1092

A VIABLE SCHEME FOR LOAD BALANCING ON PARALLEL MACHINES

Dissertation submitted to The Jawaharlal Nehru University
in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

IN

COMPUTER SCIENCE

ABDAAL KHALEEQUE

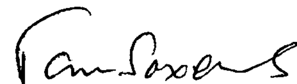
SCHOOL OF COMPUTER AND SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI-110 067

JANUARY 1994

CERTIFICATE

This is to certify that the dissertation titled " **A Viable Scheme for Load Balancing on Parallel Machines** " being submitted by **ABDAAL KHALEEQUE** to Jawaharlal Nehru University, New Delhi in partial fulfilment of the requirements for the award of the degree of **Master of Technology** is a record of the original work done by him under the supervision of **Prof. P.C.Saxena**, Professor, School of Computer and System Sciences, Jawaharlal Nehru University, New Delhi during the year 1993, Monsoon Semester.

The results reported in this dissertation have not been submitted in part or in full to any other university or institution for the award of any degree or diploma.



Prof. K.K.Bharadwaj	**Prof. P.C.Saxena**
Dean,	**Professor,**
School of Computer and	**School of Computer and**
System Sciences,	**System Sciences,**
Jawaharlal Nehru University,	**Jawaharlal Nehru University,**
New Delhi.	**New Delhi.**

To

my parents

ACKNOWLEDGEMENT

*I am immensely indebted to my supervisor **Prof. P.C.Saxena,** Professor, School of Computer and System Sciences, Jawaharlal Nehru University, New Delhi for suggesting me this topic. I express my sincere thanks to him for his personal involvement during the period of my work and his eloquent guidance which has been indispensable in bringing about a successful completion of the dissertation.*

*I extend my sincere gratitude to **Prof. K.K.Bharadwaj,** Dean, School of Computer and System Sciences,Jawaharlal Nehru University for providing me with the environment and all the facilities required for the completion of my dissertation.*

Special thanks are to Mr. Vikas Ahluwalia, Ph.D. scholar, School of Computer and System Sciences, for providing me with some very useful references and also helping in the implementation phase of the project.

I am also grateful to my friend Shanavas for taking pains to go through the manuscript and also for his useful comments.

Thanks are due to my friends Manoj, Jisnu and Sumitra for giving me company in the long hours of nights during the period of this dissertation.

Finally, I would like to take a moment to thank my family members for their patience and encouragement, without which this dissertation would not have been complete.

ABDAAL KHALEEQUE

When we mean to build,

We first survey the plot, then draw the model;

And when we see the figure of the house,

Then must we rate the cost of the erection.

William Shakespeare

CONTENTS

"Declare the things that are to come hereafter".
Isaiah 41:23

CHAPTER 1 **INTRODUCTION** ... 1

1.1 Aim of the Project ... 1

1.2 Why Parallelism ... 2

1.3 The Problem with Parallelism ... 4

1.4 Relevance of The Project ... 7

1.5 Organisation of This Report ... 7

CHAPTER 2 **CLASSIFICATION OF PARALLEL COMPUTERS** ... 8

2.1 Flynn's Classification ... 8

2.2 Structural Classification ... 11

CHAPTER 3 **SOFTWARE ISSUES IN PARALLEL COMPUTING** ... 28

3.1 The Mapping Problem ... 28

3.1.1 Decomposition of the Problem ... 29

3.1.1.1 Processor Farm Parallelism ... 29

3.1.1.2 Geometrix Parallelism ... 30

3.1.1.3 Algorithm Parallelism ... 32

3.1.2 The Mapping of Processes onto Processors ... 33

3.1.2.1 Processes Space ... 33

3.1.2.2 Processor Space ... 34

3.2 Interprocess Communication and Synchronisation ... 37

3.3 Message Passing ... 38

3.3.1 General Issues ... 39

3.3.2 Synchronous and Asynchronous Point-to-Point Message ... 42

3.3.3 Rendezvous ... 44

3.4 Expressing and Controlling Nondeterminism ... 45

3.4.1 The Select Statement ... 46

3.4.2 Guarded Horn Clauses ... 48

CHAPTER 4 **LOAD BALANCING** ... 51

4.1 Proper Load Partitioning ... 51

4.2 Simulated Annealing ... 52

4.2.1 The Algorithm ... 58

4.3 Simulated Annealing and Load Balancing ... 60

CHAPTER 5 **IMPLEMENTATION** ... 65

5.1 The Initial Assignment Module ... 65

5.2 The Process Selection Module ... 66

5.3 The Processor Determination Module ... 66

5.4 The Destination Processor Module ... 66

5.5 The No.-Of-Hops Module ... 67

5.5.1 Mesh ... 67

5.5.2 Torus ... 68

5.5.3 Binary Tree ... 69

5.5.4 Pipeline ... 69

5.5.5 Hypercube ... 70

5.5.6 WK-recursive ... 70

5.6 The Hamiltonian Calculating Module ... 72

5.7 The Router Module ... 73

5.8 Experimental Results ... 73

5.9 Conclusion ... 81

BIBLIOGRAPHY ... 82

CHAPTER ONE

INTRODUCTION

"Nothing endures but change."
Hiraclitus

1.1 Aim Of The Project

"Now!Now!" cried the Queen, "Faster!Faster!"
Lewis Carrol

Load Balancing has remained a central issue since the advent of multiprocessing. The need for optimum utilisation of multiprocessors assumes far greater importance as the bottleneck for the conventional supercomputer speeds are reached and parallel machines are being offered by increasing number of vendors. It is only recently that the parallel computers are being procured not as an add on to a computing centre but for serious computations like a supercomputer. As this happens, the load balancing implementations to exploit the raw, scalable power of parallel machines assumes businesslike importance. The problem of load balancing is NP-complete. Thus the various approaches proposed, rightly seek to obtain satisfactory sub-optimal solution in a 'reasonable' time.

A number of different techniques have been adopted to solve the problem of load balancing. The techniques I have chosen for my dissertation is based on a method known as **Simulated Annealing**. My main aim is to do the satisfactory, acceptable partitioning for the classes of problems having

contiguous as well as irregular communication needed for a distributed memory multiprocessors.

1.2 Why Parallelism

"The most general definition of beauty...Multiety in unity."
Samuel Taylor Coleridge

There are many applications which are computationally intractable for "sequential" SISD machines as defined by FLYNN's taxonomy [10]. The speed of electric current flow along a conductor is one of the natures physical phenomena which ensures that such machines can never deliver the performance demanded by the seemingly insatiable user. In many areas such as engineering, science, energy resource, medicine, military and artificial intelligence, fast and efficient computers are in high demand. Largescale computations are often performed in these application areas requiring computers that can deliver a speed of billions and sometimes trillions of megaflops (MFLOPS). We, therefore, turn to parallelism as the means to satisfy this demand. The concept of parallelism is much older than most people realize [10] but with recent trends in VLSI fabrication technology giving rise to single package processors of impressive power and low cost, parallel machines which utilise large number of such devices are becoming common sparking much research into parallel computation. With rapid progress being made in the field of VLSI fabrication, it is clear that parallel hardware

can offer a dramatic reduction in cost per megaflops. The next few years, therefore, is expected to see parallel hardware becoming available on a more everyday basis. According to Sidney Fernbach"*Todays large computers (mainframes) would have been considered 'supercomputers' 10 to 20 years ago. By the same token, today's supercomputer will be considered 'state-of-the-art' standard equipment 10 to 20 years from now*" [11].

The potential benefits of a parallel approach to problems are not only the reduction in cost per megaflops, but also the fact that only with massively parallel systems will we be able to achieve a total computational throughput far in excess of that achievable using conventional vector supercomputers. Consider for example, one of the well known supercomputer problems in Computational Fluid Dynamics (CFD). In aircraft design, computer programs are required to calculate the airflow round an aircraft for a wide range of flow and fluid parameters. At present, rather than attempt a full solution of the complete Navier-Stokes equations, industry uses simplified flow codes incorporating significant simplifying approximations to these equations. These simplified codes, for example for steady-state flight, require many hours of supercomputer time for each parameter setting. Full solutions to the Navier-Stokes equations for a wide range of Reynolds numbers, and for all possible flight conditions are out of reach of present and forseeable vector supercomputers. Second generation supercomputers are now

achieving performances in the range 1 to 10 Gigaflops (thousands of megaflops) and must already employ several vector units working in parallel to achieve peak performance. Performance in the Terraflops range (thousands of Gigaflops) such as will be required to solve full CFD problems in the complex 3-dimensional geometries can only be achieved using massively parallel machines.

Besides such examples of massive parallelism, the next few years will also see parallel hardware becoming available on a more everyday basis. Powerful engineering workstations capable of present day supercomputer performance will exploit concurrency not only for raw compute performance but also to provide real-time 3-D graphic displays. The ability both to compute and to visualize solutions to complex systems of equation will soon become an indispensable tool of all scientists and engineers [11].

1.3 The Problem With Parallelism

> **"If seven maids with seven mops**
> **Swept it for half a year,**
> **Do you suppose,"the Walrus said,**
> **"That they could get it clear?"**
> *Lewis Carrol*

The efficiency with which we can exploit the potential parallelism in a given application relates intimately to the hardware, algorithm and programming language used. Unfortunately the greater the potential gains from parallelism, the more difficult it becomes to realise these

gains. For example, the larger the number of independent processing elements at our disposal the greater the communication overhead penalty incurred by the necessity to pass data between these.

An optimum hardware for a particular application would map readily to the problem in hand. That is to say, the number of processors running concurrently ought to equal the number of independent operations which could be concurrently performed. As a consequence of this some machines map readily to certain applications. However a static implementation can only be efficient for a small class of problems and some applications have no obviously suited machine counterpart.

A massively parallel machine is of little use if the algorithm selected is badly devised. If a load is imbalanced such that most processors must continually wait for a single, slow process to execute or should the algorithm not use the lowest possible computational order. As yet compilers are not available which can intelligently partition the system on users' behalf. Nonetheless, there are some machines which keep the parallelism largely transparent to the user and parallelism lost by the sequential nature of the user language is recaptured by clever compilers [10]. However, this is not always the case and with user visibility of parallelism comes the responsibility of optimising both hardware topology and software to make good use of parallelism possibilities within the application.

Parallel programs are much more difficult to debug than sequential programs - after a bug is supposedly fixed, it may be impossible to reconstruct the sequence of events that exposed the bug in the first place, so it would be inappropriate to certify, in some sense, that the bug has actually been corrected [7].

Another important point is that parallel programs are much more difficult to prove correct than sequential programs, and it is widely believed that *proving program correctness* must eventually displace exhaustive *program testing* if real strides are to be made in developing highly reliable, largescale software systems [7].

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

Finally, the language itself is important. Older languages are inherently unsuitable for expressing parallelism having been designed from the outset with sequential machine environments in mind.

1.4 Relevance Of The Project

In this project an attempt has been made at achieving the following objectives.

1) The module should have low processing overhead

2) There should be low message overhead

3) The processors should be kept busy to the maximum possible extent.

4) The computational load on the processors should be as even as possible.

1.5 Organisation Of This Report

Chapter 2 deals with the classification of parallel machines.

Chapter 3 discuses the different kinds of problems encountered in finding parallel solutions to problems.

Chapter 4 gives a brief introduction to load balancing and explains mainly the simulated annealing algorithm.

Chapter 5 is devoted to implementation. This chapter contains the various numerical results that were obtained when the simulation was run for various topologies.

CHAPTER TWO

CLASSIFICATION OF PARALLEL COMPUTERS

"A hair perhaps divides the False and the True"
Omar Khayyam

Parallel computers can be classified in many ways based on their structure or behaviour. The major classification methods consider the number of instructions and/or operand sets that can be processed simultaneously, the internal organisation of the processors, the interprocessor connection structure, or the methods used to control the flow of instructions and data through the system. Some of the more popular classification methods are :

1. **Flynn's Classification.**
2. **Structural Classification.**

2.1 Flynn's Classification

A typical central processing unit operates by fetching instructions and operands from main memory, executing the instructions, and placing the results in memory. The steps associated with the processing of an instruction form an instruction cycle. The instructions can be viewed as forming an *instruction stream* flowing from main memory to the processor, while the operands form another stream, the *data stream* flowing to and from the processor, as shown in fig 2.1.

Michael J. Flynn has made an informal and widely used classification of processor parallelism based on the number of simultaneous instruction and data streams seen by the

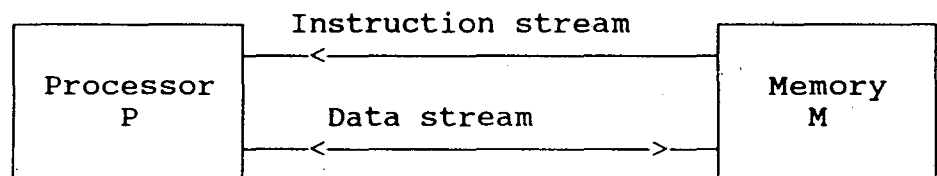


Fig. 2.1

1. Generate the next instruction address.
2. Fetch the instruction.
3. Decode the instruction.
4. Generate the operand addresses.
5. Fetch the operands.
6. Execute the instruction.
7. Store the result.

Fig. 2.2

processor during program execution. Suppose that a processor *P* is operating at its maximum capacity, so that its full degree of parallelism is being exhibited. Let m_i and m_d denote the minimum number of instruction and data streams, respectively, that are being processed in any of the seven steps associated with the execution of an instruction as shown in fig 2.2. m_i and m_d are termed the instruction- and data stream multiplicities of *P*, and measure its degree of parallelism. It is to be noted that m_i and m_d are defined by the minimum instead of the maximum number of streams flowing at any point, since the most limiting components of the system (bottlenecks) determine the overall parallel processing capabilities.

Computers can be roughly divided into four major groups based on the values of m_i and m_d associated with their CPUs.

1.*Single instruction stream single data stream (SISD):* $m_i=m_d=1$. Most conventional computers with one CPU containing a single arithmetic-logic unit capable only of scalar arithmetic fall into this category. SISD computers and sequential machines are thus synonymous.

