

997

**DESIGN OF A USER FRIENDLY
COMMUNICATION SYSTEM
FOR A DISTRIBUTED MEMORY
PARALLEL COMPUTING SYSTEM**

DISSERTATION SUBMITTED BY

VIKAS AHLUWALIA

IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF


**MASTER OF TECHNOLOGY
IN
COMPUTER SCIENCE AND TECHNOLOGY**

SCHOOL OF COMPUTER AND SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI
JANUARY 1993

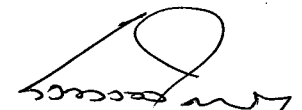
CERTIFICATE

This is to certify that the dissertation entitled "Design of a user friendly communication system for a distributed memory parallel computing system", being submitted by me to Jawaharlal Nehru University, New Delhi in the partial fulfilment of the requirements for the award of the degree of **Master of Technology**, is a record of original work done by me under the supervision of **Dr. P. C. Saxena**, Associate Professor, School of Computer and System Sciences, Jawaharlal Nehru University during the year 1992, Monsoon Semester.

The results reported in this dissertation have not been submitted in part or full to any other University or Institute for the award of any degree or diploma, etc.




Vikas Ahluwalia



Prof. R. G. Gupta
Dean,
School of Computer and System Sciences,
J. N. U., New Delhi.

4/11/93



Dr. P. C. Saxena
Associate Professor,
School of Computer
and System Sciences,
J. N. U., New Delhi.

ACKNOWLEDGEMENTS

I express my sincere thanks to Dr. P. C. Saxena, Associate Professor, School of Computer and System Sciences, Jawaharlal Nehru University, New Delhi for suggesting such a brilliant topic. I am indebted to him for his personal involvement with my work and his immense and eloquent guidance which has been indispensable in bringing about a successful completion of the dissertation.

I am also grateful to Dr. Saxena for providing me with his invaluable notes and papers related to the topic and also guiding me in my lookout for proper references.

I extend my sincere thanks to Prof. R. G. Gupta, Dean, School of Computer and System Sciences, Jawaharlal Nehru University for providing me with the environment and all the facilities required for the completion of my dissertation.

I also take this opportunity to thank all faculty and staff members and my friends who helped me in every way possible.



Vikas Ahluwalia

Contents

1.	Introduction.3
1.1	Parallel Processing.	
1.2	The Problem with Parallelism.	
1.3	Transputer based Parallel Computing System	
1.4	Relevance of this project.	
1.5	Organization of this report.	
2.	Parallel Architectures.9
2.1	Classification of Parallel Computers.	
2.2	Transputer based Parallel Computer.	
2.3	Topologies for a Transputer based Parallel Computer.	
3.	Programming Support for Parallel Systems.33
3.1	Interprocess communication and synchronization.	
3.2	Software tools for programming languages.	

4.	TupleSpace.42
4.1	Tuples.	
4.2	Implementation of Master Slave Paradigm.	
4.3	Implementation of Reader Writer Problem in TUPLE_IO.	
4.4	Issues in distributing Tuple Space.	
4.5	Representation of Tuple Space on Transputer based Parallel Computer.	
5.	Implementation.57
5.1	Preprocessor.	
5.2	Code Generator.	
5.3	Distributor.	
5.4	Main features of this communication harness.	
6.	Conclusion.71
	Bibliography75

CHAPTER 1

INTRODUCTION

1.1 Parallel Processing

Parallel Processing is universally accepted as the only answer to advanced computing requirements in science and engineering. It is only recently that the parallel computers are being procured not only as an add on to a computing center but for serious computations like a supercomputer.

There are two major motivations for creating and using parallel computing architectures. The first is that parallelism is the only avenue to achieve vastly higher speeds than are possible now from a single processor. A second motivation for the use of a parallel architecture is that this should be considerably cheaper than the sequential machines for systems of moderate speeds, that is, not necessarily supercomputers but instead minicomputers or mini-supercomputers be cheaper to produce a given performance level than the equivalent sequential systems. The goal of research in parallel computer architecture has been to achieve

price/performance through the use of parallelism than would be possible from sequential machines.

1.2. The Problem with Parallelism

The efficiency with which we can exploit the potential parallelism in a given application is directly related to the hardware, algorithm and programming language used. Unfortunately, the greater the potential gains from parallelism, the more difficult it becomes to realize these gains. For example, the larger the number of independent processing elements at our disposal the greater the communication overhead penalty incurred by the necessity to pass data between them.

Although multiprocessor machines are becoming widely available, and offer potentially impressive cost to performance ratios, they are as yet user unfriendly environments. In order to provide good user support to an operating system level, system software should allow efficient machine utilization without the need for the user to tailor his program to suit the machine architecture.

A good generalization can be made that there is good software on low and medium performance systems such as Alliant, Sequent, Encore and Multiflow

to suit the machine architecture.

A good generalization can be made that there is good software on low and medium performance systems such as Alliant, Sequent, Encore and Multiflow systems, while there is poor quality software in the highest performance systems.

The system software provided with the high performance parallel computers is at best that which would be suitable for systems that would be used by a single person or a small, tightly knit group of people.

1.3 Transputer based Parallel Computing System

The INMOS Transputer has been acclaimed across the world as the genesis of parallel processing. The T800 transputer has a 32-bit RISC CPU, a floating point unit, 4KB of fast static RAM and 4 bi-directional communication links. It can support 4 GB of external memory. Theoretically, there is no limit to the number of transputers that could be linked.

The transputer based parallel computer consists of a transputer plug-on board on a PC-AT host

machine. Each transputer has about 2 MB of local memory.

1.4 Relevance of this project

In this project, a user friendly communication harness has been developed on a transputer based parallel computing system. This model makes inter-process communication transparent to the user. The data can be communicated between processes on same or different processors.

In keeping with the requirement for a user friendly communication service, the aim of this project has been to :

Provide automatic synchronization of communicating processes which are resident on different processors.

Deliver messages to each user process as they are required so that no restrictions are placed on the user as to the order in which the messages are read.

Remove the burden of learning the message passing techniques of communication from the programmer. Instead the programmer works with

simple constructs.

Remove the connectivity problem, which the programmer faces in order to communicate between two processes on two different processors.

1.5 Organization of this report

Chapter 2 discusses the parallel computer classification, the INMOS transputer and the topologies in which a parallel computing can be configured using a transputer.

Chapter 3 describes programming support for parallel systems.

Chapter 4 describes tuple space, its operations, and distribution in a transputer based Parallel Computer.

Chapter 5 gives a detail of the implementation of this project. It describes the communication harness in detail.

Chapter 6 describes the objectives achieved by this project and suggests areas of future research in this field.

CHAPTER 2

PARALLEL ARCHITECTURES

2.1 Classification of Parallel Computers

Computer architectures have been classified by Flynn on the basis of instruction and data streams. An architecture can be classified by the multiplicity of hardware used to manipulate instruction and data streams. Given this possible multiplicity, the following four classes of computers result :

- (1) **Single Instruction stream, Single Data stream (SISD).**

Sequential computers fall in this category. Although instruction execution may be pipelined, computers in this category can decode a single instruction in unit time. An SISD computer may have multifunctional units but these are under the control of a single unit.

(2) Single Instruction stream, Multiple Data stream (SIMD).

A processor array, which executes a single stream of instructions, but contains a number of arithmetic processing units, each capable of fetching and executing its own data. Hence at any instance of time, a single operation is in the same state of execution on multiple units, each manipulating different data.

(3) Multiple Instruction stream, Single Data stream (MISD).

No computers fit into this category.

(4) Multiple Instruction stream, Multiple Data stream (MIMD).

This category contains most multiprocessor systems and the parallel computer used for the development of this project too falls in this category.

MIMD architectures can be further classified into :

1. **Shared Memory.** In this class of computers, all CPUs share the same (global) memory. Each CPU may have a small amount of cache memory. Sharing of main memory is achieved by different techniques, the prominent ones being the **Shared bus** and **Switched memory.**

i) In **shared Bus**, it is necessary to have a bus arbiter to resolve possible conflicts.

ii) In **Switched Memory**, the shared memory is divided into a number of modules which are switched among the CPUs by a global switching network.

2. **Distributed Memory.** All the processors have their own (local) memories. There is no global memory. Hence all inter-processor communication is by message passing.

The advantage of distributed memory systems over shared memory systems is that in the latter, the memory bandwidth available is the actual memory bandwidth shared by all the processors, whereas in

the former, the available memory bandwidth is the total of all the individual memories. However, in distributed memory systems, there is the communication overheads of message passing.

