USING IDLE WORKSTATIONS FOR DISTRIBUTED COMPUTING

A Thesis Presented to

The Faculty of the

Fritz J. and Dolores H. Russ

College of Engineering and Technology

Ohio University

In Partial Fulfillment

of the Requirement for the Degree

Master of Science

by
Anand Kore

November, 1998

ACKNOWLEDGEMENTS

I wish to thank my thesis advisor Dr. Mehmet Celenk for his scholarly guidance and efforts to completle this work.

My sincere thanks to Dr. Shawn Ostermann for providing and helping to use *tcpdump* and *tcptrace* programs. Special thanks to Mr. John Tysco for timely help to install these softwares on *homer*.

I am also grateful to my thesis committee members: Dr. Voula Georgopoulos, Dr. Jeffrey Giesey, and Dr. Martin Kordesch for reviewing my thesis manuscript.

None of this work would ever been possible without the financial support from my sponsors. I gratefully acknowledge the assistance of the staff at Consulate General of India, New York and Ministry of Welfare, Govt. of India, New Delhi, to finish my MS program at Ohio University.

I must also thank my wife Sushma and my parents for their constant encouragement, love, and moral support.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

Network based computing is gaining popularity as a new parallel processing paradigm. The term network of workstations (NOWs) is defined for such parallel systems which consists of a group of powerful workstations connected by a high-speed network. This resource network can be used by a distributed application to reach the power of one large computer.

A distributed computing system may be defined as one in which multiple autonomous processors, possibly of different kinds, are interconnected by a communication network to interact in a co-operative way to achieve an overall goal. To achieve the specified goal, the systems are logically integrated in a varying degrees by a distributed operating system. Figure 1.1 illustrates a logical view of distributed systems.

Although speed-up of processing through concurrent execution is one of the main objectives of distributed computing systems, this may not be always possible because of the distribution of physical resources and the dependency between processing steps. In fact, in many distributed systems, one of the key objectives is to hide the physical distribution from the user and to provide a physical transparency.

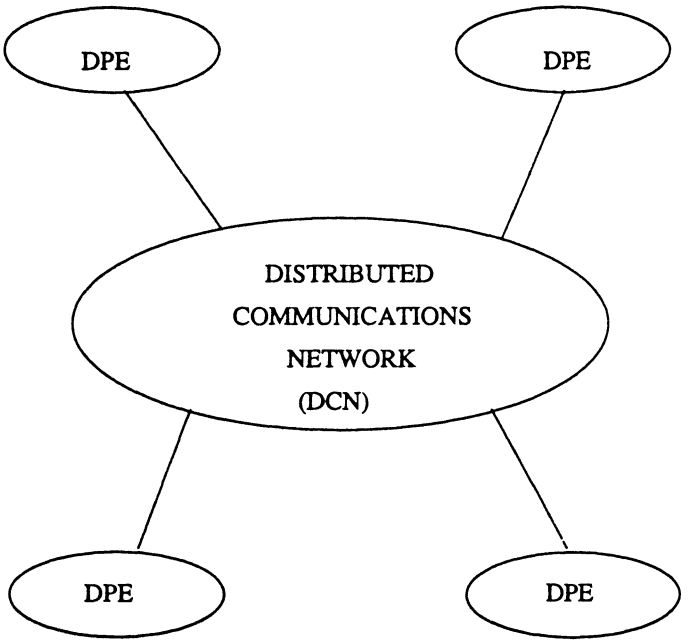


Figure 1.1: A logical view of distributed computing system architecture including distributed communications network (DCN) and distributed processing element (DPE).

A distributed computing system supports an arbitrary number of application processes. It has a modular design of distributed architecture with a messaging facility through a shared communication system and some kind of system wide control. The potential benefits that can be derived from a distributed system include improved performance, increased reliability and availability, local control, and reduced cost.

To achieve all these benefits on an existing local area network (LAN) architecture is a challanging objective. The reason is that LAN technology was not developed for parallel processing and communication overheads among workstations are still high. This is a major bottleneck on obtaining performance gain in NOWs. Inspite of this constraint, it is an attractive alternative for applications where a task can be divided into subtasks executing on separate machines.

A distributed system of NOWs uses a collection of workstation as a single integrated system. It is composed of number of autonomous processors, storage devices, and databases which interactively co-operate to achieve a common goal. Information exchange, interaction and co-ordination among various processes are achieved by communication network.

There is a number of challenges in harnessing the potential power of NOWs. This includes distributing parallel jobs in the network by keeping them transparent to both users and processes, minimizing the communication overhead involved in the message passing among workstations, and designing scheduler for the optimal load balancing.

The main objective of this research is to explore NOWs as a distributed system and to evaluate the performance for distributed computation. We will use LAN

as a loosely coupled system and study communication and scheduling oveheads to determine wheather or not if the NOWs are suitable for distributed computing.

1.1 Current Technology and NOWs

The current workstations are extremely powerful. They provide about one third (in terms of MIPS) the performance of a Cray C90 processor. Even though processors are becoming faster, the overall performance will not be any better unless the I/O performance increases. In this respect, the advantages of NOWs are making the system resources such as memory, disks, and processors available to the program in abundance. NOWs support a high degree of parallel computing in an everyday computing scenario. They provide processors with high sustained floating-point performance capability. They also provide networks with bandwidth that scales with the number of processors, parallel I/O, and low overhead communication, thus, making the environment suitable for parallel computing.

1.2 Related Work

A significant research effort has been devoted to the development of systems which allow for remote execution of jobs in a network of workstations. Most of the research work has been done in the following areas of parallel computation: 1) Development of distributed operating system [1, 2], 2) design of parallel programming languages [3, 4], and 3)devising a model or algorithm to overcome the load balancing problem and communication overhead [5, 6].

In their recent paper, Cabillic and Puaut [7] described design, implementation, and performance evaluation of Stardust, an environment for parallel programming on

networks of hetrogeneous systems. Work has also been done to develope a mechanism to share processing power in the network of workstations while taking into account local user and remote processes. A system called Butler has been designed by Nichols [8] which allows user to execute jobs on remote workstations. Butler system also allows to execute interactive shells on remote machines. However in this system, a remote process is terminated as soon as the local user reclaims the station.

Parallel Virtual Machine (PVM) [9] is a software which is a collaborative venture of Oak Ridge National Laboratory, University of Tennessee, Emory University, and Carnigie Mellon University. It supports hetrogenous network computing by providing a unified framework for developing parallel programs in an efficient and straightforward manner using the exisiting hardware. PVM provides the basic low level library functions to enable programmers to develop parallel programs. Although PVM is widely accepted, it neither attempts to create a new model of computation nor create a new language for describing parallel computation.

Distributed Automated Workload balancinG System (DAWGS) developed by Clark and McMillin [10] allows users to send programs to remote machines for execution. In DAWAGS, a distributed scheduler determines the machine the process can be run. The major drawback of DAWGS is that it does not support the parallel processing mechanism.

The V system developed by Cheriton [11] utilizes the state information to convey expected CPU and memory utilization to remote machines. The CPU utilization is measured by running a background process which periodically increments a counter. The counter is polled to determine the availability of CPU time.

Multimethod communication for remote service request has been implemented by Kesselman et al. [12]. These mechanisms have been implemented in a Nexus multithreaded runtime system. The performance characteristics of multimethod communication and Nexus-based MPI (Message Passing Interface) implementation have been illustrated.

Research on synchronization aspects of distributed computing has been presented by Reidl et al. [13]. They used Circulating Active Barrier (CAB) hardware mechanism for synchronising multiple processing elements (PEs). They have demonstrated that the synchronization times can be under $1\mu s$ for as many as 4,096 PEs using multiprocessor workstations or 1,024 single-processor workstations.

1.3 Research Objective

Although significant research has been done in distributed computing and remote execution of jobs on idle workstations, little work has been reported in the area of performance evaluation of NOWs for distributed computing applications while taking into account limitations and constraints of network environment in existing systems [22].

In this respect, the focus of this research is mainly on achieving high throuhput and equal load sharing strategy while reducing the communication overhead for message passing in an existing NOWs. Following issues have been focused in this regard:

1. Design of a dynamic scheduling algorithm in order to keep remote task execution transparent to the users of remote machines.

2. Development of an algorithm to exchange state information among the stations in a network.

3. Mesurement of communication and scheduling delays in a LAN for remote task execution.

4. Performance evaluation of the LAN for parallel task execution.

In the following chapters we describe how these goals have been achieved. In Chapter 2, we discuss the important issues in distributed computing including communication and load distribution issues. We present two important load distribution mechanisms; namely, static load distribution and dynamic load distribution. In Chapter 3, we describe the scheduling algorithm and discuss a method for measuring workstation load and finding idle remote machines. Task migration issue is also discussed in detail. Chapter 4 includes the therotical speedup analysis and explains how the scheduling and communication delays affect the system performance. In Chapter 5, we describe the design of client and server for distibuted computing of a matrix multiplication task in the network. In Chapter 6, we present the results obtained in this research and discuss how the speedup is dependent on different communication and scheduling parameters. Finally, in Chapter 7, conclusions and the topics of further research are presented.

Chapter 2

ISSUES IN DISTRIBUTED COMPUTING

A collection of physically distributed computers (nodes) interconnetced by a communication network is the simplest architecture of distributed system. Each node may be connected to several peripheral devices and information exchange is supported by network. The performance of a network based distributed computing system is degraded mainly because of communication and load scheduling overheads. Communication overhead is mainly due to the time required for information exchange among the workstations. Scheduling overhead is caused by the processing time of the algorithm which is responsible for distributing tasks among processors in the network. In the remaining part of this chapter we discuss these and other performance related issues in detail.

2.1 Using Idle Workstations

Finding idle workstations in a network has been a topic of considerable research. It has been shown [14, 15] that even at the peak time in the middle of a working day, as many as 30 percent of workstations are idle. Finding idle workstations and runing the remote processes transparently are the key issues. A number of schemes has been proposed for using these idle workstations [7, 8].

The simplest approach to use idle remote machines is by means of running *rsh*[1] program [24]. *rsh* runs specified command on a specified remote machine. *rsh* is used in LAN and supported by UNIX. Although widley used, this program has serious flaws. First, the user has to tell which machine to use. Second, the program executes in the environment of remote machines, which is different from local environment. Third, the executed remote process continues to run even if the owner of the idle machine logs on. Considerable attention has been focused on solving these problems.

2.2 Workstation Allocation Issues

A number of scheduling algorithms have been proposed to allocate idle workstations in a network [17]. These algorithms are developed based on two basic approches: Deterministic versus heuristic algorithms and centralized versus decentralized algorithms.

Deterministic algorithms are suitable when we know about the process behaviour in advance. Heuristic algorithms are used when the load is completely unpredictable. The most popular ones are centralized and decentralized algorithms. In a centralized scheduling scheme, a coordinator node keeps the state information about all the nodes in the network. It has the list of idle workstations and the description of the jobs demanding service. The basic model of a centralized scheduling service involves machines periodically sending status update messages and clients sending remote execution requests to a central server. In this scheme, machines advertise their load information.

---

[1] The general syntax of rsh program is "*rsh* remote machine, command."