2.*Single instruction stream multiple data stream (SIMD) :* $m_i=1$, $m_d>1$. This category includes machines that have a single program control unit and multiple execution units. ILLIAC IV and Distributed Array Processor are examples of this type of computers.

3.*Multiple instruction stream single data stream (MISD):* $m_i>1$, $m_d=1$. Not many computers fit into this category.

Computers like Cray-1 and CYBER-205, which rely heavily on pipeline processing, may be considered as MISD machines if the viewpoint is taken that a single data stream passes through a pipeline, and is processed by different (micro-) instruction streams in different segments of the pipeline.

4. *Multiple instruction stream multiple data stream (MIMD)* : $m_i>1$, $m_d>1$. This covers multiprocessors which are computers with more than one CPU and the ability to execute several programs simultaneously. Examples of multiprocessors are Cm* and NCUBE ten.

It is to be noted that the foregoing classification depends on a somewhat subjective distinction between control (instruction) and data. The term stream is equally vague and subject to varying interpretations. Hence it may not always be clear to which of the four Flynn classes a particular machine belongs. For example, whether to classify pipeline computers as MISD or MIMD hinges on the data and instruction streams; a case can also be made for calling these machines SIMD. Thus Flynn's classification is essentially *behavioral* and says nothing about a computer's structure. We, therefore, turn to some other ways of classifying parallel computers based on their overall structure or interconnection topology.

2.2 Structural Classification

A computer system can be viewed as a set of $n \geq 1$ processors (CPUs) P_1, P_2, ..P_n and $m \geq 0$ shared (main) memory units or modules M_1, M_2, M_3, ..M_m communicating via an

interconnection network N as shown in fig 2.3 . In a typical sequential computer, n=m=1, and N is a single shared bus over

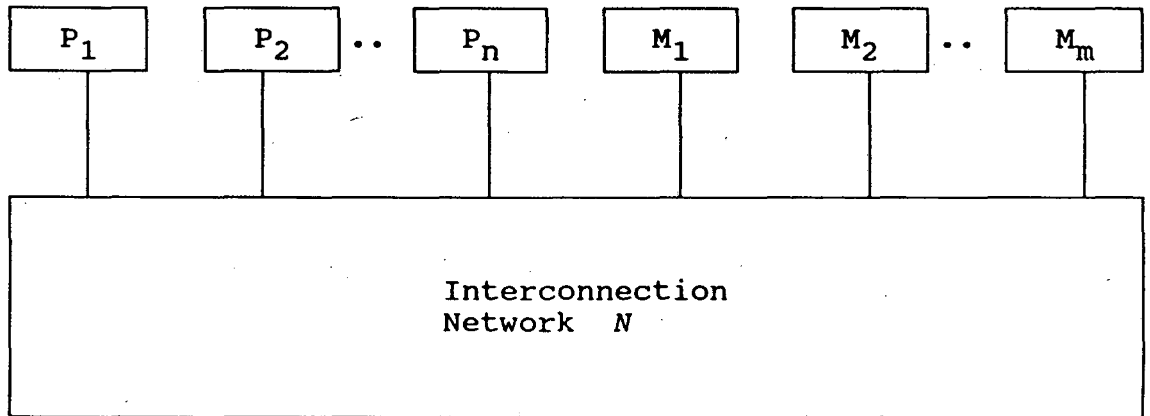


Fig. 2.3

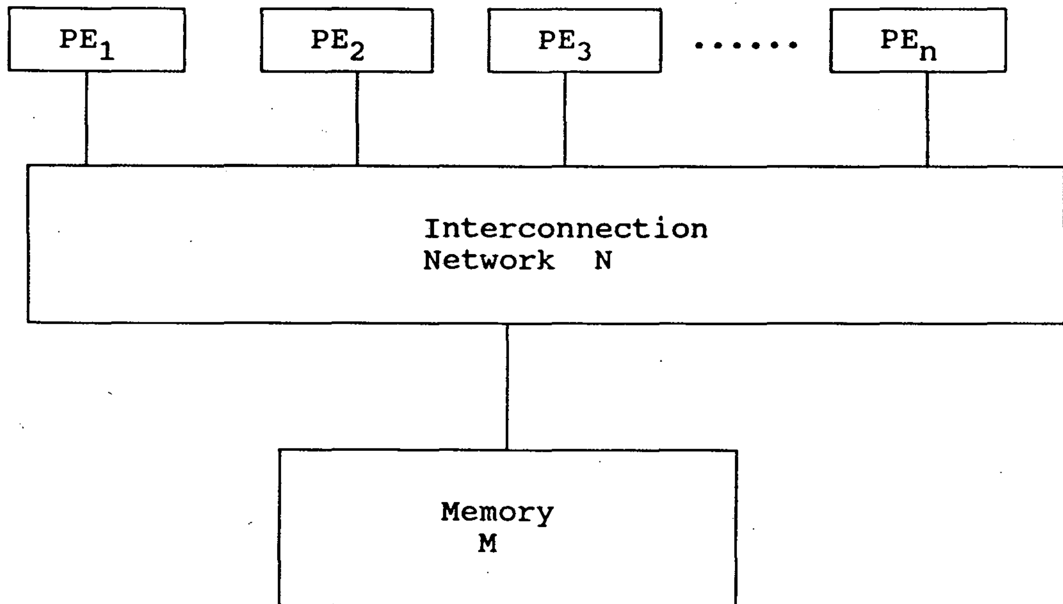


Fig 2.4

which all processor-memory communication takes place. In general, the memory units constitute a global main memory that provides a convenient message depository for processor-processor communication. A system with this organisation is called a *shared memory computer* (Fig. 2.4). A global memory can, however, be a major system bottleneck, particularly when processors must share large amounts of information since normally only one processor can access a given memory module at a time. If the processors are provided with their own local memories, then the global memory can be reduced in size or even eliminated completely. To separate the functions of processing (computation) and memory, a processor with no associated memory will be referred to as a processing element or *PE*. A processor is thus the combination of a PE and a local main memory; it may also include some external communications (IO) facilities forming, in effect, a small self-contained computer. In a system with little or no global memory, processing elements communicate via messages transmitted between their local memories as in the system of fig. 2.5 . In this case, the main memory is the sum of the local memories, and the system may be referred to as a *distributed memory computer.* The term *message passing computer* is also used for these machines. Figs 2.4 and 2.5 illustrate the main structural differences between shared-memory and distributed-memory systems.

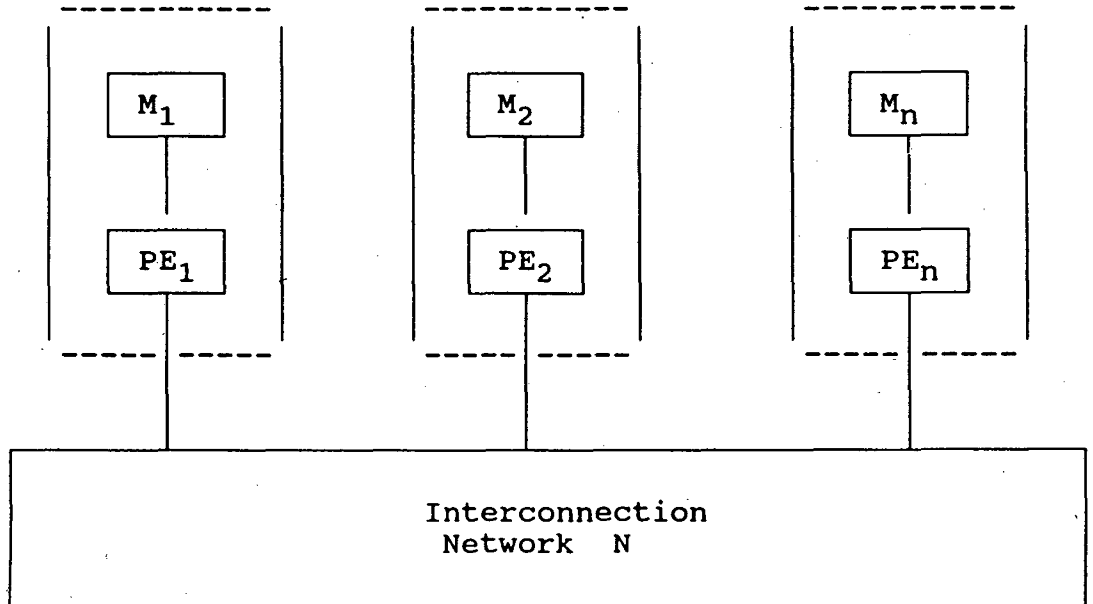


Fig. 2.5

The internal structure of the interconnection network N is also used to classify parallel computers. Figures 2.6 (a) to (c) show some of the popular topologies of the interconnection network. Because of the ease with which it can be designed and controlled, the *single shared bus* is widely used in parallel as well as sequential systems. When n, the number of **PE**s, and m, the number of main memory units, are large, extremely fast buses are required, and special design precautions must be taken to minimize contention for access to the bus. Bus contention can be relieved (but not necessarily eliminated completely) by providing multiple buses, forming the *multiple-bus* network depicted in Fig. 2.6(b). Each processor is connected to one or more of the available buses, each of which has all the attributes of an

independent system bus. Besides reducing the communication load per, a degree of fault tolerance is provided, since the system can be designed to continue operation, possibly with reduced performance if an individual bus fails. The *crossbar* interconnection network as shown in Fig. 2.6(c) is a special kind of multiple-bus system in which each PE has a (horizontal) bus linking it to all memories or, equivalently, each, memory has a (vertical) bus linking it to all PEs. An n x m crossbar allows upto MAX{n, m} bus transactions to take place simultaneously. However, in the worst case where all the processors attempt to access the same memory unit *Mi* simultaneously, the number of bus transactions drops to one. Although crossbar networks have been employed by a few computer systems, their hardware complexity quickly becomes prohibitive as m and n increase.

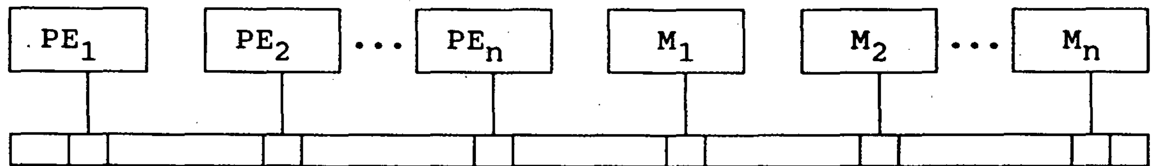


Fig 2.6 (a)

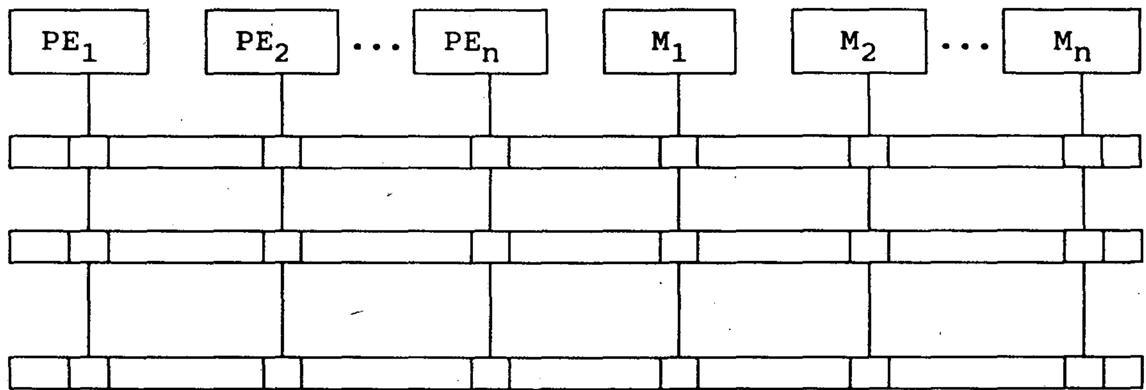


Fig. 2.6 (b)

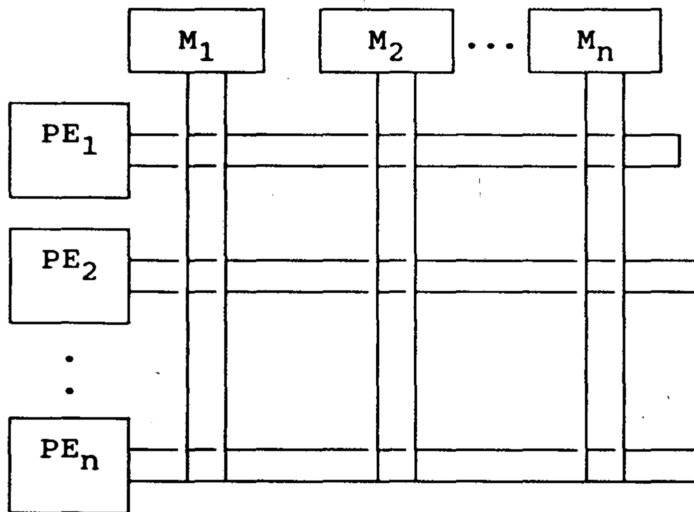


Fig. 2.6 (c)

Figures 2.7 and 2.8 illustrate various network topologies in which high speed dedicated connections are provided between each system component, which is typically an independent processor, and a small group of neighbouring components. The processors communicate among themselves through message passing. The computer structure depicted in fig. 2.7 is that of a mesh. Here the processors are arranged in the form of an m x n matrix along m rows and n columns. In this topology, the inside processors are connected to four neighbouring processors, the processors at the corners are connected to two other processors, whereas the remaining processors are connected to three other neighbouring processors.

