

LOAD BALANCING IN A DISTRIBUTED
ENVIRONMENT :
IMPLEMENTATION ON ETHERNET AND ATM

by

CHAN HUAH YONG

Thesis submitted in fulfilment of the
requirements for the degree
of Master of Science

November 1996

MIKROPIS
4557

Acknowledgements

I have been extremely fortunate in having the help of many people during the course of this project. Without their guidance, help and comments, the preparation of this report would have been impossible.

First of all, I would like to express my deepest gratitude to my field supervisor Prof. Hervé Guyennet, professor at the University of Franche-Comté, for his supervision, long discussions and helpful suggestions. I would also like to thank Prof. Zaharin bin Yusoff, for his encouragement and thoughtful criticisms and Prof. Michel Trehel for his arrangements and help. My heartfelt gratitude to the Computer Science Laboratory of Besançon, University of Franche-Comté for allowing me to use the facilities and equipment. This work is carried out there during one year. I also received helpful comments from Claude Daval-Frerot, Francois Spies, Jean Christophe Lapayre, Pascal Chatonnay, Chantal Balayer, Philippe Frossard and all the members of the Computer Science Laboratory of Besançon and 'Unit Terjemahan Melalui Komputer' at Universiti Sains Malaysia.

Lastly, my deepest appreciation to my wife, Gek Sim for her continuing patience, love and support.

Table of Contents

Abstract in Bahasa Malaysia.....	viii
Abstract	x
Part One Fundamentals	
Chapter 1 Introduction.....	1
1.1 Scope and Objectives of Thesis	1
1.2 Introduction to Load Balancing	2
1.3 Justifications for ATM	3
1.4 Justifications for MPI	4
1.5 Chapter by Chapter Summary.....	4
Chapter 2 Distributed And Parallel Systems	6
2.1 Definition	6
2.2 Characteristics.....	6
2.3 Advantages	7
2.4 Parallel Systems.....	8
2.5 Classification of Hardware	8
2.6 Concepts in Distributed And Parallel Software.....	12
2.6.1 Programming Languages.....	12
2.6.2 Types of Parallel Computing.....	13
2.6.3 Operating Systems	13
2.6.4 Interprocess Communication (SYSTEM V IPC)	14
2.6.5 Sockets	16
2.6.6 Remote Procedure Calls	18
2.6.7 Light Weight Processes - Threads.....	19
2.6.8 CHORUS	22
2.6.9 Glossary of Object Technology Terms	24

Chapter 3 ATM Networks.....	25
3.1 Evolution of Networks.....	25
3.2 Asynchronous Transfer Mode.....	26
3.3 Characteristics.....	27
3.4 ATM Layers.....	27
3.5 The ATM Physical Layer	28
3.6 The ATM Layer	29
3.6.1 Rationale For The 53-Byte Cell Size.....	32
3.7 The ATM Adaptation Layer (AAL)	33
3.8 Advantages of ATM.....	35
3.9 ATM Quality of Service.....	36
3.10 Shortcomings of ATM.....	36
Chapter 4 Load Balancing	38
4.1 Static Scheduling.....	40
4.1.1 Scheduling Tools.....	41
4.2 Dynamic Scheduling.....	41
4.2.1 Distributed Versus Centralized.....	42
4.2.2 Organization.....	43
4.2.3 Source (Sender)- Or Server (Receiver)-Initiative Strategies.....	43
4.3 Process Migration.....	44
4.3.1 Process Migration in DEMOS/MP	45
4.4 Components of Load Balancing Algorithms.....	48
4.5 Characteristics of Load Balancing Algorithms	49
Chapter 5 Message Passing Interface (MPI)	50
5.1 Messages and Processes	50
5.2 MPI Datatypes.....	51
5.3 Communication	51
5.4 Performance Analysis	54

5.5 Current Implementations.....	55
5.6 Present and Future.....	55
5.7 Message Passing Versus Distributed Shared Memory	55
Part Two Experimentation	
Chapter 6 Comparisons of Network Throughput	57
6.1 Performance Comparison Between Sockets And MPI.....	57
6.2 Performance Comparison Between IP/Ethernet, IP/ATM And AAL5/ATM.....	58
Chapter 7 Implementation of The Bidding Strategy	62
7.1 Components of the Bidding Strategy.....	62
7.2 Simulation of Processor Nodes	63
7.3 Execution Guide for The Bidding Programs	64
7.4 Results of The Bidding Algorithm.....	65
Chapter 8 Implementation of The Vector Strategy.....	70
8.1 Vector (Barak And Shiloh's) Algorithm	70
8.2 Execution Guide for The Vector Programs	71
8.3 Implementation And Results.....	71
Chapter 9 Conclusion	74
9.1 Concluding Remarks.....	74
9.2 Some Suggestions For Immediate Improvement.....	77
9.3 Future Work.....	78
Appendix	
BIBLIOGRAPHY.....	115
Index	118

Lists of Tables and Figures

Figure 1.1 Scope of experiments on load balancing in a distributed environment.	2
Table 2.1 Computer system classification of Flynn.	8
Figure 2.1 A taxonomy of parallel and distributed computer systems.	9
Figure 2.2 Multiprocessor computer organization with common bus.	10
Figure 2.3 A multicomputer consisting of workstations on a LAN.	10
Figure 2.4 (a) A crossbar switch. (b) An omega switching network.	10
Figure 2.5 (a) Grid architecture. (b) Hypercube architecture.	11
Figure 2.6 Classification of parallel machines	11
Figure 2.7 Data structures for messages.	14
Figure 2.8 Data structures for shared memory.	15
Figure 2.9 Data structures for semaphores.	15
Figure 2.10 Sockets Model.	16
Table 2.2 Some common socket commands using TCP/IP protocol.	17
Figure 2.11 The steps involved in the RPC from client to server.	19
Figure 2.12 Single-thread and multi-thread processes.	20
Figure 2.13 Per thread and per process items.	20
Figure 2.14 Three models for the use of threads in a given process.	21
Figure 2.15 Process communication in Chorus.	23
Figure 2.16 Architecture of Chorus.	23

Table 3.1 Switching technologies.	25
Figure 3.1 Types of relay systems.	26
Figure 3.2 The ATM Layers.	27
Figure 3.3 The ATM cell.	30
Figure 3.4 Multiplexing of VP/VC connections onto a physical medium.....	31
Figure 3.5 Segmentation/reassembly between source and network.	33
Table 3.2 ATM classes of services	34
Table 3.3 B-ISDN layer functions.....	35
Figure 3.6 The different formats of IP headers.	37
Figure 4.1 Scheduling system [CASA88].....	38
Figure 4.2 Characteristics of task scheduling [CASA88].....	39
Figure 4.3 A logical hierarchy of processor management nodes.....	42
Figure 4.4 Steps in a process transfer [POWE83].....	47
Figure 4.5 Components of a load balancing algorithm and their interactions.	49
Figure 5.1 Schematic representation of collective data movement in MPI.....	53
Figure 5.2 Illustration of reduce and scan.	54
Figure 6.1 Protocol hierarchy modified from [LIN94].	57
Figure 6.2 Test results of socket/ATM, socket/Ethernet and AAL5/ATM.	61
Figure 6.3 Two test machines connected with Ethernet and ATM.....	61
Figure 7.1 Graph of the results of the bidding algorithm.....	69
Table 8.1 Results of the vector algorithm running on Ethernet, ATM and MPI.....	73
Figure 8.1 Graph of the results of the vector algorithm.....	73
Table 9.1 Main differences between ATM and Ethernet.	75
Table 9.2 Main differences of two load balancing strategies on ATM and Ethernet.	76
Figure 9.1 Four workstations connected with ATM in the LIB., University of Franche-Comté.	79

Pengimbangan dalam persekitaran teragih : pelaksanaan pada Ethernet dan ATM

Abstrak

Laporan ini menerangkan pelaksanaan dua algoritma pengimbangan beban, iaitu *bidding* [GOSC91, STAN84] dan *vektor* [GOSC91, BARA85]. Kelakuan dua algoritma ini diuji untuk melihat kesan kepada kelajuan pecutan pada platform yang berlainan dan dengan menggunakan *socket* dan bahasa pengaturcaraan selari *MPI* [LAM, GROP94, MALA96]. Bagi dua algoritma yang menggunakan *socket*, keputusan yang diperolehi dibanding di antara pelaksanaan pada Ethernet dan pada ATM [ATMF93, BLAC95], manakala algoritma yang menggunakan *MPI* hanya dilaksanakan pada Ethernet.