In the decentralized scheme, each node does the searching on its own for idle or lightly loaded node. This scheme is used to reduce the scheduling complexity. The basic model of this type of scheduling system is one in which a client interested in obtaining a server sends a multicast query requesting current state information. The client receives replies back from all available machines and selects the best machine from that set. The best machine could be the one with minimum load. Though this scheme is less efficient, it is not subject to failure if a particular node crashes. Hence we will follow this scheme. The pseudo codes for the client and server programs in the decentralized scheme are given in Fig.2.1 and Fig.2.3, respectively. Flowcharts of these programs are also illustrated in Fig.2.2 and Fig.2.4.

```
int ExecuteProgram(program)
programDes * program; /* data structure that describes a
                                    program to be executed */
{
  ......       /* variable declarations */

/* send multicast message requesting remote machine status */

requestMsg.requestCode = RemoteMachineStatusRequest;
replyId = Send(RemoteMachineGroupId, &requestMsg);
bestLoad = requestMsg.RemoteMachineLoad;   /* get the load */
bestRemoteMachine = replyId;           /*pick remote machine */

/* Loop receiving reply messages until time out */

replyId = GetNextReply(&requestMsg, RemoteMachineGroupId);
while (replyId ! = Timeout){         /*calculate load*/
    if (requestMsg.RemoteMachineLoad < bestLoad){
       bestLoad = requestMsg.RemoteMachineLoad;
       bestRemoteMachine = replyId;
              }
          replyId=GetNextReply(&requestMsg, RemoteMachineGroupId);
            }
  rc = RequestProgramLoad(program, bestRemoteMachine);
  return(rc);
 }
```

Figure 2.1: Pseudo code for client program.

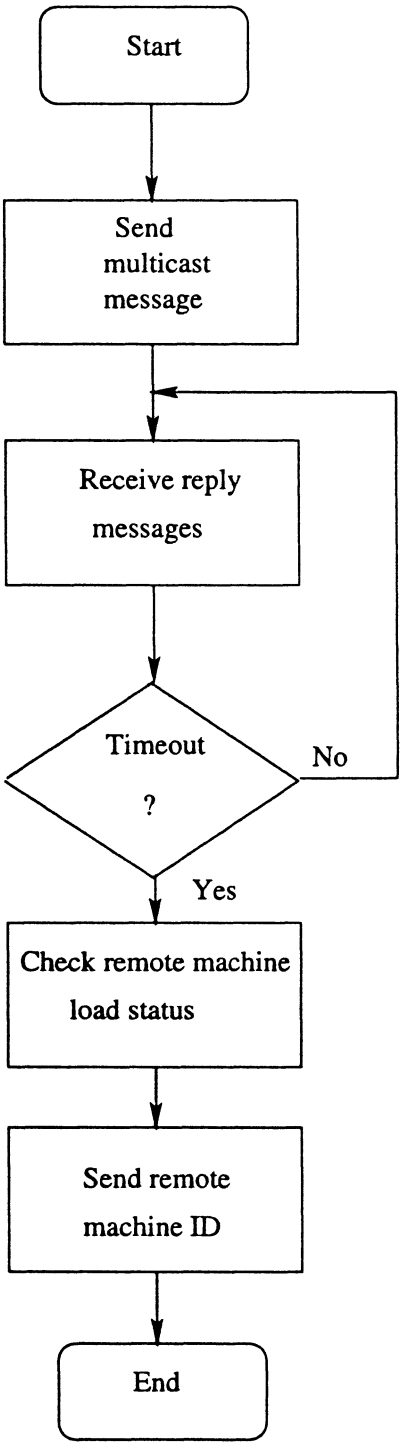


Figure 2.2: Flow chart of pseudo code for client program.

```
ServerProcess()
{
....                        /* variable declaration */

JoinMulticastGroup(RemoteMachineMulticastGroupId);
CreateThread(LoadTrackingThread);
while (FOREVER){
senderId = Receive(&requestMsg); /*get sendre machin's ID */
requestMsg.returnCode = OK;
avgLoad = ComputeAverageLoad();  /*Calculate average load */
requestMsg.RemoteMachineLoad = avgLoad;
Reply(senderId, &requestMsg);    /* Send average load */
}
}

LoadTrackingThread()
{
while(FOREVER){
Sleep (QUERY_TIME_PERIOD);
QueryHostState();               /*get remote machine's state */
        }
}
```

Figure 2.3: Pseudo code for server program.

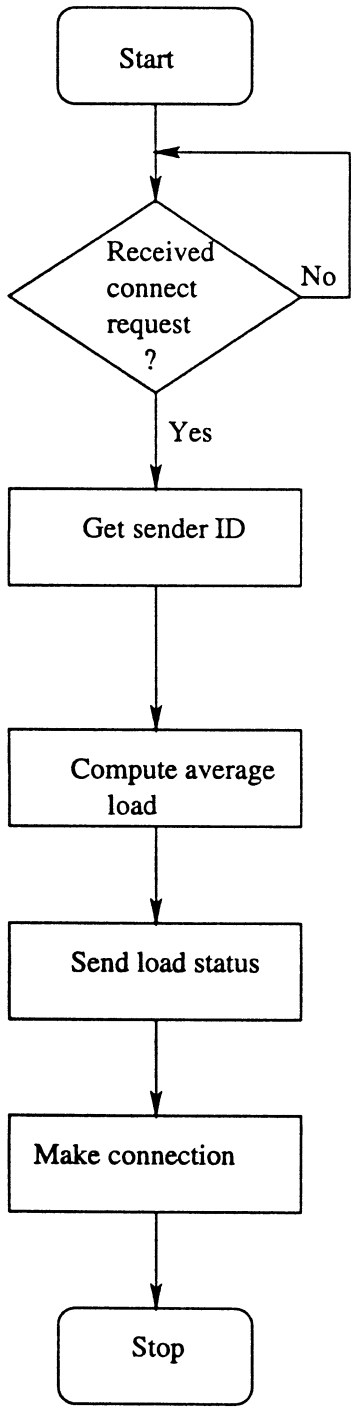


Figure 2.4: Flow chart of pseudo code for server program.

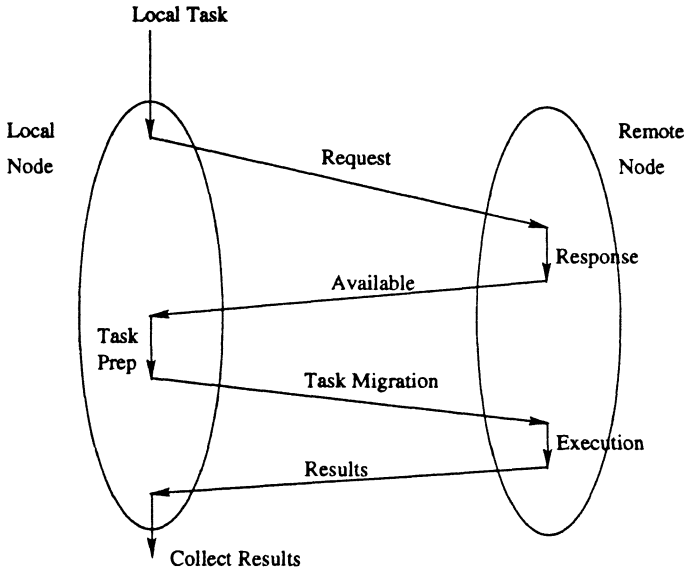


Figure 2.5: Communication overhead.

2.3 Communication Issues

In a distributed processing, coordination among the nodes is established by communication and synchronization, which constitute virtual communication among the application processes assigned to the individual processing components. At the application level, communication and synchronization are said to be virtual because there may or may not be a direct, physical communications channel between the processes. Since memory is not shared, interaction among the processes requires message passing. An interconnection network routes messages from a source process to a destination process. As shown in Fig.2.5, the interprocess communication by exchanging messages results in a communication overhead.

This communication overhead can be a sizable fraction of the total time required to execute a task. The system performance can be significantally degraded by the communication overhead. The delay of a packet crossing a link can be given as [26]

$$D = P + RL + Q \tag{2.1}$$

where P is the preparation and broadcasting time, R is the single bit transmission time, L is the packet length in bits, and Q is the queing time. These parameters are explained below.

Preparation Time: Time required to prepare information for transmission is called as communication preparation time.

Queing Time: The time it takes to wait for a packet in a queue before the transmission is called as queing time. This time also includes the time for the re-transmission in case of the packet errors.

Transmission Time: This is the time required for transmission of all the bits of the packet at sender site.

Broadcasting Time: This is the time between the end of transmission of the last bit of the packet at the transmitting processor and the reception of the last bit of the packet at the receiving processor.

From Eq. 2.1, we can see that any term can dominate the delay time. Thus, in a message passing system the communication overhead can be still higher than the computation time.

2.4 Load Distribution Issues

Load balancing is the important factor in achieving parallelism in the NOWs. The term "load balancing" means assigning equal load among the available workstations. There are two major categories of load balancing: static and dynamic load distribution.

2.4.1 Static Load Distribution

Static load distribution algorithms do not use system-state information to distribute the load. In this scheme, task graph and system parameters are utilized before the run time of the process by a smart compiler or by user. Static load distribution is easy to implement since the system does not have to keep the state information. Overall accuracy of this scheme depends upon the accuracy of the information obtained from the task graph. The main advantage of this algorithm is that it is simple to implement and it is very efficient when workload can be sufficiently well described. The disadvantage of this algorithm is that it fails to adjust to the fluctuations in the system load.

2.4.2 Dynamic Load Distribution

Dynamic load distribution algorithms use the state information of the system to distribute the load. This algorithm consists of five policies: load mesurement, transfer policy, selection policy, location policy, and information policy.

The state information provides the load state of a particular node at a given time by measuring the certain parameters such as length of CPU queue, context switch rate, system call rate, the amount of available memory, and CPU utilization. In this research, dynamic load distribution is used and load measurement is made

at each node and it is communicated to the peers. Load measurement also comes with its own overhead if the measurement scheme is to be more accurate. Dedicated programs or system calls such as SAR (System Activity Reporter) can be used to get fairly accurate load status of the node.

Transfer policy decides when to migrate the task from one node to another. When a process is about to be created, a decision has to be made whether or not it can be run on the machine where it is being generated. If that machine is too busy, the new process must be transferred somewhere else. Threshold values are used to make a decision of transfer. Different transfer mechanisms are suggested by Lin and Keller [18] and by Ni [19]. Lin and Keller have used two threshold values to decide the node's load status as lightly or heavily loaded. On the other hand, Ni has proposed that an underloaded node should also seek to accept processes from heavily loaded nodes to balance the overall load.

Selection policy decides which tasks to be transferred. This policy considers several factors such as overhead incurred in transferring the task, life of the process, and the state information, in selecting a task for transfer.

Once the transfer policy has decided to migrate a process, the location policy determines where to send it. Finding a suitable node for transfer is called as the location policy. If the system uses centralized scheduling policy, the central coordinator makes the decision of transfer. In the decentralized scheme, on the other hand, each node is polled to test its ability of load sharing.