The torus topology is shown in fig. 2.8. This topology is similar to the mesh, with the difference that the first and the last processors of each row are connected to each other. Similarly, the first and the last processors of each column have also been connected. This type of topology is very useful for transputer based systems. A transputer is basically a processor with some associated memory and four links through which it can be connected to four other transputers using a wire. In the torus topology all the four links of a transputer are utilised in connecting it to other processors thus reducing the number of links that are to be traversed when a message is sent from a source node to a destination node. For example, if a message is sent from PE_0 to PE_{15} then in the mesh topology, the minimum number of

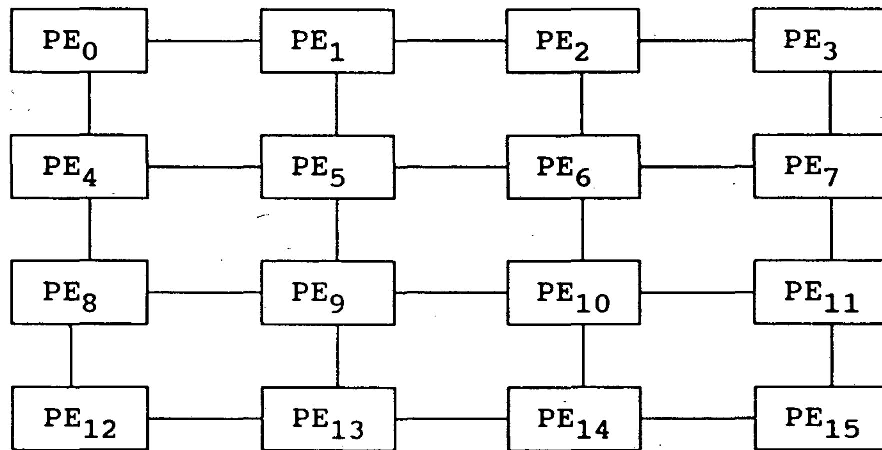


Fig. 2.7

The MESH TOPOLOGY

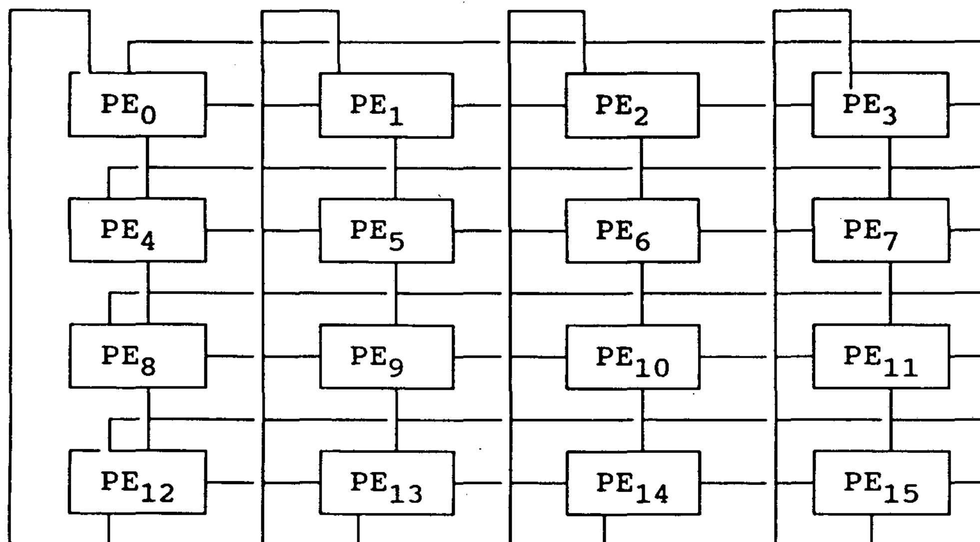


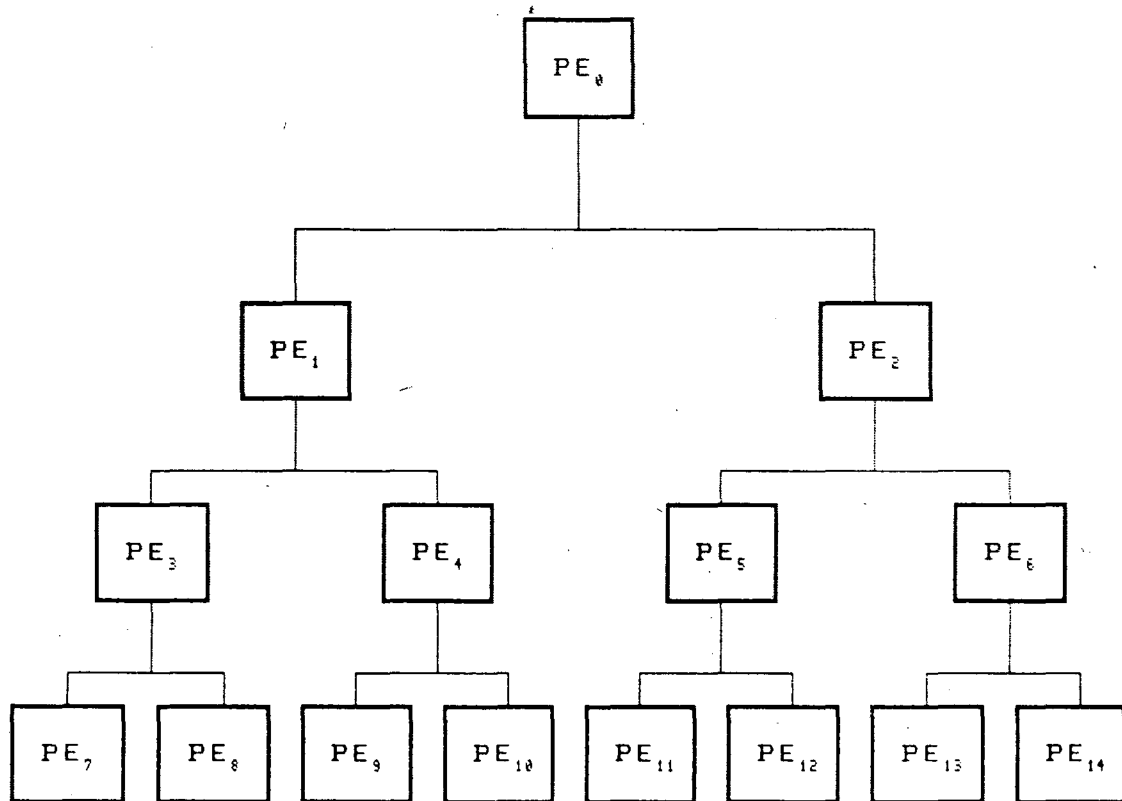
Fig. 2.8

The TORUS TOPOLOGY

links that are to be traversed is 6, whereas in the torus network this reduces to 2.

A binary tree structure (Fig. 2.9) can also be employed for connecting the processors. This topology is useful for the class of problems that fall under the category of *Divide and Conquer*. In this topology, the N processors act as the nodes of a binary tree with processor 0 being the root. The height of the binary tree is approximately $\log_2 N$. A processor **i** is connected to a processor **j** if **i** = 2***j** + 1 or **i** = 2***j** + 2. The main problem with this topology is that in case any link fails, the network is partitioned into two disjoint networks. If any one of the links connecting the root node to its children malfunctions, then the efficiency is reduced to half.

A very popular interconnection topology is the hypercube network (Fig. 2.10). A binary hypercube network of dimension d consists of 2^d nodes. A 2-dimensional hypercube is just a mesh of four processors. A 3-dimensional hypercube is made up of two 2-dimensional hypercubes by connecting the corresponding processors. A four dimensional hypercube is built by connecting the corresponding processors of two 3-dimensional hypercubes and so on. Here in order to address the 2^d elements d bits are required. The number of bits in which the binary addresses of two processors differ is termed as *Hamming Distance*. A processor **i** is connected to a processor **j** if and only if the hamming distance between **i** and **j** is 1.



A Binary Tree Interconnection

Fig. 2.9

PE_0 PE_1

PE_2 PE_3

PE_4 PE_5

PE_6 PE_7

PE_8 PE_9

PE_{10} PE_{11}

PE_{12} PE_{13}

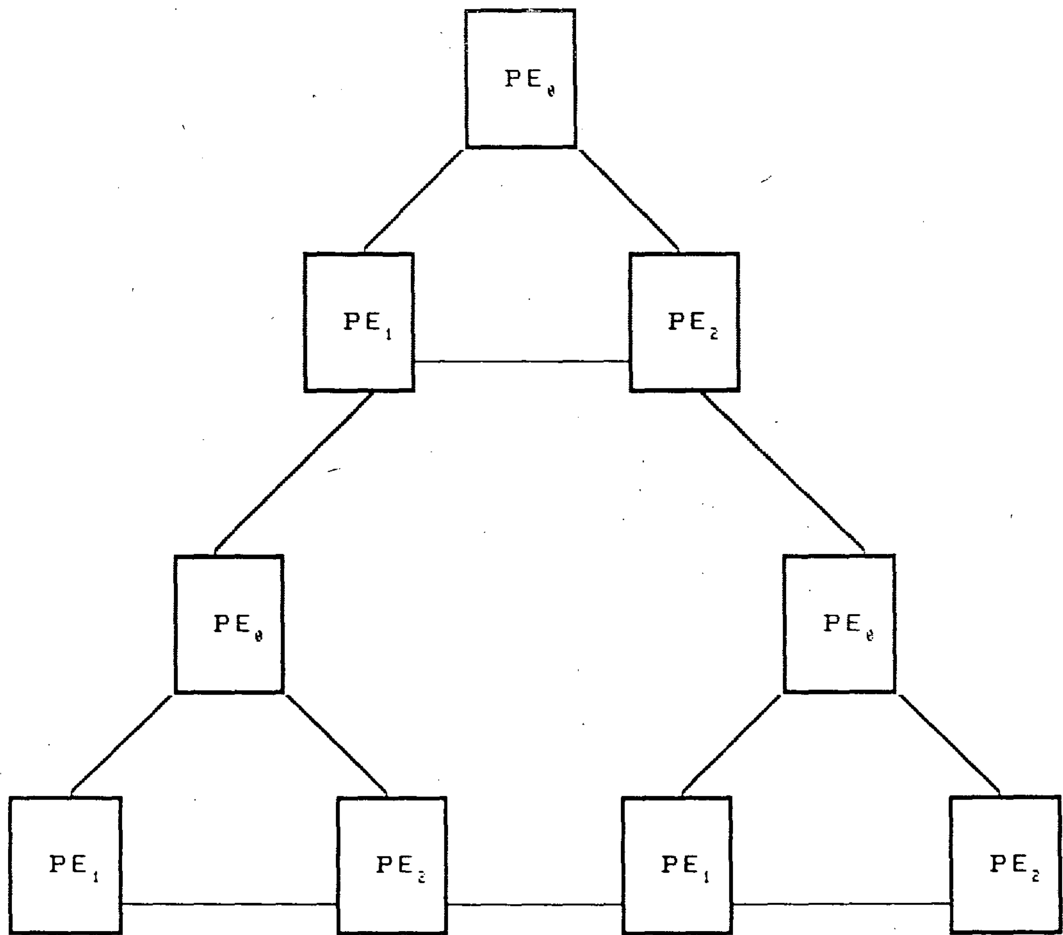
PE_{14} PE_{15}

A Hypercube Network of
16 Nodes

TH-5962

Another very useful interconnection scheme for transputer based systems is the WK-recursive class of topologies. These topologies have the advantage that they can be recursively scaled. These topologies offer a high degree of regularity and symmetry which very well confirm to modular design and implementation of distributed systems involving large number of computing elements. In addition, a network of arbitrarily large size can be built using transputers keeping the internode distances very small. At the same time, the network can be expanded as and when needed without much difficulty. Another advantage of this class of topologies is that networks built using this class of topologies admit self routing techniques for message exchanges based on an expansibility without reprogramming approach. Further such networks show a high degree of local density which allows subnetwork clusters to be outlined: this can suggest the utilization of suitable strategies for balancing the computational load.

Topology Description : If K is the node degree, using K+1 nodes a fully connected network can be built. Eliminating one node we obtain a configuration that remains fully connected still having K free links and which can be viewed as being virtually similar to each component node of degree K. Therefore, this structure constitutes what is called a virtual node and acts as a building block for building up additional configuration. Example of a virtual node obtained for K = 3 is shown in fig. 2.11 . In particular, a fully



A WK-Recursive Topology
of 9 Nodes

Fig. 2.11

connected configuration composed of K virtual nodes (i.e. K*K real nodes) again offers K free links and reproduces, at a higher abstraction level, the virtual node structure. This new structure which is called *second level virtual node* can in turn be used to build a third level virtual node by completely connecting K second level virtual nodes. The amplitude W of the l-th level virtual node as the number of its (l-1)th level virtual nodes, having of course W=K. By recursively applying this technique, it is possible to define a class of regular, scalable topologies organised according to expansion levels obtained as recursive replications of a basic fully connected structure (i.e. the first level virtual node). Because of the recursive nature of these topologies whose peculiar aspect is the equality between the amplitude and the degree of virtual nodes, these class of topologies are termed as *WK-recursive*.

