1203

AN EXPEDIENT SCHEME FOR PERFORMANCE EVALUATION ON PARALLEL COMPUTERS

DISSERTATION SUBMITTED BY

MANOJ KUMAR SARANGI

IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR DEGREE OF
MASTER OF TECHNOLOGY

IN

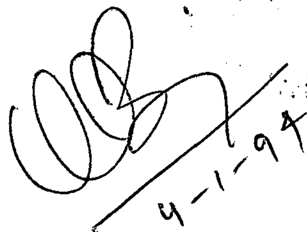
COMPUTER SCIENCE & TECHNOLOGY

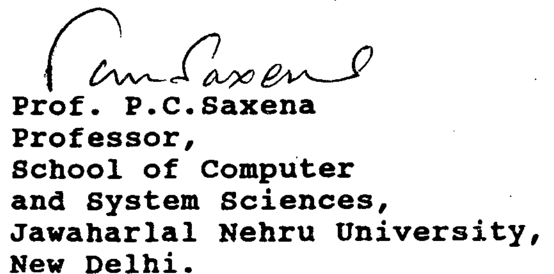
SCHOOL OF COMPUTER AND SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI-110 067
JANUARY 1994

CERTIFICATE

This is to certify that the dissertation titled **"An Expedient Scheme for Performance Evaluation on Parallel Computers"** being submitted by **MANOJ KUMAR SARANGI** 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
Dean,
School of Computer and
and System Sciences,
Jawaharlal Nehru University,
New Delhi.

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

To

my parents

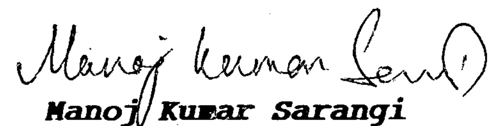
ACKNOWLEDGEMENTS

I bestow my gratitude 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 very much indebted 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.*

My sincere thanks to my friends Rajesh for giving me company in the long hours of nights during the period of this project.

Last but not the least, I thank Jisnu and Abdaal for helping me in all possible ways towards the completion of the dissertation.



Manoj Kumar Sarangi

It took five months to get the word back to the Queen Isabella about the voyage of Columbus; two weeks for Europe to hear about the assassination of Lincoln and only 1.3 seconds to get the word from Neil Armstrong that man can walk on the moon.

Isaac Asimov's Book of Facts

CONTENTS

CHAPTER ONE

INTRODUCTION 1

1.1 Towards Parallelism

1.2 Role of Performance Evaluation and Prediction

1.3 Measures of Performance

1.4 Organisation of This Report

CHAPTER TWO

MODELLING METHODS 12

2.1 Models of Computation

2.1.1 SISD Computers

2.1.2 MISD Computers

2.1.3 SIMD Computers

2.1.3.1 Shared Memory Type

2.1.3.2 Interconnection Network Type

2.1.4 MIMD Computers

2.2 Programming MIMD Computers

2.3 Analysis of Parallel Algorithms

2.3.1 Running Time

2.3.2 Number of Processors

2.4 Parallel Software Design Issues

2.4.1 Portability

2.4.2 Design (Correctness)

2.4.3 Implementation

CHAPTER THREE

PERFORMANCE ANALYSIS METHODOLOGIES 29

3.1 Importance of Performance Analysis

3.2 Task Partitioning and Task Coalescence

3.3 Discrete Event Simulation

3.4 Critical Path Analysis

3.4.1 Selection of Event Execution Time and Communication Cost

3.4.2 Number of Events to be Processed

3.4.3 Load Balanced Process Assignment

3.4.4 Interactions between the Number of Processors and the Communication Cost

3.4.5 Increasing the Problem Size

3.4.6 Increasing the Problem Size with Fixed Processes to Processors ratio

3.4.7 Increasing Message Density

CHAPTER FOUR

IMPLEMENTATION 52

4.1 Important Parameters

4.2 Computations

4.3 Performance Measures

4.4 Assumptions

CHAPTER FIVE

CONCLUSION 61

BIBLIOGRAPHY 63

CHAPTER ONE

CHAPTER ONE

INTRODUCTION

1.1 Towards Parallelism

Several decades ago the era of parallel computation has started when analog computers were applied to simulate continuous dynamic systems. Analogue computation is based on application of continuous physical laws to realize mathematical operation in parallel. The mathematical operations are simulated as a physical analogy by means of electrical and electronic circuitry. Analogue computation is parallel by nature. System simulation by way of physical processes was possible only for a few mathematical operations like *inner product* and *integration* with respect to time. For this reason parallel analog computation was mainly used for simulation of continuous systems. In particular the fact that analogue computations are executed in the continuous time domain made it easy to realize that analogue computing devices could efficiently operate and cooperate in parallel without any synchronization. Analogue computing devices produce and consume intermediate results in the continuous time set. Consequently exchange of data can simply be done by means of electrical interconnection consisting of electrical wiring.

Later on stand alone analogue computers have been replaced by **hybrid computers** (*i. e; a computer systems having analogue and digital components*). The incorporation of a

digital computer made it possible to run the analogue computer under control of a digital program. In a sense the analogue computing part acted as a very powerful coprocessor for the digital computer. From the mid 80's parallel digital computation has become more and more important in comparison to hybrid computation.

In sequential digital computation a processing task is described and programmed as sequence of digital operations. These operations are carried out one after another, where the execution time of digital operation depends on it's complexity. In digital computation the production of intermediate results and the consumption of intermediate results as well take place in discrete time sets. There belongs discrete time sets for different processing tasks. This is the main reason why it is difficult to realize that simultaneously operating digital computing devices can cooperate in an efficient way.

Supercomputers have made parallel digital computation very popular. Supercomputers are conventional digital computers ,but with a CPU architecture dedicated to execute numerical linear algebra through pipelined processing. For this reason these supercomputers are also called vector computers. Pipelined processing is a special working out of parallel processing, similar to flow production by means of an assembly line. A processing task is not executed as a sequence of subtasks in the normal way by means of one computing device, but each subtask is executed by a separate processing device. These processing devices are coupled in cascade,in case of one processing task these devices are busy one by one.

Pipelined processing can be applied very successfully in case a processing task has to be executed many times and if moreover this task can be sub divided in a sequence of subtasks of equal execution time. Now supercomputers are on the way to become parallel digital computers ,where the individual processors are very powerful with regard to efficient parallel processing.

The *near future* need of much more powerful supercomputation ask for parallel (digital) computers , containing a large number of fast processors that can also co-operate fast and efficiently. In parallel processing high performance *data processing* and ditto *data flow* are of equal importance . In practice so far loss of efficiency often happens for the technical reason that the communication system of a parallel computer has not enough capacity. Lack of communication capacity will result in *transfer bound* processing instead of *compute bound* processing. Loss of efficiency also often happens for the technical reason that the communication system of a parallel computer has not enough capacity. Lack of communication capacity will result in transfer bound processing instead of compute bound processing. Loss of efficiency also often happens because of the fact that parallel algorithmics is still in the early stage of development. That makes it difficult to define the architecture and programming of a parallel computer such that efficient implementation of parallel algorithms is possible in a wide range of applications. Moreover the applicability of parallel computa-

tion is hampered by the fact that the programming in parallel computers is still more difficult than programming in serial computers. One can expect that this will change because most of the problems in practice are parallel in nature.

1.2 Role of Performance Evaluation & Prediction

The issue of *performance evaluation and prediction* has concerned users throughout the history of computer evolution. In fact, the issue is most acute when the technology is young; the persistent pursuit of products with improved cost-performance characteristics then constantly leads to designs with untried and uncertain features.

The need for computer performance evaluation exists from the initial conception of a system's architectural design to it's daily operation after installation. In the early planning phase of a new computer system product, the manufacturer usually makes two types of *prediction*. The first type is to forecast the nature of applications and the levels of ***system workloads*** of these applications. Here the term workload means, the amount of *service requirements* placed on the system. The second type of prediction is concerned with the choice between architectural design alternatives, based on hardware and software technologies that will be available in the design period of the planed system. Here the criterion of selection of selection is what we call ***cost performance tradeoff***. The accuracy of such prediction rests to a consid-

erable extent on our capability of mapping the performance characteristics. Such translation procedures are by no mean straightforward or well established. After the architectural decisions have been made and the system design and implementation started, the scope of performance evaluation becomes more specific. The interactions among the operating system components- algorithms for *job scheduling,processor scheduling,* and *storage management* must be dealt with,and their effects on the performance must be predicted. Comparing the predicted performance with achieved performance often revels major defects in the design or errors in the system programming. Now it is universally accepted that the performance evaluation and prediction process should be an integral part of the development efforts throughout the design and implementation activities.

During the 1980s,interest in performance analysis increased, partly because, as architectures became more complicated it was recognised how important it would be to be able to predict the performance of new systems before they are built. At the same time it was more important to measure and characterize performance on existing systems because the ratio of peak to actual performance could now be several orders of magnitude. However performance analysis was made more difficult by the large number of variables that can effect performance in a highly parallel system. There was also an increased awareness of the difficulty of avoiding biases in measuring in measuring performances and of considering properly all the factors that affect performance. There

were and are many projects aimed at getting better understanding of performance measurement, carrying out performance measurements on different systems,and even attempting to characterize performance. There is a strong emphasis towards vector oriented computers, but some of the work is also done on parallel systems. The long range goals on which prime importance is given include building up an understanding of fundamental computations that are used in different application fields, for example, solving sparse linear systems in fields such as chemical engineering and electronic circuit design , and a large database of performance information both at the entire application level and at the kernel level for many different computers. A long- range goal is to characterise the variables and factors that affect performance on different computer architectures for different application classes. In the late 1980s, from approximately 1987 onwards, there are several projects initiated aimed specifically in measuring the performance of parallel computers.

Finally, one indication of the level of interest in the field of performance evaluation is that there are now several prizes given for achieving certain level of performance. One of the first was the Alan Karp prize, which was a one time prize given for first program to achieve a speed up of 200 on a real application on a real parallel computer. Gordon Bell also established what is now a series of prizes for achievements, such as best price performance, absolute top speed, and parallelism.

1.3 Measures of performance

When we say the *performance of the computer is great* it means perhaps that the quality of service delivered by the system exceeds our expectation. But the ***measure of service quality*** and the ***extent of expectation*** vary depending on the individual involved,e.g; system designers,installation managers ,terminal users etc. If we attempt to measure the quality of computer performance in the broadest context, then we must consider such issues as user response (*as well as the system response*), ease of use, reliability, user's productivity, and the like as the integral parts of the system's performance. Such discussions,however, fall within the realm of quantitative sciences that involve social and behavioural sciences. Despite our full awareness that performance analysis cannot avoid what are ultimately behavioural questions,the scope of this project work is quite limited: The performance analysis is discussed only in terms of ***clearly measurable quantities***. This is done in the same way as we conventionally define, for instance the signal-to- noise ratio probability of decoding errors as measures of performance of communication systems.