Information policy manages the information (load states of nodes) storage and information exchange mechanism. In a centralized scheduling scheme, load state of

all nodes is stored in a central coordinator node. Upon updating its current state, each node sends its state information to central coordinator node. Many schemes have been suggested for decentralized policy. In a demand driven policy suggested by Lazowska et al. [20], load information of other nodes is requested only when an individual node wants to be either a sender or receiver. In a periodical policy suggested by Liu et al. [21], each node periodically informs all other nodes of its load status. Using these load estimates, the job dispatcher on each node then makes the decision of task migration.

Chapter 3

ALGORITHM AND IMPLEMENTATION

This chapter describes the model and the algorithm used in this research. Distributing the tasks among the idle or lightly loaded workstations is also discussed. Speedup equations are formulated for the model and the therotical results are presented.

3.1 The Model

The tasks used for distributed computing in the NOWs can be represented by using the task graph format. Figure 3.1 shows the tasks[1] used in this research.

The solid lines represent the precedence relations and each circle represents one task. No connection between circles (tasks) means that these tasks can be executed in any order. The horizontal dotted line shows the division of tasks among the workstations; i.e., T_{20} to T_{25} are executed in workstation 1, T_{14} to T_{19} are executed in workstation 2, etc.

---

[1] We define a task (T) as calculation of a row of the resultant matrix in matrix multiplication.A detailed explanation of the matrix multiplication task is given in Chapter 5.

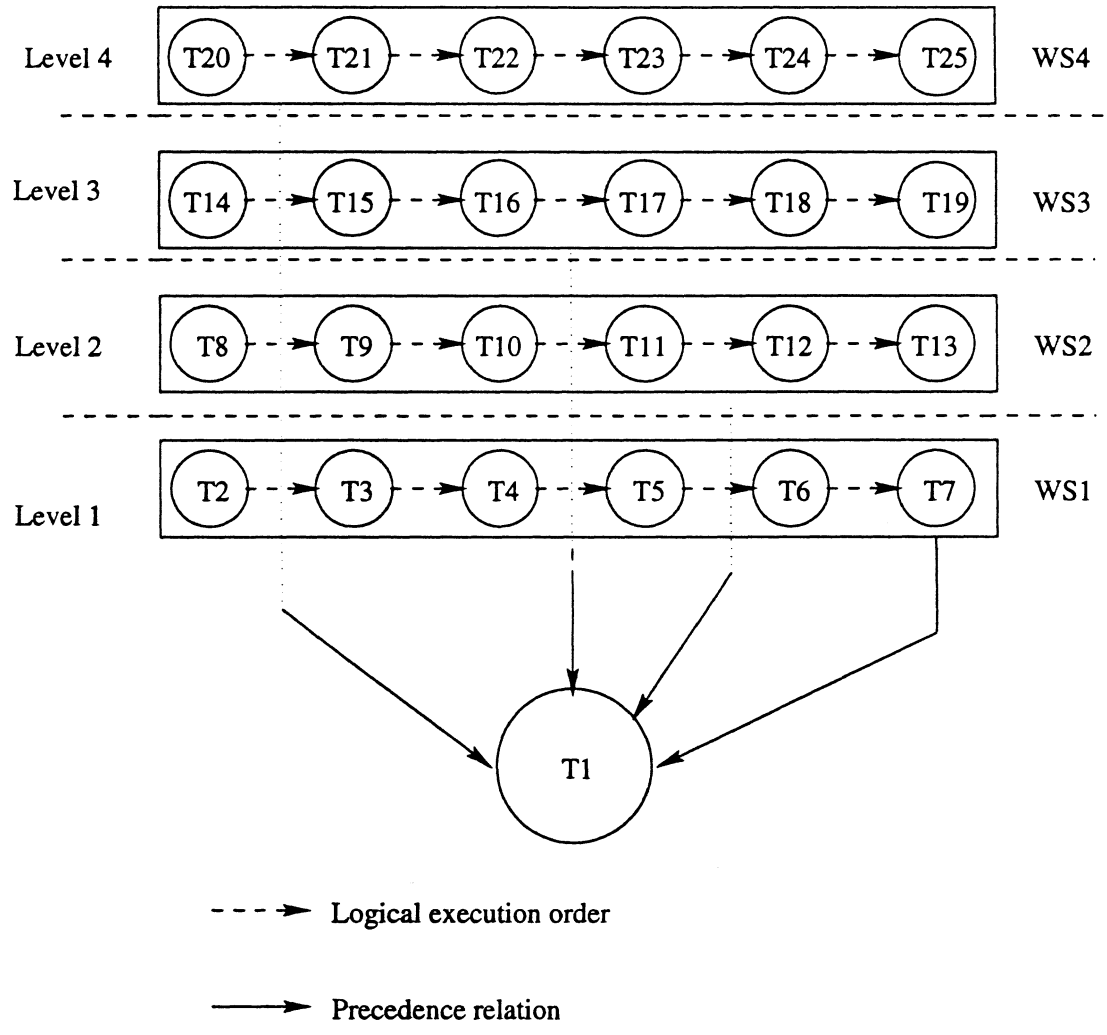


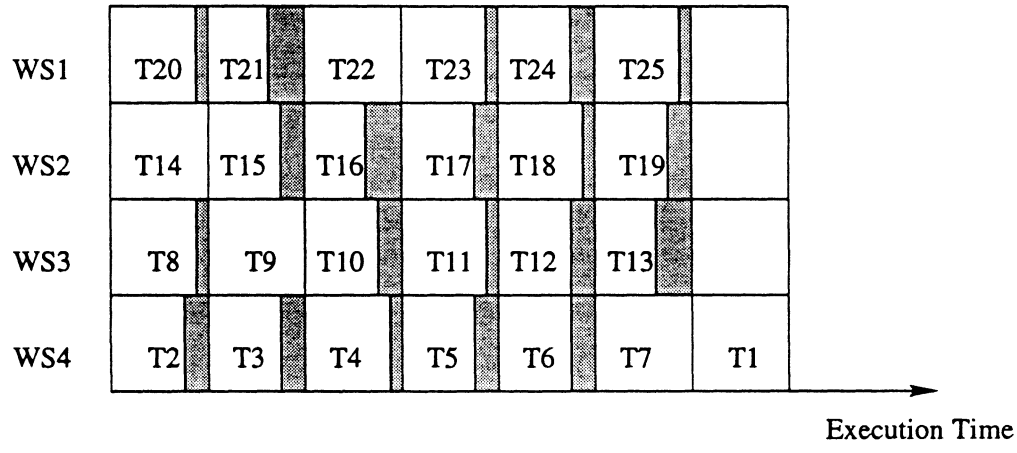
Figure 3.1: Task graph of a matrix multiplication C = A x B, where A and B are both 25 x 25 matrices.

Figure 3.2(a) shows the Gantt chart demonstrating the task graph of Fig.3.1 for each task having different execution time ($T_4 = T_8 = T_{11} = T_{18} = T_{20} = T_{23} = T_{25} = 0.9E, T_2 = T_3 = T_5 = T_6 = T_{10} = T_{12} = T_{15} = T_{17} = T_{19} = T_{24} = 0.7E, T_{13} = T_{16} = T_{21} = 0.65E, T_1 = T_7 = T_9 = T_{14} = T_{22} = E$), and (b) for each task having the same execution time ($E_1 = E_2 = ...E_{25} = E$).

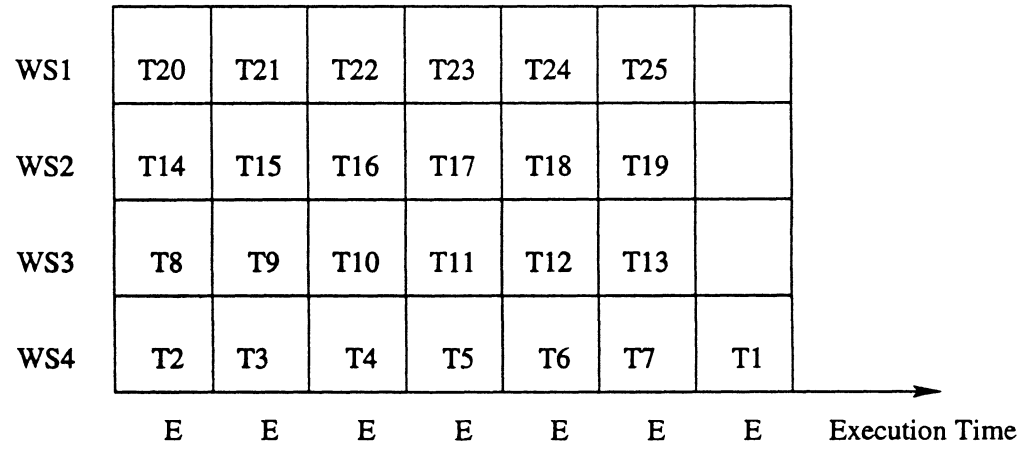
The communication and computation processes carried out by each workstation are modeled as shown in Fig.3.3. This model best describes the uniprocessor workstation in the network where each workstation executes local and remote tasks considering the communication and computation list generated by the central node prior to the execution. The function of the computation process is to carry out the actual computation. The communication process separates local and remote messages, sends the tasks to the queue, achieves the synchronization among the tasks, and eliminates the race condition.

The *request* signal of the remote machine is processed by the communication process based on FCFS (First Come First Serve) policy. The acceptance of the *request* depends on the scheduling algorithm. In order for the owner of the workstation not to be affected, local task is given the highest priority. After receipt of the remote tasks, they are assigned a priority and placed in the task queue.

As stated earlier, the tasks of the user are given highest priority even if the remote tasks are already in queue. The job of the computation process is to execute these tasks depending upon the highest-priority-first policy. The scheduling algorithm first examines the remote tasks to see whether the remote task will overload the system, or whether the remote task will be delayed for unacceptable time. If one of these situations arises, the communication process dequeues some or all of the



(a)



(b)

Figure 3.2: Gantt chart of matrix multiplication task in workstations WS1 through WS4 for the tasks with (a) different and (b) the same execution times.