A member of the class of WK-recursive topologies is identified by three parameters :

N = number of real nodes

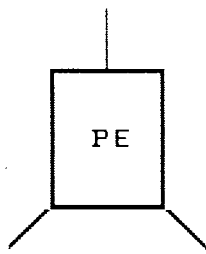
K = node degree

L = expansion level

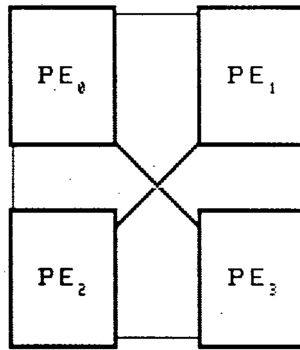
for which the following analytical relation holds :

$$L = \log_K N \qquad (2....)$$

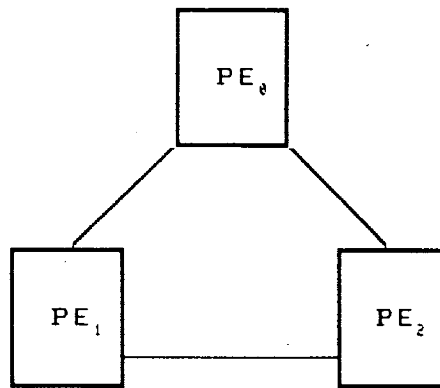
Fig. 2.12 illustrates some examples of topologies for various values of N, K and L. The special cases of K = 2 and K = N lead to the linear array and fully connected configuration,



(a)

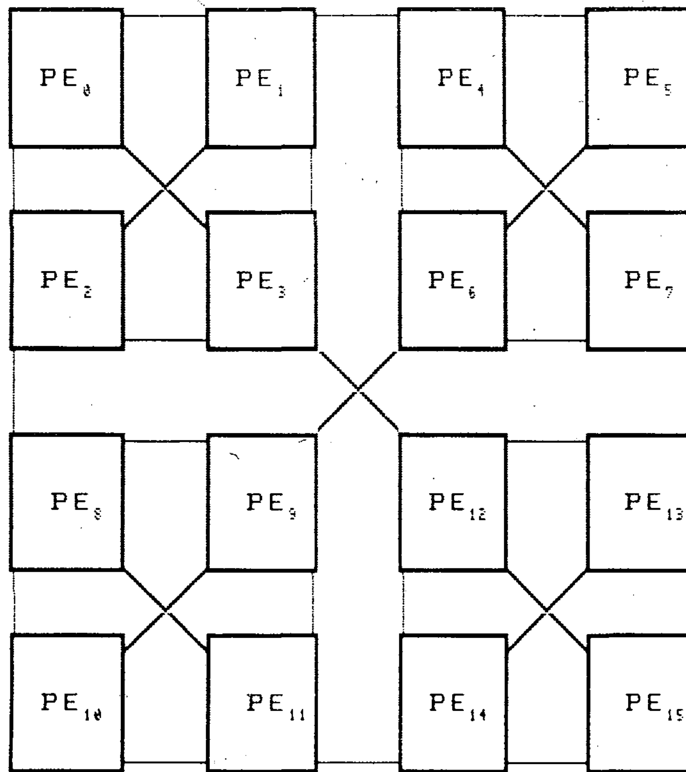


(b)



(c)

Fig. 2.12



A WK-RECURSIVE NETWORK OF
16 NODE

Fig. 2.13

respectively. The equation (2....) permits to simply define indices for characterising topologies belonging to the WK-recursive class. For instance, the maximum distance between any pair of nodes expressed as the number of routing steps required to forward a message along the shortest path connecting the nodes is given by

$$D = 2^L - 1$$

Clearly, the distance depends only on the expansion level whatever the node degree is.

Computers can be further distinguished on the basis of the unit-to-unit connection paths provided by their interconnection networks. These paths may be *static* i.e. fixed and unchangeable or *dynamic*, i.e., reconfigurable under system control. The single-bus, multiple bus, and crossbar interconnections are examples of dynamic interconnection structures, whereas tree and hypercubes are static. The conventional single system bus (Fig. 2.3) is designed to allow any of the n processors to connect to any of the m memories for one or more bus cycles, e.g., to fetch an instruction. In a subsequent cycle some other processor-memory pair may use the bus to communicate. Thus the units communicating over the bus vary dynamically. In contrast, each processor in the binary tree or hypercube configuration (Figs. 2.8 & 2.10) has dedicated buses to its nearest neighbours, and can only communicate with other processors indirectly.

CHAPTER THREE

SOFTWARE ISSUES IN PARALLEL COMPUTING

"To conquer without risk is to triumph without glory."
Pierre Corneille

The era of data parallelism took a new turn with the evolution of MIMD computers. Now the processors are not supposed to work independently as earlier; rather they are meant to work on the same subdivided task where the intermediate or end results available on one processor may be a parameter for another processor to resume execution. The high sounded words like fault tolerance and practical parallelism became a reality. However, it also posed several new problems, like interprocess communication, synchronisation and nondeterminism to name a few of them. The exploitation of the grain of parallelism also became challenging task.

3.1. The Mapping Problem

"Lead me from the unreal to the real."
The Upanishads

The speedup of the parallel execution of a problem (over the sequential algorithm) depends upon two factors, apart from the number of processors working together. These are :

1. Decomposition of the problem.
2. Mapping of the decomposed problem (processes space) onto the (processor space) target system.

3.1.1 Decomposition Of The Problem

In any parallel processing application three levels of parallelism exist. These are

i) Processor Farm Parallelism

ii) Geometric Parallelism

iii) Algorithmic Parallelism

3.1.1.1 Processor Farm Parallelism

Many scientific problems require repeated execution of the same program, with different initial data (random number seeds, for example) . Later runs of the program do not require any knowledge of previous runs, so many runs could be done simultaneously. On most computers, this option is not available, resulting, typically, in the submission of many different jobs consisting of the same program but accessing different data, or running with different parameters. By contrast this type of application can be run very efficiently on a multiprocessor machine. Little or no communication is required between processors, except that, after execution, the results from each of the processors need to be collated and, perhaps, some kind of statistical analysis performed.

A similar situation occurs when a 'controller' issues work-packets to a network of processors, without caring which processor accepts it. The only real difference is one of scales. This *farm* structure will automatically balance the load among the workers, because a worker which accepts a *difficult* packet will not accept another until it has

finished, while a worker which has an *easy* packet can take another relatively soon.

Typical architectures for these types of application are thus **farms** of processors reporting back to, and receiving instructions from, a controller. The work can be distributed down a linear chain (fig. 3.1) with a simple control structure, or on a ternary tree (fig. 3.2) with a more complex control structure but faster broadcasts. Each processor runs the same program (with data dependent branches) and has a complete, but different, set of data from its workpacket. Large amount of storage may, therefore, be required on each element. Because of the limited communication requirements, this method can be efficient, but because of memory requirements it is not necessarily cost-effective.

3.1.1.2 Geometric Parallelism

Many physical problems have an underlying regular geometrical structure, with spatially limited interactions (e. g. problems in field theory or hydrodynamics). This homogeneity allows the data to be distributed uniformly across the processor array, with each processor being responsible for a defined spatial area. This is illustrated in fig. 3.3.

A processor communicates with neighbouring processors and the communication load will be proportional to the size of the boundary of the subdomain, while the computational load

Host — Worker — Worker .... Worker

Farming Network: Linear Chain

Fig. 3.1

Host — Worker — Worker / Worker / Worker

Farming Network : Ternary Tree

Fig. 3.2

Geometric Parallelism
12 x 12 data array distributed
over 2 x 2 processor array

Fig. 3.3

will be proportional to the volume of the boundary of the subdomain. This type of parallelism is sometimes referred to as **domain decomposition** or **data parallelism** and this type is specially suitable for transputer arrays.

3.1.1.3 Algorithmic Parallelism

This is a more fine-grained parallelism in which features of the algorithm that are capable of concurrent operation are identified and each processor executes a small part of the total algorithm. Clearly, the resulting structure will be specific to the particular algorithm used in the application. This type of parallelism can be expressed naturally in a language like Occam on transputer networks.

A common feature of this approach is the construction of a number of **pipes** of processors, similar to those found in pipelined vector supercomputers. Here, however, the pipes may be more general and capable of splitting and merging in much more flexible way and operate at a different level of granularity.

In such a decomposition of the problem, the data now flows between the processing elements, and is sometimes referred to as **Data Flow** parallelism (not to be confused with the machine of the same name). The communication load on each processor is severely increased in this scheme. Indeed, without care, communication bandwidth problems can become dominant and severely degrade the performance. In addition, an elaborate communication and control structure is needed. An advantage of this type of decomposition, however, is that

little data space is required per processor. It has been found that this type of problem decomposition gives efficiencies of the order of 50% without much effort but detailed analysis and load balancing can improve the performance.

3.1.2 The Mapping Of Processes Onto Processors

3.1.2.1 Processes Space

In most procedural languages for parallel and distributed programming, parallelism is based on the notion of a *process*. A process is a logical processor that executes code sequentially and has its own state and data. Processes are declared, just like procedures.

Processes are created either implicitly by their declaration or by some **create** construct. With implicit creation, one usually first declares a process type and then creates processes by declaring variables of that type. Often *arrays* of processes may be declared. In some languages based on implicit process creation, the total number of processes is fixed at compile time. This makes the efficient mapping of processes onto physical processors easier, but it imposes a restriction on the kinds of applications that can be implemented in the language, since it requires that the number of processes be known in advance.

Having an explicit construct for creating processes allows more flexibility than implicit process creation. For example, the creation construct may allow parameters to be passed to the newly created process. These are typically used

for setting up communication channels between processes. If processes do not take parameters, as in Ada [6], the parameters have to be passed to a newly created process using explicit communication. A mechanism is needed to set up the communication channel over which the parameters are sent.

Another important issue is the *termination* of processes. Processes usually terminate themselves, but some primitive may be provided to abort other processes too. Some precaution may be needed to prevent processes from trying to communicate with a terminated process.

The problem space consists of the geometric domain in general which is populated with the process elements like grid points etc. as mentioned earlier. The distribution of the process elements in the geometric or data domain of the problem can be static or can evolve during the execution. If it is static the generally static load balancing would suffice. Even if the distribution is dynamic, we don't require to do dynamic load balancing upto 1K processors. The number of process creations and synchronisations should be minimized. Since process synchronisation is also expensive, the grain size should be made as large as possible, while keeping all the processors busy.

3.1.2.2 Processor Space

The processor space consists of the set of all processes which are physically interconnected. The goal is to connect the processors such that the degree of connectivity is high so as to reduce the communication overheads. The

distribution of computations over the available physical processors is also an important issue. This assignment of computations to processors is termed as *mapping*.

Mapping strategies vary depending upon the application to be implemented. The assignment of processes to processors will be quite different in a computation whose objective is to obtain maximum speedup through parallelism, and an application whose objective is to obtain high availability through replication, for example.

When the goal is to speedup computation time through parallelism, the mapping of processes to processors is similar to load balancing in distributed operating systems : both attempt to maximize parallelism through efficient use of available computing power. But there are important differences. An operating system tries to distribute the available processing power *fairly* over competing processes from different programs and different users. It may try to reduce communication costs by letting processes that communicate frequently run in pseudo-parallel on the same processor. The goal of mapping, however, is to minimize the execution time of a single distributed program. As all parallel units are parts of the same program, the they are cooperating rather than competing, so fairness need not be an issue. In addition, reduction of communication overhead achieved through mapping processes to the same processor must be weighed against the resulting loss of parallelism [3]. If the goal of application is to increase fault-tolerance, an

entirely different mapping strategy may be taken. Processes may be replicated to increase availability. The mapping strategy should at least assign the replicas of the same logical process to different physical processors.

There are three approaches for assigning parallel units to processors, whether the assignment is done by the programmer or the system : the processors can be fixed at compile-time, fixed at run-time or not fixed at all. The first method is least flexible, but has the distinct advantage that it is known at compile-time which parallel units will run on the same processor, allowing the programmer to take advantage of the fact that these processes will have shared memory available. With run-time approach to mapping computations to processors, a parallel unit is assigned to a processor when that unit is created. An example is the Turtle notation designed by Shapiro for executing concurrent PROLOG programs on an infinite grid of processors, where each processor can communicate with its four neighbouring processors [3]. The third approach to processor allocation, allowing a process to execute on different processors during its lifetime, is used by only a very few languages. Emerald, for example, is an object-based language that allows objects to migrate from one processor to another. The language has primitive to determine the current location of an object, to fix or unfix an object on a specific processor, and to move an object to a different processor.

3.2 Interprocess Communication And Synchronisation

> **"Tell me to whom you are addressing yourself when you say that. I am addressing myself - I am addressing myself to my cap. "**
>
> *Jean Baptist Moliere*

An important issue which must be addressed in the design of a language for parallel programming is how the pieces of a program which are running in parallel on different processors are going to cooperate. This cooperation involves two types of interaction : **communication** and **synchronisation.** For example, Process A may require some data **X** which is the result of some computation performed by Process **B**. There must be some way of getting **X** from **B** to **A**. In addition if process **A** comes to the point in its execution which requires information **X** from process **B**, but process **B** has not yet communicated the information to **A** for whatever reason, **A** must be able to wait for it.

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

from a producer process to store an item in the buffer whenever the buffer is not full; it may accept a request from a consumer process to add an item whenever the buffer is not empty. To program such behaviour, a notation is needed to express and control nondeterminism.

Expression of interprocess communication (IPC) falls into two general categories - shared data and message passing - although this categorization is not always clear-cut. Parallel logic languages that provide shared logical variables, for example, are frequently used for programming in a message passing style. It is to be noted that the *model* provided by the language for expressing IPC and the *implementation* of that model may be two entirely different things. I shall restrict my explanation to systems without shared memory since multiprocessor systems are mostly without shared variable.

3.3 Message Passing

"There is nothing more requisite in business than dispatch. "

Joseph Addison

In multiprocessor systems without any shared memory, communication among the processors is mainly through **message** passing. While sending a message, many factors come into play : who sends it, what is sent, to whom is it sent, is it guaranteed to have arrived at the remote host, is it guaranteed to have been accepted by the remote process, is

there a reply (or several replies), and what happens if something goes wrong. There are also many other considerations involved in the receipt of a message : for which process or processes on the host, if any, is the message intended; is a process to be created to handle this message; if the message is intended for an existing process, what happens if the process is busy- is the message queued or discarded; and if a receiving process has more than one outstanding message waiting to be serviced, can it choose the order in which it services messages-be it **FIFO**, by sender, by some message type or identifier, by the contents of the message, or, according to the receiving process's internal state.

3.3.1 General Issues

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

Most (but not all) message-based interactions involve two parties, one *sender* and one *receiver*. The sender initiates the action *explicitly*, for example, by sending a message or invoking a remote procedure. On the other hand, the receipt of a message may either be *explicit* or *implicit*. With explicit receipt, the receiver is executing some sort of

accept statement specifying which messages to accept and what actions to undertake when a message arrives. With implicit receipt, code is automatically invoked within the receiver. It usually creates new thread of control within the receiving process. Whether the message is received implicitly or explicitly is transparent to the sender.

Explicit message receipt gives the receiver more control over the acceptance of messages. The receiver can be in many different *states*, and accept different types of messages in each state. More accurate control is possible if the accept statement allows messages to be accepted conditionally, depending on the arguments of the message as in Concurrent C. A file server, for example, may want to accept a request to open a file only if the file is not locked. In Concurrent C this can be coded as

```
accept open(f) such that not_locked(f)  {
        . . . . .
        process open request
        . . . . .
}
```

Some languages give the programmer control over the order of message acceptance. Usually messages are accepted in FIFO order, but occasionally it is useful to change this order according to the type, sender, or contents of a message. For example the file server may want to handle read requests for small amounts of data first :

```
accept read(f, offset, nr_bytes) by nr_bytes {
        . . . .
        process read request
        . . . .
}
```

The value given in the **by** expression determines the order of acceptance. If conditional or ordered acceptance is not supported by the language, an application needing these features will have to keep track of requests that have been accepted but not handled yet.

