1994

Object oriented support on transputers

Aditya Bates

University of Wollongong

Recommended Citation

OBJECT ORIENTED SUPPORT ON TRANSPUTERS

A thesis submitted in partial fulfilment of the requirements for the award of the degree

UNIVERSITY OF WOLLONGONG
Honours Master of Science
(Computer Science)

from

UNIVERSITY OF WOLLONGONG

by

Aditya Bates, B.E. (Computer Eng)

Department of Computer Science

1994
ABSTRACT

This thesis investigates and demonstrates the successful integration of object oriented approach with concurrent programming. It demonstrates that transputers coupled with virtual channels can provide an effective and efficient coarse grained object oriented programming environment. A methodology for concurrent object oriented design is developed and explained. This methodology is used to develop an application program using the virtual channel router, which mimics the transputer T9000 and its virtual channel processor. Testing is then carried out to validate and demonstrate the working of the application developed.

The design developed is thereafter mapped onto a programming environment wherein broadcast communication is used and mimics a shared memory model instead of point to point communication in a distributed memory model. This is first done using BSP Occam and then Occam 3. An assessment of the advantages of object orientation and parallel processing on transputer T9000 is then concluded based on the research done during the design, implementation and testing stages.

The thesis has validated that object oriented concepts can be successfully implemented on transputer - occam thereby achieving an efficient and effective medium grained parallel objects.
ACKNOWLEDGMENTS

Firstly, I would like to express my gratitude and special thanks to my supervisor Dr Jonathan Gray for his guidance, support and valuable suggestions in the research and development of this thesis. His comments have significantly improved it. I would also like to thank him for his patience and time on occasions beyond normal hours.

Secondly, I would like to sincerely thank Dr Greg Doherty for his guidance, encouragement and in depth review of my thesis. He has been a source of great motivation to me in my studies for the thesis and its completion.

I would also like to acknowledge and thank the academic and support staff of the Department of Computer Science, University of Wollongong who have created one of the most pleasant study and research environment in computing that I have encountered.

Finally, thanks to my father Capt Bates for being a critic of the thesis and to my mother Shelley for her patience and understanding. I would also like to thank my sister Anjali for her assistance in typing the manuscript and graphics.
## CONTENTS

1 Introduction ...................................................................................................................... 1
   1.1 Introduction........................................................................................................ 1
   1.2 Objectives of the thesis................................................................................. 1
   1.3 Overview of thesis......................................................................................... 2

2 Parallelism....................................................................................................................... 4
   2.1 Introduction......................................................................................................... 4
   2.2 Parallel Computing.......................................................................................... 5
   2.3 Classification of Parallel Computers............................................................ 5
   2.4 Architectures of Parallel Computers............................................................ 6
       2.4.1 Distributed Memory Architecture....................................................... 7
       2.4.2 Shared Memory Architecture............................................................... 8
   2.5 Massively Parallel Processors (MPP).......................................................... 9

3 Object Oriented Concepts............................................................................................. 12
   3.1 Introduction...................................................................................................... 12
   3.2 Concept of Object Oriented Design (OOD)................................................ 13
   3.3 Advantages and Uses of OOD................................................................... 16
   3.4 Object Oriented Programming languages................................................... 17
       3.4.1 C++............................................................................................................... 17
       3.4.2 Eiffel............................................................................................................ 18
       3.4.3 Smalltalk.................................................................................................... 19

4 Concurrent Object Oriented Programming............................................................... 20
   4.1 Introduction...................................................................................................... 20
   4.2 Process and Objects..................................................................................... 20
   4.3 Parallel Object Oriented languages........................................................... 22
       4.3.1 Actors........................................................................................................... 22
       4.3.2 POOL and DOOM................................................................................... 23
6.2.1 Object identification...............................................................................42
6.2.2 Object interaction..................................................................................43
6.2.3 Attributes and operations.....................................................................43
6.2.4 Generalisation......................................................................................44
6.2.5 Process identification...........................................................................47
6.2.6 Concurrent threads identification.......................................................49
6.2.7 Shared objects......................................................................................49
6.2.8 Mapping on Transputers.................................................................50

6.3 Advantages of OO in this application.......................................................52
   6.3.1 Exploiting inheritance........................................................................52
   6.3.2 Polymorphism....................................................................................52

6.4 Parallel features in the design.................................................................53
   6.4.1 Functional parallelism in design.......................................................54
   6.4.2 Geometric parallelism in design.......................................................55

7 Implementation...........................................................................................................56
7.1 Introduction......................................................................................................56
7.2 Transputer T9000..........................................................................................56
   7.2.1 Performance........................................................................................57
   7.2.2 Architecture.......................................................................................57
   7.2.3 Communication support device.......................................................58
   7.2.4 Advantages of T9000..........................................................................58
   7.2.5 Virtual Channel Processor (VCP)......................................................59
7.3 Virtual Channel Router (VCR)........................................................................60
   7.3.1 Concept of VCR................................................................................60
   7.3.2 What is a VCR ?.................................................................................60
   7.3.3 Configuration Files............................................................................62
   7.3.4 Working of VCR...............................................................................62
7.4 Design implementation...................................................................................63
   7.4.1 Implementation of OO concepts.......................................................63
      7.4.1.1 Objects and Classes implementation........................................64
      7.4.1.2 Inheritance implementation.......................................................66
1 INTRODUCTION

Greater than the tread of mighty armies
is an idea whose time has come.
- Victor Hugo

1.1 Introduction

Conventional supercomputing now seems poised for an inexorable decline. On the verge of a gradual takeover, industry analysts believe, are massively parallel processors (MPP) which can achieve processing rates of teraflops needed by science, technology, governments and industry to solve problems of unprecedented complexity and stay competitive. There are thousands of commercial applications now in use, and many more envisioned for these more powerful parallel computing machines that are still some years away [Zorpette 92].

Distributed memory architectures can be scaled up to massively parallel architectures to build these machines. However, distributed memory processors suffer from the lack of efficient programming environment. One solution is to use object oriented concepts as they can be mapped onto the communication structure of a parallel system. Further, the limited number of links between processors is also a constraint in parallel systems, but with the availability of the concept of virtual channels this problem has been largely solved.

1.2 Objectives of the thesis

An attempt has been made in this thesis to investigate and demonstrate how the concepts of object orientation and distributed memory architecture coupled with virtual channel router can be used to build an effective programming environment.
The objectives of the thesis are:

- to develop a methodology for concurrent object oriented design,

- to map object oriented concepts on distributed memory architecture,

- exploiting the virtual channel router to allow objects on different processors to communicate with each other over an unlimited number of channels,

- to map the design developed onto BSP Occam and Occam 3 using the broadcast method of communication,

- to implement, validate and demonstrate the above concepts by designing and developing an application program in this programming environment.

1.3 Overview of the thesis

In chapter 2, parallel computing and massively parallel processors which can achieve speeds of teraflops using distributed memory architecture are described.

In chapter 3, the advantages of object oriented concepts are given. Object oriented design concepts are explained and their possible use outlined.

Chapter 4 looks at concurrent object oriented programming. A survey of parallel object oriented languages is given and how concurrent object oriented programming can be done on transputers is described. Mapping of object oriented concepts onto non object oriented languages is explained.

The methodology for developing an object oriented system for transputers is covered in
chapter 5. How this design methodology exploits both object orientation and parallelism concepts are also explained.

Chapter 6 gives an overview of the concurrent object oriented design. It explains the design, the advantages of OO and parallel features in this design.

In chapter 7, the implementation of the design is given and how the system works. It explains how OO and parallel concepts have been implemented. It also gives the overview of the working of the system. The serial component in the implementation is given and a revised design is explained to overcome this bottleneck.

In chapter 8, the above design is redeveloped and explained using the broadcast method of communication in place of message passing.

Chapter 9 covers the validation of the design and results. It also includes the conclusion of the thesis and where this research can be extended.
2 PARALLELISM

Death, taxes, and parallelism: nobody's in favour of any of them, but they're inevitable facts of life.

Paul Schneck, Director,
Supercomputing Research Centre, Bowie, USA.

2.1 Introduction

For a long time the principles of computer architecture design have largely remained static, based on von Neumann principles. The requirement to increase the performance of conventional supercomputers above tens of billions of operations per second, up from millions per second has led to the development new architectures based on parallel organisation of computations.

Parallelism denotes the concept of executing several operations simultaneously. Charles Babbage said in 1842, "When a long series of identical computations is to be performed, such as those required for the formation of numerical tables, the machine can be brought into play so as to give several results at the same time, which will greatly abridge the whole amount of the process" [Jesshope 88].

The shift to parallel processing in number crunching as well as in knowledge engineering is the challenge and opportunity of this decade. The 1990s has been termed as the age of parallel computers, as the 1980s was of PCs, the 1970s of minis and the 1960s of mainframes. Parallel computing is considered to be one of the most exciting technologies to gain prominence since the invention of electronic computers in the 1940s and by the year 2000, it is expected to be as mainstream as PCs were in 1989 [Lewis et al 92].
2.2 Parallel computing

The technique of parallel computing has been around for more than a decade, but only in the mid 1980s the industry reached a consensus that it was the only way to build powerful computers achieving processing speeds of teraflops not attainable by conventional vector supercomputers.

The parallel computing machine is based on a number of processors (hundreds or thousands) similar to those in a high performance workstation which are networked by communication links. It is the synergy of these processors working together in a parallel mode that gives these machines their high potential processing rates.

An important factor that effects the performance of parallel computers is the existance of a serial code in the implementation. Amdahl's law expresses the fact that the inherent sequentiality of an algorithm is the ultimate limiting factor in the performance on any machine [Anderson 89]. Another factor is the granularity of parallelism. Transputers are best suited for medium level of granularity [Inmos 88]. Fine grained and massive parallelism is difficult to support. Bottlenecks caused by insufficient memory or communication between processors can also be a limiting factor [Anderson 89].

2.3 Classification of Parallel Computers

Before we begin the classification of parallel computers, let us look at the taxonomy proposed by J. Flynn in 1972 for classification of computer systems. It consists of four classes of processor systems as listed below:

- Single Instruction Single Data stream (SISD)
- Multiple Instruction Single Data stream (MISD)
- Single Instruction Multiple Data stream (SIMD)
SISD is the single processor model. Here one instruction is carried out on one data item at a time. This type of hardware is not capable of supporting parallel computing.

It is generally agreed that there are no computers that can be classified as MISD, though in the taxonomy by Handler and Thurber, they have considered pipeline processors as MISD [Anderson 89].

In a SIMD computer, a single instruction is acted upon by many processors on different data. Whereas in MIMD computer, multiple instructions are executed on multiple data on different processors. Both SIMD and MIMD computers fall in the domain of parallel computing.

Since in SIMD computer, all processors execute the broadcasted instruction synchronously, in lock-step, for each cycle but on different data. Such a design can be built more simply and inexpensively than comparable MIMD design. SIMD programming is also termed as synchronous parallel computing since all its processors are synchronised but each handles a different piece of data. The criticism of SIMD is that it works best for certain classes of problems only, like image processing where the same operation is performed on every byte of pixel in an image. It is probably because of this that most parallel computer makers of general purpose machine have settled on MIMD architecture.

2.4 Architectures of Parallel Computers

A parallel computer is made up of two or more processing units connected by some network. Parallel computers can also be categorised by their memory architecture as follows:
2.4.1 Distributed Memory Architecture

In a distributed memory architecture, the computer consists of processors with their associated local memories connected by communication links. There is little or no shared memory and thus it is necessary to move data from one local memory to another by means of message passing. Each processor has exclusive access to its own local memory. Its model is given in the diagram below.

![Distributed Memory Interconnection Model](image)

*Fig 2.4.1 Distributed memory interconnection model*

The advantage of distributed memory architecture over shared memory architecture is that it can be scaled up to massive parallelism. While shared memory computers are relatively easy to program [Lewis et al 92].

Programs written for distributed memory architecture use message passing among tasks. This concept has led to a new mathematical approach to programming called
Communicating Sequential Processes - CSP [Hoare 85], which has been implemented in the development of the parallel programming language - Occam.

One of the important performance metric of parallel machines is the ratio of the time the nodes spend processing data to the time they spend communicating with each other. The system is considered balanced if the time spent communicating is around 10 to 20 percent of the time for execution of the application. But much more then that indicates a communication bottleneck [Zorpette 92].

2.4.2 Shared Memory Architecture

In the shared memory architecture of mainframes and conventional supercomputers, there is a common pool of memory shared by all processors. The advantage of shared memory computers over distributed memory computers is that they are easy to program. The disadvantage is that they are difficult to scale up to a large number of processors required by massively parallel machines. The model of the shared memory model is given below.

![Shared Memory Interconnection Topology Model](image)

*Fig 2.4.2 Shared memory interconnection topology model*

Because of the programming complexities of message passing in the distributed memory models and lack of scalability to a large number of processors in the shared memory model, several companies are attempting to develop parallel architectures that mimic the
shared memory model wherein though the memories are physically distributed, but they are managed by a combination of software and architecture like a single large global virtual memory. So far only one of these architectures, the KSR1 from Kendall Square Research is on the market, but others from Cray Research, Tera Computers, Convex Computers and others are under development [Zorpette 92].

Hardware and software needed to effectively implement the above hybrid virtual memory model with thousands of nodes, having memories physically distributed but globally and logically shared, is still few years away but this is the directions researchers think the industry will take.