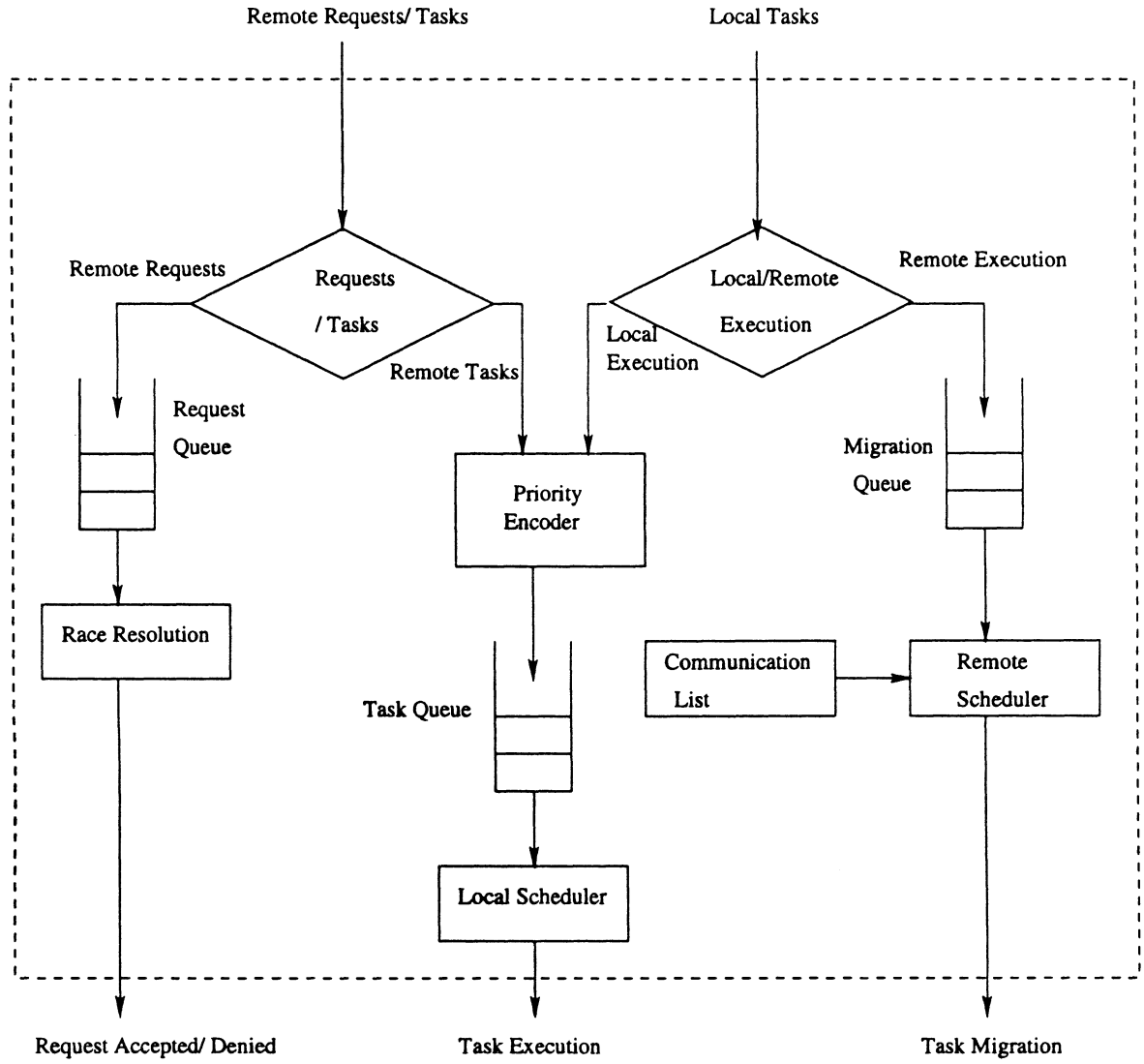


Figure 3.3: A model of communication and computation processes.

tasks and sends them back to the source nodes. To avoid this worst case situation, the communication process of the remote node takes over the computation job temporarily.

The communication list is made up of several tables as shown in Fig.3.4. All the idle or lightly loaded nodes are listed in the node entry table. An entity in the node entry table is a pointer to the statistical load table. The statistical load table has three fields. In the first field, the node ID is stored. The second field stores the time interval in which the workload of a workstation is sampled. The percentage of CPU utilization is stored in the third field. The information in the statistical load table enables the scheduler to locate idle and lightly nodes. A statistical load table is obtained by executing monitor program on each workstation as a system job in the background and it is broadcasted throughout the network periodically (e.g., once a day, every two hours, etc.).

3.2 Scheduling Algorithm

The dynamic scheduling algorithm used in this research is a hybrid of Mutka's method [15] and decentralized scheduling scheme. This algorithm is based on the percentage availability of workstation obtained daily over a period of observation time and present load condition of the remote node.

3.2.1 Load Metrices

Schedulers make better decisions if they can take advantage of information about the nature of the jobs that they are scheduling. However, in a time sharing system, exact knowledge of a job is unavailable and the estimates are either unreliable

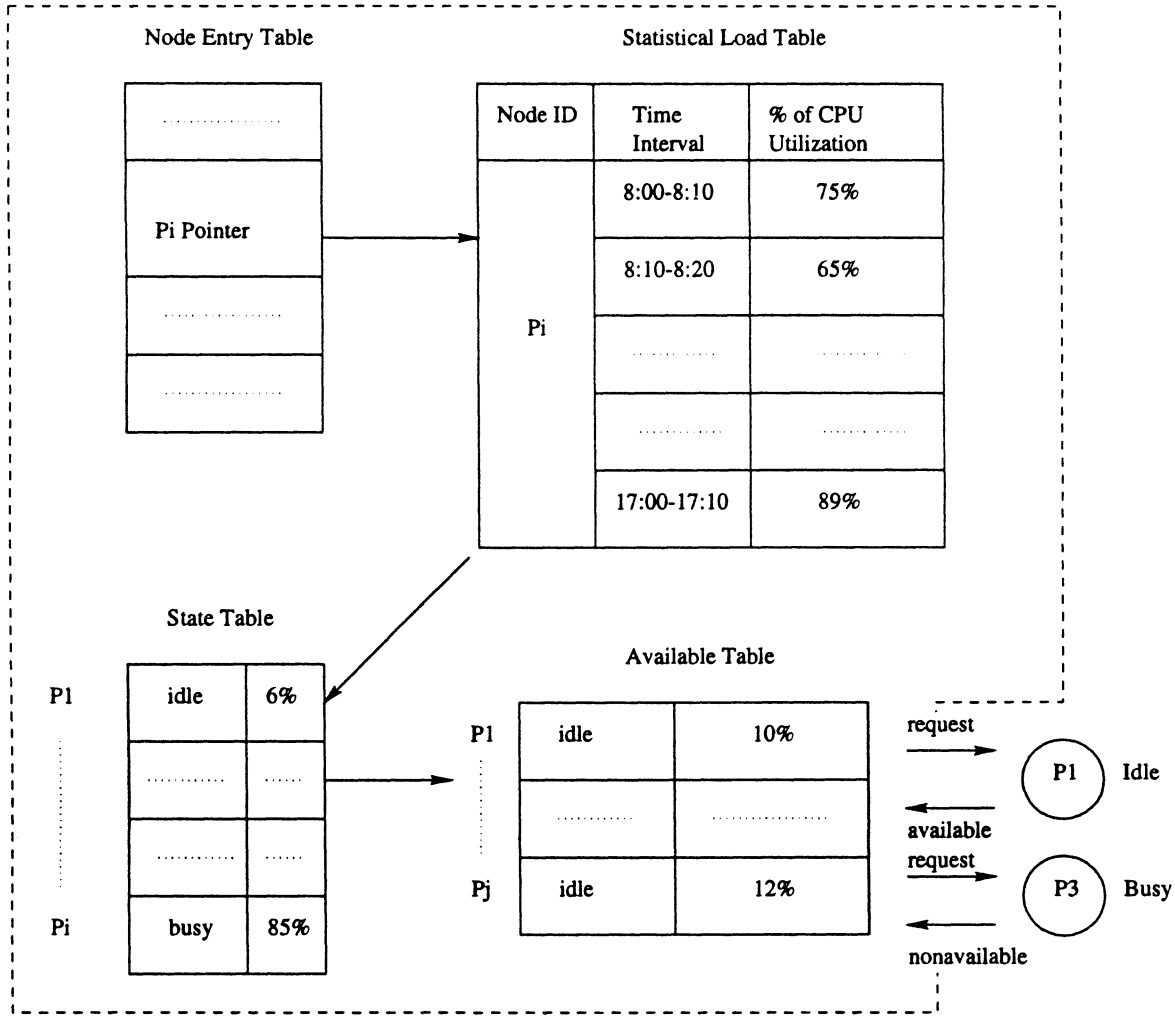


Figure 3.4: Communication list used for selecting idle or lightly loaded nodes.

or time-consuming to compute. Hence it is wise to take advantage of readily available metrices such as memory requirement, processor type, and processor utilization.

3.2.2 Workload Assumptions

To make the scheduler design little easier, we assume that most of the workstations in the network are autonomous and dedicated mainly to the local users. We assume that workload primarily consists of interactive and fast-turnaround applications and exists for relatively short periods of time such as text editing and program development. Generally, many machines are idle or lightly loaded and low-cost activities such as text editing are still handled. Periodically, however, few users may grab the entire system's free resources to run large computation-intensive applications.

3.2.3 Measurement of Workstation Load

To calculate the load state of a node, *threshold* values are expressed in terms of time percentage of CPU utilization. Two different threshold values are used for this purpose. First one is local threshold (L_t), which is decided based upon the node's capacity. If the system is homogeneous, this value is the same for all the nodes. The local threshold value is used by remote nodes to decide whether or not the node is busy. In our experimentation, L_t is high enough so that activities such as time of day clocks or graphical representation of system load does not generate user loads that rose above threshold. It is also low enough so that the station would be solely under the control of owner when the owner conducts local activity. This value is selected as 0.25 of idle CPU time. The second threshold value, state threshold, is used by communication process to determine the node's load status. The state threshold consists of two values, idle state threshold (L_i) and lightly loaded state threshold (L_l). These

values are selected as 5% and 10% of idle CPU time, respectively, because throughput drops if the slave node has load greater than 10% of idle CPU time which is explained in Chapter 4.

Whenever the communication process attempts to search the idle or lightly loaded nodes, it uses L_i and L_l to construct the state table. If the percentage of CPU utilization in statsitical load table is less than L_i, the node is marked as idle node. If the percentage of utlization of CPU in statistical load table is between L_i and L_l, the node is considered as lightly loaded node. Otherwise, the node is considered as heavily loaded node. This state transition is shown in Fig.3.5. The state table is simplification of the statistical load table. It is used by communication process to select quickly idle or lightly loaded nodes. The selected nodes are placed in the available table of Fig.3.4. The available table is used by the communication process to select required number of nodes for remote execution.

3.2.4 Remote Machine Search

Once the computation process decides the tasks to be run parallel, the communication process has to find the idle nodes from the communication list. The *request* signal is sent to a sufficient number of nodes from the communication list. If T_n is the number of tasks that can be processed in parallel, then the sufficient number of nodes could be T_n+A_n. Since the statistical table does not reflect the current load status of a node, some of the nodes could be busy at the time of request. Hence we need to choose an additional number (A_n) of nodes to avoid unnecessary repeated requests. We can select the value of A_n depending upon the time of the day. For example, during the night A_n could be zero.