2.2 Transputer based Parallel Computer

This project has been implemented on a transputer based parallel computer. The transputer based parallel computer consists of a transputer plug-on board on a PC-AT host machine. The transputer board has 1, 4, 16 or 64 transputers inter-connected by one of the common topology (described in this chapter). Generally each transputer has a minimum of 2MBytes of local memory. Compilers and debugging tools for C, Fortran and C++ for transputer are provided by many vendors. The mathematical libraries are too becoming popular.

The transputer boards are typically used as computation engines attached to a PC-AT machine. Most parallel computers based on the transputer have found acceptibility among the scientists and researchers working in computation extensive areas

of stellar dynamics, simulation of neural networks, image processing, computational aerodynamics, etc.

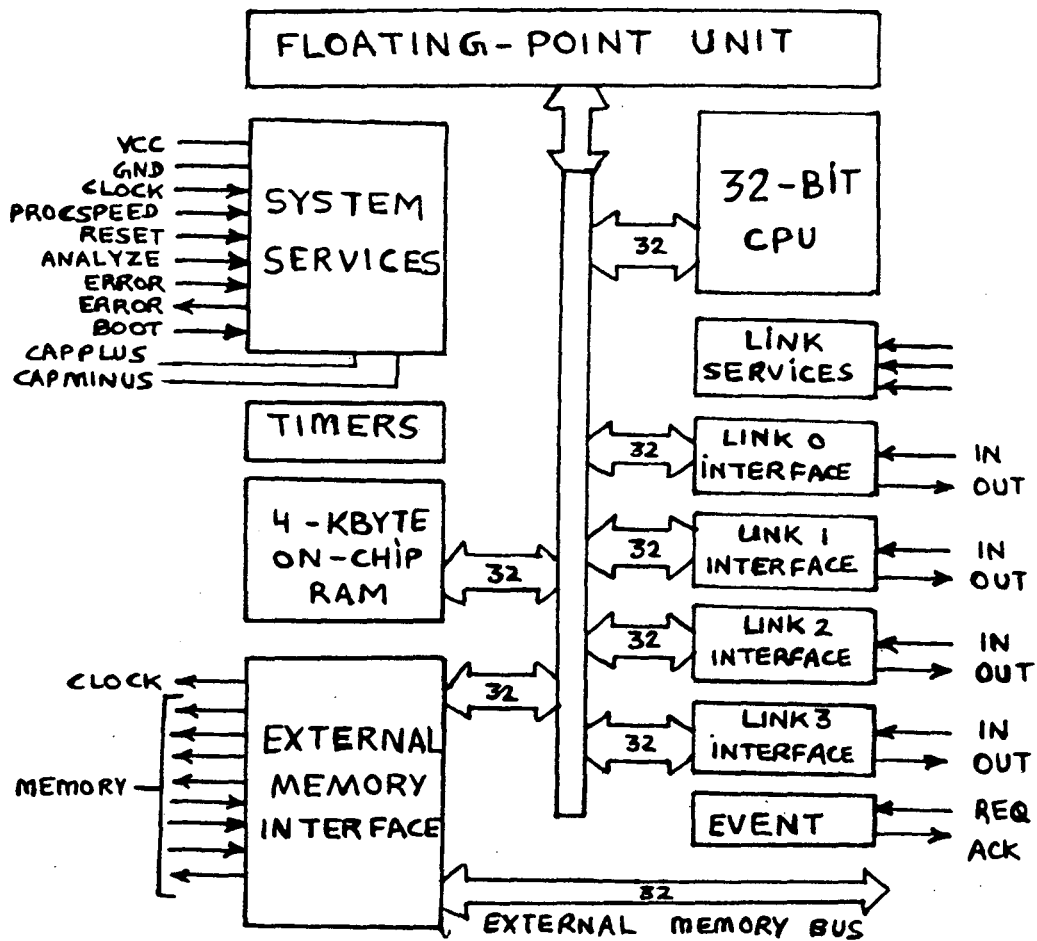
This chapter discusses the main features of transputer, its communication mechanism, topologies in which a transputer based Parallel Computing System can be configured.

2.2.1 Transputer as a Processing element

The Transputer (TRANSSistor comPUTER) has been developed by INMOS Ltd, U.K. as a parallel processing element. See fig. 2.1. It supports concurrent programming and message passing by explicitly defined channels or links providing a direct implementation of the concept of communicating sequential processes suggested by C.A.R.Hoare.

A transputer has the following components :

- i) A RISC CPU.
- ii) Fast on-chip SRAM.
- iii) Bi-directional serial communication links, which operate concurrently with the CPU and with each other.



T800 ARCHITECTURE BLOCK DIAGRAM

FIG. 2.1

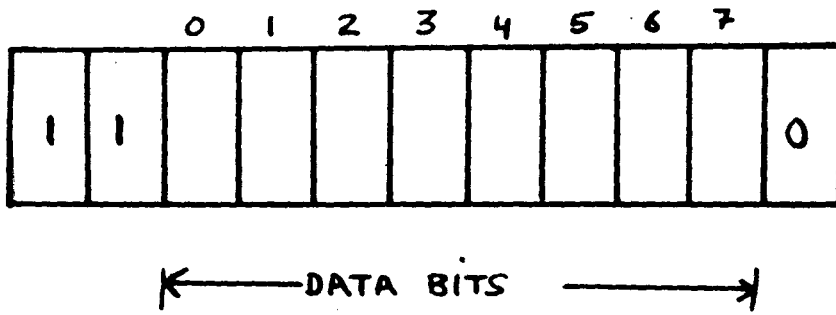
The INMOS chip, T800 has a 32-bit RISC CPU, 4KB of on-chip RAM, four links and also an on-chip FPU. It supports 4GB of external memory. The T800's links can transmit data at the rate of 1.8 MBytes/sec in one direction, or 2.4 Mbytes per second overall in both directions. The T800 has a rated performance of 1.5 MFLOPS (T800-20 MHz) and 2.25 MFLOPS (T800-30 Mhz).

Due to its low price/performance ratio, support for concurrent programming and simple expandability of hardware, the T800 has been chosen as the processing element by CDAC, Pune for their PARAM project. In India, I.I.T Delhi and B.A.R.C., Bombay are also evaluating T800 and many research teams are working on it.

The three components of the T800 can operate concurrently with each other. Once a message to be transmitted through a link is set up, the link operates independently of the other parts of the CPU. The transputer supports point-to-point communication. The advantages of this type of communication are:

1. No contention for the communication mechanism.
2. No capacitive load penalty as transputers are added to the parallel computer.
3. Communications bandwidth does not saturate as the size of the system increases. The communication bandwidth increases as the number of transputers increase.

Each link provides two channels one in each direction. At each end of the channel, synchronization of processes is automatic and does not require explicit programming. If one end of the channel (A) is ready, and the other (B) is not, then the process A is descheduled from the process queue. A descheduled process does not consume CPU time. When B is ready, process A is executed. This method is adopted to remove the need for message buffers. The message is transmitted as a sequence of bytes. After sending a byte, the sending transputer waits for an acknowledgement(ACK). Refer fig 2.2. The receiving transputer sends an ACK as soon as it starts receiving a data byte. No check is made to see if the data byte has arrived



DATA FORMAT

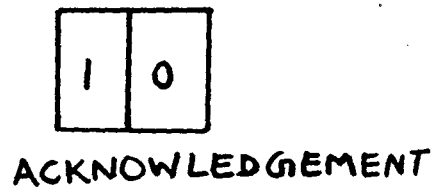


FIG 2.2 COMMUNICATION PROTOCOL
IN THE TRANSPUTER.

correctly. This strategy is adopted to reduce communication processing overheads.

Since all the transputers are mounted on the same board, the probability of data corruption is very low. If a message is sent, it will either be transmitted correctly or not at all. The protocol synchronizes communication of each data byte by sending two start bits and a stop bit with every byte. The ACK consists of one high bit followed by a low bit. The protocol is independent of the word length. If two processes sharing a channel are on the same transputer, the channel is mapped onto a memory location. For two processes on two transputers, the channel is mapped onto a hardware link.

The disadvantage of the transputer is that it does not have memory management, does not support multiple level priority interrupts and a process once executed cannot be removed involuntarily from the system. However, the last point is not a disadvantage, since the cost of moving an executing process is very high compared with the cost of moving one that has not yet started.