The performance measures can be classified into two broad categories: ***user oriented measures*** and ***system oriented measures***. The user oriented measures include such quantities as the *turnaround time* in a batch system environment and the response time in a real time and/or interactive environment.

The *turnaround time* is the length of time that elapses from the submission of the job until the availability of it's processed result. In the similar way in an interactive environment the *response time* of a request represents the interval that elapses from the arrival of the request until it's completion in the system.

In interactive systems, sometimes we use the term **system reaction time** which is the interval of time that elapses from the moment an input arrives in the system until it receives it's first *time slice* of service. It measures how effective a scheduler is in dispatching service to a newly arrived input.Turnaround time,response time,and reaction time are all considered random variables; hence their *distribution, expected values,variances* are of importance to the designer.

Usually jobs are categorised according to their priority classes. Many factors may determine the assignment of priority to a job : the job's urgency,it's importance,and it's resource demand characteristics and utilization.

Throughput is defined as the average no of jobs processed per unit time. It provide the degree of productivity that the system can provide. If jobs arrive at a system according to some mechanism that is independent of the state of the system, throughput is equivalent to the average arrival rate, provided that the system can complete the jobs without creating an ever increasing back-log. But in this case throughput is not an adequate measure of performance; rather it is a measure of system workload.

The term throughput has some meaning when either there is always some work awaiting the system's service, or the job arrival rate depends on the system's state. Considering the first case in the light of queuing theory the obvious implication is that the system is unstable in the sense that the queue or the backlog will grow without bound. So for the sake of simplicity we can define throughput over a finite interval in which the input queue is never empty. Hence the throughput thus defined is a proper indicator of the ***system's capacity***. The second case assumes importance when we take a finite number of generation sources. For example, if in an interactive system, there is a finite number **N** of terminal user actually logged in. Let's assume that a terminal is blocked while it's request is in the system, either waiting for or receiving service. If there are n jobs in the system, only the remaining N-n terminals are eligible for generating requests. Thus, the effective arrival rate is a (linearly) decreasing function of the system state, n. We can envision a similar situation in a batch-system environment: There may be a sufficiently large number of users to keep the system continually busy. In reality, however, as the system congestion level increases, a user may be discouraged from submitting a new job. Again, the job arrival rate will be some decreasing function of the number of outstanding jobs. This *negative feedback* loop inherent in the job generation mechanism makes the system always stable.

The *utilization* of a resource is the fraction of time that the particular resource is busy. The CPU utilization is the most popular measure of system usage, although it is not necessarily the most important in complex systems. When the CPU is not idle, it may be in either of two busy states: the *problem program state* (or simply the *problem state*) and the *supervisory program state* (or the *supervisor state*). The former represents the portion of time when the CPU is actually executing the programs written or called by the users; the latter is the time consumed in executing such operating system components as the scheduler and various interrupt-handling routines. The distinction is commonly assumed to be synonymous with that of "useful work" versus "overhead". Yet it must be noted that much of the supervisor-state operation provides necessary and useful service for the user programs; hence the "overhead" categorization may be misleading.

If we assume the system having single CPU, and if the CPU utilization figure excludes the supervisor state, then we find the following simple relationship between throughput α (jobs per second) and the CPU utilization μ_{CPU}

$$\mu_{CPU} = \alpha \text{Æ}_{CPU}$$

Where Æ_{CPU}(seconds per job) represents the average CPU time required to process a job.

The mean response time, which we denote by Ä , is found to have the simple relation with throughput

$$\mu \text{ Ä} = \text{ÿ}$$

in which ÿ represents the average number of jobs (waiting or being served) in the system.

1.4 Organization of the report

Chapter 2 contains different classifications of parallel computers,programming in parallel computers, their modelling techniques, the parameters and their evaluation methods.

Chapter 3 deals with different performance analysis methodologies that are often used. The importance of this work is also discussed in the beginning of this chapter.

Chapter 4 contains the details of implementation, the limitations and also the assumptions.

Chapter 5 discusses the future scope of this work.

CHAPTER TWO

CHAPTER TWO

MODELLING METHODS

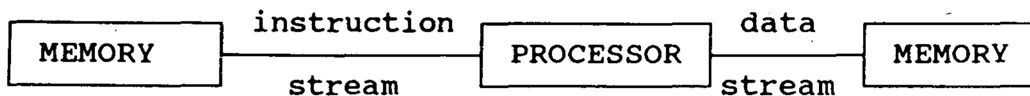
2.1 Models of Computation

Any system, whether serial or parallel,functions by executing instruction on data. A stream of instructions (the algorithm) prompts the computer what to do at each step. A stream of data i.e;the input to the algorithm is affected by these instructions. Depending on these streams the computers can be classified into four broad categories (Flynn's classification).

1. Single Instruction Stream Single Data Stream (SISD)
2. Multiple Instruction Stream Single Data Stream (MISD)
3. Single Instruction Stream Multiple Data Stream (SIMD)
4. Multiple Instruction Stream Multiple Data Stream (MIMD)

2.1.1 SISD Computers

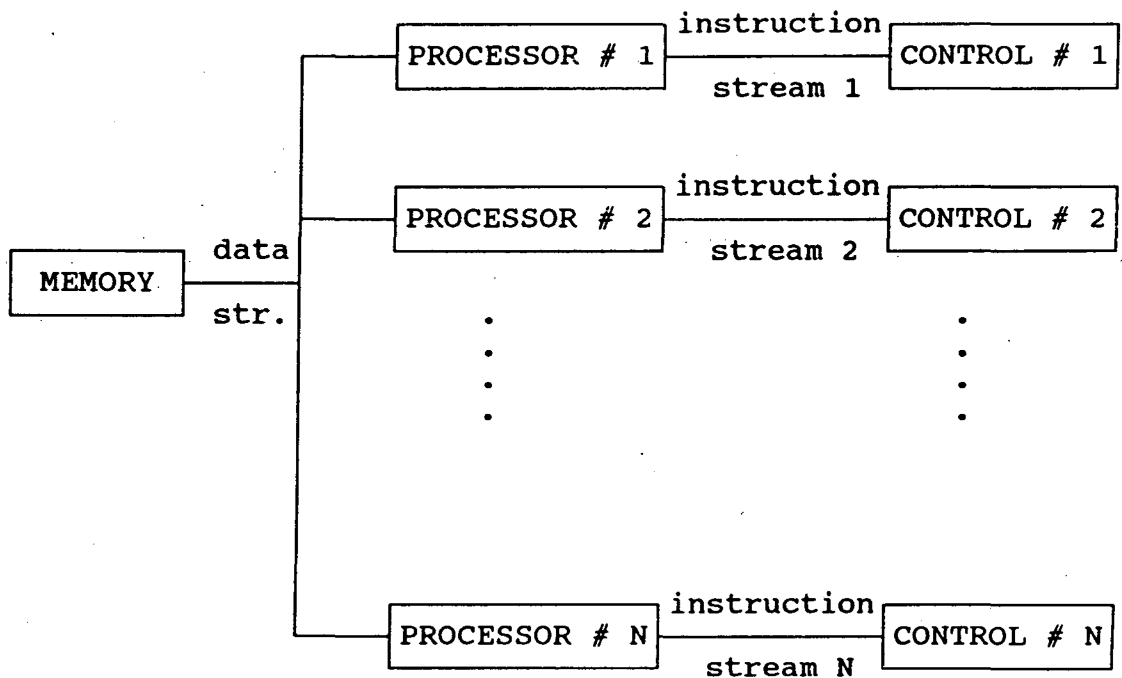
A computer in this class consists of a single processing unit receiving a single set of instructions that operate on a single stream of data.



At each step during computation the control unit omits one instruction that operate on a datum obtained from the memory unit. Such type of instruction may conveys the system to perform some arithmetical or logical operation on the datum and then put that back in memory.

2.1.2 MISD Computers

In this case a number of processors each with it's own control unit share a common memory unit where data reside. (refer figure). Here the number of instruction is same as the number of processors and there is only one stream of data.

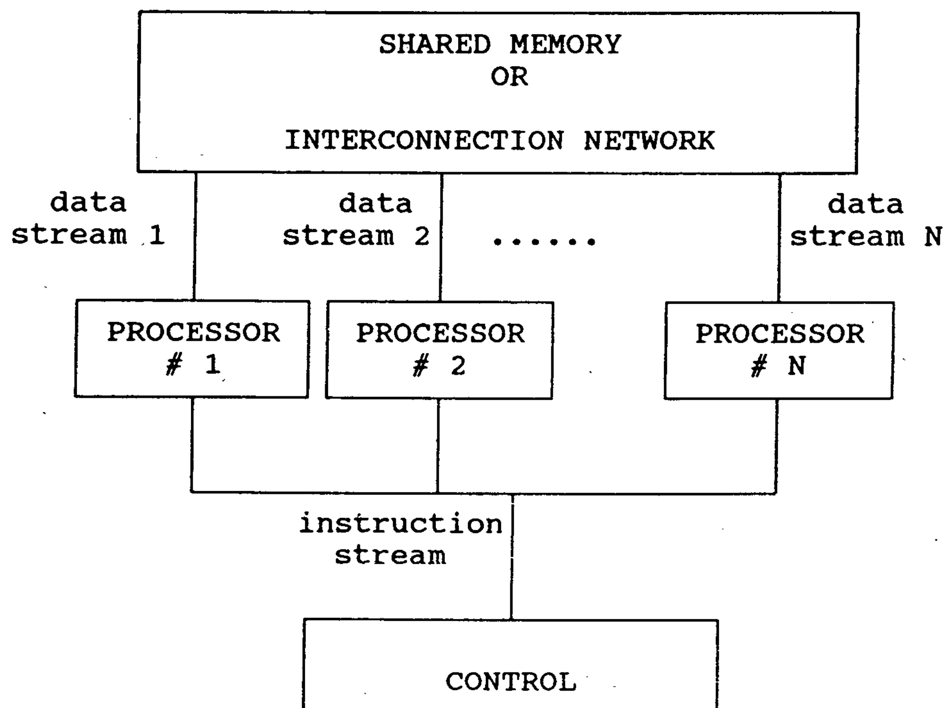


In this configuration, at each step, one datum received from memory is operated upon by all the processors simultaneously, each according to the instruction received from it's

control unit. Thus parallelism is achieved by letting the processors do different things at the same time on the same datum. This class of computers lend itself naturally to those computations requiring an input to be subjected to several operation, each receiving inputs in it's natural form.

2.1.3 SIMD Computers

In this case N identical processors with their own local memory where they can store both program and data. All processors operate under the control of a single instruction stream issued by a central processing unit. Hence the N processors may be assumed to hold identical copies of a single program, each processor's copy being stored in it's own local memory. So there are N data stream, one per each processor.