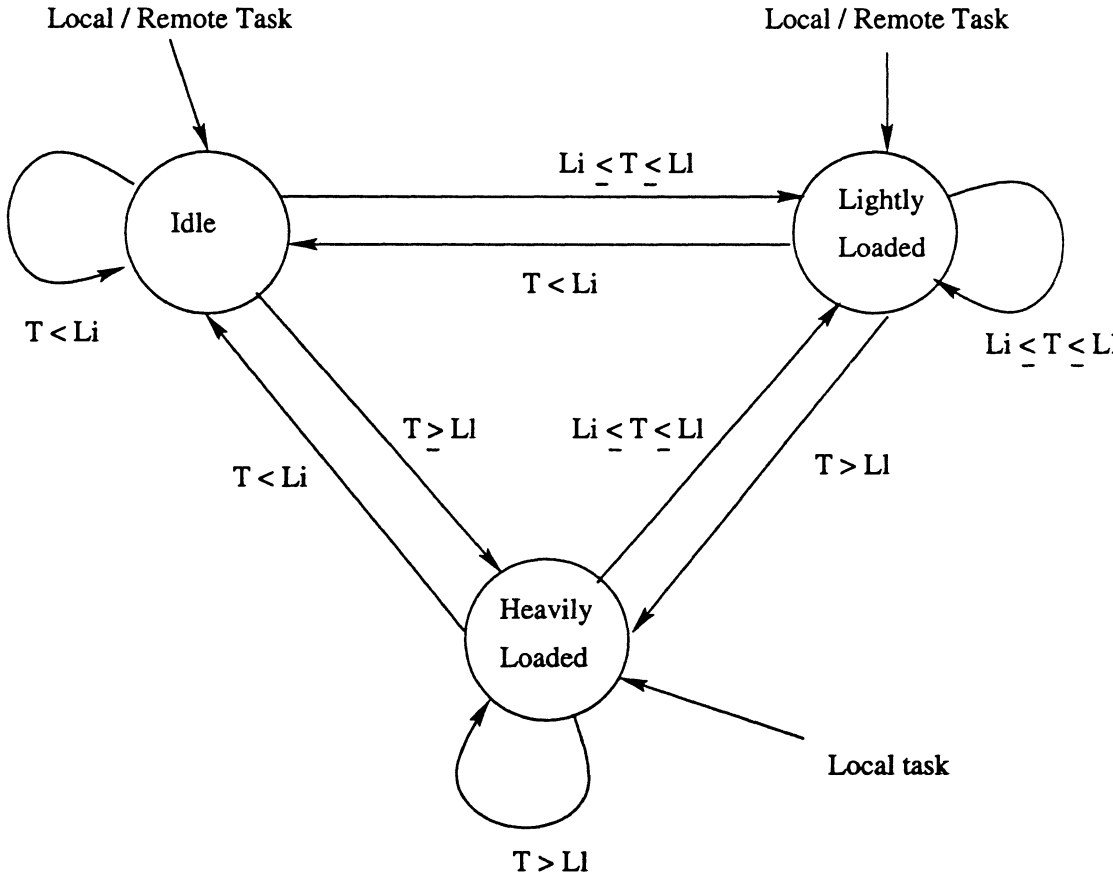


Figure 3.5: Load state transition diagram.

Next, communication process sends *request* signal to the available nodes along with the minimum load value L_{min} and the maximum load value L_{max} of the tasks to be migrated. L_{min} is the execution time of single task and L_{max} is the execution time required for $\frac{T_n}{n}$ tasks. The remote node checks the local threshold L_t with these values. If L_{min} added to the current load of the node is over L_t, then the remote station sends *unavailable* signal. If the L_{min} added to the current load of a node is under L_t, but L_{max} added exceeds the value of L_t, then the actual load value L_a is computed as L_t minus the current load L_c; i.e., if $L_{min} + L_a < L_t$ and $L_{max} + L_a > L_t$, $L_a = L_t - L_c$. In this situation, an *available* signal along with the value of L_a is sent back. If the load value, which is larger than L_{max}, added to the current load of the node is lower than L_t, then only *available* signal is sent which indicates that the remote node is idle or lightly loaded.

Deciding the value of L_t depends upon the priority of the local and remote processes. Local process has preference over the remote process. Hence the L_t should be set lower when the machine is running the local processes (i.e., the owner of the node is logged on). If the node does not have owner logged in, the value of L_t can be set high.

If the remote machine accepts the *request* signal from the source node, it sends an *available* signal. When the source node receives the *available* signal, it sends the task to the remote node. After receiving the task from the source node, the remote machine puts the task in the task queue.

A flag is set by the remote node once the *available* signal is sent to the source node and the remote node marks itself as a lightly or heavily loaded according to the load status. After the completion of remote task execution, remote node clears the flag and updates its load status.

The process of finding the idle or lightly loaded nodes is illustrated by the flowchart of Fig.3.6.

3.2.5 Migrating A Task

A task is migrated to the remote node after finding an idle or lightly loaded node for remote execution. After the remote execution of the tasks, results are sent back to the source node. For multiple task execution on remote nodes, each task should be sent separately. If a task has a data dependency, synchronization should also be considered. This synchronization delay will increase the execution time by [16]

$$S_d = Max(E_{j,i}) - Min(E_{j,i}) + C_d \tag{3.1}$$

Here, C_d is the communication delay between the node executing $T_{j-1,i}$ (i.e., the task i at the level j-1 of the task graph.) and the node executing $T_{j,k}$, and $E_{j,i}$ is the execution time of task $T_{j,i}$ (refer to Fig:4.1).

After receiving the results back from remote nodes if the master node does not have any task to migrate, it sends the *release* signal to the remote nodes and the communication process clears the flags of the remote nodes. If the remote nodes do not receive the *clear* signal, they wait for a timeout period and inform the master about their disconnection. If a remote node is still idle or lightly loaded, it can accept the tasks from other nodes.

3.2.6 Execution of Remote Process

Execution of a process at a remote machine is totally transparent to the owner of that remote node. After user's logon, the statistical table of that node is updated

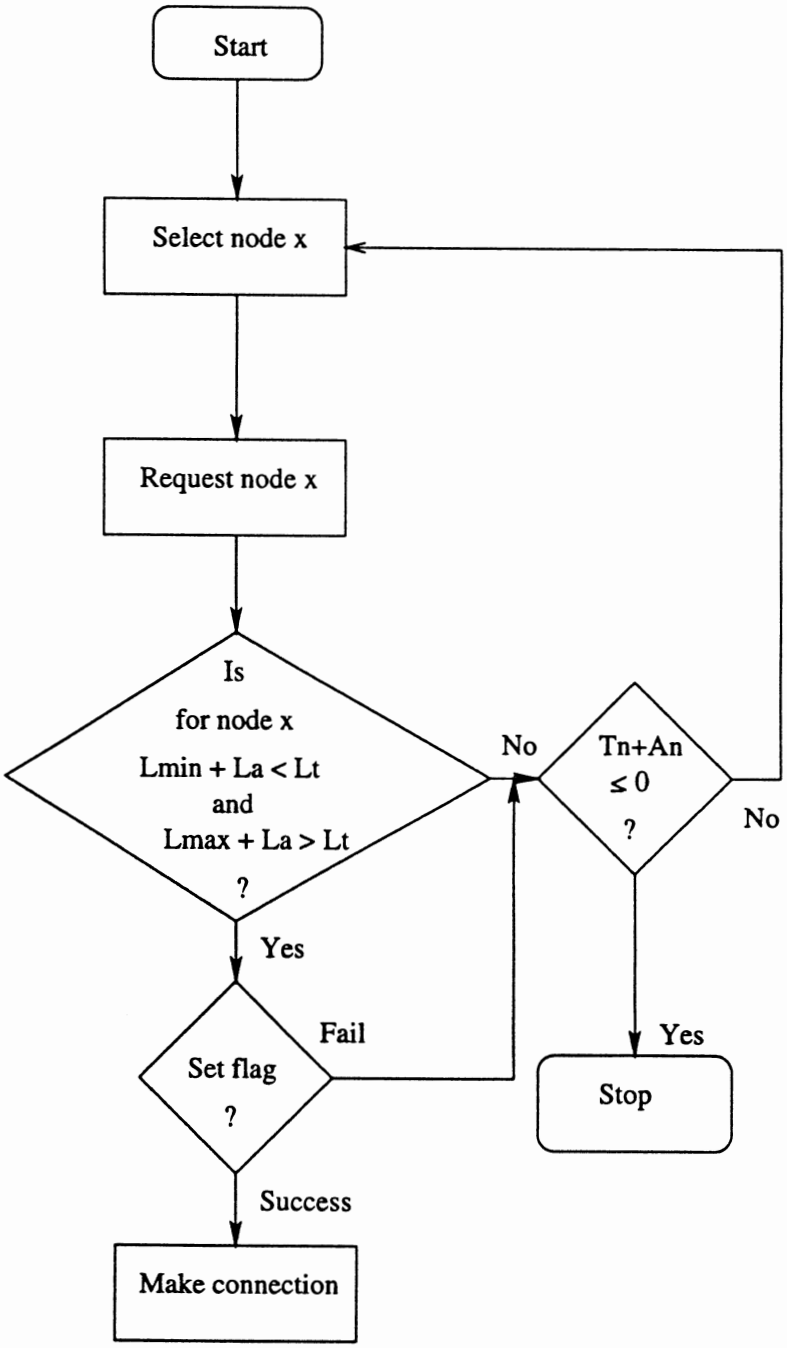


Figure 3.6: Flow chart of searching a remote node

by the communication process. It responds to the user while the computation process is still running the remote task. At this stage the task queue is checked by the communication process to see if the user resume will overload the system. It also informs the computation process to increase the priority of the remote task and finish the current job first.

Even if the user logs on, the currently running processes are not suspended immediately. This helps to increase the throughput by eliminating the time needed to find the new machine and the time required to send the task to that machine. The user's logging in different situations and the system response is discussed in detail in the following section.

3.3 Owner's Logging and System Response

As discussed in the earlier sections, the migration and execution of a remote task should be transparent to the owner of the system. The owner should not feel that the processing power of his system has been stolen. This requires that the remote machine responds quickly upon user's login. In the following sections we will discuss the different situations in which the user can login and the system response to user login.