2.3 Topologies for a Transputer based Parallel Computer

The Transputer has four links for connectivity and this section discusses six regular topologies. The regular topologies are : Mesh, Torus, Binary Hypercube, Supernode based hypercube, W-K Recursive and pipeline.

(1) Mesh

This is considered to be the simplest of all the topologies, and the transputers are organized in rows and columns as in a square matrix. The transputers at the boundary are left with only a single free link except the corner ones have two free links. Numbering is done by moving sequentially down the rows from left to right. Refer fig. 2.3.

Path algorithm

Let,

col = no. of nodes along a row

src = source processor

dest = destination processor

Then,

src_x_coord = **src** mod **col**

src_y_coord = **src** / **col**

dest_x_coord = **dest** mod **col**

dest_y_coord = **dest** / **col**

hops = **abs** (**dest_x_coord** - **src_x_coord**)
+ **abs** (**dest_y_coord** - **src_y_coord**)

(2) Torus

This is similar to the mesh in that all the free links of the mesh topology are connected with each other. The free links of the transputers in the top row are connected to those of the bottom row in the same column. This leaves no free link. Refer fig. 2.4.

Path algorithm

Let,

col = no. of nodes along a row

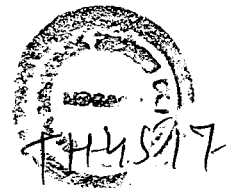
src = source node & **dest** = destination node

Then,

src_x_coord = **src** mod **col**

src_y_coord = **src** / **col**

dest_x_coord = **dest** mod **col**



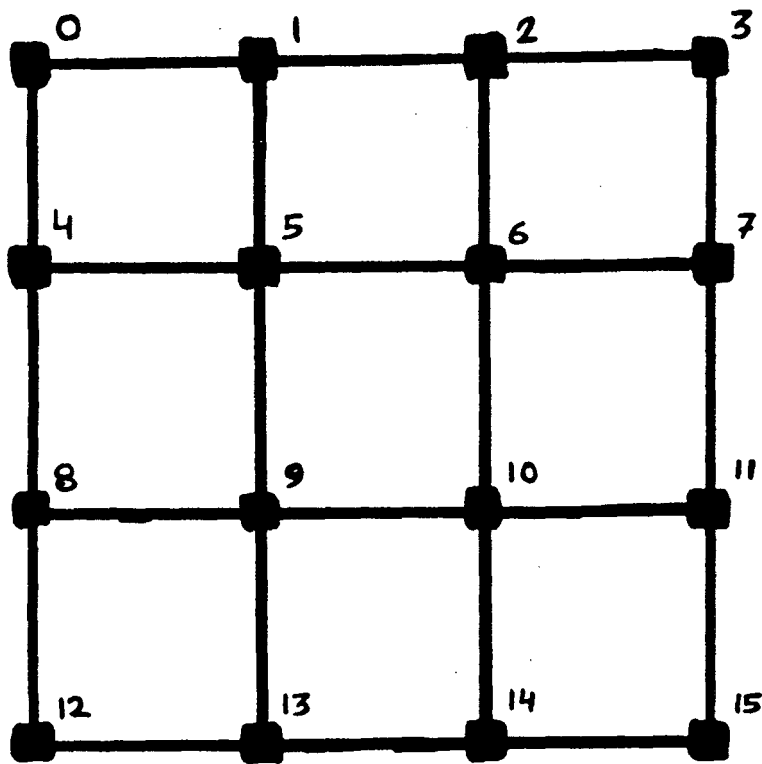


FIG 2.3 MESH TOPOLOGY

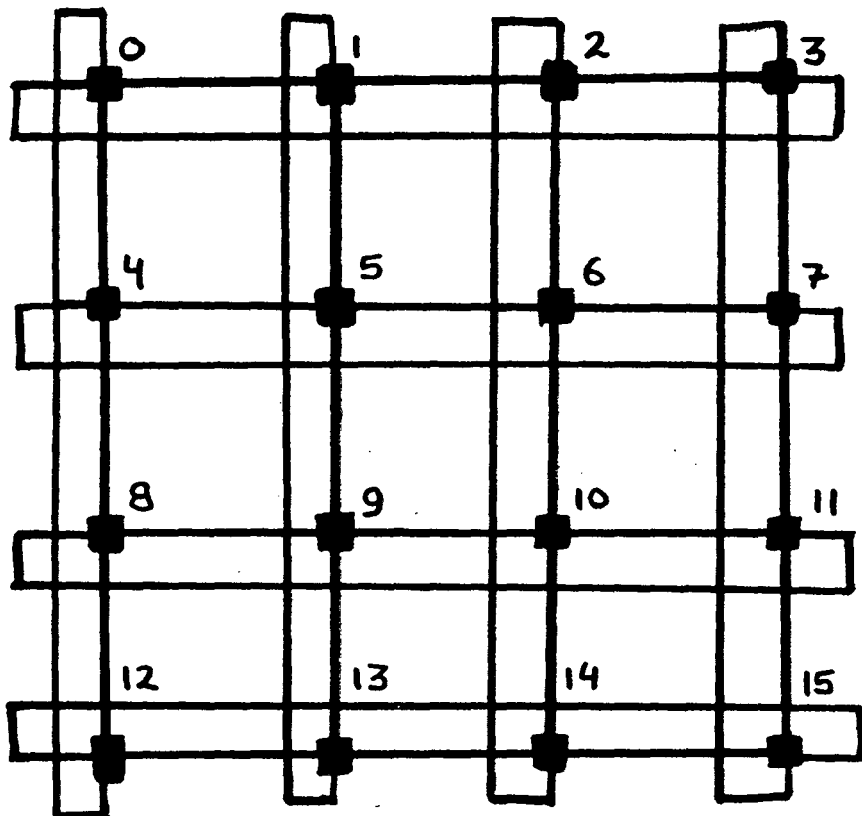


FIG 2.4 TORUS TOPOLOGY

```

dest_y_coord = dest / col

If abs(src_x_coord - dest_x_coord) > (col / 2)

then

    hops_x = abs ( abs (src_x_coord - dest_x_coord)
                  - (col/2) )

else

    hops_x = abs ( src_x_coord - dest_x_coord )

If abs(src_y_coord - dest_y_coord) > (row / 2)

then

    hops_y = abs ( abs (src_y_coord - dest_y_coord)
                  - (row/2) )

else

    hops_y = abs ( src_y_coord - dest_y_coord )

hops = hops_x + hops_y

```

(3) Binary Hypercube

The transputers are connected in the form of a cube. This cube can be of any dimension upto four and hence called a **Hypercube**. A hypercube of dimension k has 2^k nodes. Neighbouring transputers differ by one bit position in their address. Refer fig. 2.5. This is a completely connected topology.

Path algorithm

Let,

src = source node

dest = destination node

vall = **src** XOR **dest**

The **number of hops** between the two nodes is given by the number of bits are 1 in **vall** .