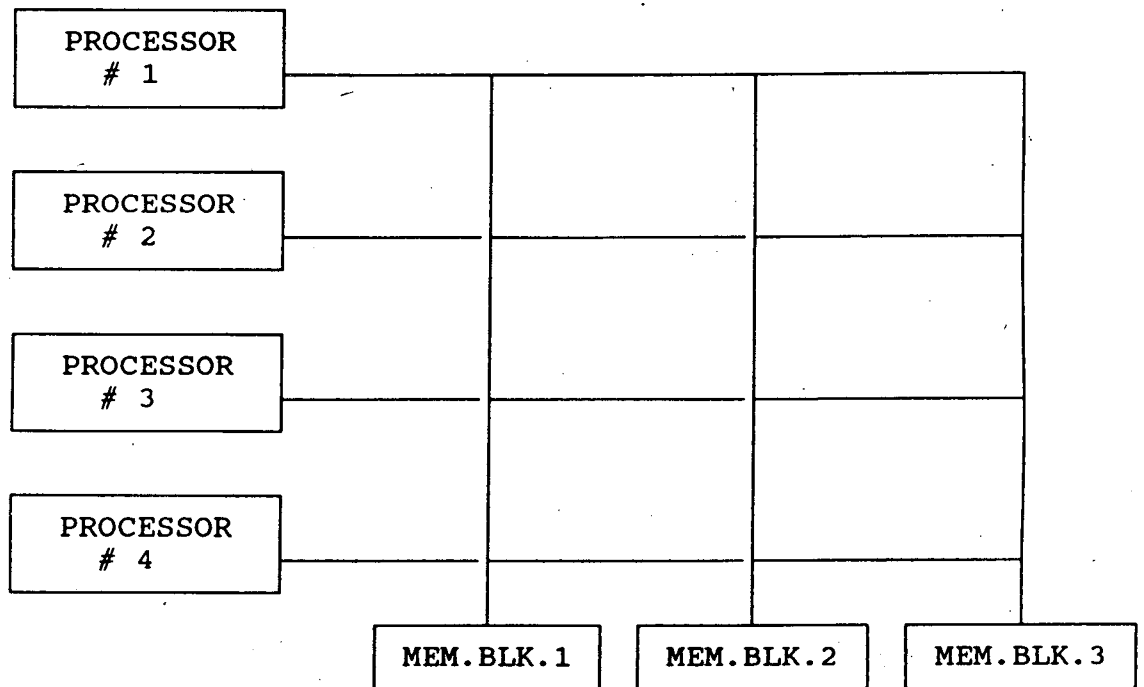


The processors operate synchronously:at each step, all processors execute the same instruction each on a different datum. The instruction could be a simple one (such as adding or comparing two numbers)or a complex one (such as merging two lists of numbers). Sometimes it may be so necessary to have only a subset of the processors execute an instruction. This information can be encoded in the instruction itself, thereby telling the processor whether it should be active(and execute the instruction) or inactive (and wait for the next instruction). There is a mechanism, such as a global clock, that ensures lock-step operation.Thus processors that are inactive during an instruction or those that complete execution of the instruction before others may remain idle until the next instruction is issued. In this type of systems it is always desirable to have good communication facilities amongst the processors in order to exchange data or intermediate results.This gives rise to two subclasses of SIMD computers they are: **shared memory communication type** and **interconnection network type.**

2.1.3.1 Shared Memory Type

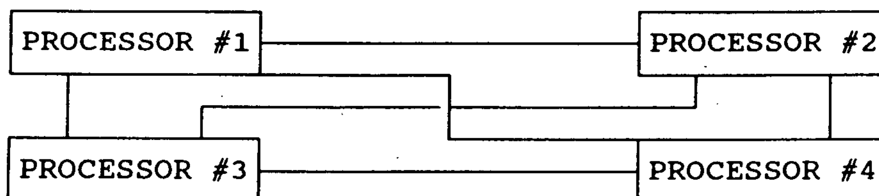
In this case during execution of a parallel algorithm, the N processors gain access to the shared memory for the reading of the input data, for reading or writing intermediate results,and for writing final results.The basic model allows all processors to gain access to the shared memory

simultaneously if the memory locations they are trying to read are different. However, the class of shared memory SIMD computers can have many further classifications.



2.1.3.2 Interconnection Network type

Here the model is constructed such that each node can communicate with each node through a direct link. Hence several processors can communicate simultaneously amongst themselves though there is some limitations involved in it.



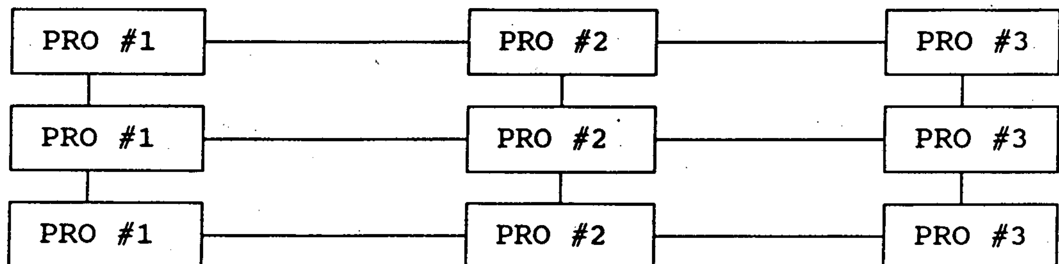
Simple Networks for SIMD Computers

i. *Linear Array*

The simplest way to interconnect N processors is in the form of one dimensional array. Here each processor is connected to it's neighbouring processor through a two way communication link as shown below.

ii. *Two dimensional array*

Two dimensional network is obtained by arranging the N processors into an m*m array, where N = m*m. This network is also known as **mesh**.



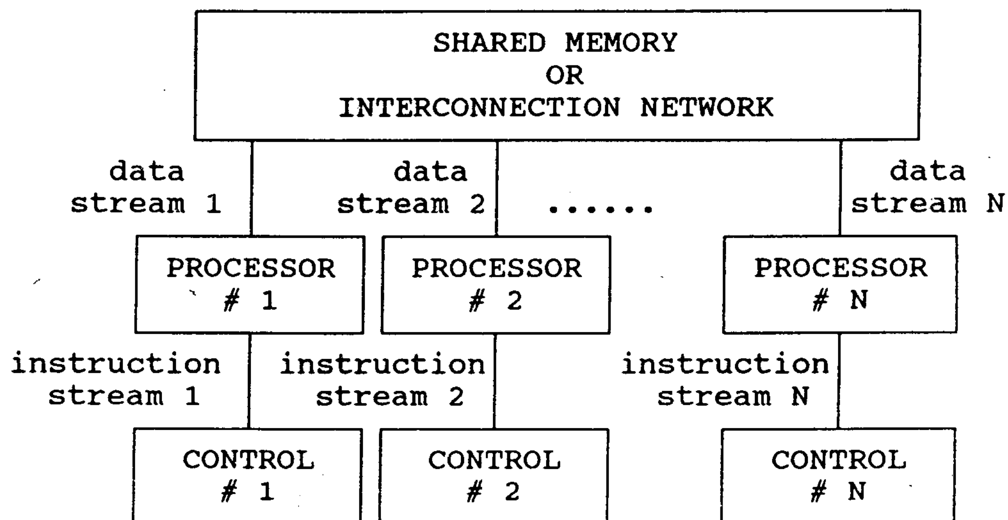
iii. **Tree Connection**

Here the processors form a complete binary tree. Such a tree has *d* levels, numbered 0 to d-1,and N =2^d-1each of which is a processor. The root processor have no parents and the leave processors have no children.

There are many other connection networks such as cube, ring and shuffle etc. which are also used nowadays.

2.1.4 MIMD Computers

This class of computers is the most general and most powerful in our paradigm of parallel computation that classifies parallel computers according to whether the instruction and/or data are duplicated. Here we have N processors, N streams of instructions, and N streams of data. The processors used here are of the same type used in MISD computers in the sense that each processor has it's own control unit in addition to it's local memory and the arithmetic and logic unit (ALU).This makes these processors more powerful than their counterparts used in SIMD computers.



Each processor operates under the control of an instruction stream issued by it's control unit. Thus the processors are potentially all executing different programs

on different data while solving different subproblems of a single problem.This means the processors typically operate asynchronously. As in SIMD computers, communication between processors is performed through shared memory or interconnection network. MIMD computers sharing a common memory are often referred to as *multiprocessors* (or tightly coupled machines) while those with an interconnection network are known as multicomputers (loosely coupled machines). Multicomputers are sometimes referred to as **distributed systems**. The distinction is usually based on physical distances separating the processors and is therefore subjective.

2.2 Programming MIMD Computers

MIMD model of parallel computation offers the most general and powerful mode of computation possible. Computers in this class are used to solve in parallel those problems that lack the regular structure required by the SIMD model. Asynchronous algorithms are difficult to design evaluate and implement. In order to appreciate the complexity involved in programming MIMD computers, it is important to distinguish between the notion of a **process** and that of a **processor.** An asynchronous algorithm is a collection of processes some or all of which are executed simultaneously on a number of available processors. Initially, all processors are free. The parallel algorithm starts it's execution on an arbitrarily chosen processor. Shortly after it creates a number of computational tasks or processes, to be performed.A process

thus corresponds to a section of the algorithm. There may be several processes associated with the same algorithm section, each with a different parameter.

Once a process is created, it must be executed on a processor.If a free processor is available, the process is assigned to the processor that performs the computation specified by the process, else the process is queued and waits for a processor to be free.

When a processor completes the execution of a process, it becomes free. If a process if is waiting to be executed, it can be assigned to any processor just freed, else if no process is waiting, the processor is queued and waits for a process to be created.

The order in which the process are executed by processors can obey any policy. The availability of a processor is sometimes not sufficient for the processor to be assigned a waiting process. An additional condition may have to be satisfied before the process starts. In the same way, if a processor has already been assigned a process and an unsatisfied condition is encountered during execution, then the processor is freed. When the condition for resumption of that process is later satisfied, a processor is assigned to it. The above mentioned are few of the scheduling problems that characterize the programming of multiprocessors. Finding efficient solutions to these problems is of vast importance if MIMD computers are considered to be useful in the long run. The vital difference between this and the SIMD computers is

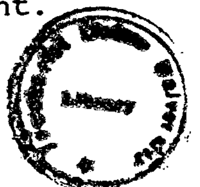
that none of these above said scheduling problems arise on the less flexible but easier to program SIMD computers.

While programming a distributed-memory parallel computer not only the question of communication between processes but also the distribution of software processes over the hardware processors is also of vital importance. Much of the current research is devoted to combining these two tasks into a single automated operation.In the absence of a universal, efficient solution to this problem, there is a strong argument for separating the two activities completely so that the logical structure of the program is unaffected by the **physical topology** of the processor network on which it is executed.

The version two of the **Occam** language is only partially successful in maintaining this separation; the distribution of logical channels over physical transputer links is kept in the header of the program (along with the global constants declarations and like), but the distribution of processes over the processors must be done in the body of the program proper.

In the Meiko's parallel programming environment, CSTools, maintains this separation of activities for C programs. It puts all the distribution information into a separate text file, called a **PAR** file. By changing this file one can redistribute a compiled program. The CSTools employs the distributed CSN (Computing Surface Network) to fool processes into thinking that all communication is point to point.