2.5 Massively Parallel Processors (MPP)

In principle a distributed memory parallel computer can be scaled to massive proportions. For example, it is technically feasible to build a massively parallel computer with an interconnection of hundred or thousands of microprocessors [Zorpette 92]. Machines in this class are called Massively Parallel Processors (MPP). Among the factors that distinguish MPPs from each other is topology ie the way in which the processing nodes are interconnected via communications links. Some common topologies are:

- hypercube,
- two-dimensional mesh,
- fat tree.

These MPP machines being developed at the cutting edge of computing technology with processing rates as high as a trillion floating-point operations per second (teraflops) would have almost 100 times the processing power of today's best sequential supercomputers. They have the promise of solving some of the fundamental problems and grand challenges facing mankind are shown in the figure on the next page.
It is common knowledge that advances in software lag behind those in hardware. Nowhere is this more obvious than in MPP. Yet there are signs of progress in the development of standard operating systems, programming languages/compilers and software tools for supporting MPP.

Current price estimates of these first teraflops MPP machines exceed $US 100 million which would place them out of reach of most users. Vendors are, however, responding
to the market forces for a lower price/performance ratio and it is only a matter of time before MPPs offering high performance are available at a cost-effective price [Cybenko et al 92]. Some of the current MPPs are given below.

<table>
<thead>
<tr>
<th>Company</th>
<th>Current Model</th>
<th>Architecture</th>
<th>Node Microprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Supercomputer Systems Division, Beaverton, Ore.</td>
<td>Paragon XP/S</td>
<td>MIMD, 2-D mesh</td>
<td>Intel i860 XP</td>
</tr>
<tr>
<td>Kendel Square Research Inc, Waltham, Mass.</td>
<td>KSR1</td>
<td>MIMD, hierarchy of rings</td>
<td>Custom Superscalar RISC chip</td>
</tr>
<tr>
<td>MasPar Computer Corp., Sunnyvale, Calif.</td>
<td>MP-1</td>
<td>SIMD, 2-D mesh</td>
<td>32 custom 4-bit processors per chip</td>
</tr>
<tr>
<td>Meiko Scientific Corp., Waltham, Mass.</td>
<td>Computing Surface</td>
<td>MIMD, variable</td>
<td>Intel i860 or Sun Sparc</td>
</tr>
<tr>
<td>nCube, Foster City, Calif.</td>
<td>nCube 2S</td>
<td>MIMD, hypercube</td>
<td>64 bit custom scalar processor</td>
</tr>
<tr>
<td>Parsytec GmbH Aachen, Germany</td>
<td>Parsytec GC</td>
<td>MIMD, 3-D mesh</td>
<td>Transputer T9000</td>
</tr>
<tr>
<td>Thinking Machines Corp., Cambridge, Mass.</td>
<td>CM-5</td>
<td>MIMD, fat tree</td>
<td>Sun Sparc</td>
</tr>
<tr>
<td>Wavetraner Inc., Acton, Mass.</td>
<td>Data Transport Computer</td>
<td>SIMD, 3-D mesh</td>
<td>Custom bit serial processors</td>
</tr>
</tbody>
</table>

Fig 2.5b Massively parallel computers at a glance [Zorpette 92].
3 OBJECT ORIENTED CONCEPTS

A tidal wave is reaching the shores of the computing world.
- Bertrand Meyer, Interactive Software

3.1 Introduction

One of the most important recent advances in software technology has been object orientation which is expected to become the mainstream method for supporting software life cycle development in the 1990s.

Object orientation (OO) is a computing methodology in which the software system to be constructed is modelled as a collection of objects that interact with one another. In an object, both the data and operations are encapsulated [Goldberg 89], [Booch 91]. An object is thus an entity that contains both the attributes that describe the state of the real world object and the actions that are associated with the real world object.

Object = unique identifier + data + operation

One of the aims of OO implementation is the development of generic object classes. The reusable object classes are stored in libraries for widespread availability. Making software reusable is important because 40 to 60 percent of code in the program is redundant [Langergan et al 89]. Reusable object classes are called components. Features that enhance reusability are encapsulation and inheritance. Encapsulation means that objects can be treated as black boxes whereas inheritance involves sharing between objects.

An object class describes a group of objects with similar properties of common
behaviour, common relationships to other objects and common semantics. It is an implementation of object type. It specifies the data structure and permissible operational behaviour that apply to each of its objects. Object orientation is generally regarded as being the synergistic embodiment of essentially three concepts:

- encapsulation and information hiding
- classification and abstract data types
- polymorphism throughout inheritance

3.2 Concept of Object Oriented Design (OOD)

The concept of encapsulation is implemented in OO design by making use of data abstraction. It is a methodical approach to problem solving. Here the information is hidden in small data types. Thomas [Thomas 89] says that "encapsulation is the technical name for information hiding". Code and data are encapsulated together into an object. Encapsulation does not guarantee information hiding although the reverse is true. Encapsulation and modularization are ideas of tying things together with an interior that can be hidden from view. It separates the internal implementation details of the object from the external aspects of the object. In figure below, code and data of objects polygon are encapsulated in polygon class by abstraction.

![Diagram](image)

*Fig 3.2a Objects and classes [Rumbaugh et al 91]*
Modularization uses the notation of an Abstract Date Type (ADT). An ADT extends this idea and allows the user to define its own data type and to use it just as if it were a basic language type. This user defined type can be anything eg table or chair. That's the nub of object oriented program, thinking of your object classes as abstract data types and using them just as you would any other data type. Rumbaugh [Rumbaugh et al 91] says that abstraction is the selective examination of certain aspects of the object.

Inheritance is an important concept. Inheritance in programming languages can be found only in Object Oriented Programming Languages (OOPs). Inheritance promotes sharing ie it allows common data structures and behaviour to be shared among several similar subclasses without redundancy. It offers the prospect of reusing designs and codes on future projects. OOPs are beginning to support multiple inheritance, where features can be inherited from two or more parents. An inheritance structure is one of the ways of offering reusability, extendability and lower maintenance cost.

Some languages use "delegation" instead of inheritance. Here the object is defined by its difference from an existing object. When an object receives a message which it does not understand, it delegates the message to another object. A object delegates or passes the message to the another object, if it does not know how to handle the message and that other object knows the address of the target object. The message passing in delegation takes place in a point-to-point fashion.
Polymorphism is a Greek word meaning many forms. A polymorphic object is any entity such as a variable or function argument that is permitted to hold values of different types during the course of execution. Thus polymorphism is a situation where one function can be used with different arguments. The advantage of polymorphism is that the code can be written once and it can be reused repeatedly with different low level abstractions.

Object oriented design concepts are thus based on:

- Objects
- Classes
- Abstract data types
- Messages
- Inheritance
- Polymorphism
3.3 Advantages and Uses of OOD

One of the most important advantages of OOD is reusability of its codes, program portions, and design. Reusability in the long term relies on developing a library of classes from both commercially available classes and in-house developments. Reusability is implemented in OO software when behaviour is inherited from another class and the code that provides the behaviour does not have to be rewritten. The same class can be reused by many users and these are known as software components. Cox [Cox 86] calls them as software ICs.

Components offered for reuse are reliable as extensive testing is warranted [Binder 94]. These components have stood the test of time and are correct. However, each reuse is in a new context and thus retesting is necessary. It seems likely that more, not less, testing will be needed to obtain quality and reliable object-oriented systems [Binder 94].

Software built by connecting objects communicating with each other, is easier to extend than systems where the design is wired and entangled in the structure. Thus message passing between objects in OO software makes the system flexible and extendable.

Object-oriented software is partitioned into objects, which correspond to and represent its real world counterparts. It is, therefore, much easier to comprehend and analyse.

Compatibility with other products is desirable and can be achieved in OOD by linking OO code modules with non OO code modules of the same language.
The main advantages of object oriented design are thus as follows:

- reusability resulting in productivity and lower costs,
- correctness and reliability of reusable components,
- message passing, extendability as well as scalability,
- encapsulation which leads to easier maintenance and evolution,
- a closer model of the real world resulting in more natural and better analysis and design.

3.4 Object Oriented Programming languages

Object Oriented programming is becoming popular as the advantages of object oriented design and methodology are being recognised. The three major properties of object oriented programs that are making it popular are object based modular structure, data abstraction and the ability to share code through inheritance. One of the main advantages of object oriented programming languages is that they offer software developers the ability to manage and develop large software projects more easily.

In object oriented programming, objects developed earlier are stored in the class library and then reused later as required. In an object oriented language, polymorphism is achieved by letting different classes implement methods that have the same name and formal parameters but a different implementation. Some of the popular OO languages are given below.

3.4.1 C++

C++ is an object oriented extension of C inspired by Simula 67 and developed by Bjarne Stroustrup at Bell Laboratories in Murray Hill, New Jersey [Stroustrup 86]. The object oriented features of C++ language are listed below:
In C++ both attributes and methods are declared together as members of a class. The creation routine constructor has the same name as the class. The constructor operation is used to initialise new instances and the destructors to clean up objects.

Inheritance in C++ is implemented by defining the superclass (or superclasses) as part of the class declaration as shown below.

```cpp
class subclass: public superclass
{
protected:
    // protected data and method
public:
    // public data and method
}
```

### 3.4.2 Eiffel

Eiffel is a strongly typed object oriented language written by Bertrand Meyer [Meyer 88]. It has a class library which includes lists, tree and stacks. It provides portability because the Eiffel compiler translates source programs into C. In Eiffel the class declaration is as follows:
3.4.3 Smalltalk

Smalltalk was developed at Xerox Palo Alto Research Centre [Goldberg 89]. It was the first popular object oriented language. Besides being a language, it also has a development environment incorporating some functions of an operating system. The language syntax is simple and the variables and attributes are untyped. The Smalltalk environment allows for rapid development of programs. The programs in Smalltalk are entered using the Smalltalk browser. All Smalltalk classes are ultimately descended from class object.
The whole is greater than the sum of its parts.

- Aristotle

4.1 Introduction

Objects in the world around us act and interact with us and with each other in accordance with some characteristic pattern of behaviour [Hoare 85]. Hoare further states that this behaviour pattern of the objects is called process and the objects in the world around us are acting and interacting with a high level of concurrency.

In OO computing, objects cooperate by exchanging messages. In concurrent programming, processes cooperate and exchange messages. The concept of concurrent OO programming can be realised by combining the concepts of object and process into one. Meyer says that the marriage between concurrent computation and object oriented programming is a union desired by practitioners in the fields of telecommunication, high performance computing, banking and operating systems and it appears easy enough, but this appearance is deceptive [Meyer 93]. He also says that the mutual attraction between OO and concurrency can be turned into a reasonably happy marriage. Object oriented paradigm provides an appropriate framework for multiple granularities of parallelism required by this union.

4.2 Process and Objects

Interaction in process is modelled by exchanging messages. Service is requested by sending a message to the service providers. The service provider acts on the received message and provides the required service. The service providers are medium to large
grained encapsulated entities as they have methods which provide the service. There can be many service providers and they can all work as concurrent processes.

Such system modelled on message passing map very closely to the object-oriented model and are a promising candidate for object-oriented model of software development. Service providers and requesters are objects in the model, the message protocol defines the interface and the message passing through the interface supports method invocation. The service providers are medium to large grained object as they encapsulate the methods for services while the service requesters are fine to medium grained objects as they don't provide any services. Though true object oriented programming on sequential computers is different and difficult to implement, but systems modelled on message passing map closely to object-oriented model where multiple granularities of parallelism can be supported.

The notation of process allows codes and data to be encapsulated into a unit that can be executed on a processor which is a physical machine whereas process is an abstract computation. Processes run on separate processors and object instances are encapsulated in these processes. Objects support a programming model whereas processes support an implementation model.

Each object has one associated class which is a piece of code describing the object. Objects interact by messages, processes by message channels.

As there are many similarities between object oriented and concurrent programming, researchers have tried to unite the two but none has so far succeeded in providing a widely acceptable solution because most have tried to integrate the full power of concurrent programming with the full power of sequential object oriented programming [Meyer 93].
4.3 Parallel Object Oriented languages

In this section some of the parallel object oriented languages are covered. These are Actors, POOL, Concurrent Smalltalk and ABCL/1.

4.3.1 Actors

The actor [Agha 89] model is a well known attempt to provide a high level of inherent parallelism for programming large parallel systems. It was first developed by Hewitt and later by others [Agha 86].

The actor model of computation exploits message passing as a basis for concurrent computation. Mail queues in actors allow asynchronous queued message passing. An actor is a computational agent that carries out its processing action in response to a communication. An actor may be described by specifying its mail address and behaviour. An actor [Agha 89] influences the action of other actors by sending communication to the mailing address of the recipient. The behaviour determines what it should do when it receives a message. The idea of actors and message passing makes the system extendable. Thus new behaviours can be added to the system with ease. An abstract representation of actor is shown below.

![Abstract representation of an actor](image)

*Fig 4.3.1 Abstract representation of an actor [Agha 86]*
The actor model uses the concept of delegation of messages to support inheritance. When an actor receives a message and cannot answer the message, it delegates the message to other actors in the system. This replaces the concept of class and subclass and provides a capability for sharing of common knowledge among objects.

Actions that an actor performs are send communication to itself or to other actors, create more actors or specify the replacement behaviour [Agha 87]. A program in Act3 (an actor based programming language) is a collection of behaviour definitions and commands to create actors and send communication to them.

4.3.2 POOL and DOOM

Parallel Object Oriented Language (POOL) [America et al 86] was first designed in 1984 to develop a programming language that could effectively support the construction of applications for distributed memory architecture of highly parallel systems.

In POOL the program is divided into a number of objects which communicate by sending messages. Objects state explicitly the destination object to which the message has to be sent. After the object accepts the message, it executes one of its methods. Parallelism is obtained by starting simultaneously the execution of several objects. These programs are executed on Decentralised Object Oriented Machine (DOOM) [Bronnenberg 89], [Bronnenberg et al 88].

