential Program

$P \geq \frac{N_i}{N_s}$.

We altered the sequential implementation so that depth sorting was handled in this way and timed execution for the same problems as those analysed in Table 4.1. The results of these timings are shown in Table 4.2.

The time taken to execute the four stages is increased, as we had anticipated, however the strictly sequential part of the method is smaller. In actual fact the combined preprocessing and tile listing stages are faster in the above table than Table 4.1 since some optimisation is possible as the transformation and projection of a sphere can now be performed within the same loop.

Preprocessing

During the preprocessing algorithm each sphere can be transformed independently. We can trivially parallelise these transformations by assigning $\frac{N_s}{P}$ transformations to each render task. The tile painting decomposition above requires each render task to know the object and shadow space positions and radii of each atom; the communication problem is similar to that found in a systolic loop approach to molecular dynamics. We define a logical ring through the render tasks, which we arrange in the configuration to correspond to a physical ring, and distribute the transformed spheres by passing data around this ring.

If we do not attempt to overlap communication with calculation then each render task simply transforms its set of spheres and thereafter sends to the next render,

and receives from the previous render, P blocks of transformed spheres.

The execution time of this algorithm varies as $a\frac{N_s}{P} + bN_s + cP$ compared to aN_s in the sequential case. In this equation a is the time taken for a transformation, b is the time taken to transfer the positions and radii to the next neighbour in the ring and c is the overhead in scheduling P message send and receive operations. Examining Table 4.1 we find that $a = 140\mu s$ whereas we anticipate $b = 22.4\mu s$ and $c < 100\mu s$, see Chapter 3.

Tile Listing

We can consider parallelising the tile listing calculation by allowing each render to calculate lists for the $\frac{N_s}{P}$ atoms which it has also transformed. This approach would require us to use a scheme similar to the above in order to to distribute these partial lists among all renders. The quantity of data communicated in this case is at least $N_t + N_i$; given the results of Table 4.1 this far outweighs the effect of distributing the initial tile intersection calculations.

It is also possible to consider distributing this calculation on the basis that individual tile lists can be calculated concurrently when the sphere projections are repeated at every render. An examination of the second table of Table 4.1 indicates that this might be a preferable scheme provided that each render task performs the projection calculation only once. The amount of data to be communicated is again at least $N_t + N_i$ which prohibits use of this method.

In both the above schemes the gains of distributing the computation were outweighed by the requirement to distribute the results among the render tasks. It would appear that we can avoid the need to distribute these results provided that we adopt a static assignment of tiles to renders in which case each render calculates lists and paints such tiles; this would suggest that we use a static load balancing technique as opposed to the dynamic method described. A further breakdown of the tile listing algorithm reveals that it will only be efficient to distribute in this way if the assignment of tiles is a regular domain decomposition of the tile mesh so

that each render task clips to its own rectangular area of the eye plane.

In view of these considerations we do not find a satisfactory method for distributing this component of the algorithm. We can exploit the concurrent operation of the transputer links and processor by arranging for this calculation to overlap the communication of transformed atomic data in the preprocessing calculation; the tile listing calculations take rather longer than the communication of data.

Performance Characterisation

We have implemented the parallelisation strategies discussed for the tile painting and depth sorting stages of the algorithm. The distribution of preprocessing calculations was not implemented; due to the fact that according to Table 4.1 a gain of no more than 0.3s is anticipated which represents only approximately 15% of the anticipated execution time in a target configuration of no more than 64 render tasks. The schema of the parallel implementation is given in Figure 4.4.

We have measured the performance of this adaption, with actual display of the image, for varying N_s and P using the same molecules as we had studied in characterisation of the sequential algorithm. The results of these calculations are shown in Figure 4.5. It is immediately clear that a significant increase in performance over the sequential implementation is obtained as the execution time for the scene of $N = 2000$ is approximately 2.14 s on 64 processors compared to about 83 seconds in the sequential case.