Pengimbangan beban merupakan satu teknik untuk mengagihkan proses kepada pemproses dalam komputer selari atau komputer berangkai untuk mengimbangkan beban kerja di antaranya. Penempatan proses dan migrasi proses digunakan secara dinamik untuk melaksanakan penjadualan proses. Tujuan utama adalah untuk memaksimumkan "throughput" sistem melalui penggunaan sumber secara optimum. ATM ("Asynchronous Transfer Mode") amat menarik dalam bidang pemprosesan selari disebabkan "throughput"nya yang tinggi serta ketepatannya. Terdapat pelbagai hasil yang menggalakkan dalam penyelidikan terhadap penggunaan ATM dalam pemprosesan selari dan teragih [LIN94, HUAN94].

MPI (Message Passing Interface) merupakan suatu bahasa pengaturcaraan selari yang piawai, mudah alih dan tertakrif secara formal. Ia menggunakan model penghantaran mesej dengan kawalan keselarian secara eksplisit.

Strategi *bidding*, algoritma pengimbangan beban yang teragih dan dinamik ini dipilih sebab ia melibatkan komunikasi mesej yang banyak tetapi pendek, dan ATM digunakan untuk meneliti sama ada penggunaannya boleh mempercepatkan komunikasi ini. Konsep *bidding* merupakan satu proses tawar-menawar melalui "bid" yang dihantar di antara pelbagai komponen sistem pengimbangan beban. *Bidding* ialah satu proses tempatan di mana perbandingan "bid" dibuat tanpa kawalan pusat.

Di sebaliknya, strategi *vektor* atau strategi Barak dan Shiloh merupakan algoritma pengimbangan beban yang teragih dan dinamik. Pertukaran vektor beban dilaksanakan melalui nod yang dipilih secara rawak. Ia tidak sama dengan teknik penyiaran ("broadcast") yang digunakan oleh *bidding*. Mesej (vektor beban) yang digunakan adalah lebih panjang berbanding mesej *bidding*.

Dalam ujikaji kami, program rangkaian "benchmark" disediakan untuk menguji "throughput" *socket* dan MPI sebelum algoritma pengimbangan beban dilaksanakan. Seterusnya, pengujian telah diadakan IP/ATM, protokol AAL5/ATM dan keputusannya dibandingkan dengan IP/Ethernet, dan khususnya analisis peratusan penggunaan "bandwidth" dijalankan.

Pada amnya, tiga modul ditakrifkan untuk kedua-dua algoritma pengimbangan beban tersebut.

1. Modul mencari nod yang terbaik untuk sesuatu proses. Program ini dipanggil apabila terdapat proses baru.
2. Modul menilai beban kerja tempatan dan menghantar maklumat ini kembali kepada permintaan modul pertama.
3. Modul yang bertanggungjawab terhadap pelaksanaan proses secara tempatan atau terpencil.

Kesimpulan yang diperolehi mengesyorkan agar kad driver ATM perlu dipercepatkan bagi paket data yang kecil meskipun memang terdapat kesukaran bagi aplikasi menggunakan keseluruhan "bandwidth" yang diagihkan. Dengan penggunaan bahasa MPI, algoritma pengimbangan beban mengalami overhead lapisan. Untuk mengurangkan overhead ini, cara yang paling berkesan adalah dengan menempatkannya dalam sistem operasi. Berdasarkan keputusan yang diperolehi, kesan pada ATM adalah lebih ketara daripada kesan pada Ethernet bagi strategi vektor berbanding strategi *bidding*. Analisis terhadap keputusan dan fenomena ini juga dilaksanakan. Untuk kajian lanjutan, algoritma yang lain akan diuji dan juga pada rangkaian komputer yang lebih besar.

Abstract

This report describes and explains the implementation of two load balancing algorithms, namely *bidding* [GOSC91, STAN84] and *vector* [GOSC91, BARA85]. The difference in speed on different platforms is studied in relation to the behavior of such algorithms when using sockets and the parallel computing language MPI [LAM, GROP94, MALA96]. For two algorithms using sockets we compare their results on Ethernet and ATM[ATMF93, BLAC95], while for two algorithms with MPI we just test them on Ethernet.

Load balancing is a technique for distributing processes amongst processors within a parallel computer or amongst workstations connected by a network in an effort to equalize workload. Job placement and migration are used to dynamically schedule processes. The objective is to maximize system throughput by optimizing the utilization of resources.

ATM (Asynchronous Transfer Mode) is of very high interest in the parallel computing community mainly because of its high and more reliable throughput. Various research results indicate that ATM shows better performance when used with parallel computing and distributed processing[LIN94, HUAN94].

MPI (Message Passing Interface) is a standard, portable and formally specified parallel (multicomputers) programming language. It uses a message passing model with an explicit control for parallelism.

The bidding strategy, an adaptive, distributed and dynamic load balancing algorithm, has been chosen because it makes use of many short messages for communication, and ATM is employed to see whether it can improve this kind of communication. The underlying concept in the bidding approach is negotiation via bid submissions between components

of a load balancing system (cooperative agents). The bidding negotiation is a local process that compares bids without a centralized control.

The vector or Barak and Shiloh's strategy is also an adaptive, distributed and dynamic load balancing algorithm. However, the exchange of load vectors is carried via a randomly selected node, which is not the same as in the broadcasting approach adopted by the earlier method. The length of exchange messages (load vectors) involved is also longer than those in the bidding strategy.

For our experimentation, first some network benchmark programs were written to test the throughput of the socket and MPI before implementing the load balancing algorithms. We then tested the ATM architecture with IP, followed by the ATM protocol and compared the results to those from Ethernet with IP. An analysis was carried out with particular emphasis on the percentage of utilization of the bandwidth.

In the implementation of the two load balancing strategies, three main components are defined.

1. A module to find the best node for a given process, which is invoked whenever a new task arises.
2. A module to evaluate the load of local processes and then send this information upon request by the first module.
3. A module to handle the execution of a process locally as well as remotely.

The conclusion that can be drawn from the experimentation is that ATM card drivers need to be fine tuned for smaller packet data size irrespective of the difficulty for applications to make full use of the bandwidth. In using the MPI language, the load balancing algorithms suffer a certain layer overhead. In order to lessen this overhead, the most efficient way would be to incorporate the above algorithms within the operating system. According to the results obtained for the two algorithms, the effect of ATM compared to Ethernet is better for the vector strategy compared to the bidding strategy. An analysis of the results are presented and the reasons for the phenomenon are given. As a suggestion for future work, we shall implement other similar algorithms and also to test them on larger networks.

Part One

Fundamentals

This part presents a general overview and the basic principles of each related topic. The following are some background knowledge required before one can proceed to the proposed implementation for this project.

Chapter 1 Introduction

Chapter 2 Distributed And Parallel Systems

Chapter 3 ATM Networks

Chapter 4 Load Balancing

Chapter 5 Message Passing Interface (MPI)

Chapter 1

Introduction

This thesis is divided into two parts: the first being on fundamentals and the second on experimentation. The first part gives an overall view and the fundamental background required for carrying out the implementation of load balancing programs on Ethernet and ATM, in particular using MPI (Message Passing Interface) and the BSD (Berkely System Development) sockets. The second part shows the ways for implementing two selected algorithms, namely *bidding*[GOSC91, STAN84] and *vector*[GOSC91, BARA85] . These two algorithms have been selected mainly due to their difference in transfer policy. *Bidding* makes use of shorter but with more exchanges in messages, whereas *vector* makes use of longer but with less exchanges in messages. ATM is tested to see its impact on these two different algorithms, and from the analysis of the results obtained, the overall effect of ATM on other load balancing algorithms may be predicted.

1.1 Scope and Objectives of Thesis

In the hope that one can benefit from one of the hottest networking technologies being ATM, and one of the most standard parallel computing languages, namely MPI, this work is carried out in order to compute their respective impacts on load balancing algorithms, namely *bidding* and *vector*, in a distributed environment (LAN).

This thesis is to prove that ATM will perform better than Ethernet for load balancing in distributed systems given the existing norm that the load balancing algorithms make use of a large message buffer size of greater than 256 bytes for process placement. The thesis also shows that the vector algorithm has a better speed up effect than the bidding algorithm when using ATM simply because the vector algorithm makes use of larger packets for message passing over the network.