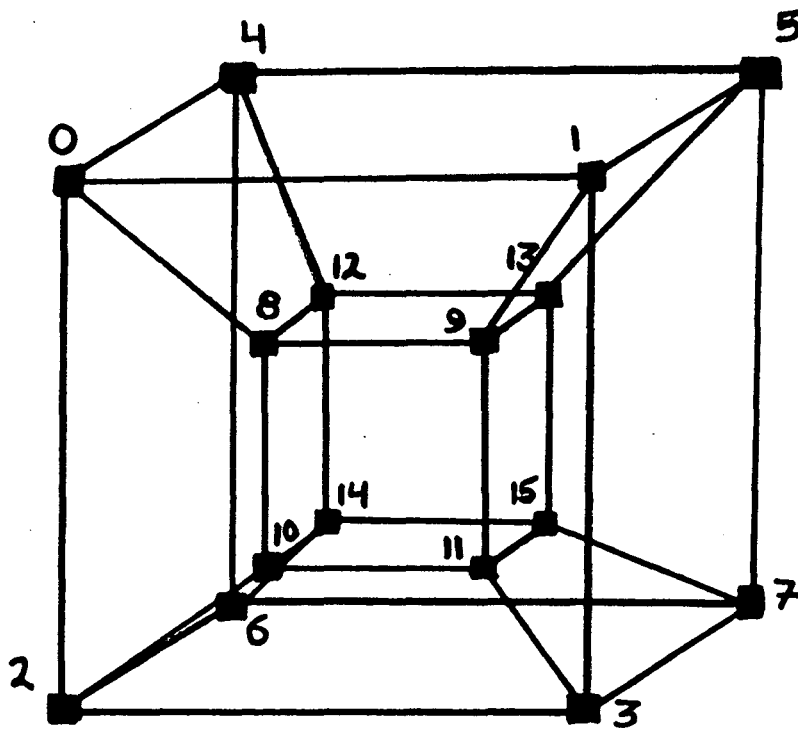


FIG 2.5 BINARY HYPERCUBE TOPOLOGY

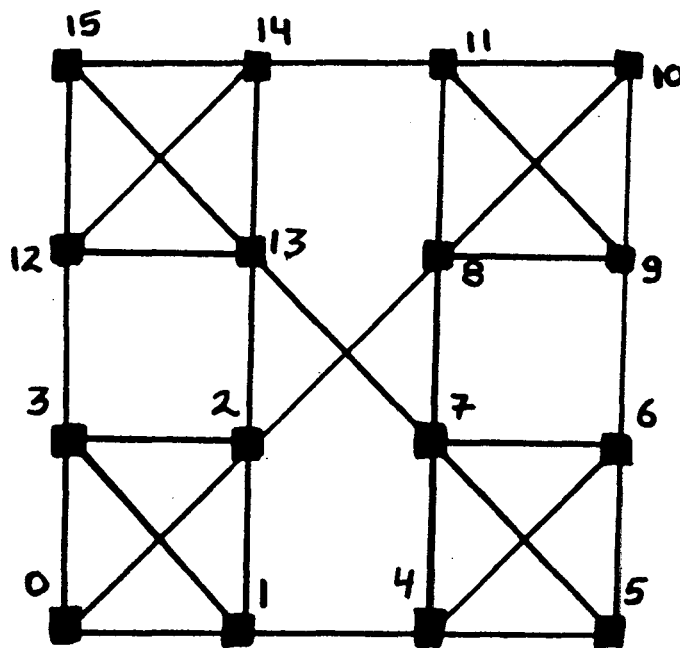


FIG 2.7 WK RECURSIVE TOPOLOGY

(4) Supernode based Hypercube

Each transputer in the hypercube is connected to a cube of eight transputers (known as supernode). All transputers of a supernode have a single free link, with the other three connected to other transputers in that supernode. This results in a supernode with 8 free links, permitting an eight dimensional supercube of hypernodes. The maximum possible transputers in such a combination is 2048 (8×2^8). Refer fig. 2.6. Since there are 8 nodes in a supernode, 3 bits are needed to specify a node in a supernode. If there are 8 supernodes, 3 bits are again needed to address a supernode. Within a supernode, numbering is done so that a node's neighbour differ in address by one bit. A free link in a supernode is numbered as per the node address. Supernodes are connected and numbered in the same manner as the nodes.

Path algorithm

Let,

n = dimension of the supernode.

src = source node

dest = destination node

Then,

- a) If **src** = **dest** then return pathlength.
- b) XOR the (n-3) most significant source and destination addresses. Let the result be **rel_addr1**.
- c) If **rel_addr1** = 0, then both lie in the same supernode. Find the number of hops between them by XORing **src** and **dest** and finding the number of bits that are 1. Add this value to existing value of path length and return.
- d) If **rel_addr1** \neq 0 then find the output link number (**op**).
- e) If **src**'s node number in a supernode is greater than that of **dest**, then find the position of the most significant non-zero bit in **rel_addr1** (**posn1**) and XOR **posn1** and 3 LSBs of **src**, to give **rel_addr2**. Otherwise XOR **op** and 3 LSBs of **src** to give **rel_addr2**.
- f) If **rel_addr2** = 0 it means that at the current node, there is an inter-supernode link. Jump across the supernodes. Otherwise calculate the position of the most significant non-zero bit in **rel_addr2** (**posn2**).
- g) Invert the bit number **posn2** in the 3LSBs of **src**.

h) Increment the pathlength and goto a).

(5) W-K-Recursive

This is a topology which can be recursively scaled. In the W-K-Recursive topology, at the lowest level (basic module) all the nodes are fully connected. Let the number of nodes be W . The link requirement of each node is W . We have $W=4$ free links at level 1. At the next level, W such modules are connected and number of free links is W . In this manner, a module of level k is built recursively from level $(k - 1)$ and needs W^k nodes. Refer fig. 2.7.

The numbering of the nodes is done as follows :
At every level, there are 4 transputers and 2 bits are needed for the address of each node at that level. Thus for a topology at level 3, three pairs of bits are used, one pair for every level. The most significant pair pertains to the highest level (logical) node of level 3. The next pair pertains to the 4 logical nodes of level 2 and the least significant pair, to the 4 physical nodes at level 1.

Path algorithm

In the lowest level of this topology, all the nodes are connected. The next level (level 2) maximum distance is 3, level 3 maximum distance is 7 and so on. Thus the maximum distance at any level can be given by $2^{\text{level}} - 1$. A jump in the lowest level can be made by changing either one or both of the last two bits. A jump at level 2 can be made by exchanging the bits for level 1 and level 2. A level 3 jump can be made by exchanging the level 3 bits with those for levels 2 & 1.

Let the number of levels in a topology be $n/2$. The number of bits required to address a node will be n . The number of nodes will be 2^n . Let **src** and **dest** be the source and destination nodes respectively.

If (**src** and **dest** are in different modules)

Move to the node which connects **src** and **dest**

Jump across to **dest** module

Move down the **dest** module to the **dest** node

This code is recursive and holds good for any level.

(6) Pipeline

The transputers are connected to one another to form a pipe. The transputer at the beginning and end of the pipe has 3 free links, while all others have 2 free links. Refer fig. 2.8.

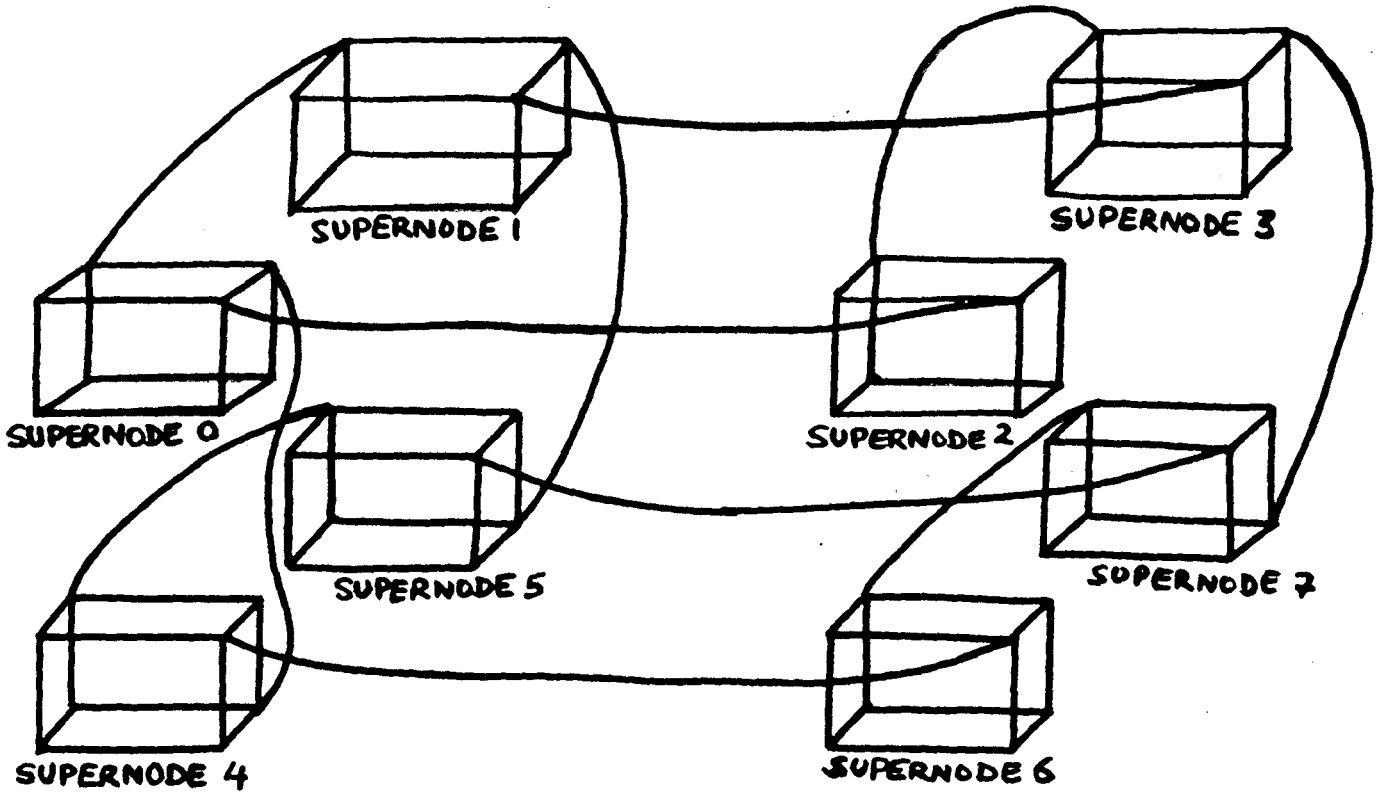


FIG 2.6 SUPERNODE BASED HYPERCUBE TOPOLOGY



FIG 2.8 PIPELINE

Path algorithm

Let,

`src` and `dest` are the source and destination processors. Then,

`hops = abs (src - dest)`

CHAPTER 3

PROGRAMMING SUPPORT FOR PARALLEL SYSTEMS

There are two main features that are expected in a parallel programming environment :