TH-5158



A message may be physically be routed via several intermediate processors, but CSTools hides it from the program and makes the programming easier just like programming in ordinary C under Unix.

The above can be illustrated through a simple example. For compiling a program for displaying "Computer Science" we may compile by giving command:

% *mcc -o computer computer.c*

and to execute it, we can give the command

% *mrun computer*

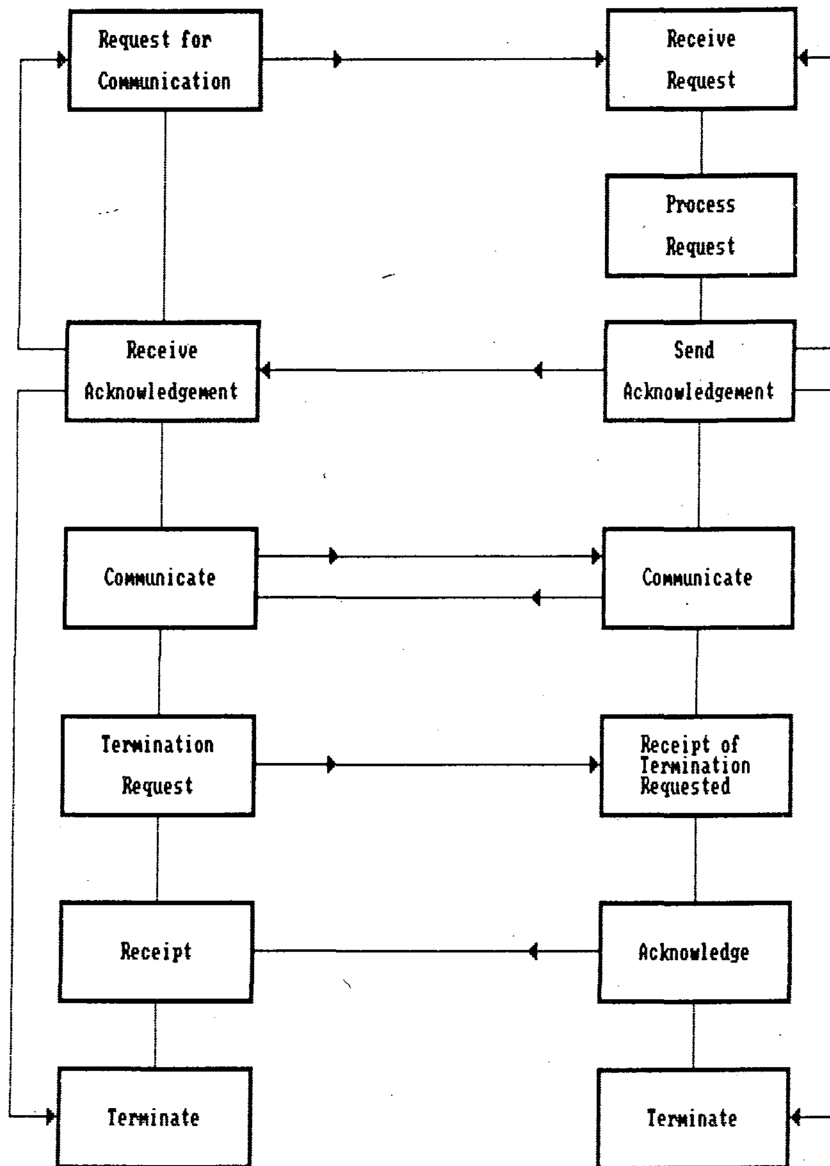
CSTools parallel loader program,*mrun*, performs the distribution of processes over the parallel processors. For a real parallel program, instead of giving *mrun* the name of the executable file, the name of the PAR file is given,which describes the placement of processes.The parallel C programs themselves contain no distribution at all, but communicate with each other via abstract entities called *transports*. The idea is illustrated through the simple example that uses two parallel processes: one to print "Computer," and the other to print "Science".

Listings shows the source code for the two processes, *computer.c*, and *science.c*. All the functions whose names begin with *csn_*are communication primitives from a CSTools library. Each process first initializes CSN with *csn_init()* and then calls *csn_open()* to create a connection the process and CSN. This connection is an object of type.

MESSAGE PASSING

Physical Link

Processor # 1 Processor # 2



```
/* Program computer.c */
   # include <stdio.h>
   # include <csn/csn.h>
   # include <csn/names.h>
   # include <cs.h>

   main (argc, argv)
   int argc;
   char* argv[];
   {
    Transport transport;
    netid_t world_id;
    int flag = 1;
    int status;
    csn_init();     /* Initialize the CSN */

    status = csn_open (CSN_null_ID, &transport);
    if (status != csn_ok)
     cs_abort ("unable to open transport\n", -1);

    status = csn_lookupname(&world_id, "Worldtransport\n");
    if (status != csn_ok)
      cs_abort ("unable to lookup WorldTransport\n", -1);

    printf("Computer"); fflush (stdout);
    csn_tx (transport, 0, world_id, &flag, sizeof(flag));
   }

/* Program science.c */
   # include <stdio.h>
   # include <csn/csn.h>
   # include <csn/names.h>
   # include <cs.h>

   main (argc, argv)
   int argc;
   char* argv[];
   {
    Transport transport;
    int flag = 1;
    int status;
    csn_init();     /* Initialize the CSN */

    status = csn_open (CSN_null_ID, &transport);
    if (status != csn_ok)
     cs_abort ("unable to open transport\n", -1);

    status = csn_register(&world_id, "Worldtransport\n");
    if (status != csn_ok)
      cs_abort ("unable to register WorldTransport\n", -1);

    csn_rx (transport,NULL, &flag, sizeof(flag));
    printf ("Science\n");
   }
```

Transport, and each transport has an address called a Net ID. The sender of the message (computer.c in this example) must know the Net ID of the intended receiver's transport. To make this possible, the receiver (science.c) registers it's transport with the CSN name service by calling *csn_registername()*, and then the sender can look up this name by calling *csn_lookupname()* and retrieve it's Net ID. It's rather like getting one's name into the telephone directory so that anyone can look into it and contact.

2.3 Analysis of Parallel Algorithms

Once an algorithm for a new problem has developed, it is usually evaluated using the following criteria: running time, number of processors used, and cost. Besides these standard matrices, a number of other technology related measures are sometimes used when it is known that the algorithm is destined to run on a computer based on that particular technology.

2.3.1 **Running Time**

As the speed is emerging to be the main reason behind the growing interest in the field of parallel computers, the most important measure a parallel algorithm is therefore the running time. According to one of the pioneers in the field of parallel processing Selim G. Akl the **running time** is defined as *the time taken by the algorithm to solve a problem*

on a parallel computer, that is, the time elapsed from the moment the algorithm starts to the moment it terminates. If the various processors do not begin and end their computation simultaneously, then the running time is equal to the time elapsed between the moment the first processor to begin computing starts and the moment the last processor to end computing terminates.

In evaluating a parallel algorithm for a given problem, it is quite natural to do it in terms of the best available sequential algorithm for that problem. Thus a good indication of the quality of a parallel algorithm is the ***speed up*** it produces. This is defined as

Speedup=

$$\frac{\textit{worst-case running time of fastest known sequential algorithm for the problem.}}{\textit{worst case running time for the parallel algorithm}}$$

2.3.2 Number of Processors

The second most important criterion in evaluating a parallel algorithm is the *number of processors* it requires to solve a problem. It costs money to purchase, maintain, and run computers. When several processors are present, the problem of maintenance, in particular, is compounded, and the price paid to guarantee a high degree of reliability rises sharply. Therefore, the larger the number of processors an algorithm uses to solve a problem, the more expensive the solution becomes to obtain. For a problem of size n, the

number of processors required by an algorithm, a function of n, will be denoted by p(n). Sometimes the number of processors is a constant independent of n.

2.4 Parallel Software Design Issues

2.4.1 Portability

Parallel programs are designed to be made more easily portable so that investment in their design and implementation can be amortized across a wide range of machines.Current trends in parallel processing hardware makes this goal of architecture independence especially difficult to achieve, since technological changes seem to oscillate between message passing based architectures and shared memory architectures. The programming methods used in these parallel architecture variants are typically quite different.

2.4.2 Design (Correctness)

Large projects require hierarchical designs. Unfortunately to understand the interaction of communicating processes require that their specification includes not only their data behaviour, but also their control behaviour. Reasoning about the interaction of much complex specifications is tantamount to reasoning about the final code. To design more effectively a technique must be found out to raise the level of behavioural abstraction Functional models have succeeded here through the use of functional composi-

tion. The traditional functional composition is deterministic and require single assignment semantics to enforce referential transparency.

2.4.3 Implementation

In traditional implementation of parallel programs, there is often no way of ensuring that the code implements designer's intentions. For example, a simple typographical mistake during coding can cause two processes to communicate when they should not, leading to disastrous, unpredictable consequences. If the design specifications could somehow be fed directly to the language processor, this unintended communication could be diagnosed syntactically. In order to be viable, the design must be formally defined as a computer language.

CHAPTER THREE

CHAPTER THREE

PERFORMANCE ANALYSIS METHODOLOGIES

3.1 Importance of Performance Analysis

Suppose we have a weather forecasting system which predicts tomorrow's weather latest by tomorrow evening then it is better not to use that system at all, or in the today's high tech warfare if a missile intercepting system detects the missile after it hits the target then the system is also useless. Here comes the word performance. *Performance analysis* is applied in almost all fields in today's world to determine the suitability of the systems in the fields in which they are supposed to be used. While analysing performance we generally prepare a model of the actual system and monitor it's behaviour for different inputs. These inputs are generally similar to the types that are actually in use.

In the computers, performance analysis is invariably done in selecting a system. The first step involved is to list and examine the important system device parameters: the capacity and cycle time of cache and main memory, the speed of the CPUs, the access time and data transfer rate (band width) and the types and characteristics of terminals and communication equipments. We also need to know software components: the job scheduling algorithm, the disk and drum scheduling algorithm, the sizes of page and block, and the file organization. Further, we may want to know the amount of

traffic (loads) anticipated for each of these components: the job arrival rate, the amount of CPU time (or instructions) per job, the memory space requirements, the page fault rate, the request rate on drums and the required rate of data transfer rate between the auxiliary storage and main memory. In multiprocessor systems this becomes more complicated. The data transfer rate between adjacent processors, blocking and freeing of interprocessor communication links, use of limited resources comes into picture.

3.2 Task Partitioning & Task Coalescence

Performance is one of the most important factors that needs to be considered during the design, configuration and development of a distributed real time computer system. To obtain the optimal system performance, which includes verifiably correct functionality, minimal resource requirement and high reliability, the process of task partitioning and allocation play an important role in the design process. Task partitioning is the decomposition of the total task into subtasks according to a specific partitioning criterion. In the task allocation process, the partitioned subtasks are allocated to processors available in the system, such that an objective cost function is minimized subject to certain constraints imposed by the application or environment. Task coalescence is the composition of all user predefined task modules into a set of subtasks to achieve a specific performance goal. The subtasks represents a group of user defined

software modules. All modules in a subtask will be assigned to the same processor during the allocation process. The total task is represented by a set of disjoint subtasks. The optimal set of such subtasks is then used for task allocation.