In order to attain our objectives above, two different load balancing programs, namely bidding and vector, are implemented. These are further implemented using two different types of network message passing, BSD sockets and the MPI LAM version 6 [LAM, GROP94, MALA96] (cf. figure 1.1). The two algorithms are tested on Ethernet and ATM respectively and measurements and readings from these experiments are recorded for analysis. Resources used are limited, currently using 2 computers for ATM but with simulation of nodes.

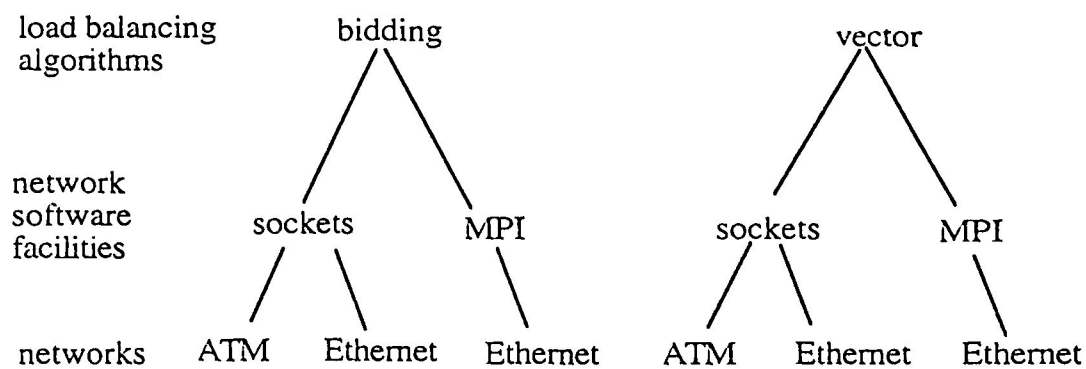


Figure 1.1 Scope of experiments on load balancing in a distributed environment.

1.2 Introduction to Load Balancing

Load balancing is a technique used to distribute processes among processors within a parallel computer or among workstations connected by a network in order to equalize the workload amongst them. The objective is to maximize system throughput by optimizing the utilization of resources. There are basically two types of load balancing : dynamic and static. Dynamic load balancing algorithms may involve process migration whereas static algorithms do not. In static load balancing, decisions on distribution are made based on assumptions on the behaviour of a process (eg. execution time) as well as on the environment of the processors (eg. current load, transfer time). However, knowledge (eg. execution time) about a process is difficult to ascertain before its execution. On the other hand, dynamic load balancing tries to monitor the load of processors at some

intervals and attempts to equalize the load by allowing the process to migrate even in the middle of an execution. However, process migration is heavy in terms of resources since it needs to recover its context of the memory.

Bidding is an adaptive, distributed and dynamic load balancing algorithm. The underlying concept within the bidding approach is the negotiation between components of a load balancing system (cooperative agents) and the submission of bids for contract. The bidding negotiation is a local process, without a centralized control. The algorithm evaluates the information from its own perspectives and a final agreement is achieved by comparing the bids. This algorithm has been chosen because it involves many short messages throughout a communication session. ATM is tested to see if it can help improve this type of communication.

Vector or Barak and Shiloah's algorithm is also an adaptive, distributed and dynamic load balancing algorithm. Each computer makes use of the same algorithm to maintain the information about the load within a certain number of computers. The exchange of load vectors is carried out through a randomly selected node. This algorithm has a different behaviour in terms of cost in communication compared to the *bidding* algorithm. The length of exchange messages (load vectors) is longer, but the whole cycle of a communication session within this algorithm is shorter than that for bidding. This is because bidding makes use of the broadcasting technique which is generally more expensive for communication (ATM).

The different results obtained from the implementation of these two algorithms could be collected and compared.

1.3 Justifications for ATM

ATM is popular in the parallel computing community mainly because of its high and more reliable throughput. Research results indicated that ATM shows better performance with parallel computing and distributed processing[LIN94], [HUAN94]. However, not much (if any) research has been done on 'load balancing using ATM'. Since ATM has

the ability to move large information chunks quickly across a network, using ATM as a carrier in a distributed environment where large processes are involved seems to be a good choice.

1.4 Justifications for MPI

The *Message Passing Interface*, commonly referred to as MPI, is a standard, portable and formally specified parallel (multicomputers) programming language. It uses a distributed memory model with an explicit control for parallelism, which is also known as the "message passing programming model". This is different from the shared-memory data-parallel model which is used by High Performance Fortran (HPF). Although it is easier to program in a shared-memory data-parallel model, the distributed memory or message passing model is more direct and is more efficient in a distributed environment where the process can only read and write to its local memory [MALA96]. Some of the reasons why MPI is selected are as follows: MPI is available freely (freeware) and it is portable. This will then allow others to further our research on the implementation of load balancing algorithms in a distributed environment. Furthermore, MPI is a new standard and exploration by researchers would indeed help further improvement.

1.5 Chapter by Chapter Summary

This thesis is divided into 2 parts containing a total of 9 chapters and 1 appendix.

The first part presents a general overview and the basic principles involved in 4 general areas, namely distributed and parallel systems, ATM networks, load balancing and Message Passing Interface (MPI).

Chapter 2 lays the groundwork for distributed and parallel systems. The definitions, characteristics and various concepts in distributed and parallel systems are explained.

Chapter 3 deals with the fundamental concepts in ATM networks. This chapter give an introduction to ATM networks which covers the characteristics, features and advantages of ATM.

Chapter 4 presents a general overview of load balancing. Various characteristics of load balancing algorithms such as static or dynamic, distributed or centralized are given here. Process migration and basic components of load balancing algorithms are also explained in this chapter.

Chapter 5 introduces the Message Passing Interface (MPI). The basic elements of MPI such as messages and processes, datatypes, MPI communication are presented. Comparisons between message passing and distributed shared memory are also given in this chapter.

The second part of this thesis covers the implementation and the analysis of results obtained from the implementation of the bidding strategy and the vector strategy. This part consists of 4 chapters ranging from 6 to 9.

Network benchmark programs are written to test the throughput on Ethernet and on ATM in chapter 6. Various programming facilities such as BSD sockets, the MPI and ATM's API are tested. Comparisons and explanation of results are provided as part of the analysis.

Chapter 7 covers the implementation of the bidding strategy. This chapter explains how the bidding algorithm was implemented using both BSD sockets and MPI. Certain restrictions and conditions under which these experiments were carried out are also stated.

Chapter 8 deals with the implementation of the vector strategy. BSD sockets and MPI are used to implement the vector strategy. Both chapter 7 and 8 provide an indepth analysis of the results obtained.

Chapter 9 provides the conclusions drawn from observations in the experiments and the analysis of the results. Suggestions are proposed for further improvement and future work.

Chapter 2

Distributed And Parallel Systems

2.1 Definition

Andrew S. Tanenbaum [TANE95] defines a distributed system as follows:

"A distributed system is a collection of independent computers that appear to the users of the system as a single computer."

A. Goscinski mentioned [GOSC91] :

"Distributed systems is a term used to define a wide range of computer systems, from weakly-coupled systems such as wide area networks to strongly-coupled systems such as local area networks to very strongly-couple systems such as multiprocessor systems."

In fact, there are no clean-cut distinctions between distributed systems and parallel systems. Distributed systems are more general than parallel systems in the sense that its hardware may include **multiprocessors** with shared memory (a parallel computer) and **multicomputers** (workstations or PCs with network).

2.2 Characteristics

Transparency is an important feature of a distributed system. There are basically five types of transparency :

Location Transparency

The user does not need to know where the resources reside.

Migration Transparency

Resources can be moved without the user's knowledge.

Replication Transparency

Resources may exist in a few copies, for the purposes of performance and security, but users may not be aware of their existence.

Concurrency Transparency

Users can share resources without bothering with problems of consistency and integrity .

Parallelism Transparency

Processes can run in parallel without receiving explicit instructions from users.