Another major issue in message passing is *naming* (or addressing) of the parties involved in an interaction: to whom does the sender wish to send its message, and, conversely, from whom does the receiver wish to accept a message ? These parties may be named *directly* or *indirectly*. Direct naming is used to denote one specific process. The name can be static name of the process or an expression evaluated at run time. A communication scheme based on direct naming is *symmetric* if both the sender and the receiver name each other. In an *asymmetric* scheme only the sender names the receiver. In this case, the receiver is willing to interact with any sender. It is to be noted that interactions using implicit receipt of messages are always asymmetric with respect to naming. Direct naming schemes, specially the symmetric ones, leave little room for expressing nondeterministic behaviour. Languages using these schemes, therefore, have a separate mechanism for dealing with nondeterminism.

Indirect naming involves an intermediate object, usually called a mailbox, to which the sender directs its message and to which the receiver listens. In its simplest form the mailbox is just a global name. More advanced schemes treat mailboxes as values that can be passed around, for example, as part of the message. This option allows highly flexible communication pattern to be expressed. Mailing a letter to a post office box rather than a street address illustrates the difference between indirect and direct naming. A letter sent to a post office box can be collected by anyone who has the key to the box. People can be given access to the box by duplicating keys or by transferring existing keys (possibly through another P. O. box). A street address, on the other hand does not have this flexibility [3].

3.3.2 Synchronous and Asynchronous Point-to-Point Message

The major design issue for a point-to-point message passing system is the choice between *synchronous* and *asynchronous* message passing. With synchronous message passing, the sender is blocked until the receiver has accepted the message (explicitly or implicitly). Thus, the sender and the receiver not only exchange data, but they also synchronize. With asynchronous message passing, the sender does not wait for the receiver to be ready to accept its message. Conceptually, the sender continues immediately after sending the message. The implementation of the language may suspend the sender until the message has at least been copied

for transmission, but this delay is not reflected in the semantics.

In the asynchronous model there are some semantic difficulties to be dealt with. As the sender **S** does not wait for the receiver **R** to be ready, there may be several *pending* message sent by **S**, but not yet accepted by **R**. If the message is *order preserving*, **R** will receive the messages in the order they were sent by **S**. The pending messages are *buffered* by the language run time system or the operating system. The problem of a possible buffer overflow can be dealt with in one of two ways. Message transfers can simply fail whenever there is no more buffer space. Unfortunately, this makes message passing less reliable. The second option is to use *flow control*, which means the sender is blocked until the receiver accepts some messages. This introduces a synchronization between the sender and the receiver and may result in unexpected deadlocks.

In the synchronous model, however, there can be only one pending message from any process **S** to a process **R**. Usually, no ordering relation is assumed between messages sent by different processes. Buffering problems are less severe in the synchronous model, as a receiver need buffer at most one message from each sender, and additional flow control will not change the semantics of the primitive. On the other hand, the synchronous model also has its disadvantages. Most notably, synchronous message passing is less flexible than asynchronous message passing, because the sender always has to wait for the receiver to accept the

message, even if the receiver does not have to return an answer [3].

3.3.3 Rendezvous

A point-to-point message establishes one way communication between two processes. Many interactions between processes, however, are essentially two way in nature. For example, in the client/server model the client requests a service from a server and then waits for the result returned by the server. This behaviour can be simulated using two point-to-point messages, but a single higher level construct is easier to use and more efficient to implement. *Rendezvous* is one such construct.

The rendezvous model is based on three concepts : the entry declaration, the entry call, and the accept statement. The entry declaration and accept statement are part of the server code, while the entry code is on the client side. An entry declaration looks like a procedure declaration. An entry has a name and zero or more formal parameters associated with it. An entry call is similar to a procedure call statement. It names the entry and the process containing the entry and it supplies the actual parameters. An accept statement for the entry may contain a list of statements, to be executed when the entry is called, as has been illustrated in the following accept statement for entry incr :

```
accept incr (x:integer; y: out integer) do
        y = x + 1;
end;
```

An interaction (called a *rendezvous*) between two processes **S** and **R** takes place when **S** calls an entry for **R**, and **R** executes an **accept** statement for that entry. The interaction is fully synchronous, so the first process that is ready to interact waits for the other. When the two processes are synchronised, **R** executes the **do** part of the accept statement. While executing these statements, **R** has access to the input parameters of the entry, supplied by **S**. **R** can assign values to the output parameters which are passed back to **S**. After **R** has executed the **do** statements, **S** and **R** continue their execution in parallel. **R** may still continue working on the request of **S**, although **S** is no longer blocked.

3.4 Expressing And Controlling Nondeterminism

"A person with one watch knows what time it is ; a person with two watches is never sure. "

Proverb

As stated before, the interaction pattern between processes are not always deterministic, but sometimes depend on the runtime conditions. For this reason, it is necessary to introduce models for expressing and controlling nondeterminism. As explained earlier, some communication primitives are nondeterministic. A message received indirectly through a port may have been sent by any process. Such primitives provide a way to express nondeterminism, but not to control it. Most programming languages use a separate

construct to control nondeterminism. Two such constructs are : the *select statement*, used by many algorithmic languages and the *guarded Horn clause*, used in most parallel logic programming languages. Both are based on the *guarded command statement*, introduced by Dijkstra as a sequential control structure.

3.4.1 The Select Statement

A select statement consists of a list of guarded commands whose format is as follows :

guard -> statements

The guard consists of a boolean expression and some sort of "communication request. " The boolean expression must be free of side effects, as it may be evaluated more than once during the course of the select statement's execution. In Hoare's Communicating Sequential Processes, a guard may contain an explicit receipt of a message from a specific process P. Such a request may either *succeed* (if P has sent such a message), *fail* (if P has already terminated) or *suspend* (if P is still alive but has not sent the message yet). The guard itself can either succeed, fail or suspend : the guard succeeds if the expression is "true" and the request succeeds; the guard fails if the boolean expression evaluates to "false" or if the communication request fails; or the guard suspends if the expression is true and the request suspends. The select statement as a whole blocks until either all of its guards fail or some guards succeed. In the first case, the entire select statement fails and has no effect. In the latter case,

one succeeding guard is selected nondeterministically and the corresponding statement part is executed.

Select statements can also be used for controlling nondeterminism other than communication. Some languages allow a guard to contain a *timeout* instead of a communication request. A guard containing a timeout of T seconds succeeds if no other guard succeeds within T seconds. This mechanism sets a time limit on a process that wants to wait for a message. Another use of select statement is in the control of *termination* of processes. In Concurrent C, a guard may consist of the keyword **terminate**. A process that executes a select statement containing a **terminate** guard is willing to terminate if all other guards fail or suspend. If all processes are willing to terminate, the entire Concurrent C program terminates. Roughly, if all children processes created by a parent process are willing to terminate and the parent process has completed the execution of its statements, all these processes are terminated. This mechanism assumes hierarchical processes.

An important point to be noted in connection with the select statements in most programming languages is that they are *unfair*. In Communicating Sequential Processes (CSP) introduced by Hoare , for example, if several guards are successful, one of them is selected nondeterministically. No assumption can be made about which guard is selected. Repeated execution of the select statement may select the same guard over and over again, even if there are other successful guards. An *implementation* may introduce a degree

of fairness, by assuring that a successful guard will be selected within a finite number of iterations, or by giving guards equal chances. On the other hand, an implementation may evaluate the guards sequentially and always choose the first one yielding "true". The semantics of select statements donot guarantee any degree of fairness, so the programmers cannot rely on it.

Various proposals have been made for giving programmers explicit control over the selection of succeeding guards. Silberchatz has suggested a partial ordering of the guards. Elrad and Maymir-Ducharme have proposed prefixing of every guarded command with a compile-time constant called the *preference control value*. If several guards succeed, the one with the highest preference control value (i. e. priority) is chosen. If there are several guards with this value, one of them is chosen nondeterministically. This feature is useful if some requests are more urgent than others. For example, the buffer process may wish to give consumers a higher priority than producers.

3.4.2 Guarded Horn Clauses

Logic programs are inherently nondeterministic. In reducing a goal of logic program, there are often several clauses to choose from. Intuitively, the semantics of logic programming prescribe that the underlying execution machinery must simply choose the "right" clause, the one leading to a proof. This behaviour is called *don't know nondeterminism*. In sequential logic languages, these semantics are implemented

using *backtracking*. At each choice point an arbitrary clause is chosen, and if it later turns out to be a wrong one, the system resets itself to the state before the choice point and then tries another clause.

In a parallel execution model, several goals may be tried simultaneously. In this model, backtracking is very complicated to implement. If a binding for a variable has to be undone, all processes that have used this binding must backtrack too. Most parallel logic programming languages, therefore, avoid backtracking altogether. Rather than trying the clauses for a given predicate one by one and backtracking on failure, parallel logic languages

i) search all these clauses in parallel and

ii) do not allow any bindings made during these parallel executions to be visible to the outside until one of the parallel executions is committed to.

This is called the **OR**-parallelism. Unfortunately, this cannot go on for long, since the number of search paths that can be worked on in parallel grows exponentially with the length of the proof.

A popular technique to control **OR** parallelism is *committed choice nondeterminism*, which nondeterministically selects one alternative clause and discards the others. It is based on guarded Horn clauses of the form :

$A :- G_1, G_2, \ldots . G_n \mid B_1, B_2, \ldots . B_m \quad (n \geq 0, m \geq 0)$

The conjunction of the goals G_i is called the guard; the

conjunction of the goals B_i is the body. Declaratively, the *commit operator* "|" is also a conjunction operator.

Just like the guards of a select statement, the guard of a guarded Horn clause can either succeed, fail or suspend. A guard suspends if it tries to bind a variable that it is not allowed to bind. If a goal with a predicate A is to be reduced, the guards of all clauses for A are tried in parallel, until some guards succeed. The reduction process then selects one of these guards nondeterministically and commits to its clause. It aborts execution of other guards and executes the body of the chosen clause.

Till now this seems to be same as the select statement, but there are some subtle differences. Guards that are aborted should have no side effects at all. Precautions must be taken against guards that try to bind variables in their environment.

CHAPTER FOUR

LOAD BALANCING

Load balancing is an essential module of a parallel operating system. This module is responsible for allocating modules transparently to one or more processors for execution. Load balancing is vital if dynamic module creation is to be provided in such a way as to maximize the use of available hardware. With this module a program need not be rewritten when the hardware topology is changed, provided there is sufficient parallelism to take advantage of the new hardware. The requirement of load balancing is to distribute the computational and communication loads in such a manner that the members communicating with each other stay as close to each other as possible and the computation is spread over all the processors of the parallel computing system evenly.

4.1 Proper Load partitioning

> **"This shows how much easier it is to be critical than to be correct."**
>
> *Benjamin Disraeli*

Efficiency of the parallel computing system with N nodes (processing elements) is given by

$$E = S / N$$

where the parallel machine runs S times faster than a sequential machine using an optimal sequential algorithm. This efficiency is reduced by

i) Unequal distribution of computation.

ii) communication overhead.

The job of allocating task to CPUs in a parallel Computing System in such a way that both the above conditions are met is called Load Balancing. This job is essentially matching the process space on to the processor space as best as possible.

4.2 Simulated Annealing

"Man is a tool making animal."
Benjamin Franklin

Simulated Annealing [12] is a powerful and general algorithm for solving optimization problems in which the problem to be optimized can be represented as a function that has many variables and many local minima. Because many real-world design problem can be cast in the form of such optimization problems, there is intense interest in general techniques for their solution. Simulated annealing is one such technique of rather recent vintage (it was introduced in 1982 by Kirpatrick, Gelatt and Vecchi) with an unusual pedigree : it is motivated by an analogy to the statistical mechanics of annealing in solids. To understand why such a physics problem is of interest, consider how to coerce a solid into a low energy state. A low energy state usually means a highly ordered state, such as a crystal lattice; a relevant example here is the need to grow silicon in the form of highly ordered, defect-free crystals for use in semiconductor manufacturing. To accomplish this, the material

is *annealed*: heated to a temperature that permits many atomic rearrangements, then cooled carefully, slowly until the material freezes into a good crystal. Simulated annealing techniques use an analogous set of *controlled cooling* operations for nonphysical optimization problems, in effect transforming a poor, unordered solution into a highly optimized, desirable solution. Thus simulated annealing offers an appealing physical analogy for the solution of optimization problems; and more importantly, the potential to reshape mathematical insights from the domain of physics into insights for real optimization problems.

Interest in such solution techniques is intense because few important combinatorial optimization problems can be solved exactly in a reasonable time. Many optimization problems arising in practice are **NP**-*complete* : i.e. all known techniques for obtaining an exact solution require a time that is exponentially distributed with respect to the size of the problem. Hence emphasis has been directed towards heuristic techniques for obtaining a solution to these optimization problems. The difference between an algorithm and a heuristic is that a heuristic is not guaranteed to give the optimum solution. Rather a heuristic is designed to provide an acceptable answer for the class of **NP**-*complete* problems. In practice, however, the terms *algorithm* and *heuristic* are often used interchangeably. Moreover, simulated annealing is not an algorithm in the sense that it prescribes a mechanical sequence of computations for solving a specific problem. Annealing is a *strategy* or *style* for solving

combinatorial optimization problems. Specifically simulated annealing is a heuristic solution strategy applicable to a wide variety of optimization problems. It gives excellent results but is very slow. Hence it is used to approximate the global minimum as closely as computational resources permit. For problems that are not well understood, it may not be possible to find an algorithm which can take advantage of problem-specific properties. For such problems, Simulated Annealing has been found to be extremely successful, even with long computation time. Simulated Annealing has been widely used to solve problems like VLSI placement and Load Balancing.