Our model of the program indicates that we anticipate the execution time will scale as $a + \frac{b}{P}$ where a, b are related to the times reported in Table 4.2. In Figure 4.6 we plot the execution time against $\frac{1}{P}$ which indicates that such a relationship is obeyed.

We expect the actual parameters to be larger on account of the messages sent to the graphics processor; a is increased by a' the maximum message through routing overhead, and b is increased by b' the accumulated message send overhead. We

```
/* Preprocessing and Tile Listing */
LOOP over all spheres in ''own'' block
  Preprocess sphere
  Project Sphere onto eye plane and clip
  Append sphere to lists of intersected tiles

/* Tile Painting and Depth Sorting */
Let Easy and Hard be zero
LOOP over all tiles
  IF one or more spheres impinge on tile
    Increment Hard
    IF Remainder(Hard,P) equals p
      Paint pixels of tile
      Use Draw Block to set tile to calculated pixels
  ELSE
    Increment Easy
    IF Remainder(Easy,P) equals p
      Use Clear Block to set tile to background colour
Signal control that render has finished
```

Figure 4.4: Schema of Parallel Algorithm

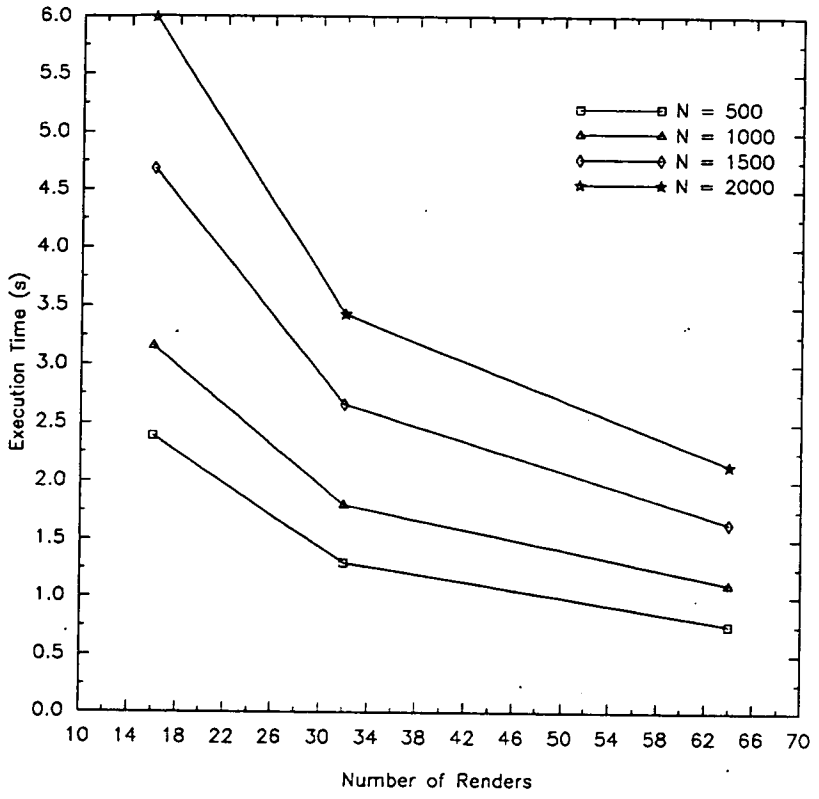

Figure 4.5: Execution Time of Parallel Program

have previously estimated $a' = 0.036s$. The processing overhead of a message send is taken to be approximately 50 μs; this is derived from the data presented in Chapter 3 coupled with the additional overheads of the FORTRAN procedural interface. We then obtain, under the same assumptions as those used in estimating a', $b' = 0.077s$. These numbers are too small to be accurately resolved from the inevitable "noise" in execution time data.

4.4 Summary

In this chapter we have described a simple but instructive application of the communication system discussed in Chapter 3. The program concerned implements an algorithm for rendering molecular images and the sequential execution time was too long for "interactive" use.