Many partitioning algorithms with constraints on both task structure and system structure have been proposed. However most approaches have ignored one or more important factors such as queuing delay, processor load, communication link load and synchronization delay that are inherent in the real time applications. The coalescence of a given set of task modules to meet performance requirements has not been thoroughly investigated.

The evaluation and optimization are made in terms of the minimal processing power(processing power is defined as a fraction of the available processing time in a processor which may be assigned to a task) required for the total task, while guaranteeing the satisfaction of user requirements. A two queue network model is used for estimating the queuing and communication delays of a task module. The response time of a task module includes a synchronization delay, job processing time and data communication time. The synchronization delay which is caused by join operators is estimated under certain assumptions. The job service time and communication time are calculated by using queuing models.

An analytic model should be sought wherever possible, since it can evaluate the performance with minimal efforts and costs over a wide range of choices in the system param-

eters and configurations. Even with simplifying assumptions and decompositions, however, the resultant analytic model is often not mathematically tractable. Then the only alternative for predicting the performance of a nonexisting system is a simulation. The term simulation has a number of connotations. In the discussion a simulation means a numerical technique for conducting an experiment (by a digital computer) of a system evolving in time. Therefore, in a simulation the concept of time is explicit. A simulation model describes the dynamic behaviour of a system, even when the system analyst may ultimately be interested in only the mean value of some measure (e.g. CPU utilization, the response time) in the steady state.

The structure and complexity of a simulator depends on the scope of the simulation experiment. The hierarchical structure should be adopted as much as possible in the construction of a simulation model also, though the motivation here is different. There is at least two features that make such a structure attractive. First, a hierarchical (or more generally decomposable) structure allows modularization of a simulation program into a set of subprograms. Modularization leads to a flexible structure of the program, so that further extensions and changes are easily handled. Second, an ingenious use of the hierarchical structure may shorten a simulation run time substantially. In general, the length of a simulation run is determined by the required accuracy of simulation estimates and the amount of correlation span (or,

equivalently, the magnitude of transient time) of the stochastic process observed in the simulation outputs. In the model structure (of figure 3.1 given below) the inter-event time in the micro level model is in micro seconds.

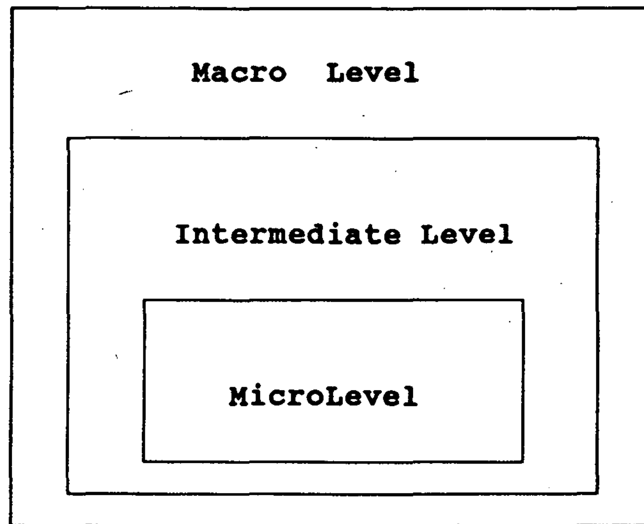


figure 3.1

The number of events observed over the simulated time of say, one second will be of order 10^6; this subsystem easily reaches its steady state within that period. During the same period, the number of events that take place at the intermediate model is in the order of 10^3. The stochastic process that characterizes the intermediate level model may possibly reach its steady state, but the sample size of 10^3 is perhaps not large enough to allow reasonably accurate estimate of a chosen performance measure. But as the macro level model, it is quite evident that one second of simulated time is too short to understand the system behaviour, since the inter event time itself is in the order of seconds.

Perhaps a simulated time of 10 ^{3} seconds or more will be required at this level to obtain an accurate estimate of all the overall performance measure. IF we were to run this simulator in its entirely over the period say 10^3 seconds in simulated time the total number of events observed at the micro level would amount to the order, of 10^9 events. Note that the actual length of computer running time for the simulation experiment is governed not by the length of simulated time, but by the total number of events handled. What the simulator performs is essentially to record all the system changes caused by the individual events. Therefore a brute force simulation often leads to an extremely costly experiment, but this is unfortunately the way in which most simulators have been structured in the past.

A more efficient approach to the simulation effort is to run different submodels separately, thereby avoiding the waste of running the micro level model for such a long period. Interfacing a lower level model to a higher level model should be achieved through summarized statistics, such as a scaling constants and service time distributions. Since the equilibrium state solution of a model of a given level depends on its surroundings we must have separate runs of the model for different sets of parameters that determine its surroundings. For example if the intermediate level model of the figure given represents a multiprogramming model we need to run the simulator of that level for different values of the degree of multiprogramming. These simulations will determine the whole range of effective processing rates that the

individual jobs receive under different congestion environments. The values of the effective processing rates are then used as parameters of the macro level model.

The above decomposition formulation naturally leads to the notion of what is sometimes called hybrid modelling :a combination of analytical procedure and simulation. So long as the interfaces between different levels or submodels are clearly established the mixing of analytic and simulation techniques should present no technical problems. In fact the approach deserves special attention, since it allows us to take the best of both worlds, the efficiency of analytic modelling and the realism of simulation modelling.

Simulation models for computer systems can be further classified as either trace driven simulation or self driven simulation. A trace is a stream if major events observed in an operational system, recorded with the time of their occurrences. Like a benchmark program a trace should be selected from a representative segment of the system workload. However a benchmark is a program that is independent of the system in question, whereas a trace is a result of both the chosen program and the machine that executes the program.

In the self driven simulation the concept of a probabilistic sequence of resource demands presented by jobs is introduced. An advantage of the probabilistic model over the trace driven model is that since the event stream is generated artificially it may be completely understood by the analyst, furthermore the workload parameters are adjustiflable.

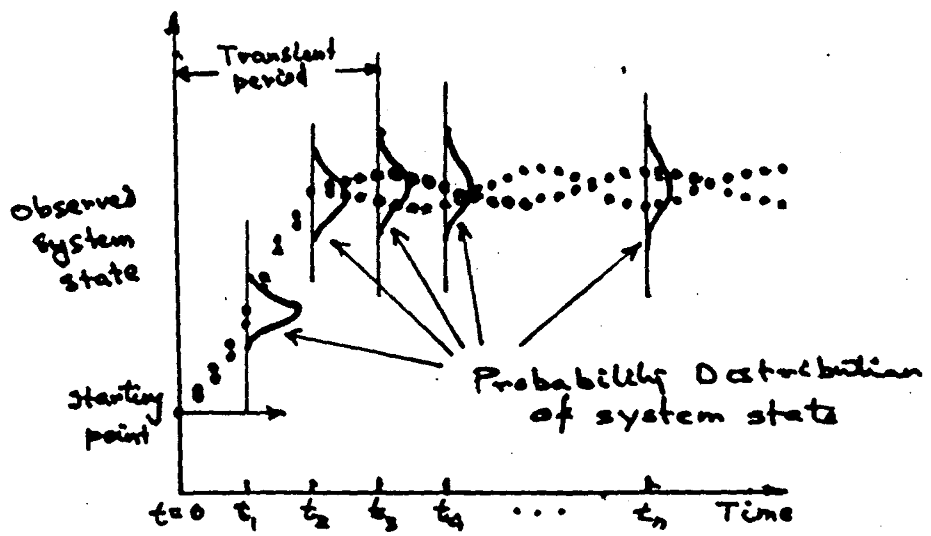


fig 3.2

In this sense, a probabilistic input model is to trace driven model as a synthetic job is to a benchmark. Self driven simulation involves the use of random number sequences and in this regard it is similar to the Monte Carlo method: a numerical technique for solving a nonprobabilistic mathematical problem by introducing a random variable whose mean or distribution corresponds to the solution of the original problem. In fact these two numerical techniques have a great deal in common: The objective of probabilistic simulation can usually be formulated as a mapping from a random vector (a sequence of random variates) to a scalar value of some performance measure as represented in the figure 3.2

For this reason many of the variance reducing techniques developed in the Monte Carlo method are applicable to variance reductions in stochastic system simulations.

3.3 Discrete Event Simulation

A discrete event simulation is represented by some set of data, called the system state, which contains all the information required to characterize the system state at one point in time. The state remains unchanged until some event occurs that causes a discrete change in the state . In other words state transitions are done via executions of series of events with times when they occur. Execution of an event modifies a subset of a system state, which moves the system to a new state. Execution of any event can give rise to any

number of events with later timestamps. A simple discrete event simulation algorithm is given below,

/ Let ϵ be the next event to be executed in the sequential simulation*/*

1.execute ϵ

/ Let e.E be the set of events scheduled due to the execution of e */*

2.for all e'ϵ e.E do

3.insert e' into the event list according to the timestamp order;

end while

Since most discrete event simulation are time consuming, several approaches have been proposed to speed up the simulation process. A popular approach is for multiple computers to cooperate to execute a simulation run. It is important to analyze the inherent parallelism of the simulation application before this is applied.

Berry and Jefferson and Livny proposed a simple technique called **critical path analysis** to study the inherent parallelism of simulation applications. This technique can also be used to evaluate the performance of existing parallel simulation protocols. From the speed-up figures, it is difficult to see whether the parallel simulation protocols are efficient. Critical path analysis indicates that the maximum speed up that can be achieved in this benchmark is 3.67. Thus, the speedups obtained by different scientists are actually quite respectable. The algorithms are easy to imple-

ment although the correctness proofs are not trivial and can be integrated with simulation languages. Another advantage of these algorithms over previous attempts are that this can be used to study load balancing under different event scheduling polices.

3.4 CRITICAL PATH ANALYSIS

The idea of parallel simulation is based on the following observation: If two events are independent of each other, they can be executed in parallel. Two events e and e', are independent if execution of e modifies the same subset of state variables (and the modified variables have the same values), no matter whether e' is executed before or after e, and vice versa. In a parallel simulation, the simulated system is partitioned into N subsystems. Subsystem i consists of a subset of state variables S, such that

$$S_i \cap S_j = 0 \quad \text{for } 1 \le i \neq j \le N, \text{ and } \quad \bigcup_{1 \le i \le N} S_i = S$$

where S is the set of state variables. These subsystems are concurrently simulated by a set of processes that communicate by exchanging time stamped messages. The events scheduled for process i can modify state variables in S_i. After the simulated system is partitioned, execution of events follows two sequential constraints.

Constraint 1. If two events are scheduled for the same process, the event with smaller timestamp must be executed before the one with larger timestamp.

Constraint 2. If an event executed at a process results in the scheduling of another event at a different process, then the former must be executed before the latter.