Heuristic strategies for solving optimization problems that attempt to find a minima of any function $f(X_1, X_2, \ldots X_N)$ for the N parameters $X_1, X_2, \ldots X_N$ come in several styles. Sometimes *constructive* heuristics can be found, which build up a good answer directly, piece by piece. Of more interest are *iterative improvement* strategies that attempt to perturb some existing, suboptimal solution in the direction of a better, lower-cost solution. The idea can be neatly explained with the help of a *ball and hills* graph as shown in fig. 4.1. The set of all values of the objective function taken over all legal configurations is termed *cost surface.* In this graph, the value of the objective function has been plotted for a single parameter i.e. N = 1, as a set of hills and valleys in the cost surface. The ball represents the current configuration that is planned to be perturbed. In practice, iterative improvement algorithms often start with a random

initial configuration, or with a heuristically constructed initial configuration that is not as costly as a random solution.

To find a good solution, we try to perturb the known solution to improve it. From the diagram, an obvious approach is to explore easily reached neighbouring configurations and to select the one with least cost, i.e. the one giving the most improvement. In practice, we attempt small random perturbation to the configuration that yields a nearby solution. This process can continue starting from the new configuration until no further improvements are obtained, at which point the algorithm terminates. This strategy seems reasonable, but it has a serious problem : it is easily trapped in local minima, solutions that look good in some small neighbourhood of the cost surface but are not necessarily the global optimum. Standard iterative improvements are a *downhill-only* style. In fig. 4.1 each new perturbation moves to a configuration *downhill* from the previous one, thus becoming trapped in local minima. In practice, one scheme to overcome this is simply to try many random initial configurations, improve this and use the best answer found. However, for very large problems, the computational expenses are great here, the number of random starts needed to adequately sample the cost surface is unreasonable, and still there is no guarantee of finding the best answer.

Simulated annealing offers a strategy very similar to iterative improvement, with one major difference : annealing

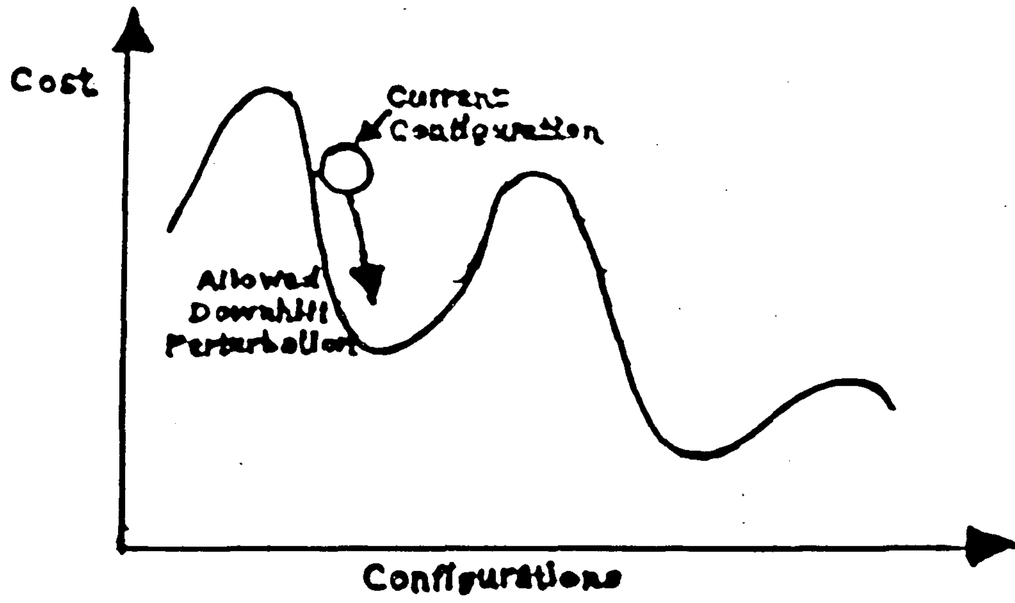


Fig. 4.1

allows perturbations to move uphill in a controlled fashion. The individual perturbations are now referred to as *moves*. Because each move can now transform one configuration into a *worse* configuration, it is possible to jump out of a local minima, and potentially fall into a more promising downhill path. However, because the uphill moves are carefully controlled, we need not worry about getting close to a good, final solution, only to randomly jump uphill to some far worse one.

The relevant analogy here is physical annealing of a solid. To coerce some material into a low energy state, it is heated, then cooled slowly, so that it comes to thermal equilibrium at each temperature. Simulating this process is very similar to a combinatorial optimization task. For the physical system, the goal is to find some arrangements of atomic particles (a configuration) that minimizes the energy of the system. The basic requirement for simulating this process is the ability to simulate how the system reaches thermodynamic equilibrium at each fixed temperature in a *schedule* of decreasing temperature used to anneal it. In physical systems, temperature has a meaning; in arbitrary nonphysical optimization problems, the temperature is simply a control parameter. The idea is to employ a *cooling schedule*, a sequence of decreasing temperatures, to moderate the acceptance of uphill moves over the course of the solution. Initially, this *effective* temperature parameter is high enough to permit an aggressive, essentially random

search of the configuration space, thus allowing most uphill moves. As the temperature cools, fewer uphill moves are permitted. In this temperature regime , annealing closely resembles standard downhill-only iterative improvement.

4.2.1 The Algorithm

The energy of the system at any state S_i is $E(S_i)$ and is determined by a cost function used to assign value to that state. Temperature is used as a control parameter to guide the system to a low cost (low energy) state. The value of temperature determines whether a perturbation that increases the energy is to be accepted. The simulated annealing algorithm is started with a temperature T equal to the initial temperature T_0 and some initial state S_0. The system is perturbed to get a new state S_n.The change in energy (sE) is calculated. If the energy is decreased, the perturbation is accepted .Otherwise the perturbation is accepted with a probability $e^{-sE/T}$. At higher temperatures, this probability is large and most of the moves which increase the energy are accepted. As temperature falls, only small perturbations are allowed. At each temperature, the algorithm reaches equilibrium and then the temperature is reduced. The system is frozen when the system will not improve despite further reductions in temperature.

An important point worth mentioning in connection with simulated annealing algorithm is *range limiting*. It is to be recalled that at colder temperatures, large uphill moves are unlikely to be accepted. Nevertheless, there evaluation takes

time, and it is worthwhile attempting to bias the generation of random moves in favour of those more likely to be accepted and reach closer to the optimal solution. Thus at lower temperatures, some other form of control is required to reach nearer to the optimal solution simultaneously reducing the computational overheads.

Another interesting point is that simulated annealing algorithm is *nondeterministic*, and hence, will produce different answers each time it is run, even on the same problem. This is because of the probabilistic nature of choosing moves and accepting uphill moves. In particular, there is no guarantee of getting precisely the optimum solution to a problem in annealing algorithm or even getting the same solution on multiple runs. What annealing really offers is some probability of getting out of *some* local minima; this is not the same as a guarantee of finding the optimum.

A fundamental question concerns the convergence of simulated annealing algorithm and asks whether it is possible formally to prove that it will converge to an optimal answer. It turns out that, by making certain simplifications, annealing algorithms can be modelled probabilistically; in fact, convergence can be proven. However these technical proofs show that annealing converges *asymptotically, in probability*. In other words, if we perform enough (infinitely many) moves, the probability that we have found a global minimum can be made as close to unity as we like.

4.3 Simulated Annealing and Load Balancing

"And this is the fashion of which thou shalt make it."
Genesis 6:15

The process to processor mapping in the case of simulated annealing is done for both types of problems mentioned on to the processor spaces configured in various topologies like pipeline, mesh, torus, hypercube, binary tree and wk-recursive. For the heuristics algorithm the implementation is done for a completely disjointed processor space. We start placing the processes on to the processors till its share of average load is satisfied. The processes to be clustered on to the single processor are determined by the process - processes communication among the processes being considered for placement. The topology is configured at the end on a demand driven basis of processor - processor communication.

To implement Simulated Annealing for a specific purpose, the following parameters have to be defined :

1. An energy (cost) function.
2. A perturbation technique.
3. A cooling schedule.

For load balancing the three parameters mentioned above are represented as follows :

1) **Energy function definition:** An energy function (Hamiltonian) is defined which sets the goals the algorithm

should achieve.The computational load for each member is say in the FEM/FDM is the same. However since the real-time communication is not known, interprocessor communication distances are sought to be minimized.Thus the intermediate processor will not be tied up in transmitting messages from and to other non-neighbouring processors.The hamiltonian thus consists of two parts :

i) **Computational part** : If W_i = computational load on processor i, then all W_i's should be equal. A measure of the inequality of this load will be obtained by finding the sum of the squares of all W_i's.

This part of the hamiltonian can thus be represented by

$$H_{comp} = A \Sigma W_i^2 \text{ for all } i.$$

H_{comp} will be minimum when all the W_i's are equal.

Here A is a constant which normalises computational and communication loads. I have assumed it to be unity.

ii) **Communication part** : If C_{ij} is the amount of communication and D_{ij} the length of the path between processors i and j then this part of the hamiltonian can be represented by the sum of the products of C_{ij} and D_{ij} as

$$H_{comm} = \Sigma C_{ij} * D_{ij} \qquad \text{for all } i,j.$$

The Hamiltonian can be written as

$$H = A \Sigma W_i^2 + \Sigma C_{ij} * D_{ij}.$$

2) **A perturbation technique** : The system can be perturbed by

i) Moving a member from one processor to another.

ii) Exchanging members between two processors.

Of these, I have used the first technique for load balancing.

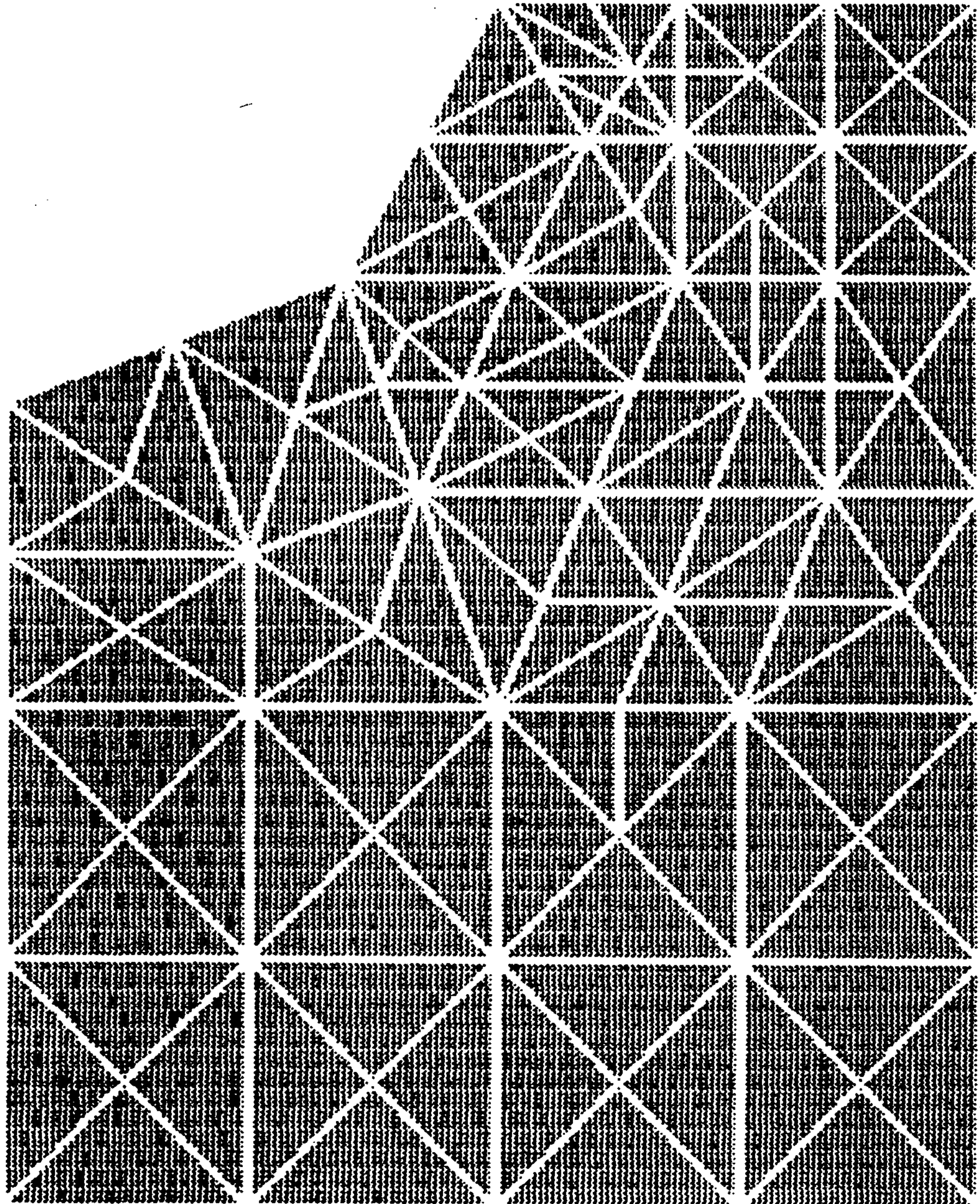
3) **A cooling schedule** : The temperature is reduced by the cooling rate when either of the following conditions holds :

i) Number of moves accepted at that temperature exceed 10 % of the number of members (N).

ii) The number of attempts made = N .