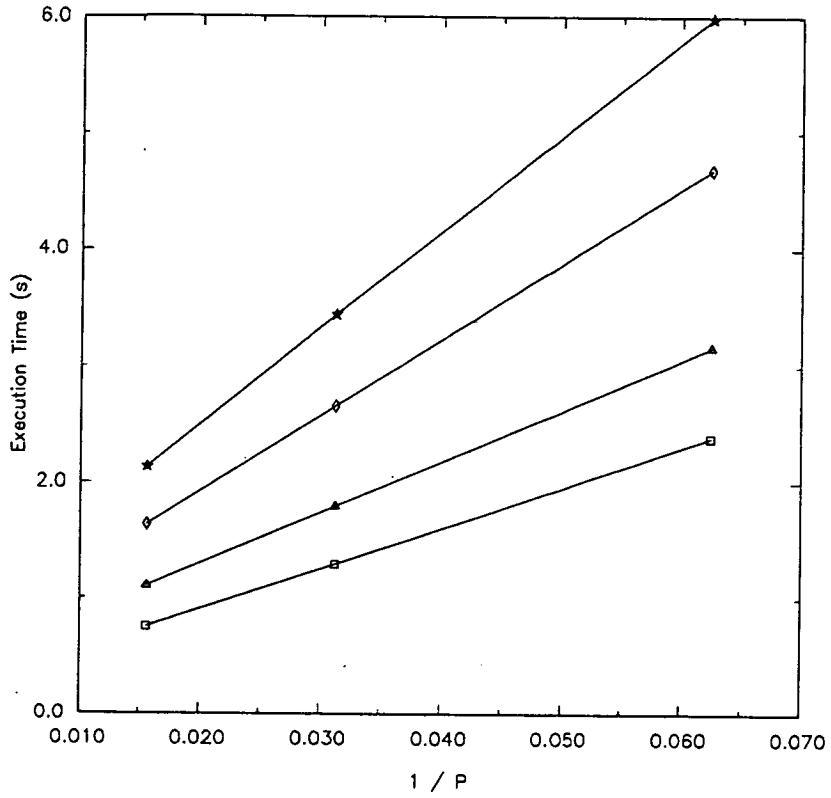

Figure 4.6: Scaling Behaviour of Execution Time

The algorithm was found to decompose into four components; processing, depth sorting, tile listing and painting. The most time consuming of these components, the painting operation, was effectively parallelised using a simple dynamic domain decomposition method, based on an estimator of the computational effort associated with domains provided by a preceding component. It was also convenient to modify the sequencing of components in order to exploit parallelism in the depth sorting operation.

It was not possible to find a satisfactory method for execution of the tile listing operation which is therefore a sequential bottleneck in the parallel adaption. Despite the fact that the final component, the preprocessing operation, is susceptible to parallel execution this was not implemented since it would provide only small gains and the FORTRAN programming environment offered poor support for the necessary concurrent structures.

A significant decrease in execution time was obtained for up to 64 processors. The

parallel performance was modelled by a simple equation and the effect of message passing on the parameters of this equation was examined. The additional overheads due to the message passing architecture and communication system were found to be negligible.

Chapter 5

Epilogue

In Chapter 2 we have discussed a number of issues pertaining to the design of communication systems for distributed memory MIMD multicomputers. The future of these machines certainly involves the provision of message through-routing in specialised hardware. In such cases it is prohibitively expensive to assign large amounts of memory to the routing hardware which indicates that circuit or wormhole switching will be used. This observation also largely precludes the utilisation of a routing table, which may limit the applicability of the methodology for construction of acyclic routing functions. The considerations of the application interface are not affected by the transferral of network into hardware.

We have implemented such a system for processor arrays composed of INMOS T800 transputers. The techniques used in this implementation, which are a specialisation of those previously discussed, were described in Chapter 3. The message passing overheads of the system were examined and found to be very small, comparing favourably with a number of hypercube multicomputers. This software has been incorporated into a large number of application from diverse fields; neural networks, genetic algorithms, computer chess, image processing, molecular dynamics, cellular automata, device simulation ...