Partitioning the simulated system into subsystems is not a trivial task. If there are too many subsystems, the communication overhead due to Constraint 2 may outweigh the benefit provided by parallelism. On the other hand, if there are too few subsystems, independent events may be executed sequentially due to Constraint 1. Several studies are devoted to the partitioning problem.

Based on Constraints 1 and 2, an event precedence graph is built for each parallel simulation. Each vertex of the graph represents an event and each edge represents a communication. An event execution time is associated with each vertex.A communication delay is associated with each edge.Since the graph is acyclic, a maximal weighted path can be found. This path is called the *critical path* and its cost is the minimal time required to finish the execution of the parallel simulation.

Examples of an event precedence graph are given in the figure(3.3). A dashed arrow represents the scheduling constraint of Constraint 1 and a continuous arrow represents the process communication due to Constraint 2. The event precedence graph for sequential simulation is given in figure (3.3a) and the graph for 3 processes is in figure (3.3b), where events e1 and e7 are scheduled for process 1, e3, e4 and e5 are scheduled for process 2, and e2, e6 and e8 are

scheduled for process 3. The cost for critical path can be derived quantitatively as follows. Let g_e be an event such that event e is scheduled due to execution of g_e. In figure (3.3b) g_{e7}=e5. An event e is prescheduled if g_e does not exist. In figure (3.3) events e1 and e2 are prescheduled. Let p_e be an event such that both events e and p_e are scheduled for the same process, and the execution of p_e is followed by the execution of e. In figure (3.3b) p_{e7} = e1. Let $\alpha(e)$ be the earliest time when e's execution starts. Let $\tau(e)$ be the execution time for the event e. Let $\alpha'(e)$ be the earliest time when e's execution completes. If every process is executed by a dedicated processor then

$$\alpha'(e) = \alpha(e) + \tau(e)$$

Let $\sigma(e)$ be the time to schedule event e. IF g_e and e are scheduled at different processes then $\sigma(e)$ represents the message sending delay. Otherwise $\sigma(e)$ is assumed to be 0 in the study. From constraints 1 and 2

$$\alpha(e) = \begin{cases} 0, & \text{if neither } g_e \text{ nor } p_e \text{ exist.} \\ \alpha'(e), & \text{if } g_e \text{ does not exist} \\ \alpha'(e) + \sigma(e), & \text{if } p_e \text{ does not exist} \\ \max[\alpha'(p_e),\ \alpha'(g_e) + \sigma(e)], & \text{otherwise} \end{cases} \qquad .....(a)$$

Let T_p be the cost for the critical path (i.e. the execution time of the optimal parallel simulation). Let T_s be the sequential execution time. Then T_p and T_s are expressed as

$T_p = \max \alpha'(e)$, and $T_s = \Sigma\ \tau(e)$

The optimal parallel simulation time is computed based on (a). However (a) is only adequate for the case when every process is executed by a dedicated processor. If the number of processors P is less than the number of processes N, then T_p is also affected by process assignment and event scheduling. If more than two processes are assigned to a processor, (3.3) only represents the time when an event is available for execution. More than two events from two different processes may be available for execution at a processor at the same time. An event scheduling policy is required to determine the next event to be executed. Let P_k be the set of indexes of processes mapped into processor k (i.e. $i \in P_k$ if process i is mapped to processor k) .Let $e'_j(t)$ be an event scheduled for process j such that for all events scheduled for process j $e'_j(t)$ is the next event to be executed after time t(and at time t, processor k is available to execute the next event).

In the critical path analysis is performed as follows. The first thing is sequential simulation and taking a trace of the events executed. The trace is then transformed into a event precedence graph. Finally the cost of the critical path in the graph is computed. The critical path analyzer proposed by Livny is integrated with a specific simulation, DISS, and the cost for the critical path is computed along with the execution of the simulation. Thus no event trace is required and no explicit construction of a precedence graph is necessary. Consider the example in figure (3.3b). If the event

Process 1

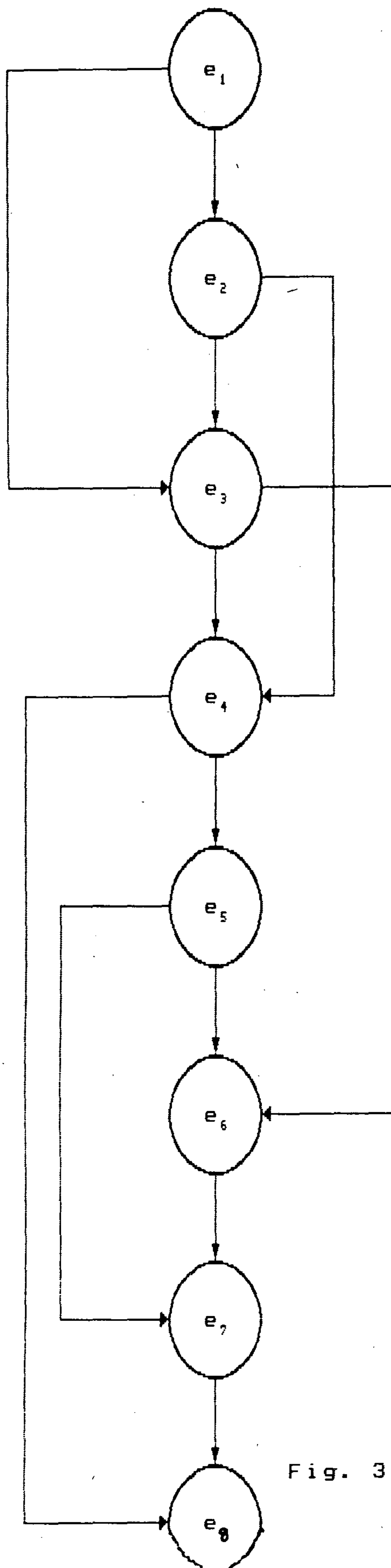


Fig. 3.3a

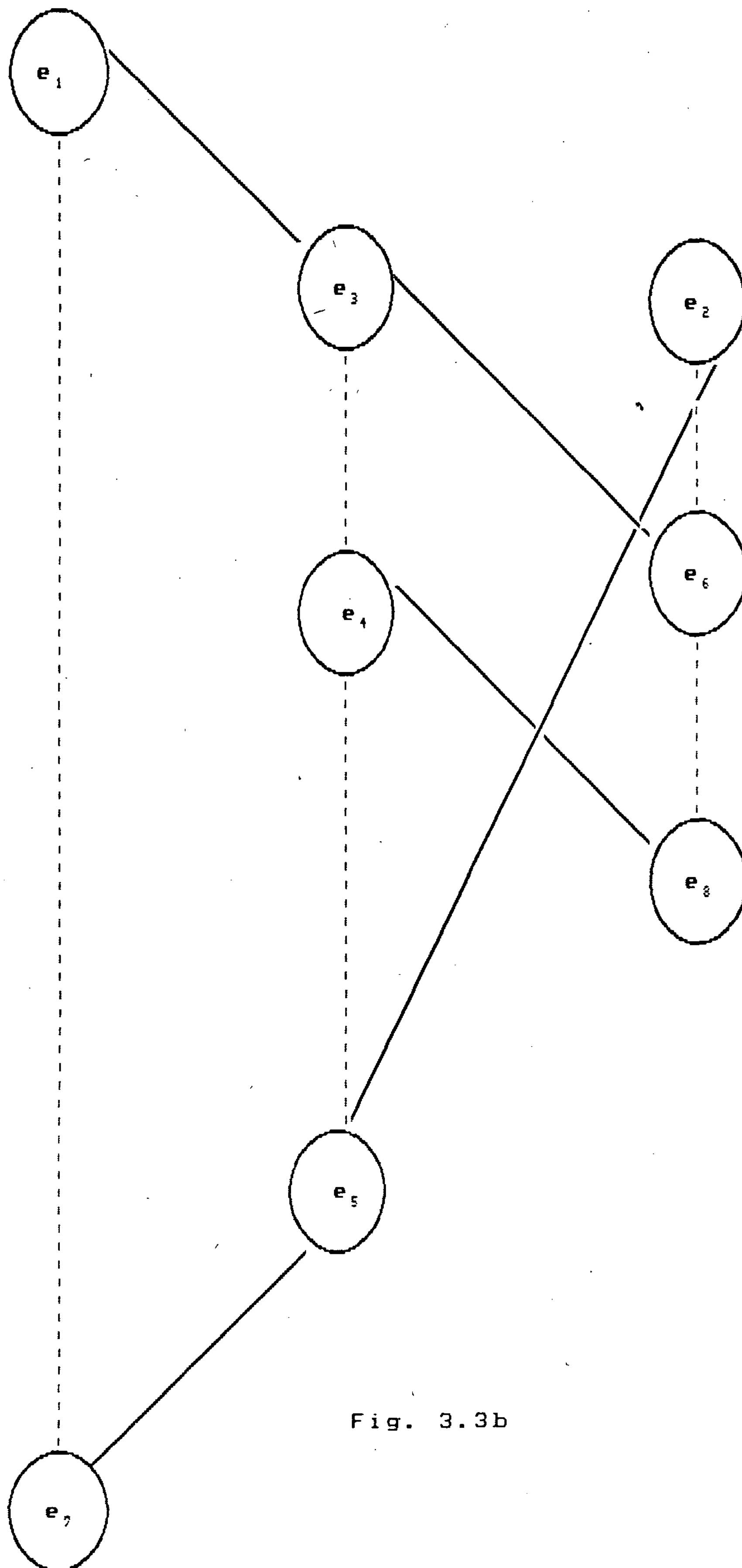


Fig. 3.3b

execution time is 1 unit and the communication cost is 0, then the execution time for the optimal parallel simulation for the first three events is 2 units. However, the value computed from Livny's algorithm is 1 unit.

The above two approaches are primarily designed for the case when P = N. Now a better critical path analyzer is described which can be integrated with the sequential simulation program. The algorithm referred to as **Algorithm 1** is designed under the assumption that every process is executed by a dedicated processor.The processes mapped into a processor are considered a *super* process, and all events are executed in the timestamp order at the processor.