1. The optimal use of multiprocessors.
2. The cooperation among the processors.

Parallel applications execute processes of their code in parallel on one or different processors. High performance applications use this parallelism for achieving speedups. Here, the goal is to make optimal use of the available processors; this issue of load balancing is discussed in ;Vikas,GSS '91.

In parallel applications, the processors sometimes have to exchange intermediate results and synchronize their actions. In a system of automated factory, processors have to keep an eye on each other to detect failing processors.

Ideally, programming support for implementing the parallel applications must fulfill all of these requirements. This support is expected in the operating system or the parallel programming

language being used. In the first case, applications are programmed in a sequential language extended with library routines that invoke operating system primitives. As a disadvantage of this approach, the control structures and data types of the sequential language are usually inadequate for parallel programming.

3.1 Interprocess communication and synchronization

An important issue in the design of a language for parallel programming is how the pieces of a program which are running in parallel on different processors are going to cooperate. This cooperation needs two types of interaction among the communicating processes : communication and synchronization. For example, Process A may require data X which is the result of some computation performed by Process B. There must be some way of getting X from B to A. In addition, if Process A comes to the point in its execution which requires the information X from Process B, but Process B has

not yet communicated the information to A for whatsoever reason, A must be able to wait for it. Synchronization and communication mechanisms are closely related and can be treated together.

An issue related to synchronization is **nondeterminism**. A process may want to wait for information from any of a group of other processes, rather than from one specific process. As it is not known in advance which member (or members) of the group will have its information available first, such behaviour is nondeterministic. In some cases it may be useful to dynamically control the group of processes from which to take input. For example, a buffer process may accept a request from a producer process to store an item whenever the buffer is not empty. To program such behaviour, a notation is needed to express and control nondeterminism.

Interprocess communication in the languages is broadly classified into two general categories - shared data and message passing.

3.1.1 Message Passing

The most elementary primitive for message-based interaction is the point to point message from one source task (the sender) to another destination task (the receiver). Languages usually provide only reliable message passing. The language run time system (or the underlying operating system) automatically generates acknowledgement messages, transparent at the language level.

Most message-based interactions involve two parties, one **sender** and one **receiver**. The sender initiates the interaction explicitly, for example by sending a message or invoking a remote procedure. On the other hand the receipt of the message may be either **explicit** or **implicit**. With explicit receipt, the receiver is executing some sort of **accept** statement specifying which messages to accept and what actions to undertake when the message arrives. With implicit receipt, code is automatically invoked within the receiver. It usually creates a new thread of control within the receiving process. Whether the message is received implicitly or explicitly is transparent to the

sender.

Another major issue in message passing is the addressing of the parties (or the tasks) involved in an interaction. The sender and the receiver can be addressed **directly** or **indirectly**. **Direct addressing** is used to denote one specific process. The name can be the static name of the process or an expression evaluated at run time. A communication scheme based on direct addressing is **symmetric** if both the sender and receiver name each other. In **asymmetric** scheme only the sender names the receiver. In this case, the receiver is willing to interact with any sender.

Indirect addressing involves an intermediate object, usually called a mailbox, to which the sender directs its messages and to which the receiver listens. This option allows highly flexible communication patterns to be expressed.

Synchronous and Asynchronous point-to-point messages

With synchronous message passing, the sender is blocked until the receiver has accepted the message

(explicitly or implicitly). Thus, the sender and receiver not only share data, but they also synchronize. With asynchronous message passing, the sender does not wait for the receiver to be ready to accept its message.

In asynchronous model, as the sender **S** does not wait for the receiver **R** to be ready, there may be several pending messages sent by **S**, but not yet accepted by **R**. If the message passing primitive is order preserving, **R** will receive the messages in the order they were sent by **S**. The pending messages are buffered by the language runtime system or the operating system.

3.1.2 Data Sharing

If two processes have access to the same variable, communication can take place by one process setting the variable and the other process reading it. This is true whether the process are running on the host where the variable is stored and can manipulate it directly, or if the process are on different hosts and access the variable by sending a message to the host on which it resides.

The shared data scheme has several advantages and

disadvantages over message passing. Whereas a message generally transfers information between two specific processes, shared data are accessible by any process. Assignment to shared data has immediate effect, in contrast, there is a measurable delay between sending a message and its being received. Shared data requires precautions to prevent multiple processes from simultaneously changing the same data.

3.2 Software tools for programming languages

A good generalization can be made that there is good software on low and medium performance systems such as Alliant, Sequent, Encore and Multiflow systems, while there is poor quality software in the highest performance systems. In addition, there is little or no software aimed at managing the system and providing a service to a diverse user community. There is typically no software that provides information on who uses the system and how much, i.e., accounting and reporting software. Batch schedulers are typically not available. Controls

for limiting the amount of time interactive users can take on the system at any one time also are missing. Ways of managing the on-line disks are non-existent.

The system software provided with high performance parallel computers is at best that which would be suitable for systems that would be used by a single person or a small, tightly knit group of people.

Unfortunately, the greater the potential gains from parallelism, the more difficult it becomes to realize these gains. For example, the larger the number of independent processing elements at our disposal the greater the communication overhead penalty incurred by the necessity to pass data between them.

Although multiprocessor machines are becoming widely available, and offer potentially impressive cost to performance ratios, they are as yet user unfriendly environments.

CHAPTER 4

TUPLE SPACE

The most important and perhaps the most distinguishing feature of the proposed communication model, hereinafter referred to as "TUPLE_IO model" is notion of tuple space. TUPLE_IO's elegance is derived from the extreme simplicity of the model. This elegance in turn leads to a reduction of the programmer's burden.

This model is based on generative communication. If two processes need to communicate, they don't share a variable, instead, the data producing process generates a new data object (called a **tuple**) and sets it adrift a region called **tuple space**. Refer fig. 4.1. The receiver processor can now access this tuple. The tuple space is conceptually a shared memory, although its implementation does not require physically shared memory. The tuple space is one global memory shared by all processes of a program.

4.1 TUPLES

An ordered collection of data constitutes a tuple. TUPLE_IO implementation permits various different types of data type to co-exist in the same tuple. Data types of arrays and pointers are handled in a special manner. The maximum number of data types is a controllable parameter in the implementation (10 in TUPLE_IO).

The total size of all the data type in a tuple is also a programmable parameter. In the TUPLE_IO model, the maximum size of the tuple is fixed as 1024 bytes (1KB). If larger messages need to be communicated, then they have to be broken into smaller tuples.

The basic operations on the tuple space are :

send

receive

peek

These operations are of two main types: those that generate tuples, those that access/extract tuples.