We discussed in Chapter 4 a simple application which uses this system, in the field of molecular graphics, in which we examined the effect of communication overheads on the parallel adaption and found them to be negligible. In the absence of an "off

the shelf" communication system the development of this application would have been considerably impeded by the requirement to implement message routing.

In this thesis we have constructed a general purpose communication system. There is also some justification for the construction of paradigm specific communication systems provided that said system can certainly support the paradigm more efficiently than the general purpose system. This is the case in the fine grained task farm paradigm and I have also been involved, in collaboration with Dominic M. N. Prior, in implementing such a system. This work made a large contribution to the implementation of the flood algorithm for topology eploration in TINY. Further work, in collaboration with Dominic M. N. Prior, Michael G. Norman and Nicholas J. Radcliffe, addressing the issue of topologies for multicomputer machines, is reported in [PNRC90].

The experience gained in the above work has, not surprisingly, also found useful applications. Since completing this work I have been involved in the adaption of a molecular dynamics application for the simulation of large bio-polymers developed in [HGS89]. It is intended to incorporate the rendering capability of the molecular graphics algorithm described above into this program, allowing the biochemist to observe atomic evolution in real computational time.

The project initially began as a simple port of an occam program from one transputer based machine to the another. Assessing the performance of this program on a larger transputer array (64+ processors) than the development machine (16 processors) revealed poor scaling behaviour; the execution time began to increase at large enough array sizes. This was easily traced to a feature of the communication structure in which the time taken to distribute the partial forces acting on the N atoms around the system of P processors increased with NP whereas this operation can be performed in a time increasing with N alone. During this investigation a source of deadlock in the communication structure was observed and a simple solution implemented using ideas discussed in Section 2.4. The resulting performance was improved by up to a factor of 5 on a machine composed of 80 processors.

The Hubbard model project mentioned in Chapter 1 was revisited with the benefit

of knowledge gained in this work. The difficult global summation problem was reconsidered in the light of our understanding of the transputer and a different approach taken. In the previous approach the accumulation of partial sums had been arranged to proceed at all nodes, firstly summing over the columns and then over the rows, taking care to ensure that the same result will arrive at every processor upon termination of the summation. Of course it is also possible to define a tree through the toroidal geometry and arrange to compute partial sums at the nodes of this tree, the final summation being completed at the root and thereafter broadcast back down the tree. Where summations are occuring in rapid progression then there is some advantage in alternating the root node between opposite poles of the torus which creates a pipelined effect on tori with edge lengths greater than 4. When it is possible to find calculations which can overlap the global summation this transpires to be a better approach. In the Hubbard model calculation there are two matrices to invert, one for each electron spin, and we can overlap the calculations of one inversion and the communications of another, with careful programming. It was also possible to assembler code a number of the critical routines, e.g. `sdot` and `saxpy`, to obtain a sequential performance of approximately 1 Mflop. The current implementation of this application is now able to use the smallest possible domain size, a single lattice site, and our model indicates that performance will be sustained at approximately 0.8 Mflops/node on arrays of up to 256 transputers — it has recently become possible to configure the ECS into such an array.

These two examples illustrate that the deeper understanding of general issues in parallel performance is also directly beneficial in specific cases.

Bibliography

[AH89] R. J. Allan and E. L. Heck. Fortnet: a parallel Fortran harness for porting application codes to transputer arrays. In L. M. Delves, editor, *Proc. Int. Conf. on Applications of Transputers*, August 1989.

[BA88] D.J. Bacon and W.F. Anderson. A fast algorithm for rendering images of solid objects with shadows, and its application in making pictures of molecules. Manuscript in Preparation, 1988.

[BKW88] Ken C. Bowler, Richard D. Kenway, and David J. Wallace. The Edinburgh Concurrent Supercomputer: Project and applications. In *I.E.E. Conference on the Design and Application of Parallel Digital Processors*, April 1988.

