Loughborough University

This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (https://dspace.lboro.ac.uk/) under the following Creative Commons Licence conditions.

creative commons

COMMONS DEED

Attribution-NonCommercial-NoDerivs 2.5

You are free:

- to copy, distribute, display, and perform the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor.

Noncommercial. You may not use this work for commercial purposes.

No Derivative Works. You may not alter, transform, or build upon this work.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code (the full license).

Disclaimer

For the full text of this licence, please go to:
http://creativecommons.org/licenses/by-nc-nd/2.5/

BLDSC no:- DX176064

LOUGHBOROUGH
UNIVERSITY OF TECHNOLOGY
LIBRARY

AUTHOR/FILING TITLE

BUTT, W.U.N.

ACCESSION/COPY NO.

04007377 6

VOL. NO. | CLASS MARK

LOAN COPY

26 JUN 1998

30 JUN 1995

25 JUN 1999

15 MAY 1996

28 JUN 1996

27 JUN 1997

0400737760

BADMINTON PRESS
18 THE HALFCROFT
SYSTON
LEICESTER LE7 8LD
ENGLAND
TEL: 0533 602917
FAX: 0533 696639

LOAD BALANCING STRATEGIES FOR DISTRIBUTED COMPUTER SYSTEMS

By

Wajeeh Uddin Nakhshab Butt
B.Sc.(Eng.), M.S.

A doctoral thesis submitted in partial fulfillment
of the requirements for the
Award of the Degree of
Doctor of Philosophy
of Loughborough University of Technology

January 1993

©Wajeeh Uddin Nakhshab Butt 1993

Loughborough University of Technology Library
July 93
040073776
D9922014

CERTIFICATE OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this thesis, that the original work is my own except as specified in acknowledgements or in footnotes, and that neither this thesis nor the original work contained herein has been submitted to this or any other institution for a higher degree.

Wajeeh Uddin Nakhshab Butt

Abstract

The study investigates various load balancing strategies to improve the performance of distributed computer systems. A static task allocation and a number of dynamic load balancing algorithms are proposed, and their performances evaluated through simulations.
First, in the case of static load balancing, the precedence constrained scheduling heuristic is defined to effectively allocate the task systems with high communication to computation ratios onto a given set of processors.
Second, the dynamic load balancing algorithms are studied using a queueing theoretic model. For each algorithm, a different load index has been used to estimate the host loads. These estimates are utilized in simple task placement heuristics to determine the probabilities for transferring tasks between every two hosts in the system. The probabilities determined in this way are used to perform dynamic load balancing in a distributed computer system. Later, these probabilities are adjusted to include the effects of inter-host communication costs.
Finally, network partitioning strategies are proposed to reduce the communication overhead of load balancing algorithms in a large distributed system environment. Several host-grouping strategies are suggested to improve the performance of load balancing algorithms. This is achieved by limiting the exchange of load information messages within smaller groups of hosts while restricting the transfer of tasks to long distance remote hosts which involve high communication costs.
Effectiveness of the above-mentioned algorithms is evaluated by simulations. The model developed in this study for such simulations can be used in both static and dynamic load balancing environments.

To my parents,
Insha Fatima and Munir Hussain
and
to the memory of my Uncle,
Nazir Hussain.

Acknowledgements

I wish to express my sincere thanks and gratitude to my supervisor, Professor D.J. Evans, for his constant advice, interest and above all, encouragement throughout my research, as well as his endeavours in reading the manuscript of this thesis.

I would also like to convey my deep gratitude to the Ministry of Science and Technology, Govt. of Pakistan, for their financial support without which this thesis would never have been completed.

Eternal thanks to my parents, brothers and sisters, for their immense love, encouragement and their sound belief in my success, without which this work would have been inconceivable.

My bona fide thanks are conferred to my brother, Dr. S. A. Butt, for his constant encouragement and support, throughout the duration of my research.

Much appreciation goes to my wife, Guriya, whose dedicated love, faith and reassurance, gave me credence to complete my writing up.

Contents

Chapter 1

INTRODUCTION

1.1 Distributed Computer Systems

A distributed computer system is composed of a collection of independent processor-memory pairs which are interconnected by a communication network and logically integrated by a distributed operating system. The communication subnet may comprise a geographically widely-dispersed collection of communication processors or of a local area network [Stankovic 1984].

Distributed computer systems have attracted increasing interest and have been the topic of research for more than two decades now. With advances in VLSI technology, the cost of microprocessors is falling substantially making multiple processor systems more economical and highly attractive. Despite the promising nature and the suitability of application of distributed computation, serious academic and pragmatic problems arise which limit the performance of distributed systems. Such problems need to be solved before any assessment of the potential of this idea can be made. A distributed oper-

ating system appears to its users as an ordinary centralised system that runs on multiple independent processors. The key idea in a distributed operating system is to make use of the multiple resources in a distributed environment invisible to the user. A distributed operating system is a program that controls the resources of several computers connected with a network and provides its users with an interface that is more convenient to use than the underlying bare machine. In a true distributed system, users are not aware of where their processes are run and files kept. Whereas, the design of a distributed operating system is very vital in facilitating the users to efficiently exploit the resources of a distributed system, several factors present in a distributed environment limit the performance of such systems. One of the most important physical limitations that effects the performance of a distributed system is the interprocess communication, which depends on the topology of the inter-connecting network and the link bandwidths. Another fundamental problem in distributed systems is the lack of global state information. It is generally a bad idea, from the point of view of reliability, to collect complete information about any aspect of the system in a single centralised table. This lack of up-to-date information makes things difficult and requires new protocols to be designed for the purpose of collection of information from distributed sources. Some of the important issues, in the design of distributed systems, in which current research is proceeding are.

- Communication Primitives
- Naming and Protection
- Fault Tolerance
- Resource Management

1.2 The Research Problem

One of the major research issues associated with the design of distributed computer systems is load balancing. It is similar to the problem of task scheduling in a centralised processor. At present, the solution of the problem of scheduling in a single processor system is highly developed and automated. Users need not worry about the architecture and the management functions within the system and may concentrate on the actual problem at hand. With the introduction of distributed computing, the problem of resource management extends to the discovery of all the resources in the system and their efficient utilization. Users should perceive the entire distributed system as a single integrated system and not be able to differentiate between accessing a locally available resource from a remote one.

Load balancing is included in the area of resource management. This is a policy used to allocate processor resources to a set of tasks which are being executed in a multiple processor environment. Automatic allocation allows user programs to be unaware of the amount of resources available at a certain time, and allows a fair and consistent policy to be applied throughout the system to optimise the allocation. Optimal load balancing is an NP-complete problem which requires exponential time complexity [Ullman 1975]. Therefore, the alternative is to find a suboptimal or a heuristic algorithm for an efficient and close to optimal load balancing policy. The objective of task allocation is to allocate tasks among processors and to co-ordinate their parallel execution in order to minimise total completion time and total inter-task communication overhead. A task is a program module that is free to reside on any processor in a distributed system. The problem of mapping a given

number of simultaneously executable tasks on a number of processors which are available at that particular instant in such a manner so that a maximum performance or a minimum task turn-around time is obtained is called static load balancing. There are conflicting issues of limiting total communication cost function and achieving the maximum parallelism by assigning tasks to different processors. In order to take advantage of parallelism, tasks should be distributed over as many processors as possible. On the other hand, the communication cost introduced due to this parallelism may outweigh the improved performance which is achieved through the use of multiple processors. Precedence among the tasks, creation of new tasks, and size of the tasks (in terms of the processor time and memory requirements) introduce further complexity into the load balancing issue.

In contrast to this, dynamic load balancing has more realistic assumptions. In this case, jobs[1] with unknown characteristics arrive at each geographically distributed host in some unpredictable manner. This problem is different than that of the static task allocation mentioned above. Under these assumptions, the load balancing is performed at a higher level in which several tasks belonging to the same application are considered as a single job. The objective of the task allocation algorithms used for these models is to improve the system-wide load balancing and job response times. To reflect the frequently changing state of the system, load information amongst the hosts is exchanged either periodically or on demand. Load balancing decisions are based on the current system state information which is available at each host.

[1] a job consists of one or more tasks communicating with each other

Majority of the research scientists in the past have concentrated on the above-mentioned two types of scheduling problem. In the first instance, the problem arises due to a conflict in achieving maximum parallelism by load balancing and reducing the communication costs simultaneously. In the second case, the following problems arise.

- Task execution times are not known in advance. There are no simple ways to obtain good estimates of task execution times.
- Task arrivals, and hence the load on each host, are unpredictable.
- There is no global system state information available. For dynamic load balancing, load information messages need to be exchanged frequently amongst the hosts of the distributed system. The overheads involved in collecting this information should be kept to a minimum.
- The load balancing algorithms should be simple, efficient and stable. A complex algorithm may involve large overheads that outweigh the potential benefits obtained from load balancing.

Several studies were performed in the past to solve the above-mentioned problems. These studies were based on different sets of assumptions for the task system and the communication network parameters. Load balancing in distributed computer systems remains an open research problem and needs further study to enhance the performance of such systems.

1.3 Aim Of The Study

The issue involving load balancing in distributed computer systems is a complex one and several parameters need to be considered simultaneously

for an effective solution. It is very difficult to consider all such parameters as it makes the computational model intractable and difficult to solve. In the past, several static and dynamic load balancing algorithms have been proposed to improve the performance of distributed computer systems. The algorithms proposed in these studies consider a subset of these parameters based on a number of assumptions. The strengths of these algorithms lie in the nature of assumptions made about the distributed load balancing problem. While this problem has been investigated by many researchers, it has not been fully explored with all possible alternative approaches and strategies. The complexity of load balancing algorithms and the potential benefits which can be obtained from load balancing still remain questionable issues.

In this study, a static load balancing and several dynamic load balancing algorithms are proposed and evaluated for their performance in a distributed computer system. A static task allocation heuristic is proposed to solve the max-min[2] problem. Its performance is compared with two other simple task allocation heuristics.

The dynamic load balancing algorithms make task transfer decisions based on the current state of the system. Each algorithm uses a different parameter to estimate the load on each host. These estimates are used to find the task transfer probabilities for transferring tasks between two hosts in the system. The results indicate that good performances can be achieved by using the queue-length alone as the load estimate on a host. Later, some simple modifications were introduced to keep the communication costs as

[2]Maximizing parallelism and minimizing the communication costs

low as possible. The algorithms are simple and can be used for dynamic load balancing very effectively.

Finally, a few simple network partitioning and grouping strategies are proposed. These add a new dimension to the solution techniques for the load balancing problem. The effectiveness of such strategies is not fully explored in this study.

An event-oriented simulation model is developed to study the performance of the above-mentioned load balancing heuristics. Several important characteristics of both the deterministic and non-deterministic load balancing models are incorporated into the design of the simulator.

1.4 An Outline Of The Thesis

The thesis is organized into eight chapters. A brief overview of the contents of each chapter is outlined here. Chapter 1 provides a concise introduction to distributed computer systems and some important issues related to distributed computing that effect the performance of such systems. It also describes the nature of the load balancing problem and the contribution of this study.

Chapter 2 presents some important concepts in the scheduling theory associated with this subject. It also gives the background of the scheduling problem and some definitions useful for understanding the material presented in this thesis.

Chapter 3 provides a comprehensive review of the related literature. All the research undertaken in the area of load balancing has been classified into a number of categories depending on the computational model used to attack the problem of load balancing.

Chapter 4 provides the simulation model design and implementation details. This simulation model is used in the subsequent chapters to study the performance of the proposed algorithms. The task-transfer mechanisms used and the overheads of load information distribution are also introduced here. A number of performance metrics are required to evaluate the performance of load balancing algorithms. The two performance metrics used in this study, average task response time and load imbalance, are introduced in this chapter.

Chapter 5 presents a static task allocation strategy used to improve the performance of deterministic task systems with high communication to computation ratios. The Precedence Constrained scheduling heuristic is proposed to minimize the inter-task communication times and to balance the load across the processor network simultaneously. The performance of this heuristic is compared with two other simple heuristics and demonstrated through examples. The schedule completion time is used to measure the performance of these heuristics.

Chapter 6 focusses on dynamic load balancing issues in a distributed computer system. It describes the queueing theoretical model used to represent the non-deterministic load balancing in a distributed computer system comprising of four independent hosts. Two simple load balancing algorithms,

one based on periodic load information broadcasts and the other based on asynchronous load information messages, are proposed and evaluated. Next, a number of probability-based dynamic load balancing algorithms, each using a different load index, are proposed and their performances evaluated. Later, simple modifications are suggested to improve the performance of these algorithms by adjusting the task-transfer probabilities to minimize the communication costs. The performance metrics used are the average response time, the load imbalance and the percentage task transfers.

Chapter 7 introduces some simple network partitioning techniques and proposes several host-grouping strategies to reduce the communication overhead costs of the dynamic load balancing algorithms. By limiting the load information distribution and task-transfers within a host-group, the scalability of load balancing algorithms can be improved and large communication delays can be avoided.

Finally, Chapter 8 presents the conclusions and suggestions for further research in this area.

Chapter 2

BACKGROUND IN SCHEDULING THEORY

2.1 Introduction

This chapter provides the background study of multiprocessor scheduling theory. In the case of multiprocessor systems, task allocation to different processors is also considered as a part of the scheduling process. In this study, only loosely coupled multiprocessors are considered. The work presented in this thesis is aimed at finding efficient heuristics for task allocation in a distributed computer system. The difference in a message passing multiprocessor system and a distributed system is that of the inter-connection network and the inter-processor communication delays. The task scheduling function can be defined as the allocation of available processing resources to a set of tasks over a certain period of time. Most of the earlier work in multiprocessor scheduling is based on the deterministic scheduling theory. When the deterministic or the stochastic theory is used to specify the scheduling

parameters to evaluate the scheduling algorithms, it is referred to as the deterministic or probabilistic scheduling theory respectively.

The problem of scheduling in a multiple processor sytem is known to be an *NP-complete* in its general form [Ullman 1975]. When an optimal solution cannot be found in polynomial time, scheduling heuristics are used to provide fast sub-optimal solutions.

The work presented in this thesis concentrates on proposing new heuristics based on both the deterministic and the probabilistic models of scheduling. In this chapter, an attempt is made to give all the relevant information on scheduling theory necessary to understand and evaluate the work presented in later chapters. The scheduling problem and its background are briefly described in the next section. A general computational model for deterministic scheduling will be presented in the next section. The remaining sections will provide definitions of the performance criteria used for evaluating the scheduling algorithms and describe the computational complexity of such algorithms.

2.2 The Scheduling Problem

The problem of scheduling was first encountered in the classic job-shop or assembly line problems in Operations Research [cf. Coffman and Denning 1973]. There was a considerable amount of operations research on partially ordered sets of tasks of interest in job shops. Later, these techniques were applied to solve the problem of task scheduling in a multiple processor system environment. The problem specified was to efficiently allocate and schedule

a partially ordered set of tasks, with known execution times, to a number of identical processors. When the task system parameters are assumed to be known and all the tasks are simultaneously available for execution, the scheduling problem is categorized as deterministic or static scheduling.

The deterministic scheduling theory is applicable to a limited number of applications. In a large number of applications, the task behaviour and the inter-task communications among a set of tasks cannot be determined in advance. For a more realistic representation of the task system parameters, the probability models were used to capture the non-deterministic and unpredictable behaviour of tasks. This type of scheduling is categorized as probabilistic or dynamic scheduling. In the early 1960s, several attempts were made to apply the results obtained from queueing theory to solve the probabilistic scheduling problem [Coffman and Denning 1973] [Krishnamoorthi 1966]. Many researchers have used the term "load balancing" for multiprocessor scheduling. Load balancing is achieved by balancing the load across all the processors by using task allocation heuristics. The proper scheduling of tasks allocated to a processor can achieve increased performances. Scheduling a number of heavily inter-communicating tasks, resident on different processors, simultaneously reduces the communication delays as well as improves the task response times.

Currently, enormous research efforts are concentrated around the area of dynamic load balancing in distributed computer systems. Sufficient work, however, is also being carried out in the area of static task allocation and scheduling in multiprocessor systems. A comprehensive review on dynamic and static load balancing strategies will be presented in chapter 3.

A general computational model for deterministic scheduling is presented in the next section.

2.3 Basic Definitions

The computational models used to study the static and dynamic load balancing strategies will be described briefly in chapters containing work relevant to them. In this section, a few basic definitions are described that are essential to the understanding of both static and dynamic scheduling systems.

2.3.1 Resource Characterisation

The resources of a computer system are categorized into two classes, namely *shared* resources and *dedicated* resources. The computational units, i.e; processors, fall into the category of dedicated resources. There is a certain number of dedicated resources of each type. A task cannot be executed on more than one processor concurrently. Similarly, no other tasks can be executed on a processor already occupied by a task.

The distributed computer systems are categorized as homogeneous and heterogeneous computer systems. A homogeneous system is a collection of hosts[1] with identical hardware and software capabilities. On the other hand, the hosts of a heterogeneous system have a number of incompatibilities. These incompatibilities arise due to different processing powers of the hosts, operating systems and file servers etc..

[1]A host is an independent computer system on the network

2.3.2 Task System Characterisation

The task systems can be characterised in two different ways. In the first case, the task system is represented by a single Directed Acyclic Graph(DAG) [Coffman and Denning 1973] or a fully connected undirected graph [Stone 1978]. This task system consists of a set of co-operating and communicating tasks that belong to a single application. In this case, the task execution times and the inter-task communications are known and it is assumed that all the tasks are available for execution after meeting the precedence constraints. This type of task systems are used to study the performance of static scheduling algorithms in multiprocessor systems.

In the second case, a large number of applications are considered for concurrent scheduling. The tasks are considered as independent entities. The task arrival and task processing times are assumed as unknown. In these systems the inter-task communications are not considered, as a task usually represents a complete application. The scheduling problem is transformed into a load balancing problem and the tasks are transferred among the processors to increase the utilization of the processing resources. The load information estimates are exchanged amongst the processors periodically or on demand. The transfer decisions are based on this load information.

2.3.3 Centralized Versus Distributed

If the task of collecting the system load information and making the load balancing decisions is assigned to a single processor, it is called physically non-distributed or centralized load balancing. On the other hand, if all the processors in the system make a collective effort towards load balancing, it

is called physically distributed load balancing.

Several studies have been performed using the centralized load balancing strategies [Ephremides et. al. 1980] [Hajek 1984] [Zhou 1988]. Centralized solutions are generally considered unattractive for the distributed systems. It has been widely accepted that the centralized load balancing algorithms tend to create performance bottlenecks and single points of failure. Also, the performance of these algorithms degrades with any increase in the size of the distributed system. However, [Zhou 1988] has shown that the best solution is environment and problem dependent. The study indicates that the centralized approach to information distribution and task placement may be simple as well as efficient provided that the interprocessor communication is relatively efficient and the system scale is limited (up to 50-100 hosts).

The distributed algorithms are considered a more suitable and natural choice for load balancing in the distributed computer systems. These do not cause communication bottlenecks as those introduced by a single central load despatcher used in centralized load balancing algorithms. It also increases the system reliability and availabilty of system resources in the case of any failures. If a processor fails, the load balancing process will continue among all other processors in the system which is not the case in centralized scheduling. The majority of current research is focussed on finding efficient sub-optimal distributed algorithms for load balancing. The distributed algorithms are further classified into co-operative and non-cooperative. In the former, the local schedulers at each processor co-operate and make decisions that are based on the situation in the whole system. In the latter, the local schedulers make decisions independently that are based on the locally avail-

able information. The majority of work presented in this thesis is based on co-operative distributed load balancing algorithms.

2.3.4 Non-Preemptive Versus Preemptive

A scheduling algorithm is *non-preemptive* if a task which once starts execution, continues without interruption until its completion. On the other hand, in a *preemptive* scheduling algorithm, once a task starts execution, it can be interrupted only to be restarted later from the point of its last interruption. In non-preemptive scheduling algorithms, there is exactly one execution interval for each task, while in those produced by a preemptive one, there might exist more than one non-overlapping execution intervals for each task.

2.3.5 Task-Transfer Mechanism

In dynamic load balancing algorithms, tasks are transferred from heavily loaded to lightly loaded processors to achieve increased performances. Two types of mechanisms are used to transfer the tasks; a task placement mechanism and a task migration mechanism. In the earlier, once a task starts execution on a processor, this task cannot be transferred and will stay on the originating processor for its lifetime. In task migration, however, a currently executing task can be transferred to a remote processor for further execution to improve the load balancing. In the case of task placement, load information is used to create new processes at the lightly loaded processors instead of trying to move running processes. In task migration, the code, data and the environment of the running process are moved to resume the execution of the process on a remote processor. These variables and data structures

related to the process are scattered inside the operating system and involve large amounts of communication overhead which makes it infeasible without implementing special protocols for this purpose in the system. Actually migrating running processes is trivial in theory, but close to impossible in practice [Tanenbaum and Renesse 1985]. The task placement algorithms incur low overheads and can be easily implemented in a majority of operating systems, whereas only a few operating systems in practice support task migrations. The load balancing algorithms proposed in this thesis assume task placement model.

2.4 Scheduling Algorithms

A scheduling algorithm is a procedure that produces a schedule for a given task system. The efficiency and the performance of such algorithms varies with the assumptions made about the task system and other system parameters. There are a few optimal scheduling algorithms for restricted assumptions of the task system and the number of processors available. In this section, optimality of scheduling algorithms will be considered and special cases of optimal and non-optimal algorithms will be presented.

2.4.1 Optimal Scheduling Algorithms

If a scheduling algorithm produces an optimal solution for every given task system then it is called an optimal algorithm. The number of possible schedules for each task system depends on the number of tasks (n) in the system. In the majority of cases, an optimal algorithm will go through an exhaustive search of all schedules to find an optimal one. These algorithms require considerably large computation times (exponential) and are not suit-

able for an environment where low response times are expected from such algorithms. An optimal algorithm is considered efficient only if it requires a reasonable computation time to produce an optimal solution. The complexity of such algorithms is bounded by a polynomial of small degree and the algorithms are termed *polynomial algorithms.* There are few known polynomial algorithms for the scheduling problem with a severely restricted set of assumptions. Polynomial algorithms are obtained in the following two cases.

- When the task graph is a tree and the tasks are assumed to have the same execution time.
- When there are only two processors available with an arbitrary task graph and tasks have identical execution times (see [Lewis and El-Rewini 1992]).

However, the general scheduling problem has been reported to be an *NP-complete* problem. Also, the scheduling problem is *NP-complete* even for the following simple cases.

- Scheduling an arbitrary task graph with unit-time tasks on an arbitrary number of processors.
- Scheduling with two processors and an arbitrary task graph with tasks having one or two units of execution time (see [Lewis and El-Rewini 1992]).

It shows that the scheduling problem in its simplest form is intractable and cannot be solved in polynomial time. Therefore, to consider more realistic static and dynamic task scheduling models, sub-optimal heuristic algorithms

are designed. In the next section, heuristic scheduling algorithms are briefly discussed.

2.4.2 Heuristic Scheduling Algorithms

In the last section, it was shown that the efficient optimal schedules cannot be designed even for the simplest cases of the general scheduling problem. The scheduling problem becomes a lot more complex when the inter-task and the inter-processor communications are considered along with an arbitrary task graph and arbitrary number of available processors. The heuristic scheduling algorithms provide near optimal schedules with low computational complexity and easy implementation. These algorithms are easy to understand and provide reasonably good schedules in a small amount of time which is very important for any process running as a part of the operating system. Different heuristics are compared on the basis of efficiency and performance. The time complexity of the heuristic algorithm is used as a measure of efficiency and the performance is measured by finding how often the solution falls near the optimal solution. Alternatively, different heuristics can be compared for a given performance index. A heuristic is said to be better than another heuristic if solution falls closer to optimality more often or the time taken by the heuristic in finding the final solution is less.

Chapter 3

A REVIEW OF LOAD BALANCING STRATEGIES

3.1 Introduction

Although the past two decades have seen major advancements in the field of distributed computing, its evergrowing use has raised several issues which are currently under study and are aimed to exploit the potential power of distributed computer systems. A number of studies [Tilborg and Wittie 1981] [Stankovic 1984] [Tantawi and Towsley 1986] [Stankovic et. al. 1978] [Enslow 1978] have reported on the importance of the load balancing issue to enhance the performance of such systems. This chapter provides a comprehensive review of a wide variety of techniques and methodologies employed for load balancing in distributed computer systems. The problem of load balancing in distributed systems is similiar to the scheduling problem in a single processor system [Coffman and Denning 1973]. However, as the work presented in the present study is aimed at load balancing in distributed computer systems,

the latter will not be considered in our discussion.

Load balancing comes under the area of resource management. It is similar to the problem of task scheduling in a centralised processor. Its objective is to allocate tasks amongst processors and to co-ordinate their parallel execution in order to minimise the total completion time and the total inter-task communication overhead. A task is a program module that is free to reside on any processor in a distributed system. Given a number of simultaneously executable tasks and a number of processors at an instant, the problem of mapping these tasks to the processors to achieve a maximum performance or a minimum task turn-around time is called load balancing. There are conflicting issues of limiting total communication cost function and getting the maximum parallelism by assigning tasks to different processors. In order to take full advantage of parallelism, tasks should be distributed over as many processors as possible. On the other hand, the communication costs introduced due to such parallelism may outweigh the improved performance achieved through the use of multiple processors. Precedence among the tasks, the creation of new tasks and the size of tasks (in terms of the processor time and memory requirements) introduce further complexity into the load balancing issue. Due to the conflicts in achieving the goals of maximizing throughput , minimizing response time and keeping the load uniform, many of the researchers try to evaluate different compromises and trade-offs. Despite the fact that work performed to date is based on the concept of co-scheduling, which takes inter-process communication patterns into account while scheduling to ensure that all members of a group run at the same time. Other researchers have worked on a totally different concept of finding clusters of tasks working together and placing these on the same machine

to reduce the costs of inter-task communication. Yet some other researchers have tried to balance the load on all the processors by preventing a situation in which some processors are overloaded while others remain empty. Each of these different approaches to scheduling makes different assumptions about what is known and what is most important. The people trying to run a group of tasks on the same machine to reduce the communication cost assume that any task can run on any machine, that the computing needs of each task are known in advance and that the inter-task communication traffic between each pair of tasks is also known in advance. In contrast to this, the people involved in dynamic load balancing have more realistic assumptions and assume that nothing about the future behaviour of a task is known. They are not particularly worried about finding optimal algorithms with weak assumptions, and instead are more interested in devising suboptimal algorithms or heuristics that can actually be used in real systems.

Recently, several models of distributed task scheduling systems have been developed. Most of the work done is based on list scheduling [Adam et. al. 1974] [Blazewicz et. al. 1986] [Rewini and Lewis 1990] [Price and Salama 1990] [Shirazi and Wang 1988], queuing models, graph-theoretic models [Bokhari 1979] [Lo 1984] [Indurkhya and Stone 1986] and Markov decision theory, and is good for static load balancing with a few exceptions of dynamic load balancing. Dynamic load balancing is a non-deterministic function of the current state and system parameters, which is an NP-complete problem and cannot be solved in polynomial time complexity. As a result, most of the designers have tried to attack this problem using heuristics.

3.2 Static Load Balancing

Most of the static load balancing algorithms are deterministic. For the purpose of static load balancing, it is assumed that the system consists of a fixed number of tasks, each with a known processor time and memory requirements. It is further assumed that the inter-task communication patterns among a set of tasks is already known. These algorithms are designed for a given topology of the underlying multiple processor network.

3.2.1 List Scheduling

Several optimal and sub-optimal list schedules have been proposed in the literature. List schedules are classified as those schedules in which tasks are ordered in a priority list for processing. Priority of each task included in the list is determined by the scheduling heuristic applied. List schedules are best suited to the scheduling problems where the task system is represented by a precedence graph and the number of processors in the network is known.

[Adam et. al. 1974] provides a comparison of list schedules for parallel processing systems. The problem consists of scheduling a set of partially ordered tasks on two or more processors. In this study, inter-task and inter-processor communications are not considered. Several simple algorithms based on level priorities and a random priority schedule are compared with the optimal schedule. The level of a precedence graph is defined in a similiar way to previously published work [Coffman and Denning 1973]. The results obtained are suggestive that the algorithm based on Highest Levels First with Estimated Times (HLFET) performed better than heuristics based on Highest Co-levels First with estimated and no estimated times and the random one.

HLFET also proved to be near-optimal in both the deterministic and non-deterministic cases. However, it is not clear why this heuristic is superior to all others.

[Shirazi and Wang 1988] presents two static task scheduling heuristics with assumptions similar to those described in [Adam et. al. 1974]. The first heuristic function Heaviest Node First (HNF) assigns the highest priority to the heaviest node at each level. This algorithm uses level by level information of the precedence graph to distribute the tasks. The second heuristic is based on the classical Critical Path Method (CPM) [Coffman and Denning 1973]. This heuristic is called Weighted Length (WL) and calculates the priority of each task by comparing the weight of the tasks in the subgraphs which are dependent on this task. A task with largest weighted length is given the highest priority. Simulations were run for these two heuristics and the CPM heuristic. Simulations carried out by the author suggest that there is no significant difference among the performance of these heuristics. However, it is indicated by simulations that the Weighted Length (WL) is better than the other two. The complexity results show that the complexity of HNF allocation heuristic is $O(n \log n)$ while the complexity of WL and CPM heuristics is $O(n^2)$. Therefore, considering the insignificant difference in simulation results, it can be proposed that the HNF algorithm is a better choice of the two.

A more recent work on static task distribution with comparatively more realistic assumptions for distributed systems has been described in [Rewini and Lewis 1990]. The computational model included in this study consists of a set of precedence constrained tasks with arbitrary communication among them

on a set of homogeneous processors taking contention into consideration. A Mapping Heuristic (MH) has been shown to map the tasks of the precedence graph onto the given processor network. In a way similar to that described in [Adam et. al. 1974], the level of each node is used to determine its scheduling priority. However, as the communication delays along the path of a node are also considered in determining the level priority for a node, the problem of finding levels could be considerably difficult if arbitrary communication delays are considered. The performance of MH heuristic was demonstrated through an example precedence graph obtained from an atmospheric science application. In this example, constant inter-task communication times were considered and a hypercube processor inter-connection topology was used. However, it remains unclear how this heuristic will behave for precedence graphs of other computation to communication ratios and for communication times with larger variance. The effect of proposed scheduling heuristic was also studied for different processor inter-connection topologies. Contention arises when several tasks on different elements are waiting for communication across a common channel. This contention may have adverse effects on the performance of a distributed system, if two communicating tasks were scheduled on the processors that used this busy channel. [Rewini and Lewis 1990] have presented a modified MH heuristic which also takes into account the communication channel contention at the time of making scheduling decisions.

[Price and Salama 1990] present several approaches to statically assign and schedule the tasks in order to achieve maximum parallelism and minimum communication overhead in both fully-connected and hypercube multiprocessor networks. The computational model is similiar to the one in [Rewini

and Lewis 1990] except that communication resource contention is not considered. Three simple heuristics are described. First heuristic is called the list heuristic, in which its tasks are initially sorted in order of increasing total communication. The reasoning for this sorting being that the task that will be assigned in the very beginning will not be based on the cost of communication with other tasks. Henceforth, they should be ones that communicate least so that minimum information is ignored. Each task from the ordered list is scheduled on a processor at the earliest time such that communication among assigned tasks is minimised and also the precedence constraints are not violated. Second heuristic is called the Cluster Heuristic and is based on clustering together of the tasks, with high communication costs and on the same processor. First of all the list of tasks is sorted in the order of decreasing communication costs. This is done to form the clusters of tasks with high communication costs first. Each time a new pair of tasks is considered, and if neither of the two is assigned then both are assigned to a least loaded processor. If one task is already assigned, the other is assigned to the same processor unless a specified load threshold has been exceeded. In case of exceeding the specified load threshold for a processor, the task is assigned to a least loaded processor. This load threshold is specified to achieve a certain degree of load balancing, which prevents large clusters that would result in poor utilization of parallel processing. In the last phase of this heuristic, the tasks are ordered within each cluster with respect to precedence constraints. Also, subsequences of tasks are arranged in time to enforce precedence constraints among tasks that belong to different clusters. The last is called the Exchange Heuristic and is based on the pairwise exchanges of tasks with the goal of decreasing communication costs. Exchanges are made only if

precedence constraints are not violated. It is an iterative-improvement algorithm that begins with any initial feasible assignment of tasks to processors and time-periods. At each iteration, a selection is made of that pair whose exchange would yield the greatest improvement in the objective. Computational results on a large number of test data are presented in this paper and are consistent with the intuitive expectations about the communication costs and schedule lengths. Overall, the List heuristic outperforms the other two simple heuristic methods, producing both shorter schedule lengths and lower communication costs.

3.2.2 Graph-Theoretic Models

A large number of algorithms for this type of scheduling use Graph-Theoretic Models. The system can be represented as a graph, with each task a node and each pair of communicating tasks connected by an arc labeled with the data rate between them. [Stone and Bokhari 1978] have presented the algorithm for finding the optimal allocation of a fixed number of tasks on a two processor system using the minimal cut method for network flow graphs. They have modified the network flow graph by adding additional nodes for the two processors and extending the arcs from each of the task nodes to the processors. The weights of the new edges are such that the edge to one processor carries the execution cost of that task on the other processor. A cut set in this graph is a collection of edges which if removed from the graph may disconnect one processor from the other. Further details of this algorithm can be found in [Stone 1978]. Each cutset in the graph corresponds in a one-to-one fashion with the task assignment. The optimal assignment corresponds to a minimal weight cutset. Also presented in their publication is a very sim-

ple algorithm for calculating optimum assignment for tasks under varying load conditions. The assumptions made in the above mentioned work that all system parameters are fixed except the load on a processor, which are not very realistic assumptions in terms of dynamic scheduling. The basic idea is that for systems in which a single processor load factor is the only variable parameter, optimum assignments can be made by calculating all critical load factors ahead of time and by comparing the computed load factors against the actual load factors experienced at a certain time. A major problem with these algorithms is the difficulty faced while finding an optimal assignment in situations where either or both processors have memories of limited size. This idea of cutset used for two processor systems is extended even further for three processor systems by defining the notion of a tri-cutset. The basic idea behind this algorithm is to run a network flow algorithm between each possible pair of processors in the graph. A tri-cutset in a graph with three processors is a subset of arcs of a graph which if removed partitions the graph into three distinct subgraphs, each with processor node of its own. Actual details of the algorithm are complex and are mentioned in [Stone 1978], whereas it finds the minimum tri-cutset very efficiently for almost all graphs. There are some graphs for which the algorithm fails to find the optimal assignment, but it does help to indicate that a suboptimal solution exists. [Stone and Bokhari 1978] present a dynamic assignment problem for a two processor case which is summarised in the next section.

[Lo 1984] presents an algorithm which operates on the network model of the n-processor system based on a known result [Stone 1977]. This algorithm consists of three parts namely,: (1) ITERATIVE, (2) LUMP, and (3) GREEDY. In each iteration of Part 1, the Max Flow/Min Cut Algorithm

is applied for each processor node to determine the subset of tasks assigned to each node. Each iteration might result in a partial assignment such that there remain some tasks which are left unassigned at the completion of each iteration. A new graph is constructed with the remaining unassigned tasks. This is repeated until the assigning of all tasks is completed, none is left for assigning in the last iteration. Proofs of the fact that no two processors are assigned the same task and the iteration task does halt are presented in [Lo 1983]. Part 2 of the algorithm computes a lower bound on the cost of an optimal n-way cut for the remaining network under the condition that more than one processor be utilised in the corresponding assignment.

$$L = \sum_{t \epsilon T_{rem}} \min(x_{ip}) + \min_{i \neq r} C(P_r, P_i)$$

L =The lower bound on the cost of an n-way cut

t =A task

T_{rem} =A set of remaining tasks

$C(P_r, P_i)$ =The cost of the minimum cut for some arbitrarily chosen two processors.

The lower bound is compared with the cost of assigning all the remaining tasks to a single processor, and the cheaper of the two is applied. This resultant assignment in combination with the assignment from Part 1 is optimal. Part 3 of this algorithm locates clusters of tasks among which communication costs are large. Tasks in a cluster are then assigned to the same processor. The resultant assignment in this case might be suboptimal.

The above mentioned algorithm uses the sum of total execution and communication costs as the performance criteria which is to be optimised. In this case, no explicit effort is made to utilise more processors in order to reduce the response time of the set of tasks. The use of total execution and communication costs as the performance measure often results in assignments which utilise only a few of the available processors. In the next algorithm, [Lo 1984] adds interference costs to the performance criteria of algorithm I;i.e; sum of execution and communication costs. Interference cost is categorised in two types, that is processor-based interference costs and communication-based interference costs. Processor based interference costs involve contention between tasks for the resources of the processor to which the tasks are assigned, they incur task switching overhead, synchronization for access to shared resources, cpu time and, I/O etc. Communication-based interference costs are due to the contention for message buffers and for synchronization for message-passing. These in turn depend on the inter-task communication services provided by the processor involved in order to send and receive messages. The last algorithm presented by Lo considers the problem of task assignment to minimise total execution and communication costs subject to a constraint on the number of tasks which are assigned to each processor. Simulation results presented in the same paper indicate that there are advantages as well as disadvantages to each of the above-mentioned performance criteria which is the simplest and which provides a good measure of the global resource usage. However it does not provide the required degree of parallelisation and hence the response time of the set of tasks is high. Although an addition of interference costs makes a better utilization of all processors and improves the overall performance but it introduces a great

deal of complexity to the problem, because the jobs of measuring interference costs with a reasonable accuracy is very difficult.

3.2.3 Clustering Techniques

[Gylys and Edwards 1976] whose work describes various clustering techniques for efficient workload partitioning on a distributed system. Their work is amongst the earlier efforts in exploring the clustering methods. Simple clustering heuristics were applied. Every pair of processors was marked as eligible for fusion into a single processor. Only in cases, where the fusion of a pair of processors into a single processor would eliminate the greatest amount of communication traffic, then the fusion is performed and the workloads are coalesced. The pair that could not be fused is eliminated from the list of eligible pairs. The second heuristic consists of assigning an initial centroid for each processor. Distance is calculated from the centroid of each cluster to the program that needs to be assigned. This program is assigned to the nearest cluster and the cluster centroid is adjusted accordingly. This study indicates that non-hierarchical cluster analysis techniques produce good sub-optimal solutions. Nevertheless, clustering methods may fail to produce a good solution for situations where the hardware resources are tight relative to the workload.

[Efe 1982] suggests a Module Clustering Algorithm (MCA) and a Module Re-assignment Algorithm (MRA) that work together for load balancing in a distributed system. MCA is used to form module clusters based on a criteria of minimizing on the total inter-module communication costs. After the initial assignment of these module clusters, if the load is unbalanced across the distributed system, MRA is activated for the load balancing. This

algorithm works sufficiently well for most of the cases, fails only in some special cases.

[Price and Krishnaprasad 1984] present heuristics for software allocation in heterogeneous distributed computer systems. Heuristics based on iterative assignment-improvement, quadratic programming, and clustering methods are described. In this section, I will only consider the clustering heuristic in detail. Clustering algorithm is carried out in two phases. In the first phase, pairs of tasks are considered in the order of decreasing inter-task communications. If none of the tasks in the pair is already assigned, both are assigned to a processor that improves the overall completion time of the schedule. In situations where one task in the pair is already assigned, the other task is also assigned to the same processor. In the second phase, tasks still unassigned after the completion of first phase, are allocated to processors on the basis of least execution costs and memory capacity constraints. The performance of clustering heuristic was studied for different computation to communication ratios for the task system. Clustering produced very good results for problems with high communication to computation ratios.

Although [Price and Salama 1990] suggest several heuristics for task allocation in a homogeneous distributed system. I will only consider cluster heuristic for this discussion. Cluster heuristic presented in this paper is designed to cluster the tasks for minimizing the inter-task communications while preserving the precedence constraints among the tasks. Tasks which are sorted in the order of decreasing pairwise communications in the first phase are assigned to the processors in the next phase. If one task in the pair is already assigned, the other task is assigned to the same processor.

If none of the tasks is assigned, both are assigned to the same processor. Every time a task is assigned to a processor, its load is compared to a specified load threshold. If the processor load exceeds the threshold value, the new task is assigned to the least loaded processor in the system. This helps to keep the load balanced across the distributed system. Next, all the tasks within the clusters are arranged to satisfy the precedence constraints. Finally, some tasks are shifted to later time periods to enforce precedence constraints among tasks that belong to different clusters. The tests performed suggest that the cluster heuristic tends to cluster tightly-coupled tasks and hence may produce lower communication costs at the expense of longer schedule length. With the incorporation of load balancing constraint, schedule length can be improved.

[Wu and Sweeting 1992] introduce a new approach for static load balancing. This approach uses clustering techniques to solve the task assignment and network structure problem at the same time. Such heuristics can be efficiently utilized for the distributed systems with re-configurable message passing networks such as those in transputer networks. Task clustering and re-assignment techniques, similiar to the ones described in [Efe 1982], are used for the assignment of task clusters to processors and to load balancing. A Link Number Reduction Algorithm (LNRA) is used to find the suitable network structure for the proposed task assignment. Finally, task scheduling is performed to arrange tasks within one processor or amongst different processors to minimize idle time which could be spent in waiting for communications. At the time of task assignment and scheduling, the precedence constraints among the tasks are not considered. No performance measures are provided for these algorithms. There applicability for various scheduling

problems is not clearly described.

[Ousterhout 1982] presents several algorithms based on the concept of co-scheduling. The author takes into account various inter-task communication patterns and collectively schedules all such tasks which belong to a given group. A group of tasks consists of all those tasks that communicate with each other. It is assumed that a sufficiently large number of machines are available to handle the largest group. It is also assumed that each such machine is multiprogrammed with N slots. The algorithm described uses a conceptual matrix in which a column consists of all the tasks that are in a particular slot of different machines. The basic idea is to have each processor use a round-robin scheduling algorithm with all processors first running the task in their slot 0, then slot 1 and so on. To keep the time slices synchronised among all the processors of the distributed system, a broadcast message can be used for task switching. Several other improvements on this algorithm are described in the same publication. One of these breaks the matrix into rows and concentrates all these rows to form a single long row. If there are 'n' number of machines then each of the 'n' consecutive slots belongs to a different machine. To allocate a new task group, an n slot wide window is laid over the long row in such a manner that the left most slot inside the window becomes full and the slot outside the left most edge of the window is empty. In cases where the number of empty slots available is not sufficient for all the tasks on the task group the window is moved by one slot to the right. This algorithm is repeated until sufficient empty slots become available in the window for all the tasks on the task group. Ousterhouts' paper discusses several other methods in detail and presents some performance results.

3.3 Dynamic Scheduling

Efficient static scheduling algorithms may provide optimal performance with the assumptions that all parallel tasks are known 'a priori'. This type of scheduling is not very efficient for real time systems, in which a parallel computation is modelled by a dynamically created task precedence graph. During the lifetime of a distributed task, it may spawn new tasks or destroy the current tasks. This new set of tasks needs to be dynamically mapped onto the network nodes using only partial knowledge of the global network state. This scheduling problem is considerably more difficult than the already difficult static scheduling problem. Algorithms for dynamic task scheduling are classified as either placement or migration schemes [Reed 1984]. A task placement algorithm assigns tasks to nodes before the task begins execution. All the tasks execute where placed initially, even if moving tasks might reduce the load imbalance. A task migration algorithm can move tasks after their initial placement. Migration algorithm might seem superior, but these involve great overheads in moving the task execution state. The hardest part is not moving the code, the data and the registers, but moving the environment such as the current position of the open files, pointers and file descriptions etc. All these problems are related to moving variables and data structures that are scattered inside the operating system. The superiority of placement or migration depends on the structure of the parallel computation. In the present discussion on dynamic scheduling techniques, some quasi-dynamic algorithms that lie somewhere in the middle of the spectrum between static and dynamic algorithms will also be considered.

3.3.1 Dynamic Algorithms

[Chow and Kohler 1979] have used the queueing network model for the performance study of a heterogeneous multiple processor system. Both the cases of a deterministic and a non-deterministic routing are considered. In the non-deterministic case, each newly arrived job selects one of the processors with a fixed branching probability. Since the branching probabilities are fixed, therefore it can be argued that the solution is non-deterministic but not a dynamic one. For dynamic routing strategies, these branching probabilities must change over time thus reflecting the load changes in the system. The branching probabilities are assumed to be directly related to the service rate of each processor in the system, independent of other state parameters. Authors have compared and analyzed several adaptive job routing strategies. Their studies indicate that queueing network models based on static non-deterministic branching probabilities may be inadequate for modelling distributed systems employing the dynamic load balancing policies.

[Smith 1980] presents the earliest work on task allocation in distributed systems using bidding algorithms. A contract net protocol was designed to provide a mechanism for interaction between nodes with tasks to be executed and nodes ready to execute tasks. A manager is responsible for monitoring the execution of a task and processing the results of its execution, whereas a contractor is responsible for the actual execution of the task. A node can take either role dynamically during the course of problem solving. In brief, available contractors evaluate the task announcements from the managers and submit bids for the suitable ones. The managers evaluate the bids and award contracts to the appropriate nodes. Different criteria can be used for

estimating the value of a bid. The overheads involved in such algorithms depend on the level of complexity used for implementing the negotiation process.

[Reed 1984] investigates the behaviour of different types of inter-connection networks under varying workloads and the feasibility of distributed scheduling. The work presents results of a large scale simulation study and establishes some important features for dynamic schedulers. Some general observations from his simulation results are given below.

- Selection of a particular network must be made with a knowledge of communication patterns and task sizes required by an algorithm.
- The maximum task branching factor, which means how many tasks can be formed by a task at a certain time, has to be constrained depending on the degree of a node in the network (network connectivity).
- Mean ratio of computation to communication time must be considered.
- Dynamic task scheduling using only locally available information seems feasible for the class of algorithms represented by dynamically changing precedence graphs.
- For newly created tasks, it is important to assign the task to the nodes that are near to the point of task creation.

Despite the above observations [Reed 1984] did not propose any specific algorithms, but suggested that heuristic algorithms can be applied for efficient dynamic scheduling.

[Gao et. al. 1984] have proposed two algorithms to equalize loads on hosts in a homogeneous distributed computer systems. A periodic load information distribution protocol is used and the tasks are transferred once during each load update interval. The task arrival rate and the total amount of unfinished work on each host are used as the load estimates. Average response time and load imbalance are used as the performance metrics. A cost function comprising of number of task transfers and the inter-processor communication cost between two hosts is minimized using the linear programming techniques.

[Tilborg and Wittie 1981] describe a distributed scheduling technique called the wave scheduling that maps dynamically created groups of tasks onto multicomputer network nodes which are readily available. The positive aspect of wave scheduling is that it doesn't make any assumptions about the arrival of tasks, inter-task communication patterns and task resource requirements, which makes it more attractive. On the other hand, the negative aspect of wave scheduling is that it does not permit dynamic creation of tasks. Wave scheduling uses the concept of task forces, which consists of a group of cooperating tasks. These task forces can not create new tasks after they have begun execution. In wave scheduling, the processing nodes are arranged hierarchially by assigning a manager to a small subset of nodes. In cases where a large number of resulting managers exist, these are further subdivided by the assignment of managers on a higher level in the tree and so on. Such management of the hierarchy does not necessarily correspond to physical connections between nodes. To avoid communication bottlenecks, each node can only exchange control information one level upward or downward. A manager node at each level of the hierarchy maintains only the summaries of the resource information known to its subnodes. The simulation studies [Tilborg

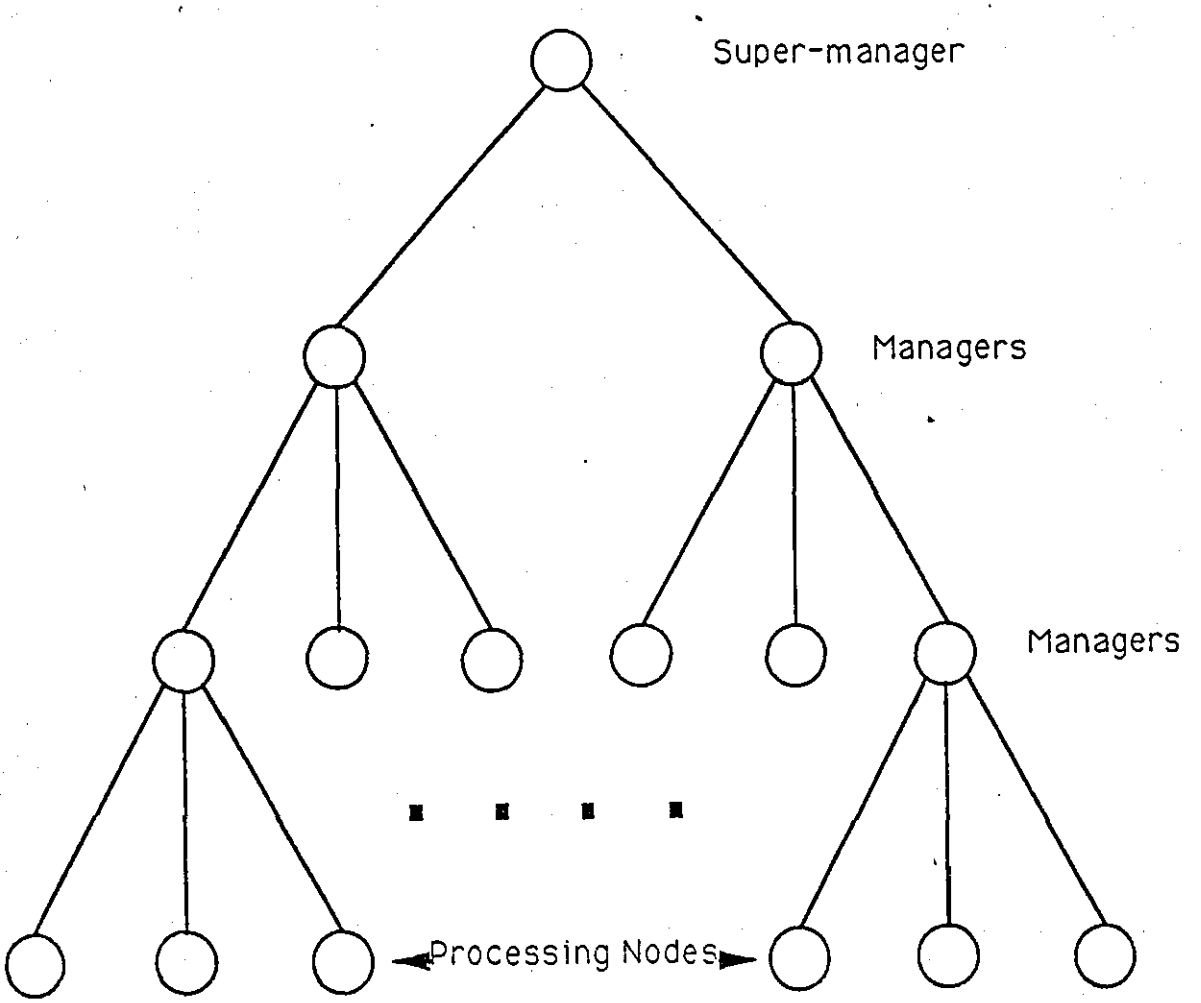

Figure 3.1: Manager Hierarchy in Wave Scheduling

and Wittie 1981] suggest that hierarchies with efficient communication paths can be created.