The tuples are distinguished from each other by

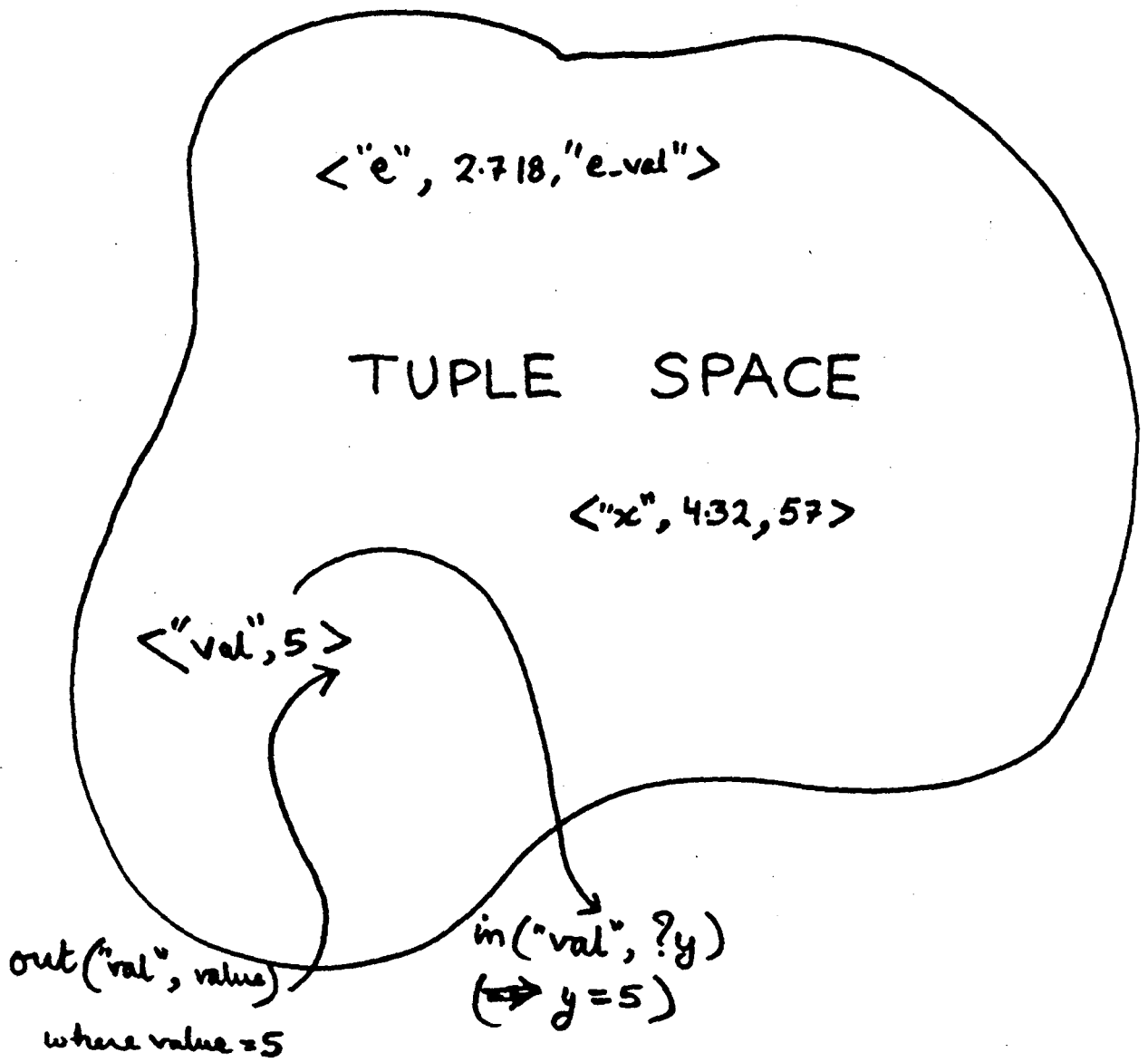


FIG 4.1 TUPLE SPACE

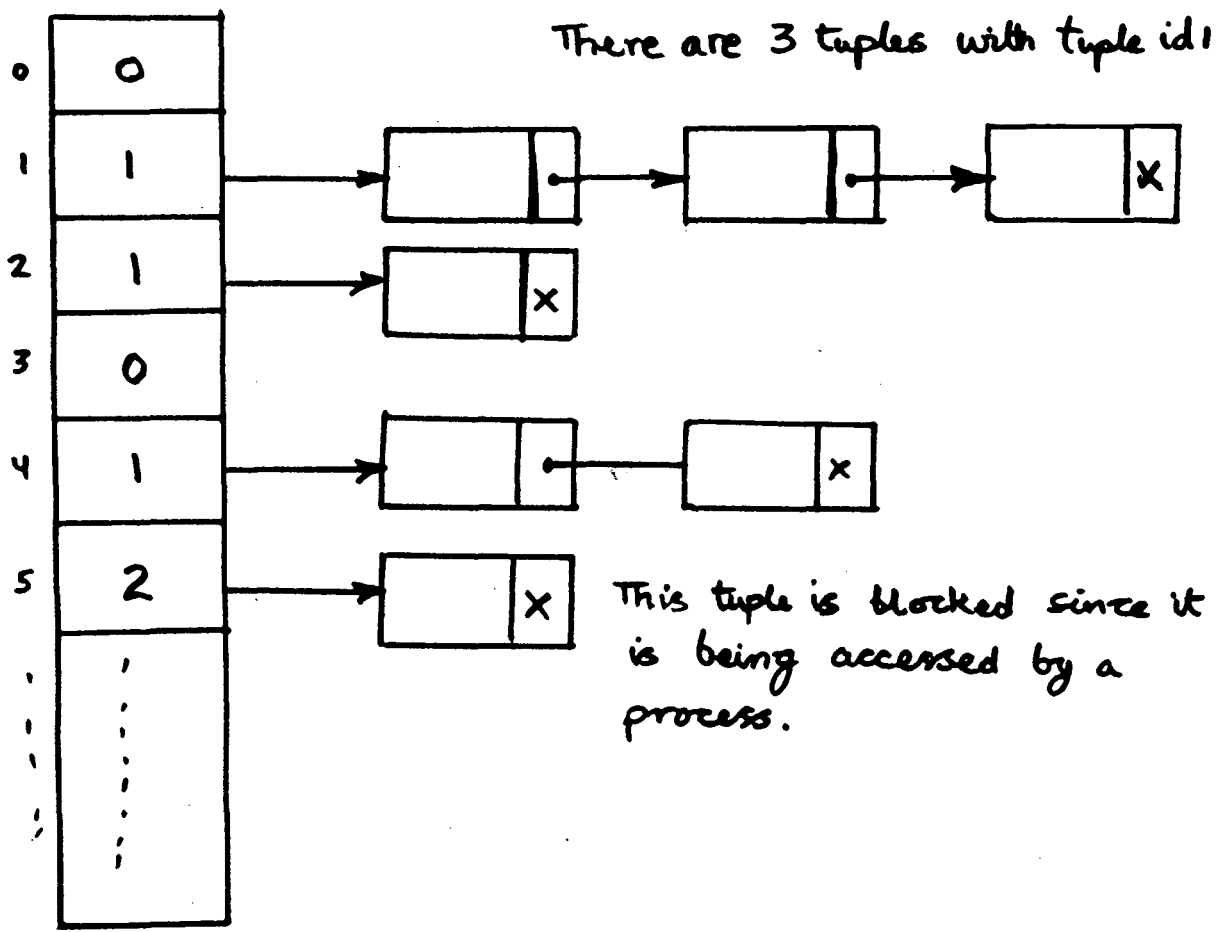
their template name called tuple-id. There can be more than one tuple with the same tuple-id in the tuple space. However, all the tuples with the same tuple-id are placed together on the same processor. When more than one tuple of the same tuple-id is present on the processor, then the distributor maintains them in the form of a linked list. See fig. 4.2.

Placement of the tuples should be done in such a manner so that they are placed uniformly over all the processors. The placement of too many tuples on a particular processor will lead to overloading as all requests to the tuples on it will be directed towards it.

If possible, the tuples are placed on or closest to the processor that operates on it the maximum number of times. Enough research has been done towards this and load balancing algorithms ;Vikas,GSS '90; using Simulated Annealing can be used.

4.1.1 Generative operation : send

When a **send** operation completes, a new tuple is created in the tuple space. The arguments of **send**



There are 3 tuples with tuple id 1

This tuple is blocked since it is being accessed by a process.

INDEX FOR TUPLES

- 0 - EMPTY
- 1 - VALID & FREE
- 2 - VALID & BLOCKED

Only 1 process is permitted to access a tuple at a time

FIG. 4.2 REPRESENTATION OF TUPLES IN MEMORY

can be data variables (integers, float, long, char) and pointer or arrays. The pointer and array data types are preceded by a special character `@` and followed by a number indicating to data size under consideration. However, functions returning any of above data type can not be an argument of `send`. The function should be first computed and then the returned value can be used as an arguement of `send`.