Bronnenberg [Bronnenberg 89] has specified criteria for placing objects onto processor nodes. The first criteria is that objects which can operate in parallel should be placed on different nodes. The second criteria is that the objects that are sharing the same memory and address space are placed on the same node.
In POOL inheritance is implemented. The top part of the body is concerned with the initialisation of the variables. This part of the body is included in the inheritance package to which the variables belong. These statements can then be fixed to the body of any class that uses the inheritance package. That is a part of supernode is copied into the body of objects in the subclass. In parallel systems, the bodies of objects are an important part of the code. Inheritance is not very useful in this form as given below

Procedure Account
Begin
  Superclass : non
  Balance
  Credit limit
End

Procedure Passbook saving a/c
Begin
  Superclass : Account
  Balance
  Credit limit
  Interest rate
End

Procedure Cheque account
Begin
  Superclass: Account
  Balance
  Credit limit
  No interest rate.
End

As shown in the code above, the superclass Account has subclasses Savings account and Cheque account which inherit the code Balance and Credit limit from it. This is achieved by copying the code Balance and Credit limit from the superclass Account to the subclass Savings and Cheque Accounts. Inheritance above is thus implemented by copying the part of the superclass into the subclass. After experimenting with inheritance in POOL, it was decided not to include it in the later version POOL-T [America 87] because of limited reusability.

POOL-T offers object oriented programming for structured parallel systems. In contrast to other object oriented languages, POOL-T does not consider classes to be objects as
classes are static entities whereas objects are instances of class and are dynamic. In POOL-T, object oriented concepts are integrated with parallelism by associating a process with every object.

4.3.3 Concurrent Smalltalk

Concurrent Smalltalk [Yokote et al 87] is an object oriented language that has parallel features. Its syntax and semantics are based on and compatible with Smalltalk. In Concurrent Smalltalk a process is realised by an instance of a class process. Processes communicate with each other using common variables as in Smalltalk. Mutual exclusion between common variables is achieved by using semaphores. Concurrent constructs and atomic objects are used to implement concurrent object oriented language.

4.3.4 ABCL/1

An Object Based Concurrent Language (ABCL/1) is based on the frame-work of object based concurrent programming [Yonezawa et al 87]. Its design is based on the premise that the semantics of message passing among objects should be clear and that every concept in the language need not be represented in the form of object and message passing.

ABCL/1 reflects the object-oriented computation model of ABCM/1. In this model computation is performed by objects. The objects are abstract entities and become active and start computation after receiving a message. Message transmission and objects becoming active take place in parallel. Parallelism is exploited in ABCL/1 by concurrent activation of multiple, independent objects. Synchronisation is achieved by an object performing a single sequence of action in response to one message. Objects don't execute more than one action at the same time also they change to waiting mode to achieve synchronisation [Yonezawa et al 87].
4.3.5 CEiffel

CEiffel was developed by introducing concurrency to sequential object oriented language Eiffel [Bruno et al 93]. No change is made to the Eiffel language, concurrency is introduced to the language by creating a concurrency class in the class library. The advantage is that they do not replace the existing software and the disadvantage is that sequential semantics impose restrictions on intra object concurrency.

Most of the earlier concurrent object oriented languages were new languages. Extending an existing object-oriented language is more recent, and has been influenced by most of the earlier work on concurrency. The idea here is the integration of process with the notion of object. This integration results in an active object. The object become active only when they inherit from the concurrent class. Concurrency is viewed as the parallel execution resulting from the creation of these active objects and their interactions with one another [Bruno et al 93].

4.3.6 CHOICE

Here like CEiffel, concurrency is introduced to the C++. Approach of introducing concurrency is via a class definition of process. Concurrency here is viewed as the parallel execution resulting from the creation of these active objects and their interaction with one another [Campbell et al 93].

Concurrent features are built using language classes and subclasses. The concept of process is introduced here, the class Processor encapsulates and represents interrupts and its handlers. The subclass VCProcessor specialises these methods to catch signals representing interrupts. The advantage here was that concurrency was added to object oriented programming by developing libraries that supported missing features like
concurrency and benefits of inheritance and polymorphism were not compromised [Campbell et al 93].

4.4 Concurrent OO programming on Transputers

Transputers have been developed to build powerful parallel computers as transistors were used to build electronic systems. The name transputer is an acronym of two combined words viz TRANSistor and comPUTER. Conceptually it is similar to software objects which are connected with each other to build powerful application programs.

With the increasing interest in concurrent object oriented programming, the union of object oriented paradigm with the process paradigm is fast moving to the parallel environment of transputers and Occam [Thomas 89], [Gray 91], [Fakamuria 91], [Corradi et al 92].

4.4.1 Parallel Objects

Thomas [Thomas 89] has proposed a model for fusion of Occam process with objects. He has first listed some of the differences between Occam processes and objects, for example that objects are not inherently concurrent as Occam processes are, and that the concept of inheritance in the object paradigm is not supported in the Occam process paradigm. He then lists some similarities like equating the object class with Occam process and that objects are driven by messages and Occam processes by communication. With this he comes up with the idea of fusion of object with Occam process whereby parallel objects are obtained by encapsulating object instances in Occam processes [Thomas 89].

Parallel objects are not only instances of their class but also capable of parallel
execution. Each parallel object is connected with its class from which it has been created. It shares with the other instances the method codes. This leads to the coresident constraint between any instance and its class.

Parallelism in a geometric problem is primarily concerned with objects and the placement of those objects within the system. Transputers (T800) are an example of processors with limited connectivity of four links. Operations on links can proceed in parallel with normal execution. Parallel objects incur both execution and communication costs. To minimise the communication time, clustering of parallel objects is resorted to.

Parallel objects are first grouped into clusters and an allocation of clusters to processors is then done. Instances of the same class are grouped together with respect to the coresidence constraint. Further aggregation is achieved by considering pairs of previously built clusters in order of decreasing communication flow. In the mapping phase, these clusters are allocated to the available processors. The aim is to optimise the resource usage that is both the execution and communication operations. The disadvantage here is that it does not lead true concurrent execution.

Since Occam is a static language, the creation of new instances is not possible and instances of process are defined at compile time through the use of PAR statements.

4.4.2 Dynamic Objects

Due to the static nature of Occam, an environment was proposed by [Thomas 89] called Object Manager. [Siet-Leng 91] have also proposed the concept of object manager and execution manager to implement the dynamic nature of objects.
4.4.2.1 Object Manager

An environment must be provided that will do checks at run time and manage the changing network of objects. Objects send messages to this run time system, which performs object creation and object passing on behalf of the requesting object. This environment can itself be structured as an object called the class object to which all objects have access. This object is called object manager. It manipulates objects by methods written in it for creating and killing objects as instances of classes [Thomas 89].

Object manager is like the kernel of a system which deals in objects. It is an object with a number of methods to introduce new classes into the system and remove not needed ones. The data and methods of these dynamic objects are stored in a database. If a new class has to be created, then new data and methods of this class are added to the database and if a class is to be deleted, the data and methods of the class are deleted from the database.

4.4.3 Integrating inheritance

There are different ways in which the sharing of information can be implemented. The strategies by which knowledge can be shared are inheritance and delegation. The inheritance mechanism is a very successful way of incorporating code sharing in programming languages. The inheritance strategy of sharing information, relies on the assumption of shared memory model. We here want to explore a strategy of sharing information in a distributed memory model. The delegation strategy is a suitable candidate, as it is free from the assumption of shared memory model. Sharing of information as achieved by inheritance, can also be achieved and implemented by delegation. In this connection, [Thomas 89] defines the superclass in the subclass. The channel pair to and from superclass are defined in the subclass and are used to delegate
the requests to the superclass from the subclass [Thomas 89]. The methods in the superclass are shared by the subclass, as instances of object subclass can send a message to the instances of object superclass, which then executes the method and sends the result back.

In Fig 4.4.3 below an object (a₁) issues an ask request to another object (a₂) and object (a₂) cannot satisfy it, it then delegates the request to another object (a₃) and this goes on until object (aₙ) is able to satisfy the request and reply back to object (a₁). The delegation scheme is implemented by communication, the delegation of task to an object is performed by message passing. This scheme is suitable for distributed memory model of Transputers. The variable and the method to be activated is performed by message passing, both are delegated to another object, where the method with the given variables is activated.

![Fig 4.4.3 Inheritance implemented by delegation of messages.](image-url)
4.4.4 Message Passing

Objects are driven by messages as processes are by communications. Messages are in effect a specialised form of communication. Each message follows a protocol which is generic among all objects. [Thomas 89] says that in Occam messages should be coupled with result as communication in Occam is synchronised by acknowledgment.

In a multiprocessor system, message passing is necessary for the exchange of data between different processors. Thus there is a need to have a high degree of interprocess connectivity. However, the present generation of transputers offers only four channels for interprocessor connectivity. On the other hand the concept of virtual channel communication between processors can offer an unlimited number of channels [Knowles 89] and it also frees the programmer from concern about the physical connectivity between processors.

4.4.4.1 Message Routing for Transputer based systems

A message router is a tool that hides the physical architecture and offers a high level interface to a user and makes the machine topology irrelevant from the programmer's point of view. It also increases code portability and reusability and uses routing algorithms.

The routing algorithm determines a path between two nodes that are not directly connected. The routing function $R: N \times N \rightarrow C$ maps the current node $n_c$ and the destination node $n_d$ to the channel $c_n$ on the route from $n_c$ to $n_d$ where $R(n_c, n_d) = c_n$ [Talia 93].

A message routing system allows communication between any two processes of a concurrent system wherever they may be located. It separates the hardware architecture
from the software configuration of the system and makes it easier for developing concurrent software. Some of the important features of a routing system are:

Deadlock freedom: A deadlock in a routing system is a condition when no message is buffered at its destination and no message can advance as all buffers in each routing path are full. Deadlock also occurs when there are cyclic paths in the network. The routing system should ensure deadlock freedom.

Network latency: Network latency is the time it takes for a message to leave the source and reach the destination. Network latency for a routing system should always be low.

4.4.5 Mapping OO concepts on non OO Languages

OO design does not require an OO programming language to implement it. OO concepts can be mapped onto a non OO language and still have the benefits of OO analysis and design. This mapping onto a non OO language like Occam is done in following steps:

4.4.5.1 Translate classes into processes

Occam modules are encapsulated in the form of objects, which are then distributed onto transputers. These objects form permanent threads. They have all the functionalities of a processes as well as modularity and encapsulation of an object. This can be regarded as integration of the concept of an object in OOD and the concept of process in the concurrent design. The concurrency here is expressed at the level of objects and these are called process objects. The design is decomposed into grain size objects. The objects requesting services are fine to medium grained and objects providing services are medium to large grained.
4.4.5.2 Pass arguments to methods

Functions in objects are called methods. Objects interact by sending messages to each other. A message is in fact a request to an object to execute one of its methods for a certain value of parameters. These parameters or arguments are passed to the method and the method is evoked. For example a message sent to an object to evoke a method with given arguments and the recipient object receiving this message is given below.

```
outchan ! method_m; arguments
outchan ? method_m; arguments
```

This results in sending a message to the object to evoke method_m with the given arguments.

4.4.5.3 Allocate objects

Objects are allocated statically by defining the processor on which the object has to reside. While allocating objects onto processors, maximum use of the physical architecture is made by carrying out load balancing.

4.4.5.4 Implement inheritance by delegation

When an object receives a message which it cannot answer because of lack of knowledge, the object delegates the message to another object. Delegation provides a way of extending the behaviour of an object. New objects can be added to the existing system thereby achieving extendability. Delegation replaces the concept of class, subclass and instances. Thus the concept of inheritance and polymorphism can be implemented by delegation of messages. Advantage of inheritance like code sharing is also achieved.

Delegation is implemented by message passing. At runtime object will delegate the
message to an other object, if it can not answer the given message. This scheme is suitable for distributed memory model eg Occam/Transputer model because it is implemented by communication. The delegation of a task to another object is performed by message passing. The arguments and the name of the method is delegated to the target object, where the given method is activated.

4.4.5.5 Deal with concurrency

Objects allocated on different processors execute concurrently. They exchange messages between themselves. Objects working concurrently can form threads of parallel execution.

4.4.5.6 Encapsulate the internal details of classes

Objects are units of protection. The data that is internal to the object can only be accessed by the object itself. The outside world does have access to the object class by sending a message to the object. Encapsulation and protection is implemented here and the decision to execute the message received is totally up to the object.
5 OBJECT ORIENTED APPLICATION DEVELOPMENT FOR TRANSPUTERS

One small step for man, one giant leap for mankind.

- Neil Armstrong

5.1 Introduction

To exploit parallel computation and object oriented concepts, an application needs to be chosen that has both OO and parallel features. Banking applications have been selected by some authors to explain OO concepts [Henderson - Sellers 90], [Rumbaugh et al 91]. Further, banking systems exhibit parallelism [Lewis et al 92]. In this thesis also a banking application using OO concepts and parallelism with transputers has been developed.

5.2 Banking application

In any real time computerised banking system, customers deposit checks or cash into one or more accounts involving the services of the ATM or cashier. Objects in the system can thus be identified as customers, ATMs, cashiers, bank computers, accounts, central computer etc. An interaction exists between these objects and they have inheritance hierarchies. The system exhibits parallelism with parallel threads of execution. A basic bank system is shown in the next page.
5.3 Methodology to develop Concurrent Object Oriented Design

A design methodology for the subject banking system application has to be developed which encompasses both OO and concurrency features. The approach adopted is a cross between methodologies developed for OOD by authors [Booch 91], [Meyer 88], [Rumbaugh et al 91] and [Henderson - Sellers 90].