2.3 Advantages

- The main advantage of distributed and parallel systems is **economy**. Such systems can have a better price/performance ratio than centralised systems. In the case of NOW (Network of Workstations), experimentation as reported in [ANDE95] showed that a number of hypothetical 256-node RS/6000 NOWs on an ATM network and with a low-overhead, low-latency communication system can deliver a better price/performance ratio than a 16 node C-90 or a 256-node Paragon.
- Distributed and parallel systems also give higher **reliability, availability and security** when compared to a single centralised system. For example, if the percentage of down time for a machine is 10%, then the percentage for 4 machines being down at the same time is 0.0001. If there are five machines in the above example, the remaining machine can still work with 20% of the total workload.
- Many **applications are inherently distributed**, examples of which include order processing systems and inventory systems.
- Another advantage is **scalability**. Distributed and parallel systems can have an incremental growth by adding or upgrading the workstations or network.
- The fifth is **sharing of data and information**. Many users may access the same database as well as communicate with each other through the network.
- Last but not least is **speed**. Spreading the workload over the machines available in the network, the throughput of applications will increase.

2.4 Parallel Systems

A parallel system is a system where many processes are running at the same time to solve a single problem. Parallel programs can run in a multiprocessor or a multicomputer environment. A multiprocessor environment exists on a shared memory basis and the system is tightly coupled, while a multicomputer environment depends on private memory and the system is loosely coupled. The multicomputer environment can also be homogeneous or heterogeneous. With the proliferation of personal computers and workstations and with rapid advances in communication technology, there is a bigger incentive to develop parallel programs in a multicomputer environment. As a result, we can now see a convergence of the two systems, namely in the form of distributed and parallel systems.

2.5 Classification of Hardware

Flynn's classification system [FLYN66] divides computers into four categories : SISD, SIMD, MISD and MIMD (cf. table 2.1). Two criteria used in this classification are the number of instruction streams and the amount of data.

Table 2.1 Computer system classification of Flynn.

SISD Single Instruction Single Data	SIMD Single Instruction Multiple Data
MISD Multiple Instructions Single Data	MIMD Multiple Instructions Multiple Data

SISD is the traditional single CPU system. SIMD refers to vector computers or array computers which are constructed using one control processor and many processor elements (PEs) that are principally made up of an arithmetic/logic unit and a local memory. The same instruction unit is executed in parallel over many different data units using the many PEs. This type of machines is very useful for computations that repeat on many sets of data. Although the third category exists, there is no known computer which is based on the MISD model. In the context of parallel and distributed system, the most interesting model is MIMD. There are two types of MIMD (cf. figure 2.1): **multiprocessors** which are tightly coupled (cf. figure 2.2), and **multicomputers** which are loosely coupled (cf. figure 2.3). Each category has two types of architecture for the interconnection network which is either bus or switched. There are many types of switches which connect CPUs and memories, for example a crossbar switch and an omega switching network (cf. figure 2.4). MIMD can have different types of network topologies (eg. grid and hypercube) which connect the PEs or the computers (cf. figure 2.5). In figure 2.6, a more detailed classification of current parallel machines is shown.

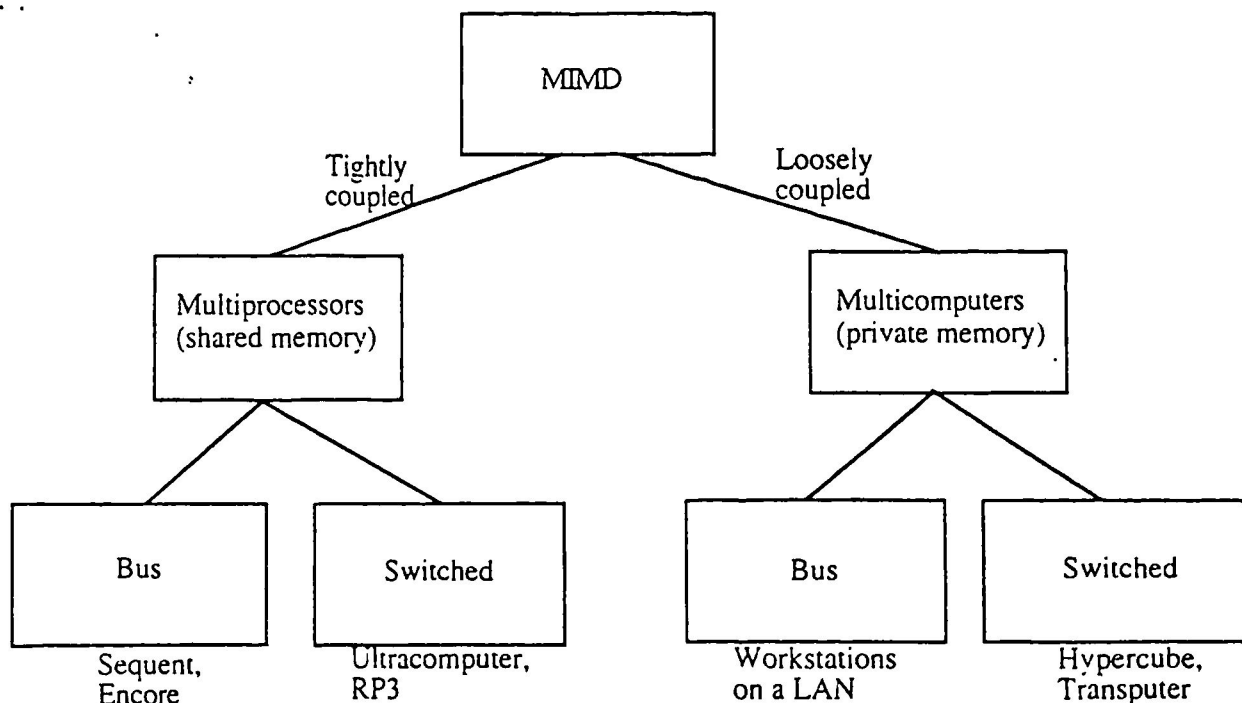


Figure 2.1 A taxonomy of parallel and distributed computer systems.

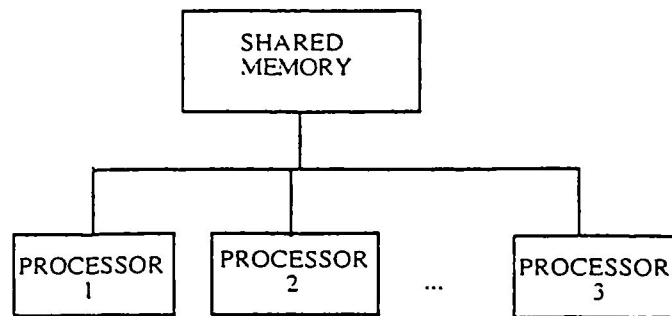


Figure 2.2 Multiprocessor computer organization with common bus.

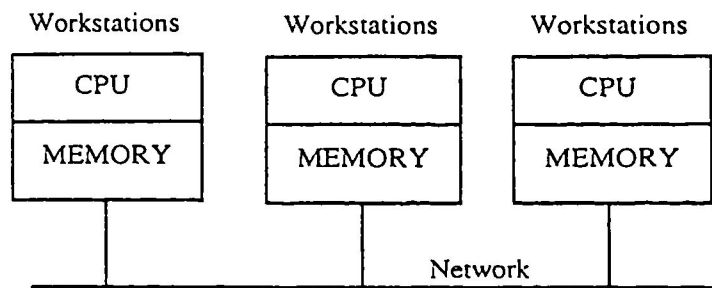
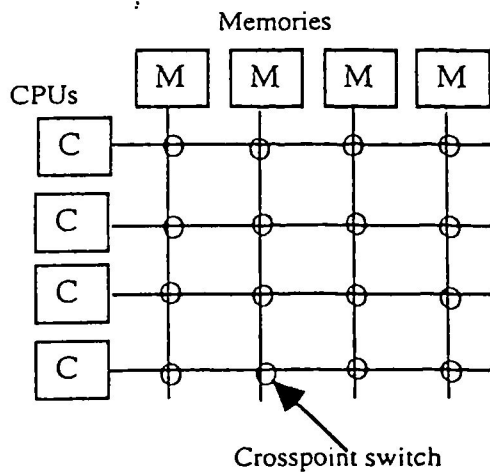
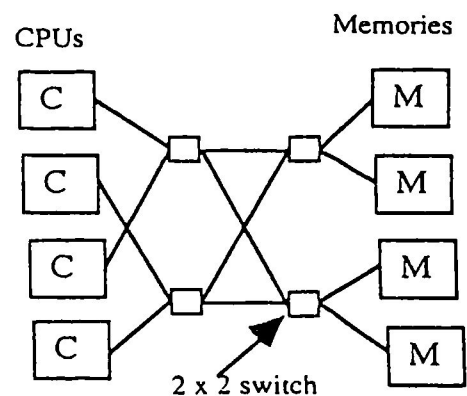


Figure 2.3 A multicomputer consisting of workstations on a LAN.