For example :

```
send ("primes", x, value, ticks, @ptr, 10)
```

This denotes that the process issuing this instruction creates a tuple with template name "primes" and 4 data variables. `x`, `value` and `ticks` are normal data types. `ptr` is a pointer (an address) to a data size of 10 bytes.

4.1.2 Accesing and Extracting Operations :

`receive` and `peek`

`receive` extracts and `peek` accesses data from tuples placed in the tuple space by a `send`. The particular tuple from which data will be extracted

is determined by tuple space's matching rules, which in the TUPLE_IO implementation are :

1. The tuple id of the tuple should match the requested template name.
2. The number of data types in the tuple should be same as that in the request.
3. Corresponding constant data types must match.

When a receive or peek executes, if no tuple in the tuple space matches, then the receive or peek will block until a send places a matching tuple in the tuple space.

Consider the tuple space to contain the following tuples:

```
< "array1" , @A , 10 , 'a' >
```

```
< "var1" , x >
```

```
< "array1" , @A , 12 , 'b' >
```

If a processor wants data items contained within tuple of template name "array1", then it does a receive operation

```
receive ("array1", ?@A, 10, 'a')
```

Since two tuples with tuple id, "array1" are present, the tuple which matches the constant data type, 'a' in this case will be removed and 10 bytes will be copied at address A.

If more than one tuple in the tuple space can match, the first one of the linked list will be matched. This ensures that all tuples arriving at tuple space are serviced in an orderly manner. However, the user of TUPLE_IO is advised not to make any assumption regarding the search order in the tuple space as the routing of the tuples is transparent to the user. When a receive finds a match, the matched tuple is removed from tuple space. When a peek finds a match, the matched tuple remains in tuple space but its fields are copied to the request.

Suppose both peek and receive are pending for the same tuple, which currently is not in tuple space, then the action is unpredictable. Nothing can be said as to what will be serviced first, receive or peek. Such situations have to be avoided while programming with these constructs.

4.2 Implementation of Master Slave paradigm

One of the simplest and yet the most useful model of parallelism is the master/worker paradigm. In its simplest form, a master generates a number of independent tasks that can be carried out by any of a number of workers. As an example, consider an application of this model to matrix multiplication. Each inner-product is an independent computation. The master may therefore generate a task for each inner product. The master first sends the matrix index to tuple space where all slaves peek to get this value. The master then sends the task structure for all slaves in tuple space. Each worker takes one of the tasks, does its assigned work and sends its result to the master. The master receives all the result structure and updates product matrix. Refer to the pseudocode in fig. 4. In general, all tasks which are independent of each other can be programmed in this mannner.

```

master()
{
    for all tasks do
    {
        /* build task structures. */
        .
        .
        .
        send ("task", task_structure)
    }

    for all tasks do
    ..
        receive("result",?task_number,?result_structure)

        /* update total result using this
           result and task number          */
        .
        .
        .
    }

} /* end of master procedure */

worker ()
{
    receive ("task", ?task_structute)

    /* execute task */
    .
    .
    .

    send("result",current_task_number,local_result_structure)

} /* end of slave procedure */

```

Fig. 4.3 : Matrix Multiplication based on master/worker paradigm using TUPLE_IO operations.

4.3 Implementation of Reader Writer

Problem in TUPLE_IO

```
long      my_chance ;
initialize_queues()
{
    out ("r/w tail", 0 )
    out ("r/w head", 0 )
    out ("r/w reader count", 0 )
}

ok_to_read()
{
    in ("r/w tail", ?my_chance )
    my_chance++ ;
    out ("r/w tail", my_chance )
    in ("r/w head", my_chance )
    in ("r/w reader count", ?count )
    count++ ;
    out ("r/w reader count", count )
    my_chance++ ;
    out ("r/w head", my_chance )
}

exit_read()
{
    in ( "r/w reader count", ?count )
    count-- ;
    out ( "r/w reader count", count )
}

ok_to_write()
{
    in ( "r/w tail", ?my_chance)
    my_chance++ ;
    out ("r/w tail", my_chance)
    in ( "r/w head", ?my_chance)
    rd ( "r/w reader count", 0 )
}

exit_write ( )
{
    my_chance++ ;
    out ( "r/w head", my_chance )
}
}
```


4.4 Issues in distributing

Tuple Space

Given that the tuple space is logically similar to a shared memory presents a problem. We can put the tuple space on one node and then direct all tuple operations to that one node. This would create an obvious bottle-neck that would almost certainly be disastrous for performance. It seems clear that tuple space should be distributed over some subset (possibly all) of the nodes.

An option is to maintain copies of the tuple space on all processors of the parallel computer. Any given global update to tuple space can be made with a constant number of bus accesses, rather than the $O(n)$ which might have been the case without broadcast. The main drawback is the profligate use of memory.

Another alternate is the "inverse" kernel. In this scheme, tuple space is distributed over the machine by leaving tuples at the nodes where they were generated. receive and peek consult

the portion of tuple space on their node of origin. If no match is found, a request for a matching tuple is broadcasted to all other nodes and a response is awaited. When a matching tuple is found it is sent directly to the requester. This scheme solves the memory problem but gives rise to others. In the case of receive & peek, first the node's local tuple space is searched and then the request is broadcasted to other nodes. This may even double the time for a match. Also when multiple match occur, then too all but one of the tuple has to be discarded.

4.5 Representation of Tuple Space on Transputer based Parallel Computer

In the TUPLE_IO model, the tuple space is mapped onto all the processors of the parallel computer due to the distributed memory. The processors (transputers) do not have a fixed size of memory allocated towards the tuple space. But as the tuples float into and out of the tuple space, the memory is accordingly

allocated or deallocated.

CHAPTER 5

IMPLEMENTATION

The proposed communication model has been implemented on a transputer based distributed parallel computing system. The software has been written in Parallel C (3L C Ltd) Ver 2.10 on a PC-AT 386/33 with an add-on card with one T800 transputer. The implementation phase constituted of encoding the communication harness; comparing the communication overheads introduced; and then testing it by running LU Decomposition application based on the master-slave paradigm.

In this chapter, a detailed description of the communication harness usage is given which can be divided into three basic parts:

- the Preprocessor,
- the Code Generator, and
- the Distributor.

5.1 Preprocessor

The preprocessor first parses the input file

and replaces the TUPLE_IO constructs by appropriate Parallel-C statements for inter process communication.

The user program written Parallel-C with TUPLE_IO constructs is given as input to the preprocessor. The preprocessor parses this program and creates two output files.

One of these files contains information regarding the occurrence of each of the TUPLE_IO constructs in the program. It describes : type of call (send, receive, or peek); line number where call occurred, number of data variables present in the tuple, and the template of the tuple, i.e. tuple-id. This information is required since these calls have to be replaced by suitable Parallel-C statements later by the code generator.

The second output file contains all the tuple-id's referred in the user program and also the number of times each is referred. The user sees this file and determines the processor with which each tuple-id has to be associated with. This has to be done with care so as to distribute the tuples uniformly on all the

processors. A particular tuple should be kept on or closest to the processor that refers to it the maximum number of times.

5.2 Code Generator

Code generator, as the name suggests, generates the code in Parallel-C for a TUPLE_IO statement. In the user program, the TUPLE_IO constructs of "receive", "peek" and "send" are replaced by procedures written in Parallel-C which build up and send two message packets on which the distributor takes action. The first packet is called a **header** and the second packet is called **msg**.

5.2.1 The first message packet : Header

Whenever a processor has to do a tuple space operation, it sends a header packet of fixed size of 15 bytes to the distributor telling it about the requirement. The header structure is as follows:

distinguisher	1 byte
tuple_procnum	1 byte
tuple_num_on_proc	1 byte
num_of_vars	1 byte
source_proc	1 byte
source_task	1 byte
msglength	4 bytes
reserved	1 byte
total_size	4 bytes

The distinguisher has four possible values :
0, 1, 2, or 3. This byte tells the distributor
about the nature of tuple space operation and
are decoded as:

- 0 the call is 'send'
- 1 the call is 'receive' and is going
towards the processor on which the
requested tuple-id resides.
- 2 the call is 'peek' and is going towards
the processor on which the requested
tuple-id resides.
- 3 the call is 'receive' or 'peek' with the

variables of the tuple picked from the
tuple space and going towards the
processor which made the request for the
tuple.
the call indicates that this process no
longer requires the services of the
Distributer.