The real world is made up of objects which behave like processes. Since it is difficult to carry out process structuring before objects are identified, classical object oriented design has been carried out first, followed by process identification and structuring for achieving concurrency.
Classical OOD: Its steps are as follows.

1. Object identification
   a) customer
   b) cheque account
   c) passbook account
   d) savings account
   e) term account
   f) cash
   g) ATM
   h) bank computer
   i) transactions etc

2. Interaction. Develop interaction between different objects.

3. Attributes and operations. Add attributes and operations to different objects.

4. Examine possible aggregation, clustering and generalisation.

Aggregation and clustering involves identifying the existence of relationship and construction of network of clients and servers that are closely related, whereas generalisation involves construction of inheritance hierarchies in the form of classes and their subclasses.

The idea of subsystem [Wirfs - Brock et al 90] has also been followed, which is similar to that of [Meyer 88] who has referred to it as clusters of objects. Objects on one subsystem have many communication channels between themselves.
Process identification and structuring for concurrency: Its steps are as follows.

Objects from the object oriented model have to be identified as processes. Processes are objects at which the activity takes place. These objects have to be identified. Activities (routines) are specific to the object (class). Process is an object executing a prescribed behaviour [Hoare 85]. All objects are thus potential processes. Objects in the object oriented design, which have behaviour and can be executed are identified as potential processes. The object ATM identified can also be identified as a process as it has a behaviours for eg check the password.

2. Concurrent threads identification.
The state diagram of objects that exchange events and messages between themselves can be grouped together as a single thread of control. A thread of control is a path through a set of state diagrams on which only a single object at a time is active. These threads of control can thus be identified in the application.

In a thread, the execution is sequential as only one object is active at a time. Execution starts with one active object sending a message to another object which then becomes active on receiving it. Such concurrently executing threads are required to be identified in the design.

3. Shared objects.
System designed should also deal with the problem of shared objects eg two customers having a joint account.

4. Processes and parallel threads identified.
Processes and parallel threads identified are to be placed and structured on
transputer processors to achieve required concurrency and efficiency.

5.4 Mapping

Concurrent OO approach as covered above is readily adaptable to the development of large applications for distributed memory parallel systems. The banking application developed in this thesis which basically represents a large system has been mapped theoretically onto Occam 3 language and T9000 transputer following the steps listed above. This was done due to the fact that the subject language and hardware are still in development stages and are not available. However, the working of the banking application developed has been successfully tested and demonstrated on Virtual Channel Router which emulates transputer T9000 as described in chapter 7.

5.5 OO concepts supported in the application developed

OO concepts that have been supported in the application developed are listed below:

Object: An object is an entity that contains attributes and methods. It could be a Bank, ATM or an account. For example the object ATM would encapsulate attributes and it methods like check the password, valid account, etc.

Class: A class defines the structure and capabilities of an object instance. It includes the state data and methods for the instance of the class eg an account class may have methods to allow deposits and withdrawals.

Instance: An instance is an object eg a person's account at a bank is an instance of the account class. It is possible to deposit or withdraw an amount from the instance of account class.
Messages: Objects communicate via messages eg when a customer requests the balance in his account, the object customer sends the request (message) to the object bank which sends the request to the object account.

Methods: These are services or behaviour of the class which are activated when the object receives the message eg withdrawals and deposits are methods of class account.

Inheritance: Here the lower classes can use the services of the higher classes in the hierarchy. It is a way of reusing services and data. In this application, the savings and cheque accounts inherit the services from class account.

Polymorphism: This is the capability of a single variable to refer to different objects that fulfil message protocol responsibilities. In the banking system, the instance variable account can be referred to savings or cheque account. No matter which type of object this variable refers to, it can send withdraw or deposit fund messages.

![Fig 5.5 Polymorphism: Same instance variable account, different object types saving and cheque accounts.](image-url)
5.6 How does this application exploit parallelism?

Most real world banks run on the concept of MIMD parallelism where many processors simultaneously execute different instructions on different data. In our application, the banking tellers (ATMs) are like processors. When the tellers are more than one, the banking system operates like a multi-processing system with ATMs and accounts communicating by message passing. The customers using the ATM are like tasks. When the customers are more than one, then the tasks have to be distributed among different tellers. Thus load balancing becomes necessary for efficient operation. The application developed thus represents a parallel system whose design can be implemented in parallel languages like Occam, Linda and Communicating Sequential Processes (CSP) [Hoare 85].

The application also deals with the problem of shared resources. For example in the subject system, a joint bank account is a shared resource. Semaphores have to be used to deal with the problem of joint bank account whereby one thread of transaction waits while the other carries out an action on the joint account.

Transactions like withdrawal or deposit are carried out by different customers on separate ATM concurrently. The application developed in this thesis has fully exploited all the above listed feature of parallelism.
6 CONCURRENT OBJECT ORIENTED DESIGN

A new scientific truth does not triumph by convincing its opponents and making them see the light, but rather its opponents gradually fade away and a new generation grows up familiar with it.

- Max Planck

6.1 Introduction

Systems in the real world are made up of objects and these objects have a behaviour pattern which is called process. Some of these processes are executed concurrently and communicate with each other by message passing. Most real world systems are therefore object oriented and concurrent.

6.2 Concurrent Object Oriented Design

The powerful concept of object oriented design is considered optimal for parallel computers. A methodology that integrates the object oriented and concurrent design approaches has been developed in previous section 5.3 and has been used to develop the design given below.

6.2.1 Object identification

We start the design process by identifying possible objects in the problem domain as stated in section 5.3. It is similar to the methodology followed by [Rumbaugh et al 91], [Booch 91] and [Henderson - Sellers 90] in which nouns are identified in the problem definition. Some of these nouns which have been identified in this application ie customer, ATM, bank etc are then classified as objects.
6.2.2 Object interaction

An interaction is developed between the different objects that have been identified as shown below.

![Diagram of object interaction](image)

*Fig 6.2.2 Interaction between objects identified*

The above representation of the application has been modelled using object model notation of [Rumbaugh et al 91]. The rectangles represent the objects and the lines connecting the objects represent the interaction between them. The degree of associativity is indicated by a dot at the end of the line. One black dot indicates multiple associativity and absence of dot indicates single associativity. The objects that have been identified in the above figure are customer, ATM, bank and account.

6.2.3 Attributes and operations

The next step in the design process is to identify attributes of the objects as shown in the figure on the next page.
Fig 6.2.3 Attributes of the objects

In the figure above for example object customer has attributes name and address.

6.2.4 Generalisation

The next step is to add inheritance in the design. It can be added in two ways, firstly by generalising the existing class into a superclass and secondly by refining the existing class into subclasses. The object class account is refined into subclasses check account and savings account as shown below.

Fig 6.2.4a Class account is a superclass cheque & savings accounts
The object model with attributes and inheritance is shown below.

![Object Model Diagram]

The object model with inheritance is constructed from the figure 6.2.3. In the object model with inheritance, the account class has been refined into subclasses savings account and cheque account using the object model notation [Rumbaugh et al 91].
Definition of some of the object oriented classes used in this application are:

1 Account class
   Superclasses: None
   Subclasses: Savings Account, Cheque Account
   Public methods:
   - withdraw
   - deposit
   - balance
   Responsibility:
   - balance: Return current balance
   - balance: Amount: Set current balance to Amount.
   - deposit: Amount: Add Amount to current balance.
   - withdraw: Amount: Subtract Amount from current balance.
   Data:
   - balance
   - withdrawal limit

The class account is an example of super class formed by generalisation. The method withdraw, deposit and balance were present in the class cheque and savings accounts. These general and common methods are used to form a new class account which is a super class of savings and cheque accounts.

2 Savings account class
   Superclasses: Account
   Subclass: None
   Public methods:
   - credit interest
   Responsibility:
   - credit interest: Add interest for last month

3 Cheque account class
   Superclass: Account
   Subclass: None
   Public method:
   - interest
   Responsibility:
   - interest: Add interest for the last month
3 Customer_class
   Superclass: None
   Subclass: None
   Responsibility
   - withdrawal limit reached: Amount
   - valid password: Password
   - name
   - address
   Data
   - password
   - accounts
   - name
   - address

6.2.5 Process identification

An important goal in concurrent object oriented design is to identify objects in the OO model that can become active concurrently. Most object classes can be considered as potential processes as they can become active. Technically all processes are subsets of object classes. These identified objects correspond to processes found in most concurrent programming languages. In this connection, Hoare says that a process is an object executing a prescribed behaviour [Hoare 85].

These active objects or processes drive the data flow graph by producing or consuming values. Active objects have their own computational power and agenda. The activation of a method in a object by a message is a creation of processes. A typical example of active object would be an ATM. The process would describe the algorithm that governs the ATM eg check the pin number, check withdrawal amount does not exceeds the maximum limit, carry out the transaction etc. In object oriented programming this would just be one of the features associated with the ATM. Processes are implemented as methods on object class. Some authors refer to active objects as actors [Agha 89].
Processes identified in the object oriented model are customer, ATM, bank, account. The object customer has behaviour methods like withdraw or deposit cash, input the password etc. Activation of these methods, would make the object customer active. This active object can then be identified as a process, as the algorithm for withdrawal and deposit of cash would be governing the process. Some of the processes and their governing algorithms identified in our object oriented model are:

Process customer,
Algorithms
- input the password,
- account, amount

Process ATM,
Algorithm
- Check the password
- Give the cash

Process Bank,
Algorithm
- check the Account

Process Account,
Algorithm
- Withdraw method
- Deposit method

In summary, after completion of the classical OOD methodology, process identification and structuring for concurrency are carried out. Active objects are identified and a model is developed. Active objects form continuous processes. Completion of activity in the active object results in implication on others. Active objects in this design are identified as customers who implicate on potential active objects like ATMs which in turn activate object bank and so on. Active objects thus form a thread of execution.
6.2.6 Concurrent threads identification

Concurrent threads are based on tasks. These task result in activating objects which in turn implicate other objects. Tasks in the our application are transactions carried out by customers. Once the task has been activated by a customer, it will be executed concurrently with other threads which have been activated by other customers.

6.2.7 Shared Objects

One of the important features of concurrency is to deal with shared objects in multiple threads of execution. In this example if there is a shared joint account, then one
transaction will be executed and thus locking the object shared joint account, while the other transaction will wait till the first transaction has been completed. Objects don't execute more than one action at the same time, also they change to waiting mode to achieve synchronisation.

Furthermore, if a customer decides to use the ATM while the bank computer is updating the customer balance due to a cheque withdrawal or deposit, then in that case the transaction by the customer will be asked to wait till the bank computer has updated the customer's balance. This can be implemented by the lock and key mechanism using the semaphores, whereby the account is locked when the bank is accessing it and the account and its transactions are not available to the customer through ATM during this period.

6.2.8 Mapping on Transputers

Active objects and threads are then mapped onto distributed transputers/virtual processors and communicate with each other through unlimited number of virtual channels as shown in the diagram on the next page.
Fig 6.2.8 Mapping of objects & threads on transputers/virtual processors
6.3 Advantages of OO in this application

In this application the advantage of extendability and reusability is achieved by replicating the objects and reusing them. New objects like customers and ATM can be added to the application by restructuring and reusing the previous design. Reusability is also achieved with inheritance because methods like deposit and withdraw don't have to be duplicated in the subclass of class account.

6.3.1 Exploiting inheritance

The concept of inheritance can best be used when the program is sequential. One of the options is that the classes and subclasses should be on the same processor (transputer) and should be sequential thus insuring advantages of code and design reuse. Since there is a lot of communication between these objects, it is efficient to place them on the same processor and this ensures that the advantage of code sharing is not compromised by communication cost.

Inheritance and parallelism can then be exploited by delegation of messages. With the advantage of the router in the next generation transputer T9000, it would be feasible to incorporate inheritance by delegation of messages. In doing this, we should aim to get the optimal balance between code reuse and speed thereby insuring that the advantage of one is not negated by the other. In our application the methods withdraw and deposit in the object account are shared by savings and cheque account.

6.3.2 Polymorphism

We have discussed above how the banking example supports the concept of polymorphism. We will now illustrate how this concept can be implemented.
Let the object account send a withdraw message to the subclass object's cheque and savings accounts. Each of them has to interpret this message differently. Withdraw is a polymorphic function as it can operate on arguments of different types and the object account can take on polymorphic forms like cheque and savings accounts.

**Procedure withdraw (Account ......)**

Begin

PAR
SEQ
send message to object cheque account
and evoke procedure withdraw cheque account.

SEQ
send message to object savings account
and evoke procedure withdraw savings account.

End.

**procedure Withdraw (cheque Account....)**

Begin

Maximum amount that can be withdraw is Max_cheque.
Minimum balance should be Min_balance_cheque.
balance := amount - withdraw_amount

End.

**procedure Withdraw (Savings Account....)**

Begin

Maximum amount that can be withdraw is Max_Sav.
Minimum balance should be Min_balance_sav.
balance := amount - withdraw_amount

End.

### 6.4 Parallel features in the design

The design illustrated above has both functional and geometric parallelism. These are covered in the succeeding sections.
6.4.1 Functional parallelism in the design

Thread or functional parallelism involves threads and their movement through the system. Threads of control and related activities is a job. Both job and threads are active objects. A thread carries out invocation on different objects. An object may restrict access to a single thread at a time or it may allow multiple threads to execute concurrently in which case semaphores are used for internal synchronisation. The algorithm can be broken down into a pipeline of processes through which data flows. Each customer forms a thread of execution. These parallel threads through which the data flows exploit function parallelism in the design.