As mentioned above, each of the managers keeps information about the available number of the computation nodes in their subtree. If we suppose that a task force needing S nodes is created at an arbitary node and K is the fraction of idle nodes that can be safely scheduled. At an arbitary time t, a manager might schedule task forces containing no more than

$$S = K[1 - Util(t)]W \quad tasks$$

$Util(t)$ is the fraction of computation nodes, in a subtree of W nodes, that are being utilised during a given time t.

If a task force is created at a manager which does not have the required number of nodes to schedule all tasks, the request is passed up the tree until a manager with sufficient available nodes is found. On the other hand, if a task force is created at a manager which has surplus number of available nodes than those which are actually required, this request is passed down the tree until it reaches a manager that minimally satisfies the requirements. Since the children of manager nodes are also managers except those at the leaf nodes of a tree, there is a competition amongst the managers of a subtree for acquiring the computation nodes. If S nodes are required to schedule a task force, a manager usually reserves $R \geq S$ nodes. A request for R nodes is divided among the submanagers of a task force manager and is propagated down the hierarchy as a wave of requests, hence the name wave scheduling. A request generated from the parent takes precedence over the requests generated at the local node. When the request reaches the lowest level, managers then reserve as many of the requested nodes as possible. These managers then tell their managers about how many nodes were reserved. Managers at each level await the responses from their submanagers before advising their parent manager about how many nodes were reserved. After these results are propagated to the requesting manager, and the sufficient nodes are not reserved, the scheduling pass fails and this manager issues a command releasing all the nodes. At a later time, the task force manager will make another attempt. If the requesting manager fails, even after several attempts, it will pass the task force upward to its parent and the process for reserving nodes will be repeated. There are two important factors that determine the performance

of wave scheduling.

- The management hierarchy should be arranged such that the communication delays are minimised.
- Reservation costs for the computation nodes should be low.

[Barak and Shiloh 1985] introduce three algorithms that work together for dynamic load balancing in a distributed computer system. These algorithms have been designed for their use in MOS (Multicomputer Operating System) described in [Barak and Litman 1985]. Barak proposes a pre-emptive load balancing algorithm for a multicomputer system having a homogeneous node architecture and a communication network that completely connects all the nodes by allowing direct communication between any pair of nodes. They introduce a local load algorithm, used by each processor to monitor its own load; the exchange algorithm, for exchanging load information among the processors, and finally the task migration algorithm that uses this load information to dynamically migrate tasks from overloaded to underloaded processors. During the course of execution, if the workloads of the processors become unbalanced, a task can be migrated to an underloaded processor to continue its execution. This algorithm utilises the task migration feature of MOS to reduce the response time and hence increase the overall performance of the system. Measurements of the instantaneous processor load show rapid fluctuations, which might mislead to inaccurate processor load information. A processor might indicate a low load value, since at the time of measurement all the tasks were waiting for the completion of I/O operations. A short time later, a task might migrate from a loaded processor to this processor and find that the load of this processor is at a higher level than the processor

from which the task migrated. To avoid such cases, the algorithm descibed uses an average value of load over a certain period of time. Further details of these algorithms and the performance results, related to MOS, can be found in [Barak and Shiloh 1985].

[Lin and Keller 1987] proposed and analysed a class of distributed scheduling algorithms based on gradient planes. In such algorithms based on gradient plane, an idle node requests tasks from its immediate neighbours, which are directly connected to it. If any of the neighbouring nodes is over-loaded with excess tasks awaiting execution, a task is transferred to an idle node. If none of the neighbours is overloaded, the request for tasks is modified and propogated to more distant nodes. A potential is associated with each node based on its current load. Idle nodes, being least busy, have the lowest potential. Excess tasks flow along gradients to these nodes of lowest potential. There is no actual implementation of such an algorithm but several simulations on randomly distributed data are performed to assess the efficiency of this algorithm. The gradient model of distributed scheduling does not readily accommodate a collection of co-operating tasks.

[Wang and Morris 1985] have presented and compared several source-initiative and server-initiative load sharing strategies with a varying level of information exchange among the nodes. It has been indicated that server-initiative algorithms have the potential of outperforming source-initiative algorithms for the same level of information available at each node. It has shown that the server-initiative strategies degrade more gracefully under server failures since a failed server will cease to request more work.

[Stankovic 1985] has proposed and analyzed an adaptive decentralized job scheduling strategy based on an application of Bayesian decision theory. This strategy devides the scheduling function into two parts: a decentralized job scheduler (DJS) and a decentralized process scheduler (DPS). The heuristic used for DJS was very simple and was run infrequently to keep the overhead costs low. A DPS was proposed to deal with multiple modules of a job in execution (i.e; processes). The heuristic suggested for DJS stores a table of maximizing actions at each node. Each controller informs a centralized component of its true state and observed state periodically. This information is used to dynamically recalculate the individual probability distributions needed by each of the controllers. Heuristics were tested for different parameter thresholds and their performances were compared. Both the static and the dynamic models for probability assignments were studied.

[Leland and Ott 1986] performed an exhaustive study of process behaviour in the VAX/UNIX[1] environment and used this information to evaluate the performance of load balancing with process migration and initial placement strategies. The results showed good performances for simple load balancing heuristics under workloads reflecting the actual behaviour of processes. The stability and the scalability of the proposed heuristics was not considered in their study.

[Zhou 1988] has studied dynamic load balancing using a simulation model driven by job traces collected from VAX/UNIX environment. Several representative load balancing algorithms were studied under moderate to heavy loads. This study revealed that load balancing algorithms using periodic and

[1]UNIX is a registered trademark of AT&T Bell Laboratories

non-periodic information distribution mechanisms yield comparable performances. It was also observed that the scalability of load balancing algorithms is limited to a few tens of hosts. Furthermore, the results indicate that the centralized approach to load distribution and job placement may be simple and efficient when the inter-processor communication is relatively efficient and the system scale is limited.

[Perihelion 1989] provides support for coarse grained parallelism, in which the smallest unit of parallelism is called a task. A task is a self-contained unit which has been separately compiled and linked. A group of tasks related in this manner is called a task force. A piece of code known as the Task Force Manager (TFM) is responsible for mapping these task forces onto the available processors in the most efficient possible way. The Task Force Manager is a distributed server and it is assumed that each of its components provides some information about its resource requirements. The Task Force Manager is a distributed server. It is hierarchially organised and consists of a number of identical servers distributed throughout the network. Each of the TFMs controls a different area of the network. TFM analyzes the current state of the network and distributes the component tasks of the task force to the most suitable processing elements. The criteria for distribution include the resource requirements of particular component tasks, connectivity of the task force, and the current status of the network. It is not described what heuristics are used for scheduling, however it seems that the heuristics used are very much similar to the wave scheduling techniques described earlier in this chapter. Also it is not clear how the current network status is measured and what factors are considered. Further, it does not state how the inter-task communication patterns and other resource requirements for the task force

are estimated. It is also not shown how the heuristics used for the task force manager scale with the increasing size of the hierarchy.

[Mirchandaney et. al. 1990b] have analyzed the effects of delays on simple load sharing strategies in distributed systems. The load sharing strategies considered were based on probe limits for searching an eligible host for load balancing. The communication delays in tranferring tasks were assumed to be non-negligible. Forward and reverse probes represented the source-initiated and server-initiated algorithms respectively. The results indicated that for low to moderate loads and high transfer delays, load balancing was not beneficial. However, for moderate to high loads, substantial benefits are obtained from load sharing even at high transfer delays. It was also observed that an algorithm that utilizes both the forward and reverse probes gives the best performance, but generates high probing overheads.

[Blake 1992] studies the problem of assigning independent tasks in a multi-computer system to minimize completion time. It was assumed that each task requires execution on a single processor and an estimate of the tasks' maximum execution time is available. Several dynamic load balancing heuristics, differing in efficiency and effectiveness, were examined. The results indicate that in many situations, a more complex scheduling algorithm fails to out perform relatively simpler schedulers.

3.3.2 Adaptive Algorithms

Adaptive algorithms constitute a subset of above-mentioned dynamic algorithms. In adaptive algorithms, the system state information may be used to modify the parameters of the algorithm, or even to choose which load

balancing strategy is to be used. As a simple example, sender-initiated algorithms tend to perform better than the receiver-initiated algorithms when the overall system load is low. When the system load is high, the sender-initiated algorithms perform disasterously. Therefore, an adaptive algorithm might choose to follow a sender-initiated strategy when the system load is low, and may switch to a receiver-initiated strategy when the load becomes high. Some of the adaptive load balancing algorithms are described below.

[Bryant and Finkel 1981] have described a new algorithm based on a load estimation method, a co-operation policy and a load balancing policy. Job transfers are achieved through process migration. Load estimates are obtained from the remaining processing time of a job. Four different methods are described to estimate remaining processing times. Co-operation policy is used to form temporary pairings between processors. This policy helps to reduce the overhead which is due to periodic unlimited broadcasts. Load estimation policy and the co-operation policy are utilized in the load balancing process. Current load estimates are used to form pairs that differ greatly in load. This algorithm is very complex and it is not clear if the performance gain is worth the added complexity.

[Livny and Melman 1982] have presented three dynamic load balancing algorithms for a homogeneous broadcast distributed system. Each host in the distributed system is modelled as an M/M/1 queue [Kleinrock 1976]. The necessity of load balancing process in a distributed system is indicated with the help of analytical results. It is shown that for more than 10 servers in a queueing system, almost all the time a customer is waiting for service at one server while another server in the system is idling. The three algorithms are

compared with the extreme cases of no load balancing (independent M/M/1 queues) and ideal load balancing (M/M/N queues). For all the three algorithms, expected turn-around time and channel utilization are compared for varying parameters, such as number of servers and server utilizations. Each algorithm performs well under a certain set of system parameters.

[Krueger and Finkel 1984] have proposed a strategy where jobs are assigned to the hosts in such a way that each host has nearly the same workload. The jobs are transferred from an above-average-load to a below-average-load processor.

[Stankovic 1984a] presents three simple adaptive load balancing algorithms. All these algorithms are compared under light, moderate and heavy load conditions. Communication delays across the subnet are assumed to be constant and the task transfer delays are derived from communication delays in the subnet and are proportional to the size of the job being transferred. It was shown that the tuning was necessary for improved performances and should be adaptive. Additional tests were performed by varying the cost incurred by the scheduling algorithms and its effect on the response time was studied.

[Chou and Abraham 1982] developed an algorithm to determine a minimal cost task-processor assignment for a given heterogeneous distributed system. A distributed program is represented by an operational precedence graph that describes probabilistic branching as well as concurrent execution in the program. Dynamic task execution is modelled with a continuous-time discrete-space semi-Markov process with rewards[Kleinrock 1975]. This algorithm is not purely dynamic as it is assumed that all the required data

is available. However, it does capture the dynamic behaviour of distributed programs.

[Eager et. al. 1986] provide the analytic model to compare the performance of simple adaptive load sharing policies with the complex ones. Two simpler load balancing policies are compared with a more complex policy that uses more state information. The important result of this paper is that the complex policy does not perform significantly better than that of the simpler policies. This suggests that a more complex usage of state information is of little benefit. It is also established that less complex policies are inherently less susceptible to instability due to processor thrashing.

[Mirchandaney et. al. 1990] have proposed two adaptive load sharing algorithms for heterogeneous distributed systems. Two types of heterogeneous models are studied. In the first type the hosts have the same processing rates but arrival rate of the local jobs at hosts may not be the same. In the second case arrival rates are the same whereas the processing rates may differ. The two algorithms are called Forward and Reverse probe algorithms depending on whether the algorithm is activated at the arrival of an external job or departure of a completed job at a host. The probe limit of these algorithms is adjusted dynamically to improve the response times and reduce the unnecessary overhead due to frequent probes thus making these algorithms adaptive. These algorithms were analyzed by modelling each host by a Markov chain. The performance of these algorithms is measured analytically and simulations were run to validate the analytical results.

The work was aimed to survey various techniques which have been developed for load balancing in the multiple processor system. Efficient load balancing techniques provide higher performance as these help improve the response time for tasks in multiprocessor systems.

In this chapter, both the static and dynamic scheduling techniques were discussed. In the case of static scheduling, the problem of load balancing is reduced to a problem of mapping tasks onto the available network nodes. In static algorithms, it is assumed that a complete future knowledge of behaviours of the program is available. Despite the assumption of available information about the inter-task communication patterns, the scheduling problem is still difficult, because usually the task communication patterns do not exactly match the network node connectivity. Many researchers have attacked the static scheduling as a graph-theoretic problem. Most of this work is directed towards finding the clusters of tasks with tight coupling by partitioning the graph into a number of subgraphs (equal to the number of available processors). Due to the nature of the assumptions made in static scheduling, it has no potential use in the real systems with tasks showing dynamic behaviour.

The researchers doing load-balancing typically make the realistic assumption that nothing about the future behaviour of a task is known and many diverse techniques have been developed for this type of scheduling. Dynamically created tasks are assigned to network nodes in real-time by a distributed scheduling algorithm executing on the multicomputer network. Most of the algorithms, developed so far, do not take task migration into consideration due to a large amount of overhead associated with it.

Chapter 4

DESIGN AND IMPLEMENTATION OF THE SIMULATION MODEL

4.1 Introduction

This chapter provides the design and implementation details of the simulation model developed for the performance study of different scheduling techniques in a distributed computer system environment. The simulation model so provided can be used to develop new heuristics for efficient load balancing and scheduling of multiple processor systems. All the important features required by several load balancing algorithms are incorporated in the design of this model. The design efforts are concentrated at capturing the characteristics of both the deterministic and the probabilistic models for load balancing. Routines are also provided to simulate the communications and task transfer facilities in a dynamic load balancing enviroment.

Several simulation modelling approaches are available for studying the performance of computer systems. The two most commonly used approaches are the event-oriented and the process-oriented models. The event-oriented simulations are generally used for small-to-medium scale models, whereas the process-oriented simulations are preferable for large scale models. The design approach followed is based on an event-oriented simulation model. The entire simulator is implemented in the C programming language under the Unix system environment. In addition, interfaces to some NAG library routines are provided. These routines were used for generating the required probability distributions and to solve simple linear programming problems.

Input and output interfaces for the simulation model are not sophisticated yet serve the required purpose with ease and efficiency. A special piece of software was written to provide a visual interface for displaying the Gantt charts in a deterministic model of scheduling. This software was coded using the Sunview pixrect library routines. All the simulations presented in this thesis were carried out using this model.

4.2 Simulator Design Issues

The design of a simulator and the level of detail needed for its implementation, called system abstraction, is very closely related to the nature of the problem and the data available to represent the work at this level. In this study, the nature of the problem is sub-divided into two main parts: i) task allocation and scheduling and ii) distributed computer system and communication parameters. Whereas, the workload characterisation is to represent the task system and its related parameters.

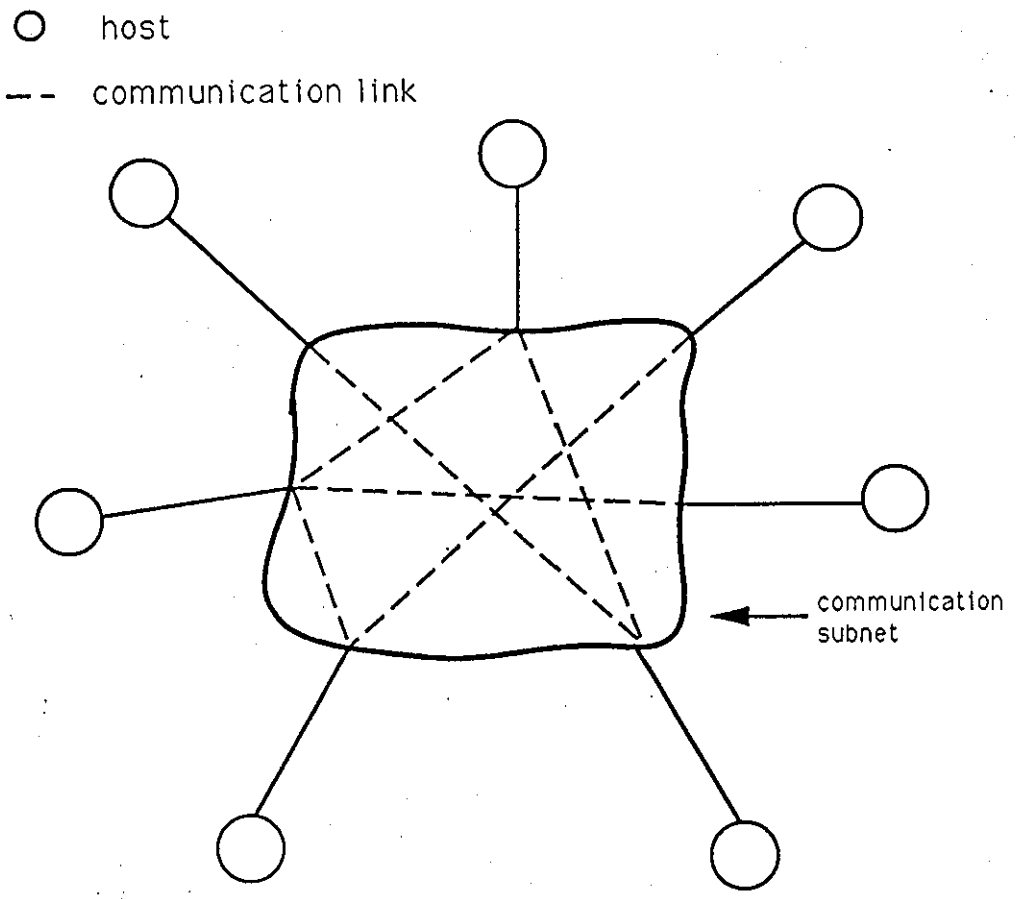

Figure 4.1: A Distributed Computer System

4.2.1 System Abstraction For Simulation

System abstraction is a very important step in the design of a simulation model. This specifies the facilities and operations involved in accomplishing the work at a level of detail suitable to the problem under study. The operations and other details given in this subsection are valid for both the static and dynamic load balancing paradigms.

A Distributed Computer System

The distributed computer system under consideration is illustrated in Fig.4.1. A distributed system is a collection of a number of hosts[1] inter-connected by a reliable communication network. Each host has its own central processing unit (CPU), Input/Output (I/O) and memory etc. In this study, a distributed computer system is represented by a graph G(H,E), where a node 'H' represents a host and an edge 'E' represents a communication link in the system. It is assumed that the 'n' hosts are physically and functionally identical. It is also assumed that each host has an unlimited amount of memory, therefore any number and size of tasks can be handled. In this model, each host is approximated as an infinite queueing model. The model is illustrated in Fig.4.2.

The simulator treats a processor as an independent entity represented by a data structure. There is a set of attributes related to this entity and a set of events are defined to effect the state of this entity. These sets of attributes and events are described next.

```
processor{
        integer     server;
        integer     load_vector[n];
        integer     linknum;
}
```

[1]The terms processor and host are interchangeably used in this thesis

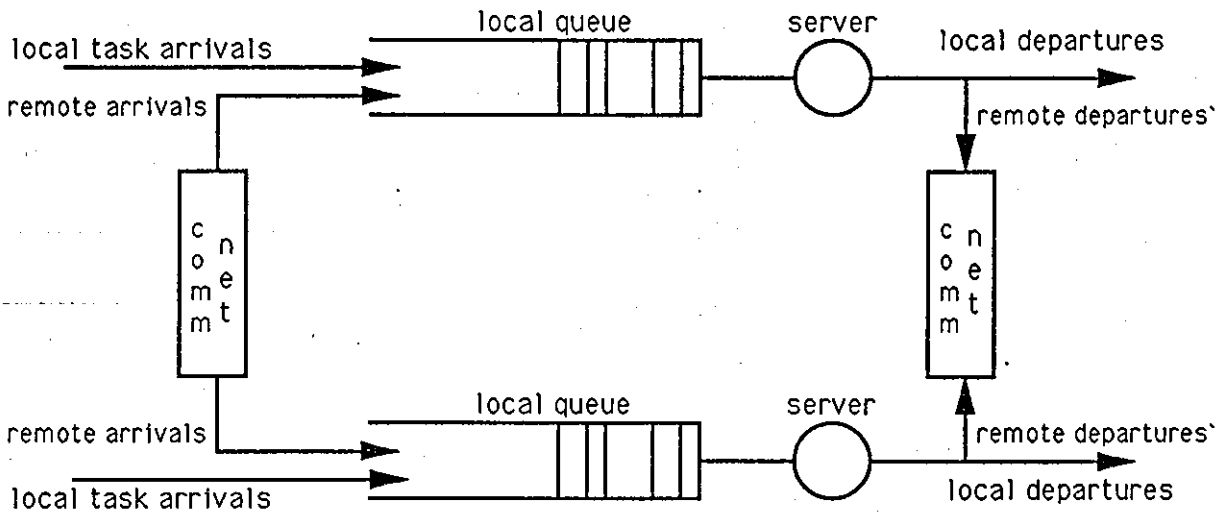

Figure 4.2: An Infinite Queueing Model

The attribute server is a unique identifier for each processor. The Second attribute *load_vector* is used to save the load information received from all other processors in the system. Finally, the attribute *linknum* is defined to store the total number of communication links attached to this processor.

The events related to the processor are *schedule, request,* and *release.*

Schedule Event: The arrival of a local task is announced through the schedule event. All the remote task arrivals are placed at the end of the waiting queue if the processor is busy.

Request Event: Each arriving task on a processor requests an immediate service.

If the requested processor is idle, it starts processing the task otherwise the task is placed at the end of the processor queue. The placement of the task in the processors' queue depends on the priority discipline used for the scheduling of tasks on the processor.

Release Event: Whenever a processor finishes processing a currently executing task, it calls the release event. This event updates the state of the processor and schedules a task from the head of the processors' waiting queue.

The Communication Network

Here, the level of detail used in simulating the communication network is described. Considering the distributed load balancing, it is very important for a simulator to provide mechanisms for studying the overheads invloved in distributing the load information and transferring the tasks across the underlying communication network. It was assumed that the processors are inter-connected in an arbitrary fashion. In this implementation of the simulator, only periodic message broadcasts were simulated. However, other message communication methods can be implemented with minor modifications.

The data structures used for the two simulated entities, a link and a packet are given below.

```
link {
        integer     end1;
        integer     end2;
        integer     busy;
```

```
        integer     link_capacity;
}

packet {
        integer     type;
        integer     src_processor;
        integer     sequence;
        integer     msg_len;
}
```

where, *end1* and *end2* are used to identify the processors at the ends of the communication link. The attribute *busy* is used to check the status of the communication link. The final attribute *link_capacity* specifies the transmission rate of the communication link in bits/second. The data structure packet is used to carry the messages across these communication links. The *type* of the packet is used to take different actions on the receiving processor depending on the value of type. Other attributes specify the originating processor of the packet, the packet sequence number to avoid repeated messages and the length of the message contained in the packet.

Task System

Here, it is described how a task is viewed by the simulation model and what operations are performed on it. The operations specified for tasks are the same in both the static and the dynamic scheduling environments. Specifying the characteristics of the task system for scheduling is called the

workload characterisation. A separate workload characterisation is required for static and dynamic load balancing paradigms. The data structure used to represent a task in the simulation system is given below.

```
task {
        integer     id;
        integer     status;
        integer     processor;
        integer     service_time;
        integer     arrival_time;
        integer     start_time;
        integer     finish_time;
}
```

The attribute *status* indicates if the task was executed locally or remotely. The *processor* attribute stores the processor identification where the task was executed. The last two attributes are used to store the times when the task started the execution, which is different than the task arrival time, and when the task finished executing. The service time required for the task is given by the *service_time* attribute. Other important characteristics of a task system consist of the inter-task communication times and the precedence relations among the tasks. These are described in the next section.

4.3 Simulation Model For Static Load Balancing

This section provides the implementation details of the simulation model. The simulation model described here can be used for performance evaluation of static load balancing strategies. Several extensions were made to make this model suitable for dynamic load balancing strategies. These extensions will be described in the next section. Several components of the simulator described in this section are common to the operation of both the static and dynamic load balancers.

The entire simulator can be sub-divided into the following major modules.

- Precedence Graph Input and Initialization
- Processor and Network Initialization
- Data Input and Error Checking
- Routines for Queuing and Event Handling
- Main Scheduler
- Output table giving required performance measures
- Gantt Chart Display

4.3.1 Precedence Graph Initialization

With the advancement of distributed processing, there have been significant advances in the area of partitioning the large programs into smaller modules to run on several machines in parallel. Throughout this discussion, a program module will be referred to as a task. These tasks can be considered as a set of co-operating processes designed to solve a well-defined problem.

It was assumed that the order in which these tasks are executed and the inter-task communication costs are known. It was further assumed that the size of each task is known, irrespective of the fact that it may or may not be the suitable grain size for the most efficient solution. The best way to describe this set of tasks is in the form of a Precedence Graph. The simulator was designed to obtain the precedence relation information from an input file called PREC_GRAPH. For each task 'T' in the precedence graph, there is one line that gives the dependencies for this task and all the other tasks. A small example file is given below and the corresponding precedence graph is shown in Fig.4.3. An alternate method which generates the task graphs randomly was also designed. In this case, a fixed number of nodes were randomly distributed among different levels and the nodes at each level were randomly inter-connected. The node execution times and the communication times were randomly generated.

Input File Format :

$T1$

$T2$

$T3 > T1 : C_{13}$

$T4 > T1 : C_{14}$

$T5$

$T6 > T3 : C_{36}, T4 : C_{46}$

$T7 > T4 : C_{47}, T5 : C_{57}$

$T8 > T4 : C_{48}, T6 : 68, T7 : C_{78}$

In the file description given above, tasks T1, T2 and T5 are independent of any other tasks and are ready for scheduling at any time, whereas task T3

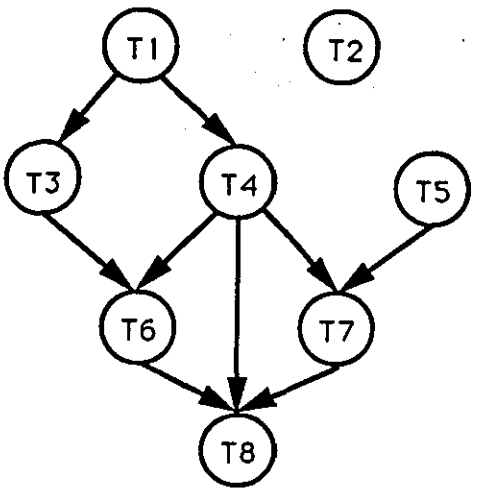

Figure 4.3: An Example Precedence Graph

will have to wait for task T1 to finish before it can start execution.

Another parameter stored in this file was the inter-task communication costs. Each task dependency carries some communication cost. These communication costs are specified next to each dependent task separated by a colon. These values are read from the file and stored in the form of a connectivity matrix C[i,j] shown diagrammatically in Fig.4.4.

4.3.2 Processor And Network Initialization

The problem of static scheduling in a distributed system is simply described as a mapping function from a set of tasks onto the available number of processors. Therefore, it is very important to consider the underlying processor network architecture at the time of task scheduling. When the distributed computer system is mentioned, it means a multiprocessor or a multicom-

$$C[i,j] = \begin{bmatrix} 1 & \text{for j is a predecessor of i} \\ & 1 \leq i,j \leq m \\ 0 & \text{for j is not a predecessor of i} \\ & 1 \leq i,j \leq m \end{bmatrix}$$

Figure 4.4: A connectivity matrix for inter-task communications

puter system. The type of architecture that is currently being considered, is that of a multicomputer system in which there is a loose coupling among the individual computers. However, everything described here applies equally well to multiprocessing systems with message passing as a communication primitive. In the current implementation of the simulator, it is assumed that the processors are inter-connected in an arbitrary configuration. It is assumed that all the processors are homogeneous (identical). The assumption of homogeneous processors as opposed to the heterogeneous ones reduces the complexity of the scheduling algorithm and helps to concentrate on the actual problem of scheduling. The simulator does provide the facility for considering the heterogeneous processor case, though the heterogeneity is limited to the processing speeds only. All the information about the inter-connection topology and the link bandwidths is stored in the file called NET_CONFIG. This file also stores the distance between each two nodes measured in hops. At the time of initialization, the simulator reads this distance into an array hops[i,j], where each element of the array gives the distance from processor

'i' to processor 'j'.

4.3.3 Data Input And Error Checking

The execution times for all the tasks and the inter-task communication times are entered at run time. The simulator provides separate routines for error checking for the data entered at the run time and the data stored in the input files. These routines provide the mechanism for checking the unrealistic or false data and prompts for re-entering the correct values. The values for task service times and the inter-task communications are taken from the profiles of some real programs using finite element techniques. These times can be generated from different probability distributions. Routines for random, exponential, Erlang, and hyper-exponential variate generation are also provided in the simulator.

4.3.4 Queueing And Event Handling

This section describes two basic and very important functions used in the implementation of the simulator. These functions are related to the queueing management of the processors and the event handling mechanism. In the simulation model, a processor comprises a single queue and a single server. Whenever a reservation for a processor is requested for a task, if the processor is free then it is reserved for the requested task. This reservation of the processor by the task implies that the task execution is started. If the requested processor is busy then the task enters the processors' queue. When the executing task releases the processor, the task at the head of the queue re-issues the request for the processor. Queues are ordered on the basis of priority. If the two queue entries have equal priority, these are ordered on a

first in, first out basis. There is no limit on the processor queue lengths.

Another data-structure of important concern is the event-list. Since it is a discrete event simulator, all the events happen in discrete time. Schedule() and cause() are two routines used to add and remove the entry from the event-list respectively. An event-list entry contains the event number, the inter-event time, and a task associated with the event. The event number indicates the type of event associated with this event-list entry, i.e; task execution start or task execution finish etc. The event-list is ordered in ascending values of the event occurrence times. The entries with the same event occurrence times are ordered on the basis of arrival. Simulated time is represented by a global variable using the unit time approach. The simulator increments the time by small increments and then checks to see if there are any events that can occur. Further information about the management of the simulation time is described in the next section.

4.3.5 Main Scheduler

This is the main() routine for the simulator, where the actual heuristic for scheduling is implemented. The major steps performed in the scheduler routine are described below.

- Call the initialization routine for setting up the simulation environment. Reset the simulation clock.
- Take the tasks from the ready queue (these tasks are ready to execute at the start and are not waiting for any other tasks to finish) and schedule for execution. Record the simulation time as the task execution start time.

The following steps are repeated until all the tasks finish execution.

- Check for the tasks in the waiting queue (these tasks are waiting for other tasks to finish) If a task is ready then calculate the tasks execution start time and schedule for execution. A waiting tasks' execution start time is calculated by adding the largest finishing time among its predecessor tasks finishing times to any communication delays among the two tasks.

- Next step is to remove an entry from the head of the event-list and check for the event type and the associated task. There are three different types of events corresponding to start, continue and finish execution of the associated task. If the event is start execution, then the following steps are carried out to find the suitable processor for scheduling this task.

  Next step is dependent on the task allocation strategy used for load balancing.

- In the simplest heuristic, a randomly chosen processor is considered for execution. Where the distribution of the tasks on the processors is uniform.

- Once the task is allocated to a processor then it is scheduled for the execution event, where the simulation clock is incremented on each event occurrence and all the executing tasks are checked for the finish times. Whenever a tasks' finish time gets equal to the current simulation time, this task is scheduled for the finish execution event.

- In the finish execution event, the finished task releases the processor. If the processor queue is not empty then the task at the queue head is removed and scheduled for the start execution event on that processor. If the processor queue is empty then the processor is left idle.

- The simulation program terminates when all the tasks finish execution.

4.3.6 Output Table

At the end of the simulation run, each processors' statistics are printed in a table. For each processor, mean busy period, utilization and the average queue length are measured and printed in this table. These measures indicate the load distribution on the network of processors. The final value of the simulation clock is also printed.

4.3.7 Gantt Chart

The final task schedule for all the processors is best represented by a Gantt chart. Gantt chart provides a very clear picture of each processors' busy and idle periods. It also shows the start and finishing times for each task.

Special software was written to provide this facility for the simulator. This software is developed using the Sunview[2] graphics routines on a Sun Sparc station. The program was written in *C* language using the SunView tool layer or application layer. The application layer provides a set of high level objects that can be used for designing various applications and tools. The implementation of these objects is similiar to the objects used in an object-oriented programming environment. The objects provided by the SunView

[2]SunView and Sun sparc-station are trademarks of Sun Microsystems, Incorporated.

application layer include windows of different types, menus, and scrollbars, etc. Most of the functionality of the window system is based around a window abstraction. In this section, the functionality and the characteristics of windows used in implementing the Gantt chart display will be described. Several routines are provided to access the screen for drawing. The interface to these routines is provided through the pixwin library package. All the functions described in the pixwin library are based on the pixrect library functions.

The Pixrect graphics library contains routines for performing low-level raster operations to develop device-independent applications for Sun products. It is a low-level graphics package sitting on top of the display device drivers. For further explanation, it is important to define some important terms.

Pixel: A *pixel* is the smallest possible picture element that can be displayed. A pixel is defined by an address that specifies its location on the screen and a value that controls the color display. The pixel address is absolute or relative to some other rectangular subregion of the screen.

Bitmap: A *bitmap* is a rectanglar region on the screen. A bitmap can consist of one to several pixels in it. A pixrect bitmap can be up to $2^{15} - 1$ pixels wide and up to $(2^{15} - 1)$ pixels high.

Pixrect: All the routines provided in the Pixrect library operate on an object called a *pixrect.* A *Pixrect* is defined similar to an instance of a class used in an object-oriented programming environment. It consists of the data (bitmap data) and the operations to be performed on this data.

In drawing the Gantt chart, the following basic steps were performed.

- Creating a window for drawing and specifying its attributes.

- Drawing the Gantt chart and adding text to it.
- Closing the window.

In creating a window for drawing the Gantt chart, the following routines were used.

window_create()
canvas_pixwin()

In the process of creating a window for drawing the Gantt chart, the first step was to create a base frame using the window_create() routine. A frame itself is not very useful for drawing, however, it provides the facility to manipulate the window by changing its size or moving it to a different location, etc. The routine window_create() is called once again to create a canvas[3]. The difference between the two calls to the window_create() routine is that of the parameters passed to it. In the first call, a pointer to the structure "FRAME" was passed. Whereas, in the second call, a pointer to the structure "CANVAS" was passed. These structures are specified in the header files sunview.h and canvas.h respectively. After these windows were created, the next step was to provide the access to the canvas for drawing the Gantt chart. For this purpose, canvas_pixwin() routine was used to obtain a handle for the canvas, called a pixwin. This pixwin was used in the subsequent routines for drawing. The routines from the pixwin library were used to access the pixels for drawing on the canvas. The two basic routines used for this purpose are given below.

[3]A canvas is a window into which one can draw

pw_vector()

pw_text()

Both these routines require the pixwin of the canvas as an argument. This pixwin is returned by a prior call to canvas_pixwin(). The routine pw_vector draws a vector from (x1,y1) to (x2,y2) in the addressed pixwin. These coordinate values are passed as arguments to the pw_vector(). All the boxes representing the task execution times and the diagonal lines representing the inter-task communications are drawn using this routine.

Since the task execution times and the inter-task communications had a large variance, it was not possible to provide a single scale for drawing the Gantt chart. The Gantt chart values cannot be directly entered as the coordinates of the pw_vector() as there is a limit on the maximum value of x and y that can be drawn on the canvas. Similarly, to draw a task of execution time equal to 1, an x value of 10 is required to show it clearly in the Gantt chart.

The next step was to display the text corresponding to the times of task execution start, communication delay start, task execution end, and the communication delay end. This was done using the routine pw_text(). This routine takes a string of characters as an argument and displays it onto the addressed pixwin. To display any text, first it was read into a character string using the sprintf() routine and later the pointer to this string was passed to the pw_text(). This routine also requires a (x,y) value to determine the

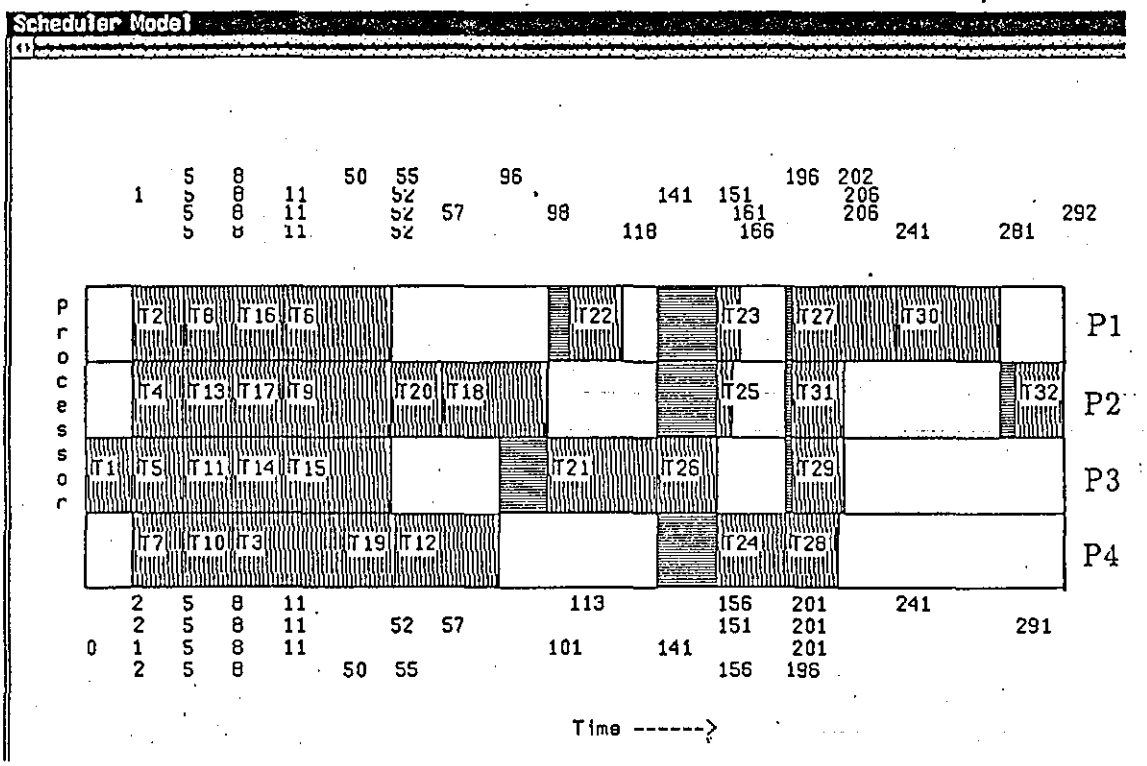

Figure 4.5: An Example of Gantt Chart

location of the text.

A Gantt chart for a large number of processors and tasks can be displayed on a window of full screen size. An example of a Gantt chart is shown in Fig.4.5. The boxes filled with the diagonal lines indicate that the processor is busy, whereas the empty boxes are the idle period. The boxes filled with horizontal lines indicate the communication delays suffered by a task before it starts execution. The start time for each task is displayed below the Gantt chart, where the finish times are given above. The inner-most lines give the times for processor P1 and the outermost lines give the times for processor Pn (for n-processor Gantt chart).

4.4 Extensions To The Simulator For Dynamic Load Balancing

In the last section, it was assumed that the simulations will be carried out for a deterministic model of load balancing. The workload used in simulations was determined at the time of initialization and no information policy was used to distribute the current state of each processor. To simulate a dynamic load balancing algorithm, it is very important to characterize the load in a more realistic and non-deterministic way and to provide mechanisms for load information distribution. Since, the work presented in this thesis used a queueing theoretic model to study the performance of dynamic load balancing strategies, therefore the parameters used in simulations were derived from the queueing theory.

Inter-arrival and Service Times: Poisson distribution is very commonly used in the analysis of queueing systems. It is assumed that the task size and the task arrival distribution are unknown at the time of making a dynamic load balancing decision. The memoryless property of a Poisson distribution makes it suitable for simulating the task arrival and service times in such systems. For these simulations, task service times were generated from Poisson distributions, whereas the task inter-arrival times were generated from negative exponential distribution. NAG[4] Fortran Library routines G05DBF and G05ECF [NAG 1990] were used to generate pseudo-random numbers from negative exponential and Poisson distribution respectively. Interfaces were provided to call these routines in the 'C' programs.

Task Transfers: While simulating the static load balancing environment,

[4]NAG is a registered trademark of Numerical Algorithms Group Limited

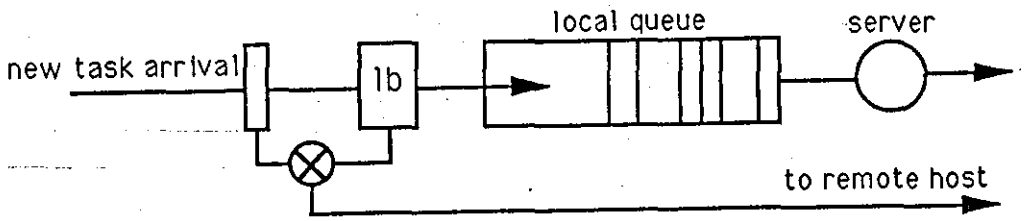

Figure 4.6: Transfers before scheduling

no consideration was given to task transfers as the task placement decisions are made only once during the lifetime of a task. In dynamic load balancing, a task may be transferred to some other processor for an improved response time. There are two possible ways to simulate the task transfers. One way is to only transfer newly arrived tasks from a heavily loaded host to a lightly loaded host. In this case, once the hosts are identified as heavily loaded or lightly loaded, a fixed number of newly arrived tasks is transferred to balance the load of the system. Second method is to remove a task from the queue of a heavily loaded host and to place it in a lightly loaded hosts' queue. Both these methods were implemented in the simulation model. The cost of transferring a task can be estimated from the mean service time of the task and the communication cost of the link between the two hosts. Both the methods of transferring a task are depicted in Fig.4.6 and Fig.4.7.

Simulating The Probabilities: The dynamic load balancing algorithms described in chapter 6 use probabilities for transferring tasks among the hosts. To simulate the probabilities accurately is not an easy problem.

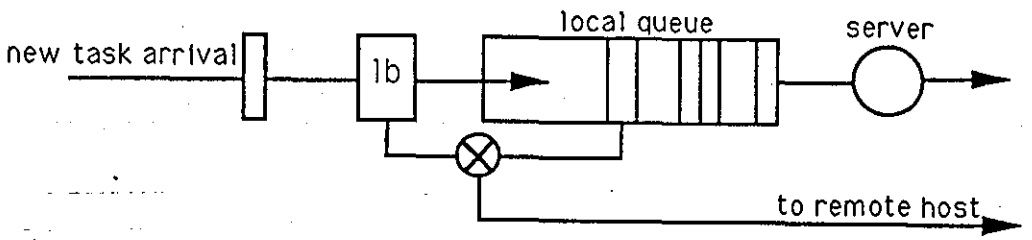

Figure 4.7: Transfers after scheduling

However, a simple method was adapted in the simulations for generating the approximate probabilities. The probabilities for transferring the tasks between any two hosts were multiplied with a constant value of 10 (a value obtained as an estimate of average task arrivals during a specified time interval) to obtain the number of tasks that need to be transferred for a transfer probability of 0.1. The probability of processing a task locally was also determined from these probabilities. Whenever a new task arrives, it is transferred to a destination host with the highest transfer probability. After each task transfer to a host, a value of 0.1 is subtracted from the transfer probability to that host. Once all the probabilities reach a value of 0, the old probability values are restored and the process is repeated. New probabilities are computed at each execution of the load balancing algorithm.

Message Broadcasts: The dynamic load balancing algorithms execute in a distributed fashion on each host in a distributed computer system. These algorithms use global state information to make load balancing decisions. Each host has a load table which contains the most recent information of load

state of other hosts in the system. Load information messages are exchanged among the hosts to keep these tables updated. Various communication protocols are used for this purpose. It is very important to study the overheads of such protocols on the load balancing algorithms. This simulator provides a simple periodic broadcast protocol to distribute each hosts' load information. Any other protocols can also be incorporated into this simulator with minor modifications. The percentage of the overhead introduced by a protocol on each communication link is also measured. The additional overhead on the communication links due to task transfers was also simulated. For the given communication link capacities, the amount of time that a link was utilized by a load balancing algorithm was measured to obtain estimates of this overhead.

Simulation Transient Removal: The initial part of the simulations is called the transient state or the warm up state. The problem of identifying the transient state and to delete the results collected during this period is called the transient removal. Most simulations are designed to study the system performance under stable conditions. In simulating the task scheduling problem using an infinite queueing model, it is important to wait till the host queues are filled and the system is operating in a steady-state. The effect of initial transient for scheduling in a distributed computer system is illustrated in Fig.4.8.

There are several heuristic methods for the transient removal [Jain 1991]. Among these various methods, the first proper initialization method was chosen for transient removal. To implement this method, the queues of all the hosts were filled with some tasks before starting the scheduling simulation. The number of tasks was determined from the previous simulations. This

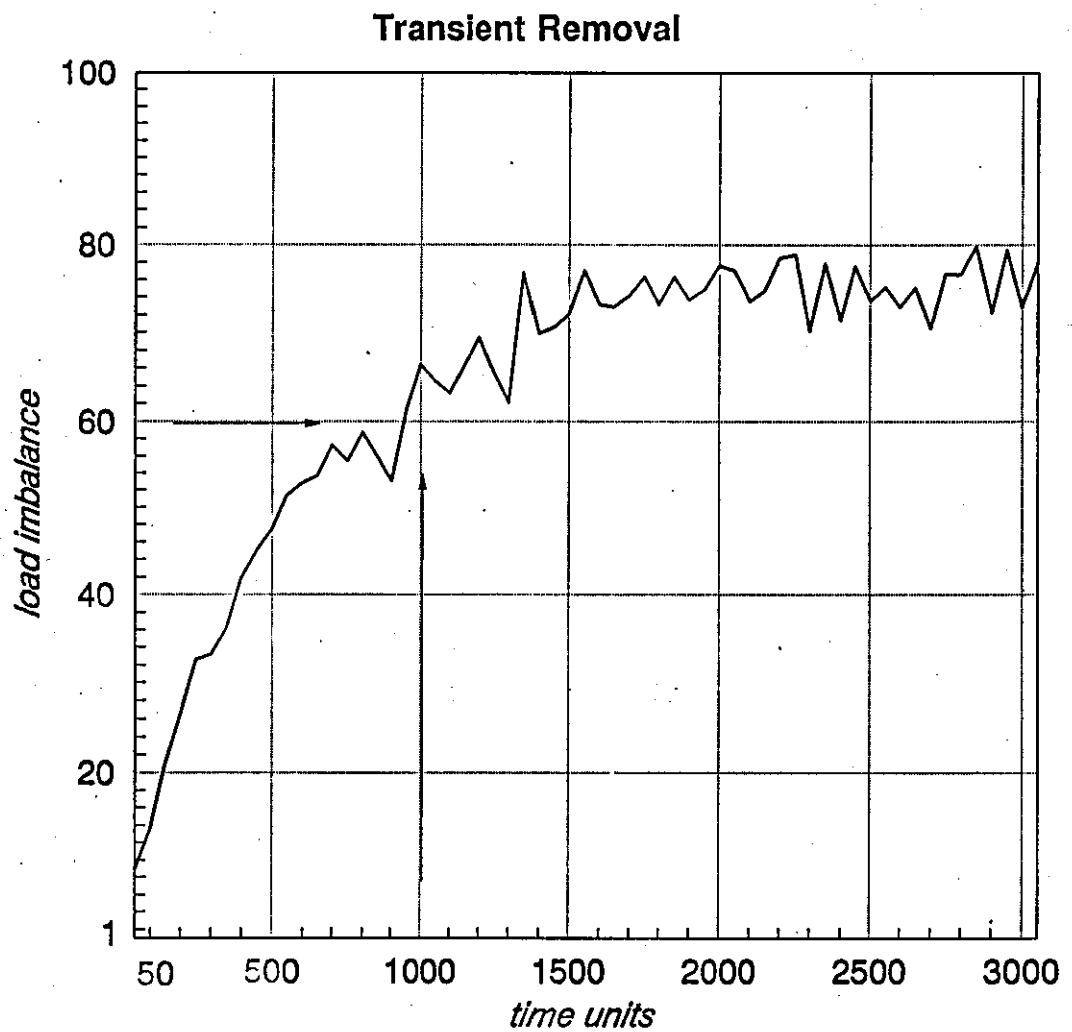

Figure 4.8: The Initial Transient Removal

method was rejected as it complicated the measurement of response times of the tasks in the system. Later, the method of initial data deletion was selected and implemented. The length of the transient interval was determined by studying the previous simulations.

Confidence Interval: As mentioned earlier, all the performance measurements were collected during the steady-state operation of the system. It is expected that the averages obtained during the steady-state do not change considerably. However, the randomness in the parameters used for the simulations does cause the observations to change for different runs even during the steady state. A confidence interval is used to specify the range in which the sample mean lies to the actual mean. To reduce the effect of randomness, several samples were obtained for each simulation run. A different seed value was used for each random number generator to reduce the effect of correlation. A large number of samples (30) was used to obtain high confidence levels. The confidence interval was estimated using the 't' distributions for the sample mean.

$$Confidence\ \ Interval = \pm t\alpha/2; N-1\ \ s/N^{1/2}$$

where α is called the confidence coefficient. For a 95% confidence interval, α has a value of 0.05. $S/N^{1/2}$ is the standard deviation of the distribution of the sample mean. For N samples of X and a mean sample value of $\overline{X}$, it is given as,

$$s^2 = \sum_{i=1}^{N} (X_i - \overline{X})^2/(N-1)$$

4.5 Performance Metrics

A number of various performance metrics are required to evaluate the performance of load balancing algorithms in a distributed computer system. In a static task allocation and scheduling system, a deterministic model with a limited number of processors and tasks is used. Therefore, the performance of such a system can be best studied with the use of a Gantt chart. The Gantt chart described in section 4.3.7 was used to evaluate the performance of static task allocation strategies presented in chapter 5. A Gantt chart specifies the distribution of the tasks on the processors and the total completion time of the task system on a given distributed system.

Gantt charts cannot be readily applied to measure the performance of dynamic load balancing algorithms. The overheads involved in dynamic load balancing and the order of task execution cannot be easily presented by Gantt charts. The performance metrics used in this study are the *mean response time, the load imbalance* and the *percentage task transfers.* The mean response time of a task is defined as the amount of time a task stays in the system. It consists of the processing time, queueing delays, and the possible communication delays in transferring a task during its lifetime. A major objective of this study is to reduce the mean response time of a task.
A simple mathematical model for calculating the mean response time is given below.

The parameters used are:

a_j^q = Arrival time of task j at a local processors' queue.

d_j^q = Departure time of a task j from a local processor.

a_j^s = Arrival time of task j at the server.

x_j = Communication costs for transferring a task j to a remote processor for execution and receiving the results.

s_j = Service time for task j.

n = Total number of processors in the system.

m = Number of tasks serviced on one processor.