(a)



(b)

Figure 2.4 (a) A crossbar switch. (b) An omega switching network.

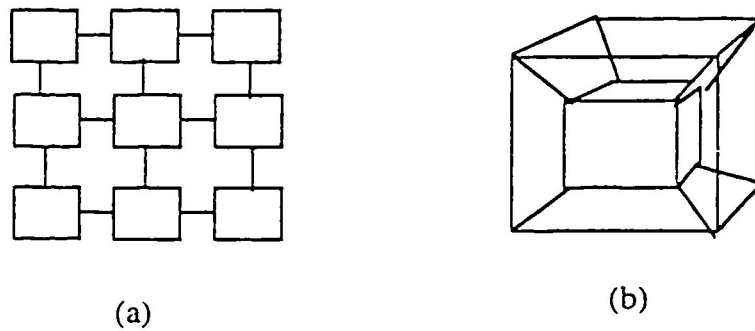


Figure 2.5 (a) Grid architecture. (b) Hypercube architecture.

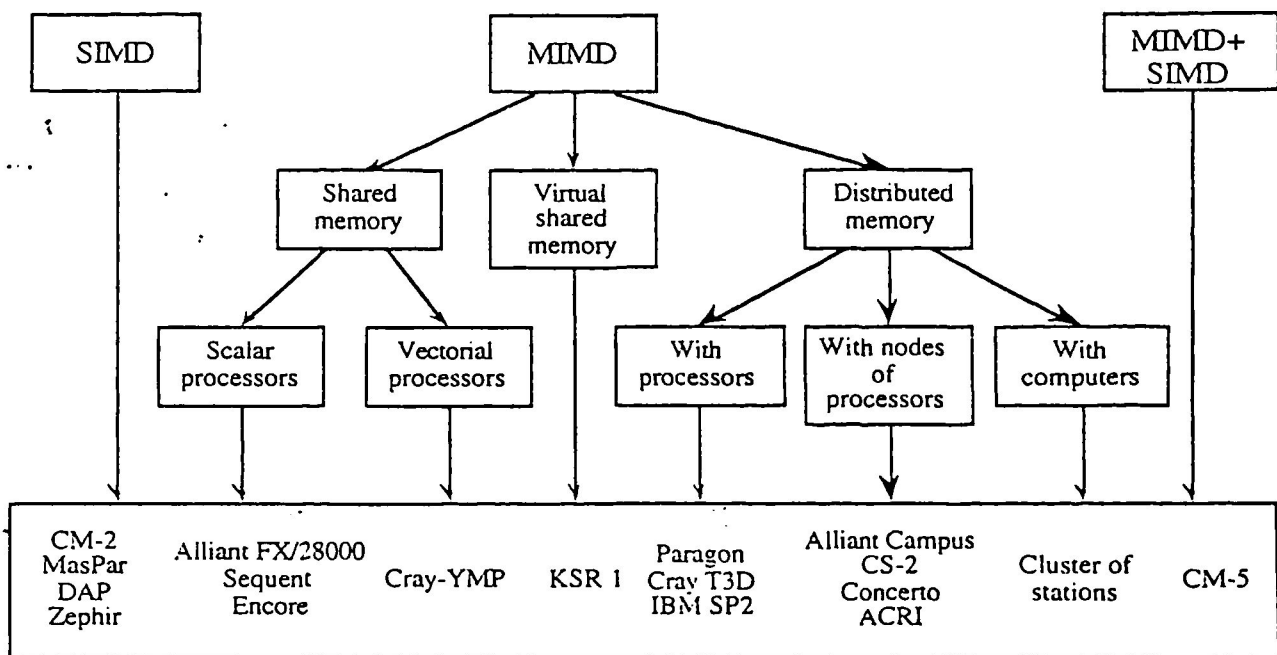


Figure 2.6 Classification of parallel machines

2.6 Concepts in Distributed And Parallel Software

2.6.1 Programming Languages

Similarly as in loosely and tightly-coupled hardware, it is also possible to separate software into two basic categories: the **multiprocessor** model and the **multicomputer** model. These are two extremes in a spectrum of parallel computer architecture within which many other intermediate computer architectures exist. These two families of programming models have become widely used. HPF (High Performance Fortran) employs the first model with shared memory and implicit parallelism, whereas MPI (Message Passing Interface) uses the second model with distributed local memory.

Another important distinction in parallel programming is whether the parallelism is explicitly coded by the programmer or it is generated automatically by the compiler. It is indeed very interesting to be able to let the compiler identify parallelism implicitly because then existing sequential codes could be easily ported to parallel or distributed computers. However at this point in time, it is still very difficult for a compiler to fully exploit the parallel features of the variable parallel architecture and identify all the potential parallelism in a given computation.

The third major difference in parallelism is extracted from the level of operators of a programming language or from the level of its control structures. For example, multiplication of two matrices in parallel can be expressed in at least two different ways.

In the first case: `matrice_A px matrice_B, /* px is a parallel operator of multiply */`

In the second case: `forall (i=1; i < 100; i++) /* forall executes the for loop concurrently */`

`forall(j=1; j < 100; j++)`

`forall(k=1; k < 100; k++)`

`matrice_C[i,j] += matrice_A[i,k] * matrice_B[k,j]`

These major distinctions give rise to different characteristics in different programming models.

2.6.2 Types of Parallel Computing

There are three models for parallel computing depending on the way operations are used on the given data. These are Data-parallel or SIMD, SPMD and MIMD. Data-parallel uses the same operations on different data; the second, SPMD uses the same program on different data; and finally, MIMD uses different programs on different data.

2.6.3 Operating Systems

One can also divide operating systems for the distributed environment into three types as follows: network operating systems, distributed operating systems and multiprocessor timesharing systems.

A network operating system is a loosely-coupled software running on a loosely-coupled hardware. The objectives are to improve computer performance within the bounds of the processor capacity and to share the available resources. The network operating system works as an intermediate layer between the local operating system and the communication module. When the local operating system requires a service from the remote resource, the request is passed to the network operating system which then provides access to the remote resource. Usually, the user needs to know where the requested remote resource is located. In other words, the facility does not provide location transparency, but users have a high level of autonomy of their machines.

A distributed operating system is a tightly-coupled software running on a loosely-coupled hardware. It provides location transparency which makes the entire network seem to be a single system to users. Users need not be aware of where their files are stored or where a program is executed. Fault tolerance is also very high and resource management is done globally.

In the case of the multiprocessor timesharing system, the system is a tightly-coupled software running on a tightly-coupled hardware. There is only one copy of the operating system running. Communication is achieved through shared memory and the system makes full use of the architecture of the parallel machine.

2.6.4 Interprocess Communication (SYSTEM V IPC)

The UNIX System V IPC package consists of three mechanisms: messages, shared memory and semaphores. Messages allow processes to send formatted data streams to arbitrary processes. Shared memory allows processes to share parts of their virtual address space. Semaphores allow processes to synchronise execution. All three share some common features, one of which is a table containing entries describing all instances of the mechanism. Each entry has a numeric *key*. A "get" system call is used by each mechanism to create a new entry or to retrieve an existing one. All three mechanisms have a set of read-write-execute permissions similar to the file permission mode, as well as a "manipulation" system call to query, update or remove an entry from the table. Data structures for messages, shared memory and semaphores are shown in figures 2.7, 2.8 and 2.9.

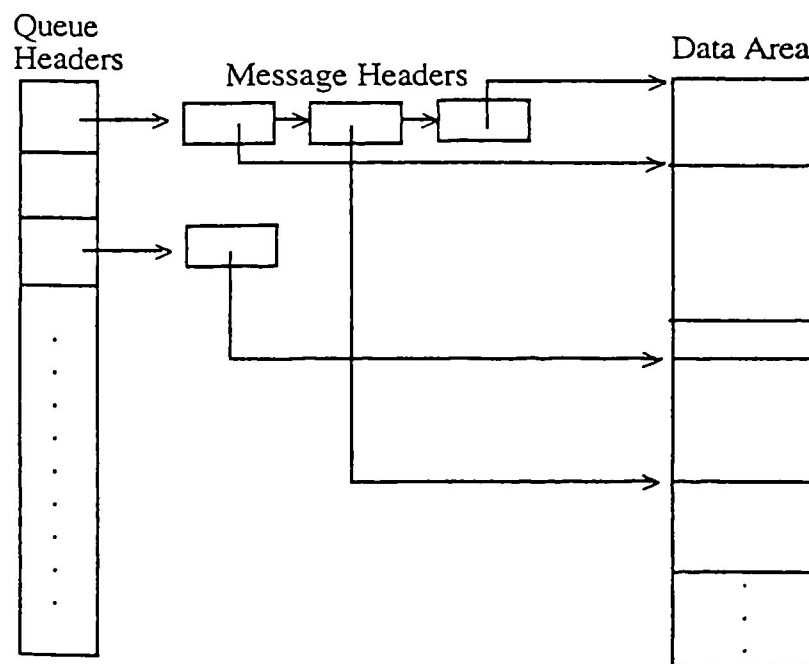


Figure 2.7 Data structures for messages.