/ initialization*/*

0a for all i do $T_i = 0$

0b for all e pre-scheduled in the event list do

0c $e.\alpha = 0$

end for

/ the main loop*/*

while not complete do

*/ * Let e be the next event to be executed in the sequential simulation */*

1a execute e

/ Let e be scheduled for process i */*

1b $T_i = max(T_i, e.\alpha) + \tau(e)$

/ Let e.E be the set of events scheduled due to the execution of e */*

2 for all e' ϵe.E do

3a e'.α = T_i + σ(e')

3b insert e' into the event list according to the timestamp order,

end for

4 T_S = T_S + τ(e)

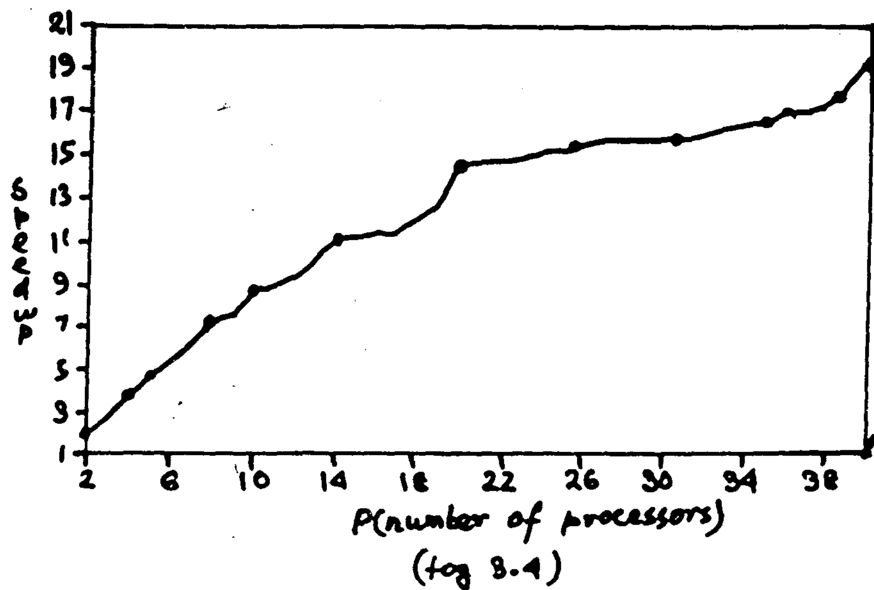
end while

Let event e be scheduled for process i. In this algorithm, e.α represents the time when event e arrives at process i. The value of T_i after the execution of Line 1b is the completion time of e's execution in the optimal parallel simulation.

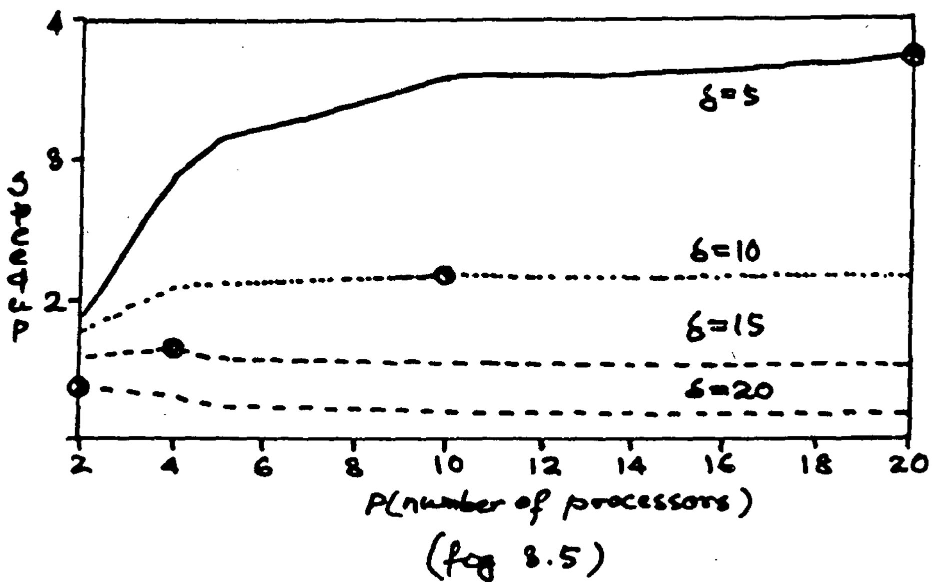
The following issues should be taken care of when the critical path analysis is applied to simulation application.

3.4.1 Selection of event execution time and communication cost:

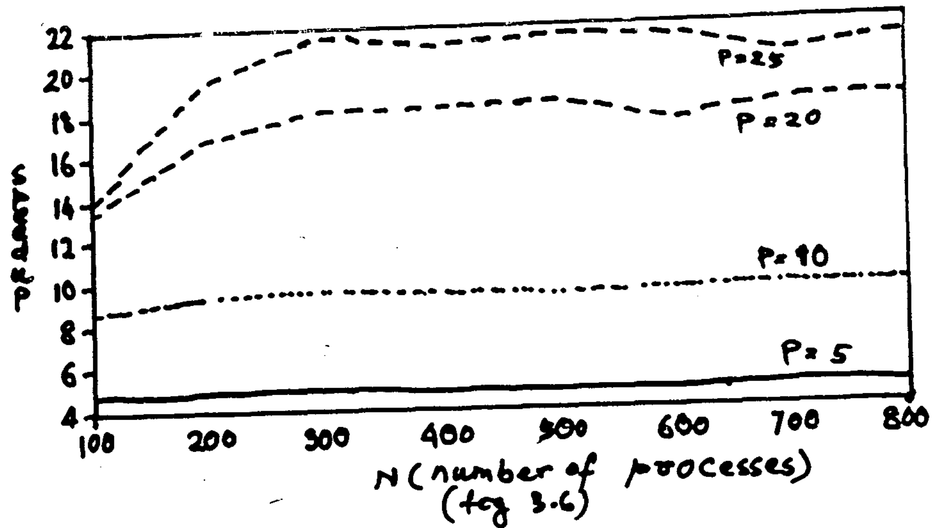
In many examples only a few types of events exist in the simulation. For each type of event the execution times are fixed and can be easily determined through measurement. If the event execution time varies from one to another then it is needed to sample the event execution times and determine an event execution time distribution to be used in critical path analysis. Sometimes an event execution time is too short to be measured. In such a case the execution of an event can be repeated several times, then the average value can be found.



(fig 8.4)

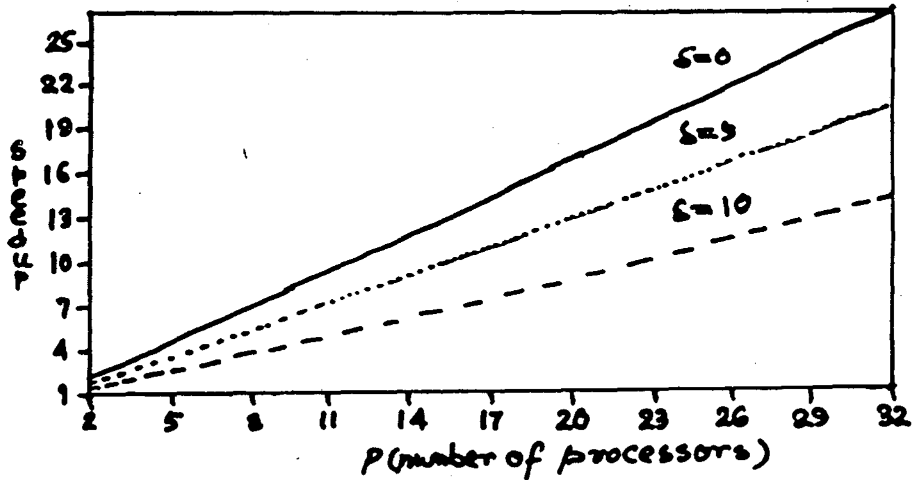


(fig 8.5)



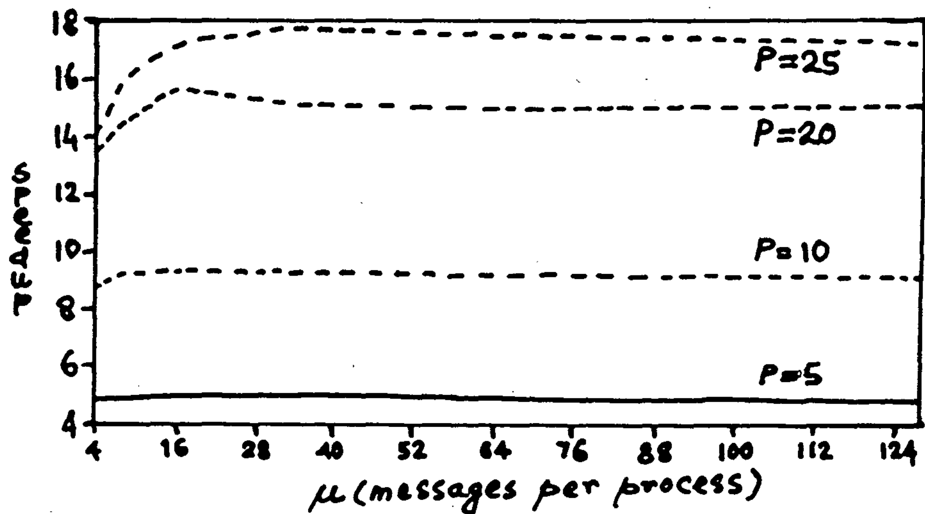
(fig 3.6)

INCRESING PROBLEM SIZE



(fig 3.7)

INCRESING PROBLEM SIZE WITH FIXED PROCESS TO PROCESSOR RATIO



(fig 3.8)

INCRESING MESSAGE DENSITY

As a first approximation of critical path analysis τ can be assumed as constant. The communication cost is usually obtained from the targeted architecture.

3.4.2 Number of Events to be Processed:

A large number of events must be processed in critical path analysis before a reliable speed figure can be obtained. For steady state simulation experiments indicate that the reliable speed up figure can be obtained only after the transient effect of the simulation disappears.

3.4.3 Load Balanced Process Assignment:

Figure 3.4 shows the effect of load balancing. The circles represent the speed ups of the balanced points where P divides N It is observed that for a small P, the distance between two balanced points is short and the unbalanced points in between are not significantly affected by the unbalanced load. For large P it is most beneficial to add extra processors if it is close to the next balance point.

3.4.4 Interactions Between the Number of Processors and the Communication Cost:

The number of processors for parallel simulation must be selected to balance the effects the constraints 1 and 2 in order to yield the maximum speed up. It is clear that if the communication cost σ is high assigning extra processors to a parallel simulation may degrade speed up. Figure 3.5

shows how the interactions between P and σ affect speed up. The ∞ symbols mark the maximum speed ups. It is observed that when the communication cost is 20 times of an event execution time, the maximum speed up occurs when P = 2 and adding extra processors to the parallel simulation only degrades the performance.

3.4.5 Increasing the problem size:

For a fixed number of processors if the problem size increase the inherent parallelism also increases. Figure 3.6 indicates that if N.. P speed up of P can be expected. This observation supports Nicol's conclusion that a simple parallel simulation protocol can yield good speed up if the problem size is sufficiently large. It is noted that when P is large a much larger N is required to fully exploit processor power.

3.4.6 Increasing the problem size with fixed processes to processors ratio:

When both the problem size and the number of processors increase i.e. N/P is constant then the number of events executed at a processor does not change statistically, but the number of processors to be communicated increases. Figure 3.7 shows that the speed up increases linearly if the problem size increases with fixed N/P ratio. The above observation implies that the number of processors to be communicated with a processor does not affect the inherent parallelism.

3.4.7 Increasing Message Density:

By increasing the number of message per process, the work load to the simulated system is increased. In figure 3.8 it is observed that when δ increases speed up increases and then slowly decreases. Similar phenomena was observed for conservative parallel simulation protocols.