The design can be broken down into concurrent operations and each processor implements a small part of the total design. A common feature here is the construction of pipes of processors. In such a decomposition of the problem, the data flows between the processing elements.

Objects communicating by messages, and pipes of processes are related, as both are modelled on message passing. In section 4.2 the relationship between process and object is explained. Pipes or threads are formed by the communication between the processes. These threads of processes can be related to the data flow achieved by message passing in the object oriented model.

Form the object oriented model, functional or algorithmic parallelism can be identified by breaking the problem into objects and the flow of data between them. In the application developed, the objects that have been identified are object customer, ATM, bank and account. The data flow between these objects results in the construction of pipes. The algorithmic or functional parallelism specifies the pipes or the data flow between the objects and the OO model specifies the objects.
6.4.2 Geometric parallelism in the design

Parallelism in geometrical problem comes from being able to partition the data into a number of sections and to operate on each section in parallel. This parallelism is primarily concerned with objects and their placement within the system. The problem is broken down into a number of similar processes and each operates on a different set of data. In this design similar processes are ATMs and different set of data are the customers.

Objects are used to define the geometric structure of the application. Objects can thus be distributed uniformly on the processors in a network and would be exchanging data with the objects on the neighbouring processors.

Encapsulation makes objects a convenient elementary structural entity for distributed and parallel systems [Beccard et al 90]. Objects can thus be distributed to execute freely on processors. This distribution method eliminates the possibility of idle processors and thus yields a fine grained parallelism.

In summary, geometric parallelism can be identified from the object oriented model by breaking the problem into objects and distributing the objects onto the processors. In the application developed, objects customers and ATMs are distributed across the processors as shown in the figure 6.2.8 to exploit the geometric parallelism.
7 IMPLEMENTATION

"Would you tell me, please, which way I got to go," asks Alice
"That depends a good deal where you want to go," replies the Cat
- Lewis Carroll, Alice in Wonderland

7.1 Introduction

In this chapter the working of the banking application and how the design has been implemented are given. Before proceeding to implementation, the transputer T9000 [May et al 92] and Virtual Channel Router (VCR) [Debbage et al 91a], [Debbage et al 91b], [Debbage et al 91c] are described, as the design has been implemented on them.

7.2 Transputer T9000

IMS T9000 [Inmos 91] is the latest member of the transputer family developed by Inmos Ltd. In its development, they have used advanced CMOS technology to integrate a 32 bit integer processor, a 64 bit floating point processor, a 16 Kbytes of cache memory, a communication processor and four high bandwidth serial communication links on a single chip. It has a RISC based architecture.

The T9000 transputer excels in real time embedded applications delivering exceptional single processor performance and scalable multiprocessor capabilities. It is binary compatible with the previous transputers. It decouples the physical connectivity of a system from its logical connectivity. Thus between two directly connected T9000 transputers, an unlimited number of virtual channels can be established. Its link system also enables the transputer to be connected via a network of C104 packet router which allows virtual channels to be established from any transputer to any number of other transputers.
7.2.1 Performance

The T9000 transputer has exceptional single processor performance. Its new superscalar CPU is capable of a peak performance of 200 MIPS and 25 MFlops. It offers sub microsecond interrupt response and context switch times, making it ideal for high performance real time systems. Its four communication links provide a total of 80 Mbytes/sec bidirectional bandwidth. The interprocessor communications architecture provides scalable performance as the performance of the system can be increased by adding more processors. Transputer T9000 performs 2.5 times as many data accesses per cycle as compared to T805.

7.2.2 Architecture

The T9000 comprises a pipelined superscalar architecture having a CPU with a FPU, a communication processor, four communication links, control unit with its associated links, an external memory interface and on chip cache memory.

One of the important design decisions of the transputer T9000 was that it should be programmable in a high level language. The instruction set has been defined for simple and efficient compilation. It contains three registers viz. A register, B register and C register which are used for expression evaluation and form a hardware stack. The transputer also uses three other registers when executing codes viz. instruction pointer, work space pointer and operand register.

The T9000 has the ability to effectively run several tasks or processes concurrently. The processes are created and scheduled by the host operating system which provides the ability to the processors to communicate with each other and with the operating system.
The CPU of the transputer uses 32 bit linear addressing and has up to 4 Gbyte memory. It implements the same instruction set as the earlier version transputer T805. The T9000 includes a hardware kernel for scheduling processes and performing communications. These operations are directly supported in the instruction set. It also has a cache memory.

The instructions are executed in a five stage pipeline. The first can fetch two local variables, the second can perform two address calculations for accessing non local or subscripted variables, the third stage can load two non local variables, the next can perform an ALU or FPU operation and the final stage can do the conditional jump.

7.2.3 Communication support device

Its communications support device C104 ensures that any size of T9000 system can be constructed including connections to first and second generation transputers. The C104 chip allows the construction of efficient communication networks whereby communication channels can be specified between processors regardless of their physical location.

The C104 is a complete 32 by 32 nonblocking crossbar switch with microsecond latency. It allows fast communication between T9000 transputers that are not directly connected. A number of C104s can be connected together to make larger and more complex networks.

7.2.4 Advantages of T9000

There are some problems in program development for present generation transputer based multiprocessor systems as the programmer has to be aware of the underlying hardware and use this knowledge in application development. The T9000 has simplified
this programming complexity as it eliminates the need to map logical communication channels onto physical links.

The transputer toolset is a set of development tools for programming, configuring and debugging mixed transputer systems. The toolset is available on a variety of host computers including IBM PCs, NEC PCs, VAX, SUN 3, SUN 4. The T9000 is supported by a range of compilers like ANSI C, C++, Fortran, Occam, Ada.

The T9000 also provides efficient hardware support for controlling access to a shared resource. This could be a hardware resource or a piece of software running on a particular processor in a network. Transputer links can be used to implement point to point communication between transputers. This allows transputer network of arbitrary size and topology to be constructed. Point to point links have advantages over bus based communication as they are efficient, simple and hardware independent.

7.2.5 Virtual Channel Processor (VCP)

The solution chosen in the T9000 was to add multiplexing hardware to allow any number of processes to use links so that the subject physical links could be shared transparently. These hardware channels which allow sharing of links are known as 'Virtual Channels' and have the same behaviour as the software channels. This multiplexing hardware is called the VCP [May et al 90], [Inmos 91].

The VCP does message routing with packets and also adds a header to each packet to identify the destination process. When the packets are received, the VCP uses the header to send the data in each packet to the intended process.

A communication channel can also be established between any two processes even if they are running on transputers which are not directly connected. The header specifies
the destination of the packet and it is routed by the VCP. The link bus in T9000 facilitates a deadlock free, unacknowledged packet transfer. For each input link from the application processes, packets are demultiplexed according to the destination address contained in the linkbus.

7.3 Virtual Channel Router (VCR)

In the absence of T9000 and C104 hardware, the University of Southampton has developed software called the Virtual Channel Router (VCR) [Debbage et al 91a] which effectively emulates the features of virtual channel communication hardware.

7.3.1 Concept of VCR

The Occam language imposes no restriction on the number of channels connected to each process whereas the transputer network has a limitation of 4 physical links per node. Two communicating processes that are mapped on a single transputer can exchange messages according to the CSP (communication sequential processes) protocol developed by Hoare. The implementation of a full Occam system can be done with the provision of virtual channels and message routing on a transputer network. It is typically an additional set of processes that executes in parallel with the user application and intercepts each message in order to route it to the correct receiver. The Virtual Channel Router software implements these facilities.

7.3.2 What is a VCR?

A VCR is a router that provides communication between processes in a network of Transputers. It was developed as the current generation Transputer T805 did not support virtual channel communication between two processes mapped on two distant transputers. It breaks the message into packets and the physical transputer link is shared
among many virtual channels. Packet store and forward is the method employed by the VCR. Through a pipeline communication, the packet routing tries to hide the distance effect between the sender and the receiver.

The VCR [Debbage et al 91b] software was developed by the University of Southampton within the ESPRIT P2701 PUMA project to provide unrestricted channel communication across networks of T-series transputers. It allows distributed transputer program to be written, compiled and configured in a topology independent format and then bound it to a topology dependent routing kernel at run time.

Some of the features of the VCR are:

- it provides Occam channel communications over networks of 32 bit T-series transputers,

- it is deadlock free and places no restrictions on message size,

- at the configuration level, link placement is eliminated and the processors valency limit removed,

- VCR user programs are completely independent of the target VCR network,

- debugging is supported using the standard Occam toolset,

- programs written in VCR are portable because the topology details are hidden,

Occam channels come in only two varieties viz soft and hard. Soft channels are used to implement internal processor communication and require memory. Soft channels are bounded only by memory availability and thus arbitrarily complicated topologies can be
formulated. Hard channels are used for communication between transputers and are bound by links fabricated on the transputer.

Under virtual channel router, static virtual channels are declared at the configuration level with channel ends in different user processors. Channels implemented by this soft channel mechanism, may require run time checks to ascertain their nature. In this virtual channel routing system, a kernel process is given control of some (or all) of the links and implements packet routing across the network to give many virtual channel across the physical links. VCR also incorporates a graceful error reporting mechanism.

7.3.3 Configuration Files

Virtual channels configuration language is very similar to that used for standard tools configuration but without channel placements. Virtual channels are declared at the outermost level and placed on processes in the conventional manner.

The difference between the VCR running on T800 networks and T9000/C104 system is that, of necessity, intermediate T800s must use store-and-forward routing. The communication performance of the VCR in terms of latency and effective bandwidth will be improved on the T9000/C104 system.

7.3.4 Working of VCR

In architecture independent programming, the programmer does not have to worry about the size and the connection topology of the physical processor network. In Occam, parallel processes may communicate by sending and receiving messages via software channels. These channels can be assigned to a physical link of the transputer to allow interprocessor communication. But the problem is of limited number of links (4 per transputer). Thus all the user processes cannot be directly connected in a network of
physical processors.

A solution is found by building a communication harness for routing the message packets between the user processes. The harness is a simple software module which can be added to the existing program. Each processor has a router processor which communicates with the user process and with every processor link. The router may receive a message from any link or from the user process. It then checks the destination and the message is directed via the optimal route towards its destination.

The transputer network is explored and the routing table is calculated. After the exploration phase is over, router processes compute the minimal paths between all virtual processors using a distributed algorithm. In this connection 'n' virtual processors are mapped onto a network of 'm' physical processors where 'm' is less than 'n'. The routing table is a look up table. The routing of messages is based on point to point strategy. The sender specifies the destination processors and the destination process. The message packet is transferred to the destination only.

7.4 Design implementation

The design was implemented in VCR version 2.0k [Debbage et al 91d] and new Occam tool set D7205. Configurations of one, two and four transputers of T800 series connected to a host computer was used for implementation and testing. The test results and the configuration of transputers are placed at Appendix B.

7.4.1 Implementation of OO concepts

Though Occam is a non object oriented language, object oriented features can be incorporated in it as given in section 4.4.5. The implementation of OO concepts is covered in the sections below.
Object oriented concepts are represented in the form of object class. Object class are similar to process concept in parallelism. Processes are used to build a concurrent system. Unification of object oriented and parallelism results in the unification of class and process concepts.

A process is an object but not every object is a process. Thus objects can be of two types viz process objects or objects. A process object is an instance of a class inheriting from the process and active by itself, whereas an object is one that is waiting for a call to execute its routines. When an object has a reference to a process, it communicates with that process by the inter process communication mechanism.

The codes that are used to represent an process object class in Occam are listed below:

PROC Process_Object( CHAN OF PROTOCOL message_to_object, message_from_object)

INT x,y: -- Private Data

PROC PrivateMethod()

...

PROC PublicMethod1()

...

PROC PublicMethod2()

...

SEQ
message_to_object ? CASE -- message selector
tag1;........
PublicMethod1()
The class defined above is an Occam procedure with the name Process_Object. Messages are received by object class. Methods are evoked depending on the type of message received by the object class. In the code above, if tag1 is specified in the message then PublicMethod1 is evoked, if tag2 is specified then PublicMethod2 is evoked.

The instances of the object class are defined in the configuration file which is listed below.

-- Include Files

-- Constant data

-- Communication Channels

PLACED PAR
  PROCESSOR 0 TA
  customer1( fs,ts, ca[0], ac[0], stopper )

PROCESSOR 1 TA
  customer2(fs,ts,ca[1],ac[1], stopper )

...  
 ...

PROCESSOR 8 TA
...
...

PLACED PAR i = 0 FOR num.nodes-1
  PROCESSOR (i+8) TA
  atm (fs,ts, ca[i], ac[i], bat[i], atb[i])

In the above configuration, there are 8 instances of object class ATM. It may be noted that more objects which are static can be added to it. The design is similar to a digital
system design wherein channels are like pins and objects are components or ICs. The application developed is not only extendable but also reconfigurable. Before extendability can be achieved, the application developed must be reconfigured. Communication channels between objects are reconfigured for new objects that are added.

7.4.1.2 Inheritance implementation

The advantage of inheritance is code sharing. Inheritance and polymorphism have been implemented by delegation of messages as done by other authors in building parallel systems. The implementation of super class account is shown below. The methods withdraw and deposit in the super class account are shared by subclasses savings and cheque accounts.