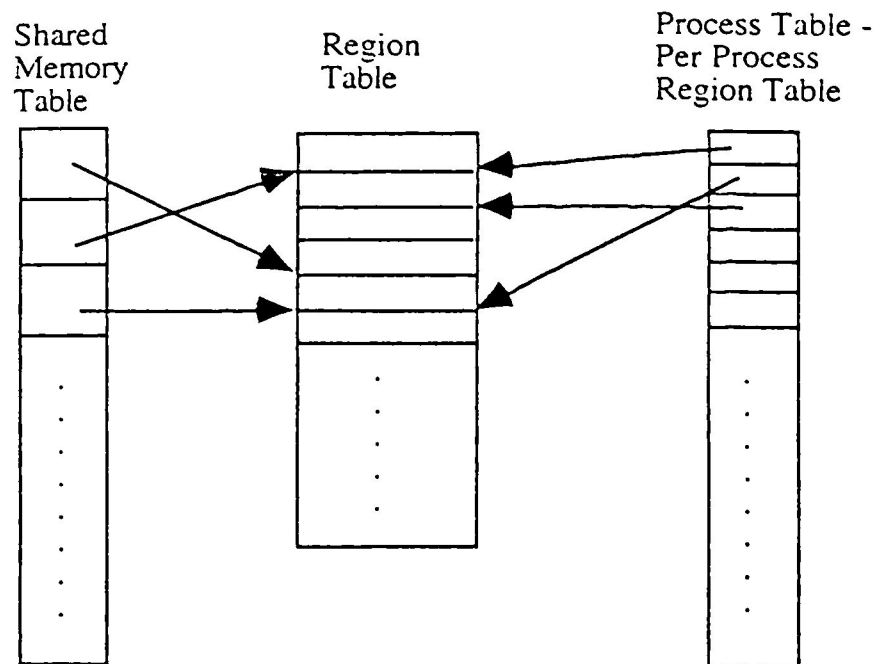


Figure 2.8 Data structures for shared memory.

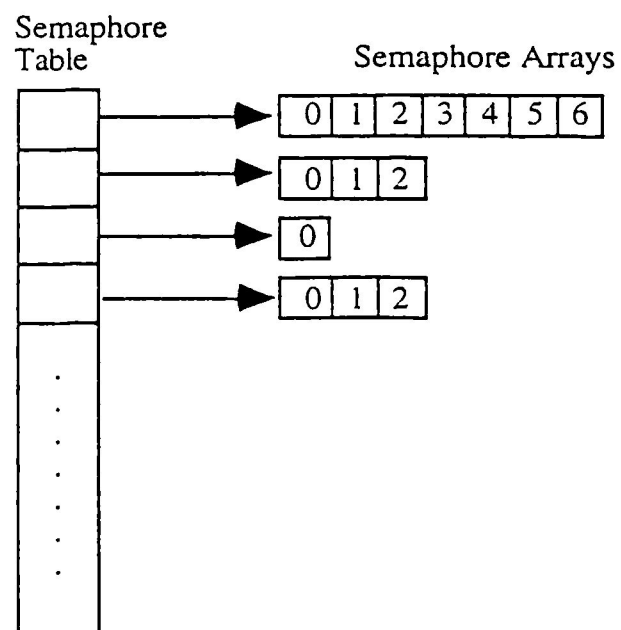


Figure 2.9 Data structures for semaphores.

2.6.5 Sockets

To provide common methods for interprocess communication and to allow for the use of sophisticated network protocols, the BSD system provides a mechanism known as *sockets*. The kernel structure consists of three parts: the socket layer, the protocol layer, and the device layer.

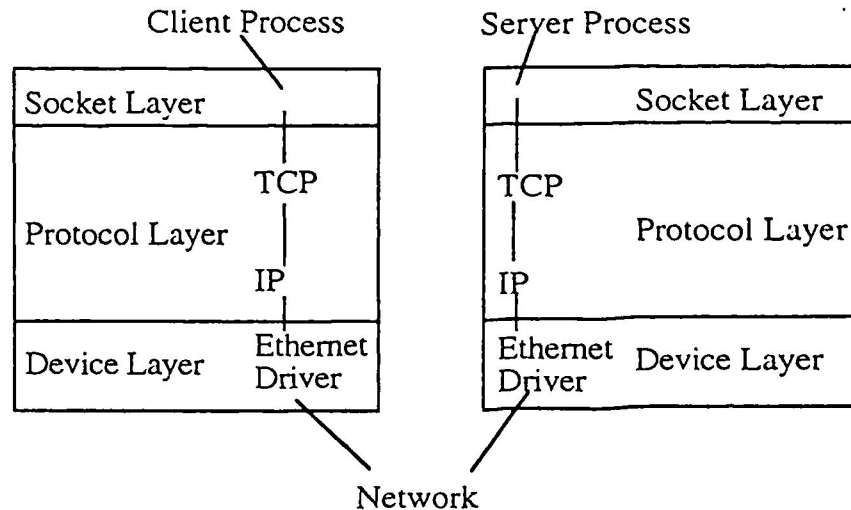


Figure 2.10 Sockets Model.

The socket layer provides the interface between system calls and the lower layers. The protocol layer contains a protocol module used for communication. The device layer is made up of network device drivers.

A client process make use of a socket to communicate with the server on the other end, the latter making use of another socket. Sockets support the "UNIX system domain" for processes communicating on one machine and the "Internet domain" for processes communicating across a network. Each socket has a type: virtual circuit (stream socket) or datagram. The Transmission Control Protocol (TCP) provides a virtual circuit service which allows for a reliable sequential delivery of data. The User Datagram Protocol (UDP) provides a datagram service which does not guarantee reliability, sequentiality nor unduplicated delivery. However one has to pay a cost for the 'reliability' offered by TCP/IP. This cost comes in the form of increased network usage and increased delays in transmission.

The socket mechanism of TCP/IP involves several system calls as shown in the following table.

Table 2.2 Some common socket commands using TCP/IP protocol.

Command	Brief Explanation
<code>sd=socket(format, type, protocol);</code>	establishes the communication link.
<code>bind(sd, address, length);</code>	associates a name with the socket descriptor.
<code>connect(sd, address, length);</code>	connects to an existing socket.
<code>listen(sd, qlength);</code>	specifies maximum queue length.
<code>nsd = accept(sd, address, addrlen);</code>	receives incoming request.
<code>count = send(sd, msg, length, flags);</code>	transmits data over a connected socket.
<code>count = recv(sd, buf, length, flags);</code>	receives data over a connected socket.
<code>shutdown(sd, mode);</code>	closes a socket connection.

...

The basic steps in a server process using a socket are as follows. The process establishes a communication socket by calling *socket*. Next, it *binds* a name or address to the socket, after which it invokes the *listen* call to specify the internal queue length for incoming messages and enters a loop waiting for incoming messages. Upon receiving a message, the *accept* call returns a new descriptor for the incoming request. The server will *read* or *recv* the message and *close* their respective descriptors.

The basic steps in a client process corresponding to a server process begins with the creation of a socket using the *socket* call, and then it issues a *connect* call to the socket of the server process. The client uses the *write* or *send* call to transmit the message across before it *exit* s.