The placement of an incoming task is decided by the load balancing policy. An incoming task from a user terminal is entered into the queue of a local processor or transferred to a remote processor for execution.

$$a_j^q = a_j^q + [1 - \delta_j]x_j/2$$

A processor is represented by an infinite queueing model with a single server and a single queue (M/M/1). If there is no other task in the processors' queue, then the task receives an immediate service. If the server is busy, then a task has to wait w_j time, depending on the number of tasks in the servers' queue, before it receives service.

The time of the tasks' departure from the queue is given by,

$$d_j^q = a_j^q + w_j$$

This is also the tasks' arrival time at the server,

$$a_j^s = d_j^q$$

After a task has received service and the results, in case of a remote execution, are returned, the departure time of a task from the server is given by,

$$d_j^s = a_j^s + s_j + [1 - \delta_j]x_j/2$$

The response time of a task j on any processor can be given by,

$$r_j = d_j^s - a_j^q$$

For m tasks arriving on each processor and n processors in the distributed computer system, the mean response time (R) is given by,

$$R = \sum_{i=1}^{n} \sum_{j=1}^{m} r_j / \sum_{i=1}^{n} m_i$$

Next, the *load imbalance* is defined to observe the difference of loads among the processors in the distributed system. It is defined as the average of the root mean square difference in the loads of every two processors in the system at a given instant of time.

$$load \;\; Imbalance(t) = \{\sum_{i \neq j} [load_i(t) - load_j(t)]^2 / [n(n-1)/2]\}^{1/2}$$

where *load* of each processor is determined by multiplying the queue length of a processor with the mean service time.

Chapter 5

A STUDY ON TASK ALLOCATION IN DISTRIBUTED COMPUTER SYSTEMS

5.1 Introduction

This chapter concentrates on the problem of static task allocation in a distributed computer system. The problem of scheduling in distributed systems is divided into two major categories, i.e; static scheduling and dynamic scheduling. The static scheduling is best represented with a deterministic model in which all the parameters related to the task behaviour, inter-task communication and the inter-processor communication are assumed to be known 'a priori'. Furthermore, in the static scheduling, all the scheduling decisions are made before any task starts execution. In the case of dynamic

scheduling, the task behaviour and the communication patterns are only known at run-time and load information is exchanged periodically or on demand amongst the processors for load balancing purposes. Static scheduling can be effective for dedicated applications in which the task behaviour and inter-task communication can be approximated and the underlying processor and network resources do not change. In this study, the term distributed system is used for any multiple processor or multiple computer system interconnected in an arbitrary fashion. The task allocation strategies are used to map a given application program onto a given distributed system. Such strategies are particularly useful for systems running dedicated distributed processing applications [Enslow 1978]. The task allocation strategies are used to minimize the completion time of a single application at a time. Each application is considered independently of the other application programs for scheduling purposes. An application program consists of a set of independent and inter-communicating program modules called tasks. These tasks are represented by a precedence constrained graph [cf. chapter 4]. Throughout this chapter, the word scheduling stands for both task allocation and task scheduling. Task allocation helps in balancing the load on given processors and minimizing the communication costs, whereas task scheduling may improve the overall completion time for a given task allocation scheme.

Several different models have been proposed to solve the problem of static scheduling. The major approaches used for solving the static scheduling problem are graph-theoretic, list scheduling, clustering and queueing theory [cf. chapter 3]. The assumptions made about the task system differ for all the above mentioned approaches. The task system used for scheduling in this study is represented with a precedence constrained graph with varying

inter-task communication times. Such a task system is best solved by using the list scheduling heuristics.

The optimal task scheduling that minimizes total execution time is known to be NP-complete even for a set of tasks that do not communicate [Ullman 1975]. The purpose of this study is to find new heuristics that can be applied towards finding near optimal schedules. Finally, the effectiveness of some simple heuristics in improving the overall completion time of the task schedule is demonstrated through examples. The final task schedules for different task allocation heuristics are represented by Gantt charts.

5.2 Assumptions And Definitions

It is assumed that all the necessary information related to the application is available in the form of a precedence graph (DAG). This information can be extracted from the original application program [Ward and Romero 1984]. In this precedence graph, the weight of each node represents the size of the task and the edges represent the communication delay in transferring the control and synchronization information.

It is assumed that the topology of the processor inter-connection network is known. It is a loosely coupled multiprocessor system with identical processors. It is also assumed that each processor can overlap communications with the processing. Furthermore, all the tasks running on the same processor incur a negligible (zero) communication cost.

Since the purpose of this study is to investigate and propose an efficient heuristic for the allocation of task systems with varied computation to com-

munication ratios, therefore different communication costs are assigned with each edge of the precedence graph. Several other researchers have assumed similar computational models with identical communication costs [Shirazi and Wang 1988] [Rewini and Lewis 1990].

Next, some definitions are described which will be used for discussions in the remaining sections.

- Length of a path is the summation of execution times, represented by a node, and the communication delays, represented by an edge, along the path.

- The Longest Path for a node X is a path from the root node of the precedence graph to the node X, whose length is the longest path among all possible paths from the root node to X.

- A Predecessor of a task X is a task Y that finishes its execution before the task X can start execution. Task X can be termed as the successor of task Y.

- An immediate predecessor (successor) of a task X is a task that is one level above (below) the task X in the precedence graph.

- A task is said to be *ready*, if it is ready to be assigned to a processor. Such a task has no parent tasks or all of its parent tasks have already been assigned to some other processor for execution and the communication time between this task and its parents have elapsed.

- The remaining processing time of a processor i, RPT_i, for $1 \leq i \leq$ *no. of processors*, is the total time needed for the processor i to finish

all the tasks assigned to it up to the current time. It can also be seen as an estimate of the current load on a processor.

5.3 The Task Allocation Heuristics

In the current implementation of this simulator, the task allocation heuristics are static and the task graph is assumed to be deterministic. Therefore, this scheduler is useful for compile time partitioning and scheduling [Sarkar 1989]. Using the simulator described in chapter 4, some simple heuristics were incorporated and their effect on the schedules' completion time was studied. The two heuristics called the *simple load balancing* and the *precedence constrained load balancing* were proposed and their performances were compared with each other and with a uniform load balancing case. In the case of uniform load balancing, each ready task was allocated to a randomly selected processor. The random selection was uniformly distributed among all the processors. All the tasks in the queue of a processor are executed in the First-in-First-out order. The other two heuristics are described below.

5.3.1 Simple Load Balancing Heuristic

The simple load balancing derives its name from a very simple heuristic used to allocate the tasks of a precedence graph to a given set of processors to distribute the computational load in the most efficient way. The aim of this heuristic is to minimize the completion time of the schedule by distributing the load evenly across the distributed system. This heuristic does not make any effort to reduce the communication delays by considering the inter-task communications at the time of making a scheduling decision. However, it does take the inter-processor communications into account when more than

one processor has the lowest accumulated time.

In most of the task allocation problems where the communication delays are not considered or fixed communication delays are considered irrespective of the task allocation, the level of a task in the precedence graph is fixed [Adam et. al. 1974] [Coffman and Denning 1973]. This level is determined for each task by calculating the length of the longest path from this task to the exit task. When the communication delays are effected by the location of a task on a processor, the level of a task cannot be determined in advance. The heuristics described here make use of the co-level of a task which is determined after all the preceding tasks are allocated and their inter-communication times are known. A co-level defined in [Coffman and Denning 1973] is the same as the length of the longest path for a task defined in the previous section.

The major steps of this heuristic are outlined below.

- Input a precedence graph with n tasks and their inter-communication times. Also, input the parameters related to the communication network.
- All the tasks without any precedent tasks are kept in the ready list. $R = \{T_i \mid$ all the tasks preceding task i have already been allocated$\}$
- If ready list is not empty repeat the following.

  - Dequeue a task T at the head of the ready list.
  - Find a set $S = \{P_i \mid P_i = min(RPT_i)$, for $1 \leq i \leq n\}$
  - If $\mid S \mid > 1$

{Allocate task T to processor P in $S \mid P = min\{hops(P, P_{pred})\}\}$,
where, P_{pred} is the processor where a task preceding task T
is scheduled.

- Allocate task T to the only processor in S.
- After the communication time between the task T and any of its children has elapsed, remove the edge from the task T to that child.
- Remove task T from the ready list.
- Check for any other ready tasks and update the ready list. Repeat.

- Output the Gantt chart and other performance statistics.

The simple load balancing heuristic allocates a new task to the most lightly loaded processor available in the system that time. It does not consider the inter-task communication delays at the time of making the scheduling decisions. If there is more than one processor with the same minimum load, then it chooses a processor that is minimum hops away from the processor where its precedent task is allocated.

5.3.2 The Precedence Constrained Scheduling Heuristic

This heuristic is designed to efficiently allocate the task systems with high inter-task communication costs on a set of available processors. It is commonly proposed to schedule several precedent tasks of a task on the same processor. Based on the observations from different schedules produced for a number of applications, it was noticed that in most of the cases there are only one or two tasks that produce the long delay paths. If a new task is scheduled on a

processor with the longest delay precedent task, this task can start execution immediately after the preceding task finishes without any communication delays. Here it must be noticed that all the other precedent tasks to this task had finished earlier and the communications were completed.

Intuitively, it might appear that this type of heuristic alone may considerably improve the overall completion time of the schedule. From the experiments, it was noticed that this type of heuristic tends to imbalance the load on the processors. Though it works better than the one in which all the precedent tasks are scheduled on the same processor.

A number of dynamic load balancing algorithms use a parameter called the load threshold to determine a heavily or a lightly loaded status of a processor [Zhou 1988] [Eager et. al. 1986] [Boutaba and Folliot 1990]. So far, the idea of determining the load threshold for load balancing is only applied in dynamic load balancing algorithms. In this study, a method of determining the load threshold from the task graph for application in static allocation strategies is defined. There is a trade-off between the maximum parallelism that can be achieved and the communication costs. In trying to achieve the maximum parallelism by distributing tasks on different processors, one might end up in high communication costs sometimes resulting in communication network congestion. On the other hand, an attempt to schedule all the related tasks on the same processor to avoid the communication costs might result in overloading some processors and a poor utilization of the others. It is, therefore, necessary to incorporate heuristics for minimizing the communication costs and also to set a threshold value for the load on each processor at the same time.

A very simple method is used to calculate the value for the load threshold used in the precedence constrained scheduling heuristic.

LT = Load Threshold.

m = Total number of tasks.

n = Total number of processors.

l = Number of levels in the precedence graph.

X_i = Execution time for task i.

$$LT = (\sum_{i=1}^{m} X_i/m)t + max(X_i)$$

Where t is a parameter that depends on the average number of tasks available at each level of the precedence graph. It is computed in the following way.

$$t = \lfloor (m/l)/n \rfloor$$

Next, some important steps of the precedence constrained scheduling heuristic are described.

- Input a precedence graph with n tasks and their inter-communication times. Also, input the parameters related to the communication network.
- All the tasks without any precedent tasks are kept in the ready list. $R = \{T_i \mid$ all the tasks preceding i have already been allocated$\}$
- If ready list is not empty repeat the following.

- Dequeue a task T at the head of the ready list.
- Find a processor $P = \{P \mid$ task i with $max(co - level(i) + com(i,T))$ is scheduled on P$\}$, where $i \in S$, and S is a set of parents of T.
- If $RPT_p < LT$
  Allocate task T to processor P.
- Else delete P from the set of parents and repeat the last three steps.
- After the communication time between the task T and any of its children has elapsed, remove the edge from the task T to that child.
- Remove task T from the ready list.
- Check for any other ready tasks and update the ready list. Repeat.

- Output the Gantt chart and other performance statistics.

5.4 Experimental Results

The experiments were carried out by using the static simulation model described in chapter 4. The performance criterion used was the minimization of the overall completion time and the results were displayed using the Gantt charts. Other statistical results were also presented to support the results. From these experiments, some intuitive but important facts were verified. The effects of scheduling a task on the processor where a preceding task with largest communication cost is scheduled were studied. Scheduling a task on such a processor where one of its precedent tasks is scheduled improves the overall completion time of the schedule. This results from the fact that the cost of inter-task communication for the two tasks is eliminated in

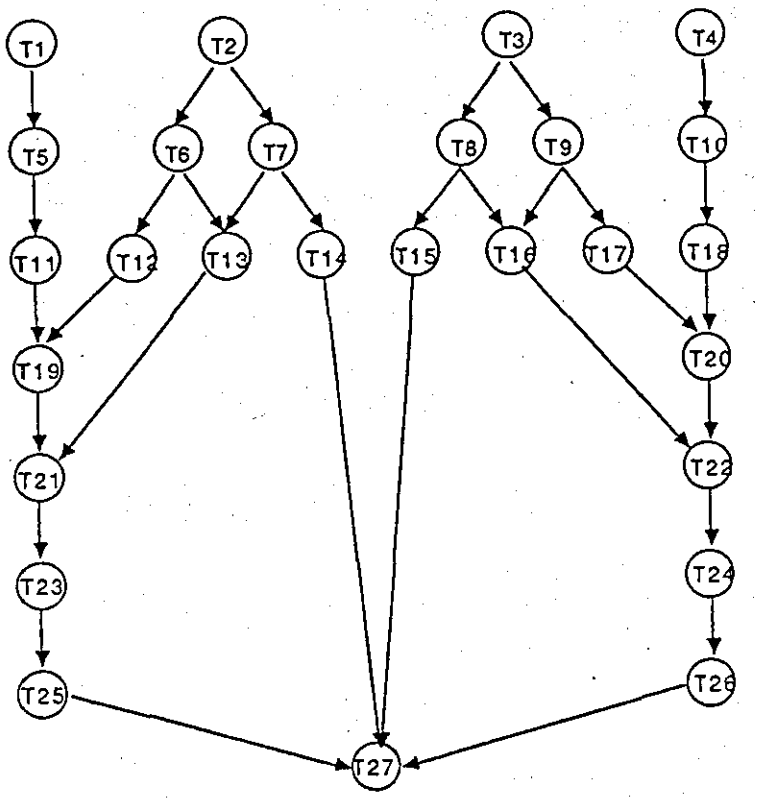

Figure 5.1: Task Graph used for Scheduling (Gantt Charts; Figures 5.2-5.4)

this processor. To maximize this effect, a task from a set of preceding tasks was chosen which corresponded to the longest path leading to this task. The degree of improvement in the overall completion time of the schedule using this type of heuristic entirely depends on the computation to communication ratio of the task system. Considerable improvements in the schedule were achieved for the task sytems with lower computation to communication ratios.

5.4.1 Example 1

The task graph (Fig 5.1) used in the following experiment is one of the several systematically generated test data sets described in [Price and Salama 1990]. These data sets have been described to expose the strengths and

weaknesses of different scheduling heuristics. It consists of 27 tasks with execution times ranging from 5 to 50 time units. Execution times for all the tasks and the inter-task communication times used in these experiments are given in Appendix A.

Figures 5.2-5.4 below show the Gantt charts for the schedules produced by the simulator. Figure 5.2 is produced under the conditions where there is uniform load balancing. Task arrival is uniformly distributed over a number of processors. Since the number of tasks is uniformly distributed over a number of processors, and as there are no subsequent task transfers, the load allocated to each processor is nearly the same.

Figure 5.3 is produced under the conditions of simple load balancing. In this case, each new task is assigned to an idle processor if available, or to a processor with minimum queue length. In cases where more than one idle or minimum queue length processors exist, a processor that is nearest (minimum hops away) to the processor, where the preceding task was allocated, is selected.

Figure 5.4 is produced under the conditions where a new task is scheduled on the processor with the longest path preceding task. If the processor with the longest path preceding task is already overloaded then a processor with the next longest path preceding task is selected.

Several interesting results were revealed from these schedules. In Figure 5.2, with uniform load balancing, nearly equal number of tasks are allocated to each processor. One might conceive that this heuristic will give good performance due to uniform allocation of tasks. However, such a heuristic

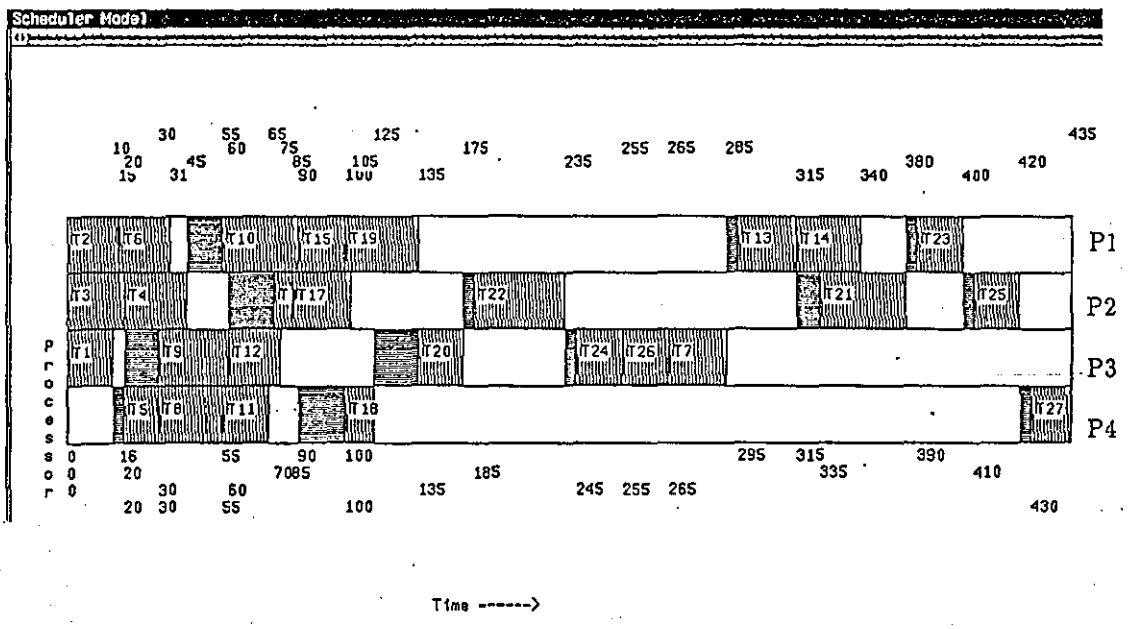

Figure 5.2: Gantt Chart with Uniform Load Balancing (Experiment 1)

can only perform well when task execution times are identical and inter-task dependencies are not considered. For applications with widely varying task execution and inter-task communication times used, it results in a poor performance as no intelligence is used to utilize the idle or lightly loaded processors. In this schedule, several tasks are queued on the already busy processors, where as some other processors are idle at that time. It can be seen from task graph (Fig.5.1) that tasks T1 to T4 are ready for simultaneous execution. Tasks T1 to T3 are executed simultaneously on three different processors. Task T4, initiated on processor P2, awaits tasks T3 on processor P2 to finish before it can start execution. Since the current status of processors is not used when allocating tasks , Task T4 is not executed on processor P4 which is idle at that time. The schedule produced, without load

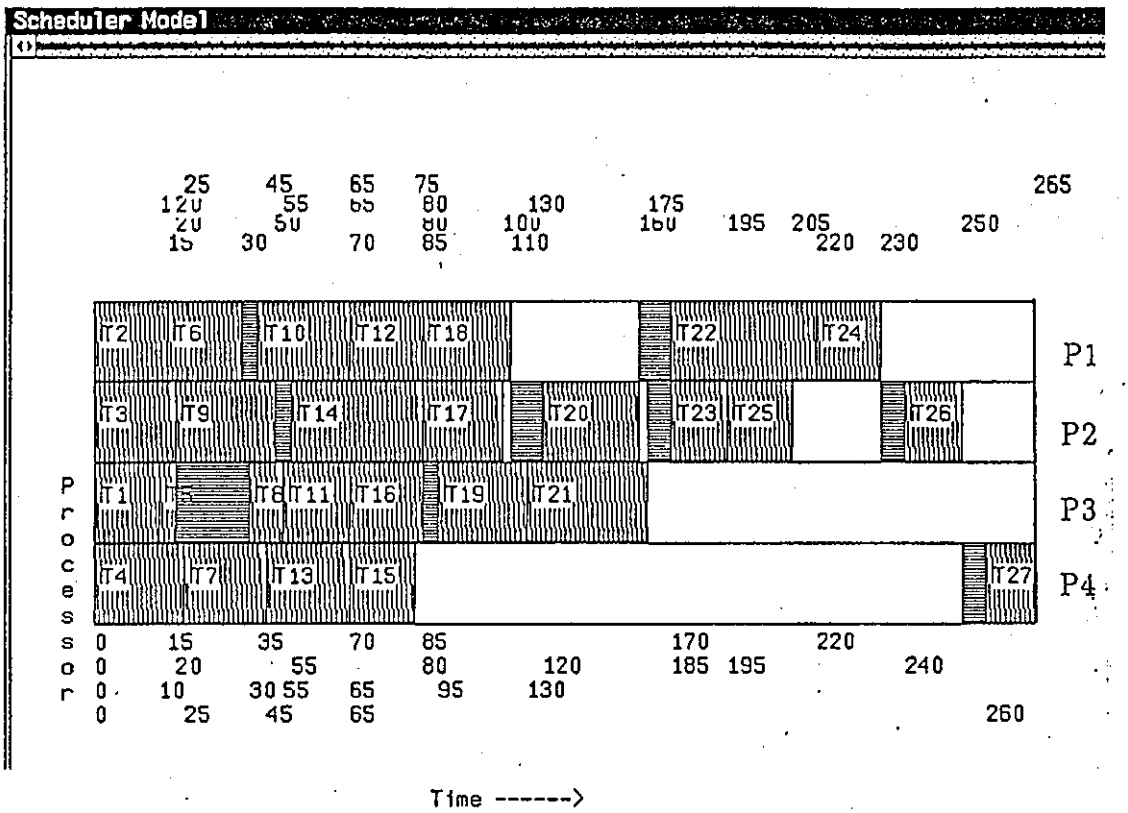

Figure 5.3: Gantt Chart with Simple Load Balancing (Experiment 1)

balancing, has a large completion time of 435 time units. This large completion time is mainly due to the lack of co-ordination among the processors to share each others load. As shown in Fig.5.2, each processor is idle or busy communicating with some other processor for a large percentage of the total completion time.

In the next schedule shown in Fig.5.3, an attempt is made to extract maximum parallelism by trying to distribute the tasks evenly among the available number of processors without violating any precedence constraints. In contrast to uniform load balancing where each ready task was allocated to a randomly chosen processor, in this heuristic each new task is scheduled for execution on the most lightly loaded processor available. This technique

shows considerable improvement in the overall completion time of the schedule. This type of schedule takes full advantage of the parallelism inherent in the task graph. This effort to load balancing by distributing the load over all the processors conflicts with the idea of the minimization of the inter-task communication times. The completion time of the schedule in this case is 265 time units, which is an improvement of 39 percent over the schedule produced with uniform load balancing. Also, as shown in Fig.5.3, there are fewer idle times which are caused by the precedence constraints amongst the tasks.

Finally, the Gantt Chart of Fig 5.4 shows further improvement in the overall completion time of the schedule. In this case, the placement decision of a newly arrived task is based on the location of the precedent task with the longest delay path. Here, it is noticed that the number of tasks in the system is not uniformly distributed over the number of processors. It is also noticed that several tasks are waiting in the queues of some processors while other processors are idle. Both these situations indicate the load imbalance in the system. Looking at the task graph (Fig.5.1) and the Gantt Chart (Fig.5.4) simultaneously, it is easy to find out the cause of this load imbalance. Task T1 is scheduled on processor P4 for execution. Task T5 is dependent on task T1 and is waiting for T1 to finish before it can start execution. Task T5 is scheduled to run on the same processor as task T1 to cut down the communication cost between the two tasks. Similar is the case with task T11 waiting for task T5 to finish. Next, looking at the task graph, it was found that task T19 is dependent on T11 and T12. After task T11 has finished, task T19 is waiting for T12 to finish before it starts execution. Task T12 is scheduled to execute at time 95 on processor P2. If task T19 is scheduled to

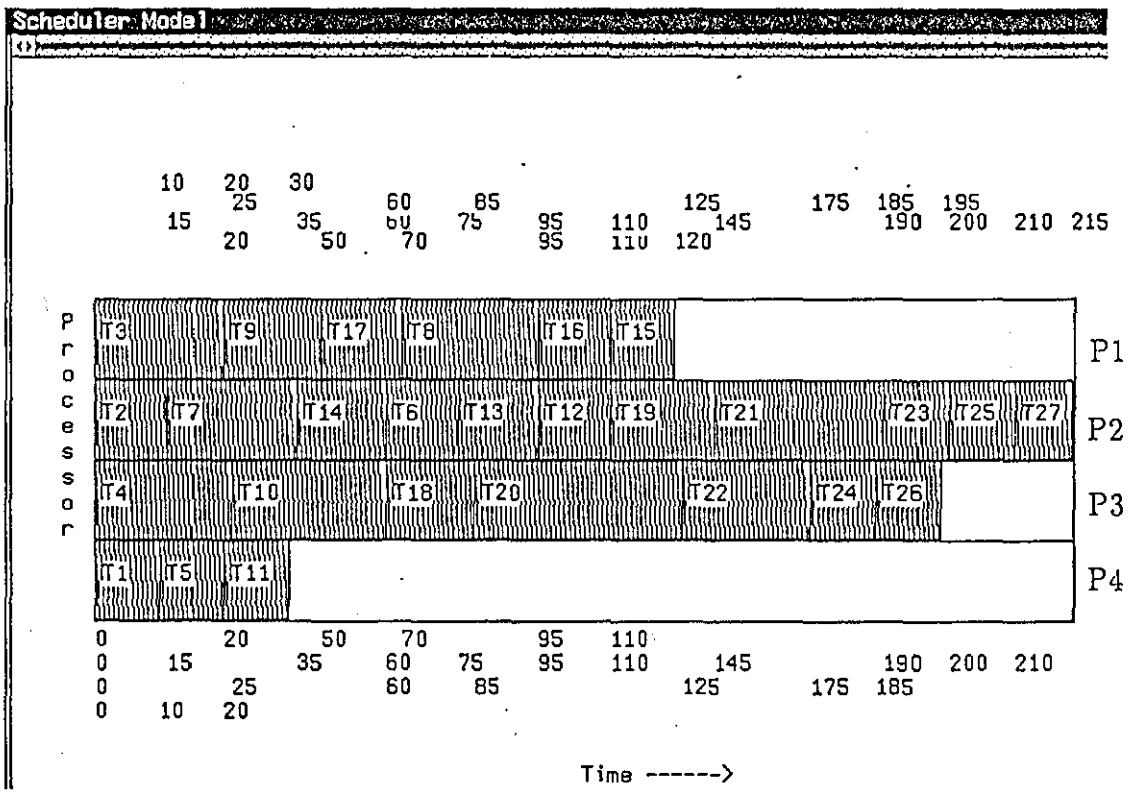

Figure 5.4: Gantt Chart with Precedence Constrained Load Balancing (Experiment 1)

execute on processor P4 instead of processor P2, it will have to experience further delay due to communications between T12 and T19 before it can start execution. This attempt to minimize the communication cost conflicts with the uniform loading of the processors, and leaves processor P4 lightly loaded. This schedule has a completion time of 215 time units, which gives a further improvement of nearly 20 percent over the schedule with simple load balancing.

Some of the other performance measures for each schedule are given in Appendix B. In the worst case, with uniform load balancing, average processor utilization is approximately 40 percent. With simple load balancing, average processor utilization increases to 55 percent. Precedence constrained scheduling introduces a further increase to 65 percent in the average utilization. The maximum average processor utilization, that can be achieved, depends on the communication-computation ratio of the task system.

5.4.2 Example 2

The task graph (Fig.5.5) used in this example is an extension of the graph used in [Rewini and Lewis 1990]. There is a large variance among the task sizes and the communication delays are higher. There are 30 tasks with execution times ranging from 5-50 time units. These tasks are scheduled on 4 processors inter-connected in the form of a ring topology.

The task system used in this example has inherently more parallelism than the task system used in the previous example. All the tasks at each stage are enabled for execution at the same time. The behaviour of this task system is similiar to the behaviour of barrier synchronization problems, in which several tasks execute concurrently and then wait at a common point

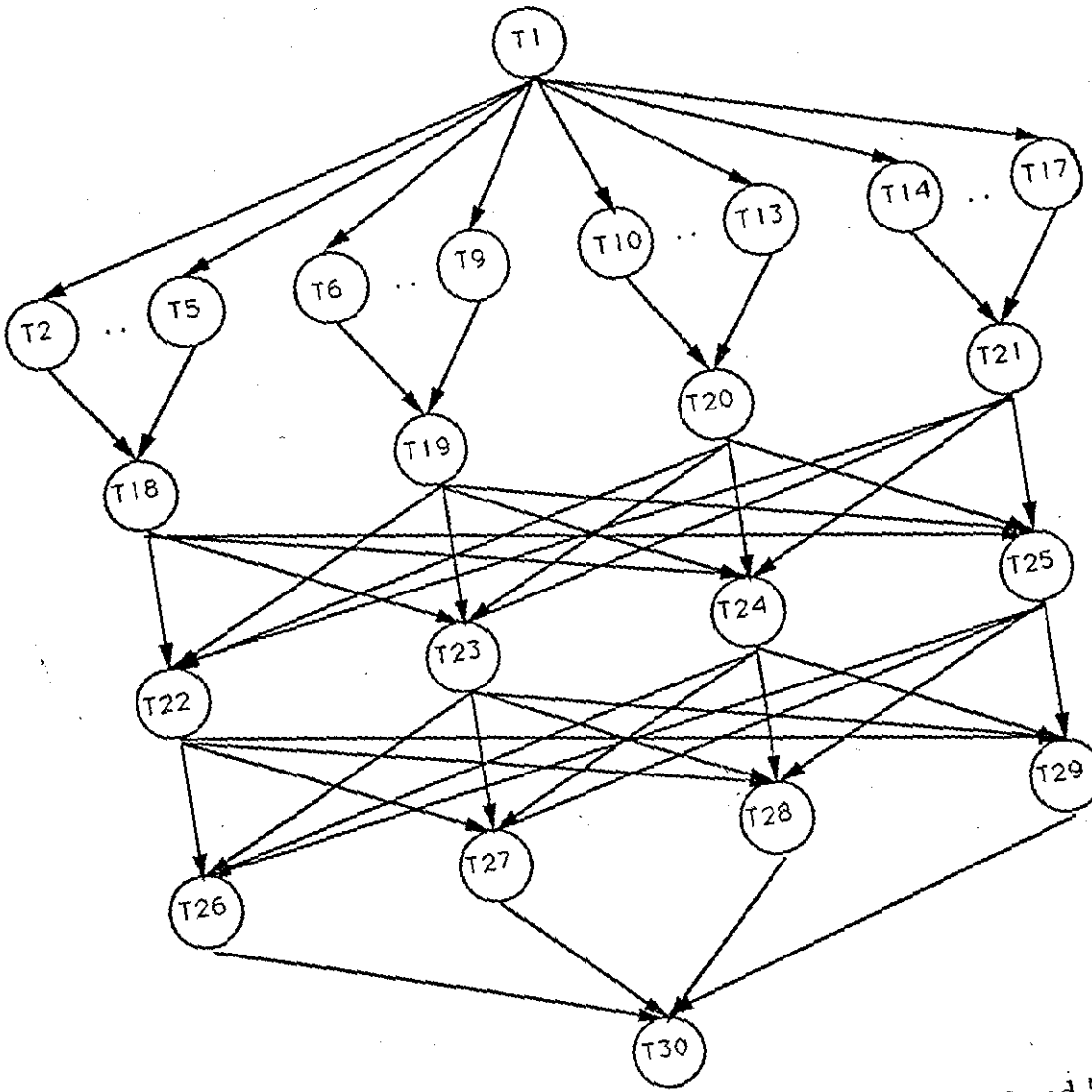

Figure 5.5: Task Graph used for Scheduling (Figures 5.6 and 5.7)

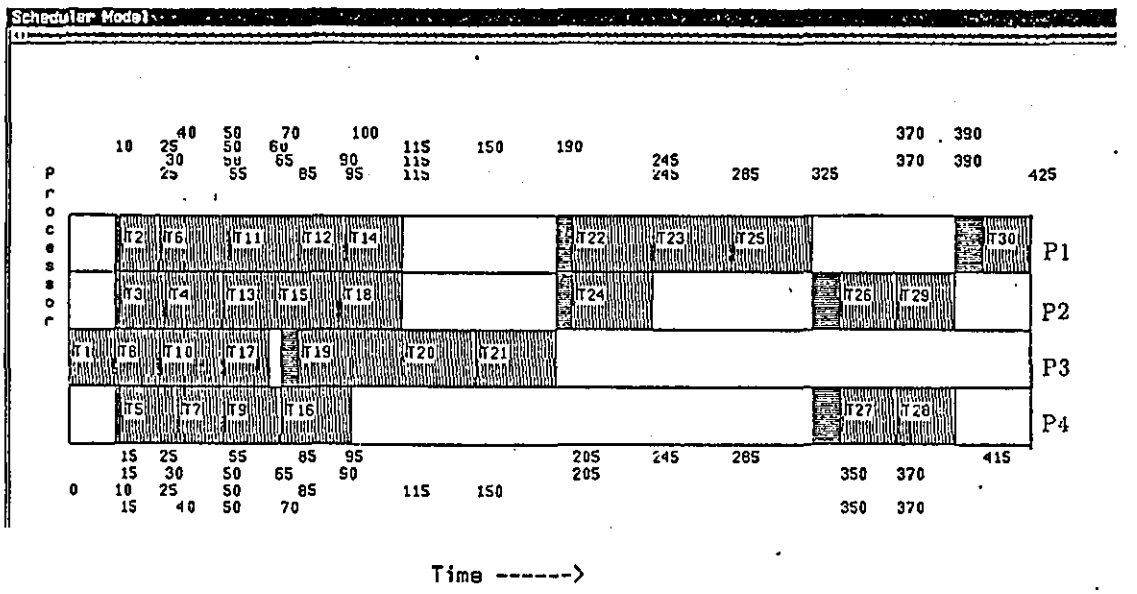

Figure 5.6: Gantt Chart with Uniform Load Balancing (Experiment 2)

(called barrier) for synchronization before a new set of tasks starts execution.

Schedules with uniform load balancing, with simple load balancing and with precedence constrained load balancing were produced using this task graph. As compared to the task graph in the previous example, this task graph has more tasks and the average size of the tasks is also larger. Completion time of the schedule with uniform load balancing (Fig. 5.6) is 425 time units, this is nearly the same as for the previous example. Lesser completion times indicate more parallelism among the set of tasks in this example. In the case of uniform load balancing, this parallelism is utilized by the initial uniform distribution of tasks among the processors. This can also be confirmed by comparing the utilization figures for the two schedules provided in Appendix B.

Gantt Charts produced for schedules with simple load balancing and with precedence constrained load balancing had slightly different schedules with the same completion time. Only the Gantt Chart for the latter is shown in Fig.5.7. Completion time of these schedules is 280 time units, which is an improvement of nearly 36 percent over the schedule with uniform load balancing. The reasons for this improvement are very intuitive and similiar to the ones described in the previous example.

Equal completion times for the two schedules is an interesting result, which indicates that scheduling a new task on a processor with the largest delay precedent task does not further improve the schedule. Now, looking carefully at Fig.5.5 and Fig.5.7, it was found that each new task (Tasks T22 to T30) depends on a set of equal delay precedent tasks. For example, Task T26 depends on all the four tasks (T22-T25). In this case, scheduling T26 on a processor with any one of these four tasks does not minimize the communication costs, since T26 cannot start until each of these four tasks is finished and communications have elapsed. This suggests that precedence constrained load balancing does not perform any better than simple load balancing for this particular type of applications (represented by a task graph with constant delay paths at each level and high computation to communication ratio). If scheduling of all preceding tasks of a task on the same processor is considered, it might be useful for only applications having a very low computation to communication ratios. Though it causes a considerably large imbalance in the processor loads, it might still produce a schedule with improved completion time. If such a scheduling policy is applied on this example, tasks T2 to T4 and tasks T18 to T30 will be scheduled on the same processor. Therefore, half of the total load will be assigned to a single processor.

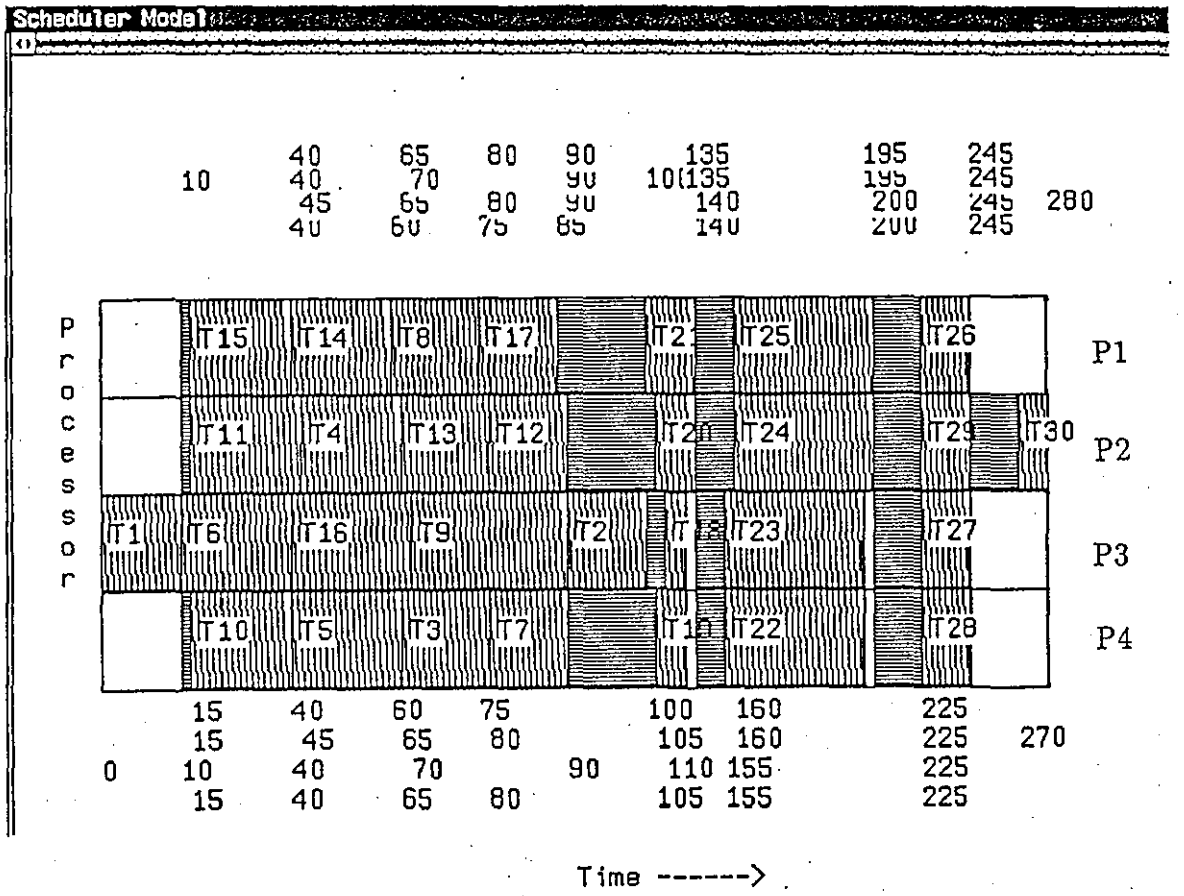

Figure 5.7: Gantt Chart with Precedence Constrained Load Balancing (Experiment 2)

Now, looking at the two schedules carefully (Fig.5.6 and Fig.5.7), it was noticed that there was not much difference in communication costs for the two schedules. All the performance gain came from the *simple load balancing.*

5.5 Conclusions and Future Directions

The assumption of a precedence graph with known execution times and interprocessor communication costs does not capture the dynamic behaviour of the real task systems. However, it simplifies the scheduling problem and helps to concentrate on other critical issues common to both the static and the dynamic schedulers. In this study, I have tried to experiment with two central issues in scheduling i.e; minimization of interprocessor communication costs and the load imbalance. It was shown that for an efficient solution to the scheduling problem, it is very important to have a compromise between maximizing the parallelism and minimizing the communications. It is suggested that an appropriate choice of scheduling heuristics that minimize the communication costs and set an upper threshold for the load on each processor to avoid the load imbalance can provide improved performances. Such a heuristic works well for a wide range of computation to communication ratios and, therefore, can be used with a large number of applications. It might also be interesting to investigate the usefulness of load threshold (LT) formula in a dynamic scheduling environment. This can be done by estimating the mean service time of the system over a certain period that corresponds to $\sum X_i/m$ in our case. Similarly 't' can be approximated by estimating the average number of task arrivals in the system during an interval equal to the mean service time. In future, it will be interesting to study the effect of task ordering with precedence constrained scheduling on

the completion time of the schedule. In this study, it was assumed that the task size is known. In practice, it is very difficult to analyze the behaviour of a task and to provide good estimates on a tasks' execution time.

Chapter 6

DYNAMIC LOAD BALANCING USING TASK-TRANSFER PROBABILITIES

6.1 Introduction

In the preceding chapter, the work performed concentrated on deterministic scheduling techniques which made use of list scheduling to achieve load balancing. It was assumed that all the parameters related to the task system and the communication network were known at the time of making load balancing decisions. In this chapter, the research undertaken is focussed on job[1] allocation strategies that perform dynamic load balancing in a distributed

[1]A job may consist of more than one inter-communicating tasks. In this discussion, the word task is used instead of job

computer system. All the dynamic load balancing solutions are suboptimal and heuristic. A load balancing algorithm should be distributed, efficient and stable. It is assumed that the information about the task behaviour and the network parameters are not known in advance. The dynamic algorithms described, try to obtain load estimates on the frequently changing load conditions on the hosts in a distributed computer system. These load estimates, obtained by monitoring the system for a specified interval, are utilized in making efficient load balancing decisions dynamically.

In the past, several studies have been carried out on dynamic load balancing strategies with an aim to improve the performance of the system. A review of such strategies is described in chapter 3. In the majority of dynamic load balancing algorithms an information update policy is used to exchange the load information amongst the hosts in the distributed system. A load threshold is determined using the available load information. Each host in the system compares its load with a certain load threshold to make a decision for the transfer of tasks.

In section 3 of this chapter, two simple dynamic load balancing algorithms are described to demonstrate the effectiveness of such algorithms to enhance the performance of distributed systems. In most of the cases, queue-length is chosen as the load index and the load threshold is based on the average value of the load.

In section 4, an attempt is made to explore the factors that contribute to the load imbalance in a distributed computer system. The effects of task arrival distribution and service time distribution on the performance of a

distributed computer system were studied. The arrival and service time estimates were obtained and the load balancing was performed using these estimates. It was assumed that each host broadcasts its estimated load information to every other host in a homogeneous distributed computer system. Each host computes the probabilities of transferring a task to every other host in the system. These probabilities were changed dynamically over a fixed interval of time. Average response time was used as the performance metric for comparing different cases. Root mean square difference in host loads was measured to compare the performance of load balancing in different cases.

[Gao et. al. 1984] have described two static load balancing algorithms. In these algorithms, necessary parameters required by each algorithm are exchanged amongst all the hosts periodically for the purpose of load balancing. The two algorithms perform load balancing by balancing average arrival rates and the amount of unfinished work on each host respectively. In these algorithms, for given communication cost between any two hosts and the number of job transfers between any two hosts, the cost function is minimised using linear programming techniques. It was observed that any such solution that involves linear programming is very expensive for a load balancing algorithm. A load balancing algorithm has to be very fast and efficient to achieve high performance in a distributed computer system. The algorithms suggested in this paper are simple and fast and provide high performances.

6.2 Model And Assumptions

To study the performance of non-deterministic, dynamic load balancing algorithms, it is very important to formulate the dynamic scheduling problem

in terms of a probability model. Probability models are more realistic than deterministic models as the former can capture the time varying characteristics of a distributed scheduling problem. When these probability models are used to study the properties of dynamic scheduling techniques they take the form of queueing systems [Coffman and Denning 1973]. The dynamic scheduling problem is very complex in nature and requires a large number of parameters to be considered for high performance gain. At the same time, when there is a large number of inter-related parameters under consideration, it becomes very important to make simplifying assumptions for a mathematically tractable queueing model.

Queueing systems are a powerful tool for performance analysis and prediction and have been extensively used for modelling computer systems. In agreement with many researchers, the work was performed using queueing models. The models, though simple when compared with those proposed by other researchers, were very useful and served the required purpose.

In the formulation of the queuing model, each host in the distributed system is represented by an M/M/1 model. A single server queue with Poisson arrivals and exponentially (negative) distributed service times is considered.

where,

$$A(x) = 1 - e^{-\lambda x} \quad for \ x \geq 0$$

and

$$S(x) = 1 - e^{-\mu x} \quad for \ x \geq 0$$

where $A(x)$ and $S(x)$ are arrival and service distributions respectively.

$$Arrival\ Time = \frac{1}{\lambda}$$

and

$$Service\ Time = \frac{1}{\mu}$$

The work, presented in this chapter, was performed using an infinite queueing model. There was no limit on the queue length and the task scheduling was based on First in First out (FIFO) mechanism.

The model used can be best described by Fig. 6.1. This figure presents four hosts in a distributed computer system inter-connected by an arbitrary communication network. This simplified host model has four major components of our interest. These include a load balancing component, a queue for waiting tasks, a server as the processing unit, and a communication network for task transferrence between any two hosts.

$\lambda_i(t)$ = Task arrival rate at host i, this information is updated in each host after a broadcast interval of Δt.

$\mu_i(t)$ = Task service rate at host i, updated in each network host i after an interval Δt.

Δt = Time interval between two successive information update messages.

$\lambda_i'(t)$ = Task arrival rate at host i, after the load balancing has been performed

The load balancing component 'lb', also referred to as the job dispatcher, decides whether a newly arrived job will be executed locally or remotely. In our experiments, no cost is associated with the load balancing component.

P_{ij} = Probability of task transferrence from host i to host j

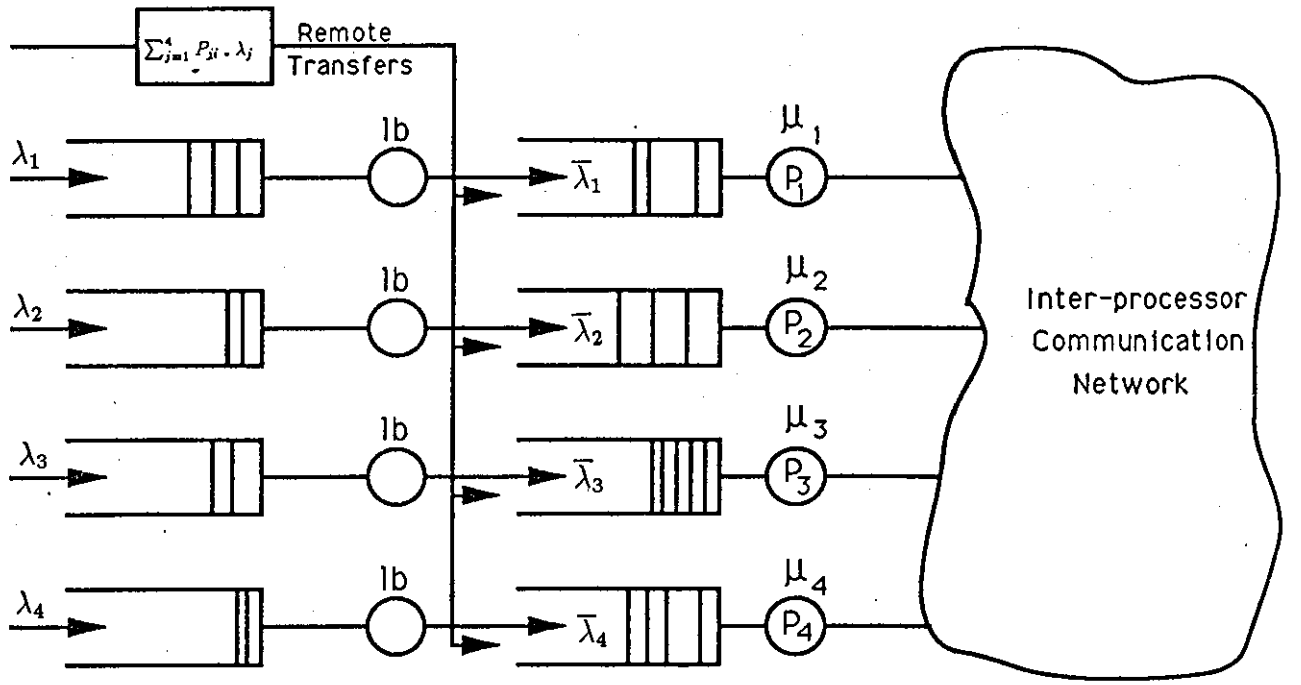

Figure 6.1: A queueing network model for dynamic load balancing

$$\lambda_i'(t) = \lambda_i(t) - \sum_{j=1}^{n} P_{ij}\lambda_i(t) + \sum_{j=1}^{n} P_{ji}\lambda_j(t)$$

In the above equation, $\sum P_{ij}\lambda_i$ are the tasks outgoing from the host h_i, whereas $\sum P_{ji}\lambda_j$ are the incoming tasks to the host h_i. A large difference between these two components will ensure that there is no load thrashing amongst the hosts. Similarly, the task transferrence rate X_{ij}, can be formulated as,

$$X_{ij}(t) = P_{ji}\lambda_j(t) - P_{ij}\lambda_i(t)$$

If X_{ij} is positive then tasks are transferred from host i to host j, if X_{ij} is negative then the tasks are transferred in the opposite direction.

The work performed assumes that the network is comprised of hosts with identical processing capabilities. For such a homogeneous distributed system, varying service times indicate that the size of tasks entering the hosts are different on separate hosts. Furthermore, different task arrival rates are used for different hosts. These varying arrival and service rates enable us to introduce the imbalance among the hosts which is a characteristic of any true distributed system.

To simplify this model, it was assumed that the communication costs between any two hosts are the same. Considering such an assumption reflects these communication costs in such a way that it does not effect the load balancing results.

For load balancing purposes, it is very important that each host is kept informed of the current state of the system. For current studies, system state depends on the load condition of each host in the system. At any time, each host in the system stores a vector of length equal to the total number of hosts in the system. An element of this vector keeps the current load status on a host in the system.

To keep this information up-to-date, different protocols are designed which exchange this information amongst the hosts in a distributed system. An underlying protocol that distributes each host load among all other hosts in the system periodically after each time interval of Δt is assumed. The length of the time interval (Δt) can be adjusted such that the requirements of the load balancing algorithm are met effectively as well as commmunication overhead due to information exchange is minimised.

$$load_i(t) = \{q_1(t), q_2(t), q_3(t) \ldots q_n(t)\}$$

$load_i(t)$ is the load vector stored by host i at time instant t. Whenever an information update message from a host j is received at host i, $q_j(t)$ in load vector $load_i(t)$ is updated.