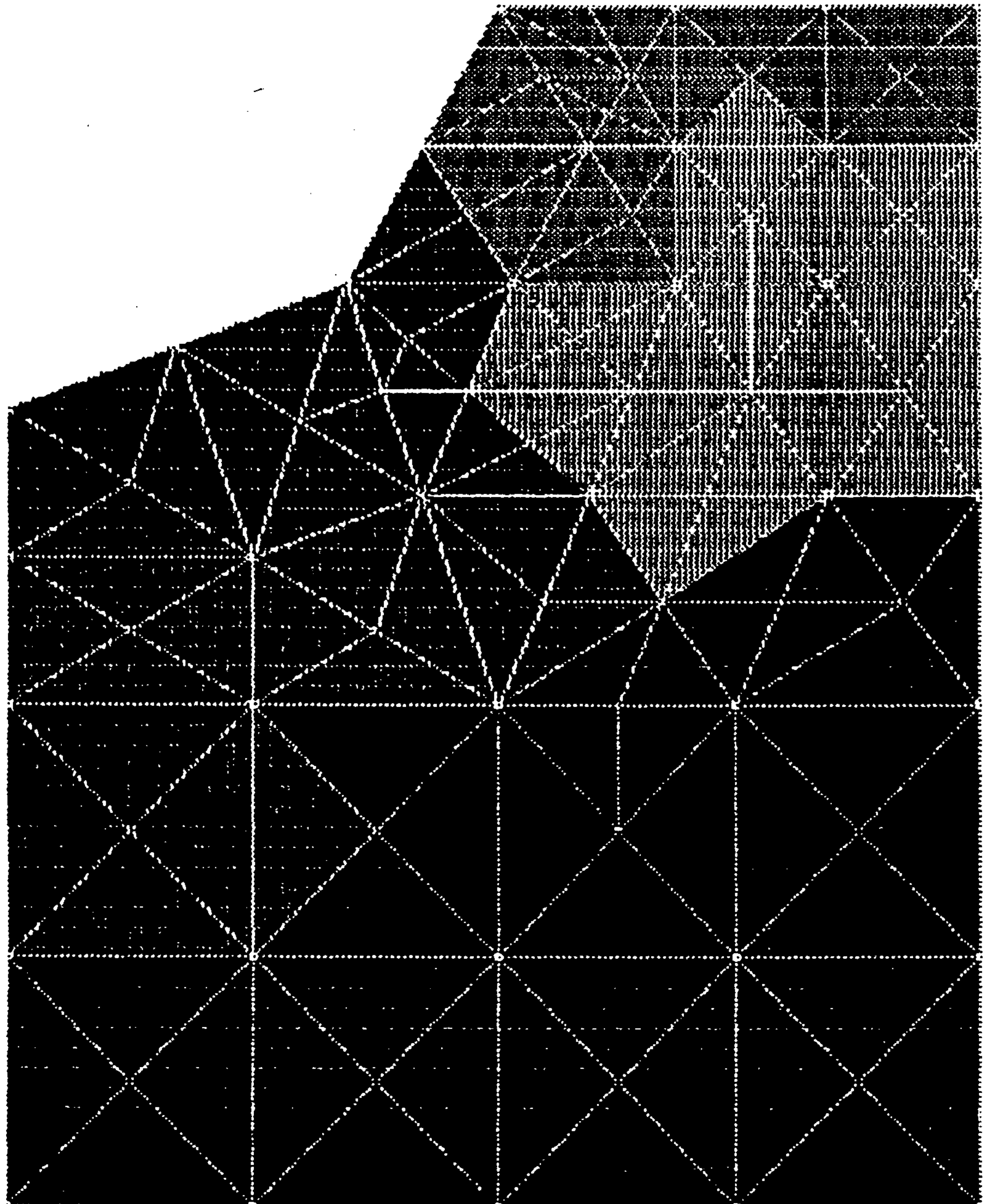
Iterations are started with a high value of temperature. As an initial condition, members (to be balanced) are distributed randomly to processors. A member chosen at random is attempted to be moved to another processor. The move is accepted if the Hamiltonian decreases as a result of this move. If the Hamiltonian increases because of this move, it is accepted with a probability $e^{-sH/T}$, where T is the current temperature of the system and sH = Increase in Hamiltonian. The temperature is reduced as per the cooling schedule mentioned above.

The results obtained by using the simulated annealing algorithm for load balancing are extremely encouraging as has been shown in fig. 4.2 and 4.3 where the processes have been mapped onto a four processor machine. The fig. shows clearly that the processes requiring communication have been placed either on the same node or on a neighbouring node.



Mesh used to generate input communication file for Load Balancing

Fig. 4.2



Elements of mesh mapped onto 4 processors of a parallel system after load has been balanced

Fig. 4.3

CHAPTER FIVE

IMPLEMENTATION

"All words,
And no performance!"
Philip Messenger

The proposed simulated annealing algorithm has been simulated on a VAX-11/780 computer system. The software has been written in C. The software was tested for various topologies such as mesh, torus, binary tree, hypercube pipeline and wk-recursive.

The topologies consisted of sixteen nodes each . The software initially reads two files, the first one of which contains the computational load on each processor whereas the second one contains the identities of different processes that require communication alongwith the amount of communication.

The algorithm consisted of the following modules

(i) The initial assignment module

(ii) The process selection model

(iii) The processor determination module

(iv) The destination processor module

(v) The No. of hopes module

(vi) The hamiltonian calculating module

(vii) The router module

5.1 The Initial Assignment Module

This module is called by the main program. This module initially assigns processes to all processors. In this

simulation two types of initial assignments were considered : random and grid. In the random model, a process was arbitrarily assigned to a processor by generating a random integer between 0 and 15. In the grid start, a processes are numbered like grid points and are serially assigned to processors.

5.2 The Process Selection Module

This module is responsible for selecting on process that is to be moved from its present processor to some other processor. In this implementation, the selection of process was random.

5.3 The Processor Determination Module

This module is used to determine the processor on which the process selected to be moved is presently stored. Since the processes were to be moved regularly from one node to another, this was achieved by carrying out a linear search of the processqueue of each processor until the selected process was found.

5.4 The Destination Processor Module

This module returns an integer which is the identification number of the processor to which the selected member is to be moved. This selection was of two types : random and restricted. In the random move, a member was

attempted to be moved to any randomly generated processor. In the restricted move, the new processor was chosen from among the processors on which the neighbours of the member, chosen to be moved, are lying.

5.5 The No.-of-Hops Module

This module returns the no. of links that are to be transferred when a process is to be moved from a source node to a destination node by the shortest route. This value is required in the calculation of the second part of the hamiltonian. Although the value returned by this module could also have been obtained from path module which returned the path along which the process was to be moved, it was highly desirable to have a separate and simpler module for the purpose of determining the no of links in order to reduce the time complexity since a process selected might not be moved each time. The no. of hop module for various topologies is described below.

5.5.1 Mesh

The processors are arranged in a rectangular array along m rows and n columns.

In this case m = n = 4

If s = source node and d = destination node then

```
source_x_coord = s%n;
source_y_coord = s/n;
dest_x_coord = d%n;
```

```
dest_y_coord = d/n;
hops = abs(source_x_coord - dest_x_coord) +
       abs(source_y_coord- dest_y_coord);
return hops;
```

In the worst case when the processor s and d are at the two diagonally opposite ends, the no. of links needed to be traversed becomes m+n-2. For a 4 x 4 mesh this value was 6.

5.5.2 Torus

Here also the processors are connected along m rows and n columns with the difference that the processors at the boundary of one end are connected to those at the other end. For m = n = 4, the no of hops algorithm is given as follows:

If s = source node and d = destination node then

```
sour_x_coord = s%n;
sour_y_coord = s/n;
dest_x_coord = d%n;
dest_y_coord = d/n;
If abs(sour_x_coord - dest_x_coord) > n/2
      hops_x = abs(abs(sour_x_coord - dest_x_coord) - (n/2));
else
      hops_x = abs(sour_x_coord - dest_x_coord);
If abs(sour_y_coord - dest_y_coord) > m/2
      hops_y = abs(abs(sour_y_coord - dest_y_coord) - (m/2));
else
      hops_y = abs(sour_y_coord - dest_y_coord);
hops = hops_x + hops_y;
return hops;
```

5.5.3 Binary Tree

In this case the processors are numbered from 0 to n - 1, where n is the number of nodes. For n = 16, they are numbered from 0 to 15. The processor no. 0 acts as the root node of the binary tree. As explained in chapter 2, a processor i is connected to a processor j if i = 2*j + 1 or i = 2*j + 2 or vice versa. The code for finding the number of hops is recursive.

If s = source node and d = destination node then

```
        if (s == d) hops = 0;
        else if (s > d)
        {
          if even(s) hops = 1 + no_of_hops(s/2-1, d);
          else hops = 1 + no_of_hops(s/2, d);
        }
        else
        {
          if even(d) hops = 1 + no_of_hops(s, d/2-1);
          else hops = 1 + no_of_hops(s, d/2);
        }
return hops;
```

5.5.4 Pipeline

In this case, the no of links required to be traversed for sending a message from one node to another through shortest route is given by the absolute value of the difference between s and d, where s is the source node and d is the destination node.

5.5.5 Hypercube

Here the no. of hops is calculated by comparing the binary values of the two nodes and finding the number of bits in which they differ.

Thus, if s = source node and d = destination node then

temp = s ^ d where ^ is the exclusive OR operator.

The number of bits which are 1 in the binary representation of temp gives the no of hops. To count the number of bits in temp which are 1, the most significant bit is checked each time and temp is rotated left until it becomes 0. Each time the most significant bit is found to be 1, the counter is incremented by 1. In this case, the no of hops will be maximum when all bits of s are the complements of the corresponding bits of d.

5.5.6 WK-recursive

In this case the number of hops is calculated by using the following code.

```
hops = 0;
s0 = s & 3;
s1 = s & 12;
s1 = s1 >> 2;
s2 = s & 48;
s2 = s2 >> 4;
s3 = s & 192;
s3 = s3 >> 6;
d0 = d & 3;
d1 = d & 12;
```

```
d1 = d1 >> 2;
d2 = d & 48;
d2 = d2 >> 4;
d3 = d & 192;
d3 = d3 >> 6;
if ( s3 != d3 ){
        if( s0 != d3){
                s0 = d3;
                hops++;
        }
        if( s1 != d3 ){
                s1 = d3;
                hops++;
                hops++;
        }
        if( s2 != d3){
                s2 = d3;
                hops += 4;
        }
        s0 = s1 = s2 = s3;
        hops++;
}
if( s2 != d2){
        if( s0 != d2){
                s0 = d2;
                hops++;
        }
```

```
        if( s1 != d2 ){
                s1 = d2;
                hops++;
                hops++;
        }
        s0 = s1 = s2;
        hops++;
}
if( s1 != d1){
        if( s0 != d1){
                s0 = d1;
                hops++;
        }
        s0 = s1;
        hops++;
}
if( s0 != d0){
        hops++;
}
return hops;
```

5.6 The Hamiltonian Calculating Module

This module calculates the value of the hamiltonian during each iteration and, if it is less than the value of hamiltonian in the previous iteration, calls the router module to determine the path alongwhich the process is to be moved. If the value of hamiltonian increases, it calculates the probability to find whether

5.7 The Router Module

This module is used to determine the path alongwhich a selected process is to be moved from one node to another. In this module Dijkstra's shortestpath algorithm has been used. Since the simulation was carried on a serial machine, it was not possible to actually move a process from one node to another. However, for statistical purposes, the transmission and delay time at each node was calculated to analyse the effects of communication among the processes. This module takes three parameters - the address of the source node, the address of the destination node and an array in which it copies the path alongwhich the message is to be routed.

In addition to above, some other modules that were used in this simulation are **addqueue** and **deleteQ**. Addqueue adds a process to the processqueue of a processor. DeleteQ removes a process from the process queue of a node.

5.8 Experimental Results

The experiment was carried out within a temperature range of 4 to .1 degrees. The cooling rate was 0.97. The total number of members to be balanced were taken as 208. The input communication patterns were of two types :

a) Regular : In problems with automatic grid generation, the members are numbered in order. This represents a regular communication pattern in which communication is of contiguous nature.

b) Irregular : If the members are not properly numbered, (e.g. 0 talks to 127 which in turn talks to 61 etc.), it represents an irregular communication pattern. This input was generated by generating two random numbers, communicating with each other.

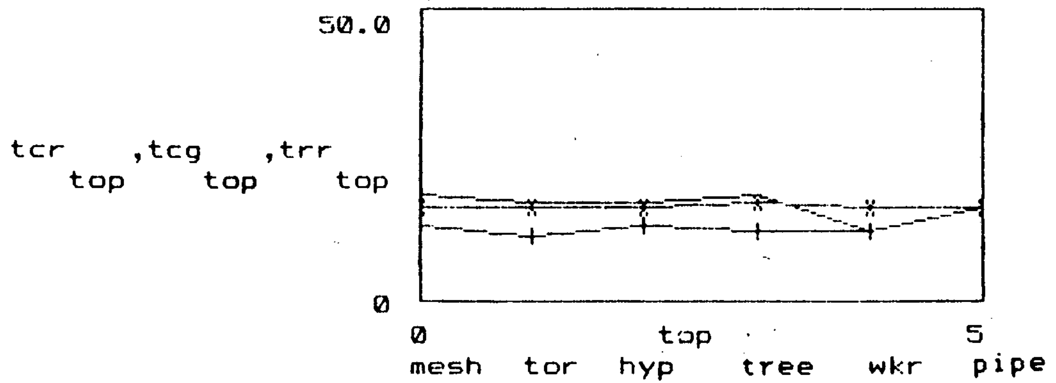
The results have been shown in table 5.1.