[BR89] Luc Bomans and Dirk Roose. Benchmarking the iPSC/2 hypercube multiprocessor. *Concurrency: Practice and Experience*, 1(1):3–18, September 1989.

[DKPR87] Simon Duane, A. D. Kennedy, Brian J. Pendleton, and Duncan Roweth. Hybrid monte carlo. *Physics Letters B*, 195(2):216–222, September 1987.

[DS88] William J. Dally and Charles L. Seitz. The torus routing chip. *Distributed Computing*, 1:187–196, 1988.

[DZ83] J. D. Day and H. Zimmerman. The OSI reference model, December 1983.

[Fox89] Geoffrey C. Fox. Parallel computing comes of age: supercomputer level parallel computations at Caltech. *Concurrency: Practice and Experience*, 1(1):63–103, September 1989.

[FvD82] J. Foley and A. van Dam. *Fundamentals of Interactive Computer Graphics.* Addison Wesley Publishing Company, 1982.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[Gun81] K. D. Gunther. Prevention of deadlocks in packet switched data transport systems. *I.E.E.E. Transactions on Communications*, 29(4):512–524, 1981.

[HGS89] Helmut Heller, Helmut Grubmüller, and Klaus Schulten. Molecular dynamics simulation on a parallel computer. Submitted to Molecular Simulation, 1989.

[Hlu88] Micheal G. Hluchyj. Queueing systems in high performance packet switching. *I.E.E.E. Journal on Selected Areas of Communications*, 6(9):1587–1597, December 1988.

[Int86] Intel iPSC system overview. Intel Scientific Computers, 1986. Order Number 310310–001.

[LB85] D. P. Landau and K. Binder. Phase diagrams and critical behaviour of Ising square lattices with nearest-, next-nearest-, and third-nearest-neighbour couplings. *Physical Review B*, 31(9):5946–5953, May 1985.

[LS86] D. P. Landau and R. H. Swendsen. Monte carlo renormalization-group study of tricritical behavior in two dimensions. *Physical Review B*, 33(11):7700–7707, 1986.

[Mcn] N. McNally, University of Southampton. Private Communication.

[MS80] P. Merlin and P. Schweitzer. Deadlock avoidance in store and forward networks. *I.E.E.E. Transactions on Communications*, 28(3), 1980.

[NLW88] D. A. Nicole, E. K. Lloyd, and J. S. Ward. Switching networks for transputer links. In *8th Occam User Group on Developments using occam*, 1988.

[Nug88] S. F. Nugen. The iPSC/2 direct connect communications technology. Intel Scientific Computers, 1988. Order Number 280115-001.

[NW88] Michael G. Norman and Sam Wilson. TITCH: *Topology Independent Transputer Communications Harness.* Edinburgh Concurrent Supercomputer Project, 1988.

[PNRC90] Dominic M. N. Prior, Michael G. Norman, Nicholas J. Radcliffe, and Lyndon J. Clarke. What price regularity? *Concurrency: Practice and Experience*, 2(1):55–78, March 1990.

[Pou90] Dick Pountain. Virtual channels: The next generation of transputer. *BYTE Magazine*, April 1990. Appeared in the *Europe and World* section.

[Ros87] A. W. Roscoe. Routing messages through networks: An exercise in deadlock avoidance. In T. Muntean, editor, 7^{th} *Occam User Group on Parallel Programming of Transputer Based Machines*, September 1987.

[Sei85] Charles L. Seitz. The cosmic cube. *Communication of the ACM*, 28(1):22–23, January 1985.

[Sur] M. W. Surridge, University of Southampton. Private Communication.

[Sur90] M. W. Surridge. ECCL a general communications harness and configuration language. In D. J. Pritchard and C. J. Scott, editors, *Proceedings of the Second International Conference on Applications of Transputers*, pages 341–354, 1990.

[Tan89] Andrew S. Tannenbaum. *Computer Networks.* Prentice-Hall International, second edition, 1989.