To measure the effectivenesss of load balancing algorithms, two performance metrics were used. These are load imbalance [Gao et. al. 1984] and average response time. Load imbalance is defined as the root mean square difference among the host load.

$$load\ imbalance(t) = \{\sum_{i \neq j} [load_i(t) - load_j(t)]^2 / [n(n-1)/2]\}^{1/2}$$

Where average response time is the average time spent by each job in the system. Response time of a single job is the time elapsed between its submission to its completion.

A simulation model was designed for the dynamic load balancing environment. For this study, the task inter-arrival times and the task service times were generated from a negative exponential distribution. A task despatcher was simulated to allocate tasks to the processors in First-Come-First-Serve (FCFS) order. All the results were recorded once the initial transient period had elapsed and the simulation system was running in a steady state.

6.3 Effectiveness Of Some Simple Dynamic Load Balancing Strategies

In a distributed computer system, it is quite common that some hosts are heavily loaded while others are lightly loaded at a particular instant of time. This results in a very poor performance of the system. Dynamic algorithms react to the changes in the system state dynamically. In this section, two simple dynamic load balancing strategies are proposed and their effectiveness is evaluated through simulation studies. The simulation model and the performance metrics used for this study are described in the previous section.

6.3.1 Random Policy

It is a very simple non-deterministic policy that uses current state of the system to dynamically transfer the jobs amongst hosts for load balancing. The current state of the system comprises of the load status of each host. The information regarding the load of each host is distributed amongst all the hosts in the system. In a distributed system, all activities concerning the collection of status information and the decision to balance the load are performed by all hosts. The load of a host is determined by the total number of jobs present in the hosts' queue. Each host maintains an information table that contains the load of each host in the system. For the host 'i', the load at time t is denoted by $q_i(t)$.

Each host in the system uses the load information table to estimate the average load of the system. The average load calculated in this manner is purely an estimate as it may not reflect the actual average load at that in-

stant of time. For shorter information update intervals, these estimates will provide a better reflection of the actual load. However, running the information update policy frequently, incurs a large overhead on the communication subnet resulting in the performance degradation of the distributed system.

The random policy uses the estimated average load to transfer tasks from the heavily loaded hosts to one of the other hosts in the system chosen randomly. Another parameter used is called the 'bias', it offsets the effects of communication delays in transferring the jobs between two hosts.

This algorithm repeats after each update interval. The algorithm is distributed and is executed by each host independently. Important steps for the execution of this algorithm are described below.

ALGORITHM : /* random */

```
for i=1 to no. of hosts {
    if (q_i(t) ≥ (q̄_i(t) + bias) {
        while(dest=i) {
            dest=random(1,n)
        }
        transfer[i][dest] = q_i(t) − (q̄_i(t) + bias)
    }
}
```

In this algorithm $\overline{q}_i(t) + bias$ determines the load threshold for each host. Any host that has jobs in excess of the load threshold is considered to be heavily loaded. Any jobs over the load threshold are transferred to a randomly chosen host.

This algorithm is not purely dynamic. Load balancing is performed on the basis of load information available from the previous interval and a fixed number of tasks are transferred in the subsequent interval. A purely dynamic algorithm reacts to any changes in the system load immediately. Such algorithms are described in the next section.

6.3.2 Threshold-Window Policy

This algorithm transfers tasks from the heavily loaded hosts to the lightly loaded hosts dynamically. A host is categorized as lightly or heavily loaded on the basis of a defined threshold-window. All the hosts lying outside the upper edge of the window are defined as heavily loaded, whereas the ones lying below the lower edge of the window are defined as lightly loaded. A threshold-window is defined by setting an upper load threshold (UT) and a lower load threshold (LT). All the hosts lying within the window are neutral and do not want to participate in load balancing.

In this algorithm, each host monitors its load continuously. If a hosts' current load falls below the lower threshold, it sets a RECV flag to indicate its acceptance for receiving any transferred tasks from some other host. The flag condition is distributed to all the hosts in the system.

On the other end, if a hosts' current load is greater than the upper threshold then this host transfers one task to each of the hosts with a RECV flag. After the task is transferred, this flag is reset. The lower threshold is adjusted such that if a majority of hosts in the system is heavily loaded, transferring one task from every such host to a lightly loaded host will not exceed the latters' load above the upper threshold.

This algorithm does not require periodic broadcasts for distributing load information. However, a message is broadcast to all other hosts whenever a host sets its RECV flag. It indicates that if the threshold-window is adjusted carefully, this algorithm will provide improved performances with low communication overhead. It is also expected that for light system loads such algorithms will perform better than the algorithms using frequent periodic load updates.

An improvement in this algorithm was made by introducing task-transfers to randomly chosen hosts if no lightly loaded host was available. In this case, better performances can only be achieved if a good estimate of average load on the system is available. Important steps for the execution of this algorithm are given below.

ALGORITHM : /* threshold-window */

```
for i = 1 to no. of hosts {
    if(q_i(t) < LT) {
        RECV[i]= 1;
        brdcst(RECV[i]);
```

```
    }
}
for i = 1 to no. of hosts {
    if(q_i(t) > UT) {
        for j = 1 to no. of hosts {
            if(RECV[j]==1) {
                transfer[i][j]++;
                RECV[j]=0;
            }
        }
    }
}
```

The information exchange mechanism used in threshold-window algorithm requires a lightly loaded host to transmit its flag condition to all the hosts whenever its load falls below the lower threshold and the flag is set to '1'. The minimum interval between two broadcast messages is set equal to the maximum possible task-transfer delay between any two hosts. A transmission delay parameter was used in our simulations to simulate the average delay in getting this information through to all the hosts in the system. On the other hand, if a heavily loaded host finds its load greater than the upper threshold, it transfers a task to each host with a RECV flag set and resets the flag to '0'. Since several tasks can be transferred from heavily loaded hosts to a lightly loaded host, therefore the frequency of RECV message broadcast is important to the performance of this algorithm.

Host	1	2	3	4
Inter-Arrival Time	11	15	20	12
Service Time	10	8	12	10

Table 6.1: Queueing Parameters Used For Simulations

6.3.3 Performance Evaluation By Simulation

The performance of the two algorithms was evaluated using the simulation model described in chapter 4. These algorithms were proposed to study the effectiveness of simple load balancing policies. The mean values used to generate the arrival and service distributions are described in Table 6.1. These parameter values are also used in [Gao et. al. 1984], and were used here to provide a qualitative comparison with their results.

To maintain the simplicity of these algorithms, commonly used heuristics were incorporated. For Random policy, it was assumed that the load information is broadcast periodically and load balancing decisions were made by comparing each hosts' load with an estimate of the average load. Hence, this algorithm uses the global load information for making task-transfer decisions. [Zhou 1988] has compared the performance of a random algorithm with several other simple algorithms using trace-driven simulations. In his studies, the Random algorithm makes decisions on the basis of local load information. The authors' results have shown that the performance of the random algorithm are comparable to several other algorithms.

The Random algorithm described here was expected to perform better than the random algorithm which used local load information. Performance metrics used for the evaluation are described in section 6.2. Fig. 6.2 shows the response time curve for different lengths of the information update interval. Each point of the curve is obtained from an average of several samples. The response time curve indicates the degradation of performance by increasing the interval length. For interval lengths from 200 to 1200, the gradual increase in response times indicates the importance of frequent load information exchange. This curve nearly flattens for the interval lengths greater than 1200 indicating that the amount of load balancing performed is not sufficient to balance the load across the system. For interval lengths greater than 1200, the estimates obtained for the average load may not reflect the load variations during longer intervals thus leading to inefficient load balancing decisions. Furthermore, this algorithm is not purely dynamic and a fixed number of tasks are transferred by heavily loaded hosts after each interval. The number of tasks transferred is equal to the number of tasks in excess of the load threshold. This algorithm is executed at the time of every new load update, therefore, fewer tasks are transferred for longer update intervals in comparison to the shorter ones.

A dynamic load balancing policy cannot achieve higher performances by simply balancing the load amongst all the hosts in the distributed system while neglecting the communications related issues. In a distributed system, it is very important to consider the impact of a load balancing policy on the communication subnet. In estimating the response times, the communication delays offered in transferring the tasks were not considered. However, a parameter 'bias' was used to define the load threshold. The value of bias

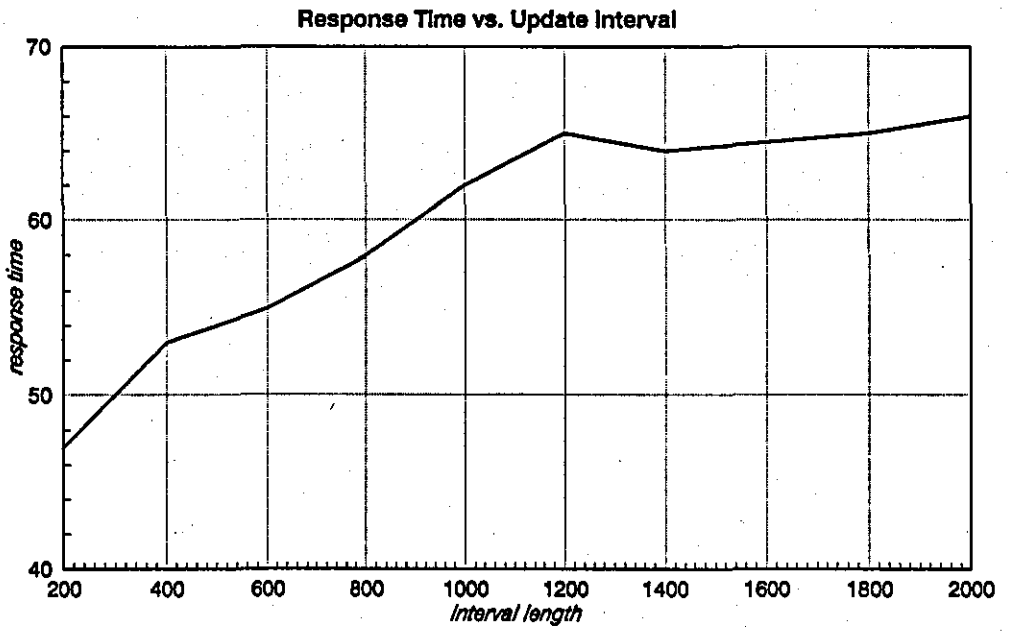

Fig.6.2: Average Task Response Times for Random Policy.

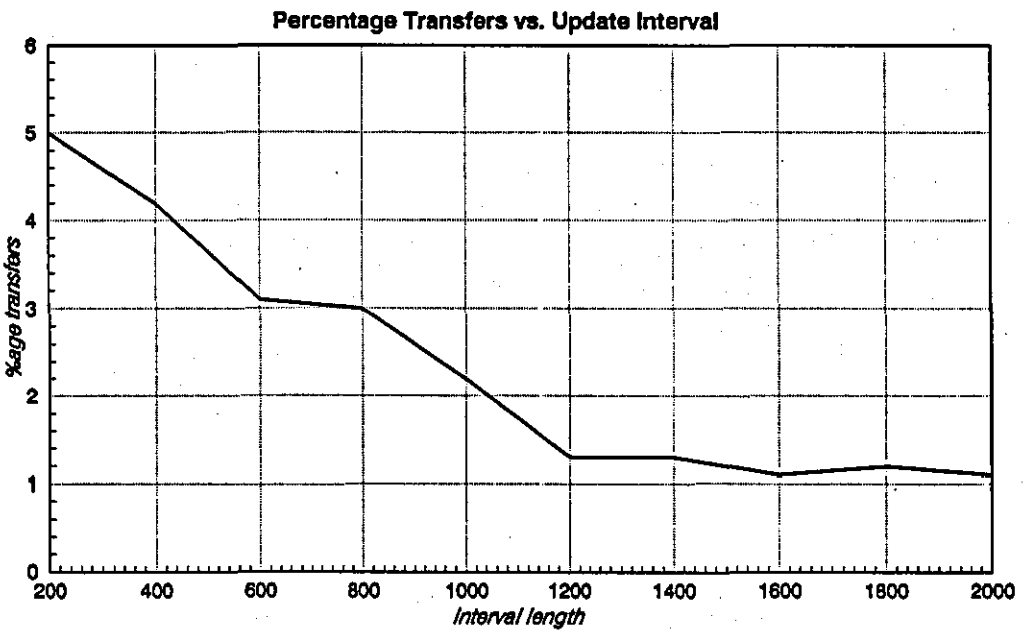

Fig.6.3: Percentage Task-Transfers for Random Policy.

should be adjusted such that it is greater than the average cost in transferring a task between any two hosts in the system. In this study, bias is assigned a fixed value of '2'. If a hosts' load is greater than the average value by an amount less than or equal to bias, better response times are expected in executing these tasks locally than remotely. However, introducing the bias in defining the load threshold does not solve the communications problem, as the frequency of load update messages and a large number of task transfers may cause unexpected queueing delays in the network and create bottlenecks. To investigate the impact of this algorithm on the communications network, the amount of task transfers for different update interval lengths was studied.

Fig. 6.3 shows the percentage of task transfers for different load update intervals. This curve shows that there is a considerable reduction in percentage of task transfers up to an interval length of 1200 and remains constant thereafter. Due to the random selection of a destination host for transferring a task from a heavily loaded processor, it was expected that this algorithm might cause processor thrashing by choosing another heavily loaded host. It was noticed that the amount of processor thrashing was very small and a large number of transfers were useful for balancing the load. This was probably due to the small number of simulated hosts and the choice of arrival and service parameters. In a small system with unbalanced load, there is a large probability that given a heavily loaded host, any other randomly chosen host will be a lightly loaded one. Also, as mentioned above, the Random algorithm described here transfers a limited number of tasks during each interval and these tasks are transferred on the basis of a global load information available at each host. If the distributed system is very large and the overall load on the system is moderate to high, this algorithm is susceptible to processor

thrashing.

Now looking at the two figures (6.2 and 6.3) simultaneously, the conflicting issue of reducing the communication costs and achieving the maximum parallelism is observed. On one end, higher response times were achieved at the cost of frequent broadcasts and high transfers, whereas, at the other end, communications overhead was reduced for relatively poor response times.

Next, the performance of threshold-window algorithm was evaluated and compared with the Random algorithm. The major difference between the two policies is that of the information update mechanism. The threshold-window policy does not use periodic broadcasts and instead relies on asynchronous messages for transmitting required load information. It is a server-initiated algorithm.

The two main factors that influence the performance of the threshold-window algorithm are the window size and the message transmission delay. The performance of this algorithm was studied for a large number of different window sizes. Fig. 6.4 presents the load imbalance amongst the hosts of the distributed system for different window sizes. As expected, any changes in the size or the selection of upper and lower thresholds had a considerable effect on the load imbalance and average response times. Amongst a limited number of experiments on window sizes and window ranges, it was observed that the best performances were achieved for a window size of 5 with the thresholds of $LT = 5$ and $UT = 10$. It was also observed that there are relatively fewer task transfers for this window size as compared to the other two. This can be explained as a result of choosing a small value for the

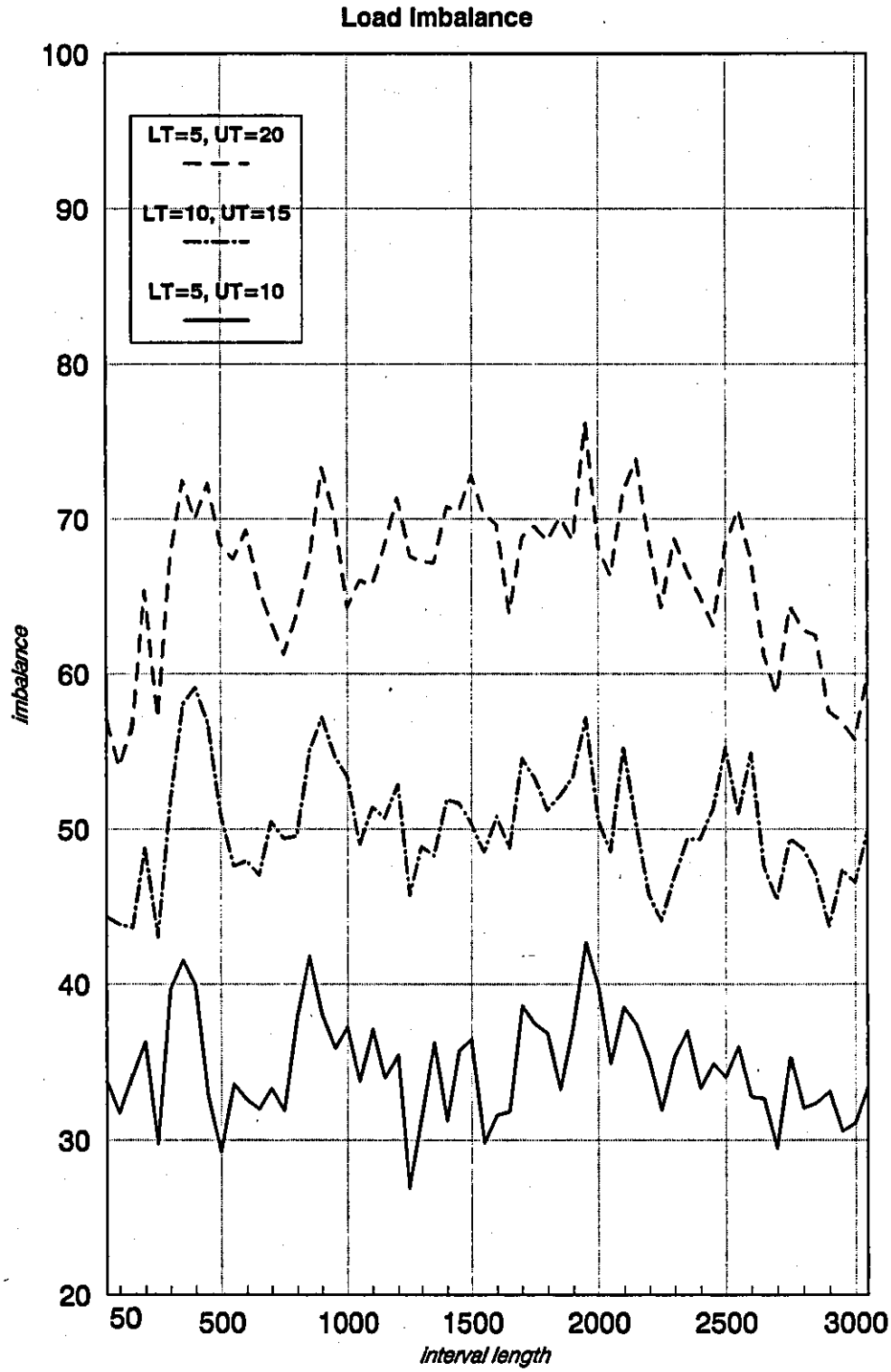

Fig.6.4: Load Imbalance Comparison of Threshold-Window Algorithm For Different Window Sizes.

lower threshold. As already mentioned, this algorithm is a server-initiated algorithm [Wang and Morris 1985] and the transfers can only take place when a lightly loaded server announces its willingness to accept some extra tasks for processing. For a smaller value of lower threshold, the chances are reduced that a host will be available to share the load from other heavily loaded hosts. Hence, a proper selection of the lower threshold is crucial to the operation of this algorithm. A too low value will reduce the overhead of task transfers but it might limit the amount of load balancing. On the other hand, if the value of lower threshold is too high, it might result in a poor load balancing by reducing the number of hosts eligible for transferring a task. Similiar arguments are valid for the choice of a proper upper threshold. From Fig. 6.4, a very high load imbalance was observed for an upper threshold of 20. In this case, upper threshold was much higher than the average load value of the system, hence a very small number of hosts were eligible for transferring tasks for the purpose of load balancing. When lower threshold was increased to 10 and the upper threshold was decreased to 15, a considerable reduction in the load imbalances was achieved. A further decrease in the upper threshold resulted in improved performances. It suggests that the upper threshold of 10 and the lower threshold of 5 manage each host to maintain its load within a close range of the average load of the system. In a real distributed computer system, the average load of the system is changing dynamically, therefore it is very difficult to obtain a single estimate of average load for adjusting the thresholds. However, if the window size is increased and proper threshold values are selected, small changes in the average load values can be effectively handled by this algorithm. The load imbalance and the response times given in Table 6.2 indicate the performance dependence of this algorithm on the

LOWER THRESHOLD	LT=5	LT=10	LT=5
UPPER THRESHOLD	UT=20	UT=15	UT=10
RESPONSE TIMES	55	49	41

Table 6.2: Average Response Times for Threshold-Window Algorithm

choice of window thresholds.

Next, the value of the message broadcast period was varied and its effect on the response times and task-transfers was studied. Figure 6.5 shows the average response time for different values of the message broadcast period. It is shown that the average response times are very high for the values of message broadcast period less than the mean task service time. Since task transfers were free in these simulations and it was assumed that an eligible task is immediately transferred to a lightly loaded host without any delay, the frequency of RECV message broadcasts less than the mean task service time floods a lightly loaded host and causes processor thrashing subsequently. It can be seen that the best performances for this algorithm are achieved when the message broadcast period is set in the range of 1-3 times the mean task service time. For a small distributed system of four hosts, when the message broadcast period increases beyond three times mean task service time then the average response times start increasing. This increase in the response times is due to the limit on load balancing imposed by a larger RECV message broadcast period. In this way, a heavily loaded host is not informed of a lightly loaded host even though there are some lightly loaded hosts in the system. If the message broadcast period is set large, load balancing performance can be improved by either increasing the number of tasks transferred

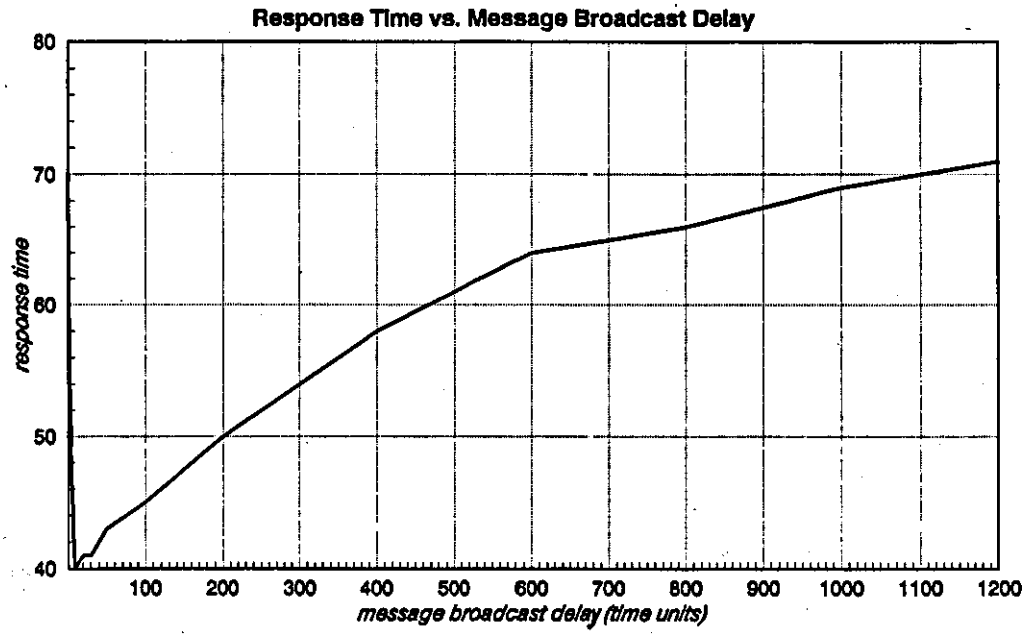

Fig.6.5: Response Time For Different Message Delays (Threshold-Window).

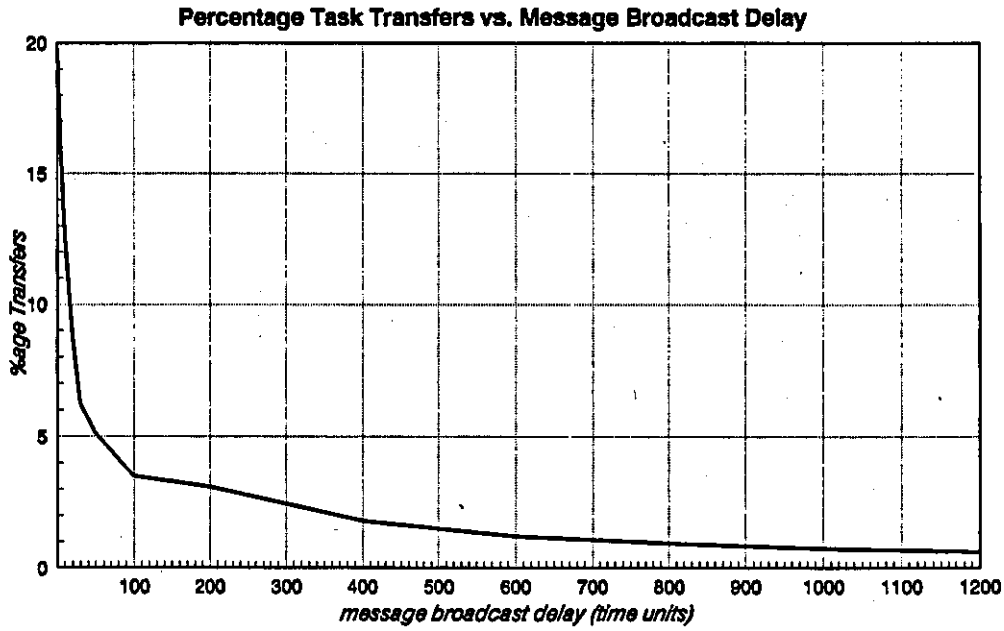

Fig.6.6: Percentege Task-Transfers for Different Message Delay Periods.

(Threshold-Window Algorithm)

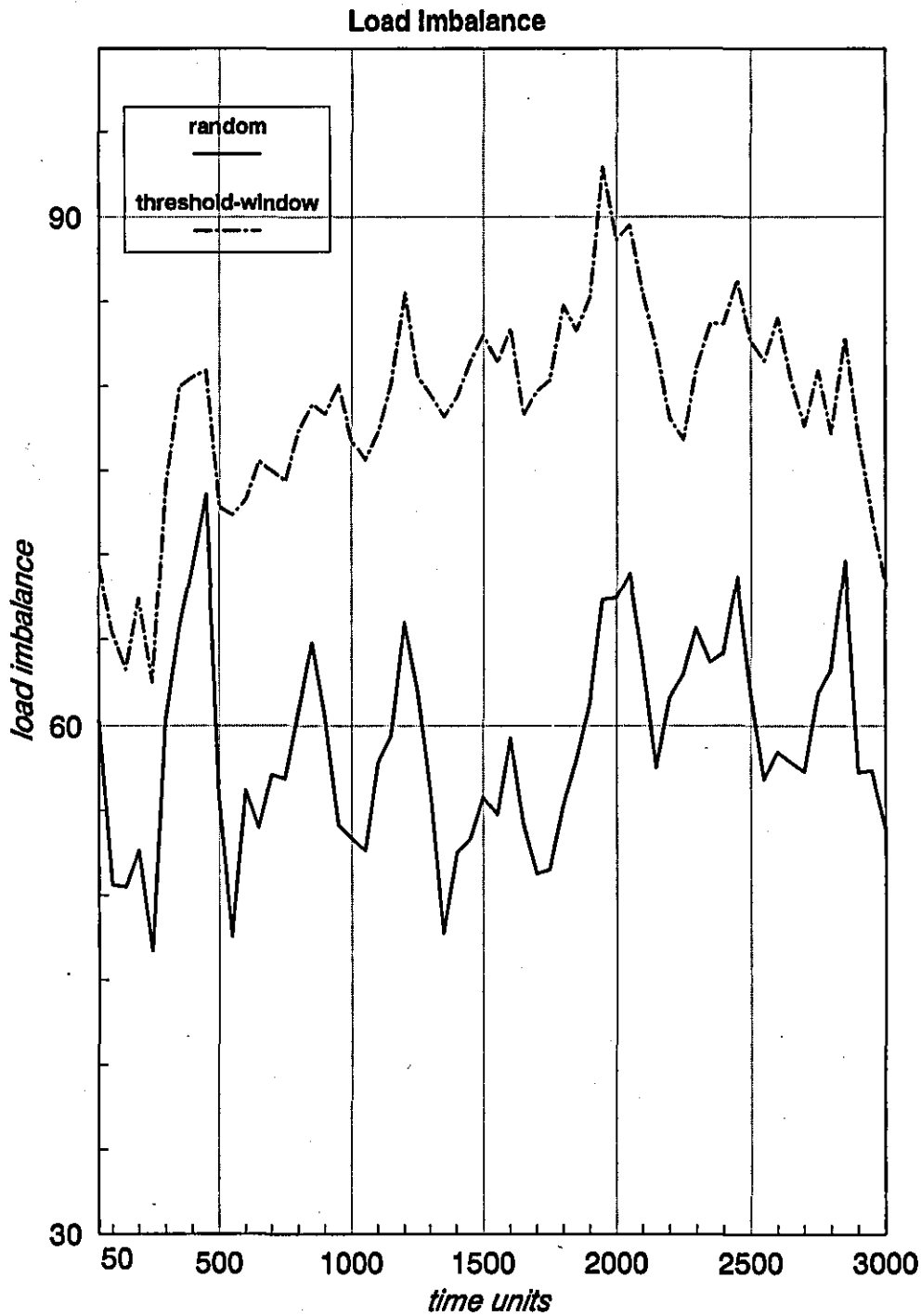

Fig.6.7: Load Imbalance Comparison of Random and Threshold-Window Algorithms.

by a heavily loaded host at the receipt of a RECV message or by allowing a heavily loaded host to send a task to a randomly chosen host if the RECV flag is not set. The threshold-window with random task transfers will be discussed later in this section.

In Fig. 6.6, the effect of message broadcast period on the number of tasks transferred is studied. The results indicate very high percentage of tasks transfered for message broadcast periods less than 30. Looking at the response times and percentage task-transfers simultaneously, it is suggested to use the message broadcast periods that provide reasonable response times with small percentage of tasks transfers, i.e; 40-200. These values largely depend on the size of the distributed system, the overall load conditions and the difference of load among the hosts.

Fig. 6.7 provides a load imbalance comparison between the Random algorithm and the threshold window algorithm for a broadcast period of 400. Random algorithm gives low load imbalance at the cost of higher task transfers and expensive periodic broadcasts amongst all the hosts. For a broadcast period of 400, the percentage of task transfers for Random algorithm is over 4 while it is slightly over 1 for the threshold-window algorithm. It should also be noted here that the periodic message broadcasts for threshold-window algorithm are only from lightly loaded hosts.

As already mentioned, the choice of upper and lower thresholds greatly effects the performance of threshold-window algorithm. It was decided to improve the performance of this algorithm by setting two upper thresholds. Whenever, a heavily loaded host reaches the first upper threshold, it checks

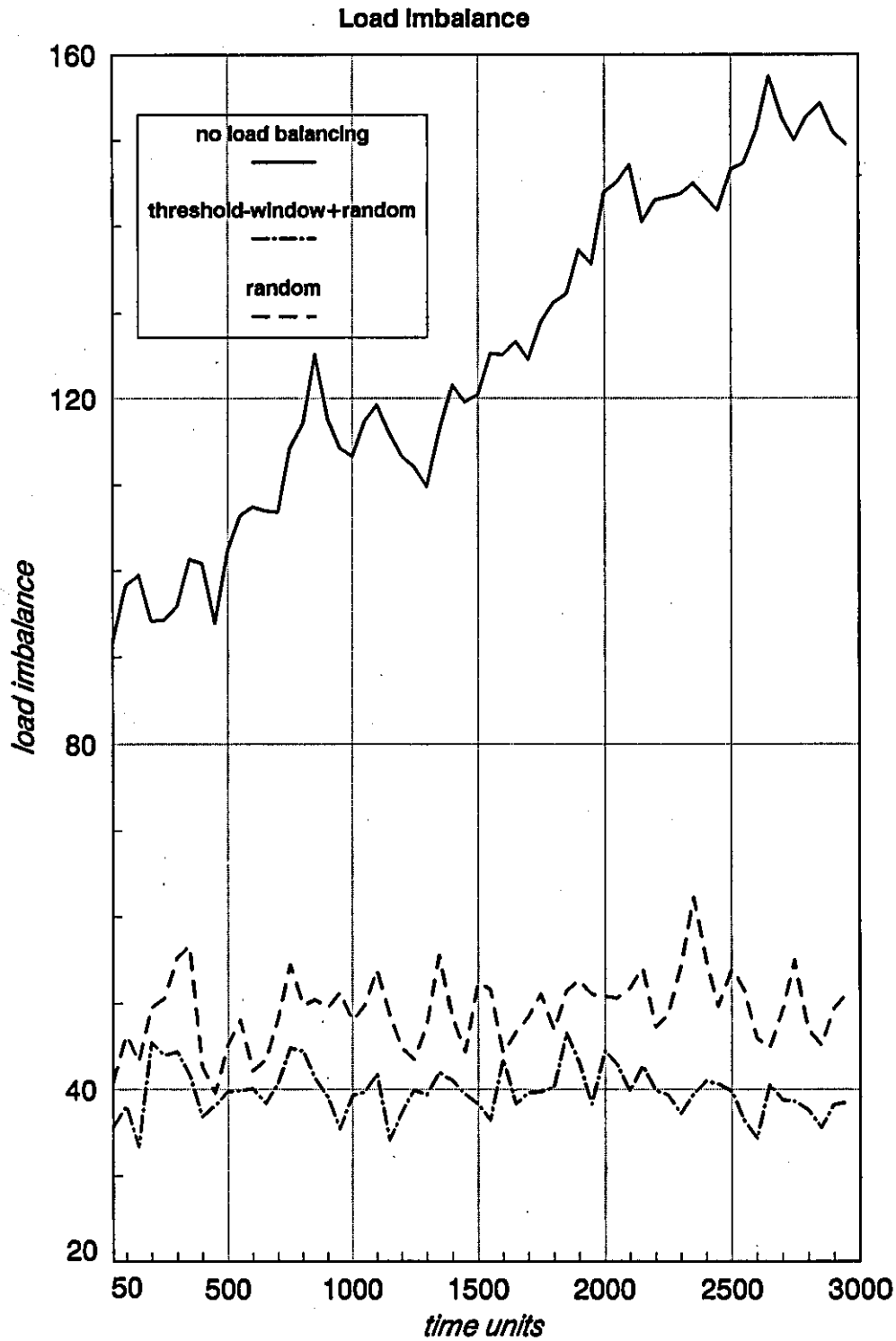

Fig.6.8: Load Imbalnace Comparison of Random and Modified Threshold-Window Algorithms

if the RECV flag of a host is set and transfers the load to that host. If a heavily loaded host does not find RECV flag set for a considerable amount of time and reaches the second upper threshold, it transfers a task to one of the hosts selected randomly. Fig.6.8 compares the load imbalance performance of modified threshold-window and Random algorithm for a broadcast period of 200 time units. It was observed that the modified threshold-window algorithm gives a better performance than the Random algorithm at comparatively lower communication costs. It should also be noticed that, in the worst case when the lower and upper threshold are not chosen carefully or there are large load variations, this algorithm will achieve some degree of load balancing by transferring tasks to randomly chosen hosts.

6.4 Dynamic Load Balancing Using Task-Transfer Probabilities

In this section, various dynamic load balancing strategies are proposed and the factors influencing their performance in a distributed system environment are discussed. Several dynamic algorithms, ranging from very simple ones to really complex ones, have been designed to improve the performance of load balancing in distributed systems [Eager et. al. 1986]. All these algorithms have shortcomings in terms of the assumptions made concerning the task model and the processing system. The dynamic algorithms, presented here, assume a distributed computer system with the following properties.

- Task transfers are through task placement and not task migration. This reduces the cost of task transfers and can easily be implemented in many systems.

- Task behaviour is completely unknown. No assumptions are made about the task size or the distribution of task arrival.
- All the processors are identical.
- After a fixed interval of time, each processors' current load is distributed to all other processors in the system.
- Task transfer decisions are made at the time of arrival of each task.

In these algorithms, based on the current state of the system load, task transfer probabilities were computed for each host in the system. These probabilities, denoted by P_{ij} are between a host i and every other host j. Task transfer probabilities are computed at the end of each periodic interval and subsequently adjusted after the transfer of each newly arrived task.

6.4.1 Dynamic Task-Transfers Using Estimated Service Times

In the first case, mean service rate was used as the load index to perform the dynamic load balancing. In this model, during a given time interval, mean service rate μ can be estimated by keeping record of the total time used by the host in serving tasks and the number of task departures during that interval. Therefore, at a particular instant of time t,

$$S_i(t) = \frac{1}{\mu_i(t)} = \frac{\Delta t}{d_i(t)}$$

where,

$S_i(t)$ = Total service time for local jobs and remote jobs Δt

$d_i(t)$ = Total tasks leaving the host in Δt

similarly,

$$\frac{1}{\overline{\mu}_i(t)} = \overline{S}_i(t) = \frac{S_1(t) + S_2(t) + \ldots + S_n(t)}{n}$$

where n is the total number of hosts in the system. The necessary condition for load balancing using this algorithm is that at least one host in the system is heavily loaded and constantly processing tasks. If a host was idle at any time during the load update period, its service time is set equal to zero.

$\overline{\mu}(t)$ is the average service rate for the entire system. The dynamic load balancing algorithm using mean service rate as the load index is given below. This algorithm is executed at the end of each load information distribution interval.

ALGORITHM I :

```
For i = 1 to number of processors in the system {
    estimate the service time μ(t) from the load updates
    received in the last interval.
}
for i = 1 to number of processors{
    if (S_i(t) ≥ S̄(t) + ε)
/* add host to the donor group */
        donor++;
    else if (S_i(t) < S̄(t) − ε)
/* add host to the acceptor group */
        acceptor++;
    }
```

/* Determine the weight for donor hosts for finding transfer probabilities */

for $i = 1$ to all donors

/* excess tasks on hosts i compared to the average */

$$W(t) = \frac{(S_i(t) - \overline{S}(t))}{S_i(t)} ;$$

/* finally calculate the probabilities for the transference of tasks among donor and acceptor hosts */

for $i = 1$ to all donors {

for $j = 1$ to all acceptors {

$$P_{ij} = (S_j(t) / \sum_{j=1}^{all\ acceptors} S_j(t)) W_i$$

}

}

/* probability that a task will be executed on the local hosts */

for $i = 1$ to all donors {

$$P_i = 1.0 - \sum_{j=1}^{all\ donors} P_{ij};$$

}

The algorithm described above attempts to improve the performance of a distributed system by balancing the load through the estimated service time on each host. Each host estimates its own service time during each interval and distributes this information to all the other hosts. After this information is passed to all the hosts in the network, transfer probabilities are carried out for each host. These transfer probabilities are used to transfer tasks during the next interval for the purpose of load balancing.

[Gao et. al. 1984] have introduced the idea of donor and acceptor hosts to indicate the load levels in each host. A donor host is a heavily loaded host and is capable of transferring a few tasks to an acceptor host, a lightly loaded host, for an improved performance. These definitions also apply in this study.

Given the estimates of service times for each host, the next step is to check if a host is a donor or an acceptor. To check this, each host compares its average service time with the average service time over the whole system and if the difference exceeds a threshold ϵ[2] then this host is categorized as a donor host. The value of ϵ varies with the processing speed of hosts, the task size, and the communication parameters of the network. The value of ϵ should be such that the load values equal to ϵ, in excess of the average load value, when transferred to a remote host do not provide any performance gain. Any load values greater than average load upto an ϵ should be executed locally as the waiting time at local host might be lesser than the time spent in communications for transfer to a remote host. Similarly, if the estimated service time of a host is less than the average service time of the system by an amount greater than ϵ , this host is categorised as an acceptor host.

Remaining steps of the algorithm are carried out by the donor hosts. After a host has been categorized as a donor host, this host determines the weight factors to compute the transfer probabilities. This weight factor is defined as the ratio of excess load as compared to the overall average load of the system.

[2]An ϵ is similar to the parameter 'bias' used earlier

Before defining the transfer probabilities, it is important to define the term Acceptance Ratio.

$$Acceptance\ \ Ratio = \frac{\sum_{j=1}^{all\ acceptors} S_j(t) - S_j(t)}{\sum_{j=1}^{all\ acceptors} S_j(t)}$$

Hosts with smaller service times have high acceptance ratios. Therefore, more tasks from donor hosts are transferred to the acceptor hosts with higher acceptance ratios.

Finally task-transfer probabilities are determined between a donor and acceptor host by multiplying the weight factor of each donor host with the corresponding acceptance ratios of the acceptors.

6.4.2 Dynamic Task-Transfers Using The Combined Effect Of Task Arrivals And Departures In A Distributed Computer System

In the previous algorithm, task departures from a host were considered to estimate the service times for the purpose of load balancing. By taking task departures, over a given period of time, into consideration, it is possible to get an estimate of the average task size on a host and to determine that a host is idle in case of no departures. This limited amount of information may not be sufficient for efficient load balancing, as the load on a host is also dependent on the arrival pattern of the tasks. Therefore, the next algorithm takes both the arrivals and departures into consideration for making a load balancing decision.

In this algorithm, the following values are recorded, during each interval (Δt), to be utilized in making load balancing decisions.

$d_i(t)$ = Number of tasks leaving host i at time instant t.

$a_i(t)$ = Number of tasks arriving at host i at time instant t.

These values are distributed to every other host periodically after $\Delta(t)$ interval.

$$\delta_i(t) = a_i(t) - d_i(t)$$

$\delta_i(t)$ is the load on each host during an interval $\Delta(t)$

Average load over the entire system is given by,

$$\overline{\delta}_i(t) = \frac{\sum_{i=1}^{n} \delta_i(t)}{n}$$

Here, n is the number of hosts in the distributed computer system.

ALGORITHM II :

```
for i=1 to n {
   if (δ_i(t) ≥ δ̄(t) + ε)
/* Add a host to the donor group */
      donor++;
   }
   else if ((δ_i(t) ≤ δ̄(t) − ε)){
/* Add a host to the acceptor group */
      acceptor++;
   }
```

/* Determine the weights for donor hosts for finding the transfer probabilities */

for $i = 1$ to all donors {

$W_i(t) = \delta_i(t) - \overline{\delta}(t)$

}

/* Finally calculate the probabilities for the transferrence of tasks amongst the donor and acceptor hosts */

for $i = 1$ to all donors {

for $j = 1$ to all acceptors {

$P_{ij}(t) = ((\overline{\delta}(t) - \delta_j(t))(W_i(t)/\sum_{i=1}^{all\ donors} W_i))/\overline{d}(t)$

}

}

This algorithm is very much similar to the first algorithm described. The only difference is the load index. In this case it is expected that the load index will reflect a better estimate of each hosts' load. By monitoring the arrivals and departures, on each host during the interval Δt, it can be estimated how the load will vary during the next interval.

This can be expected on the basis that there is a greater likelihood that the events that happened in the immediate past will be happening in the immediate future.

The weight factors are determined by the difference of the hosts' load level and the average load of the system. To estimate the probability of tasks that need to be transferred from a donor, the ratio of this weight to the total weight of the whole system is calculated. To find the probability of task-

transfer, from a donor to a particular acceptor, the weight ratio is multiplied with the acceptance level of the acceptor. The acceptor level is determined by the difference in the average load of the system and the current load on the acceptor. The larger the difference the higher the acceptance level. In the end, the resultant is normalised by the average number of jobs processed during the interval to get the final probability.

6.4.3 Dynamic Task-Transfers Using Queue-Length As The Load Index

The algorithm described in this section is based on a widely used load index, i.e., queue-length, for dynamic load balancing in a distributed computer system. A large number of research efforts have been directed on finding a useful load balancing policy using queue-length as the load index. It is generally believed that if task behaviour is known in advance, efficient load balancing strategies can be devised. One of the major unknowns in task behaviour is the overall processing time required for the completion of a task. The previous two algorithms, attempted to improve the performance of a distributed system using the estimates of task arrivals and task processing times. The combined effect of the arrival and service rates is automatically reflected into the queue-lengths of a host. Instead of estimating these quantities independently and using these for load balancing purposes, queue-length estimates of different hosts are used to perform the load balancing.

Like the previously described two algorithms, this algorithm is distributed and runs on each host periodically. The initial steps of finding the average load on each host and identifying the donor and the acceptor hosts are similar to the ones described earlier.

Instantaneous queue length for each host 'i' is represented by $q_i(t)$ and is broadcast to all hosts in the system periodically. Average load over the distributed system during each interval is given by:

$$\overline{q}(t) = \frac{\sum q_i(t)}{n}$$

where n is the total number of hosts in the system.

ALGORITHM III :

for $i = 1$ to no. of processors {
  if $q_i(t) > \overline{q}(t) + \epsilon$
/* Add host to the donor group */
    donor++;
  else if $q_i(t) < \overline{q}(t) - \epsilon$
/* Add host to the acceptor group */
    acceptor++;
}

for $i = 1$ to all donors {
  $W_i^d(t) = q_i(t) - \overline{q}(t)$
}

for $j = 1$ to all acceptors {
  $W_j^a(t) = \overline{q}(t) - q_j(t)$
}

/* Determine the weights for finding transfer probabilities */

for $i = 1$ to all donors {

for $j = 1$ to all acceptors {

$W_{ij}(t) = (W_i^d(t) / \sum_{i=1}^{all\ donors} W_i^d(t)) W_j^a(t)$

}

}

/* Calculate the transfer probabilities between each donor and acceptor hosts */

for $i = 1$ to all donors {

for $j = 1$ to all acceptors {

$P_{ij}(t) = W_{ij}(t) / \sum_{i=1}^{all\ donors} W_{ij}(t)$

}

}

In this algorithm, the weight factors for the donor, W^d and acceptor hosts, W^a are determined by comparing each hosts' queue length with the average length of the system. Next, the transfer weights between a donor 'i' and an acceptor host 'j', W_{ij}, is calculated by multiplying the ratio of the donors' weight to the sum of all donors' weights with each acceptors' acceptance level. An acceptors' weight is taken as its acceptance level. The transfer probabilities for a donor 'i' and an acceptor 'j', P_{ij} is simply the ratio of transfer weight of these two hosts to the sum of all transfer weights.

These task-transfer probabilities are adjusted after each task transfer for dynamic load balancing.

6.5 Simulation Results And Discussion

The simulations were carried out to study the performance of algorithms described in the previous section. The simulation model was based on the assumptions and characteristics described in section 6.2. In these simulations, it was assumed that the load information for each host is immediately available to every other host in the system for making transfer decisions. In real systems there is a considerable delay in spreading this information across the system. The delays in distributing load information depend on the communication capabilities and the communication protocols of the network. However, the effect of transfer delays is considered while defining the load threshold of each host. As described in section 6.2, parameter ϵ is used to minimize the effect of these delays on load balancing decisions.

The interval between two successive updates is varied to study the effect of the frequency of load information updates on the load balancing algorithms. The choice of a suitable period length is necessary for a good load balancing algorithm with minimum overhead on the communication subsystem.

The parameters used to generate the arrival and service rates are the same as described by [GAO et. al. 1984], given in Table 6.1. These parameters are selected such that the utilization, ρ, of the queueing system is in the stable region.

$$(\rho = \lambda/\mu) < 1$$

For each algorithm, the load imbalance and the average response time of tasks was compared with the no load balancing case. A 'no load balancing' can be viewed as an independant M/M/1 queueing system, where each arriving task on a host resides on the same host for its lifetime, independent of the load on this host. An ideal load-balancing system can be viewed as an M/M/m queuing system. If a single central queue is used and each task is sent to a processor on demand, the system will behave similar to an M/M/m queuing system. All the results, shown in Figures 6.9-6.11 are obtained for an update interval of 200. It is commonly believed that with known task sizes, the performance of load balancing algorithms can be increased tremendously. Figure 6.9 presents the load imbalance curves for the no load balancing and load balancing using estimated service times. From the no load balancing curve, it can be observed that as long as the new tasks are arriving and the host queues are full, the overall load imbalance tends to increase. These results are very sensitive to the arrival and service rate distributions. The amount of load imbalance is dependent on the arrival and service parameters used for each host. The curve generated for load balancing algorithm using estimated service time as the load index improves the performance by decreasing the load imbalance. However, the low performance gain can be associated to some extent with the small variations of service rate among the hosts. The load imbalance is reasonably controlled for the first half of the curve and then it gradually increases and then again falls to the same initial range. These fluctuations in the load imbalance could be the cause of largely varying arrival rates among the hosts. These un-controlled arrival rates make the behaviour of the load balancing algorithm unpredictable.