TABLE 5.1

Number of members = 208

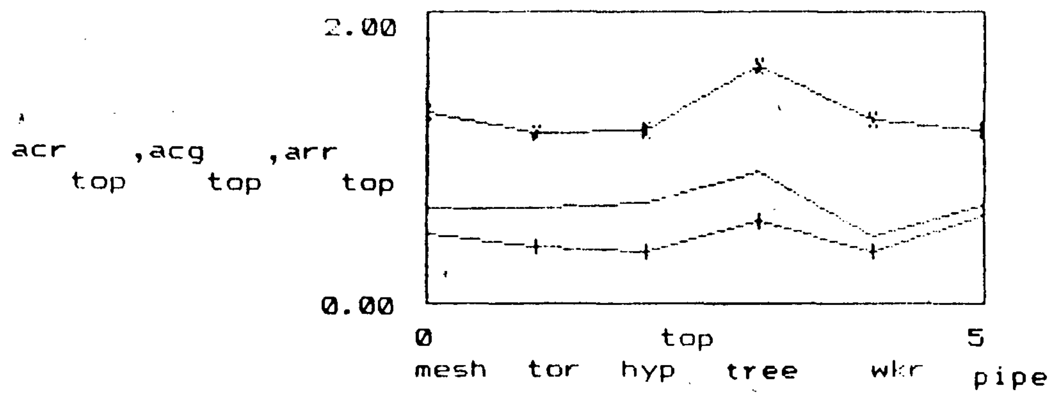
Number of processors = 16

Topology	Best % Mixed	Best % Grid	Time (Sec) Mixed	Time (Sec) Grid
Mesh	40 - 5	25 - 10	18	13
Torus	100 - end	30 - 10	17	11
Tree	100 - end	30 - 10	18	12
hypercube	100 - end	10 - 0	17	13
pipeline	40 - 10	-	12	-
wkrecursive	100 - end	100 - end	11	11



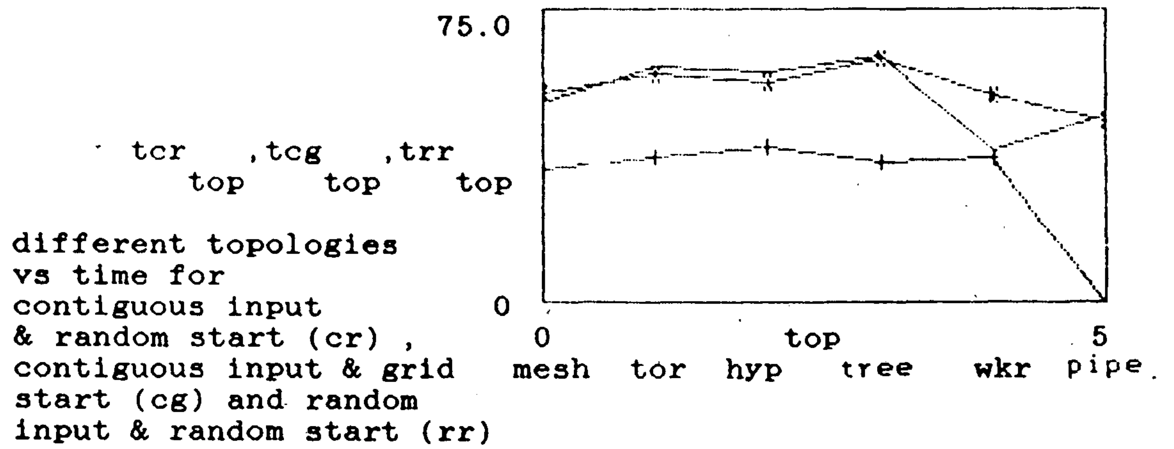
Different topologies vs
time (line, plus, diamond)
for regular input & random
start-tcr,& grid start -tcg,
random input & random start
(trr)

Fig. 5.1



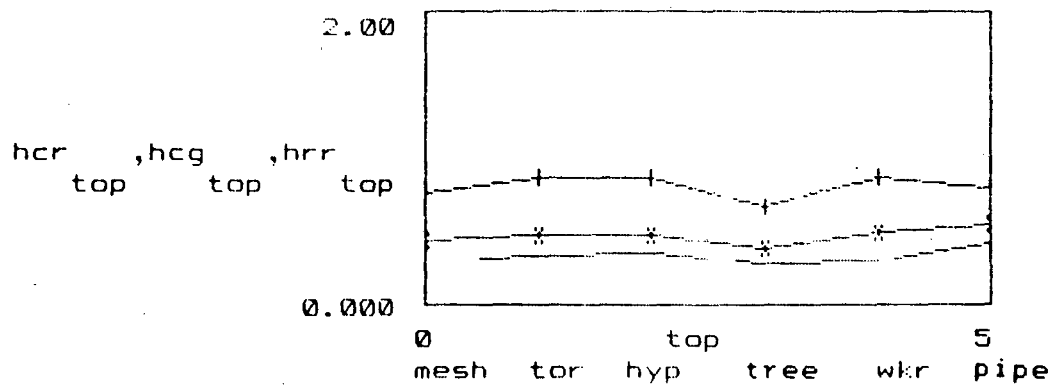
Different topologies vs
av. hops (line, plus, X)
for regular input & random
start-acr,& grid start -acg,
random input & random start
(arr)

Fig. 5.2



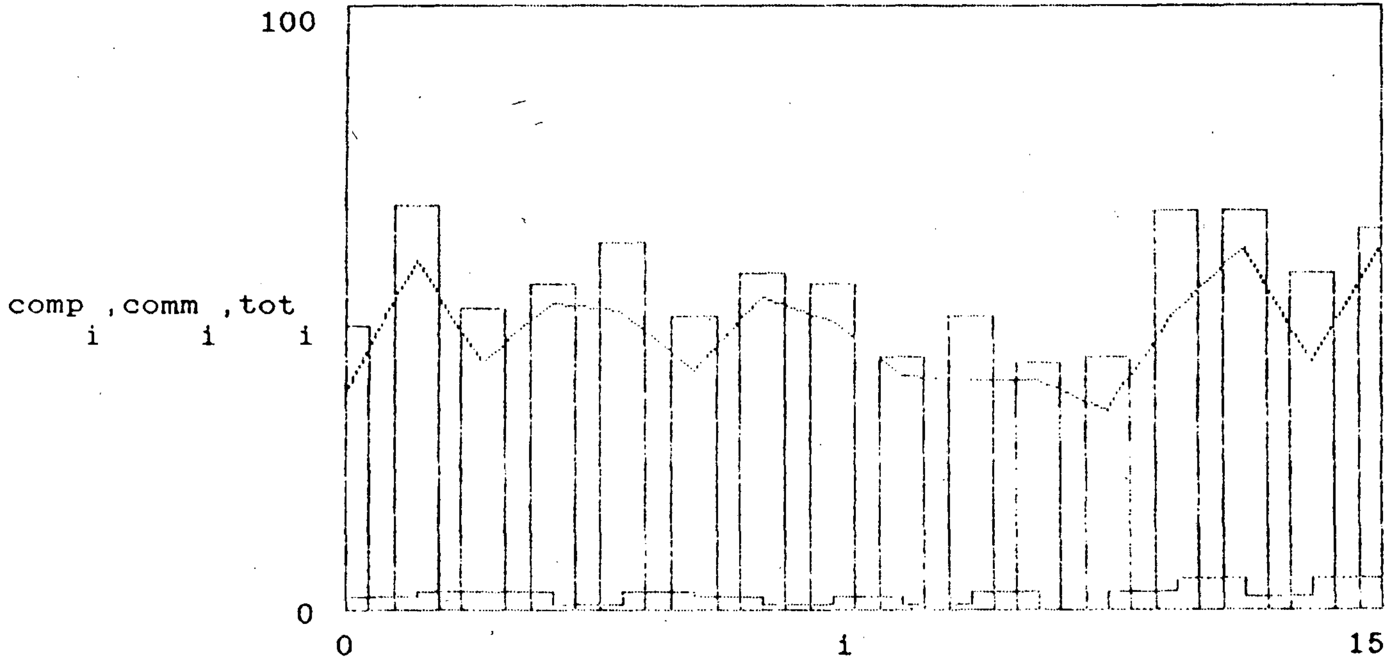
KEY To The FIGURES : (cr) is 'line' (rr) is 'x'
& (cg) is '+'.

Fig. 5.3



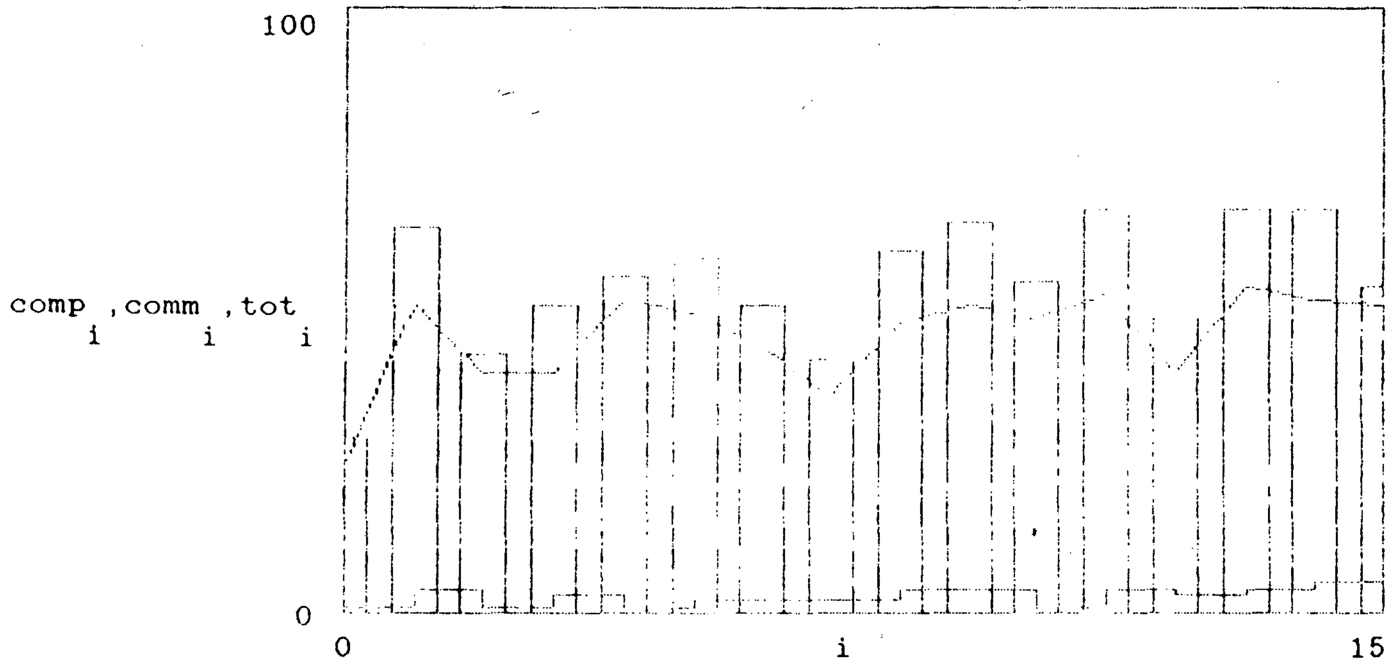
Different topologies vs hamiltonian (line, plus, X) for regular input & random start-hcr,& grid start -hcg, random input & random start (hrr)

Fig. 5.4



Members : 208, Communication pattern : Contiguous
Topology : WK Recursive, Procs : 16
Initial assignment: Random
Hamiltonian : 333
ave.load : 53.4, ave.hops : 0.58,
Nonneighbour communications : 29
Time: 39 secs.
Type of moves : Random from 40% to 10% of temp.
Temperature range : 4 to 0.1

Fig. 5.5



Members : 208, Communication pattern : Contiguous
Topology : Mesh, Procs : 16
Initial assignment: Random
Hamiltonian : 358
ave.load : 54.5, ave.hops : 0.66,
Nonneighbour communications : 25
Time: 51 secs.
Type of moves : Random from 40% to end of temp
Temperature range : 4 to 0.1

Fig. 5.6

5.9 Conclusion

> **"Better is the end of a thing than the beginning of it."**
>
> *Ecclesiastes 7:8*

The results obtained on the basis of experiments carried out on simulated annealing have been quite encouraging. The algorithm can, therefore, be applied on a real parallel machine for the purpose of load balancing to achieve the desirable performance of reducing the computation time for massively CPU bound tasks.

The algorithm, however, suffers from a major drawback : i.e. it requires too much of time to achieve the desired goal of proper load partitioning. Another drawback of the algorithm is that the balancing achieved is static. Thus, in those applications where processes are dynamically created and destroyed, the performance of the parallel program may be far from desired. Nonetheless, it is to be kept in mind that parallel programs are mainly used in largescale scientific applications where the processes are normally static and a fairly good speedup can be achieved.

BIBLIOGRAPHY

"For in thy book all things are written"
Psalm 139:16

[1] Ahluwalia, V., "*Design of a User Friendly Communication System for a Distributed Memory Parallel Computing System*", M.Tech. Dissertation, School of Computer and System Sciences, Jawaharlal Nehru University, New Delhi, January 1993.

[2] Ahluwalia, V & Singh, G.S., "*Load Partitioning on a Transputer Based Parallel Computing System Using MIMD Algorithms of Simulated Annealing and Heuristics*", ASME '90 Conference, Tirupati, December 1990.

[3] Bal, H.E., Steiner, J.G.and Tanenbaum, A.S.,"*Programming Languages for Distributed Computing Systems*", ACM Computing Surveys, Vol. 21 No. 3, September 1989.

[4] Bultan, T. & Aykanat, C., "*Circuit Partitioning Using Parallel Mean Field Annealing Algorithms*",IEEE symposium on Parallel and Distributed Processing,December 1991.

[5] Christie, D.J.E., "*Virtual Channels on the MIEKO MMVCS*", M.Sc. Degree, University of Edinburgh, September, 1988.

[6] Department of Defense, U.S., "*Reference Manual for the ADA Programming Language*", ANSI/MIL-STD-1815A, January 1983.

[7] Dietel, H.M., "*An Introduction to Operating Systems*", Addison-Wesley,1990.

[8] Gichev, D., "*An Algorithm for Routing Messages Between Processing Elements in a Multiprocessor System Which*

Tolerates a Maximal Number of Faulty links",Mathematical and Computer Modelling vol. 16,No. 12, 1992.

[9] Hey, A.J.G., "*Transputers and Occam*".

[10] Hockney, R.W. & Jessophe, C.R., "*Parallel Computers*", 1981

[11] Hwang, K. & Briggs, F.A., "*Computer Architecture and Parallel Processing*", McGraw Hill Book Company 1985.

[12] Rutenbar, R.A., "*Simulated Annealing Algorithms : An Overview*", IEEE Circuits and Devices Magazine, January 1989.

[13] Singh, G.S. & Deshpande, K.R.,"*On Fast Load Partitioning by Simulated Annealing and Heuristic Algorithms for General Class of Problems*".

[14] Singh, G.S., "*An Integrated Feasible Approach to the Design of Parallel Computing System - Part 2*",Bhabha Atomic Research Centre, 1988.

[15] Stone, H.S., "*High Performance Computer Architecture*", Addison Wesley Publishing Company,1990.

[16] Tanenbaum, A.S., "*Computer Networks*", Prentice Hall of India, 1989.

[17] Vecchia, G.D. and Sanges, C., "*Recursively Scalable Networks for Message Passing Architecture*", Parallel Processing and Applications,1988.

[18] Vecchia, G.D. & Sanges, C., "*An Optimized Broadcasting Technique for WK-Recursive Topologies*", IMACS 1988.