3.3.1 Response During Logging

If the owner logs in when the node is processing a request from a source node; i.e., the time duration t_3 in Fig.3.7, the system should immediately respond to the owner and abandon any request from other nodes. This needs a highest priority

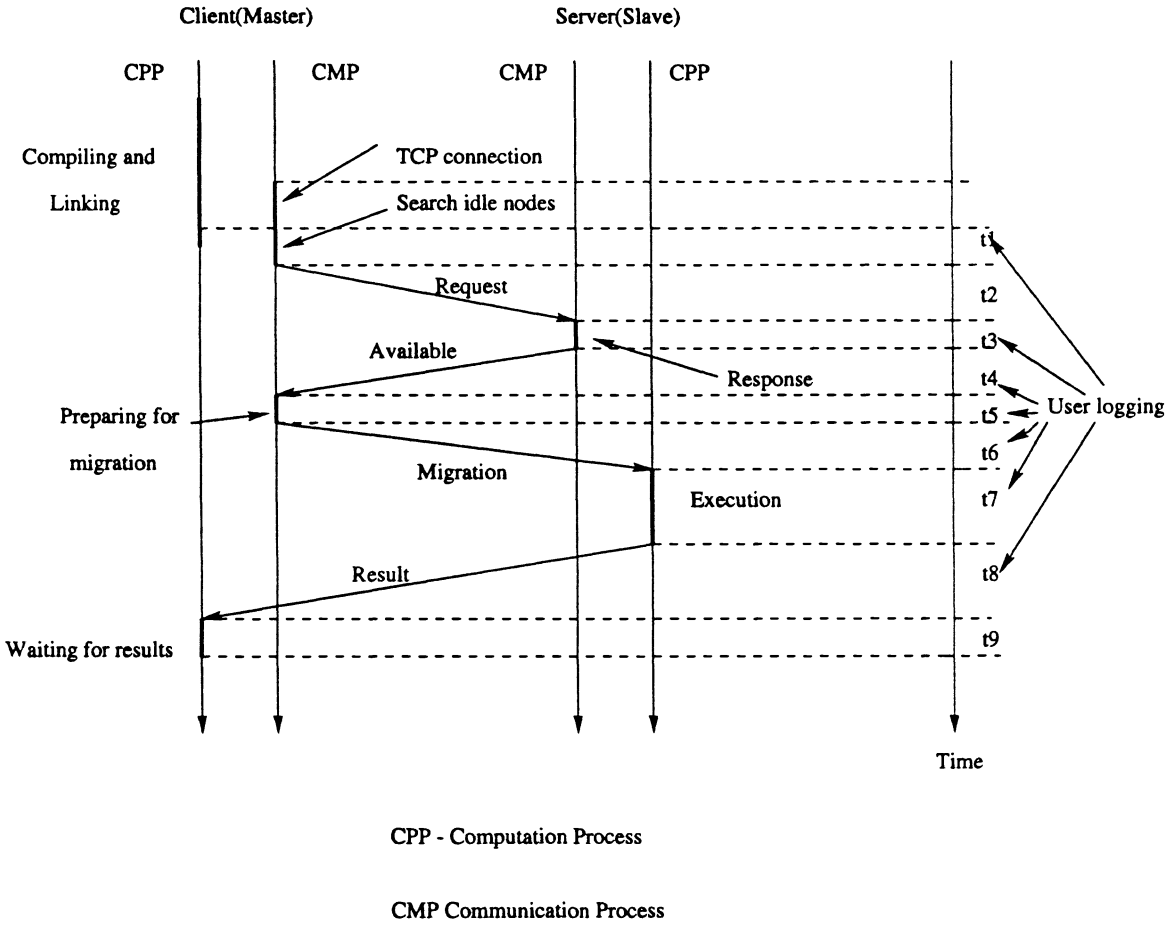


Figure 3.7: Timing diagram of remote task execution process and user logging.

interrupt to the system. In this situation, the node sends the *unavailable* signal to all the requesting nodes and resumes the local owner's tasks.

3.3.2 Logging Before Execution

As soon as the owner logs onto the node, the "busy" flag is set. If the *request* from another node comes after that, an *unavailable* signal is sent. The time it takes to receive the *available* or *unavailable* signal is called the response time. If the user logs in after the response time and before the task execution signal; i.e., during time intervals t_5 and t_6 in Fig.3.7, the situation becomes complex. In this situation, an

unavailable signal is sent back and the "busy" flag is set. Any task arrives after that is ignored. This will cause additional delay for the source node to find a new remote machine for task execution.

3.3.3 Logging During Execution Time

If the user logs onto the machine when it is executing a task; i.e., during time t_7 in Fig.3.7, it can either complete the current execution or it can abondon the execution and responds to the user. In the latter case, the overall performance of the system will be very poor. If the system decides to abondon the current execution, the source node has to find a new node. This will obviously increase the delay which includes the time required to find a new machine, communication time, response time, and migration time. In order to increase the system performance, we will allow the computation process to complete the current execution of the task and let communication process respond to the owner.

The time instances of all above cases are depicted in Fig.3.7. The time instances other than those discussed above are not critical for system response.

Chapter 4

SYSTEM PERFORMANCE

In the last chapter, we described how the idle machines are selected for remote task execution. We also discussed the system response to the various situations of owner's login. This chapter evaluates the system performance for a parallel task graph execution in a distributed computing network.

4.1 Performance Analysis of Distributed Computing in a Network

Performance of the network of workstations for distributed computing is measured in terms of the speedup. The speedup factor S is given by

$$S = \frac{t_s}{t_p} \tag{4.1}$$

where t_s is the sequential execution time of the task in the slowest workstation in the network and t_p is the parallel execution time in the network. The total sequential time t_s can be expressed as

$$t_s = \sum_{i=1}^{m} E_i \tag{4.2}$$

where m is the total number of tasks and E_i is the execution time of a subtask T_i.

For a k level task graph (see Fig.4.1), if there are m_i number of tasks in the ith order, then, we have

$$t_s = \sum_{i=1}^{k} \sum_{j=1}^{m_i} E_{i,j} \tag{4.3}$$

Substituting equations 4.3, 4.4, and 4.5 into 4.1 yields

The parallel execution time t_p is composed of three parts. They are execution time ($t_{p.exec}$), scheduling time ($t_{p.sch}$), and communication time ($t_{p.com}$). Hence, t_p is given by

$$t_p = t_{p.exec} + t_{p.sch} + t_{p.com} \tag{4.4}$$

The maximum value of t_p is encountered when there are no enough idle workstations to distribute the tasks. In this case, $t_{p.exec}$ is given by

$$t_{p.exec} = \sum_{i=1}^{k} max(E_{i,j}) \lceil \frac{m_i}{n} \rceil \tag{4.5}$$

where n is the number of processors and $\lceil \rceil$ is the ceiling value operator.

$$S = \frac{t_s}{t_p} = \frac{\sum_{i=1}^{k} \sum_{j=1}^{m_i} E_{i,j}}{\sum_{i=1}^{k} max(E_{i,j}) \lceil \frac{m_i}{n} \rceil + t_{p.sch} + t_{p.com}} \tag{4.6}$$

Equations 4.6 suggests that the system speedup can be improved by reducing the communication and scheduling time and by executing long-lasting tasks. This helps to determine the ideal and worst case values for S. If $t_{p.sch}$ and $t_{p.com}$ are much smaller than $t_{p.exe}$, we get

$$S = \frac{t_s}{t_p} = \frac{\sum_{i=1}^{k} \sum_{j=1}^{m_i} E_{i,j}}{\sum_{i=1}^{k} max(E_{i,j}) \lceil \frac{m_i}{n} \rceil} \tag{4.7}$$

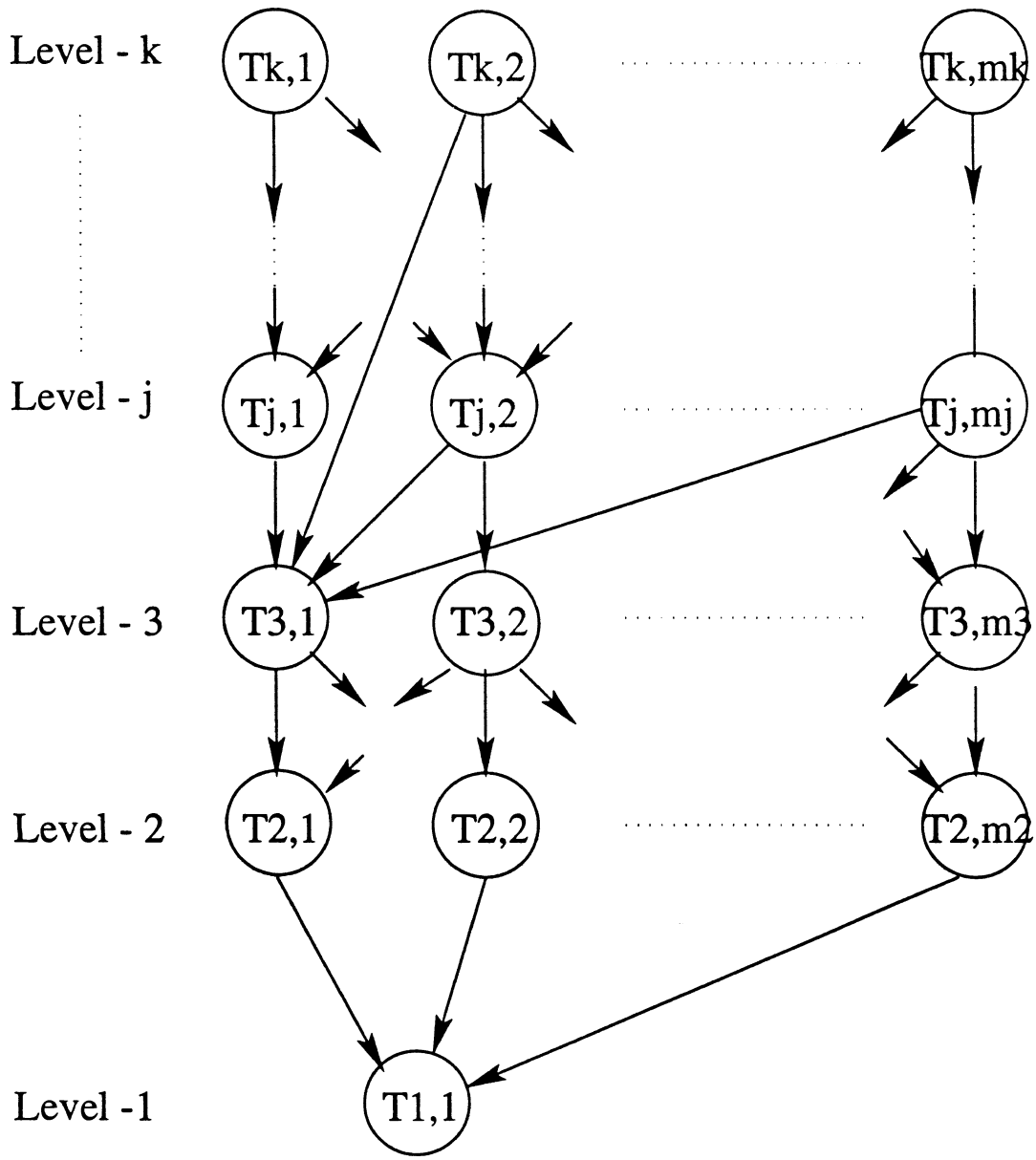


Figure 4.1: General task graph.

Further, if each task has the same execution time, then equation 4.7 simplifies to

$$S = \frac{t_s}{t_p} = \frac{mE}{\sum_{i=1}^{k} E\lceil \frac{m_i}{n} \rceil} = \frac{m}{\sum_{i=1}^{k} \lceil \frac{m_i}{n} \rceil} \tag{4.8}$$

If we assume that there are enough idle or lightly loaded nodes to carry out remote execution, then we have $n \geq m_i$. Considering this, we obtain from Eq.4.8

$$S \leq \frac{m}{k} \tag{4.9}$$

If this condition is not met; i.e., if there are not enough idle or lightly loaded nodes ($n \leq m_i$), then we have

$$S \geq \frac{m}{2k} \tag{4.10}$$

Thus the ideal speedup can be predicted as

$$\frac{m}{2k} \leq S \leq \frac{m}{k} \tag{4.11}$$

In the worst case situation, we claim that $0 \leq S \leq \frac{m}{k}$. Hence

$$0 \leq S \leq \frac{m}{k} \tag{4.12}$$

These are the upper and lower bounds on S.