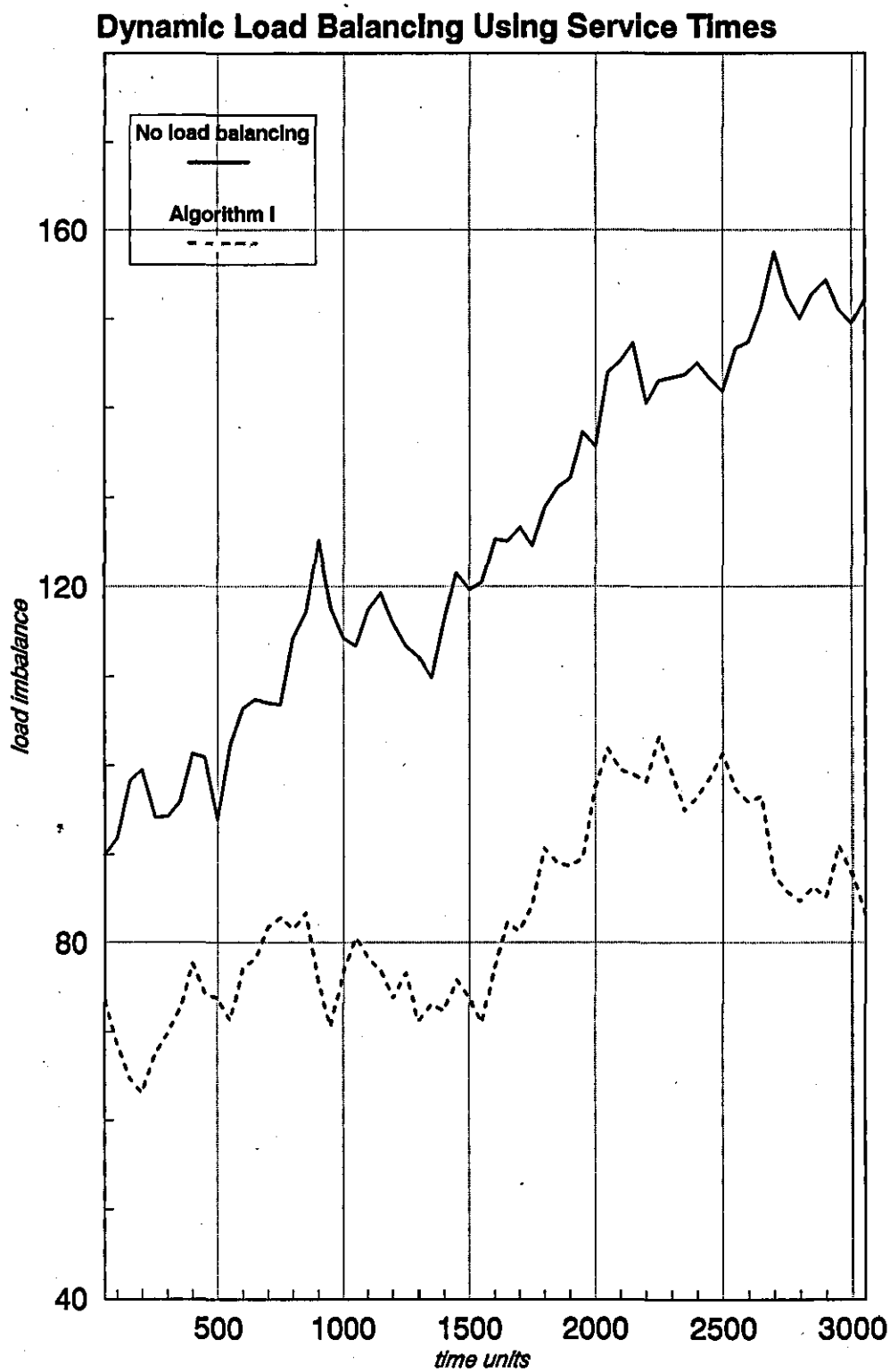

Fig.6.9: Load Imbalance For Algorithm I vs. No Load Balancing

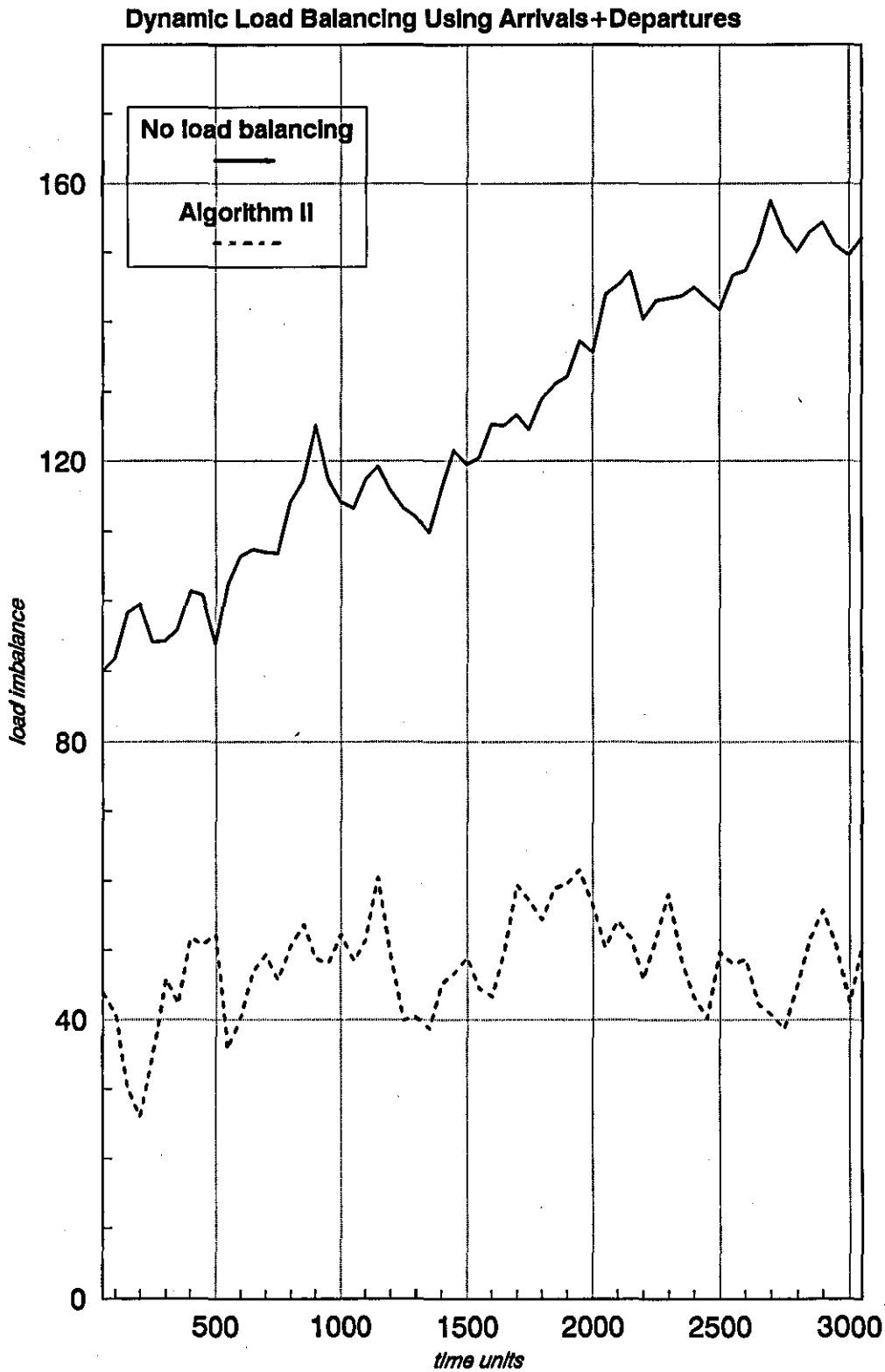

Fig.6.10: Load Imbalance For Algorithm II vs. No Load Balancing

Figure 6.10 shows the performance curve for load balancing algorithm that uses the estimates of both the arrival and service rate, to make the task transfer decisions. In this case, it was observed that there was considerable improvement in the load balancing. The load imbalance was nearly uniform over the whole simulation period. There were no wide variations in the load imbalance as observed in the first case. It indicates that a combination of arrival and service rates can be effectively used to determine a load index for good load balancing. Such algorithm performs well for widely varying task arrivals and task sizes, thus the performance of load balancing algorithms can be increased tremendously.

The estimates used in this algorithm are achieved dynamically from the tasks executing in the previous interval. The next algorithm does not utilise any such estimates of task size or task arrivals.

Figure 6.11 shows the performance of load balancing using queue length as the load index. The queue length was chosen as the load index based on the intuition that both the arrival rate and the service time of a host are reflected in its queue-length. In all the three algorithms described, load balancing decisions are made on the basis of load conditions observed during the last update interval. The effect of load estimates obtained in the past intervals is not considered.

In this figure, a further improvement in the performance of load balancing is observed. This suggests that the automatic reflection of arrival and

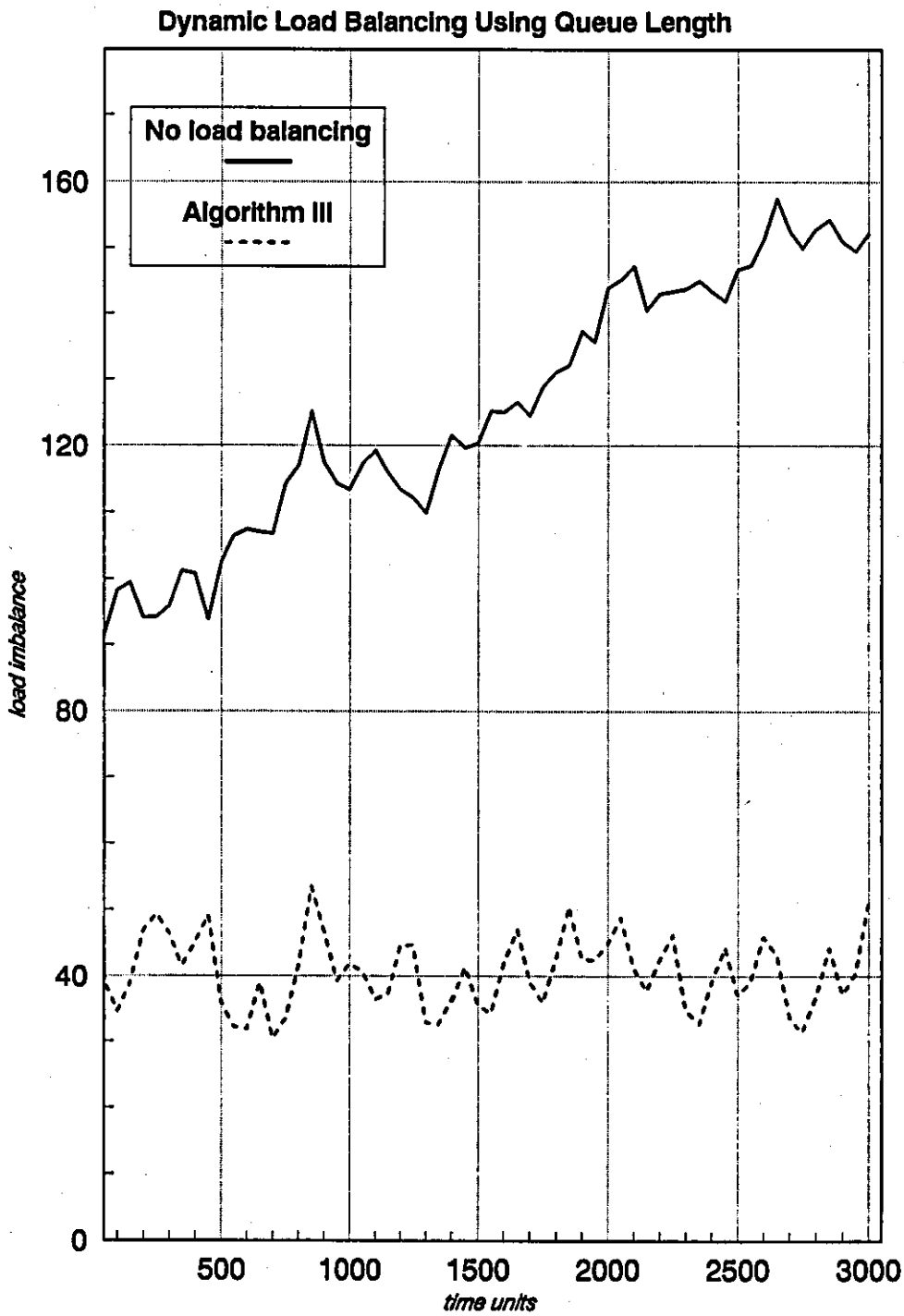

Fig.6.11: Load Imbalance For Algorithm III vs. No Load Balancing

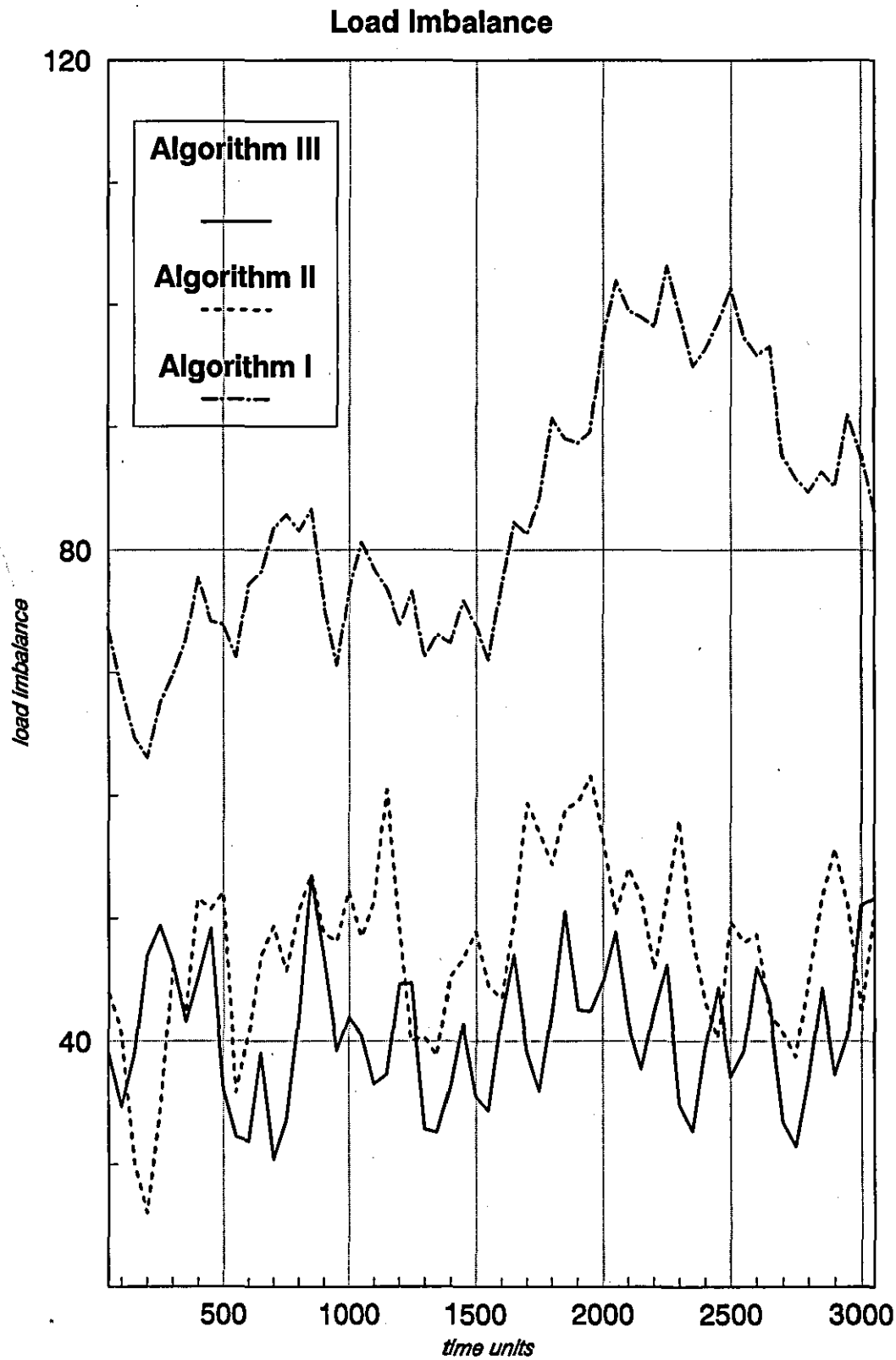

Fig.6.12: Load Imbalance Comparison for Algorithms I, II, and III

service times in the queue-length provides a better measure of the load on a host, as compared to obtaining the separate independent estimates for arrival and service times. It also involves lesser overhead in maintaining and distributing the load information.

The relative performances of the three algorithms are described in Fig. 6.12 with their corresponding load imbalance curves. As already mentioned, the load imbalance for Algorithm *II* and Algorithm *III* was considerably lower than Algorithm *I*. For smaller interval lengths, it was observed that there were not significant differences between the performances of Algorithm *II* and *III*. An interval length of 200 was used for the results in Fig.6.12. For larger intervals, Algorithm *III*performed significantly better than Algorithm *II*. This is probably due to the nature of parameters used in determining the load estimates. The arrival and service time parameters used in Algorithm *II* change more frequently than the queue-length, therefore the load estimates obtained for the former are not very effective for longer update intervals.

It is very important for a dynamic load balancing algorithm to be stable. In distributed computer systems, frequent changes in the load of hosts and the out-of-date information may lead to inefficient load balancing decisions. The selection of update interval is very crucial in dynamic load balancing algorithms. The length of this interval should be greater than the average task service time and the average communication delay, across the network. In this model of the distributed computer system, it was assumed that the tasks are immediately transferred between any two hosts, which helps to provide us with an updated load information of each host instantly for an efficient load balancing decision. In real environment, these algorithms may

ALGORITHM	NO LOAD BALANCING	I	II	III
AVERAGE RESPONE TIME	77	57	46	43

Table 6.3: Average Response Times For Algorithms *I*, *II*, and *III*.

show some instability due to processor thrashing. This processor thrashing is caused by unnecessary exchange of tasks between two processors due to incorrect load balancing decisions, based on out-of-date information. However, with a proper adjustment of the load threshold, these algorithms can avoid processor thrashing.

The average response time of the three algorithms are given in Table 6.3. These response times are precisely due to the waiting time in the processors' queue and the processing time of the task.

Next the performance of algorithm *III* was studied by changing the information update period lengths. As already mentioned, the frequent information update messages incur a large amount of overhead and are likely to degrade the performance of distributed systems due to communication bottlenecks. Therefore, it is necessary to find the interval lengths that provide good response times with a manageable amount of communication overheads.

Figures 6.13-6.14 show the response time and percentage task-transfers for different interval lengths. In Fig. 6.13, the response times obtained for Algorithm *III* show a gradual increase with an increase in the interval lengths. For longer intervals, load estimates do not capture fine load variations. These estimates are used to calculate the probabilities for transferring the tasks

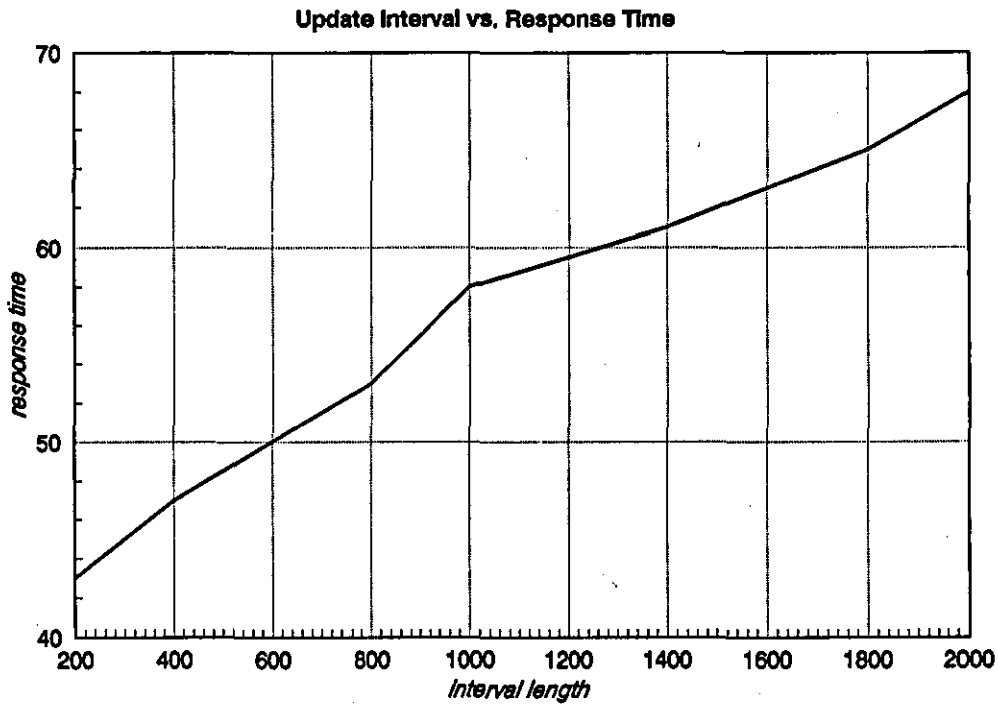

Fig.6.13: Average Task Response Times for Algorithm III.

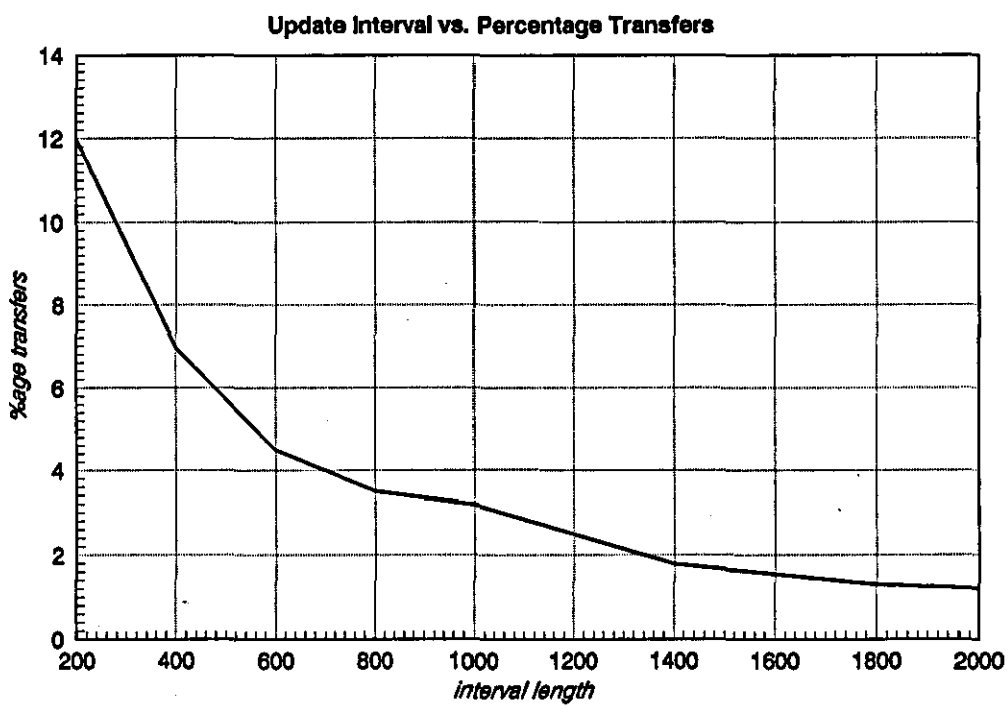

Fig.6.14: Percentage Task-Transfers for Algorithm III.

which remain effective during the succeeding interval until new estimates are obtained. With large interval lengths, the difference in the estimated and actual load increases and the probabilities based on these estimates are used for longer intervals hence effecting the performance of load balancing. However, comparing the performance of Algorithm *III* with Random and threshold-window algorithms that transfer tasks once during the whole interval length, the former gives very good performances for longer interval lengths as it tries to adapt with the dynamically changing load conditions on the hosts.

Fig. 6.14 shows a sharp decrease in the percentage task-transfers for an increase in the interval lengths up to 600. After the interval length of 600, there is a gradual decrease up to the interval length of 1200. From 1200 onwards the percentage of task transfers stays nearly constant. Looking at the response times and the percentage transfers for Algorithm *III*, it can be suggested that very good performances can be achieved for the interval lengths of 400 and 600 with low task-transfers and response times. With a proper choice of the interval length, a suitable compromise between communication costs due to task-transfers and response times can be achieved. It should be noted here that with an increase in the interval length, the communication overhead due to periodic load information broadcasts is greatly reduced.

[Gao et. al. 1984] have used a similar model to study the performance of load balancing algorithms. The two algorithms described in the authors' study transfer a fixed number of tasks during each interval. The load estimates used to make the load balancing decisions are based on the task arrivals and the amount of unfinished work. The authors' results indicate

that the algorithm using arrival rate as the load index performs poorly as compared to the estimate of the total unfinished work. The total unfinished work on each host is estimated from the estimate of the mean service time of local tasks on the host multiplied by the queue-length. It should be noted here that if the service time of the local tasks on a host is low, the tasks with larger service times will be constantly transferred to this host and yet unaccounted for in the load estimate, thus resulting in poor load balancing. The algorithms described in this study take both the local and remote tasks into account for estimating the current load on a host. These algorithms make dynamic load balancing decisions for transferring newly arrived tasks after the task-transfer probabilities have been computed. Algorithm *II* and Algorithm *III* give a better performance as compared to Algorithm *I*.

6.6 Dynamic Load Balancing With Communications

The dynamic load balancing algorithms discussed so far in this chapter do not consider communication costs while making the task-transfer decisions for load balancing. For load balancing, tasks are transferred from heavily loaded hosts to the lightly loaded hosts. A task on a heavily loaded host may be eligible for transfer to more than one lightly loaded hosts. A suitable choice of the lightly loaded host that minimizes the transfer costs can further improve the response time of the task. In this section, Algorithm *III* described previously is modified to include the inter-host communication costs. The host model and the parameters used are the same as described in section 2 of this chapter.

COMMUNICATION LINK	C_{12}	C_{13}	C_{14}	C_{23}	C_{24}	C_{34}
HIGH COMMUNICATIONS	16	2	8	6	4	16
LOW COMMUNICATIONS	8	1	4	3	2	8

Table 6.4: Inter-host communication parameters

The communication subnet is considered to be fully connected and the communication costs for each link are given in Table 6.4. The communication costs considered here simply reflect the possible delays experienced by a task between two hosts. These delays could be due to the link speed, communication interface or network congestion etc. A set of high and low communication costs was considered to study the performance of the modified algorithm.

In Algorithm *III*, after the weights have been assigned to each donor-acceptor pair based on the current system load, these weights are subsequently adjusted to reflect the inter-host communication costs. Following steps are carried out to find new weights used in determining the task-transfer probabilities.

W_{ij} = Weights determined in Algorithm *III* for each donor i and acceptor j.
$newW_{ij}$ = Weights determined after the inter-host communications are considered, for donor i and acceptor j.
max_com_i = maximum communication cost between a host i and one of the acceptors.

Algorithm :

for $i = 1$ to all donors {
    for $j = 1$ to all acceptors {
        $max\_com_i = max\{C_{ij}\}$
    } } for $i = 1$ to all donors {
    for $j = 1$ to all acceptors {
        if $(C_{ij} < max\_com_i)$ {
            $newW_{ij} = W_{ij} + ((max\_com_i - C_{ij})/max\_comi)$
        }
        else{
            $newW_{ij} = W_{ij} - ((max\_com_i - C_{ij})/max\_comi)$
        }
    }
}

$newW_{ij}$ determined in this way was used to compute the probabilities, P_{ij}, for transferring tasks from a host i to a host j, as described in the previous section.

6.6.1 Performance Results

Simulations were performed to evaluate the performance of the modified algorithm. The simulation model parameters were the same as used for Algorithm *III*. Two different sets of communication parameters used in simulations are given in Table 6.4. The performance was measured by comparing the results obtained for this algorithm with those of Algorithm *III*. The

ALGORITHM	HIGH COMMUNICATION COSTS	LOW COMMUNICATION COSTS
ALGORITHM *III*	66	51
MODIFIED ALGORITHM *III*	55	47

Table 6.5: Average Response Times for Modified Algorithm *III*.

performance measures used were the average response time and the load imbalance. As expected, significant improvements in the response times were achieved. When the difference of communication costs amongst the communication links is higher, larger improvements in the response time were achieved. The average response times achieved for the two types of communication costs are given in Table 6.5.

The effect of the modified algorithm on the load imbalance for high and low communications costs is shown in Fig.6.15 and Fig.6.16 respectively. It appears that the improvement in the response times was achieved at the cost of higher load imbalances. Algorithm *III* which only tries to balance the load amongst the hosts without taking communications into consideration, has low load imbalance values. The conflicting issue of minimizing the communication costs and maximizing the load balance is apparent from the results of the response times and load imbalance. In Fig.6.16, the load imbalance for modified algorithm is compared with Algorithm *III* for lower communication costs. In this case, the increase in the load imbalance for the modified algorithm is not very high and better response times were achieved with small adjustments in the host loads.

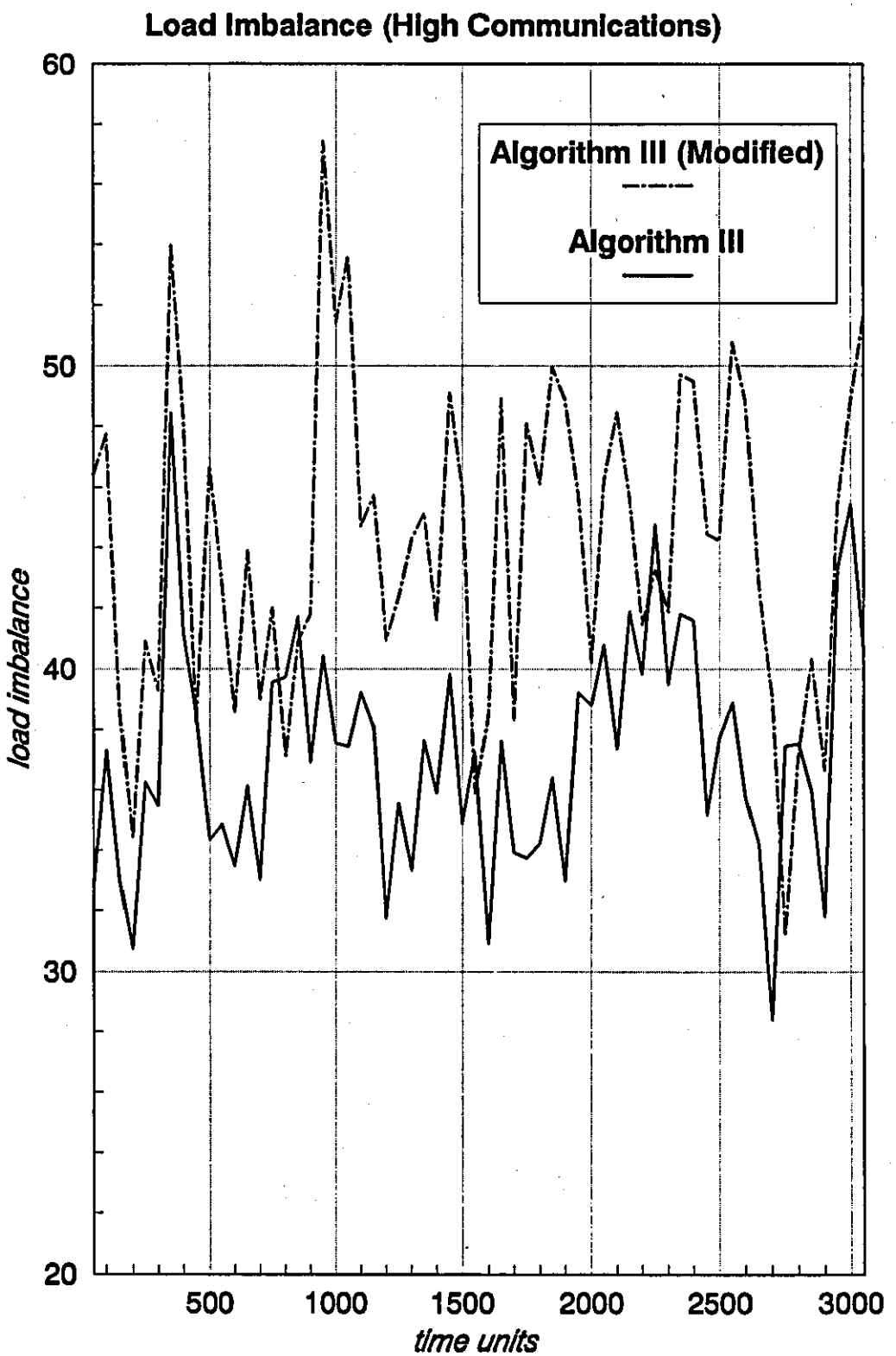

Fig.6.15: Load Imbalance for Algorithm III and Modified Algorithm III (With High Communications)

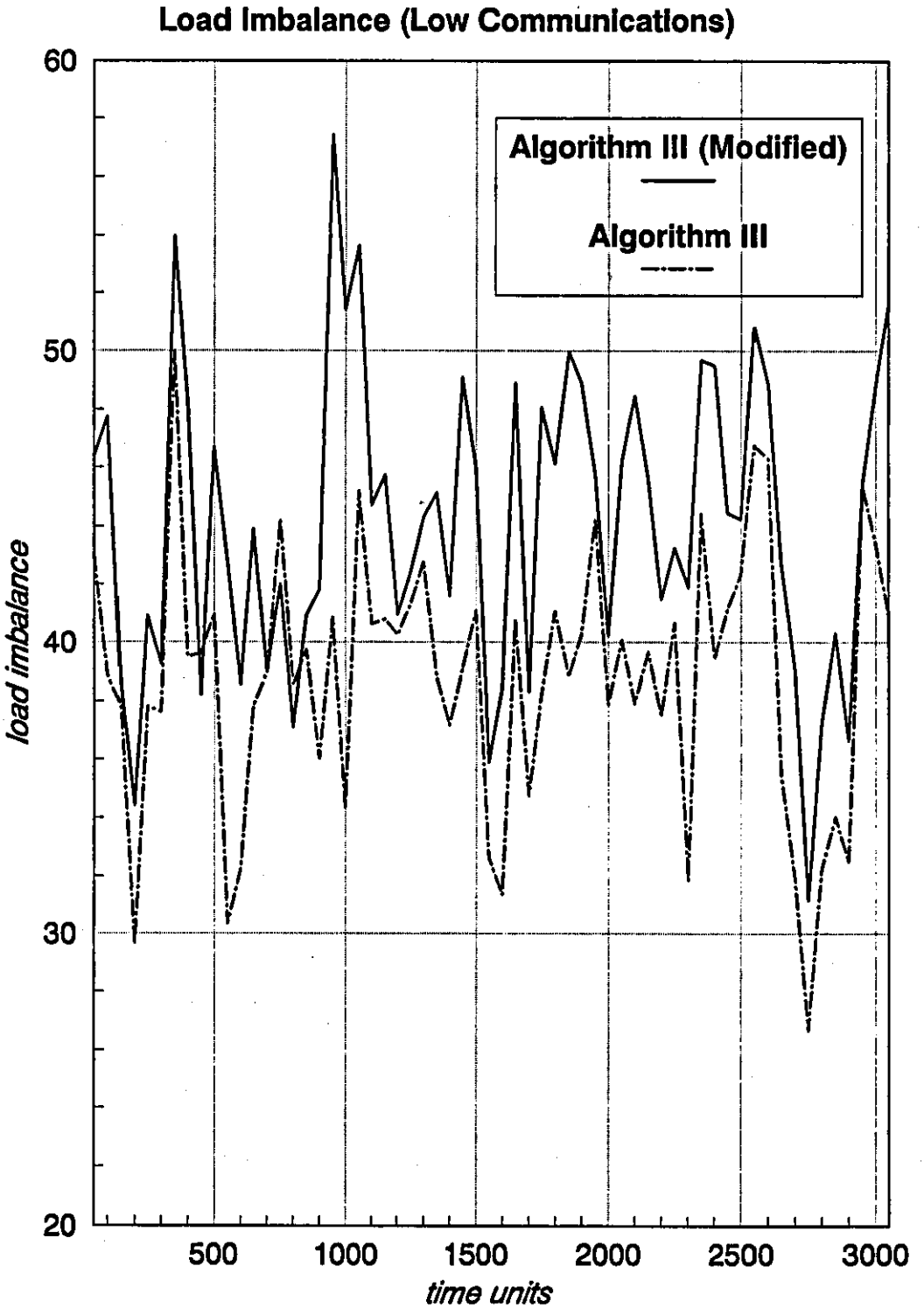

Fig.6.16: Load Imbalance For Algorithm III And Modified Algorithm III (With Low Communication Costs)

Chapter 7

LOAD BALANCING WITH NETWORK PARTITIONING USING HOST GROUPS

7.1 Introduction

As mentioned earlier, every dynamic load balancing algorithm consists of an information policy which is used to distribute load information amongst hosts of the distributed system. This load information is susequently used to make task transfer decisions for balancing the load across the distributed system. In the previous chapters, major emphasis was given to the design of load balancing algorithms with the assumptions of a periodic broadcast mechanism for load information distribution. There were no efforts made to reduce the communication costs incurred by such algorithms. The communication overhead for load balancing algorithms is dependent on both the information policy and the load balancing policy used. Information policy

decides the nature of the information which needs to be distributed and about the protocol to be used to distribute such information. Similarly, the amount of tasks transferred and the destination of such tasks is determined by the load balancing policy. The communication overhead also depends on the number of hosts and inter-connection topology of communication network in the distributed system. As the number of hosts and the size of the communication network increases, communication delays increase and tend to cause communication bottlenecks. Furthermore, with an increase in the number of hosts in the system, each host needs to increase its table space for maintaining the load information. As the table size increases, the computation time for maintaining the load information also increases.

In this chapter, an attempt is made to suggest various network partitioning techniques which could improve the performance of large distributed computer systems. The study presented here is suitable for a large distributed computer system with several hosts. These systems are distributed over large geographical areas and their hosts are inter-connected with long haul communication lines. The techniques presented here are aimed at reducing the communication overhead introduced by load balancing algorithms by restricting the application of these algorithms to a group of hosts. Several different criteria can be used for the grouping of these hosts. The present study assumes the geographical partitioning of the network along the natural boundaries of the geographical regions. Subsequently, the host grouping strategies are modified to achieve better load balancing performances.

Each host group is considered as a complete system. In this way, the load balancing process is limited to the host group by identifying the heavily and

lightly loaded hosts within the group and transferring tasks amongst them. It is expected that introducing such techniques would reduce the overhead in maintaining the load table and distributing the load information.

The next section describes some techniques for network partitioning and discusses the rationale for grouping the hosts to reduce the overheads. The remaining sections describe some simple network partitioning techniques and their performance through simulations.

7.2 The Network Partitioning

It is a very common practice to increase the size of communication networks for increased availability and reliability of the computing resources. However, at certain times, further increase in the computing resources creates problems for the software which manages these resources. To simplify the management of resources in a large distributed system, a reverse process is suggested to partition a large physical network into relatively small sized logical sub-networks. From a dynamic load balancing perspective, if a dynamic load balancing algorithm is applied directly to a large distributed system, it would require a large amount of overhead in maintaining the load table, distributing the required load information and transferring the tasks to achieve a required performance measure. The partitioning of a large network into smaller units reduces the number of hops and hence the communication cost incurred in load balancing.

There are several possible ways of partitioning a communication network. The two major categories can be described as;

- Hierarchical Partitioning
- Flat Partitioning

The relative performances of both types of partitionings mentioned above are subject to the implementation of the load balancing algorithm. The *wave scheduling* technique [cf. chapter 3] implements load balancing by scheduling task forces on a hierarchically organized network of processors. It involves extra overhead to maintain the hierarchy and the messages exchanged among different layers of the hierarchy add to the total communication costs. Furthermore, wave scheduling is not appropriate for all applications, since it deals tasks with sizes typical of the processes on a uniprocessor. However, any load balancing algorithm can easily be applied to the flat partitioned network without any extra overhead. The network is broken down into several small host groups. In this case, all the hosts in different host groups lie at the same level. The above mentioned approaches are illustrated in Fig.7.1 and Fig.7.2.

There are several factors to be considered at the time of partitioning a network. The two major ones include the size of the partitions or host groups and the criteria for grouping the hosts together. The size of the host groups can be fixed or changed dynamically in which case hosts can join or leave a group over a given time. However, the number of hosts in a group depends on the overall size of the network and on the criteria used to form the host groups. There are several criteria that can be used to form the host groups some of which include the geographical location of a processor, load on a processor and the transmission capacity of the links etc. The grouping of

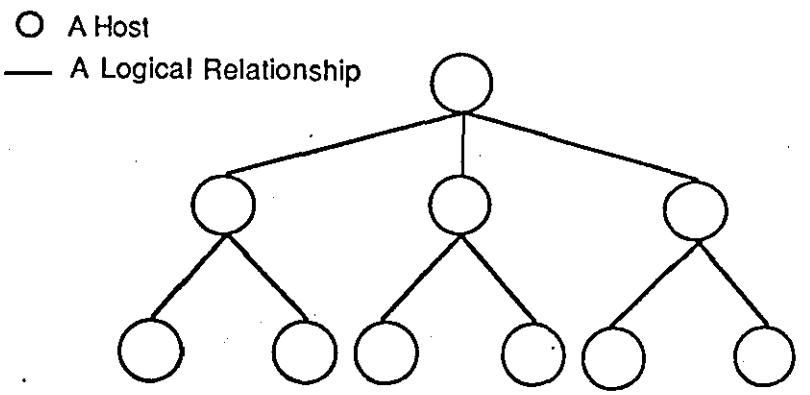

Figure 7.1: A Hierarchical Partitioning of the Distributed Computer System

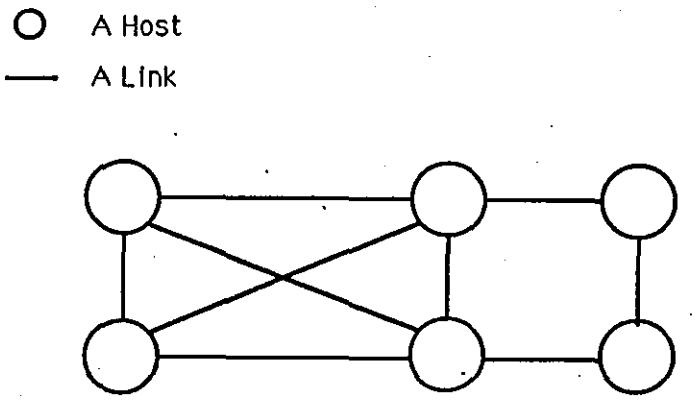

Figure 7.2: A Flat Partitioning of a Distributed Computer System

hosts on the basis of geographical location is very simple and partitions the network along the natural boundaries. All the hosts located within the same geographical region are the members of the same host group. The second method is to consider the load on each host for making the host groups. This requires an additional overhead of distributing the load information amongst all the hosts prior to partitioning of the network into various host groups. This information is used to rank the processors and form the host groups on the basis of these rankings.

Since the load on hosts changes dynamically, it might be necessary to repeat this process periodically. Another drawback is that the hosts grouped together on the basis of load might be located at large physical distances which might introduce large overheads in transferring tasks within the same host group. The final criteria of grouping the hosts based on the transmission link capacities tries to bring together hosts which are inter-connected with links of similar transmission capacities.

The work presented in this chapter is based on flat partitioning with geographical location as the host grouping criterea. In the next section, the idea of grouping hosts is further explained.

7.3 The Host Group Model

The idea of grouping hosts together to reduce the inter-processor communication costs is similar to that of clustering techniques used for task systems to cut down the inter-task communication costs. Several load balancing algorithms based on task clustering techniques have been proposed and studied in

the literature [cf. chapter 3]. However, a few researchers have concentrated on clustering the processors to improve the load balancing performance. The task clustering techniques are not very suitable for dynamic load balancing strategies due to the assumptions of a deterministic task system. Whereas, the host grouping techniques presented in this chapter can be easily adapted by any dynamic load balancing algorithm. No attempts are made to bring the tasks with high inter-task communications together on the same processor. By restricting the task transfers within the same host group, some reduction in communication costs are achieved.

The basic idea of a host group model was proposed as a way to support multicasts in an internet environment. Existing communication standards such as the OSI [1] reference model support message communication only to a single destination [Tanenbaum 1986]. This type of communication is called unicast. It is costly to unicast separate copies of the same message to a set of different destinations. Multicast is the delivery of a packet to some specified subset of possible destinations [Aguilar 1984]. In this study, the multicasts will not be treated in detail. The purpose of introducing multicasts was to familiarize with the message communication techniques that can be incorporated in conjunction with the host group model to improve the performance of message communications in the large area communication networks. [Deering 1988] defines two different types of host groups. In a *closed group*, only the members of the same group are allowed to communicate among each other, whereas in an *open group*, a sender need not be a member of the destination group. The grouping strategies proposed in this study can be implemented using one of these models.

[1] Open Standards Interface

As mentioned earlier, the work presented in this chapter is based on flat partitioning. The host groups are formed along the geographical boundaries. Each host in a group is called a member of the group. It is assumed that a host can join or leave any host group. A host can also change its membership from one host group to another dynamically [Deering 1988]. Host groups are formed at the time of initialization. Only the hosts within the same host group interact with each other to make the load balancing decisions. Each host in the host group stores a table which keeps the load information of all other hosts in that group. One of the hosts from each host group is designated as the group manager. All the group managers interact and exchange load information messages amongst each other. In this way, the group managers incur additional overhead of distributing and managing load information among themselves. The purpose of group managers is to provide a global load balancing.

7.4 Effectiveness Of Host Group Strategies For Load Balancing

In this section, a simple dynamic load balancing strategy is proposed and its effectiveness for several different host grouping techniques is studied in subsequent sections. The performance of these strategies is studied through simulations. The network is partitioned on the basis of the geographical location of the hosts. All the hosts lying within the same geographical area belong to the same host group. The current installations of local-area networks provide communication facilities that are several times faster than those of the long-haul networks. Therefore, it is assumed that the communication cost of

transferring a task within the host group is smaller as compared to the cost in transferring the task to a different host group.

The load balancing algorithm, considered in this study, captures the important characteristics of a majority of dynamic load balancing algorithms.

- It is assumed that the task arrivals and the task service times are unknown at the time of making load balancing decisions.
- The periodic broadcast mechanism is used to distribute the load information amongst the hosts.
- A host is categorized as heavily loaded or lightly loaded on the basis of load threshold. The load threshold is determined from the average load of the hosts over the entire distributed system.

The distributed system comprises of several hosts inter-connected in an arbitrary fashion. All the hosts are assumed to be identical. In the remaining section, host grouping strategies are described in detail.

7.4.1 Intra-Cluster Load Balancing Strategy

It is the simplest form of host grouping strategy proposed for load balancing. In this strategy, the host groups are formed on the basis of geographical location of hosts. The load balancing is performed locally among the hosts of each group. No attempt is made to provide the load balancing across the boundaries of the host groups. All the load information messages and other load balancing activities are limited to the scope of the host group.

This strategy breaks the network into several smaller network components which work independently from each other. Therefore, this strategy does not provide global distributed load balancing. However, this strategy provides some insight into the potential benefits from the partitioning techniques. It is expected that several heavily loaded hosts will find a suitable lightly loaded host within their own host group and may not need to transfer tasks to a remote host from some other group. Such a strategy will result in large reductions of the communication costs in load balancing. If there are large differences in the load levels amongst the host groups, this strategy will result in a poor global load balancing.

7.4.2 Inter-Cluster Load Balancing Strategy

The intra-cluster load balancing strategy fails to provide global load balancing for the distributed systems. This strategy is a modification of the intra-cluster load balancing strategy and tries to extend the load balancing activities across the entire network. The host group model of this strategy is shown in Fig.7.3. This strategy and the ones proposed in the following sections are fully distributed in the case of load balancing within the host groups, whereas these are partially centralized for the load balancing among the host groups. This centralized aspect is introduced by the fact that the inter-cluster load balancing decisions are made by the group managers. If a group manager fails then the host group will not be able to participate in the global load balancing. This strategy can be viewed at two levels.

- Load balancing within a host group.
- Load balancing among all the host groups.

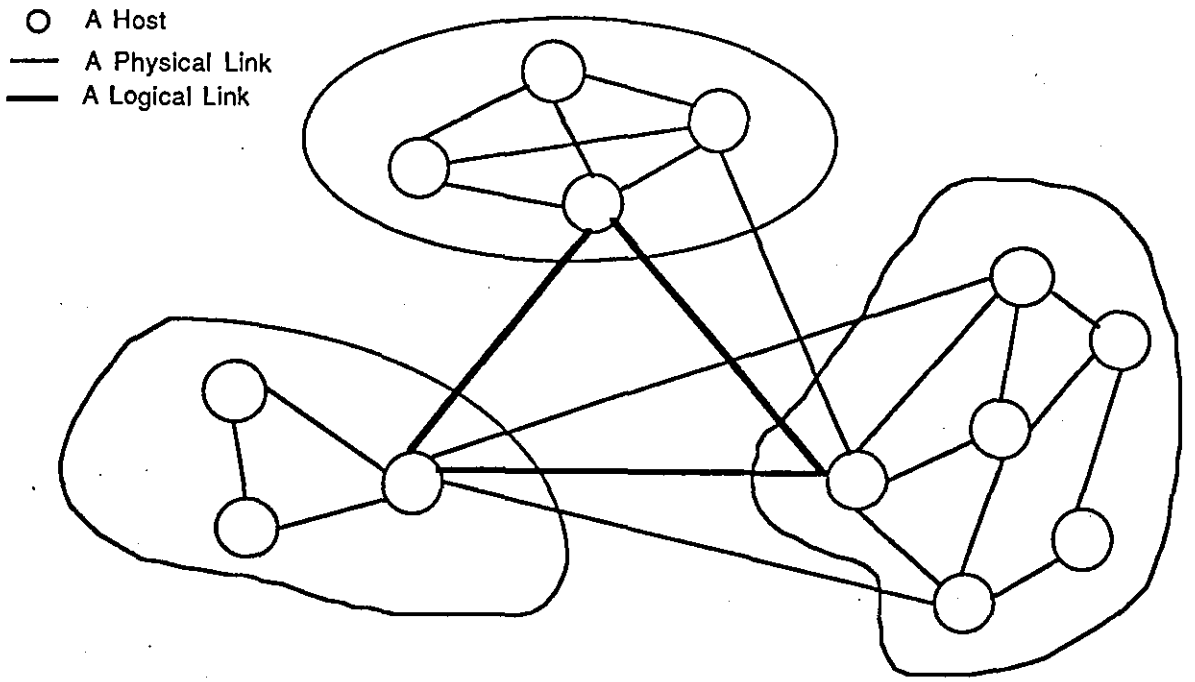

Figure 7.3: The Host Group Model For Inter-Cluster Load Balancing Strategy

Load balancing within a host group is similar to the one described for intra-cluster load balancing. As mentioned above, to achieve the global distributed load balancing, it is very important to provide sufficient interaction among the host groups. It is very likely that, at a certain time, all the hosts of a host group are busy whereas those of some other host group are sitting idle. If the load balancing is performed within these host groups, no significant performance gains will be achieved. Therefore, it is necessary to transfer some tasks from the heavily loaded to the lightly loaded host groups. The following modifications are introduced to perform the load balancing among the host groups.

- Each host group designates one of its hosts as the group manager.

- The load information is exchanged periodically among the group managers.
- Task transfers among the host groups are established through the group managers.

Several different criteria can be used to select a group manager. One criterion is to choose a host with the lightest load among all those included in the group. The reason to select a host with the lightest load is that the group manager has to carry out additional responsibilities and load balancing activities. However, the frequent changes in the host loads make this approach infeasible and rather costly. The overheads involved in such an approach may exceed the performance gains from load balancing. The second approach is to select a host that provides efficient and fast communication facilities. It is easy to select such a host at the time of system initialization. Furthermore, since all the tasks are transferred through group managers and load information is exchanged periodically among the group managers, this approach will provide better performance. The second approach is assumed in this study.

It is one of the responsibilities of the group manager to provide sufficient information about the load levels within a host group. Two different methods are suggested here which could be used to estimate the load levels of host groups. The first involves finding the average of host loads in the group periodically. This average load on each host group is distributed among all the host groups. This information is used to find the average load on the network. Load balancing among the host groups is performed by using the average network load as the load threshold to identify the heavily loaded

and the lightly loaded host groups. The second method involves simply using the instantaneous load on the group manager as an indicator of the average load on each host in the host group. This approach is based on the expectation that the load on each host in the host group is nearly the same as the average of load on all hosts in the host group. This depends on the performance of load balancing within the host group. If the performance of the load balancing algorithm within the host group is high, there is a good chance that the load on all hosts in the group will almost be identical. In this case, the load information is exchanged among all the group managers and the load balancing is performed on the basis of the load threshold which is determined from the average load on all group managers.

The load information is exchanged at two levels. First, there is load information exchange among all the hosts in the same group periodically. Second, there is an exchange of load information among the group managers. The frequency of the load updates can be adjusted to achieve required load balancing performance.

To balance the load over the entire network, tasks are transferred from the heavily loaded to the lightly loaded host groups. These task transfers are allowed only among the group managers and not to any other members of the host group. As already mentioned, it is assumed that there are high speed transmission facilities provided among the group managers. If several tasks are transferred from the group manager of a heavily loaded to the group manager of a lightly loaded host, the group manager of the former will become lightly loaded in its own group. Nevertheless, the local load balancing algorithm will transfer tasks from other hosts of the group to the

group manager, hence lowering the load level of the host group.