PROC account([], CHAN OF ASACC account.to.savingsacc,
             [], CHAN OF SACCA savingsacc.to.account,
             [], CHAN OF ACACC account.to.chequeacc,
             [] CHAN OF CACCA chequeacc.to.account)

PROC withdraw([10] BYTE account, INT amount, INT accno)
    ...
    : PROC deposit([10] BYTE account, INT amount, INT accno)
    ...
    : SEQ
...
SEQ i=0 FOR SIZE savingsacc.to.account
SEQ
    savingsacc.to.account[i] ? action
    SEQ
    IF
    eqstr(action,"withdraw ")
SEQ
withdraw(account, amount, accno)
eqstr(action,"deposit ")
SEQ
deposit(account, amount, accno)

SEQ i=0 FOR SIZE chequeacc.to.account
SEQ
chequeacc.to.account[i] ? action
SEQ
IF
eqstr(action,"withdraw ")
SEQ
withdraw(account, amount, accno)
eqstr(action,"deposit ")
SEQ
deposit(account, amount, accno)

:

In the subclasses savings and cheque accounts shown below, the super class account has been defined in the channel declarations. The channel declarations which specifies the superclass account are to.account and from.account.

PROC savingsaccount ([] CHAN OF TOACC to.account,
                    [] CHAN OF FROMACC from.account)
    to.account ! action
:

PROC chequeaccount ([] CHAN OF TOACC to.account,
                    [] CHAN OF FROMACC from.account)
    to.account ! action
:

Polymorphism is implemented via inheritance as the variable account can represent savings or cheque accounts.
7.4.2 Working of the application

The application developed was mapped onto virtual processors. As mentioned earlier, VCR allows the design to be mapped onto various configuration of transputers irrespective of the actual number of processors available. This promising capability of VCR can facilitate the development of massively parallel applications. Further while writing codes, programmers don't have to worry about the actual number of processors available. The application developed is also fault tolerant and topology independent.

Fig 7.4.2 Application on one Transputer
The design as explained in chapter 6 has been implemented. The classes identified in the
design were mapped and allocated on the processors as shown in the Fig 7.4.2. As can
be seen, infinite number of processors can be specified and applications for massively
parallel architecture can be developed.

The object customer1 has been placed on the processor 1, ATM on processor 8 and the
other objects as shown in the Fig 7.4.2. Customer objects become active immediately
on start of the application which in turn activates ATM objects. These objects which
have been activated, execute in parallel and communicate by message passing through
virtual channels forming parallel threads of transactions. In a single thread of
transaction, object customer sends a message to the object ATM specifying the type of
transaction, account and password. A typical message could be:

to.atm ! transaction; account; password

Once the ATM has received the message, it checks if the customer is a valid by verifying
the password. This active object then carries the transaction further by sending the
message to object bank. Object bank also receives messages from other ATMs through
virtual channels which constitutes parallel threads of execution. On receiving the
messages, the object bank checks the type of account specified by the customer. The
message specifying the transaction withdraw or deposit is sent to this object account. In
case there is a joint account, one thread has to wait while the other proceeds. Once the
transaction has been completed, the results are printed. Multiple parallel threads of
transaction have thus been executed in our application.

The application was developed initially to work on one transputer. If the application
works on one transputer, then it can be considered to work on any configuration of
transputers, as the application has been developed using VCR which makes it topology
independent. The objects on the virtual processors are then distributed on transputers in
7.5 Design revisited

In the above given implementation, the object bank receives messages and carries out requests from other object ATMs through virtual channels. The object bank has access to object ATMs and object Account through unlimited number of channels. The virtual channels, a new feature in T9000 eliminates the constrains and complexities of programming. In the code given below, the object bank acts on each request sequentially, which it receives. This is because the single object bank can perform a single sequence of action in response to one request.

```plaintext
SEQ i=0 FOR SIZE atm.to.bsys
  SEQ
    atm.to.bsys[i] ? CASE
      noacc;accno;password;acc;act;amt
```

Each request is then processed, a method is evoked and the bank object becomes active. As, only one method can be active at a time, the request are processed and acted on sequentially.

```
PROC banksys([] CHAN OF BSYSATM bsys.to.atm,
                [] CHAN OF ATMBSYS atm.to.bsys,
                [] CHAN OF BSYAC bank.to.account,
                [] CHAN OF ACBSYS account.to.bank)

  -- {{{ check password
  PROC check(INT password, INT accno, INT validity)
    check if the pin number entered is valid
    :
      -- }}

Include library files
Declare the variables and there types

SEQ
SEQ i=0 FOR SIZE atm.to.bsys
SEQ
  atm.to.bsys[i] ? CASE
    noacc;accno;password;acc;act;amt
```
SEQ
check if the pin number is valid
IF
(validity < 0)
SEQ
If pin number not valid
stop the transaction
bsys.to.atm[i] ! pworderrorba;perror
(validity > 0)
SEQ
If pin number is valid
continue with the transaction
bank.to.account[i] ! badata;accno;amt;acc;act
account.to.bank[i] ? CASE
balanceacb; bal
SEQ
bsys.to.atm[i] ! balanceba;bal

It can be concluded from the code above, that a single bank object is a serial component in the design thus inhibiting true parallelism. As all ATMs are communicating with one single object, they all have access to it sequentially. The single bank object is a limiting factor and forms the bottleneck of the design. The timing results given in section 9.4, show that there is no performance gain from one transputer network to four transputer network and thus reinforcing Amdhal's law that sequential component is the ultimate limiting factor in the performance of the system.

This serial component in the design can be removed if each ATM has its own bank object. This can be achieved by replicating the bank object. The new configuration file, would replace,

PROCESSOR 9 TA
  #USE "banksys.cah"
  banksys(bat,atb,ba,ab)

with;

PLACED PAR i = 0 FOR num.nodes-1
PROCESSOR (i+1) TA
  #USE "banksys.cah"
  banksys(bat[i], atb[i],ba[i],ab[i])

were num.nodes refer to the number of ATMs.
The bottle neck is eliminated, as each ATM has its own bank object, which forms a complete end to end parallel threads as shown in the figure below.

In the figure given above, the object bank has been replicated and is been executed in parallel by other ATM objects. The revised design now has complete end to end threads which can execute in parallel. Synchronisation is required where the account is being shared. The cheque A/C in the figure above is shared by customer1 and customer2. Sharing is achieved by synchronisation and virtual channels.
8.1 Introduction

Research and development in distributed memory architecture is being directed at developing software environment which can mimic the shared memory architecture to overcome programming complexities of message passing. In this chapter we will examine how this is being achieved using broadcast method of communication in BSP Occam [Allwright 91] with VCR as an intermediate step and the shared channel of communication in Occam 3 [Barrett 90], [Barrett 91].

8.2 VCR as an intermediate step

VCR is an intermediate step towards developing an effective programming environment. VCR solves the problem of limited number of communication channels of transputers by providing unlimited number of virtual channels between them. But this still does not fully solve the programming complexities of message passing. BSP Occam and Occam 3 are possible solutions of this problem.

8.3 BSP Occam

BSP Occam with VCR and Occam toolset provides a global memory and eliminates the requirement of communications by channels as in Occam. BSP Occam differs from standard Occam in that it:
- provides a global memory,

- has ability to synchronise multiple user processes across more than one processor,

- supports PLACED PAR (static and dynamic) to overcome the limitations of standard Occam configuration level.

BSP Occam was designed for use with a programming style in which communication by channels was not required. It was developed within work package 6 of the PUMA project to provide a means of evaluating the PRAM (Parallel Random Access Machine) model which was conceived for SIMD architecture [Allwright 91].

Subsequent work has shown that, subject to conditions, a MIMD machine can implement a shared virtual global memory with the performance characteristics of the PRAM model. The necessary conditions are that each processor is running at least \(O(\log n)\) processes and that interprocessor communication bandwidth scales as \(\log n\) where \(n\) is the number of processors. The term Bulk Synchronous Parallelism (BSP) has been coined to refer to the abstract model shared by all such machines [Allwright 91].

Declaring a sync object or global variable involves broadcasting a message to all servers in the network. If two processors simultaneously try to declare global memory, then it is possible for requests to be received in different orders on different processors, causing an error condition. Hence all arrays should normally be declared on one processor.

An important design feature of the BSP Occam is that any number of user processes may co-exist on one physical processor and can use the library procedures by which
they can all talk to a single server as shown below:

![BSP Occam machine](image)

The global memory is hashed over the processors in such a way that the size of the hashed unit is equal to the unit size of the object declared (e.g., for an array with units of data type [7]BYTE, the seven bytes for each array element will appear continuously on the same processor). Element $i$ of an array is at location $i/n$ of the block on processor $(h(i/n)+i) \mod n$

where \[ h(x) = \text{some hash function} \] [Allwright 91]

In BSP Occam, a server process is present on each of the processors. VCR is used to connect every server to each other. If an element is to be read from the global memory, the procedure `gread.type()` defined in the BSP library is used to send a message to its local server. The server finds the processor on which the memory resides. The server then sends a message to that processor and reads the value.
BSP Occam uses broadcast method of communication. The banking example for BSP Occam can be developed as illustrated below.

PROC customerl (CHAN OF SP fs, ts,
CHAN OF INT startup,
CHAN OF BOOL stopper)

#USE "vhostio.lib"
#USE "bsplib.lib" -- BSP library
#USE "bsplib2.lib" -- BSP library 2

INT a.sync

SEQ
so.write.string.nl(fs, ts, "customerl")
...
...
...
startup ? a.sync

-- gnew.type creates a global variable of type given, the handle to this variable is returned

  gnew.int (h_accno, [1])
  gnew.int (h_password, [1])
  gnew.byte (h_acc, [1])
  gnew.byte (h_actn, [1])
  gnew.int (h_amt, [1])

-- synchronises the handle with the other user processors

gnew.sync (sync, 5)

-- broadcast the handles to all the other processors

gwrite.parameters (a.sync, [haccno, hpassword, hacc, hactn, hamt])
gsync (sync)

-- write value in the variable declared

gwrite.int (haccno, [1], accno)

gwrite.int (hpassword, [1], password)

gwrite.byte (hacc, [1], acc)

gwrite.byte (hactn, [1], actn)
gwrite.int (hamt, [1], amt)

In the object customer as shown above, gnew.int(handle, [1]) creates an handle of the
type integer. Once the handles have been created, they are broadcasted to the other processors. The values for these variables is written by the code gwrite.int(handle, [1], value).

PROC ATM
SEQ
    startup ? a.sync
    gread.parameters(a.sync, parameters)
    VAL INT haccno IS parameter[0]:
    VAL INT hpassword IS parameter [1]:
    VAL INT hacc IS parameter [2]:
    VAL INT hactn IS parameter [3]:
    VAL INT hamt IS parameter [4]:
    gread.int (haccno, [1], accno)
    gread.int (hpassword, [1], password)
    gread.byte (hacc, [1], acc)
    gread.byte (hactn, [1], actn)
    gread.int (hamt, [1], amt)

The code gread.parameters in the object ATM reads the handles broadcasted. Once these handles are read, then their variables are read from the shared memory by the command gread.type() as shown above. This can be read by any other object irrespective of the processor on which it resides.

-- configuration file
#include "hostio.inc"
#include "bspvals.inc"
VAL INT n.servers IS 5:
[n.servers][n.servers] CHAN OF SERV.REQUEST request:
[n.servers][n.servers] CHAN OF SERV.REPLY reply:
placed par
placed par
processor 0 T4
    #use "hosthook.cax"
    #use "customer1.c8h"
    #use "server.cah"
    chan of sp fs,ts:
    chan of bool stopper:
    [1]chan of int startup:
In the configuration file, each object is placed on a different processor. The server on each processor is connected to the VCR. The message to be broadcasted is first sent by the object to its server. On receiving the message the server sends the message to the VCR which broadcasts the message to all the other servers.

8.5 OCCAM 3

Occam programming language was designed by Inmos Ltd. It was first introduced in
1982 and this version was referred to as Occam 1. It was developed along with the transputer. As new features were added to the transputer, the Occam language was accordingly developed further and its current version is Occam 2.

With the future release of T9000 transputer a successor to Occam 2 would be released. This new version of Occam (called Occam 3 in this thesis) would be based on the draft document Occam91 produced by Geoff Barrett [Barrett 91]. Some of the new additions to the language are user identified data types, records and unions. It also introduces the concept of 'shared channels', which provide connection between a single process and any arbitrary number of processes.

A new feature of 'Module' in Occam 3 provides a mechanism for structuring processes in the form of independently compiled object entities. The user communicates with a module via a number of channels and cannot modify the module as it does not have access to its contents. These modules can only be changed by the module designer, further new modules can be plugged into the program without any alteration to the program by the designer.

8.5.1 Shared channels in Occam 3

Shared channels in Occam 3 provide a connection between each process to a number of processes. Unlike the communication channels and call channels which provide a point to point connection, shared channels provide a broadcast connection. Shared communication channels is a way of mimicking a shared memory architecture in a distributed memory architecture. Point to point communication is replaced with broadcast communication.
8.5.1.1 Shared channel communication

In the banking example, the customer object broadcasts the message close account to object bank and account as shown below:

PROC customer

-- Communication channels are shared as records.
CHAN TYPE RPC
RECORD
  CHAN OF INT accountno, result:
:
SHARED RPC close :

SEQ

-- The process has access to the non shared end of the record.
-- This process must first grant the record to the claim process.
-- The message is broadcasted to the other executing processes.
GRANT close
  INT acc_no:
  INT answer:
SEQ
  close[accountno] ! acc_no
  close[result] ? answer

The object customer that is broadcasting the message requires access to the non shared end of the communication. The object customer is first granted the non shared end of the communication by the command GRANT. Once this is done the message is broadcasted as shown above.

PROC bank

SEQ
-- This process has access to the shared end of the record.
-- This process must first claim the record before it can use
-- the shared end.