2.6.6 Remote Procedure Calls

Most of the facilities for distributed processing are built around the communication of input (receive) /output (send) messages. To make a distributed system seem more like a centralized system, a remote procedure call (RPC) is proposed. The idea here is to hide message passing processes from the programmer. The RPC works like a local procedure call except that it passes the procedure together with its parameters to the client stub and on to the kernel where these are packed as a message and sent to the server. The procedure is executed in the server and the result is sent back to the client. The control is then returned to the suspended client procedure. The intermediate steps of message passing and parameter marshalling are transparent to the programmer, which is a more elegant paradigm compared to the general message passing model. Although the basic idea sounds simple, subtle problems do exist. Firstly, the machines may be heterogeneous and so parameter conversion may be required. Secondly, as each machine is running in a different address space, the parameter for the pointer to memory space may cause a problem. One of the possible solutions to the **call-by-reference** parameter problem is by using the **call-by-copy/restore**. To this remote procedure is associated a formal specification of the procedure, specifying exactly what the parameters are, which ones are the input and which ones are the output (or both), and their respective sizes. This information would also eliminate certain unnecessary operations. For example, input parameters need not be sent back to the client and the output parameters need not be copied to the server, thus optimizing the call-by-copy/restore operation. Another possible problem is that both machines may crash at the same time and so certain protocols are needed to handle the problems that may arise due to such a failure.

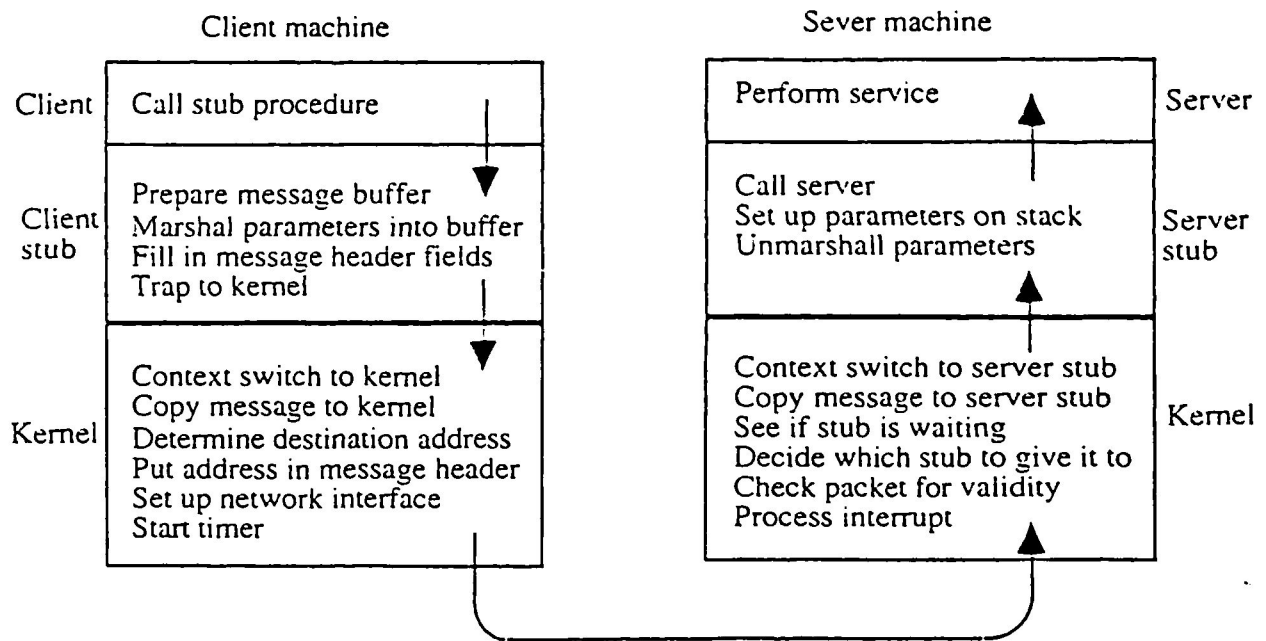
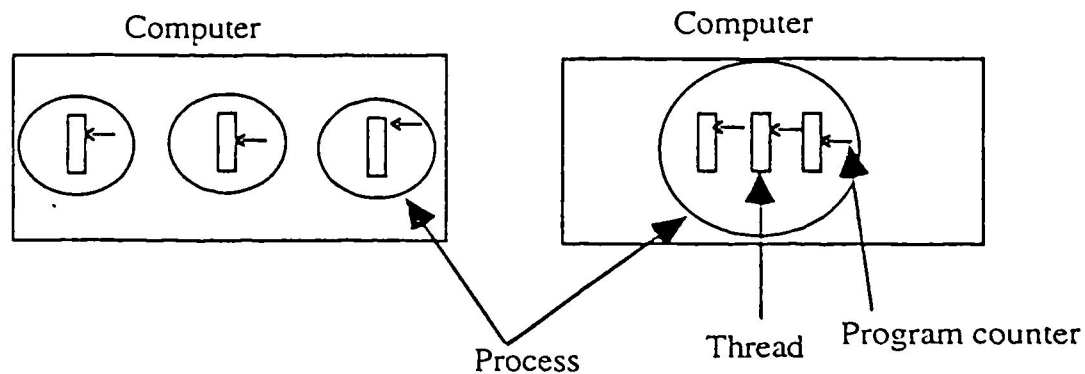


Figure 2.11 The steps involved in the RPC from client to server.

2.6.7 Light Weight Processes - Threads

In traditional operating systems, each process has an address space and a single thread of control. However, one sees that performance in terms of speed can be enhanced with the use of multiple threads which share the same address space and the same set of open files and with child processes running in quasi-parallel. For example, while one thread is waiting for the disk, the second thread can take over the CPU. Threads are like processes but they are highly related to each other. Sharing the same resources, as in the same address space, makes it easier to communicate with each other and more efficient to execute (a light weight process). The relation between processes and threads is illustrated in figures 2.12 and 2.13.



(a) Three single-thread processes.

(b) One process with three threads.

Figure 2.12 Single-thread and multi-thread processes.

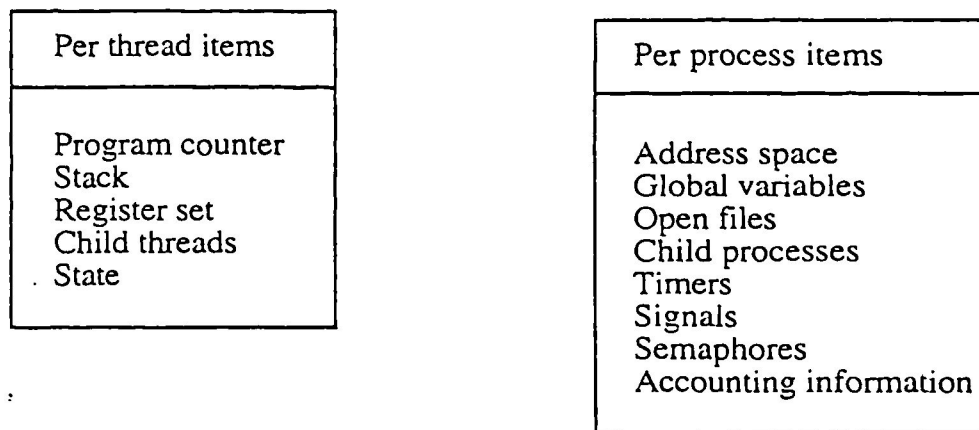


Figure 2.13 Per thread and per process items.

In a given process, a thread can be used following one of the three models shown in figure 2.14. The dispatcher/worker model can be applied to applications such as the file server. The dispatcher reads incoming requests for work from the system mailbox (buffer) and delegates a request to an idle worker thread by waking it up from its sleep. If a worker thread is going to sleep while awaiting completion of a disk operation for a needed block, another thread, possibly another dispatcher, is invoked. This will keep the CPU busy all the time.

In the team model, all threads are treated as equal and do not have a coordinator thread like in the dispatcher model. As such, each thread receives and processes its own requests. A thread

may be assigned to handle only a particular task, and so if an incoming task cannot be handled by this thread, it is then be put into a job queue and may be picked up by another thread.

In the pipeline model, the output generated from a thread is passed as the input to the next thread. Each thread is responsible for a particular task. For example, pipelining may be employed in the UNIX command lines (eg. `command1 | command2 | command3`).

Threads are useful for handling signals such as interrupts from a keyboard. Threads can work equally well on a single timesharing CPU or a on true multiprocessor.

A set of primitives for threads is called a thread package. Since threads can access to the same memory space, the control for concurrent access to this critical region is therefore needed. A binary semaphore, called **mutex** is usually used, and this has only two states which are locked and unlocked. Another synchronization feature is a condition variable which is used for long-term waiting periods for resources to become available.