The inter-cluster load balancing will prove useful in the situations where there are large differences among the host group loads. For small differences in the loads, its performance is comparable to the intra-cluster load balancing. For small differences in the host group loads, it is beneficial to keep the frequency of load updates at very low levels among the group managers.

7.4.3 Membership-Exchange Strategy

The strategies introduced in the preceding sections are based on fixed geographical partitions of the network into the host groups and the members of a group remain in that group forever. This strategy and the one described in the forthcoming section are based on a slightly different concept of the host membership. In these strategies, hosts are allowed to change their memberships in an attempt to improve the load balancing performance.
The membership-exchange strategy is an improvement over the inter-cluster load balancing strategy. The network is partitioned and the host groups are formed in a way similar to the inter-cluster load balancing strategy. The following parameters are exchanged among the host groups during each information update interval.

- An instantaneous load value on each host for load balancing within a cluster.
- An instantaneous load value on each group manager for inter-cluster load balancing.
- An average load value estimated on each host group to make membership exchange decisions.

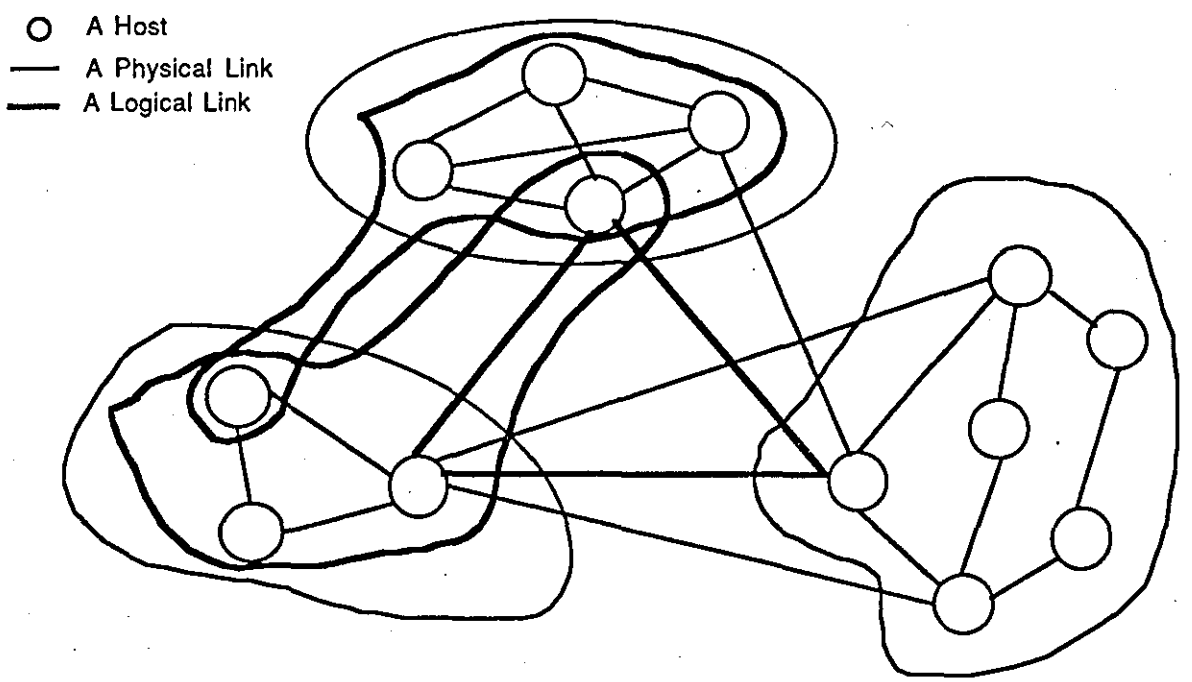

Figure 7.4: The Network Partitioning For The Membership-Exchange Strategy

In the membership exchange strategy, different host groups of the network try to exchange their members to improve the load balancing performance. Since the partitioning of the network is based on geographical boundaries, and as no consideration was given to the individual host loads, a high load imbalance is expected among the host group loads. The inter-cluster strategy will transfer several tasks among the host groups to achieve the desired load balancing performance.

In the membership-exchange strategy, the load on each host is continuously monitored to find the average load value during a specified interval. This average host load is used to determine the average load on each host group

and the average load of the entire network.

$$f(t_i) = q(t_i) \cdot \overline{x}$$

$$Average\ Load\ = E[f(t)] = \sum_{i=1}^{n} f(t_i) \cdot \Delta t/t$$

where,

$q(t)$ =Queue-length as a function of time.

$\overline{x}$=Mean task execution time.

and, $\Delta t = t_i - t_{i-1}$

These two measures are used to categorize the host groups as heavily or lightly loaded. Using this information, pairs of most heavily and most lightly loaded host groups are formed. The next step involves choosing a host each with the highest and lowest average loads from the heavily loaded and the lightly loaded host groups respectively. These two hosts exchange their memberships and this process continues until the difference among the host groups is very small.

This strategy tries to minimize the transfers of tasks among the host groups with membership exchange mechanism. However, it violates the geographical partitioning of the host groups. A host group may have one or more host members from a distant geographical region. This strategy tries to bring some hosts from heavily loaded host groups into lightly loaded host groups and vice versa. In this way the hosts with large differences in their loads are grouped together in a single host group. As a consequence, the task transfers are shifted from the inter-cluster level to the intra-cluster level. The information necessary to improve the load balancing is propagated relatively quicker than before. Fig.7.4 illustrates the membership-exchange strategy.

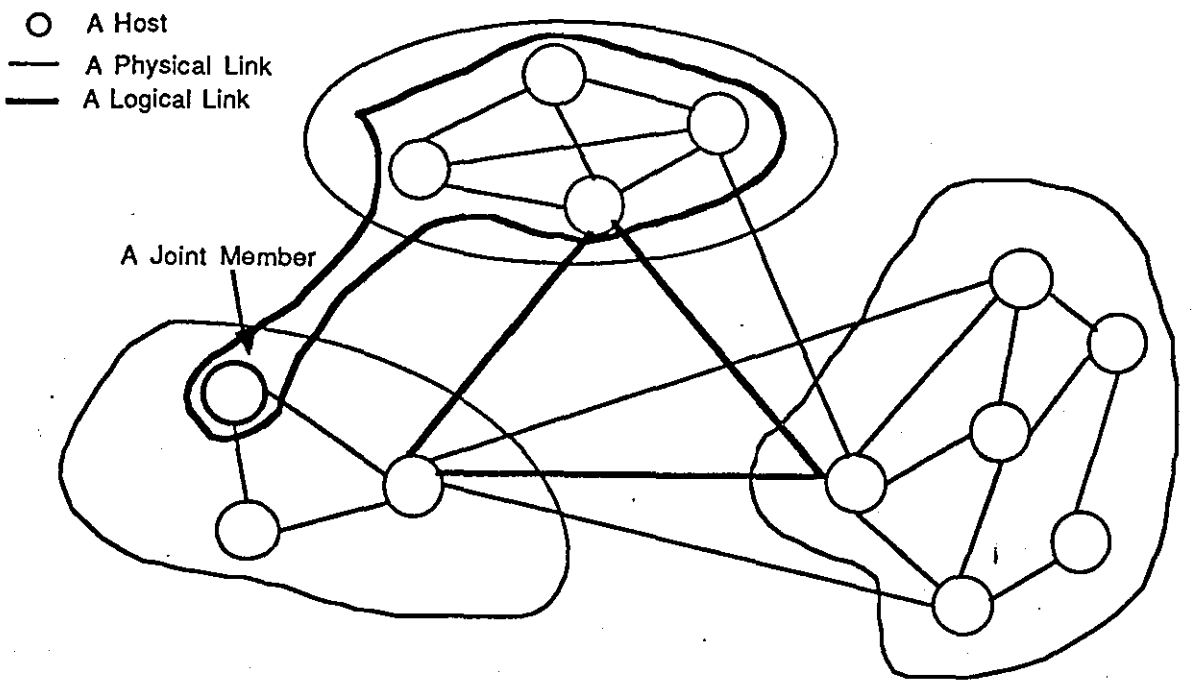

Figure 7.5: Network Partitioning For The Joint Membership Strategy

7.4.4 The Joint Membership Strategy

This strategy tries to improve the load balancing performance by extending its membership to the hosts of other host groups. A host can be a member of more than one host group. It is similar to a closed group model proposed by [Deering 1988]. In this case, the size of a host group increases with the number of joint memberships. This strategy provides some degree of load sharing between a heavily and a lightly loaded host group. In the present study, a lightly loaded host, which is a member of a lightly loaded host group, is offered joint membership by a heavily loaded host group.

The parameters exchanged during the load update period are the same as those described above for the membership-exchange policy. The average

load on the host groups is used to determine both the heavily and the lightly loaded host groups. After the identification of both the heavily and the lightly loaded host groups, a suitable candidate from the latter group is offered a membership in a heavily loaded host group. The term load is used for the average load on the host and not for the instantaneous load. The choice of a host for the joint membership is for the one which has the least load in a lightly loaded host group. In this way, this least loaded host, belonging to a lightly loaded host group, offers its services to share the load of a heavily loaded host group.

If a host group changes its status from a heavily loaded to that of a lightly loaded one, it cancels all the joint memberships of its hosts. Fig.7.5 illustrates the host group structure for the joint membership strategy.

7.5 Simulation Model And Performance Evaluation

The performance of the network partitioning strategies was evaluated through simulations. The simulation model described in chapter 4 was used for the evaluation of performance. Each host of the distributed computer system was modelled as an M/M/1 queueing system. The task service times were generated from an exponential distribution with a different mean value for each host. The average of these mean values was 8. The distributed system comprised of 16 hosts. Initially, the network was divided into four fixed sized partitions called host groups with each host group consisting of four hosts. One of the hosts in each such group was designated as the group manager. The distributed computer system used for simulations is illustrated in Fig. 7.6.

It was assumed that the network is completely connected. To simulate the geographical partitioning of the network, the transfer costs associated with the tasks transferred within a host group were lower than the transfer costs for the tasks transferred among the host groups via group managers. The transfer costs used in the simulations were assigned fixed values independent of the size of the tasks transferred. The transfer costs used for the tasks transferred within the host groups, and for the tasks transferred among the host groups were 1 (nearly, 12% of the mean task service time) and 3 (36% of the mean task service time) respectively.

The following assumptions were made about the distributed system.

- All the hosts in the distributed system were assumed to be identical. It is assumed that the hosts are loosely coupled in a large scale communication network environment.
- It was assumed that the host group model proposed in [Deering 1985] can be adapted for load balancing strategies described here.
- The inter-task or the inter-processor communications were not considered, however, the inter-processor communications are partly reflected in the transfer costs mentioned next.
- The transfer costs were chosen to reflect the physical proximity of the hosts. The smaller transfer costs were associated with the tasks transferred within a local area network as compared with those in a wide area network.
- The transfer of tasks is based on the task placement model. A task already in execution is non-transferrable.

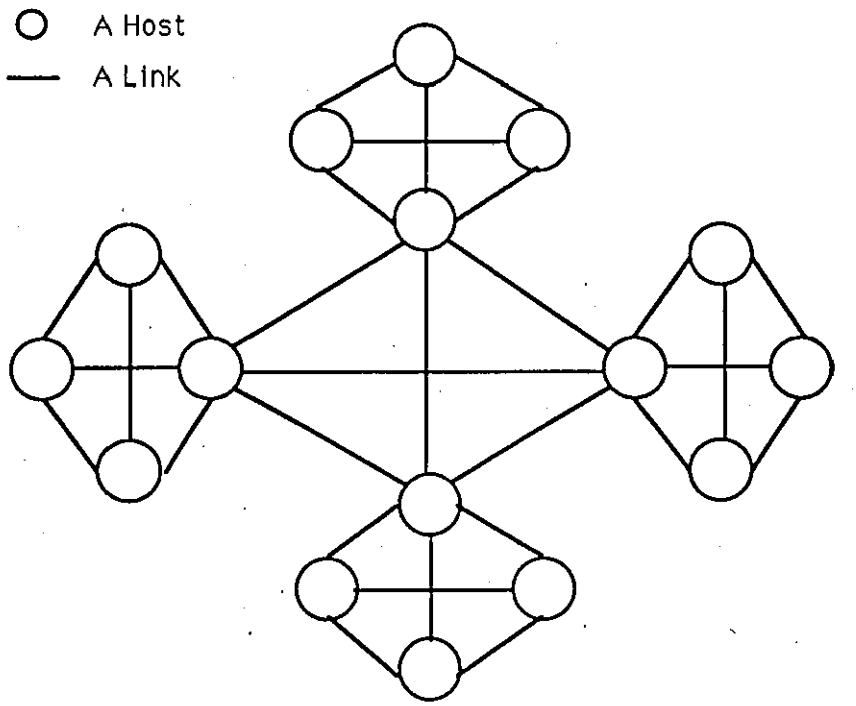

Figure 7.6: The Distributed System Model Used For Simulations

The load balancing algorithm used to measure the performance of host grouping strategies is based on a load threshold value which is determined from the average of load on the hosts participating in the load balancing. The load information is exchanged periodically among the participating hosts. All the hosts compare their load with the load threshold value to determine their status. Separate load update intervals were used for load balancing within and amongst the host groups.

The simulations were carried out for each type of grouping strategies. The performance metrics used for the evaluation of these strategies included the average response time of a task and the load imbalance among the hosts of the distributed computer system [cf. chapter 4]. The measurements were

obtained from the average of several samples which were obtained from each simulation run. All the recorded values were within a confidence interval of 95 percent. The performance of the strategies was compared amongst each other and with a 'no load balancing' case. It should be noted here that there are no communication costs involved in the 'no load balancing' case, as all the tasks execute on the host of their origin.

The average response times were plotted against the increasing host loads. The increase in the host loads is obtained by increasing the task arrival rate on each host for the same task service times. The load imbalance was recorded after an interval of 30 time units while the system was running in a steady-state.

Fig.7.7 shows the average response time against increasing arrival rates for the intra-cluster load balancing strategy. In this case, the load balancing was performed only within but not among the host groups. Its performance is compared with the no load balancing case. Since all of the host groups are isolated from each other and no task transfers can take place between the hosts of two different groups, the chances of finding a lightly loaded host for a heavily loaded host are limited in this case. Despite this limitation, there is a large improvement in the average response time of a task for the intra-cluster load balancing compared with that of the no load balancing. Therefore, the result obtained suggests that for very large distributed computer systems, in order to avoid the complexity of managing a large number of resources and to minimize the communication costs, a better utilization of the resources can be achieved by partitioning the system into smaller logical network components for the purposes of load balancing. If some of the host groups are very

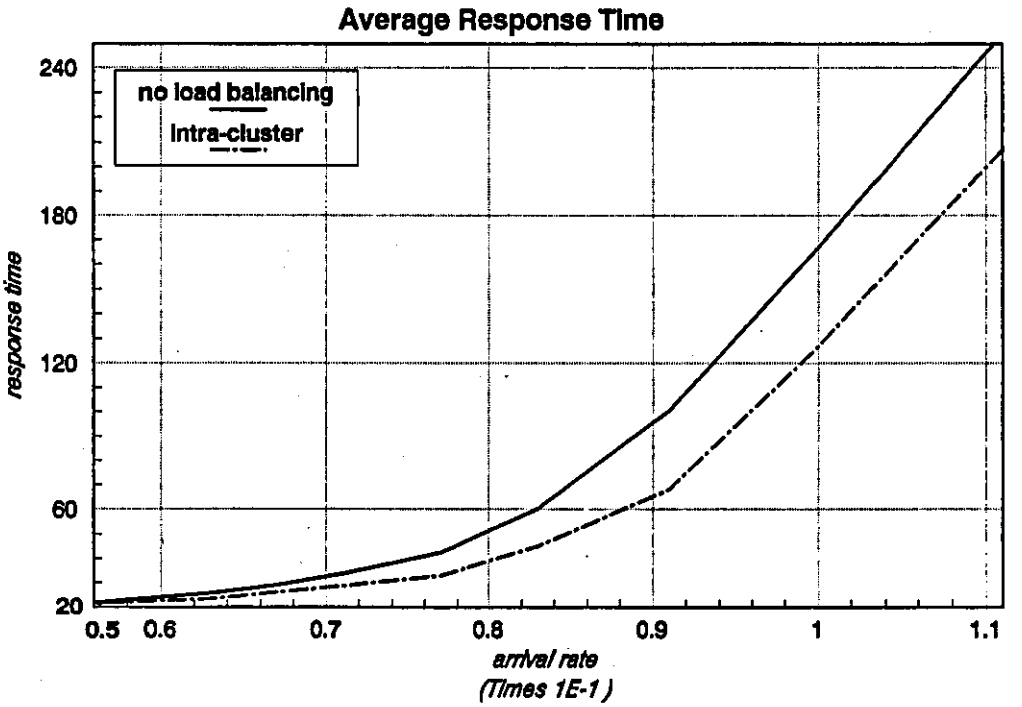

Fig.7.7: Average Response Time Comparison of Intra-Cluster Strategy With 'No Load Balancing'.

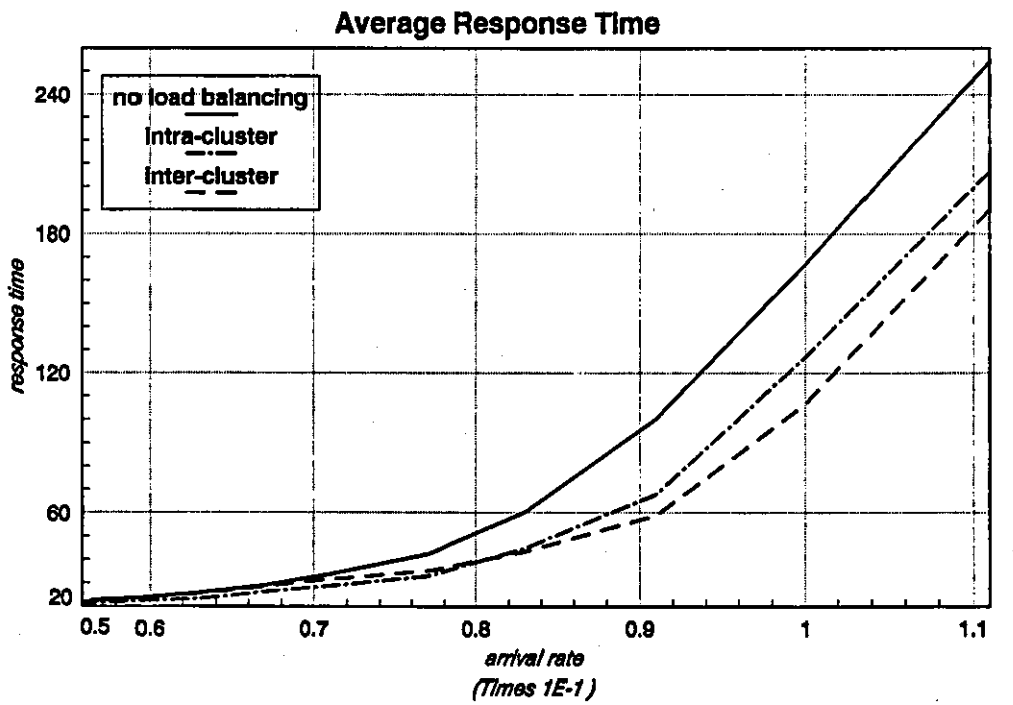

Fig.7.8: Average Response Time Comparison of Inter-Cluster and Intra-Cluster Strategies With 'No Load Balancing'.

heavily loaded, whereas others are very lightly loaded, it is expected that this strategy will not prove very effective.

An improvement in the average response times was observed by providing load balancing among the clusters in addition to the intra-cluster load balancing. Fig.7.8 shows the average response times for the inter-cluster load balancing strategy. Despite higher costs for the task transfers amongst the host groups, a noticeable improvement in the average response times was achieved for higher load values. It indicates that the delays experienced by tasks in the queues of a host group are larger than the transfer costs in case of a remote execution. However, for light loads it was observed that the intra-cluster strategy performs slightly better than the inter-cluster strategy. This clearly shows that the loads are not heavy enough to support task-transfers amongst the host groups. These results are in agreement with the results obtained in [Mirchandaney 1989] for high task transfer costs and light loads. However, as the arrival rate increases and the difference in the loads of hosts with dissimilar service rates becomes pronounced, the performance improvement introduced by the inter-cluster strategy can be noticed.

Also, the load of a host manager did not provide a good estimate of the average load of the host group due to relatively infrequent periodic broadcasts. By performing the intra-cluster load balancing frequently, better estimates of the average load of the host group will be reflected in the group managers' load. In Fig.7.8, the update interval used for exchanging load information among the group managers was larger than the update interval used for hosts within individual host groups. While the former was 400 time units, the interval in the case of the latter was 200 time units only. The degree of

interaction for load balancing among the host groups can be controlled by increasing the length of the update interval used among the group managers. A further improvement in the response times will result for lower task transfer costs amongst the host groups. In these simulations, the transfer costs used among the host groups were three times higher than those assumed for the task transfers within the host groups.

If the difference of the load levels amongst the host groups is not very large, this strategy will perform similar to the intra-cluster load balancing strategy. Later, it was thought that the inter-cluster load balancing strategy can be further improved by modifying the host grouping criteria. In this study, it was not investigated how the load balancing performance will vary subsequent to any changes in the size of the host groups. However, it is expected that an increase in the size of a host group will improve load balancing as the probability of finding a suitable host to transfer the load within the host group increases. The next strategy evaluated does not change the host group size, but it does effect the group memberships.

Fig.7.9 and Fig.7.10 illustrate the performance of mem bership-exchange strategy and compare it with the intra-cluster and inter-cluster load balancing strategies. It outperforms the two startegies by achieving major reductions in the average task response times. These reductions in the average response times were achieved by grouping together such hosts that contribute to an effective load balancing. In the load balancing process, the hosts of the network are categorized in three different ways. These comprise heavily, lightly and moderately loaded hosts. As those with the moderate loads do not actively participate in the load balancing process, the tasks can only be

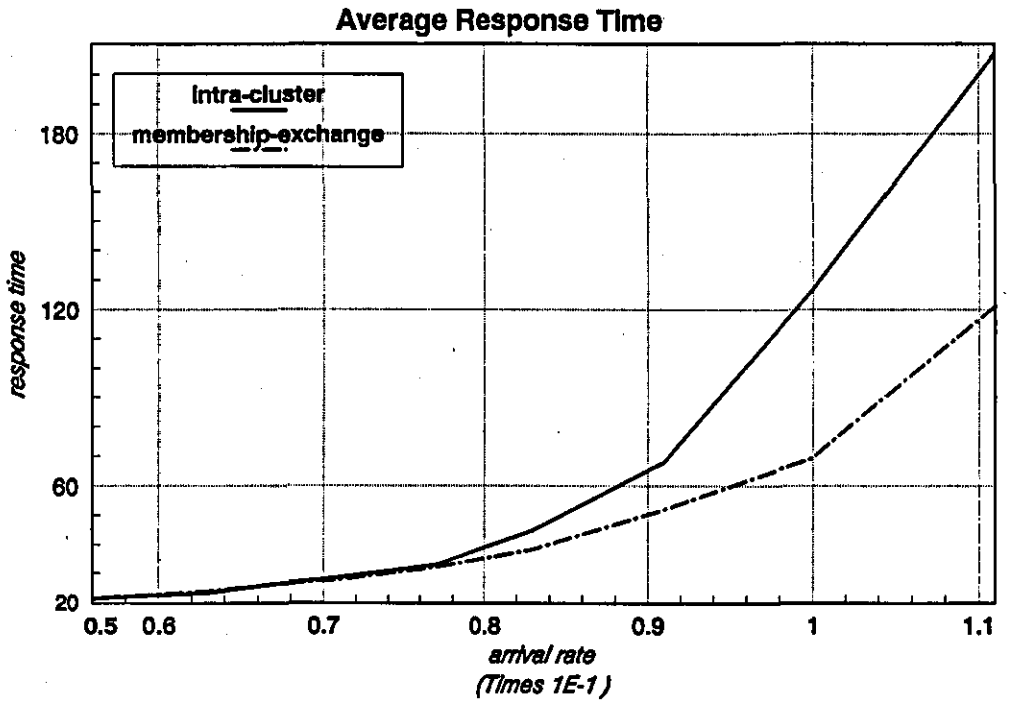

Fig.7.9: Average Response Time Comparison of Membership-Exchange Strategy With Intra-Cluster Strategy

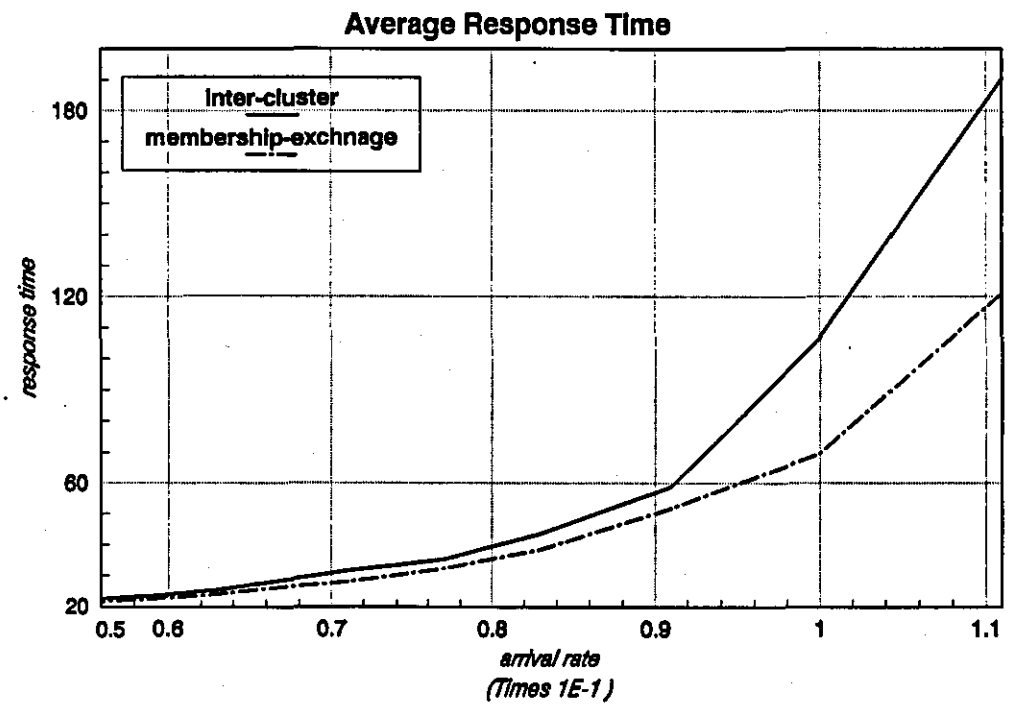

Fig.7.10: Average Response Time Comparison of Membership-Exchange With Inter-Cluster Strategy

transferred from a heavily loaded to a lightly loaded host. In the case of the inter-cluster load balancing, the difference among the host loads within a single group may not be sufficient to provide an effective load balancing. However, the difference among the host groups may be large and this information needs to be distributed across the entire network so that the task transfers could be arranged among the hosts of different host groups. The membership-exchange strategy improves its performance by bringing together the hosts belonging to different groups into a single group for a more frequent information exchange and direct transfers between the two hosts involved in the process of load balancing. Also, it is expected that the average load value obtained for a host group provides a better estimate for load balancing than the average load value obtained for the entire network to balance the load among the host groups. Therefore, better estimates are obtained for the amount of tasks that need to be transferred between the hosts when the two hosts, eligible for load balancing, are in the same host group. The aging of information as it travels through the system and the delays in making the transfer decisions and causing the required transfers support such a strategy. Furthermore, it was observed that the intra-cluster strategy performs better than the inter-cluster strategy for light load conditions. As shown in both figures, the membership-exchange strategy performs well for light loads as well. It indicates that even the slightest differences in the loads, under light load conditions, are detected and adjusted for improvements. The task transfer costs for the tasks transferred within the groups and those transferred among the host groups were similar to the ones used for the inter-cluster load balancing strategy.

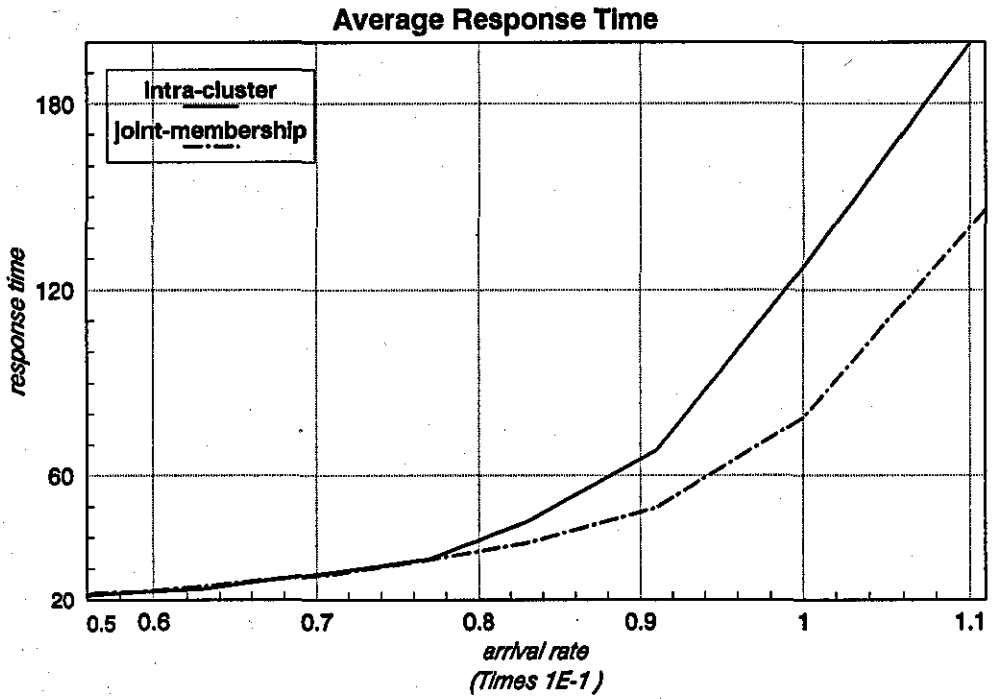

Fig.7.11: Average Response Time Comparison of Joint-Membership Strategy With Intra-Cluster Strategy

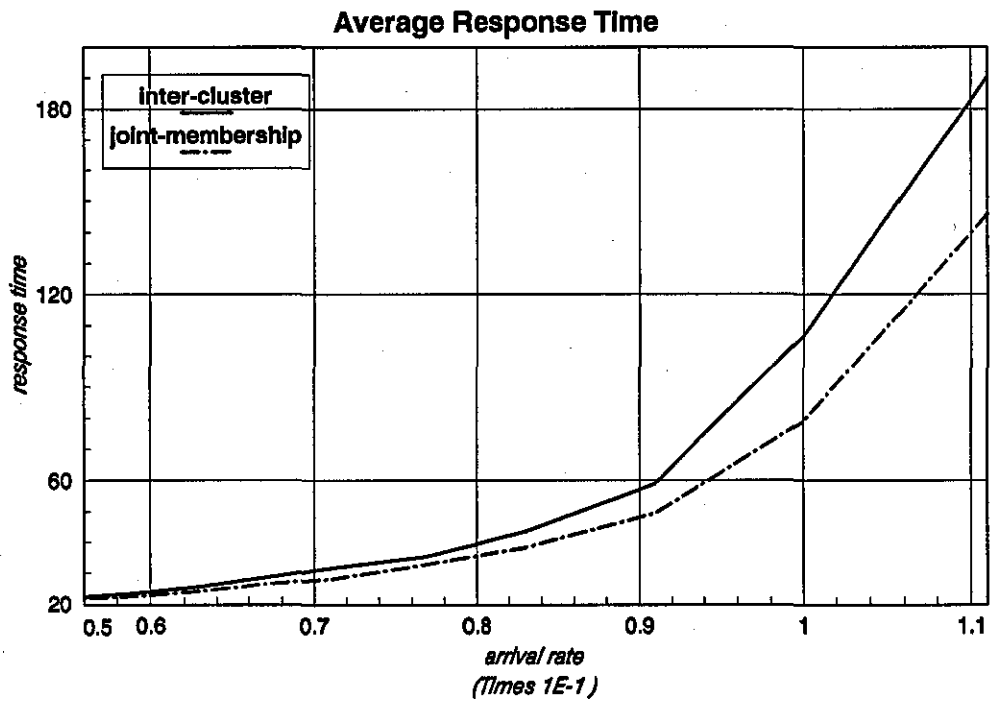

Fig.7.12: Average Response Time Comparison of Joint-Membership Strategy With Inter-Cluster Strategy

For heavy system loads, significantly improved response times were achieved. All the efficiency stems from bringing together the hosts suitable for load balancing into the same group. The cost of such a strategy depends on the frequency of the host exchanges performed amongst the host groups.

Fig.7.11 and Fig.7.12 show the performance of the joint membership strategy for load balancing. Its performance is compared with intra-cluster and inter-cluster load balancing strategies. It is recalled that the joint membership strategy tries to improve load sharing in a distributed computer system by extending the membership of lightly loaded hosts from lightly loaded host groups to heavily loaded host groups. If all the host groups are moderately loaded, this strategy will operate like the inter-cluster strategy and will show an improvement in the average response times for all load conditions. A host working as a joint member provides a bridge between the two host groups and tries to bring the average load on these host groups to the same level. In this way, instead of providing a single group manager as an interface amongst the host groups, several other hosts participate in sharing the load. Also, a joint member extends the size of a host group which increases the availability of the resource.

Finally, Fig.7.13 illustrates the average response time curves for all the host grouping strategies. As expected, all these strategies improve the performance of a distributed system by providing adequate load balancing as compared to the 'no load balancing' case. Among these four strategies, while the membership-exchange strategy shows the best, the intra-cluster strategy gives the worst performance in comparison to others. It is important to mention here that the performance of all these strategies is largely dependent

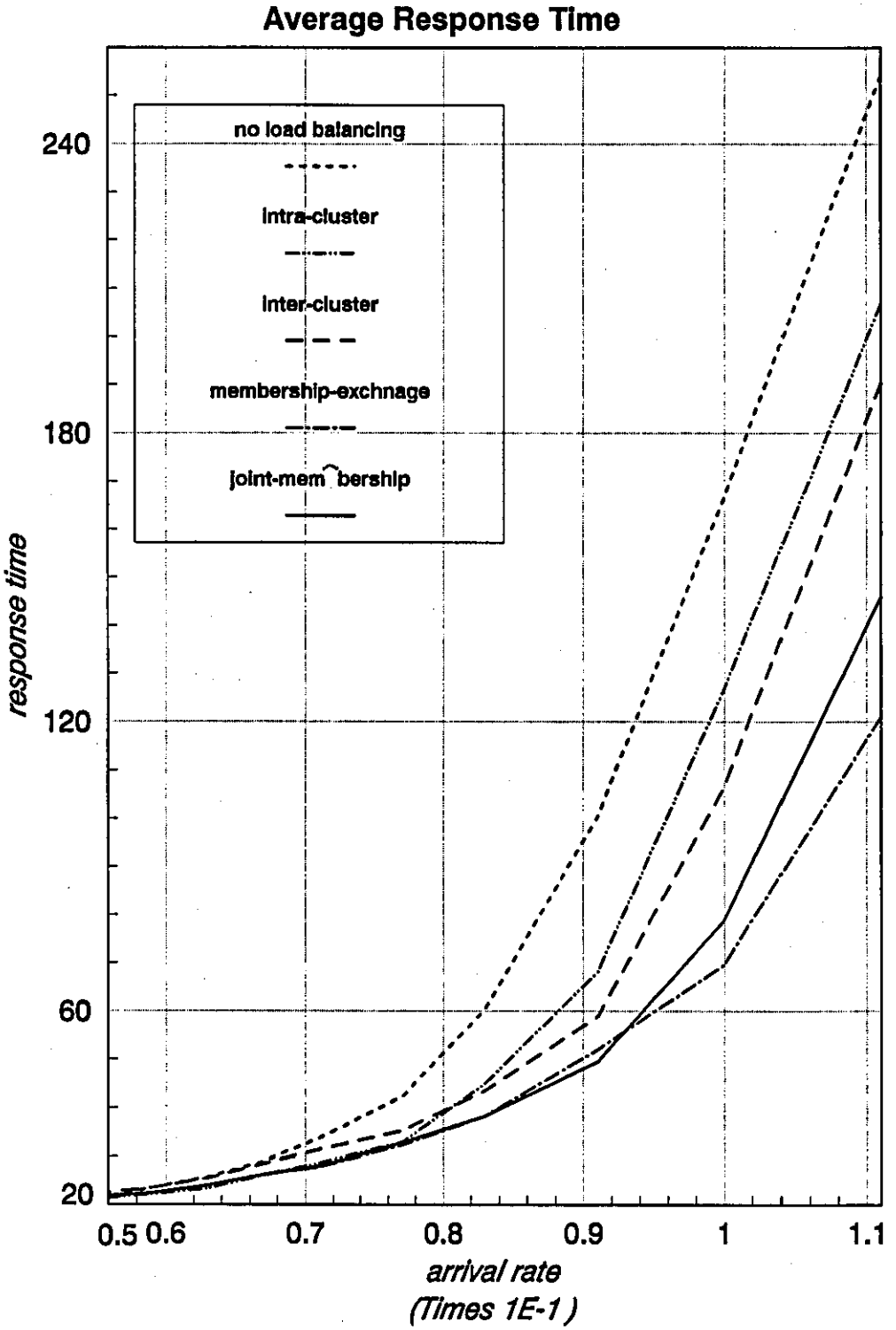

Fig.7.13: Average Response Time Comparison of All Strategies.

on the characteristics of the algorithm used for the dynamic load balancing. The host grouping strategies are proposed for the class of algorithms where hosts of a distributed system frequently exchange their system state information to perform the load balancing. The improved task response times for the membership-exchange strategy suggest that the host loads be used as the host grouping criteria. Since the host loads change frequently, a frequent regrouping of host groups will be required to meet this criteria. Instead, the membership-exchange strategy starts with geographically partitioned groups and performs load balancing by grouping together the hosts with large load differences from different groups. In this way, the overhead due to periodic regrouping of all hosts on the basis of load is avoided and only a few hosts are exchanged for the sake of load balancing performance improvement.

Finally, Fig.7.14 and Fig.7.15 are shown to illustrate the load imbalance among all the hosts of the distributed system for different host grouping strategies. These results were obtained for moderate system loads and update interval lengths of 200 and 400 for load balancing within and amongst the host groups respectively. In Fig.7.14, low load imbalance is achieved for both the intra-cluster and inter-cluster strategies as compared to 'no load' balancing. However, slightly better load balancing is achieved for intra-cluster as compared to the inter-cluster strategy. Despite of this imbalance in loads, better response times were achieved for inter-cluster strategy for moderate to heavy loads. For the given cost ratio of 1 to 3 for task transfers within and amongst the host groups, these results suggest that intra-cluster strategy should be used for light to low moderate loads and inter-cluster strategy to be used for moderate to high loads.

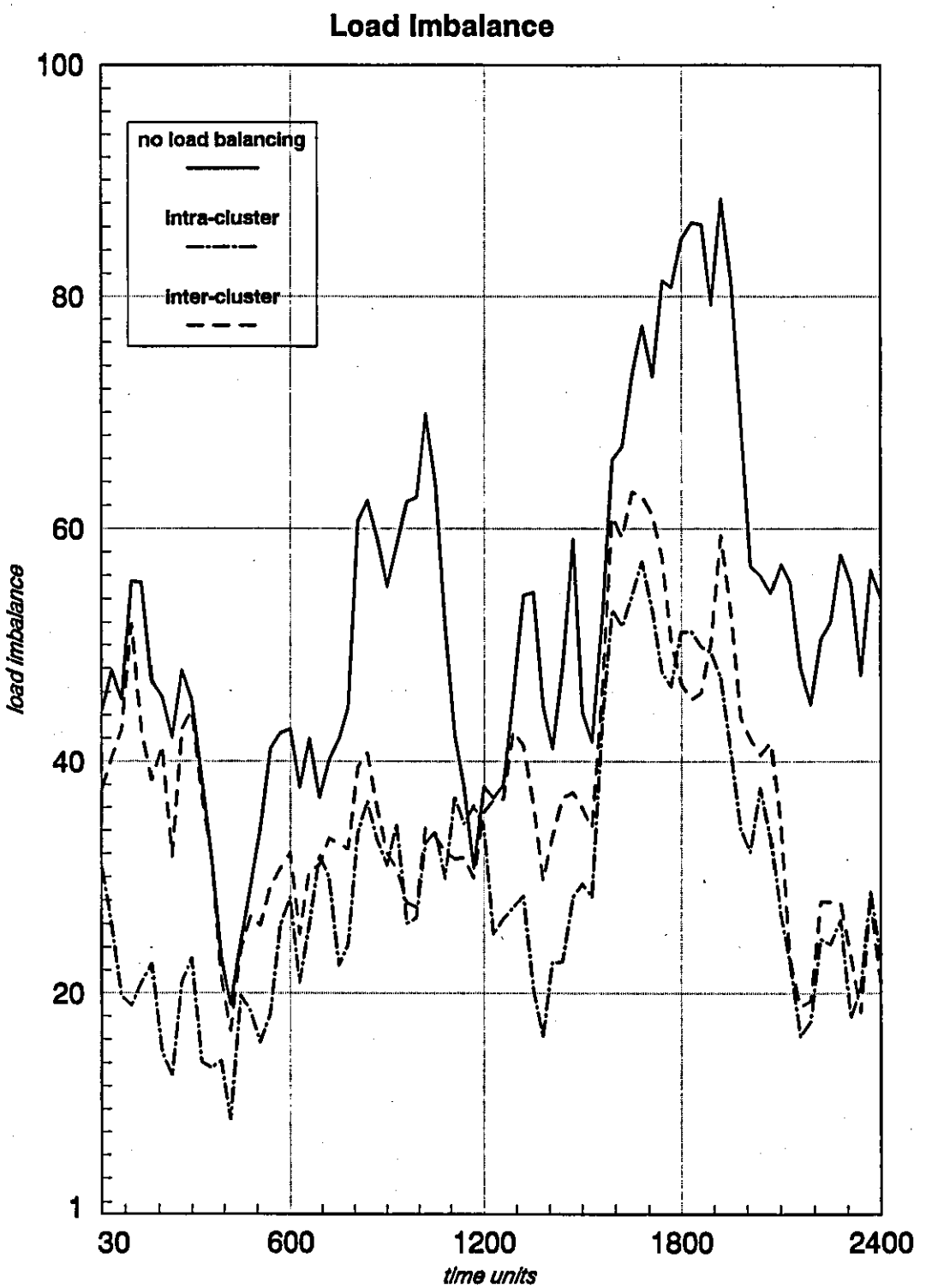

Fig.7.14: Load Imbalance Comparison of Intra-cluster, Inter-cluster strategies and 'noload balancing'.

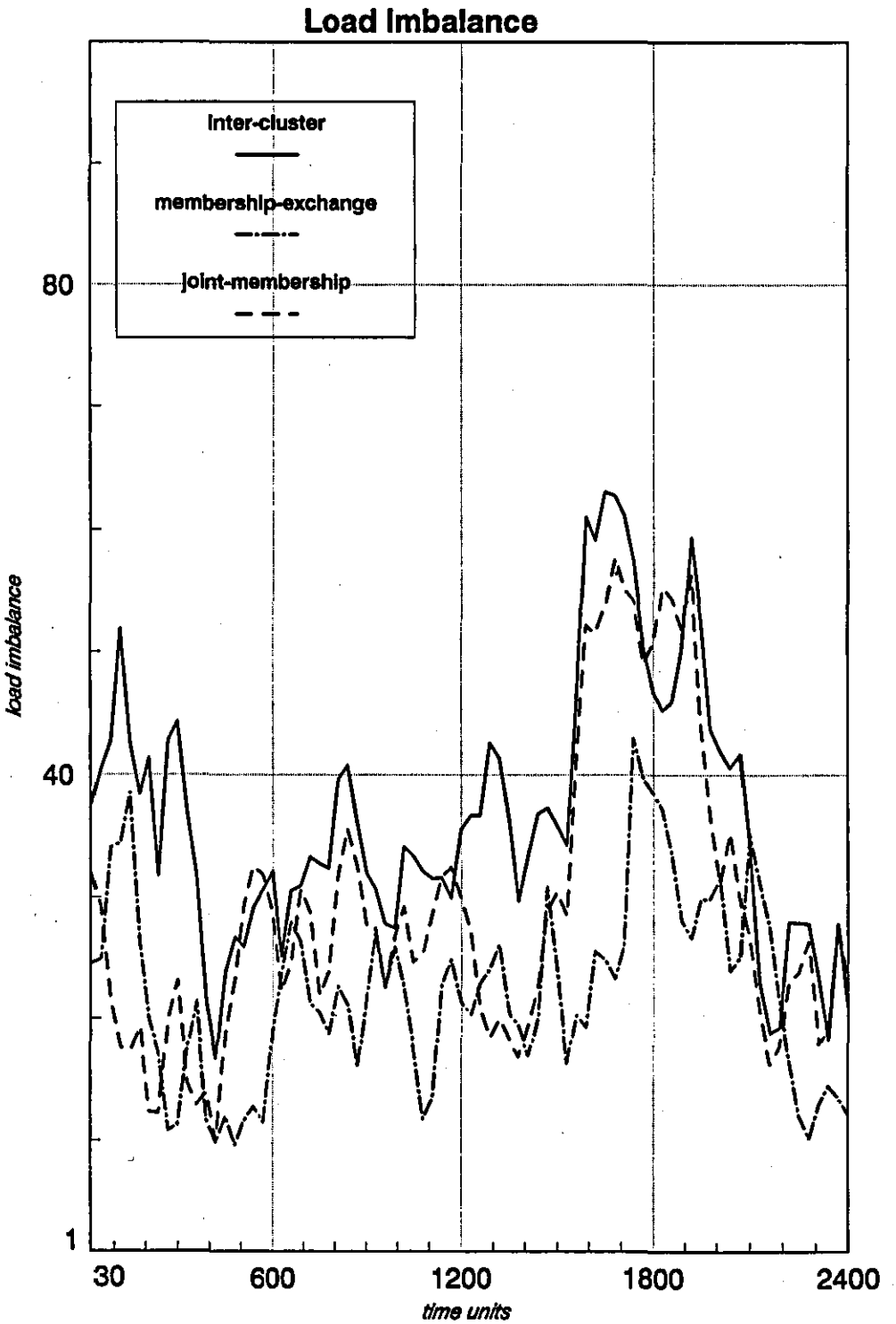

Fig.7.15: Load Imbalance comparison of Joint-Membership, Membership-Exchange and Inter-Cluster Strategies.

Fig.7.15 compares the load imbalance of inter-cluster, membership-exchange and joint membership strategies. It appears that the difference in load imbalance for inter-cluster and joint membership strategies is small while the difference between these two and the exchange-membership strategy is significant. The membership-exchange strategy dynamically changes the host group memberships and brings together the hosts with large differences in their loads. In this way, the information is quickly propagated and load estimates are more accurate as well as the task transfers are established immediately leading to an overall improved load balancing performance. The small values of load imbalance of membership-exchange and joint membership strategies as compared to inter-cluster strategy indicate that better load estimates are obtained for hosts within a host group resulting in accurate load balancing decisions. When the two hosts involved in load balancing are separated by long distance, by the time the load information is propagated and the required task transfers are implemented, the load on these hosts may change significantly resulting in poor load balancing.

Chapter 8

CONCLUSIONS AND FUTURE DIRECTIONS

8.1 Conclusions

An increasingly large amount of work has been performed in the area of static and dynamic load balancing. The basic intent of this study was to consider those important factors that influence the performance of load balancing in a distributed computer system and to propose strategies for improving the performance of such a system.

To study the performance of static and dynamic load balancing algorithms, a simulation model was designed. The simulation model described in chapter 4 provides all the basic routines which are important and are useful in the evaluation of load balancing heuristics.

A static task allocation heuristic is presented in chapter 5. It attempts to deal with two central issues of static task allocation and scheduling, i.e.,

minimization of inter-task communication costs and the load imbalance. The study performed on a task system with high inter-task communication costs indicates that it is very important to have a compromise between maximizing the parallelism and minimizing the communications. Also, for such systems with high inter-task communication costs, The Precedence Constrained scheduling heuristic described in chapter 5 gives a good performance. Such a heuristic works well for a wide range of computation to communication ratios and, therefore, can be utilized in a large number of applications. The results suggest that an appropriate choice of task allocation heuristics, that helps minimize the communication costs and set an upper threshold for the load on each processor in order to avoid the load imbalance, can provide improved performances.

As static task allocation heuristics which are available at the present time are not highly practical and have limited applications, a more realistic model of task systems is considered in chapter 6. Here, several quasi-dynamic and dynamic load balancing algorithms are considered. In this case, the inter-task communications are not considered due to the fact that the complete jobs are transferred among the hosts. Nevertheless, the inter-processor communications are central to the operation of these algorithms. The inter-processor communication costs depend on the information policy used to distribute the load information and the number of tasks transferred. A periodic broadcast is used in the study which distributes the load information amongst hosts of the distributed system but with an exception of threshold-window algorithm where load information is broadcast periodically from lightly loaded hosts only.

The results show that the simpler algorithms such as those of Random and Threshold-Window can achieve good load balancing performances under certain conditions. If a comprehensive policy is used for distributing the load information and any changes in the host loads are propagated immediately, the performance of any simple load balancing algorithm is comparable to a more complex one. First, comparing the Random and Threshold-Window algorithms, the former that used a better load update policy gave better performance than the latter for update intervals above 200 time units. In contrast, when period lengths in the order of two to three times the mean task service time were used for the threshold-window algorithm, performances better than the Random algorithm were achieved. This suggests that the frequency of load information exchanges can only be determined from the characteristics of the load balancing algorithm. A more frequent information exchange will only introduce extra overheads on the communication subnet whereas an out-of-date information will result in incorrect load balancing decisions, hence degrading the performance of distributed systems.

When more complex algorithms based on task-transfer probabilities were used for dynamic load balancing, better performances were achieved for Algorithms *II* and *III* but Algorithm *I* performed poorly. In this case, the difference in the performance of these algorithms was caused by the choice of respective load indices which were used for estimating the load on each host. The choice of a load index reflects the quality of information exchanged amongst the hosts. These algorithms transfer the newly arrived tasks dynamically which make use of the transfer probabilities which are determined from the load estimates obtained from the previous interval. The load estimates obtained from service and arrival rate change quickly and do not reflect any

loads accumulated over the previous intervals, therefore resulting in poor load balancing for longer load update interval lengths. In contrast to this, queue lengths tend to change slowly and provide better load estimates for load balancing. Algorithm *III* gives an improved load balancing performance for a wide range of interval lengths.