`tuple_procnum` is the processor number on which
the tuple with the particular tuple-id resides.
It is provided so that the distributor can
direct the header and the message packets along
the appropriate links of the transputer so as to
reach the destination processor.

`tuple_num_on_proc` is the number associated with
the tuple-id on a particular processor.

`num_of_vars` is the number of data variables
present in the field of the tuple.

`source_proc` is the processor from which the
request has been made.

`source_task` is the number associated with the
process (task) on the particular processor.

`msglength` is the size of the second message
packet which is following the header.

`found_tuple_byte` provides information in case of 'rd' statement and informs the processor whether the designed tuple was found or not.

`total_size` is the sum total of the size of each data variable in the field of the tuple.

5.2.2 Second message packet: msg

The second packet of message is a set of contiguous bytes whose size is mentioned in the header. It is of different structures which depends on the distinguisher value.

1. If distinguisher is 0, the msg is of `total_size` contiguous bytes with `num_of_vars` data variables.

var1var2var3.....varn

2. If distinguisher is 1 or 2, then the message structure is :

the first `num_of_var` bytes check whether data item is known or unknown.

unknownvar1unknownvar2.....unknownvarn

next `4*num_of_vars` bytes tell the size of each

variable

size_of_var1size_of_var2.....size_of_varn

next 4*num_of_vars bytes are pointers to each variable

ptr1ptr2ptr3.....ptrn

For each of num_of_var variables, the value of the variable is written if it is a known variable.

3. if distinguisher is 3, then the message is

var1var2var3.....varn

4. if distinguisher is 4, then message is irrelevant.

5.3 Distributor

The distributor is the main communication harness of the TUPLE_IO model described in this report. It handles all the requests made by the processes and performs the necessary action accordingly. The distributor runs in parallel to the executing processes on all the processors of the parallel computing system.

The distributor has as many software ports as the sum of the hardware links of the processor and the executing processes on that processor. A buffer and a semaphore is allocated to each of the output port. A thread is created for each of the input port of the distributor and the distributor procedure is run on all the threads. The semaphore prevents simultaneous access of a channel by two processes.

The distributor waits on each thread till it receives a message packet (the header). On receiving the header, the distributor performs the action according to the distinguisher byte.

1. If distinguisher is 0.

The distributor checks if the tuple has to be placed on the current processor. If so, then it allocates `total_size` bytes in the local memory of the processor and copies the second message into this memory. Else, the distributor sends the header and `msg` to the next processor on route to the destination processor on which the tuple-id resides through appropriate links of the current processor.

2. if distinguisher is 1.

If the tuple-id does not reside on the current processor, then the distributor sends the header and `msg` to the next processor on route to the destination processor on which the tuple-id resides through an appropriate link of the processor.

If the tuple resides on this processor and the current processor is the source processor too, then the distributor searches for the tuple of the required tuple-id with the matching known

variables in the tuple space of the current processor. If it is not found, then it deschedules the search and later tries again. When the tuple is found, it then copies the data variables of the tuple onto the address whose pointers it receives in msg. The tuple is removed from the linked list and memory is deallocated.

If the tuple resides on this processor and the current processor is not the source processor, then the distributor searches for the tuple of the required tuple-id with the matching known variables in the tuple space of the current processor. If it is not found, then it deschedules the search and later tries again. When the tuple is found, it then copies the tuple into msg. The tuple is removed from the linked list and memory is deallocated. It also changes the distinguisher byte of the header to 3, and the msglength to total_size. It then sends the header and msg through appropriate links to the next processor on route to the processor which made the request.

3. If distinguisher is 2.

All actions are same as when the distinguisher is 1, except that the tuple is not removed from tuple space, and the search for the tuple is made only once. If tuple is found then found_tuple_byte is set high (i.e. 1).

4. If distinguisher is 3.

If current processor is one which places the request for the tuple, then each of the data variable is copied into the address indicated by the corresponding pointer in the msg.

If current processor is not the one which requested for the tuple, then the header and msg are passed onto the next processor on route to the processor that requested the tuple.

5. If distinguisher is 4.

This indicates that the the processor generating this tuple has to do no more communication. The distributor program terminates the thread associated with this

process.

5.4 Main features of this communication harness

An effort has been made to provide a user friendly programming environment for programmers. This will save a programmer to learn the communication syntax/mechanism not yet standard of parallel system. The communication harness developed provides simple constructs for message passing.

It has been noticed that a new programmer is reluctant to study the topology of the parallel computer so as to route messages along the shortest path. The proposed model handles the routing of messages.

This communication harness does not make the sender wait till receiver is ready and hence is asynchronous in nature. However, the ordering of messages is ensured.

Another advantage is that a sender wishing to send a message to more than one processes has to

send it only once, all others peek at it and
receive it.

CHAPTER 6

CONCLUSION

This proposed communication harness has been tested with message passing simulation programs; ping-pong message passing from one to many processes and between two processes; LU Decomposition technique for solving a set of linear equations.

Unfortunately, testing this harness on multiple processors could not be achieved due to unforeseen delays which were faced due to the bugs in the Parallel C compiler Ver 2.1.

This harness is based on asynchronous communication paradigm. LU Decomposition tests for solving a set of linear equations when conducted on this harness and another synchronous harness (developed by a research team at BARC, Bombay) clearly show the speedup achieved over the existing synchronous message passing techniques. To solve a set of 16 variable linear equations it took 6 seconds on the synchronous harness while it took less than a minute in TUPLE_IO model.

This harness removes the message routing and

all connectivity problems from the programmer. The programmer does all the necessary communications using the three available TUPLE_IO constructs. All the tests conducted prove that these three constructs are enough, and adding more constructs will only lead to programming complexities, removing which has been the main aim of this project.

This harness suits specially those situations where message passing is from one to many process of the same data. While using this harness, the programmer generates the message using the send construct, whereas all processes needing it can peek at it and read it. Later this message should be destroyed.

Future work on this harness ought to include the following :

1. The harness should be checked on multiple processes running on multiple processors. This would ensure that the protocol holds at even boundary cases. Currently the values of maximum processes and processors are hard coded. These should be dynamically managed.

2. Communication traffic monitoring should be added to enable the placement strategy of the user program to be quantitatively evaluated.

3. When the preprocessor scans the users input source program then it should also check for communication deadlocks. This will be of great usefulness to parallel programming community.

4. A major debatable issue is whether the user should be made to timeout if the message requested by the user is not yet ready. May be in the future version an option should be provided.

BIBLIOGRAPHY

1. Hwang, Kai. Advanced Parallel Processing with Supercomputer Architectures, Mc Graw Hill Press.
2. OCCAM 2, Reference manual, INMOS Ltd., 1987.
3. Singh, G.S. Trends in Parallel Processing, Feb 1988.
4. Quinn, M. Design of efficient algorithms for Parallel Computers, Mc Graw Hill Press.
5. Ni, C.N. and Hwang, Kai. "Optimal load balancing mutiprocessor system with many job classes". May 1985.
6. Ellis, G.E. "Transputers Advance Parallel Processing", Research & Development, March 1989.
7. Transputer Reference Manual, INMOS Ltd.
8. Ahluwalia, V and Singh, G.S. "Load Partioning on a transputer based Parallel Computing System using MIMD algorithms of Simulating Annealing & Heuristics". AMSE '90 Conference, Tirupati. Dec '90.
9. Singh, G.S , Khare A.N. and Ghodgaonkar, M. "A

High Performance system", Feb 1988.

10. Ahuja, Carriero and Gelernter. "Linda and Friends". IEEE Computers 19(8). Aug 1986.

11. Bal H.E. "Programming Distributed Systems". Silicon Press. 1990.

12. Andrews, G.R and Schneider, F.B. "Concepts and Notations for Concurrent Programming". ACM Computing Surveys. 15(1). March 1983.

13. Bal H.E. and Tanenbaum, A.S. "Distributed Programming with Shared Data". Proc. of IEEE CS 1988 Int. Conf. on Computer Languages, Miami. Oct 1988.

14. Bal, H.E. "Languages for Parallel Programming". Vrije University, Oct'1990

14. Bal, H.E. and Tanenbaum et. al. "Programming Languages for Distributed Computing Systems". ACM Computing Surveys, Vol21(3), Sep' 89.

15. Messina, P. "Parallel computing in the 1980s - one person's view". Concurrency : Practice and Experience, Vol 3(6). Dec '91.

16. Carriero, N and Gelernter, D. "Linda in Context". Comm. of ACM, Vol 32(4). April 1989.