4.2 Scheduling and Communication Delay

To investigate the effect of $t_{p.sch}$ and $t_{p.com}$ on speedup, we consider the timing diagram of Fig.3.7. Here, the time required to search idle nodes (t_s) and establish connection between master and slave nodes (t_c) is t_1. t_2 is the time required for the *request* signal to travel through the network. t_3 is the time that a remote workstation

spends to respond to a request from the master node. t_4 and t_5 are the *available* signal delay and time required to prepare the task for migration. t_6 is the time needed to migrate the task and t_7 is the remote execution time. t_8 is the time required to send back the results and t_9 is the time it takes to wait for all results to arrive. From this timing diagram, we can write the scheduling delay as

$$t_{p.sch} = t_c + t_{1p.sch} \tag{4.13}$$

where

$$t_{1p.sch} = t_s + t_3 + t_5 + t_9 \tag{4.14}$$

Here, t_s is the time spent to search for idle nodes. For a linear search method this time is equal to O(n-1) [16], where n is the total number of nodes in the network. If we use a binary search method, the time required to search nodes is $O(\log(I_n))$ [16], where I_n is called the time interval number. Also, time needed to search for the available state by verifying all the information in the state table is O(n). Hence we can write

$$t_s = [O(n-1)O(log(I_n)) + O(n)]t_u \tag{4.15}$$

where t_u is the unit execution time. From equation 4.15, it is clear that the time required to search the nodes is proportional to the number of nodes (n) in the network and the time interval of statistical load table.

In equation 4.14, t_3 is the time required for the slave node to respond to a *request* signal. For n nodes, in worst case situation, there can be n-1 request signals. Hence, the maximum value of t_3 can be t_3 = (n-1) t_r, where t_r is the quantum for the response time.

The other two timing parameters in equation 4.14 are t_5 and t_9. t_5 is the time required to prepare a task for migration (t_m) and t_9 is waiting time (t_w) for results. t_w depends on the difference between the execution times of subtasks in a particular level and the protocol used for sending tasks. Now we can rewrite equation 4.14 as

$$t_{1p.sch} = [(O(n-1)O(log(I_n)) + O(n)]t_u + (n-1)t_r + t_m + t_w \tag{4.16}$$

Considering Fig. 3.7, the total communication delay can be given by

$$t_{p.com} = t_c + t_2 + t_4 + t_6 + t_8 \tag{4.17}$$

If the master node has q tasks to send to slave nodes, then

$$t_{p.com} = t_c + t_2 + t_4 + \sum_{i=1}^{q}(t_6 + t_8) \tag{4.18}$$

Substituting the delay time given in equation 2.1 into equation 4.18 and ignoring the queuing time, we get

$$t_{p.com} = t_c + t_2 + t_4 + \sum_{i=1}^{q}(t_6 + t_8)$$

$$= t_c + P_{t2} + P_{t4} + \sum_{i=1}^{q}(P_{t6} + P_{t8}) + R\left(L_{t2} + L_{t4} + \sum_{i=1}^{q}(L_{t6} + L_{t8})\right)$$

$$\leq t_c + 2P_{max} + 2qP_{max} + R(2L_{max} + 2qL_{max})$$

$$\leq t_c + 2(1+q)(P_{max} + L_{max}R)$$

$$\leq t_c + 2(1+q)D_{max} \tag{4.19}$$

Here, P_{max} and L_{max} are the maximum propagation time and maximum packet length and D_{max} is the maximum delay time of a packet in the network, respectively. Since $q = \sum_{i=1}^{k} \lceil \frac{m_i}{n} \rceil$. Hence, we have

$$t_{p.com} \leq t_c + 2(1 + \sum_{i=1}^{k} \lceil \frac{m_i}{n} \rceil) D_{max} \tag{4.20}$$

Substituting equations 4.20 and 4.13 into 4.6, we obtain

$$S = \frac{\sum_{i=1}^{k} \sum_{j=1}^{m_i} E_{i,j}}{\sum_{i=1}^{k} max(E_{i,j}) \lceil \frac{m_i}{n} \rceil + 2 \left(1 + \sum_{i=1}^{k} \lceil \frac{m_i}{n} \rceil\right) D_{max} + t_c + t_{1p.sch}} \tag{4.21}$$

Assuming the execution time for all the subtasks to be the same, we find

$$S = \frac{m}{\sum_{i=1}^{k} \lceil \frac{m_i}{n} \rceil + 2 \left(1 + \sum_{i=1}^{k} \lceil \frac{m_i}{n} \rceil\right) d_1 + d_2} \tag{4.22}$$

where $d1 = \frac{D_{max}}{E}$ and $d2 = \frac{t_c + t_1.psch}{E}$.

4.3 Speedup Curves

To evaluate the system performance we have plotted the speedup curves as a function of subtask level(k) based on equation 4.22 for different values of d_1, d_2, m, and n. Figure 4.2 shows the task graph format as a function of k.

The plots are shown in Figures 4.3, 4.4, and 4.5. These plots confirm the fact that the communication and scheduling delays affect significantly the speedup performance of the system. Speedup drops as the level of subtask is increased.

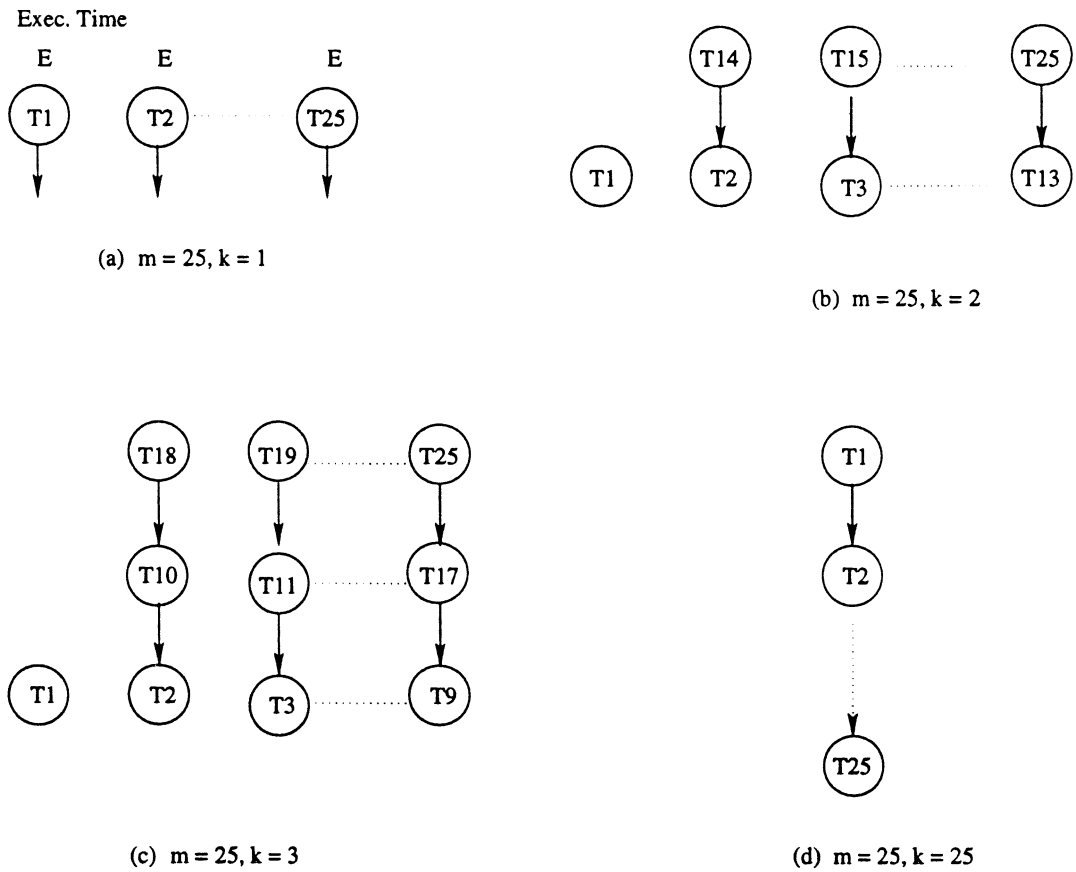


Figure 4.2: Task graph as a function of k for (a) m=25, k=1 (b)m=25, k=2, and (c)m=25, k=25.

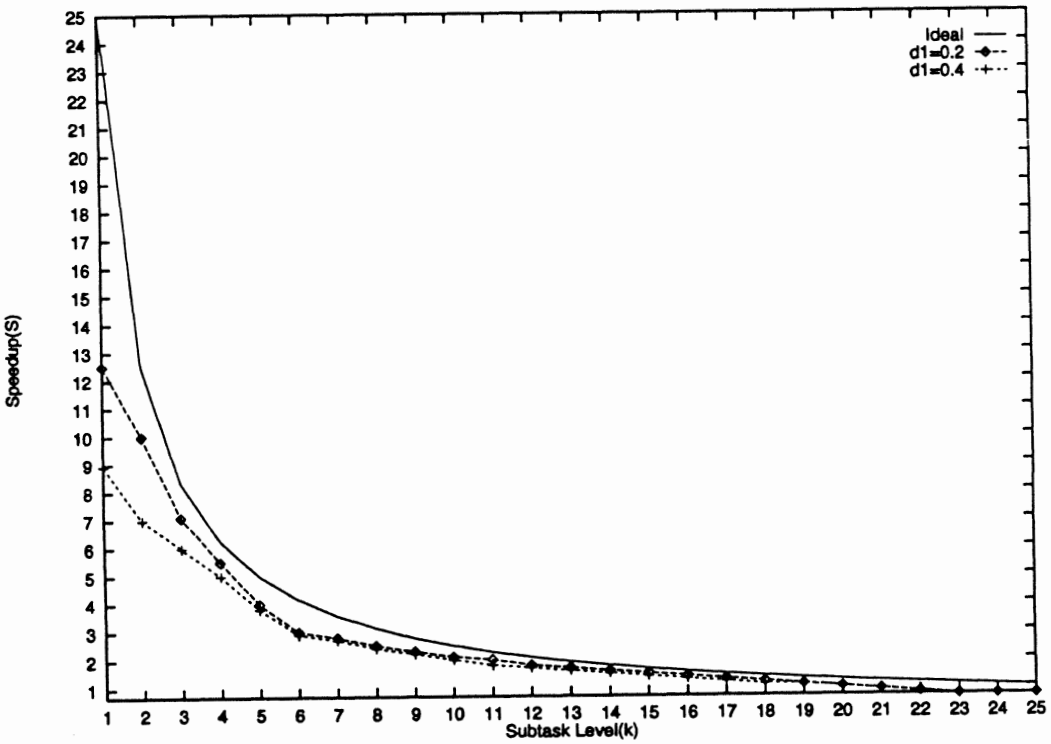


Figure 4.3: Speedup curves for n=25, m=25, d2=0.2, and different values of d1.