It is also shown that when communication delays are significant, improved performances for such algorithms can be achieved by properly reflecting the interprocessor communication costs in determining the task-transfer probabilities. These improvements in the average task response times are achieved at the cost of slight increase in the load imbalance.

One of the major issues concerning the efficiency and effectiveness of dynamic load balancing algorithms is their scalability. All distributed systems are expanded over time for achieving increased availability and reliability of resources. As the size of the distributed system increases, the overheads of a load balancing algorithm may increase resulting in a poor scalability. In chapter 7, several strategies are presented for network partitioning to limit the scope of load balancing algorithms to such host groups which each contains only a small number of hosts. Initially, these host groups are formed on the basis of geographical partitioning of the network. One of the hosts is designated as manager in each host group. A certain degree of interaction is provided amongst the managers to implement global load balancing. Major reductions in communication overhead are achieved by limiting the exchange of load update messages within smaller groups of hosts while restricting the transfer of tasks to long distance remote hosts which involve high communication costs. The results indicate improved load balancing performances for

strategies that modify host group memberships dynamically with changing load conditions on hosts. The change in group memberships should be limited to minimize its related overheads. This study suggests that when load balancing is provided within the host groups only and with a minimum of interaction among these host groups, the performance achieved is comparable to that of any other more complex strategy for light load conditions. For heavy load conditions, however, strategies that work on dynamically changing host memberships with changing load conditions show large improvements in the load balancing performance.

8.2 Suggestions For Further Research

The aim of this study was to point out the problems faced with static and dynamic load balancing strategies and to provide efficient solutions to these problems. Like any other research study, many questions remain unanswered and require further attention.

When considering static task allocation strategies, it is assumed that the information relating the task system is available in the form of a precedence graph. Also, simple FIFO scheduling was used to execute the tasks allocated to a processor. Instead, it will be interesting to study the performance of Precedence Constrained scheduling heuristic when tasks with low communication costs with their successors are executed first. This ordering of tasks may result in minimizing any possible idle times. Furthermore, a precedence graph does not represent two simultaneously inter-communicating tasks. These tasks must be scheduled simultaneously on two different processors. A large number of applications consist of such tasks and need to

be represented in the task system. A more realistic task system model is considered in dynamic load balancing environment discussed below.

In evaluating the proposed dynamic load balancing algorithms in chapter 6, a fixed set of arrival and service parameters is chosen. These parameters reflect moderate loads in distributed systems with small variations in arrival and service times. It will be interesting to study the behaviour of these algorithms for light and heavy load conditions in a distributed computer system. Also, larger differences in arrival and service times may be used to study the pronounced effects of load balancing.

Although an attempt is made to study the communication overhead of these algorithms by comparing the percentage task transfers, in future, it is recommended to consider effects of transfer delays and the inter-processor communications to measure the communication overhead of such algorithms. The transfer delays should be derived relative to the task sizes.

Finally, host grouping strategies are proposed to reduce the amount of high communication overheads incurred by load balancing in large distributed computer systems. As the results indicate, significant improvements in load balancing performance can be achieved with high reductions in communication costs by restricting the task transfers and load update messages to nearby hosts. Due to the time constraint, very limited results are obtained and thus require further research to fully explore these strategies. In this study, average response time and load imbalance measures are obtained for light, moderate and heavy load conditions. In future, it will be interesting to study the amount of reduction in communication costs for each strategy.

Also, the effects of changing the size of the network and the size of the host groups on the performance of the host grouping strategies would be a good topic for further investigation.

References

[Adam et. al. 1974] T. L. Adam et. al. *A Comparison of List Schedules for Parallel Processing Systems,* Communication of the ACM, vol.17, no.12, December 1974.

[Aguiler 1984] L. Aguiler. *Datagram Routing for Internet Multicasting,* Communications of ACM, pp.58-63, September 1984.

[Andre et. al.1988] F. Andre et. al. *Synchronization of Parallel Programs,* Published by North Oxford Academic, 1988.

[Angouras 1991] G. N. Angouras. *Scheduling of Parallel Programs on Multi-Programmed Parallel Processor Systems,* Technical Report; CSRD Report 1160, University of Illinois at Urbana-Champaign, January 1991.

[Barak and Shiloh 1985] A. Barak and A. Shiloh. *A Distributed 'Load-balancing Policy for a Multicomputer,* Software-Practice and Experience, vol. 15, no. 9, pp. 901-913, September 1985.

[Basu et. al. 1989] A. Basu et. al. *Performance of Loosely Synchronous Algorithms on a Message Passing Multiprocessor,* Proc. Indo-US

Workshop on Spectral Analysis in One and Two Dimensions, New Delhi, November 1989.

[Baumgartner and Wah 1988] K. M. Baumgartner and B.W. Wah. *A Global Load Balancing Strategy for a Distributed Computer System,* IEEE Technical Report, Dept. of Electrical and Computer Engineering and the Co-ordinated Sciences, University of Illinois at Urbana-Champaign, pp.93-102, September 1988.

[Baumgartner and Wah 1988] K. M. Baumgartner and B.W. Wah. *GAMMON: A Load Balancing Strategy for Local Computer Systems with Multiaccess Networks,* IEEE Transactions on Computers, vol. 38, no. 8, pp.1098-1109, August 1989.

[Baumgartner et. al. 1989] K. M. Baumgartner et. al. *Implementation of Gammon: An Efficient Load Balancing Strategy for a Local Computer System,* Int. Conf. on Parallel Processing, Comp. Syst. Group, University of Illinois at Urbana-Champaign, pp.77-80, 1989.

[Blake 1992] B. A. Blake. *Assignment of Independent Tasks to Minimize Completion Time,* Software Practice and Experience, vol. 22(9), pp.723-34, September 1992.

[Blazewicz et. al 1986] J. Blazewicz et. al. *Scheduling Multiprocessor Tasks to Minimize Schedule Length,* IEEE Transactions on Computers, vol. 35, no. 5, pp.389-393, May 1986.

[Boglaev 1991] Y. P. Boglaev. *Exact Dynamic Load Balancing of MIMD Architecture with Linear Programming Algorithms,* Parallel Com-

puting, Computer Science Division, Russian Academy of Sciences, pp.615-623, October 1991.

[Bokhari 1979] S. Bokhari. *Dual Processor Scheduling With Dynamic Reassignment,* IEEE Transactions on Software Engineering, vol. se-9, July 1979.

[Bowen et. al. 1988] N. S. Bowen et. al. *Hierarchial Workload Allocation for Distributed Systems,* ICPP, vol. 2, pp.102-109, 1988.

[Boutaba and Folliot 1990] R. Boutaba and B. Folliot. *Multi-Criteria Algorithm for Repartition in Heterogeneous System,* Technical Report, Laboratory of Methodology in Architecture and Systems Information, 1990.

[Bryant and Finkel 1981] R. M. Bryant and R. A. Finkel. *A Stable Distributed Scheduling Algorithm,* Proc. 2nd Int. Conf. Distributed Computer System Los Alamintos, Calis, pp.314-323, 1981.

[Butt and Evans 1992] W. U. Butt and D. J. Evans. *A Simulation Model for Task Scheduling in Distributed Systems,* Parallel Computing and Transputer Applications, IOS Press, Amsterdam, ISBN:84-87867-13-8 CIMNE, Barcelona, 1992.

[Casavant and Kuhl 1987] T. L. Casavant and J. G. Kuhl. *Analysis of Three Dynamic Distributed Load-Balancing Strategies with Varying Global Information Requirements,* IEEE Transactions on Computers, pp.185-192, 1987.

[Casavant and Kohl 1988] T. L. Casavant and J. G. Kohl. *A Taxonomy of Scheduling in General-purpose Distributed Computer Systems,*

IEEE Transactions on Software Engineering, vol. 14, no. 2, pp.141-154, February 1988.

[Chatterjee and Bassiouni 1987] S. Chatterjee and M. A. Bassiouni. *Analysis of Waiting Time in Polling Networks With Bribing Priority,* Technical Report, University of Central Florida, vol. 2. no. 4, October 1987.

[Chou and Abraham 1982] T. C. K. Chou and J.A. Abraham. *Load Balancing in Distributed Systems,* IEEE Transactions on Software Engineering, vol. 8, no. 4, pp.401-413, July 1982.

[Chow and Kohler 1979] Y. C. Chow and W. H. Kohler. *Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor Systems,* IEEE Transactions on Computers, vol. c-28, no. 5, May 1979.

[Chu et. al. 1980] W. W. Chu et. al. *Task Allocation in Distributed Data Processing,* IEEE Computer, University of California, pp.57-59, November 1980.

[Coffman and Denning 1973] E. G. Coffman and P. J. Denning. *Operating Systems Theory,* Published by Prentice Hall, 1973.

[Cybenko 1987] G. Cybenko. *Dynamic Load Balancing for Distributed Memory Multiprocessors,* Journal of Parallel and Distributed Computing vol. 7, part 2, pp.279-301, 1987.

[Darte 1991] A. Darte. *Two Heuristics for Task Scheduling,* Technical Report, Laboratory for Parallel Information, pp.1-21, September 1991.

[Deering and Cheriton 1985] S. E. Deering and D. R. Cheriton. *Host Groups: A Multicast Extension to the Internet Protocol,* RFC 966, SRI Network Information Center, December 1985.

[Deering 1988] S. E. Deering. *Multicast Routing in Internetworks and Extended LANs,* SIGCOMM, pp.55-64, 1988.

[Deering 1988a] S. E. Deering. *Host Extensions for IP Multicasting,* RFC 1054, SRI Network Information Center, May 1988.

[Diab and Harmoush 1990] H. Diab and W. Harmoush. *Packet Queuing Simulation for Parallel Processing Systems,* Technical Report, American University of Beirut, pp.13, September 1990.

[Eager et. al. 1986] D. L. Eager et. al. *Adaptive Load Sharing in Homogeneous Distributed Systems,* IEEE Transactions On Software Engineering, vol. 12, no. 5, pp.662-668, May 1986.

[Eckhouse et. al. 1978] R. H. Eckhouse et. al. *Issues in Distributed Processing-An Overview of Two Workshops,* Computer: Army Research Office, National Science Foundation, pp.22-26, January 1978.

[Efe 1982] K. Efe. *Heuristic Models of Task Assignment Scheduling in Distributed Systems,* IEEE Computer, pp.50-56, June 1982.

[Enslow 1978] P. H. Enslow Jr. *What Is A "Distributed" Data Processing System,* Computer, vol. 11, pp.13-21, January 1978.

[Ephremides et. al. 1980] A. Ephremides et. al. *A Simple Dynamic Routing Problem,* IEEE Transactions on Automatic Control, vol. 25, pp.690-693, August 1980.

[Foschini and Salz 1978] G. J. Foschini and J. Salz. *A Basic Dynamic Routing Problem and Diffusion,* IEEE Transactions on Communications, vol. 26, no. 3, pp.320-327, March 1978.

[Frank et.al. 1985] Frank et. al. *Multicast Communication on Network Computers,* IEEE Transactions on Software Engineering, pp.49-61, May 1985.

[Gao et. al. 1984] C. Gao et. al. *Load Balancing Algorithm in Homogeneous Distributed Systems,* IEEE Technical Report No.UIUC-DCS-84-1168, Department of Computer Science, University of Illinois at Urbana-Champaign, pp.302-306, 1984.

[Gil 1991] J. Gil. *Fast Load Balancing on a PRAM,* Technical Report, University of British Colombia, pp.1-14, June 1991.

[Grama et. al. 1991] A.Y. Grama et. al. *Experimental Evaluation of Load Balancing Techniques for the Hypercube,* Technical Report: Dept. of Computer Science, University of Minnesota, Minneapolis, MN 55455, USA., 1991.

[Gottlieb 1992] I. Gottlieb. *Deterministic Task Distribution in the Butterfly,* Technical Report, Bar-Ilan University, Israel. 1992.

[Greenberg and Jacquet 1991] A. Greenberg and P. Jacquet. *Load Balancing in Multiprocessor Architecture,* Technical Report: no. 1370, National Institute for Research in Information and Automatic, January 1991.

[Gylys and Edwards 1976] V. B. Gylys and J. A. Edwards. *Optimal Partitioning of Workload for Distributed Systems,* Technical Report, Texas Instruments Incorporated, pp.353-357, 1976.

[Hac and Jin 1987] A. Hac and X. Jin *Dynamic Load-balancing in a Distributed System Using a Decentralized Algorithm,* Proc. 7th IEEE International Conference on Distributed Computing Systems, pp. 170-177, Sept. 23-25, 1987.

[Hac and Johnson 1988] A. Hac and J. Johnson. *Dynamic Load Balancing Through Process and Read-Site Placement In A Distributed System,* AT & T Technical Journal, September/October 1988.

[Hac and Jin 1990] A. Hac and X. Jin. *Dynamic Load Balancing in a Distributed System Using a Sender-Initiated Algorithm,* J. Systems Software, vol. II, part 2, pp.79-94, 1990.

[Hajek 1984] B. Hajek. *Optimal Control Of Two Interacting Service Stations,* IEEE Transactions on Automatic Control, vol. 29, pp.491-499, June 1984.

[Indurkhya and Stone 1986] B. Indurkhya and H. S. Stone. *Optimal Partitioning of Randomly Generated Distributed Programs,* IEEE Transactions on Software Engineering, vol. 12, no. 3, pp.483-489, March 1986.

[Jaffe and Moss 1981] J. M. Jaffe and F. H. Moss. *A Responsive Distributed Routing Algorithm For Computer Networks,* IEEE Transactions on DCS, pp.348-353, 1981.

[Jain 1991] R. Jain. *The Art Of Computer Systems Performance Analysis,* Published by John Wiley and Sons, Inc., 1991.

[Jensen 1978] E. D. Jensen. *The Honeywell Experimental Distributed Processor,* Computer, vol. II, January 1978.

[Jereb and Pipan 1991] B. Jereb and L. Pipan. *The Problem of Scheduling Nondeterministic Task Systems,* Technical Report, 1991.

[Joosen and Verbaeten 1992] W. Joosen and P. Verbaeten. *Dynamic Load Balancing in Adaptive Parallel Applications,* Preliminary Paper for the NATO Advanced Research Workshop on Software for Parallel Computation, pp.1-12, 1992.

[Kleinrock 1976] L. Kleinrock. *Queueing Systems Theory,* vol. 1, Wiley, Newyork, USA., 1976.

[Krishnamoorthi and Wood 1966] B. Krishnamoorthi and R. C. Wood. *Time Shared Computer Operation With Both Interarrival and Service Time Exponential,* J. ACM, vol. 13, no. 3, pp.317-338, July 1966.

[Krueger and Finkel 1984] P. Krueger and R. Finkel. *An Adaptive Load Balancing Algorithm For A Multicomputer,* Computer Science Technical Report no. 539, Computer Science Department, University of Wisconsin, Madison, 1984.

[Kruskal and Weiss 1984] C. P. Kruskal and A. Weiss. *Allocating Independent Subtasks on Parallel Processors,* Technical Report, Dept. of Compt. Science, University of Illinois at Champaign-Urbabna, 1984.

[Lawler et. al. 1982] E. L. Lawler et. al. *Recent Developments in Deterministic Sequencing and Scheduling,* Technical Report: Deterministic and Stochastic Scheduling, pp.35-73, 1982.

[Legge and Ali 1990] G. Legge and M. Ali. *Unix File System Behaviour and Machine Architecture Dependency,* Vol. 20, pp.1077-1096, November 1990. Dept. of Comp. Science, University of North Texas, Texas, U.S.A. 1990.

[Leland and Ott 1986] W. E. Leland and T. J. Ott. *Load-balancing Heuristics and Process Behaviour,* Proc. ACM Sigmetrics Conf., pp. 54-69, May 1986.

[Lewis 1989] T. G. Lewis. *Parallel Programming Support Environment Research,* Technical Report, Oregon Advanced Computing Institute, 1989.

[Lewis and El-Rewini 1992] T. G. Lewis and H. El-Rewini. *Introduction to Parallel Computing,* Prentice-Hall International Editions, USA., ISBN 0-13-498916-3, 1992.

[Lin and Keller 1987] F. C. H. Lin and R. M. Keller. *The Gradient Model Load Balancing Method,* IEEE Transactions on Software Engineering, January 1987.

[Lint and Agerwala 1981] B. Lint and T. Agerwala. *Communication Issues in the Design and Analysis of Parallel Algorithms,* IEEE Transactions on Software Engineering, vol. 7, no. 2, pp.174-188, March 1981.

[Livny and Melman 1982] M. Livny and M. Melman. *Load Balancing in Homogeneous Broadcast Distributed Systems,* Proc. Modeling Perform. Eval. Comput. Sys., ACM SIGMETRICS, pp.45-55, 1982.

[Lo 1983] V. M. Lo. *Task Assignment in Distributed Systems,* Technical Report, Dept. of Computer Science, University of Illinois at Urbana- Champaign, October 1983.

[Lo 1984] V. M. Lo. *Heuristic Algorithms for Task Assignment in Distributed Systems,* IEEE 4th Int. Conf. on Distributed Computer Systems-Proceedings, pp.30-39, 1984.

[Ma et. al. 1982] P. R. Ma et. al. *A Task Allocation Model for Distributed Computing Systems,* IEEE Transactions on Computers, vol. 31, no. 1, pp.41-47, January 1982.

[MacDougall 1987] M. H. MacDougall. *Simulating Computer Systems,* Computer Systems Series, The MIT Press, 1987.

[McGregor and Boorstyn 1975] P. V. McGregor and R. R. Boorstyn. *Optimal Load Balancing in a Computer Network,* Proc. 1975 International Conference Communication, vol. III, part 40, pp.14-19. September 1975.

[Mirchandaney et. al. 1990] Mirchandaney et. al. *Adaptive Load Sharing in Heterogeneous Distributed Systems,* Journal of Parallel and Distributed Computing, vol. 9, no. 4, pp.331-346, August 1990.

[Mirchandaney et. al. 1990b] Mirchandaney et. al. *Analysis of the Effects of Delays on Load Sharing,* IEEE Transactions on Computers, vol. 38, no. 11, pp.1513-1525, November 1990.

[Misra 1986] J. Misra. *Distributed Discrete-Event Simulation,* Computing Surveys, vol. 18, no. 1, March 1986.

[NAG 1990] Numerical Algorithms Group Inc. *NAG Fortran Library,* Mark 14, vol. 7, ISBN: 1-85206-053-0, 1st. Edition, April 1990.

[O'Leary et. al. 1982] D. P. O'Leary et. al. *A Transportable System for Parallel Processing,* Technical Report, University of Maryland, chapter 3, pp.25-34, 1982.

[Ousterhout 1982] J. K. Ousterhout. *Scheduling Techniques For Concurrent Systems,* In Proceedings Of The Third International Conference On Dist. Comp. Sys., IEEE, New York, pp.20-30, 1982.

[Perihelion 1989] Perihelion Software Limited. *The Helios Operating System,* ISBN: 0-13-386004-3, Prentice Hall, 1989.

[Price and Krishnaprasad 1984] C. C. Price and S. Krishnaprasad. *Software Allocation Models For Distributed Computing Systems,* IEEE Proc. 4th Int. Conf. on Distributed Computing Systems, pp.40-48, 1984.

[Price and Salama 1990] C. C. Price and M. A. Salama. *Scheduling of Precedence-Constrained Tasks on Multiprocessors,* The Computer Journal, vol. 33, no. 3, pp.219-229, February 1990.

[Ramamoorthy et. al 1972] C. V. Ramamoorthy et. al. *Optimal Scheduling Strategies in a Multiprocessor Systems,* IEEE Transactions on Computers, February 1972.

[Reed 1984] D. A. Reed. *The Performance of Multimicocomputer Networks Supporting Dynamic Work- loads,* IEEE Transactions on Computers, vol. 33, no. 11, pp.1045-1050, November 1984.

[Rewini and Lewis 1990] H. E. Rewini and T. G. Lewis. *Scheduling Parallel Program Tasks onto Arbitary Target Machines,* Journal of Parallel Distributed Computing, vol. 9, pp.138-153, January 1990.

[Ryou and Wong 1989] J. Ryou and J. S. K. Wong. *A Task Migration Algorithm for Load Balancing in a Distributed System,* Proc. Hawaii Int. Conf. on System Sciences. IEEE Transactions on Computers, vol. II, pp.1041-1048, 1989.

[Sahni 1984] S. Sahni. *Scheduling Multipipeline and Multiprocessor Computers,* Technical Report, pp.333-337, 1984.

[Sarkar 1989] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors,* Published by Pitman Publishing, 1989.

[Shen and Tsai 1985] C. C. Shen and W. H. Tsai. *A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion,* IEEE Transactions on Computers, vol. c-34, no. 3, March 1985.

[Shirazi and Wang 1988] B. Shirazi and M. Wang. *Design and Analysis of Heuristic Functions for Static Task Distribution,* IEEE Transactions on Computers, pp.124-131, July 1988.

[Smith 1980] R. G. Smith. *High Level Communication and Control in a Distributed Problem Solver,* IEEE Transactions on Computers, vol. c-29, no. 12, December 1980.

[Stankovic 1984] J. A. Stankovic. *A Perspective on Distributed Computer Systems,* IEEE Transactions on Computer, vol. c-33, no. 12, December 1984.

[Stankovic 1984a] J. A. Stankovic. *Simulations of Three Adaptive, Decentralized Controlled, Job Scheduling Algorithms,* Computer Networks, vol. 8, pp.199-217, December 1984.

[Stankovic 1985] J. A. Stankovic. *An Application of Bayesian Decision Theory to Decentralized Control of Job Scheduling,* IEEE Transactions on Computers, vol. c-34, no. 2, February 1985.

[Stone 1978] H. S. Stone. *Critical Load Factors in Two-Processor Distributed Systems,* IEEE Transactions on Software Engineering, vol.se-4, no.3, pp.254-258, May 1978.

[Stone and Bokhari 1978] H. S. Stone and S. H. Bokhari. *Control of Distributed Processes,* IEEE Computer, pp.97-106, July 1978.

[Sun 1990] Sun Microsystems Inc. *SunView Programmers Guide,* No. 825-1253-01, Printed in USA, 1990.

[Sun 1990b] Sun Microsystems Inc. *SunView System Programmers Guide, Pixrect Reference,* No. 825-1249-01, Printed in USA, 1990.

[Tanenbaum 1981] A. S. Tanenbaum. *Network Protocols,* Computer Surveys, vol. 13, no. 4, pp.453-487, December 1981.

[Tanenbaum and Renesse 1985] A. S. Tanenbaum and R. V. Renesse. *Distributed Operating Systems,* Computing Survey, vol. 18, no. 4, pp.419-470, July 1985.

[Tanenbaum and Renesse 1986] A. S. Tanenbaum and R. V. Renesse. Technical Report: Vrijie University, Amsterdam, The Netherlands.pp.420-467, 1986.

[Tantawi and Towsley 1985] A. N. Tantawi and D. Towsley. *Optimal Static Load Balancing in Distributed Computer Systems,* Journal of the Association for Computing Machinery, vol. 32, no. 2, pp.445-465, April 1985.

[Towsley 1986] D. Towsley. *Allocating Programs Containing Branches and Loops Within a Multiple Processor System,* IEEE Transactions on Software Engineering, vol. se-12, no. 10, pp.1018-1024, October 1986.

[Thomasian 1987] A. Thomasian. *A Performance Study of Dynamic Load Balancing in Distributed systems,* IEEE DCS, pp.178-184, 1987.

[Tilborg and Wittie 1981] A. M. V. Tilborg and L. D. Wittie. *Distributed Task Force Scheduling in Milticomputer Networks,* in Proc. AFIPS, 1981.

[Ullman 1975] J. D. Ullman. *NP-Complete Scheduling Problems,* Journal of Computer and System Sciences, no. 10, pp.384-393, 1975.

[Vornberger 1988] O. Vornberger. *Load Balancing in a Network of Transputers,* vol. 312 pp.116-126, 1988.

[Wang and Morris 1985] Y. T. Wang and R. J. T. Morris. *Load Sharing in Distributed Systems,* IEEE Transactions on Computer, vol. c-34, no. 3, March 1985.

[Ward and Romero 1984] M. O. Ward and D. J. Romero. *Assigning Parallel-Executable, Intercommunicating Subtasks to Processors,* IEEE Transactions on Computers, pp.392-394, 1984.

[Wikstron et. al. 1982] Wikstron et. al. *Myths of Load Balancing,* Technical Report: Dept. of Comp. Sciences, Iowa State University, Ames, 1982.

[Wu and Sweeting 1992] S. S. Wu and D. Sweeting. *Heuristic Algorithms for Task Assignment and Scheduling in a Processor Network,* Technical Report; Department of Aeronautical Engineering, Queen Mary and Westfield College, London E1 4NS., pp.1-24, March 1992.

[Zaks 1985] S. Zaks. *Optimal Distributed Algorithms for Sorting and Ranking,* IEEE Transactions on Computers, vol. c-34, no. 4, pp.376-379, April 1985

[Zhou 1988] S. Zhou. *A Trace-Driven Simulation Study of Dynamic Load Balancing,* IEEE Transactions on Software Engineering, vol. 14, no. 9, September 1988.

Appendix A

Inter-task Communication Times (Chapter 5)

X[1] = 10	X[2] = 15	X[3] = 20	X[4] = 25
X[5] = 10	X[6] = 15	X[7] = 20	X[8] = 25
X[9] = 30	X[10] = 35	X[11] = 10	X[12] = 15
X[13] = 20	X[14] = 25	X[15] = 10	X[16] = 15
X[17] = 20	X[18] = 25	X[19] = 35	X[20] = 40
X[21] = 45	X[22] = 50	X[23] = 10	X[24] = 10
X[25] = 10	X[26] = 10	X[27] = 5	

Execution Times for the Task Graph (Figure 5.1)

C[1][5] = C[5][1] = 10	C[2][6] = C[6][2] = 10
C[2][7] = C[7][2] = 10	C[3][8] = C[8][3] = 10
C[3][9] = C[9][3] = 10	C[4][10] = C[10][4] = 10
C[5][11] = C[11][5] = 10	C[6][12] = C[12][6] = 10
C[6][13] = C[13][6] = 10	C[7][13] = C[13][7] = 10
C[7][14] = C[14][7] = 10	C[8][15] = C[15][8] = 10
C[8][16] = C[16][8] = 10	C[9][16] = C[16][9] = 10
C[9][17] = C[17][9] = 10	C[10][18] = C[18][10] = 10
C[11][19] = C[19][11] = 10	C[12][19] = C[19][12] = 10
C[17][20] = C[20][17] = 10	C[18][20] = C[20][18] = 10
C[13][21] = C[21][13] = 20	C[19][21] = C[21][19] = 10
C[16][22] = C[22][16] = 20	C[20][22] = C[22][20] = 10
C[21][23] = C[23][21] = 10	C[22][24] = C[24][22] = 10
C[23][25] = C[25][23] = 10	C[24][26] = C[26][24] = 10
C[14][27] = C[27][14] = 20	C[15][27] = C[27][15] = 20
C[25][27] = C[27][25] = 20	C[26][27] = C[27][26] = 20

Inter-task Communication Times for the Task Graph in Figure 5.1.

X[1] = 10	X[2] = 10	X[3] = 15	X[4] = 20
X[5] = 25	X[6] = 30	X[7] = 10	X[8] = 15
X[9] = 20	X[10] = 25	X[11] = 30	X[12] = 10
X[13] = 15	X[14] = 20	X[15] = 25	X[16] = 30
X[17] = 10	X[18] = 25	X[19] = 30	X[20] = 35
X[21] = 40	X[22] = 40	X[23] = 40	X[24] = 40
X[25] = 40	X[26] = 20	X[27] = 20	X[28] = 20
X[29] = 20	X[30] = 10		

Execution Times for the Task Graph (Figure 5.5)

C[1][2] = C[2][1] = 5	C[1][3] = C[1][3] = 5
C[1][4] = C[4][1] = 5	C[1][5] = C[5][1] = 5
C[1][6] = C[6][1] = 5	C[1][7] = C[7][1] = 5
C[1][8] = C[8][1] = 5	C[1][9] = C[9][1] = 5
C[1][10] = C[10][1] = 5	C[1][11] = C[11][1] = 5
C[1][12] = C[12][1] = 5	C[1][13] = C[13][1] = 5
C[1][14] = C[14][1] = 5	C[1][15] = C[15][1] = 5
C[1][16] = C[16][1] = 5	C[1][17] = C[17][1] = 5
C[2][18] = C[18][2] = 10	C[3][18] = C[18][3] = 10
C[4][18] = C[18][4] = 15	C[5][18] = C[18][5] = 15
C[6][19] = C[19][6] = 15	C[7][19 = C[19][7] = 15
C[8][19] = C[19][8] = 15	C[9][19] = C[19][9] = 15
C[10][20] = C[20][10] = 10	C[11][20] = C[20][11] = 10
C[12][20] = C[20][12] = 15	C[13][20] = C[20][13] = 15
C[14][21] = C[21][14] = 15	C[15][21] = C[21][15] = 15
C[16][21] = C[21][16] = 15	C[17][21] = C[21][17] = 15
C[18][22] = C[22][18] = 15	C[19][22] = C[22][19] = 15
C[20][22] = C[22][20] = 15	C[21][22] = C[22][21] = 15
C[18][23] = C[23][18] = 20	C[19][23] = C[23][19] = 15
C[20][23] = C[23][20] = 15	C[21][23] = C[23][21] = 15
C[18][24] = C[24][18] = 25	C[19][24] = C[24][19] = 15
C[20][24] = C[24][20] = 10	C[21][24] = C[24][21] = 15
C[18][25] = C[25][18] = 25	C[19][25] = C[25][19] = 15
C[20][25] = C[25][20] = 10	C[21][25] = C[25][21] = 15
C[22][26] = C[26][22] = 20	C[23][26] = C[26][23] = 25
C[24][26] = C[26][24] = 25	C[25][26] = C[26][25] = 25
C[22][27] = C[27][22] = 20	C[23][27] = C[27][23] = 25
C[24][27] = C[27][24] = 25	C[25][27] = C[27][25] = 25
C[22][28] = C[28][22] = 20	C[23][28] = C[28][23] = 25
C[24][28] = C[28][24] = 25	C[25][28] = C[28][25] = 25
C[22][29] = C[29][22] = 20	C[23][29] = C[29][23] = 25
C[24][29] = C[29][24] = 25	C[25][29] = C[29][25] = 25
C[26][30] = C[30][26] = 20	C[27][30] = C[30][27] = 25
C[28][30] = C[30][28] = 25	C[29][30] = C[30][29] = 25

Inter-task Communication Times for Task Graph in Figure 5.5.

Appendix B

Performance Measures (Chapter 5)

PERFORMANCE MEASURES

MODEL: scheduler model TIME: 435.000

PROCESSOR	UTIL.	MEAN BUSY PERIOD	MEAN QUEUE LENGTH	TASKS ASSIGNED	QUEUE
processor[1]	0.4460	24.250	0.202	8	3
processor[2]	0.5379	33.429	0.046	7	1
processor[3]	0.3747	23.286	0.064	7	1
processor[4]	0.2368	20.600	0.076	5	2

Performance Measures for Schedule in Gantt Chart (Figure 5.2)

PERFORMANCE MEASURES

MODEL: scheduler model TIME: 265.000

PROCESSOR	UTIL.	MEAN BUSY PERIOD	MEAN QUEUE LENGTH	TASKS ASSIGNED	QUEUE
processor[1]	0.6792	25.714	0.226	7	3
processor[2]	0.7547	25.000	0.132	8	2
processor[3]	0.6415	24.286	0.170	7	2
processor[4]	0.3396	18.000	0.075	5	2

Performance Measures for Schedule in Gantt Chart (Figure 5.3)

PERFORMANCE MEASURES

MODEL: scheduler model TIME: 215.000

PROCESSOR	UTIL	MEAN BUSY PERIOD	MEAN QUEUE LENGTH	TASKS ASSIGNED	QUEUE
processor[1]	0.5581	20.000	0.302	6	3
processor[2]	1.0000	19.545	0.302	11	3
processor[3]	0.9070	27.857	0.000	7	0
processor[4]	0.1395	10.000	0.000	3	0

Performance Measures for Schedule in Gantt Chart (Figure 5.4)

PERFORMANCE MEASURES

MODEL: scheduler model TIME: 425.000

PROCESSOR	UTIL	MEAN BUSY PERIOD	MEAN QUEUE LENGTH	TASKS ASSIGNED	QUEUE
processor[1]	0.6471	30.556	0.871	9	6
processor[2]	0.5294	28.125	0.471	8	5
processor[3]	0.4235	25.714	0.259	7	4
processor[4]	0.3647	25.833	0.412	6	4

Performance Measures for Schedule in Gantt Chart (Figure 5.5)

PERFORMANCE MEASURES

MODEL: scheduler model TIME: 280.000

PROCESSOR	UTIL.	MEAN BUSY PERIOD	MEAN QUEUE LENGTH	TASKS ASSIGNED	QUEUE
processor[1]	0.8393	33.571	0.518	7	3
processor[2]	0.9643	33.750	0.571	8	3
processor[3]	0.8393	29.375	0.607	8	3
processor[4]	0.8036	32.143	0.554	7	3

Performance Measures for Schedule in Gantt Chart (Figure 5.7)

Appendix C

Important Data-Structures Used In Simulations

```
/* Definitions and Data-structures used in the simulations of various load
balancing strategies */


#define ON      1
#define OFF     0
#define TRUE    1
#define FALSE   0
#define MAX_LNKS     20
#define MAX_NODES    4
#define max_brdth     60
#define max_path      25
#define group_size    4


define random(min,max)    ((rand() % (max-min + 1)) + min)

#define MAX_SEQ      2000          /* changes with the no. of packets transmitted */
#define lb_exec      0
#define X_cost       0
#define nump         16
#define numt         6400


int     TRACE;
int     FLAG[nump + 1];

typedef struct  token{          /* A TASK */

        int     id;
        int     status;
        int     processor;
        int     x_time[nump + 1];       /* Execution time for a
                                           task on different processors */
        int     arrival;
        int     st_time;
        int     finish_time;
}TASKS;
```

```
typedef struct  link{           /* A LINK */

        int     end1;
        int     end2;
        int     busy;
        int     lnk_cap;
}LINK;


typedef struct  node{           /* A HOST */
        int     id;
        int     cluster;
        int     linknum;
        int     server;
        int     load_vector[nump + 1][2];       /* keeps the global
                                                   load info, 1 is old and 2 is new   */
        LINK    link_id[MAX_LNKS];
        int     recvd[MAX_SEQ];
}NODE;

typedef struct  group{          /* A HOST GROUP */
        int     id;
        int     hostnum;
        int     host[nump + 1];
        int     manager;
}GROUP;

typedef struct  location{       /* Used to store a pixels' location */
        int     id;
        int     xmid;
        int     ybottom;
}LOCATION;


typedef struct pack{            /* A PACKET   */
      int    type;
      int    src_processor;
      int    seq;
        int     msg_len;
}PACK;

typedef struct  event{          /* AN EVENT */
      PACK   packet;
      int    delay;
      struct    event   *next_event;

}EVENT;


int     sim_time,transmitted;
int     ave_util,sum_lnk;
int     util_time[nump + 1][MAX_LNKS];
int     pack_count[nump + 1][MAX_LNKS];
int     maxq_len[nump + 1][MAX_LNKS];
struct  node    processor[nump + 1];
PACK    deque();                /* packet received on a link and broadcasted
```

```
                                    on all other incident links */
EVENT *enque();
EVENT *que[nump+1][MAX_LNKS];
int    que_len[nump+1][MAX_LNKS];
double ranf(), expntl(), erlang();

/* definitions used in implementing a randomly created task graph  */

int    proc_gen;
int    level_cnt[max_path+1],level[max_path+1][max_brdth+1];
```