CHAPTER FOUR

CHAPTER FOUR

IMPLEMENTATION

In the project simulation of an multiprocessor environment is done to evaluate the performance of different standard computations under various topologies. A totally new topology **Z_circle** is found to be very much fault tolerant and also easy to upgrade (upgrade here means adding of new processors). All the standard topologies like **Bus, Ring, Torus, Hypercube, Mesh & Tree** are considered. The simulation of this project is done in **C** language on **DEC VAX 11/780** under **VMS** environment.

In the **z_circle** network the processors are connected in a z-ladder fashion. Here both the two processors at the ends are connected in a circular fashion. A major advantage is that in most of the transputers four links are available, and all the four links are utilised here. Another advantage is that the shortest path determination overhead is minimized by only simple subtraction. It is also considerably fault tolerant.

The input to this simulator is given after balancing load with a suitable load balancing technique. Here at each processor two queues are maintained: a **ready queue**, and a **communication queue**. In the beginning the ready queue at each processor contains all the processes assigned to that processor and the communication queue is kept empty. The round_robin job scheduling technique is followed at each

processor, i.e; each process at a processor is given a time slice for execution. An execution cycle is followed by a communication cycle. In the communication cycle the processes requiring communication among themselves communicate. Before any of the two processes communicate first the links connecting them through shortest path is examined. If the path is found to be free then, it is made busy and the message is routed. As the path is made up of links a flag associated with each link enables efficient examining of the path, i.e; if the flag is 1 then the link is busy else it is free. A path is made busy by making the flags of all links 1 connecting the two processes which are to communicate.If the path is found to be busy i.e; any of the links connected in the path is 1 then, the process is put in the communication queue. A counter is maintained at each process to see the number of cycles it is executed. If after a communication cycle the process requires further execution then, it is put in the rear of the ready queue and it waits to get a slice of the CPU. When all the queue are exhausted then the program terminates. It also calculates the time of execution of each process in terms of processor time slice it also calculates time of completion of each queue. That is done by adding execution time of all the processes at each processor separately.

4.1 Important Parameters

sdl_array : this is an array of records consisting of four fields. They are

i) *processor_id* – this is defined as integer and contains the identification of each processor.

ii) *add_rq* – this contains the addresses of different processes at the ready queue of each processor.

iii) *add_cq* – this contains the addresses of different processes in the communication queue.

iv) *flag* – this stores the state of the processor in the running cycle (i.e; executed or not).

In C language the declaration for the above was done as shown in the next page.

```
struct schdl
    {
     int processor_id;
     int add_rq;
     int add_cq;
     int flag;
    } sdl_array [N];
```

oplist : This contains the details of communication for each process. A separate list is prepared for each process. This is defined as array of records, consisting of two fields, they are:

i) *processor_id* – this contains the processor identification.

ii) *process_id* – this contains the addresses of the processes with which the processes will need communication.

In C this is declared as:

```
struct list
    {
     int processor_id;
     int process_id;
    } oplist [K];
```

where K is defined as

*K = N*10;*

ALGORITHM

step 1. Select the topology to be used.

step 2. Connect as selected above.

step 3. Input the number of nodes **N** to be connected.

step 4. Check whether the topology with **N** no of nodes is permitted, if permitted then,

call procedure *connect (topo,N)* to connect the nodes in the topology selected,

else go to step 2.

step 5. Take the values of the *sdl_array* and *oplist* as input.

step 6. Call the procedures *reset_link()* and *reset_flag()*. Take the first non-zero element from the *sdl_array*. If there is no non-zero element in the *sdl_array* then go to 22.

step 7. Take the address of the ready queue for this element.

step 8. Choose the first process from the ready queue.

step 9. Take the address of the *oplist* from the process.

step 10. Take the first element from the *oplist*.

step 11. Take the processor name from the oplist and the from the *sdl_array* find out the ready queue address for this processor.

step 12.Mark the flag field of the *sdl_array* element selected, i.e; 1.

step 13. Check whether the first element of this queue is same as the process name of the *oplist* chosen,

if not go to step 19.

step 14. If the above condition is satisfied, then take the *oplist* address of the destination process chosen.

step 15. If the first element of the destination process *oplist* matches with the process name then call *test_link (start, dest)*.

Else go to step 19.

step 16. If the *test_link(start, dest)* returns one then,

i) increment the *counter(i)* for both the processes,

ii) delete the first elements of both the *oplist* chosen,

and iii) mark the corresponding *flag* field of the *sdl_array* of the destination process as selected (i.e; 1).

Else go to step 19.

step 17. Check the oplist of both the processes, if there is any operation left in any/both the process/es, then put the corresponding process/es in the rear of the ready queue of the corresponding processor, else remove the process/es from the ready queue.

step 18. Check the ready queue and the communication queue of the processor if both are empty, then put zero in the processor ID field of the *sdl_array*.

Go to step 20.

step 19. Put the process in the communication queue.

step 20. Choose the next non-zero element from the *sdl_array*.

step 21. If there's no non-zero element left in the processor_id field of the *sdl_array* then go to step 5,

else check whether the *flag* field of the chosen processor is 1,

if it is 1 then goto step 20,

else goto 5.

step 22. Check the maximum of the *counter(i)* and the minimum of the *counter(i)*.

step 23. Add the *counter(i)* values of the processes at each processor and put them in *exe(j)* array (Where $0 \le j \le N$).

Go to step 25.

step 24. Terminate the program with the message "Topology entered with the given no of nodes is not permitted".

step 25. Find out the processor utilisation for each processor separately (i.e; calculated for *n*th processor by dividing the time used by processor n divided by total time taken for computation).

step 26. Calculate the system utilisation.

step 27. Calculate the speedup (the speedup is calculated by adding the execution time of all the processors and then dividing it by maximum value of *exe(j)*).

step 28. Report end and terminate the program.

4.2 Computations

In this model, in a cycle, the processor execute a computation step and after finishing they synchronise and perform data exchange the next page. If during execution of an algorithm, all the processors are performing computations in all cycles then the system utilisation is 1. However it is found that in some algorithms all the processors may not

participate in computation in all the cycles, as some processors may be waiting for the for results generated by some other processors.The value for such algorithms is less than one.

4.3 Performance Measures

System utilisation

In an execution cycle, all the processors may not participate in execution and may be idle throughout an execution cycle, waiting for results from other processors. The utilisation of the system in terms of the number of processors used in an execution cycle is quantified by the parameter S_u, which is referred to as system utilisation.

Consider an algorithm which is executed in **r** cycles on **P** processors. Suppose that in an execution cycle of t_1 time units, P_1 processors are used, in the next execution cycle of t_2 time units P_2 processors are used, and so on then,

$$S_u = (P_1*t_1 + P_2*t_2...+ P_r*t_r)/(P*(t_1+t_2+..+t_r))$$

Processor Utilisation

When the sub-domains assigned to different processors are not equal, then some processors finish computation earlier than others, and as synchronisation takes place at the end of every cycle, these processors wait for others to finish. This leads to idling and under-utilisation of some processors which is quantified by the parameter P_i^u for processor i. It characterises the load balancing of the system. Perfect

load balancing occurs when the sizes of the sub-domains assigned to all the processors are equal i.e, when $P_i^u = 1$, for i = 1,2,...,P (where P is the number of processors in the system).

Inter-Processor Communication Time

In a message passing multiprocessor, if $t_{start-up}$ represents the message start-up overhead or latency and t_{send} is the transmission time (which is inverse of the link bandwidth) k byte between two neighbouring processors, involves a time t_{comm} equal to $t_{start-up} + t_{send} * k$.

When the communication is not between two near neighbours, the communication time is estimated by assuming that it takes place in hops, and each hop corresponds to a near neighbour communication. The communication time between two processors separated by n hops is then equal to $n * t_{comm}$.

4.4 Assumptions

The model proposed here for performance predication assumes that all interprocessor communication times can be estimated *a priori* and that there are no unpredictable queuing delays in the system. An input file having two fields containing processor-ID name and process is available. Load balanced system is available. Any process can complete its message passing in one communication cycle if the route is free and the receiving process is ready. Any single mes sage passing will make the route busy for one communication cycle.

CHAPTER FIVE

CHAPTER FIVE

CONCLUSION

The field of parallel computers is a growing one. It is the ideal low cost supercomputing facility for a country like India. So in the future we will be seeing this field to grow like anything to replace the costly fast processor based supercomputers. Alongwith the growth of parallel computers, the performance, it's prediction and evaluation is going to be prime consideration in the selection of a system.

The level of details required in the validation of a simulator should depend on how that simulator is to be used in decision making. In other words, we must return to the principal objective of the simulation study and choose some performance measure that indicates whether the observation data generated by the simulator agree sufficiently with those of the real system. If the performance measure thus obtained is some mean value (e.g., CPU utilisation, the average response time), then the notion of *significance level* and *confidence interval* should be applied to quantify the statistical significance of the difference between measured and simulated effects. The *analysis of variance* technique can be used to test the hypothesis that the mean of a series of data generated by the simulator is equal to the mean of the corresponding observed data of the real system.

The model discussed here determines the performance of a static system. With some modifications, it can be made to work in dynamic environment also. The model discussed has got some limitations. It has got context switching time, which is a pure overhead. It's advantage is that it helps the smaller processes to complete execution by providing them time slices. In many cases the intermediate results provided by such processes is used by other processes to continue execution. Since in most cases, parallel computers are used for similar kind of jobs repeatedly, by monitoring the communication pattern, the execution cycle can be varied to reduce the context switching overhead.

BIBLIOGRAPHY

1. Akl,S.J.,
   The design and analysis of parallel algorithms,
   Prentice Hall, Inc., New Jersey 1989.
2. Alt.H., Hagerup, T., Mehlhorn, K., & Preparata,F.P.,
   Deterministic Simulation of Idealized Parallel Computers on More Realistic Ones, SIAM journal on Computing, Vol.16, No.5 October 1987, pp-808-835.
3. Howe,C.D., and Moxon,B.,
   How to Program Parallel Processors.
   Spectrum, Vol.24 No.9, Sept. 1987 pp-36-41.
4. Ferrari, D.,
   Computer Systems Performance Evaluation,
   Prentice Hall, 1978.
5. Hwang, Kai & Briggs, Faye A.,
   Advanced Computer Architecture and Parallel Processing,
   McGraw Hill, New York 1989.
6. Jamieson, L.H., Gannon, D.B., Douglas, R.J., (editors)
   The Characteristics of Parallel Algorithms,
   The MIT Press.
7. Quinn,M.J.,
   Designing Efficient Algorithms for Parallel Computers,
   McGraw Hill, New York 1987.
8. Snow, C.R.,
   Concurrent Programming
   Cambridge University Press, 1988.

9. Stuck, B.W., and Arthurs, E.,
A Computer and Communication Network Performance Analysis Primer.
Prentice Hall, 1985.

10. Towsley, D.,
Approximate Models of Multiple Bus Multiprocessor Systems. IEEE Trans.on computers, March 1984.

11. Svobodova, L.
Computer Performance Measurement and evaluation Methods: Analysis and application.
Elseivier, New York.