CLAIM close
  INT acc_no:
  INT answer:
SEQ
  close[accountno] ? acc_no
  .. close the account
  close[result] ? answer

PROC Account

SEQ
  -- This process has access to the shared end of the record.
  -- This process must first claim the record before it can use
  -- the shared end.

CLAIM close
  INT acc_no:
  INT answer:
  SEQ
    close[accountno] ? acc_no
    .. close the account
    close[result] ? answer

Objects which want access to the broadcasted message, have to first claim the shared end. The object bank and account claim the shared end and then read the message broadcasted. As shown above the object Account and Bank claim the shared end of channel close by the code CLAIM close. Now both the objects Account and Bank can receive messages on the shared end of communication close. It should be noted that shared end of communication gives no performance advantage.
9.1 Introduction

In this chapter an attempt has been made to assess and validate the results that have been obtained in the design, development and implementation of the banking system application; conversion of the application to BSP Occam [Allwright 91] and Occam 3 [Barrett 91]; and to conclude that an effective parallel programming environment has been achieved.

9.2 Effective Programming Environment

Effective and efficient programming environment using VCR emulating Occam 3 language and Inmos T9000 transputer was fully realised and achieved by designing, developing, implementing and testing of the banking system application developed. An unlimited number of virtual communication channels between processors were obtained with the use of VCR. It eliminated the requirement to design and develop routers and multiplexers normally required to augment communication channels in a network of processors and allowed objects to communicate with each other through unlimited number of message channels. This can be termed as a significant advantage as it considerably reduces the programming load in a parallel computing environment.

The software design was developed on Virtual Processors and not on actual processors of the transputer hardware. It enabled programming to proceed without regard to the actual configuration and specification of the transputers. The software developed was
successfully mapped onto networks of one, two and four T800 Inmos transputers. It has demonstrated and proved that software codes developed on Virtual Processors can be ported and mapped to any number and configuration of transputers. This is another advantage as it facilitates parallel and massively parallel programming as any number of Virtual Processors can be specified during the software development.

9.3 Reusability and extendability

In the design and development of the application reusablility and extendability is achieved by replicating and using the objects. It enabled and provided the flexibility to add, delete and reuse new or old objects as required to meet user requirements.

Developed and tested objects can be stored in libraries and reused later on resulting in economies of effort and costs. In addition, the design can be readily modified or extended. In the application developed, the extendability and reusability were proved and achieved by adding additional customer and ATM objects to the program.

9.4 Timing Results

Each parallel threads refer to a transaction. Timing is recorded for each transaction from the start to the end of the transaction. The unit of timing here is in milliseconds. The timing clock starts in the customer object where the transaction begins and stops also in the customer object where the transaction ends. This is achieved by placing a clock in the object customer.

It can be seen from the code below that the clock starts when the customer starts the transaction by sending a message saying "I want to withdraw this amount from this account and this is my pin number" and stops when it gets the a message from the ATM saying " Your transaction is successful and this is your amount and this is your
balance". The objective here was to measure the time fro each transaction.

PROC Customer (....)

..

TIMER time.out:

SEQ

...

time.out ? time1
cust.to.atm ! accont_no; password; amount
atm.to.cust ? balance; amount
time.out ? time1

....

As the number of transactions increase so does the parallel threads in the design. But it can be seen from the timing above that increase in number of transputers has no effect on the timing. This is because object bank is a serial component in the design which forms the bottleneck as given in section 7.5 of chapter 7.

Timing results are an indicator of the efficiency and time response of the application developed. Timing results measured on running the program on one, two and four transputers were as follows:

<table>
<thead>
<tr>
<th>Parallel Threads</th>
<th>One Transputer</th>
<th>Two Transputers</th>
<th>Four Transputers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>26</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>61</td>
<td>63</td>
</tr>
<tr>
<td>3</td>
<td>76</td>
<td>89</td>
<td>94</td>
</tr>
<tr>
<td>4</td>
<td>101</td>
<td>106</td>
<td>110</td>
</tr>
<tr>
<td>5</td>
<td>126</td>
<td>127</td>
<td>127</td>
</tr>
<tr>
<td>6</td>
<td>154</td>
<td>153</td>
<td>152</td>
</tr>
<tr>
<td>7</td>
<td>176</td>
<td>174</td>
<td>172</td>
</tr>
</tbody>
</table>
The above timing results indicate the following:

As processing load increases with increasing number of threads, the execution time on four transputer and one transputer is nearly the same. This is due to the serial component in the design and is the ultimate limiting factor in the performance of the system.

9.5 Future MPP will give the illusion of shared memory

When data is required by a process from another process, it is obtained by message passing. Because of the complexities of message passing, software has been developed for distributed memory architecture which imitates a shared virtual global memory. This replaces the concept of point to point message passing with broadcast messaging in which data broadcasted by a process is received by all the processes in the network.

Distributed memories in parallel computers can thus be managed as a single large virtual shared memory. In it when a processor tries to access data, it first looks for it in its own memory and if it is not there, a high speed search is launched to find its address in the memories of other processors. The address of the data is same throughout the machine, regardless in which processor memory the data happens to reside.

The above concept of broadcast messaging and illusion of shared virtual memory have been theoretically demonstrated with BSP Occam and then with Occam 3 in chapter 8.

Hardware and software which can imitate a shared global virtual memory model have been developed. These machines have memories which are physically distributed but the software provides the capability of a shared memory thereby eliminating the complexities of message passing in distributed memory architectures.
9.6 Achievement of Objectives of the Thesis

The following objectives were set for this thesis and their achievements is summarised below:

- To develop a methodology for concurrent object oriented design. This has been successfully achieved as described in chapters 5 and 6.

- To map object oriented concepts on the distributed memory architecture and exploit the virtual channel router to provide unlimited number of channels between objects. This has been achieved as described in chapter 7.

- To map the design developed onto BSP Occam and then Occam 3 using the broadcast method of communication. This has been achieved as given in the chapter 8.

- To implement, validate and demonstrate the above concepts by designing and developing an application program in a parallel programming environment. A banking system application has been developed, tested and demonstrated to prove the concepts.

9.7 Future Directions of Research

The above research could be carried further with Occam 3. The features of Occam 3 like shared channel of communication implementing broadcast messaging and modules which act like objects needs further study and research.

The efficacy and effectiveness of the shared virtual memory model on transputer network could also be researched along with making Occam 3 and its later versions
9.8 Conclusion

An attempt has been made in this project to investigate the extend to which OO concepts can be supported in the transputer - occam environment of parallel computing. The thesis has confirmed that OO concepts and parallel systems are compatible and can be integrated to provide synergies though there are currently some constraints like:

- efficiencies of execution are somewhat compromised due to inheritance and communication in OO model,

- Occam process is static and determined at compile time whereas OO languages are dynamic and instantiation occurs at run time.

Notwithstanding above, the thesis I believe has validated that OO concepts implementation on transputers - occam can provide an efficient and effective parallel programming environment. It has also satisfied Occam's rule: "Pluralitas non est poenda sine necessitate" which translates into: "Don't create complexity with out necessity".

If there is one conclusion that can be drawn from this thesis, it is that the Occam-Transputer-OO combination has greatly contributed to the launch of a new era in parallel systems computing. Transputer - Occam combination along with its competitors (like Intel i860, Alpha, Sun Sparc, Fujitsu’s VPP 500 processors, etc) are being used to build powerful machines of teraflops speeds which have the promise of solving some of the complex and massive problems of science and technology and may hopefully make our world a better place to live.
REFERENCES


[Corradi et al 92] Corradi, L Lconardi, D Vigo. “Massively parallel programming
environments. How to map parallel objects on transputers”.


[Wirfs-Brock 90] Rebecca Wirfs-Brock, Wilkerson, Weiner. Designing Object -
Oriented Software. Prentice Hall, 1990.


APPENDIX A: Transputer T9000

A.1 Introduction

The Inmos T9000 is the latest member of the transputer family. It provides a higher performance and improved communication than its predecessors. It has been built using the advanced CMOS technology to integrate a 32 bit processor, 64 bit FPU, 16 Kbytes of cache memory, a communication processor and four high bandwidth communication links on a single chip.

The T9000 CPU contains a 32 bit arithmetic and logic unit and a 64 bit floating point unit (FPU). The FPU operates on 32 and 64 bit floating point numbers as specified by the IEEE 754 standard. The CPU also includes instructions for byte and half word operations. It uses 32 bit linear addressing and can address up to 4Gbytes of memory. It is also compatible with previous transputers. It can be used for real time embedded applications. Its block diagram is shown below.
A.2 Performance

- Single processor performance: The CPU is capable of a peak performance of 200 MIPS and 25 MFLOPS.

- Real-time performance: It offers sub-microsecond interrupt response and context switch times.

- Communication performance: T9000 communication links provides a total of 80 Mbytes/second bidirectional bandwidth.


A.3 Architecture

The CPU of transputer T9000 contains three registers (Areg, Breg, Creg) used for expression evaluation and forming a hardware stack. The transputer has three other registers used when executing code. These are:

- The instruction pointer which points to the next instruction to be executed.

- The workspace pointer which points to an area of store where local variables are kept.

- The operand register which is used in the formation of instruction operand.

A.3.1 Pipelined

To increase the execution rate of the transputer instruction set, IMS T9000 is able to
issue several instructions per cycle. The instructions are executed in five stage pipeline; the first can fetch two local variables; the second performs two address calculations for accessing non-local or subscripted variables; the third stage can load two non local variables; the next stage can perform an ALU or FPU operation; and the final stage can do a conditional jump or write.

A.4 Multiprocessing

The hardware scheduler of T9000 supports creating and scheduling of any number of concurrent processes.

A.4.1 Support for concurrent processes

The T9000 transputer includes a hardware kernel with the ability to execute many software processes at the same time to create new processes rapidly, and to perform communication between processes within a transputer and between processes on different transputers.

A.5 Communications support device

T9000 has some communication support devices which extend the communication capabilities of T9000. The IMS C1xx family communication support chips can be used to build any size of IMS T9000 system.

IMS C104 packet routing switch is a complete routing switch on a single chip. It connects 32 links to each other via a 32 by 32 way non blocking crossbar switch with sub microsecond latency. The C104 routing switch can be used to build larger and more complex networks by linking any number of IMS T9000 transputers or any other devices that use the link protocol.
A.5.1 Communications systems

To support interprocessor communications, there is a complete communications subsystem on the chip. This includes four 100 Mbits/ full duplex, serial communication links each with its own pair of direct memory access (DMA) channels. The links can be directly connected between transputers with no external buffering or other glue logic.

A.5.2 T9000 Links

To solve the problem of unlimited number of links in the transputer a multiplexing hardware was added to allow any number of processes to use each link. These channels which share the links were known as virtual channels and they have the same behaviour as the software channels.

The message sent across the link is divided into packets. The packet has an header to identify the destination process. This makes the transputer easy to use as it separates the software configuration from the hardware. It also makes better use of the hardware as it keeps the links busy with messages from different channels. The VCP (Virtual Channel Processor) routes messages to and from processes on the IMS T9000. It shares each physical link between any number of processes. It provides virtual channels between any two transputers.
A.5.2 Message routing

The IMS C104 is a communication support device. In order to minimise latency, the switch uses wormhole routing. As the header is read and the connection through the crossbar is set up, the header and the body of the packet are then transmitted from the output link. The path through the switch disappears after the packet token has passed through. Thus a single virtual channel cannot hog a route through a network.

A.6 Hierarchical memory system

IMS T9000 has a complete, hierarchical memory system providing fast and efficient access to data and instructions. IMS T9000 includes a 16 Kbyte unified cache to provide single cycle access to instructions and data. The cache provides a peak bandwidth of 200 Mwords/sec. It also has another cache for the most frequently used local variables of a program which provides another 150 Mwords/sec of memory bandwidth.
A.6.1 Main cache

The main cache consists of four independent banks, each containing 256 lines. Each line holds data from four consecutive words (16 bytes) in memory. The four cache banks are accessed by a number of different functional units in the IMS T9000, some of these units have multiple ports into the cache. To allow four simultaneous reads and writes to take place in each cycle, there are four sets of address and data buses.