Appendix D

Selected Software Examples.

```
/* This program simulates the static task allocation strategy described in
chapter 5. Several other routines are called inside, which are defined in other
files. The routine display() defined at the end of this program is used to
display the Gantt chart, described in Chapter 4.
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "smpl.h"
#include "sched.h"
#define busy 1

/* libraries and definitions used in the implementation of Gantt chart */

#include   <suntool/sunview.h>
#include   <suntool/tty.h>
#include   <suntool/canvas.h>
#include   <sunwindow/pixwin.h>
#include   <sunwindow/window_hs.h>
#include   <suntool/scrollbar.h>
#define    pr_line_h_DEFINED
#include   <pixrect/pr_line.h>
#define  font_offset(font)     (-font->pf_char['n'].pc_home.y)
#define  font_height(font)     (font->pf_defaultsize.y);


#define LEFT_MARGIN    5
#define RIGHT_MARGIN   5
#define BOTTOM_MARGIN  5
#define SUBWINDOW_SPACING      5


#define CANVAS1_WIDTH  100
#define CANVAS1_HEIGHT  100
#define CANVAS2_WIDTH  1000
#define CANVAS2_HEIGHT  1000

extern  struct token task[numt+1];   /* array of tasks, where each task is a
                                        structure                  */
```

```
extern  struct  node   processor[nump+1];     /* array of the processors  */


int     running[nump+1];
extern  int  LT;
extern  int  LQ;
extern  int  SUM_EXEC;
extern  int  ready[numt+1];   /* This list stores the ready tasks  */
extern  int  I1[512],I3[512];
extern  int  waiting[numt+1][numt+1];       /* Tasks waiting for their preceding tasks
to finish  */
extern  int  comm[numt+1][numt+1];          /* Inter-task communication times */
extern  int  prec[numt+1][numt+1];          /* Dependency between two tasks  */
extern  int  hop[nump+1][nump+1];           /* Distance between two hosts measured
in hops  */
extern  real  clock;              /* simulation time   */
int     event;

int     com_delay[numt+1];
int     order[numt+1];
int     display();       /* displays the Gantt chart for the current schedule */
Canvas  canvas1, canvas2;
Pixwin  *pw1, *pw2;  /* handles used to access a window  */
Rect    framerect;
PIXFONT *font;


int  main(){

        int     iter=0;
        int     releas=0;
        real    sim_time;
        int     temp,set,max_com,max_finish,min_hop;
        int     total_comm;
        int     idle,minim,done,origin;
        int      i,j,k,l,m,s,loop,count,ord=1;
        int     total_response;
        int     task_submit_time[numt+1];
        int     load[nump+1];
        int     pr_order[numt+1];
        int     released[numt+1],scheduled[numt+1];
        TASKS  *p;



        printf("Do you want to set TRACE ON? Enter (1):");
        scanf("%d",&TRACE);
        total_comm=0;
        for(m=1; m<=numt; m++)
                com_delay[m]=0;
        init();             /* Initialization routine for setting up the
                               simulation environment       */
        smpl(0,"scheduler model");
        for(i=1; i<=numt+1; i++)
                released[i]=scheduled[i]=0;
        for(i=1; i<=nump; i++)
                load[i]=0;
        for (i=1; i<=nump; i++){
```

```
                processor[i].server=facility("processor",1);  /* A single que-single
                                                              server facility  */

        }
        for (i=1; i<=numt; i++){
           if(ready[i]==0){
                continue;
              }
           schedule(1,clock,ready[i]);  /* Schedule the independent processes  */
           task_submit_time[ready[i]]=(int)clock;
        }
        loop=numt;
        s=1;
        while(loop)
        {
                printf("\n");
                printf("\n");
                for(k=1; k<=numt; k++)
                pr_order[k]=0;
                count=0;

                for(k=1; k<=numt; k++){
                        for(l=1; l<=numt; l++){
                           if(waiting[k][l]==1 || ready[l]==k)
                             break;
                           if(l==numt){
                             max_com=0;
                             max_finish=0;
/*
                             for(m=1;m<=numt;m++){
                                     if(prec[k][m]==TRUE){

                                             if(task[m].finish_time>max_finish)
                                                     max_finish=task[m].finish_time;

                                            if(task[k].processor!=task[m].processor)
                                                     if(comm[k][m]+task[m].finish_time
>max_finish){
                                                     max_finish=comm[k][m]+task[m].
finish_time;
                                                }
                                           }
                                        }

*/

  /* tasks are ordered on the bases of the task-size  */

                                count++;
                                pr_order[count]=k;
                                printf("order-count = %d\n",pr_order[count]);
                                     }
                          }
                        }
                        for(k=1; k<=count; k++){
                          if(pr_order[k]==0)
                             break;
                          for(l=1; l<=count; l++){
```

```
                    if(l>k){
                     if(pr_order[l]==0)
                       break;
                       if(task[pr_order[k]].x_time[1]<=task[pr_order[l]].x_time[1])
                         continue;
                       else{
                         temp=pr_order[k];
                         pr_order[k]=pr_order[l];
                         pr_order[l]=temp;
                               }
                        }
                      }
                    }
                     for(k=1; k<=count; k++)
                    printf("task order is %d\n",pr_order[k]);

                 for(k=1; k<=count; k++){
                  if(pr_order[k]==0)
                     break;
                  if(scheduled[pr_order[k]]!=1){
                    printf("clock .. =%f\n",clock);
                    schedule(1,clock+(real)k,pr_order[k]);

task_submit_time[pr_order[k]]=(int)clock;
                    task[pr_order[k]].st_time=max_finish;
                    ready[pr_order[k]]=pr_order[k];
                  }
                 }

              cause(&event, &i);
              printf("event no=%d and task no=%d\n",event,i);
              p=&task[i];
              if(event==3)
                 p->finish_time=(int)clock;
              if(event==2 && scheduled[i]==0)
                     event=4;
              printf("clock = %f\n",clock);
              switch(event)
              {

              case 1: if(scheduled[i]!=1){
                        done=0;
                        temp=0;
                        set=0;
                        idle=random(1,nump);
                        printf("task %d initiated on processor %d\n",i,idle);

/* Largest communication time among all the preceding tasks is considered */

/*                max_com=0;
                  for(k=1;k<=numt;k++){
                    if(prec[p->id][k]==TRUE){
                        if(task[p->id].processor!=task[k].processor)
                           if(comm[p->id][k]>max_com) {
                                 max_com=comm[p->id][k];
                                 max_finish=task[k].finish_time;
                                 temp=task[k].processor;
                                 printf("temp = %f\n",temp);
```

```
                        }
                    }
                }
*/

/* Determine the scheduled task's start time  */

                    for(m=1; m<=numt; m++){
                     if(prec[p->id][m]==TRUE){
                        if((comm[p->id][m]+task[m].finish_time) >= max_finish){
                                max_finish=task[m].finish_time;
                                temp=m;
                                com_delay[p->id]=0;
                        }
                     }
                    }

                    for(k=1; k<=numt; k++){
                     if((prec[p->id][k]==TRUE) && (k != temp)){
                        if((comm[p->id][k]+task[k].finish_time) >=
task[temp].finish_time){
                            if(task[temp].processor!=task[k].processor){
                                max_finish=task[k].finish_time+comm[p->id][k];
                                com_delay[p->id]=comm[p->id][k];
                            }
                        }
                     }
                    }
                    if(temp != 0){
                    if(((load[task[temp].processor]+task[p-
>id].x_time[task[temp].processor]) <= LT) &&
(l3[processor[task[temp].processor].server] <= LQ)){
                        idle=task[temp].processor;
                            set=1;
                      }
                        else{
                            max_finish=0;
                            for(m=1;m<=numt;m++){
                               if(prec[p->id][m]==TRUE){

                                  if(comm[p->id][m]+task[m].finish_time>max_finish){
                                      max_finish=comm[p->id][m]+task[m].finish_time;
                                              com_delay[p->id]=comm[p->id][m];
                                  }
                                 }
                     }
                        }
                 }
                task[i].st_time=max_finish;
                }

        /* Find a non-busy processor, that's minimum hops away from the
        processor where the task initiated          */

                    if(set==0){
                    origin=idle;
                    if(running[idle]==busy){
                        done=0;
```

```
                    for(k=1; k<=nump; k++){
                     if(running[k]==0){
                      idle=k;
                min_hop=hop[origin][k];
                      done=1;
                      break;
                     }
                }
                    for(k=1; k<=nump; k++){
                     if(running[k]==0){
                      if(hop[origin][k] < min_hop){
                        min_hop=hop[origin][k];
                        idle=k;
                       }
                      }
                      if(k==nump && done==1)
                        printf("task %d scheduled on idle processor %d\n",i,idle);
                     }
                      if(done!=1){
                         for(j=1; j<=nump; j++){
                          if(l3[processor[j].server]==0){
                              idle=j;
                              min_hop=hop[origin][j];
                              done=1;
                              break;
                            }
                          }
                         for(j=1; j<=nump; j++){
                            if(l3[processor[j].server]==0){
                              if(hop[origin][j] < min_hop){
                           min_hop=hop[origin][j];
                                    idle=j;
                             }
                            }
                            if(j==nump && done==1)
                              printf("scheduled on processor %d with 0 q-
length\n",idle);
                           }
                      }
                      if(done!=1){
                              minim=l3[processor[origin].server];
                              for(j=1; j<=nump; j++){
                                    if((minim>=l3[processor[j].server]) &&
/*
(load[j]<=LT)){  */
                                 if(minim>l3[processor[j].server]){
                                    minim=l3[processor[j].server];
                                    idle=j;
                                   }
                               }
                              printf("scheduled on processor %d with min. q-
length\n",idle);
                        }
                      }
                    }
                      if(request(processor[idle].server,i,1)!=0){
                         load[idle]=load[idle]+task[i].x_time[idle];
                         scheduled[p->id]=1;
                         printf("All the processors are busy:\n");
```

```
                                printf("processor[%d] q-
length = %d\n",s,l3[processor[idle].server]);
                            }
                          else{
                              scheduled[p->id] = 1;
                              released[p->id] = 1;

/* Largest communication time among all the preceding tasks is considered */

/*                        max_com = 0;
                          for(k = 1;k< = numt;k + +){
                        if(prec[p->id][k] = = TRUE){
                              if(task[p->id].processor! = task[k].processor)
                                if(comm[p->id][k] > max_com){
                                      max_com = comm[p->id][k];
                                      printf("max-com = %f\n",max_com);
                                 }
                      total_comm = total_comm + comm[j][k];
                              }
                          }                                                    */
                              if(p->st_time < = (int)clock){
/*                                  if(com_delay[p->id] > ((int)clock-p->st_time))
                                      com_delay[p->id] = com_delay[p->id]-
((int)clock-p->st_time);

*/
                                  com_delay[p->id] = 0;
                                  p->st_time = (int)clock;
                              }
                              p->processor = idle;
                              load[idle] + = task[i].x_time[idle];
                              running[idle] = busy;
                              printf("processor start time = %d\n",p->st_time);
                              schedule(2,clock,i);
                          }
                        break;

              case 2:

                        printf("st_time = %d\n",p->st_time);
                        printf("x_time = %d\n",p->x_time[p->processor]);
/*                      if ((p->st_time + p->x_time[p->processor]) < = clock){
                            printf("x-time = %d\n",p->x_time[p->processor]);
                              schedule(3,0.0,i);
                              scheduled[i] = 0;
                        }
                        else{
*/
                         releas = 0;
                         for(k = 1; k< = numt; k + +){
                           if(scheduled[k] = = 1 && released[k] = = 1){
                              if ((task[k].st_time + task[k].x_time[p->processor]) < =
clock){
                           schedule(3,clock + numt,k);
                                  scheduled[k] = 0;
                                  releas = 1;
                              }
                           }
```

```
                        }
                        if(scheduled[i] = = 1){
                                schedule(2,clock,i);
                                if(releas! = 1)
                                   clock = clock + 1.0;
                         }
                        break;

                case 3:  release(processor[p->processor].server,i);
                        idle = p->processor;
                        load[idle] -= task[i].x_time[idle];
                        order[ord] = i;
                        ord + +;
                        for(k = 1; k< = numt; k+ +)
                          waiting[k][i] = 0;
                        running[p->processor] = 0;
                        loop--;
                        break;

                case 4:  printf("process %d already finished\n",i);
                        break;
                }
        }
        total_response = 0;
        for(j = 1;j< = numt;j+ +){
          printf("task %d response time = %d\n",j,(task[j].finish_time-
task_submit_time[j]));
          total_response = total_response + (task[j].finish_time-task_submit_time[j]);
        }
        printf("Mean Task Size = %d\n",(SUM_EXEC/numt));
        printf("total response = %d\n",total_response);
        printf("Mean Response Time = %d\n",total_response/numt);

        printf("All the processes executed in %f time units\n",clock);
        printf("completion time = %f\n",clock);
        reportf();

        if(TRACE = = OFF){
                display();
        }


}


display(){


        int     j,k,m;
        int     ready_count = 0;
        int     count,diff,pr_no,greater;
        int    task_st_trak[numt + 1],task_fin_trak[numt + 1];
      int    ytop,ybottom,horz_lin,xleft[nump + 1],xright[nump + 1];
      int    xright_trak[nump + 1],xbase[nump + 1];
      int    l_com[nump + 1], r_com[nump + 1],ybase,ybase1;
        char   tex[20],st_tex[20],fin_tex[20];
```

```
        LOCATION      locat[numt+1];
      Tty     tty;
      Frame   frame;
      static  Notify_value    catch_resize();
      Scrollbar  sb;
      int     diag_top,diag_bot;
      int     tty_fd;


      frame=window_create(NULL, FRAME,WIN_WIDTH, 1000, WIN_HEIGHT, 1000,
FRAME_LABEL, "Scheduler Model", 0);
        tty=window_create(frame, TTY, 0);
        tty_fd=(int)window_get(tty, TTY_TTY_FD);
      canvas2=window_create(tty, CANVAS,CANVAS_AUTO_SHRINK, FALSE,
WIN_HORIZONTAL_SCROLLBAR, scrollbar_create(SCROLL_LINE_HEIGHT, 10,
SCROLL_BUBBLE_MARGIN,4,0), 0);
        canvas1=window_create(NULL, FRAME, 0);
        pw1=canvas_pixwin(canvas1);
        pw2=canvas_pixwin(canvas2);
      font=pf_default();

      resize(frame);
      (void) notify_interpose_event_func(frame, catch_resize, NOTIFY_SAFE);

        j=0;
        ybase=100;
        for(k=1; k<=nump; k++){
                l_com[k]=r_com[k]=0;
                xright_trak[k]=0;
                xbase[k]=50;
                xleft[k]=xbase[k];
        }
        count=numt;
        while(count!=0){
                j++;
                pr_no=task[order[j]].processor;
                ytop=ybase+(pr_no)*50;
                ybottom=ytop+49;
                diff=task[order[j]].finish_time-task[order[j]].st_time;
                greater=0;

/* Record the maximum communication delay between this process and any of its
   precedents  */

                for(m=1; m<=numt; m++){
                        if(prec[order[j]][m]==1){

/* scale adjustments  */

                          if(task_fin_trak[m] > greater){
                            greater=task_fin_trak[m];
                              if(xleft[pr_no] < greater){
                                if(task[order[j]].st_time!=task[m].finish_time &&
xleft[pr_no]!=xbase[pr_no])
                                  xleft[pr_no]=greater+(task[order[j]].st_time-
task[m].finish_time);
                                  else
```

```
                                xleft[pr_no]=greater;  /*  +(task[order[j]].st_time-
task[m].finish_time); */
                            }
                        }
                    }
                }
                xright[pr_no]=xleft[pr_no]+diff+30;

/* Adjust the new processes finish time with the previously finished processes
   according to the scale  */

                for(m=1; m<=j-1; m++){
                  if(task[order[j]].finish_time > task[order[m]].finish_time)
                        if(xright[pr_no] < task_fin_trak[order[m]])

xright[pr_no]=task_fin_trak[order[m]]+task[order[j]].finish_time-
task[order[m]].finish_time;
                }

/* Adjust the processors, if two tasks have the same start times  */

                for(m=1; m<=j-1; m++){
                  if(task[order[j]].st_time == task[order[m]].st_time)
                    xleft[pr_no]=task_st_trak[order[m]];     /* -
task[order[m]].x_time[pr_no]-30;  */
                }

                for(m=1; m<=j-1; m++){
              if(task[order[j]].st_time == task[order[m]].finish_time)
                xleft[pr_no]=task_fin_trak[order[m]];
                }
                for(m=1; m<=j-1; m++){
              if(task[order[j]].st_time > task[order[m]].finish_time)
                if(xleft[pr_no]<=task_fin_trak[order[m]])
                        xleft[pr_no]=task_fin_trak[order[m]]+(task[order[j]].st_time-
task[order[m]].finish_time);
                }


/* Adjust the processors, if two tasks have the same finish times  */

                for(m=1; m<=j-1; m++){
                  if(task[order[j]].finish_time == task[order[m]].finish_time)
                        xright[pr_no]=task_fin_trak[order[m]];
                }

                if(com_delay[order[j]] != 0){
                        greater=0;
                        l_com[pr_no]=xleft[pr_no];
                        xleft[pr_no]=xleft[pr_no];  /*  +com_delay[order[j]]; */
                        for(m=1; m<=numt; m++){
                           if(prec[order[j]][m]==1){
                                if(task_fin_trak[m] > greater){
                                   greater=task_fin_trak[m];
                                }
                           }
                        }
                    if(xright_trak[pr_no] < greater)
```

```
                {
                        l_com[pr_no]=greater;
                        printf("greater = %d\n",greater);
                }
                else
                {
                        l_com[pr_no]=xright_trak[pr_no];
                        printf("xright trak = %d\n",xright_trak[pr_no]);
                }

                   r_com[pr_no]=xleft[pr_no];
                   printf("task %d lcom %d = \n",order[j],l_com[pr_no]);
                   printf("rcom %d = \n",r_com[pr_no]);
            }

            sprintf(tex,"T%d",order[j]);
            sprintf(fin_tex,"%d",task[order[j]].finish_time);
            sprintf(st_tex,"%d",task[order[j]].st_time);

      pw_vector(pw2, xleft[pr_no], ytop, xright[pr_no],ytop,PIX_SRC,1);
      pw_vector(pw2, xleft[pr_no], ybottom, xright[pr_no],ybottom,PIX_SRC,1);
      pw_vector(pw2, xleft[pr_no], ytop, xleft[pr_no],ybottom,PIX_SRC,1);
      pw_vector(pw2, xright[pr_no], ytop, xright[pr_no],ybottom,PIX_SRC,1);
      diag_bot=xleft[pr_no]+3;
      diag_top=xleft[pr_no];
      while(diag_bot<=xright[pr_no]){
              pw_vector(pw2, diag_top, ytop, diag_bot,ybottom,PIX_SRC,1);
          diag_bot=diag_bot+3;
             diag_top=diag_top+3;
      }

/* Print start and finish times for all the tasks on all the processors  */

            pw_text(pw2,xleft[pr_no]+5,(ytop+ybottom)/2,PIX_SRC, 0,tex);
            pw_text(pw2,xright[pr_no],ybase+30-(pr_no*12),PIX_SRC, 0,fin_tex);
              pw_text(pw2,xleft[pr_no],ybase+(nump+1)*50+(pr_no*15),PIX_SRC,
0,st_tex);
              task_st_trak[order[j]]=xleft[pr_no];
              task_fin_trak[order[j]]=xright[pr_no];

              xright_trak[pr_no]=xright[pr_no];
              xleft[pr_no]=xright[pr_no];

/* Draw the horizontal line pattern to indicate the communication time elapsed */

              if(com_delay[order[j]] != 0){
              pw_vector(pw2, l_com[pr_no], ytop, r_com[pr_no],ytop,PIX_SRC,1);
          pw_vector(pw2, l_com[pr_no], ybottom, r_com[pr_no],ybottom,PIX_SRC,1);
          pw_vector(pw2, l_com[pr_no], ytop, l_com[pr_no],ybottom,PIX_SRC,1);
          pw_vector(pw2, r_com[pr_no], ytop, r_com[pr_no],ybottom,PIX_SRC,1);

              horz_lin=ytop+2;
              while(horz_lin <= ybottom){
                pw_vector(pw2,l_com[pr_no] , horz_lin, r_com[pr_no],
horz_lin,PIX_SRC,1);
                horz_lin=horz_lin+2;
              }
              }
```

```
                count--;
        }
        pw_text(pw2,30,ytop-32,PIX_SRC, 0,"P");
        pw_text(pw2,30,ytop-16,PIX_SRC, 0,"r");
        pw_text(pw2,30,ytop,PIX_SRC, 0,"o");
        pw_text(pw2,30,ytop+16,PIX_SRC, 0,"c");
        pw_text(pw2,30,ytop+32,PIX_SRC, 0,"e");
        pw_text(pw2,30,ytop+48,PIX_SRC, 0,"s");
        pw_text(pw2,30,ytop+64,PIX_SRC, 0,"s");
        pw_text(pw2,30,ytop+80,PIX_SRC, 0,"o");
        pw_text(pw2,30,ytop+96,PIX_SRC, 0,"r");
        pw_text(pw2,(50+xright[pr_no])/2,ytop+(nump+1)*50,PIX_SRC, 0,"Time ------
>");
          for(m=1; m<=nump+1; m++){
                pw_vector(pw2, 50, ybase+50*(m),
xright[pr_no],ybase+50*(m),PIX_SRC,1);
          }
          pw_vector(pw2, 50, ybase+50, 50,ybase+50*(nump+1),PIX_SRC,1);
          pw_vector(pw2, 50, ybase+50, xright[pr_no],ybase+50,PIX_SRC,1);

          ybase1=CANVAS2_HEIGHT+50;
        if(ready_count==1){
          locat[1].id=1;
          locat[1].xmid=1000/2;
          locat[1].ybottom=ybase1+30;
          pw_vector(pw1,locat[1].xmid-10, locat[1].ybottom-20, locat[1].xmid+10,
locat[1].ybottom+10,PIX_SRC,1);
          pw_vector(pw1,locat[1].xmid-10, locat[1].ybottom, locat[1].xmid+10,
locat[1].ybottom,PIX_SRC, 1);
          pw_vector(pw1,locat[1].xmid-10, locat[1].ybottom-20, locat[1].xmid-10,
locat[1].ybottom,PIX_SRC, 1);
          pw_vector(pw1,locat[1].xmid+10, locat[1].ybottom-20, locat[1].xmid+10,
locat[1].ybottom,PIX_SRC, 1);
        }

          window_main_loop(frame);
          exit(0);
}



static  Notify_value
        catch_resize(frame, event, arg, type)
          Frame         frame;
          Event         *event;
          Notify_arg    arg;
          Notify_event_type     type;
{
        Notify_value    value;

        value=notify_next_event_func(frame,event,arg,type);
        if(event_action(event) == WIN_RESIZE)
              resize(frame);
        return(value);
}


resize(frame)
```

```
  Frame    frame;
{

      Rect    *r;
      int     canvas2_width;
      int     stripeheight;

      if((int)window_get(frame, FRAME_CLOSED))
            return;

      r=(Rect *) window_get(frame, WIN_RECT);
      framerect=*r;
      stripeheight= (int) window_get(frame, WIN_TOP_MARGIN);
      canvas2_width=CANVAS2_WIDTH+(int) scrollbar_get(SCROLLBAR,
SCROLL_THICKNESS);
      window_set(canvas2,
            WIN_X,         0,
            WIN_Y,         0,
            WIN_WIDTH,     CANVAS2_WIDTH,
            WIN_HEIGHT,    CANVAS2_HEIGHT,
            0);
      window_set(canvas1,
            WIN_X,         framerect.r_width-CANVAS1_WIDTH-LEFT_MARGIN-
SUBWINDOW_SPACING,
            WIN_Y,         0,
            WIN_WIDTH,     framerect.r_width-CANVAS2_WIDTH-LEFT_MARGIN-
SUBWINDOW_SPACING-RIGHT_MARGIN,
            WIN_HEIGHT,    framerect.r_height-CANVAS2_HEIGHT-stripeheight-
SUBWINDOW_SPACING-BOTTOM_MARGIN,
            0);
}
```

```
/*
**********************    *
*                         *
*  Dated: February  1,92.  *
*  Written By:  Wajeeh Butt  *
**********************    *

This is the main program that implements the probabilistic queueing
model for a multiple server system as in the figure 2.  Each host is
modelled as a M/M/1 queue.

Each host broadcasts the load information to each other host periodically.
Each host estimates the average queue length over the whole system.
During the next interval, based on average q-length , load balancing is
performed dynamically by finding out the transfer probabilities.

*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "smpl.h"
#include "sched.h"
#define  busy  1

extern  struct token task[numt+1];    /* array of tasks, where each task is a
                                         structure                  */
extern  int       evl;
extern  struct  node    processor[nump+1];          /* array of the processors  */
int       running[nump+1];                          /* status of a processor */
extern    int   l1[4024],l3[4024];
extern  int   hop[nump+1][nump+1];
extern  real  clock;                    /* simulation time   */
extern  short   jf;                     /* used to store the value of seed  */
int     period=0;                       /* used to send the load information updates
                        periodically   */
int       flag;
int       MIG[nump+1][nump+1];          /* Number of tasks transferred between two
processors
 */
double  P[nump+1][nump+1];              /* Stores the task-transfer probability */
double  rate[nump+1][nump+1];           /* used to determine the probability for the
                                           next task arrival  */
FILE    *fd;
PACK    transfer_pack;                  /* A packet simulating the task transfer overhead
*/
int       interval=400;                 /* load update interval  */
int       sample;                       /* Number of sample runs for each simulation */
int       bias=2;                       /* A bias value used to determine the load
threshold */
int       q_mean[nump+1];               /* Stores the value of average queue-length  */
int       sim_time=0;
int       total_res;                    /* Used to calculate the task response times  */
int       task_submit_time[numt+1];     /* Task submission time used to calculate the
response time of a task          */
void    weight();
```

```
double unbalance();             /* Calculates the load imbalance  */
double Avg_W[nump+1];
double P_sum[nump+1];       /* Sum of probabilities */
double old_P[nump+1][nump+1];
int     load[nump+1];
int     done[numt+1];
long    total_response;
int     event;
int     C[nump+1][nump+1];          /* Stores the inter-processor communication times
*/
int    recent_update[nump+1][nump+1];

int  main(argc,argv)

int     argc;
char    *argv[];

{
        int     numtask,numproc;                /* Used to store the argument values */
        int     try=0,iter;
        int     donor,acceptor;                 /* Indicates the number of donor and
acceptor hosts */
        int     donorq_sum,newW_sum;
        int     ex_value = 0;
        int     releas=0;
        int     lb_releas=0;
        int     A_index[nump+1];    /* Used to save the identification of an acceptor
host */
        int     D_index[nump+1];    /* Used to save the identification of a donor host
*/
        int     delta_qd[nump+1];
        int     delta_qa[nump+1];
        int     transfer[nump+1][nump+1];
        double  rms[1000];
        double  W[nump+1];
        double  newW[nump+1][nump+1];
        int     lock;
        int     hl;     /* keeps count of the heavily loaded processors       */
        double  ex_clock=0;
        double  max_P;
        int     temp,set,max_com,max_finish,min_hop;
        int     RECV[nump+1];
        int     total_comm,balancer[nump+1];  /* com_delay[numt+1]  */
        int     greater,pr_no,idle,minim,select,origin,ready_count;
        int     i,j,k,l,m,s,link,loop,count,diff,ord=1;
        int     lb_start[numt+1];

        int     task_trak[numt+1],pr_order[numt+1],order[numt+1],released[numt+1],
lb_released[numt+1],scheduled[numt+1],lb_scheduled[numt+1];
        TASKS  *p;
        FILE    *fd1, *fopen();


        if(argc!=3){
                printf("Usage: find pattern\n");
                exit(0);
        }
        while(--argc>0){
```

```
        if(argc==2){
                numtask=(int)argv[1];
        }
        if(argc==1){
                numproc=(int)argv[argc];
        }
}
if((fd1=fopen("crap1","w"))==NULL){
        fprintf(stderr,"Cannot open the file 'imbalance' to write\n");
        exit(1);
}
if((fd=fopen("statcrap1","w"))==NULL){
        fprintf(stderr,"Cannot open the file 'stat' to write\n");
        exit(1);
}
printf("Do you want to set TRACE ON? Enter (1):");
scanf("%d",&TRACE);
TRACE=1;
for(i=1; i<=1000; i++)
rms[i]=0;
for(i=1; i<=nump; i++)
for(j=1; j<=nump; j++)
        ave_util[i][j]=0;

sample=0;
 while(sample <= 10){
sample++;
jf=stream(sample);
printf("seed = %d\n",stream(sample));
iter=0;
clock=0.0;
sim_time=0;
period=0;
ex_value=0;
releas=0;
lb_releas=0;
ex_clock=0;
total_res=0;
smpl(0,"scheduler model");
reset();
init();             /* Initialization routine for setting up the
                       simulation environment       */

/* initializations for the broadcast routine   */

for(i=1; i<=nump; i++){
        for(j=1; j<=nump; j++){
                util_time[i][j]=0;
        }
}
transmitted=0;

for(i=1; i<=numt+1; i++)
        released[i]=lb_released[i]=scheduled[i]=lb_scheduled[i]=0;

for(i=1; i<=numt; i++){
        done[i]=0;
}
```

```
for(i=1; i<=nump; i++)
        for(j=1; j<=nump; j++)
                transfer[i][j]=MIG[i][j]=0;

for(i=1; i<=nump; i++)
        W[i]=Avg_W[i]=0.0;

for(i=1; i<=nump; i++)
        RECV[i]=load[i]=0;
for (i=1; i<=nump; i++)
        processor[i].server=facility("processor",1);  /* A single que-single
                                                         server facility  */
for (i=1; i<=nump; i++)
        balancer[i]=facility("balancer",1);  /* A load balancing
                        component associated with each processor   */

period=interval;
loop=numt;
while(loop)
{
        if(TRACE==2){
          printf("\n");
          printf("\n");
        }
for (i=1; i<=numt; i++){
 if(done[i]==0){
  if(((int)clock==task[i].arrival) && (scheduled[i]==0)){
   schedule(4,(real)(task[i].arrival),i);
   task_submit_time[i]=task[i].arrival;
   task[i].st_time=(int)clock;
   lb_start[i]=task[i].arrival;
   done[i]=1;
  }
 }
}
    for(k=1; k<=nump; k++){

/*  Each processor's own load is updated immediately    */

          processor[k].load_vector[k][2]=l3[processor[k].server];
          }
          if(lock==50){
                iter++;
                rms[iter]=rms[iter]+unbalance();
                lock=0;
          }


/* Based on the information exchange for the queue-length calculate the mean
   que_length for load balancing  */

      if(((int)clock==(period-1)) && (clock != ex_value)){
          ex_value=clock;
          weight();

/* Find the donor and acceptor host groups   */
```

```
        donor=0;
        acceptor=0;
        donorq_sum=0;

        for(i=1; i<=nump; i++){
                if(l3[processor[i].server] > (q_mean[i]+bias)){
                        donor++;
                        D_index[donor]=i;
                        delta_qd[i]=l3[processor[i].server]-(q_mean[i]+bias);
                        donorq_sum=donorq_sum+delta_qd[i];
                }
        }
                        for(j=1; j<=nump; j++){
                                if(l3[processor[j].server] < (q_mean[j]-bias)){
                                        acceptor++;
                                        A_index[acceptor]=j;
                                        delta_qa[j]=(q_mean[j]+bias)-
l3[processor[j].server];
                                }
                        }

/* Calculate the weight factors to determine the probabilities   */

        for(i=1; i<=donor; i++){
          W[D_index[i]]=(double) delta_qd[D_index[i]]/donorq_sum;
                printf("W[%d] = %f\n",D_index[i],W[D_index[i]]);
        }

/* Multiply the weight factors with the corresponding acceptance ratios  */

        for(i=1; i<=donor; i++){
          if(D_index[i]!=0){
                for(j=1; j<=acceptor; j++){
                  if(A_index[j]!=0){
                        newW[D_index[i]][A_index[j]]=W[D_index[i]]*delta_qa[A_index[j]]
;
                        newW_sum=newW_sum+newW[D_index[i]][A_index[j]];
                  }
                }
          }
        }

/* Determine the transfer probabilities   */

        for(i=1; i<=nump; i++)
                P_sum[i]=0;

        for(i=1; i<=nump; i++)
          for(j=1; j<=nump; j++)
                old_P[i][j]=P[i][j]=0.0;

        for(i=1; i<=donor; i++){
          if(D_index[i]!=0){
          for(j=1; j<=acceptor; j++){
                if(A_index[j]!=0){

old_P[D_index[i]][A_index[j]]=P[D_index[i]][A_index[j]]=(double)newW[D_index[i]][A_ind
ex[j]]/newW_sum;
```

```
                P_sum[D_index[i]] = P_sum[D_index[i]] + P[D_index[i]][A_index[j]];
            }
          }
         }
        }

        for(j = 1; j <= donor; j++){
            old_P[D_index[j]][0] = P[D_index[j]][0] = 1.0-P_sum[D_index[j]];
        }

        for(i = 1; i <= nump; i++)
            W[i] = 0.0;
            for(j = 1; j <= nump; j++)
                newW[i][j] = 0.0;

        newW_sum = donorq_sum = 0;

  }
/*  Check the clock value and broadcast the load information of each processor
      load  */

        if((int)clock == (period + interval-2))
            period = period + interval;

            dist_monitor();
/* Save the old load information for calculating the rate of change of que   */
                if(TRACE == 2){
              for(k = 1; k <= nump; k++){
                printf("LOAD TABLE FOR PROCESSOR %d\n",k);
                for(l = 1; l <= nump; l++){
                        processor[k].load_vector[l][1] = processor[k].load_vector[l]
[2];
                    printf("load on processor %d = %d\n",k,l3[processor[l].server]);
                }
              }
                    }


          for(k = 1; k <= nump; k++){
              for(l = 1; l <= nump; l++){
                if( (int)clock == recent_update[k][l] + period-interval){
                    processor[k].load_vector[l][2] = l3[processor[l].server];
                }
              }
          }

            if(evl == 0){
                    event = 8;
            }
            else{

            cause(&event, &i);
            p = &task[i];
            if(TRACE == 2){
              printf("event no = %d and task no = %d\n",event,i);
            }
            if(event == 6)
               p->finish_time = (int)time();
```

```
}
if(event==5 && scheduled[i]==0)
        event=7;
if(TRACE==2)
  printf("clock = %f\n",clock);
switch(event)
{

case 1:

        if(lb_scheduled[i]!=1){
            select=p->processor;
        }
        else{
            select=p->processor;

        }
        if(request(balancer[select],i,1)!=0){

             lb_scheduled[p->id]=1;
             p->processor=select;
            if(TRACE==2)
             printf("balancer[%d] q-
length=%d\n",select,l3[balancer[select]]);

        }
        else{
            lb_scheduled[p->id]=1;
            lb_released[p->id]=1;
              if(lb_start[p->id] <= (int)clock){
                    lb_start[p->id]=(int)clock;
              }
            p->processor=select;
            schedule(2,clock,i);
        }
        break;

case 2:
         lb_releas=0;
         for(k=1; k<=numt; k++){
           if(lb_scheduled[k]==1 && lb_released[k]==1){
              if((int)clock >= (lb_start[k]+lb_exec)){
                    schedule(3,clock,k);
                    lb_scheduled[k]=0;
                    lb_releas=1;
              }
           }
         }
         if(lb_scheduled[i]==1){
              schedule(2,clock,i);
              if(lb_releas!=1)
                 clock=clock+1.0;
                 ++lock;
          }
         break;

        if((int)clock >= (lb_start[p->id]+lb_exec)){
          release(balancer[p->processor],i);
```

```
                    schedule(3,0.0,i);
                  }
                  else{
                    schedule(2,0.0,i);
                  }
                  clock=clock+1.0;
                  break;

        case 3:   release(balancer[p->processor],i);
                  schedule(4,clock,i);
                  select=p->processor;
                  break;

        case 4:
                /* Check the load on all processors and based on the
                   probabilities assign the task to the lightly loaded
                   processor     */

        temp=0;
        if(scheduled[i]!=1){
                    if(TRACE==2)
                      printf("task %d initiated on processor
%d\n",i,task[i].processor);

                      origin=task[i].processor;
                      task[i].status=0;
                      idle=origin;
                for(j=0; j<=nump; j++){
                      if(idle != j){
                        if(P[idle][j] > 0.0){
                              max_P=P[idle][j];
                              temp=j;
                              break;
                        }
                      }
                if((j==nump) && (try >= 10)){
                      try=0;
                      for(k=1; k<=nump; k++){
                        for(l=1; l<=nump; l++){
                              P[k][l]=old_P[k][l];
                        }
                      }
                      for(k=0; k<=nump; k++){
                        if(idle != k){
                              if(P[idle][k] > 0){
                                max_P=P[idle][k];
                                temp=k;
                                break;
                              }
                         }
                      }
                }
        }


                for(k=0; k<=nump; k++){
                  if(idle != k){
                      if(P[idle][k] > max_P){
```

```
                        max_P=P[idle][k];
                        temp=k;
                    }
                }
            }

            if((temp != 0) && (temp != origin)){
                transfer[idle][temp]++;
                transfer_pack.type=1;
                transfer_pack.src_processor=idle;
                transmitted++;
                transfer_pack.seq=transmitted;
                transfer_pack.msg_len=task[i].x_time[1];
                for(k=1; k<= processor[idle].linknum; k++){
                  if(processor[idle].link_id[k].end2==temp)
                        link=k;
                }
/*              printf("temp = %d\n",temp);
                printf("i and j are %d and %d\n",idle,link); */
                que[idle][link]=enque(idle,link,0,&transfer_pack);
                idle=temp;
                task[i].status=1;
                C[idle][temp]=160000/processor[idle].link_id[link].lnk_cap
;
                task[i].x_time[1]=task[i].x_time[1]+(160000/processor[id
le].link_id[link].lnk_cap);
                task[i].processor=idle;
                P[origin][temp]=P[origin][temp]-0.1;
            }
            else{
                P[origin][0]=P[origin][0]-0.1;
            }
        }
/*              printf("inside idle = %d\n",idle);      */


/*          printf("idle=%d\n",idle); */

            if(request(processor[idle].server,i,1)!=0){
                load[idle]=load[idle]+task[i].x_time[1];
                scheduled[p->id]=1;
                task[i].processor=idle;
/*          printf("All the processors are busy:\n");
            printf("processor[%d] q-
length=%d\n",idle,l3[processor[idle].server]);          */

            }
            else{
                scheduled[p->id]=1;
                released[p->id]=1;
                if(p->st_time <= (int)clock){
                    p->st_time=(int)clock;
                }
                p->processor=idle;
                load[idle] += task[i].x_time[1];
                running[idle]=busy;
                if(TRACE==2)
                  printf("process start time = %d\n",p->st_time);
```

```
                    schedule(5,clock,i);
                }
            break;

        case 5:
            if(TRACE==2){
             printf("st_time = %d\n",p->st_time);
             printf("x_time = %d\n",p->x_time[1]);
            }
             releas=0;
             for(k=1; k<=numt; k++){
               if(scheduled[k]==1 && released[k]==1){
                   if ((task[k].st_time+task[k].x_time[1]) <= (int)clock){
                schedule(6,0.0,k);
                        scheduled[k]=0;
                        releas=1;
                   }
               }
             }
             if(scheduled[i]==1){
                   schedule(5,clock,i);
/*                 schedule(2,0.0,i);                    */
                   if(releas!=1){
                      clock=clock+1.0;
                      ++lock;
                   }
             }
            break;


        case 6:  release(processor[p->processor].server,i);
/*               printf("%d processor is released\n",p->processor); */
                 idle=p->processor;
                 load[idle] -= task[i].x_time[1];

                 order[ord]=i;
                 ord++;
/*
                 for(k=1; k<=numt; k++)
                   waiting[k][i]=0;
*/
                 running[p->processor]=0;
                 loop--;
                 break;

        case 7:
                 if(TRACE==2)
                   printf("process %d already finished\n",i);
                 break;
        case 8:  clock=clock+1.0;
                 ++lock;
                 break;


/*      default: iter--;
                 clock++;
                 break;                          */
        }
```

```
        }
        for(j=1;j<=numt;j++){
          if(TRACE==2)
          printf("task %d response time = %d\n",j,(task[j].finish_time-
task_submit_time[j]));
        total_res = total_res+(task[j].finish_time-task_submit_time[j]);
        }
        total_response=total_response+(total_res/numt);
/*
          printf("          proc1      proc2      proc3    proc4      proc5\n");
        for(j=1;j<=numt;j++){
          printf("task %d        ",j);
          switch(task[j].processor)
          {

                case 1: printf("    %d\n",task[j].x_time[1]);
                        break;
                case 2: printf("                %d\n",task[j].x_time[1]);
                        break;
                case 3: printf("                  %d\n",task[j].x_time[1]);
                        break;
                case 4: printf("
%d\n",task[j].x_time[1]);
                        break;
                case 5: printf("
%d\n",task[j].x_time[1]);
                        break;
           }
      }
*/
        for (i=1; i<=nump; i++){
                for (j=1; j<=nump; j++){
                        if((i != j) && (transfer[i][j]>0))
                                fprintf(fd,"tasks transferred from %d to %d =
%d\n",i,j,transfer[i][j]);
                }
        }
        for (i=1; i<=nump; i++){
                for (j=1; j<=processor[i].linknum; j++){
                        ave_util[i][j]=ave_util[i][j]+util_time[i][j];
                }
        }


      fprintf(fd,"Mean Task Size = %d\n",(SUM_EXEC/numt));
      fprintf(fd,"total response = %d\n",total_response);

        fprintf(fd,"All the processes executed in %f time units\n",clock);
/*      printf("total communication cost is %f time units\n",total_comm); */
        fprintf(fd,"completion time = %f\n",clock);
        reportf();
}

      fprintf(fd,"Mean Response Time = %d\n",total_response/sample);
        for (i=1; i<=nump; i++){
                for (j=1; j<=processor[i].linknum; j++){
          fprintf(fd,"percent link utilization[%d][%d] =
%f\n",i,processor[i].link_id[j].end2,(((double)ave_util[i][j]/sample)*100.0)/(int)clock);
```

```
                }
        }

        for(i=40; i<=100; i++){
                fprintf(fd1,"%f   %7.4f\n",(float)((i-39)*50),rms[i]/sample);
        }

}

void    weight()
{
        int     i,j;
        int     sum[nump+1];
        int     W[nump+1][nump+1];

        for(i=1; i<=nump; i++){
          sum[i]=0;
          for(j=1; j<=nump; j++){
                sum[i]=sum[i]+l3[processor[j].server];
           }
           q_mean[i]=(int)ceil((double)sum[i]/nump);
/*         printf("q_mean on processor %d = %d\n",i,q_mean[i]);   */
        }
}

double unbalance()
{
        int     i,j;
        double  sum,phi;

        sum=0.0;
        for(i=1; i<=nump; i++){
                for(j=1; j<=nump; j++){
                  if(j > i){
                        sum=sum+(((10*l3[processor[i].server])-
(10*l3[processor[j].server]))*((10*l3[processor[i].server])-(10*l3[processor[j].server])));
                  }
                }
        }
        phi=sqrt(sum/(double)((nump*(nump-1))/2));
        return(phi);
}
```

```
/*
************************
*                      *
* Dated: June 25,92.   *
* Written By: Wajeeh Butt *
************************

This program impements the joint membership cluster load balancing strategy.
It implements the queueing model for a multiple server system. Each host is
modelled as a M/M/1 queue.

The instantaneous and the average queue length values are estimated for each
host and exchanged amongst all the hosts in the system.
Initially the network is partitioned into equal size host groups. Based on
the load information estimates available at each host, a lightly loaded host
in a lightly loaded host group is offered the joint membership from a
heavily loaded hosts. The details of the Algorithm are described in section
7.6.
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "smpl.h"
#include "sched.h"
#define busy 1
#define cluster_sum 4

extern struct token task[numt+1];   /* array of tasks, where each task is a
                                       structure                */
extern int      evl;
extern struct node  processor[nump+1];          /* array of the processors  */
extern struct group cluster[cluster_sum+1];     /* different clusters (host groups)
devide the whole network     */
int     running[nump+1];
extern int  LT;
extern int  LQ;
extern int  SUM_EXEC;
extern double mu[nump+1];           /* Service time of each processor is stored to
determine the unfinished work for finding the load imbalance  */
extern int      ready[numt+1],l1[4024],l3[4024];
extern real     l5[4024];           /* saves the values of queue-length and time
product used to calculate the average queue-length  */
extern int  hop[nump+1][nump+1];
extern real clock;                  /* simulation time   */
extern short    jf;
extern int      managers[cluster_sum+1];
extern double transfer_delay[nump+1][nump+1];       /* A value used to
simulate the task transfer delay between two hosts.  */

int    period=0;        /* used to send the load information updates
                           periodically within the group   */
int    net_period=0;    /* used to send the load information updates
                           periodically among the managers   */
int     MIG[nump+1][nump+1];
```

```
double rate[nump+1][nump+1];        /* used to determine the probability for the
                                       next task arrival  */
FILE    *fd;
int     group_interval=200;   /* load update interval used for hosts within a host group
*/
int     net_interval=600;       /* load update interval used for host group managers for
load balancing amongst the host groups */

int     sample;
int     bias=1;
int     net_bias=3;
int     q_mean[cluster_sum];
int     q_mean_network;
double  lambda;
double  total_res;
int     task_submit_time[numt+1];           /* To calculate the response time
                                               of a task      */
void    weight();
void    avgQ();
double  unbalance();
int     load[nump+1];
double  avg_load[nump+1];
double  avg_q[cluster_sum+1];
double  avg_Q;
int     done[numt+1];
double  total_response;
int     event;
int     recent_update[nump+1][nump+1];

int  main(argc,argv)

int     argc;
char    *argv[];

{
        int     token,numtask,numproc;
/*      int     flag1[nump+1],flag[numt+1];  */
        int     flag;
        int     iter;
        int     number,number1,donor,acceptor;
        int     ex_value = 0;
        int     pr_value = 0;
        int     releas=0;
        int     lb_releas=0;
        int     sim_time;
        int     D_index[cluster_sum+1],A_index[cluster_sum+1];
        double  rms[1000];
        int     lock;
        int     hl;     /* keeps count of the heavily loaded processors        */
        double  ex_clock=0;
        int     pr_clock=0;
        int     net_clock=0;
        int     temp,next,recv,set,max_com,max_finish,min_hop;
        int     save,save1,save2;
        int     RECV[nump+1];
        int     total_comm,balancer[nump+1];   /* com_delay[numt+1]  */
        int     greater,pr_no,idle,minim,select,origin,ready_count;
        int     i,j,k,l,m,s,loop,count,diff,ord=1;
```

```
        double  min,minload,max;
        int     lb_start[numt+1];

        int     task_trak[numt+1],pr_order[numt+1],order[numt+1],released[numt+1],
lb_released[numt+1],scheduled[numt+1],lb_scheduled[numt+1];
        TASKS  *p;
        FILE    *fd1,*fd2, *fopen();


        if(argc!=3){
                printf("Usage: find pattern\n");
                exit(0);
        }
        while(--argc>0){
            if(argc==2){
/*              printf("Total no. of tasks for current simulation = %s\n",argv[argc]); */
                numtask=(int)argv[1];
            }
            if(argc==1){
/*              printf("Total no. of processors for current simulation =
%s\n",argv[argc]); */
                numproc=(int)argv[argc];
            }
        }
        if((fd1=fopen("inbalancejnt5","w"))==NULL){
                fprintf(stderr,"Cannot open the file 'inbalance' to write\n");
                exit(1);
        }
        if((fd2=fopen("responsejnt5","w"))==NULL){
                fprintf(stderr,"Cannot open the file 'response' to write\n");
                exit(1);
        }
/*
        if((fd=fopen("stat4x4m","w"))==NULL){
                fprintf(stderr,"Cannot open the file 'stat' to write\n");
                exit(1);
        }
*/
/*      printf("Do you want to set TRACE ON? Enter (1):");
      scanf("%d",&TRACE);   */
        TRACE=1;

  lambda=6.0;
   while(lambda <=  16){
   lambda=lambda+1.0;
   init();          /* Initialization routine for setting up the
                                simulation environment      */
    total_response=0.0;
    sample=0;
    for(i=1; i<=1000; i++)
        rms[i]=0.0;
    while(sample <= 10){
        sample++;
/*      jf=stream(sample);
        printf("seed = %d\n",stream(sample)); */
        iter=0;
        clock=0.0;
        period=0;
```

```
        ex_value=0;
        pr_value=0;
        releas=0;
        lb_releas=0;
        ex_clock=0;
        total_res=0.0;
        smpl(0,"scheduler model");
        reset();
        for(i=1; i<=numt+1; i++)
                released[i]=lb_released[i]=scheduled[i]=lb_scheduled[i]=0;

        for(i=1; i<=numt; i++){
                done[i]=0;
        }

        for(i=1; i<=nump; i++)
                for(j=1; j<=nump; j++)
                        MIG[i][j]=0;

        for(i=1; i<=nump; i++)
                RECV[i]=avg_load[i]=load[i]=0;
        for (i=1; i<=nump; i++)
                processor[i].server=facility("processor",1);  /* A single que-single
                                                        server facility  */
        for (i=1; i<=nump; i++)
                balancer[i]=facility("balancer",1);  /* A load balancing
                                component associated with each processor   */

/*      printf("all the tasks have been scheduled\n");  */
        period=group_interval;
        net_period=net_interval;
        loop=numt;
        while(loop)
        {
                if(TRACE==2){
                  printf("\n");
                  printf("\n");
                }
        for (i=1; i<=numt; i++){
         if(done[i]==0){
           if(((int)clock==task[i].arrival) && (scheduled[i]==0)){
            schedule(4,(real)(task[i].arrival),i);
            task_submit_time[i]=task[i].arrival;
            task[i].st_time=(int)clock;
/*          lb_start[i]=task[i].arrival;    */
            done[i]=1;
          }
         }
        }

/*  Each processor's average q-length    */

              for(k=1; k<=nump; k++){
/*                  printf("processor %d average q-length =
%f\n",k,l5[processor[k].server+1]/clock);   */
                }
              for(k=1; k<=nump; k++){
```

```
/*  Each processor's own load is updated immediately    */

                processor[k].load_vector[k][2] = l3[processor[k].server];
                }
                if(lock = = 30){
                        iter + +;
                        rms[iter] = rms[iter] + unbalance();
                        lock = 0;
                }


/* Based on the information exchange for the queue-length calculate the mean
   que_length for load balancing  */
        if(((int)clock = = (period-1)) && (clock ! = pr_value)){
                pr_value = clock;
                weight();
        }

        if(((int)clock = = (net_period-1)) && (clock ! = ex_value)){
                ex_value = clock;
                avgQ();

                donor = 0;
                acceptor = 0;
                for(i = 1; i < = cluster_sum; i + +){
                        D_index[i] = A_index[i] = 0;
                }

                for(i = 1; i < = cluster_sum; i + +){
                  if(avg_q[i] > = avg_Q + 1){
                        donor + +;
/*                      printf("donor = %d\n",donor);   */
                        D_index[donor] = i;
/*                      printf("donor cluster = %d\n",D_index[donor]);  */
                  }
                  if(avg_q[i] < = avg_Q-1){
                    if(cluster[i].hostnum > cluster_sum){
                        for(j = cluster_sum + 1; j < = cluster[i].hostnum; j + +){
                          cluster[i].hostnum--;
                          cluster[i].host[j] = 0;
                        }
                    }
                        acceptor + +;
/*                      printf("acceptor = %d\n",acceptor);   */
                        A_index[acceptor] = i;
/*                      printf("acceptor cluster = %d\n",A_index[acceptor]);  */
                  }
                }
                for(i = 1; i < = donor; i + +){
                  if(D_index[i] = = 0)
                        break;
                  else{
                   save = D_index[i];
                   for(j = 1; j < = acceptor; j + +){
                        if(A_index[j] = = 0)
                                break;
                        else{
                          minload = avg_load[cluster[A_index[j]].host[1]];
```

```
                        for(k=1; k<=cluster[A_index[j]].hostnum; k++){
                        if(cluster[A_index[j]].host[k] != cluster[A_index[j]].manager){
                          if(avg_load[cluster[A_index[j]].host[k]]<minload){
                            minload=avg_load[cluster[A_index[j]].host[k]];
                            save2=cluster[A_index[j]].host[k];
                          }
                        }
                        }
                        recv=0;
                        for(k=1; k<=cluster[save].hostnum; k++){
                          if(cluster[save].host[k] == save2){
                              recv=1;
                              break;
                          }
                        }
                          if(recv==0){
                               cluster[save].hostnum++;
                               cluster[save].host[cluster[save].hostnum]=save2;
                          }
                        for(k=1; k<=cluster_sum; k++){

transfer_delay[cluster[save].host[k]][save2]=transfer_delay[save2][cluster[save].host[k]]
=2.0;
                                }
                            }
                            }
/*                  for(j=1; j<=cluster_sum; j++){
                                printf("cluster[%d].host=%d\n",j,cluster[j].hostnum);
                    }
*/

          }
          }
      }


/* performs the dequeing and transfer of jobs between two managers  */

  if(clock != ex_clock){
        ex_clock=clock;
        for(j=1; j<=nump; j++){
           for(k=1; k<=nump; k++){
                set=0;
                if(MIG[j][k]>0){
/*                      printf("mig[%d][%d]=%d\n",j,k,MIG[j][k]);       */
                        token=dq(processor[j].server);
                        MIG[j][k]--;
/*                      flag[token]=1; */
                        flag=1;
                        set=1;
                        break;
                }
           }
        if(set==1)
           break;
        }
   }
```

```
/*  Check the clock value and broadcast the load information of each processor
      load  */


            if ((int)clock = = period){
/*                  dist_monitor();
        if(clock > 2000.0){                         */
          for(l=1; l<=cluster_sum; l++){
                hl=0;
                for(j=1; j<=cluster[l].hostnum; j++){
                        if(l3[processor[cluster[l].host[j]].server] > q_mean[l])
                                hl++;
                }
                for(j=1; j<=cluster[l].hostnum; j++){
                  if((l3[processor[cluster[l].host[j]].server] - q_mean[l]) > bias){
/*                  if(j!=k){  */
                        for(k=1; k<=cluster[l].hostnum; k++){
                          if((l3[processor[cluster[l].host[k]].server]) < (q_mean[l]-bias)){
/* -bias   */
                                MIG[cluster[l].host[j]][cluster[l].host[k]]=((l3[processor[cl
uster[l].host[j]].server] - (q_mean[l]+bias)));
                                break;
                          }
                        }
                  }
                }
                }
/*              if(((int)clock >= period) && ((int)clock !=pr_clock)){
                pr_clock=(int)clock;
                for(j=1; j<=cluster_sum; j++){
                  for(k=1; k<=cluster[j].hostnum; k++){
                   temp=0;
                   if(q_mean[j] >= 1){

while((temp=cluster[j].host[random(1,cluster[j].hostnum)])==cluster[j].host[k]);
                     if((l3[processor[cluster[j].host[k]].server] - q_mean[j]) >=
(bias*2+MIG[cluster[j].host[k]][temp])){
                        MIG[cluster[j].host[k]][temp]=MIG[cluster[j].host[k]][temp]+1;
                          printf("%d tasks transferred to random processor
%d\n",MIG[cluster[j].host[k]][temp],temp);
                     }
                   }
                }
              }
             }
*/
            }

        if(clock == (period+group_interval-1))
                period=period+group_interval;

/* Save the old load information for calculating the rate of change of que   */

                        if(TRACE==2){
                  for(k=1; k<=nump; k++){
                    printf("LOAD TABLE FOR PROCESSOR %d\n",k);
                    for(l=1; l<=nump; l++){
```

```
                                processor[k].load_vector[l][1]=processor[k].load_vector[l]
[2];
                        printf("load on processor %d = %d\n",k,l3[processor[l].server]);
                    }
                }
                    }


        if ((int)clock == net_period){
            hl=0;
            for(l=1; l<=cluster_sum; l++){
                        if(l3[processor[cluster[l].manager].server] > q_mean_network)
                                hl++;
            }
            for(l=1; l<=cluster_sum; l++){
                if((l3[processor[cluster[l].manager].server] - q_mean_network) >
net_bias){
                    for(k=1; k<=cluster_sum; k++){
                        if((l3[processor[cluster[k].manager].server]) <
(q_mean_network-net_bias)){  /* -bias  */
                                MIG[cluster[l].manager][cluster[k].manager]=((l3[process
or[cluster[l].manager].server] - (q_mean_network+net_bias)));
                                break;
                        }
                    }
                }
            }
        }



/*
        if(((int)clock >= net_period) && ((int)clock != net_clock)){
                net_clock=(int)clock;
            for(j=1; j<=cluster_sum; j++){
                temp=0;
                if(q_mean_network >= 1){
                 while((temp=managers[random(1,4)])==managers[j]);
                  if((l3[processor[managers[j]].server] - q_mean_network) >=
(net_bias*4+MIG[managers[j]][temp])){
                        MIG[managers[j]][temp]=MIG[managers[j]][temp]+1;
                printf("%d tasks transferred to random manager
%d\n",MIG[managers[j]]][temp],temp);
                    }
                }
            }
        }
*/

        if(clock == (net_period+net_interval-1))
                net_period=net_period+net_interval;


                if(evl==0){
                        event=8;
                }
                else{
```

```
                cause(&event, &i);
                p=&task[i];
                if(TRACE==2){
                  printf("event no=%d and task no=%d\n",event,i);
                }
                if(event==6)
                   p->finish_time=(int)time();
                }
/*              if(event==5 && scheduled[i]==0)
                        event=7;                                */
                if(TRACE==2)
                 printf("clock = %f\n",clock);
                switch(event)
                {


                case 1:

                        if(lb_scheduled[i]!=1){
                                select=p->processor;
                        }
                        else{
                                select=p->processor;

                        }
/*                      printf("select = %d\n",select);        */
                        if(request(balancer[select],i,1)!=0){

                                lb_scheduled[p->id]=1;
                                p->processor=select;
                               if(TRACE==2)
                                printf("balancer[%d] q-
length=%d\n",select,l3[balancer[select]]);

                        }
                        else{
                                lb_scheduled[p->id]=1;
                                lb_released[p->id]=1;
                                  if(lb_start[p->id] <= (int)clock){
                                        lb_start[p->id]=(int)clock;
                                  }
                                p->processor=select;
                                schedule(2,clock,i);
                        }
                        break;

                case 2:
                         lb_releas=0;
                         for(k=1; k<=numt; k++){
                           if(lb_scheduled[k]==1 && lb_released[k]==1){
                              if((int)clock >= (lb_start[k]+lb_exec)){
                                        schedule(3,clock,k);
                                        lb_scheduled[k]=0;
                                        lb_releas=1;
                                }
                           }
                         }
                         if(lb_scheduled[i]==1){
```

```
                                schedule(2,clock,i);
/*                              schedule(5,clock,i);    */
                                if(lb_releas! = 1)
                                   clock = clock + 1.0;
                                    + + lock;
                             }
                            break;

/*
                           if((int)clock > = (lb_start[p->id] + lb_exec)){
                             release(balancer[p->processor],i);
                             schedule(3,0.0,i);
                           }
                           else{
                             schedule(2,0.0,i);
                           }
                           clock = clock + 1.0;
                           break;
*/

                 case 3:   release(balancer[p->processor],i);
                           schedule(4,clock,i);
                           select = p->processor;
                           break;

                 case 4:
                        /* Check the load on all processors and based on the
                           probabilities assign the task to the lightly loaded
                           processor    */


                 if(scheduled[i]! = 1){
                              if(TRACE = = 2)
                                printf("task %d initiated on processor
%d\n",i,task[i].processor);



                              origin = task[i].processor;
                              idle = origin;
                 }
/*                               printf("inside idle = %d\n",idle);       */
          /* Check for the upper threshold, If the current load exceeds the upper
             threshold then perform load balancing        */

                 if(scheduled[p->id] = = 1){
                        idle = p->processor;
/*                      if(clock > 2000.0){             */
                              for(k = 1; k < = nump; k + +){
                                 next = 0;
                                 if((MIG[idle][k] > 0)&& (flag = = 1)){
                                      p->st_time = p->st_time + transfer_delay[idle][k];
/*                                    MIG[idle][k]--;
                                      flag[p->id] = 0;  */
/*                                    printf("a task transferred from %d to
%d\n",idle,cluster[processor[idle].cluster].host[k]);          */
                                      idle = k;
                                      p->processor = idle;
```

```
                                    next=1;
                                    flag=0;
                                    break;
                                }
                            }
                    }

/*                          printf("idle=%d\n",idle);  */

                            if(request(processor[idle].server,i,1)!=0){
                                load[idle]=load[idle]+task[i].x_time[1];
                                scheduled[p->id]=1;
                                task[i].processor=idle;
/*                          printf("All the processors are busy:\n");
                            printf("processor[%d] q-
length=%d\n",idle,l3[processor[idle].server]); */

                            }
                            else{
                                scheduled[p->id]=1;
                                released[p->id]=1;
                                if(p->st_time <= (int)clock){
                                    p->st_time=(int)clock;
                                }
                                p->processor=idle;
                                load[idle] += task[i].x_time[1];
                                running[idle]=busy;
                                if(TRACE==2)
                                  printf("process start time = %d\n",p->st_time);
                                schedule(5,clock,i);
                            }
                        break;

                    case 5:
                        if(TRACE==2){
                          printf("st_time = %d\n",p->st_time);
                          printf("x_time = %d\n",p->x_time[1]);
                        }
                         releas=0;
                         for(k=1; k<=numt; k++){
                           if(scheduled[k]==1 && released[k]==1){
                              if ((task[k].st_time+task[k].x_time[1]) <= (int)clock){
                            schedule(6,0.0,k);
                                    scheduled[k]=0;
                                    releas=1;
                              }
                           }
                         }
                         if(scheduled[i]==1){
                              schedule(5,clock,i);
/*                            schedule(2,0.0,i);                */
                              if(releas!=1){
                                 clock=clock+1.0;
                                 ++lock;
                              }
                         }
                        break;
```

```
                case 6:   release(processor[p->processor].server,i);
/*                        printf("%d processor is released\n",p->processor); */
                          idle=p->processor;
                          load[idle] -= task[i].x_time[1];

                          order[ord]=i;
                          ord++;
/*
                          for(k=1; k<=numt; k++)
                            waiting[k][i]=0;
*/
                          running[p->processor]=0;
                          loop--;
                          break;

                case 7:
                          if(TRACE==2)
                            printf("process %d already finished\n",i);
                          break;
                case 8:   clock=clock+1.0;
                          ++lock;
                          break;


/*              default:  iter--;
                          clock++;
                          break;                          */
                }
        }
        for(j=1;j<=numt;j++){
          if(TRACE==2)
         printf("task %d response time = %d\n",j,(task[j].finish_time-
task_submit_time[j]));
       total_res = total_res+(double)(task[j].finish_time-task_submit_time[j]);
        }
        total_response=total_response+(double)(total_res/numt);
/*
          printf("          proc1      proc2      proc3    proc4      proc5\n");
        for(j=1;j<=numt;j++){
          printf("task %d        ",j);
          switch(task[j].processor)
          {

                case 1: printf("    %d\n",task[j].x_time[1]);
                        break;
                case 2: printf("                  %d\n",task[j].x_time[1]);
                        break;
                case 3: printf("                    %d\n",task[j].x_time[1]);
                        break;
                case 4: printf("
%d\n",task[j].x_time[1]);
                        break;
                case 5: printf("
%d\n",task[j].x_time[1]);
                        break;
          }
      }
```

```
*/
/*      fprintf(fd,"Mean Task Size = %d\n",(SUM_EXEC/numt));
      fprintf(fd,"total response = %f\n",total_response);

        fprintf(fd,"All the processes executed in %f time units\n",clock); */
/*      printf("total communication cost is %f time units\n",total_comm); */
/*      fprintf(fd,"completion time = %f\n",clock); */
        reportf();
  }

/*      fprintf(fd,"Mean Response Time = %f\n",total_response/sample); */

        fprintf(fd2,"%f   %7.4f\n",lambda,total_response/sample);

   if(lambda==13){
        for(i=20; i<=99; i++){
                fprintf(fd1,"%f   %7.4f\n",((float)(i-19)*30),rms[i]/sample);
        }
    }
 }

}

void    weight()
{
        int     i,j,temp,sum[cluster_sum+1];

        temp=0;
        for(i=1; i<=cluster_sum; i++){
          sum[i]=0;
          for(j=1; j<=cluster[i].hostnum; j++){
                sum[i]=sum[i]+l3[processor[cluster[i].host[j]].server];
           }
           q_mean[i]=(int)ceil((double)sum[i]/cluster[i].hostnum);
/*         printf("q_mean for cluster%d = %d\n",i,q_mean[i]);          */
           temp=temp+q_mean[i];
        }
        q_mean_network=(int)ceil((double)temp/cluster_sum);
/*      printf("q_mean_network = %d\n",q_mean_network);     */
}

void    avgQ()
{

        int     i,j;
        double  temp,sum[cluster_sum+1];

        for(i=1; i<=cluster_sum; i++){
          sum[i]=0;
          for(j=1; j<=cluster[i].hostnum; j++){
                sum[i]=sum[i]+(l5[processor[cluster[i].host[j]].server+1]/clock);
           }
          avg_q[i]=sum[i]/cluster_sum;
/*        printf("avg_q[%d] = %f\n",i,avg_q[i]);        */
        }

        for(i=1; i<=nump; i++){
                avg_load[i]=(l5[processor[i].server+1]/clock);
```

```
/*              printf("average q on proc %d = %f\n",i,avg_load[i]);  */
                temp = temp + avg_load[i];
        }
        avg_Q = temp/nump;
/*      printf("average Q for the network = %f\n",avg_Q);  */
        for(i = 1; i< = nump; i + +){
                l5[processor[i].server + 1] = 0.0;
        }
}

double unbalance()
{
        int     i,j;
        double  sum,phi;

        sum = 0.0;
        for(i = 1; i< = nump; i + +){
                for(j = 1; j< = nump; j + +){
                  if(j > i){
                        sum = sum + (((mu[i]*l3[processor[i].server])-
(mu[j]*l3[processor[j].server]))*((mu[i]*l3[processor[i].server])-
(mu[j]*l3[processor[j].server])));
                  }
                }
        }
        phi = sqrt(sum/(double)((nump*(nump-1))/2));
        return(phi);
}
```