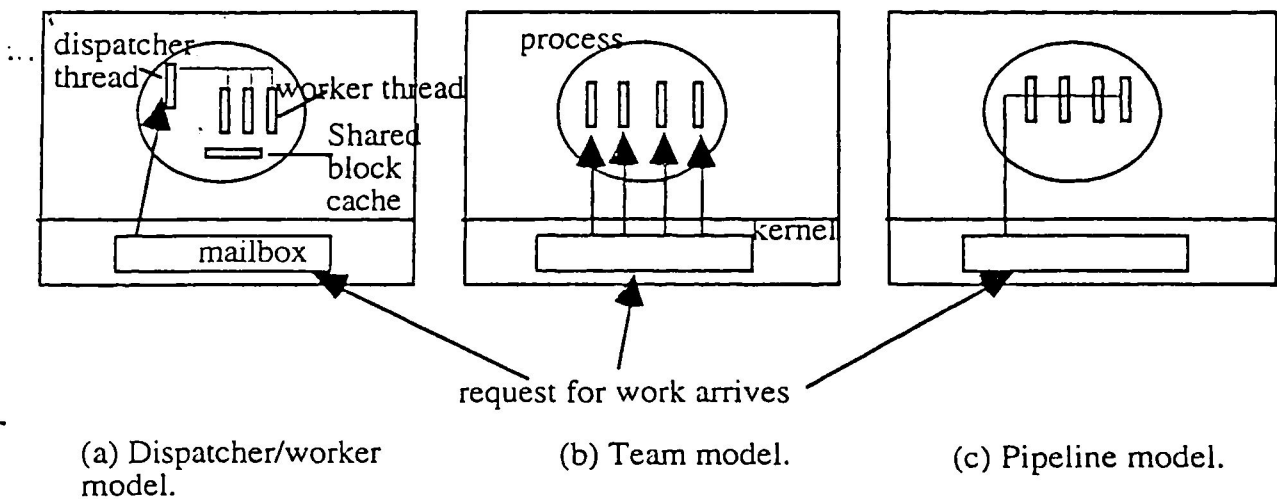


Figure 2.14 Three models for the use of threads in a given process.

2.6.8 CHORUS

One of the most modern, microkernel-based operating system, called Chorus [TANE95, BRIC91, CHOR, GRAB91], is presented here as a case study in relation to the various concepts covered in this chapter. A microkernel is an operating system containing only the essential services. Chorus started out at the French research institute INRIA in 1980 as a research project in distributed systems. It has since evolved through a few versions and has now matured as a commercial product.

Chorus is structured in layers. The lowest layer is the microkernel or nucleus which consists of some low level kernel calls. The microkernel has functions such as a scheduler, virtual memory management, and IPC. The scheduler is a priority-based preemptive FIFO, with time sharing and runs in real-time. Virtual memory management is divided into two parts: the first being within the microkernel and the other outside. The communication protocol between the two parts is well defined. The memory management module adopts a paging technique, but with the higher level memory management being akin to **mappers** in MiX which can use more specific concepts like *region* (contiguous range of virtual addresses) or *segment* ... (contiguous collection of protected bytes). Distributed shared memory is also supported by Chorus. The low layer interprocess communication manager responsible for UIs, ports and the sending of messages in a transparent way, provides such services to the higher layer IPC manager which consists of a System V IPC package, RPC communication, and synchronization.

Due to the microkernel architecture adopted, various other OS facilities can be built on top of Chorus, amongst which are SVR 4.0, SVR 3.2, SCO ODT 3.0, BSD 4.3, and POSIX real-time. These are called subsystems and they can emulate very well the said monolithic OSs by providing binary compatibility and good performance.

Chorus has evolved from a communication model to a processing model by providing multi-thread processes and RPC. Multi-threading allows for multiplexing of servers, simplifying their control structures and potentially increasing parallelism. A process (actor) can be

associated with one or more ports which serve as buffers for incoming messages. Ports can be grouped together from different processes (cf. figure 2.15).

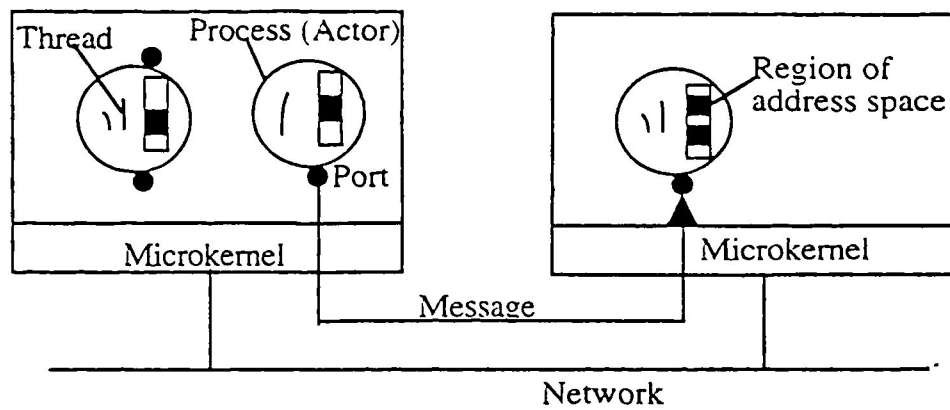


Figure 2.15 Process communication in Chorus.

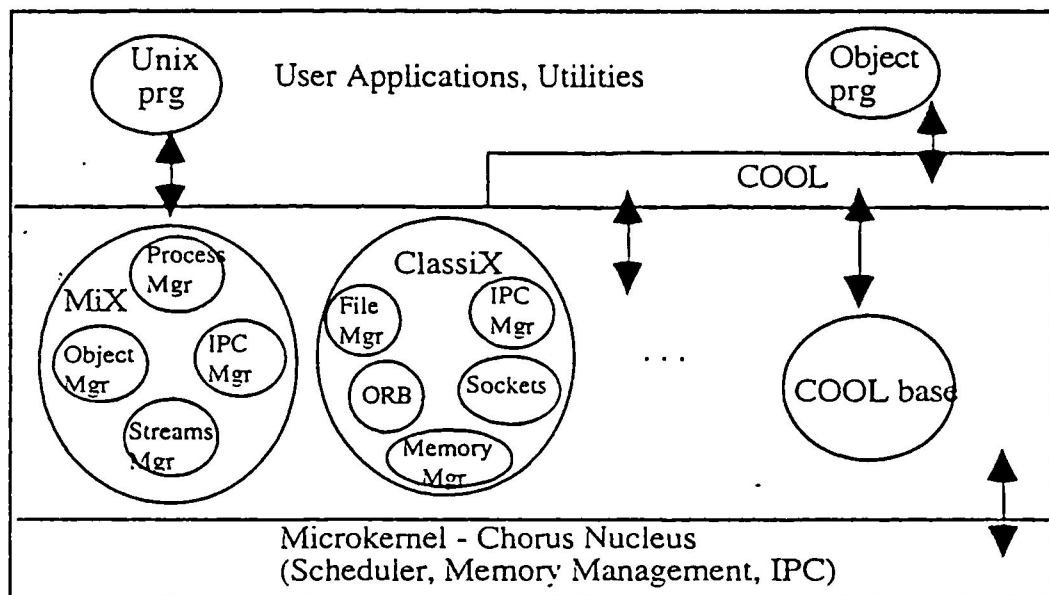


Figure 2.16 Architecture of Chorus.

COOL (Chorus Object-Oriented Layer) is designed as an object oriented environment. An *OMG COBRA 2.0* compliant *Object Request Broker (ORB)*, COOL narrows the gap between the OS and object-oriented programming languages. A COOL program is able to run side by side with UNIX programs owing to the microkernel layer as shown in figure 2.16, where the COOL base layer provides location transparency like a distributed file system. One of COOL's most important services is memory abstraction, which is based on a cluster which can hold a group of objects together. A cluster can also be mapped into address spaces of multiple processes running on different machines.

2.6.9 Glossary of Object Technology Terms

- *Object Management Group (OMG)* was founded in 1989 to create a component-based software marketplace by furnishing some guidelines and object management specifications to provide a common framework for application development.
- *Common Object Request Broker Architecture (COBRA)* is an industry wide standard for distributed object technology, first introduced by OMG in 1991. It is a standard for building distributed applications that are platform-, language- and machine-independent.
- *Object Request Broker (ORB)* provides the middleware that establishes the client/server relationship between objects. ORBs provide location transparency and interoperability between applications on heterogeneous distributed environments.