A.6.2 Workspace cache

The workspace cache can hold a copy of the first 32 words of procedure stack and workspace. It is triple ported, allowing two reads and a write in every cycle. It is used for accessing local variables quickly. These variables are read in the first stage of the pipeline and can then be used for non local address calculations in the next stage. It is a part of the processor pipeline, in some ways it is equivalent to the general purpose register found on the other microprocessors providing fast excess to frequently used data.
APPENDIX B: Object Classes, protocols, configuration descriptions and testing

B.1 CLASSES

Some of the classes defined in this OO system are shown below:

a) class customer

Include the protocol files

PROC customer( CHAN OF SP fs,ts,
CHAN OF CUSTATM cust.to.atm,
CHAN OF ATMCUST atm.to.cust,
CHAN OF BOOL stopper )

#USE "vhostio.lib"
Define the data types for the variables

SEQ
cust.to.atm ! noac;accno;password;acc;actn;amt
atm.to.cust ? CASE
   balanceac; fatmbalance
   SEQ
      Transaction successful
   amtwrong;amtvld
   SEQ
      Transaction unsuccessful because the amount exceeds the limit
   pworderror;peiror
   SEQ
      Transaction unsuccessful because the pin number entered is wrong

Deal similarly with other queries

so.exit(fs,ts,sps.success )
stopper ! TRUE

b) class bank

PROC banksys([] CHAN OF BSYSATM bsys.to.atm,
[ ] CHAN OF ATMBSYS atm.to.bsys,
[ ] CHAN OF BSYAC bank.to.account,
[ ] CHAN OF ACBSYS account.to.bank)

--{{ check password
PROC check(INT password, INT accno, INT validity)
check if the pin number entered is valid

:
Include library files
Declare the variables and their types

SEQ
SEQ i=0 FOR SIZE atm.to.bsys
SEQ
atm.to.bsys[i] ? CASE
noacc; accno; password; acc; act; amt
SEQ
check if the pin number is valid
IF
(validity < 0)
SEQ
If pin number not valid
stop the transaction
bsys.to.atm[i] ! pworderrorba;perror
(validity > 0)
SEQ
If pin number is valid
continue with the transaction
bank.to.account[i] ! badata; accno; amt; acc; act
account.to.bank[i] ? CASE
balanceacb; bal
SEQ
bsys.to.atm[i] ! balanceba; bal


c) class account

Include the protocol files

PROC account([] CHAN OF BSYAC bank.to.account,
[] CHAN OF ACBSYS account.to.bank,
[] CHAN OF ACACC account.to.check,
[] CHAN OF CACCA check.to.account,
[] CHAN OF ASACC account.to.saving,
[] CHAN OF SACCA saving.to.account)

PROC withdraw(INT bal, INT amount, INT fbal)
:

PROC deposit(INT bal, INT amount, INT fbal)
:

Include the library files
Declare the variables

SEQ
SEQ i=0 FOR SIZE bank.to.account
SEQ
bank.to.account[i] ? CASE
badata; accno; amount; account; act
SEQ
IF
eqstr(account,"savingsacc")
SEQ
send message to savings account
eqstr(account,"checkacc")
SEQ
send message to check account
account.to.check[k]! dataac; accno; account; amount; act

B.2 PROTOCOLS

Protocol for class account to class bank
PROTOCOL ACBSYS
CASE
balanceacb; INT
amountacb; INT

Protocol for class ATM to class bank
PROTOCOL ATMBSYS
CASE
noacc; INT; INT; [10] BYTE; [10] BYTE; INT

Protocol for class bank to ATM
PROTOCOL BSYSATM
CASE
vld; INT
actn; [10] BYTE
amnt; INT
balanceba; INT
pworderrorba; INT

Protocol from class ATM to class customer
PROTOCOL ATMCUST
CASE
balanceac; INT
amtwrong; INT
pworderror; INT

Protocol from class customer to class ATM
PROTOCOL CUSTATM
CASE
noac; INT; INT; [10] BYTE; [10] BYTE; INT

Protocol from class savings account to class account
PROTOCOL SACCA
CASE
  balancesaa; INT
:

Protocol from class bank to class account
PROTOCOL BSYAC
CASE
  badata;INT;INT;[10]BYTE;[10]BYTE
:

Protocol from class account to class savings account
PROTOCOL ASACC
CASE
  dataas;INT;[10]BYTE;INT;[10]BYTE
:

Protocol from class account to class check account
PROTOCOL ACACC
CASE
  dataac;INT;[10]BYTE;INT;[10]BYTE
:

Protocol from class check account to class account
PROTOCOL CACCA
CASE
  balanceca; INT
:

B.3 Configuration

Include the protocol file

Define the constants

Define the channels

PLACED PAR
PROCESSOR 0 TA
#USE "hosthook.cah"
#USE "ctomer1.cah"
#USE "ctomer2.cah"
[2] CHAN OF SP fs,ts :
[2] CHAN OF BOOL stopper :
PAR
  customer1( fs[0],ts[0], ca[0], ac[0],stopper[0] )
  customer2( fs[1],ts[1], ca[1], ac[1],stopper[1] )
  
  customer8( fs[7],ts[7], ca[7], ac[7],stopper[7] )
hosthook( fs[0], ts[0], stopper[0])
hosthook( fs[1], ts[1], stopper[1])

PLACED PAR i = 0 FOR num.nodes-1
PROCESSOR (i+1) TA
#USE "atml.cah"
B.4 Network Configuration Files

The network configuration file (ncf) generated by the VCR for one, two and four transputer network are:

a) One Transputer

```
1 0
0
T800c-20
4100K
h x x x
0 2
4 0
```

b) Two Transputer

```
2 1
0
T800c-20
4100K
h x 01:1 x
1
T800c-20
1028K
```
c) Four Transputer

4 2

0
T800c-20
1028K
h o1:0 o2:0 o3:3

1
T800c-20
1028K
i0:1 n3:0 x x

2
T800c-20
1028K
i0:2 n3:1 x x

3
T800c-20
1028K
n1:1 n2:1 x i0:3

0 8
4 0 1 0 2 0 3 0

1 8
0 0 4 0 0 1 1 0

2 8
0 0 0 1 4 0 1 0

3 8
3 0 0 0 1 0 4 0

x i0:2 x x
0 4
4 0 2 0
1 4
1 0 4 0
B.4.1 Mapping of virtual processors to the physical processors

The virtual processors are mapped to the physical processors according to the following defaults:

- Virtual processor 0 maps to the physical processor 0.
- Virtual processors with ids greater than zero wrap around the physical processors with ids greater than zero in modulo fashion [Debbage et al 91d].

a) Two Transputer

b) Four Transputer
B.5 Testing

a)
Booting root transputer...ok

vcr : Virtual Channel Router, Version 2.0k, 11:03:30 Oct 16 1992 M. Debbage, M. Hill, University Of Southampton, ESPRIT PUMA P2701

No devices for marks
Loading virtual processor 0
Loading virtual processor 1
Loading virtual processor 2
Loading virtual processor 3
Loading virtual processor 4
Loading virtual processor 5
Loading virtual processor 6
Network loaded successfully

This is the customer1
NAME Mr A Smith
Address: 56 Market st,
SYDNEY

Amount: 100
Balance: 357

This is the customer2
NAME Mr J Collins
Address: 12 Union st,
SYDNEY
b)
Booting root transputer...ok
vcr : Virtual Channel Router, Version 2.0k, 11:03:30 Oct 16 1992 M. Debbage, M. Hill, University Of Southampton, ESPRIT PUMA P2701

No devices for marks
Loading virtual processor 0
Loading virtual processor 1
Loading virtual processor 2
Loading virtual processor 3
Loading virtual processor 4
Loading virtual processor 5
Loading virtual processor 6
Loading virtual processor 7
Network loaded successfully

This is the customer1
NAME Mr A Smith
Address: 56 Market st, SYDNEY

Amount: 100
Balance: 357

This is the customer2
NAME Mr J Collins
Address: 12 Union St, SYDNEY

Amount: 200
Balance: 799

This is the customer3
NAME Mr Andrew
Address: Charles St, SYDNEY

Balance: 500
C.1 VCR 2.0

VCR (Virtual Channel Router) is a software package which provides unrestricted channel communication across network of T-series transputers. The user code is written in pure Occam. A configuration file places these code units onto virtual processors. An off-line tool extracts the topology information from the network definition file and then constructs the routing tables and outputs a textual network configuration file (ncf) which contains the processor type, memory size, booth path, connectivity and routing tables. The network configuration file is used to map the virtual processors at load time.

The core of this packet router is that it delivers asynchronous datagrams around arbitrary networks. This forms the basis of version 2.0 of the virtual channel router (VCR) package. The use of this package is to allow the programmer to construct applications where machine details and limitations are hidden, increasing portability and code reusability.

VCR allows occam program writers to ignore the usual per processor valency restrictions by providing a fully integrated router capable of simulating direct channel communications between all processors in the network. This work has been supported by the E.C ESPIRIT P2701 PUMA project. A diagram of a general purpose environment is shown below.
C.1.2 Performance

The performance of a routing system is crucial to its acceptance as programming environment. The memory requirement is less than 100K per network node. It achieves raw data rates with a packet switch time of around 60μsec.

C.1.3 Generating a VCR topology

The VCR topology is described by a network configuration file (.ncf). Tools are used to generate these file from the check utility.

```
check | mtest | routegen >mach1.ncf
```

The network configuration file describes the transputers in the topology. Once the
network configuration file has been generated, it can be checked for deadlock freedom by the command

ncfcheck <mach1.ncf

C.2 BSP Occam Library Procedures

BSP Occam library procedures are given in the following sections.

C.2.1 Process Synchronisation

PROC gnew.sync(INT sync.handle, VAL INT n)
Declares a sync object for synchronising n user processes and returns an integer, sync.handle, by which the sync object may be subsequently accessed. This procedure is called only once and the handle is then distributed to other processes using gwrite.parameters.

PROC gsync(VAL INT sync.handle)
Synchronises with the other processes.

C.2.2 Global variable Manipulation

PROC gnew.int(INT handle, VAL []INT limits)
A global array of INTS is of size given by limits and returns a handle by which the array can be accessed.

PROC gread.int(VAL INT handle, VAL []INT subscripts, INT value)
An element of the global array is read and it is accessed through handle and indexed by subscripts.

PROC gwrite.int(VAL INT handle, VAL []INT subscripts, VAL INT value)
Writes the value into the global INT array accessed through handle at a position given by subscripts.

PROC gnew.type(INT handle, VAL []INT limits)
PROC gread.type(VAL INT handle, VAL []INT subscripts, TYPE value)
PROC gwite.type(VAL INT handle, VAL []INT subscripts, VAL TYPE value)
In the above procedures type can be byte, int16, int32, real32 etc. The behaviour of these procedures is similar to the behaviour of procedures for int.

PROC gnew.special(INT handle, VAL INT size, VAL []INT limits)
The above procedure gnew.special declares a global area of data type whose unit is a byte array.

PROC gread.special(VAL INT handle, VAL [] INT subscripts, VAL [] BYTE value)
The above procedure reads in data of value type [] BYTE from the location given by subscript and the global array specified by the handle.

PROC gwrite.special(VAL INT handle, VAL [] INT subscripts, VAL [] BYTE value)
The above procedure writes in data of value type [] BYTE from the location given by subscript and the global array specified by the handle.

PROC gdispose(INT handle)
The above procedure invalidates a handle.

C.2.3 Global Parameter Distribution
PROC gwrite.parameters(VAL INT system.handle, VAL [] INT parameter.list)
This procedure broadcasts the array of INTS to all the other user processes

PROC gread.parameters(VAL INT system.handle, VAL [] INT parameter.list)
receives the integer array parameter.list

PROC gccall(VAL []BYTE SName, VAL [] INT parameter.list, INT target)
Dynamically loads and runs the code given by SName onto the processor target,
passing it the array []INT parameter.list.
Occam 3 has all the features of its predecessor Occam 2. Besides that some new features have been added introduced which are listed below:

D.1 Records of channels

Some times the communication between processes is achieved over a number of channels in both directions. This group of channels can be defined as a record of channels.

CHAN TYPE GOC
   RECORD
      CHAN OF REAL16 variable, result:

This declaration introduces a channel of type GOC, which is a record of two channels one named variable and the other named result.

The record of this type is declared as shown below

GOC chan

The record chan may be used as shown below:

A call channel is declared
PAR
   REAL16 x,y:
SEQ
   chan[variable] ? x
   chan[result] ! x*x
SEQ
   chan[variable] ! 36.6
   chan[result] ? y

D.2 Remote call channels

Remote call channels provide the ability to pass parameters from one process to a
procedure which is executed by another process. A call channel is declared with its name and formal parameter list. For example

CALL square(RESULT INT result, VAL INT x):

The call is then made. For example

square(sq, 8)

These parameters are passed to the procedure body which has the accept process

ACCEPT square(RESULT INT result, VAL INT x)
SEQ
  calls := calls+1
  result := x*x

On acceptance of the call the process increments the count of the number of the calls.

D.3 Sharing

Shared channels provide connections between a single process with an arbitrary number of processes.

D.3.1 Sharing call channels

Shared call channel is declared similar to the ordinary call channel accept that the declaration is preceded by the key word shared.

SHARED CALL square(RESULT INT result, VAL INT x):

This call channel may be shared between many processors.
D.3.2 Shared communication channels

Communication channels are not shared singly but as a record. A typical record is declared as follows:

\begin{verbatim}
CHAN TYPE GOC
  RECORD
    CHAN OF REAL16 variable, result:
\end{verbatim}

This declaration introduces a channel of type GOC, which is a record of two channels one named variable and the other named result.

The shared record of channel is declared as shown below

\begin{verbatim}
SHARED GOC chan
\end{verbatim}

The record chan may be used as shown below:

\begin{verbatim}
A process which wishes to use the shared end of a channel must first claim it.
CLAIM chan
  REAL16 x,y:
  SEQ
    chan[variable] ? x
    chan[result] ! x*x
\end{verbatim}

The process which has access to the non shared end of the record must first grant the record to first claim the process.

\begin{verbatim}
GRANT chan
  REAL16 x,y:
  SEQ
    chan[variable] ! 36.6
    chan[result] ? y
\end{verbatim}

D.3 Modules

Modules provide a mechanism of structuring processes. A module is like a black box
with a number of channels which can be used to communicate with the contents of the box. Inside the box there are processes which service the channels. New modules can be plugged to the existing application without requiring any changes.

For example

```
MODULE TYPE TWO.BUFFER ()
  CHAN OF INT in, out :
  RESOURCE
    ... buffer process
  :
  :
```

Instances of the module type are declared as follows:

```
MODULE buffer IS INSTANCE TWO.BUFFER () :
```

These Module defined can be stored in the library and later reused.

**D.4 User Defined types**

The user can define types as follows:

```
DATA TYPE BOX IS REAL32:
DATA TYPE LENGTH IS REAL32:
BOX box1, box2:
LENGTH leng1, leng2:
```

Records can also be defined

```
DATA TYPE CAR
  RECORD
    INT cylinder
    INT power
    REAL16 speed
    INT mileage
```