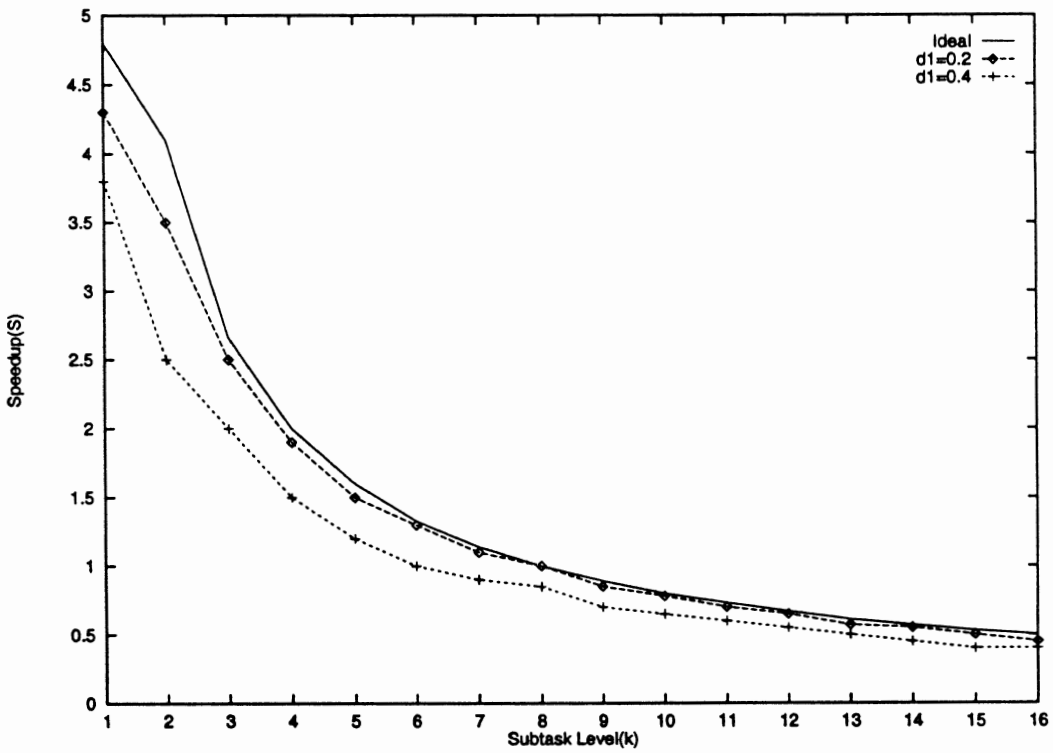


Figure 4.4: Speedup curves for n=8, m=25, d2=0.2, and different values of d1.

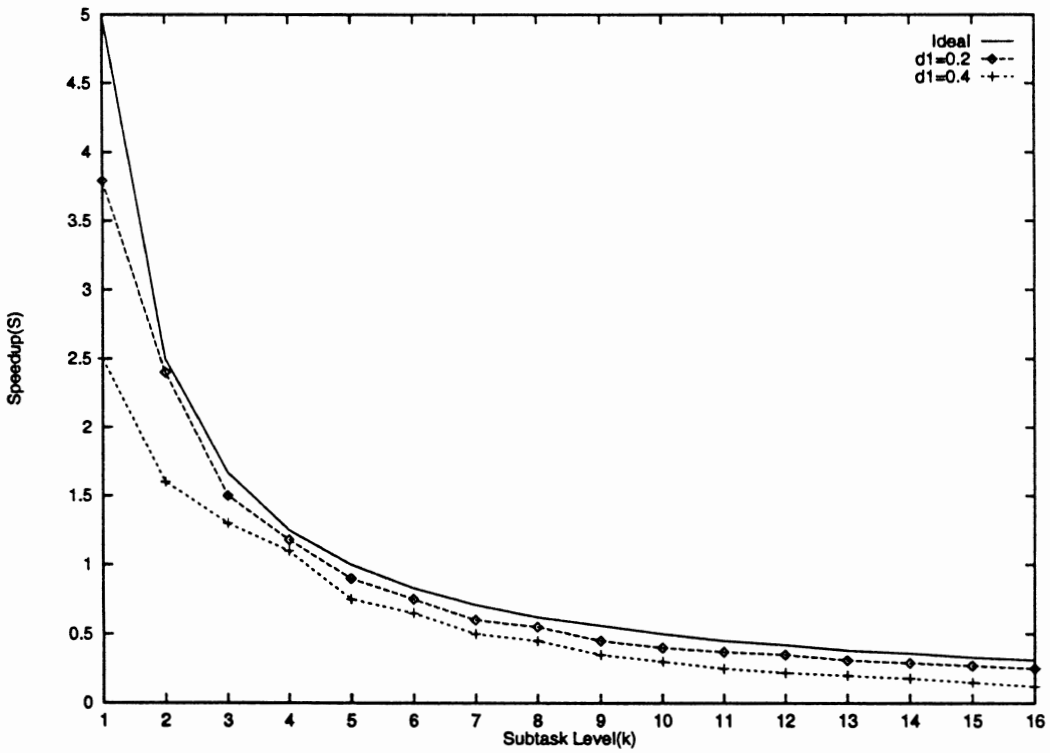


Figure 4.5: Speedup curves for n=5, m=25, d2=0.2, and differnt values of d1.

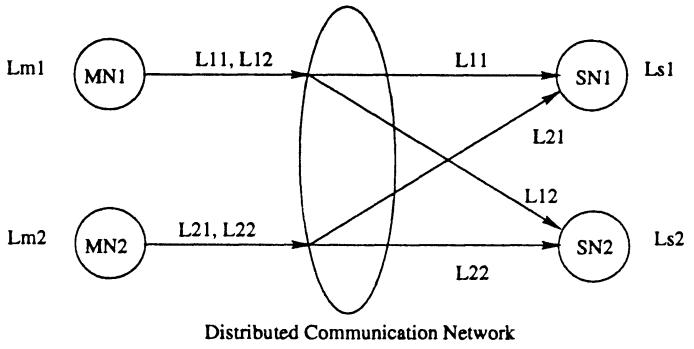


Figure 4.6: Load distribution model.

4.3.1 Load Distribution

For a network with imbalanced load distribution, speedup equation 4.22 becomes

$$S1 = \frac{m}{\sum_{i=1}^{k}(1+\Delta_{max})\lceil\frac{m_i}{n}\rceil + 2\left(1+\sum_{i=1}^{k}\lceil\frac{m_i}{n}\rceil\right)d_1 + d_2} \tag{4.23}$$

where $\Delta_{max} = max(\Delta_{ms})$ and Δ_{ms} is the percentage of the execution load difference between the master node (MN) m and slave node (SN) s.

To study the effect of imbalanced load distribution in the network, we refer to the network model given in Fig.4.6. In this model, there are two master nodes (MN_1 and MN_2) and two slave nodes ($SN1$ and $SN2$). MN_1 sends the loads L_{11} and L_{12} to SN_1 and SN_2, and MN_2 sends L_{21} and L_{22} to slave nodes SN_1 and SN_2 while SN_1 and SN_2 have the loads L_{s1} and L_{s2}. The total loads in SN_1 and SN_2 are then $L_{11} + L_{21} + L_{s1}$ and $L_{12} + L_{22} + L_{s2}$. Assuming MN_1 and MN_2 have loads L_{m1} and L_{m2}, then we have

$$\Delta_{11} = \frac{(L_{11} + L_{21} + L_{s1}) - L_{m1}}{L_{m1}} \times 100\% \tag{4.24}$$

$$\Delta_{12} = \frac{(L_{22} + L_{12} + L_{s1}) - L_{m1}}{L_{m1}} \times 100\% \tag{4.25}$$

These values are given relative to the master node MN_1. In general we can write equations 4.24 and 4.25 as

$$\Delta_{ls} = \frac{\left(\sum_{j=1}^{N} L_{ji} + L_{si}\right) - L_{ml}}{L_{ml}} \times 100\% \tag{4.26}$$

where l denotes the master node number, i denotes slave node number, and N is the total number of master nodes.

To study the effect of slave load condition on the speedup, we have ploted speedup vs. level of subtask for various load conditions of slave nodes. These plots are shown in Fig.4.7. These plots show that heavily loaded workstations are not suitable for remote execution of the tasks. We can also see from the plot that throughput drops faster for the remote nodes having loads greater than 10%. Hence we choose the threshold values L_i and L_l as 5% and 10%, respectively.

The effect of number of nodes on speedup has also been studied. The plots of speedup vs. number of nodes are shown in Fig.4.8. It has been found that increasing the number of nodes does not necessarily increase the speedup unless the levels of the task graph is decreased. Also, the effect of d_1 (communication delay) was much more significant than d_2 (scheduling delay).

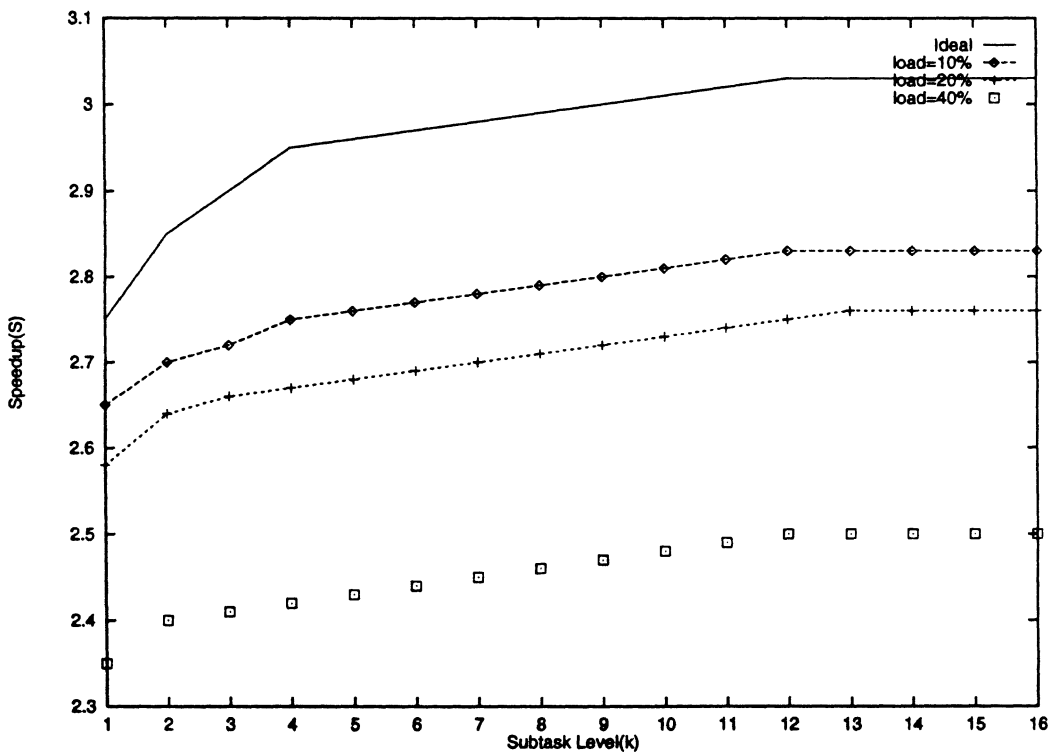


Figure 4.7: Speedup vs. subtask level for n=4, m=25, d1=0.2, d2=0.2, and different values of slave loads.

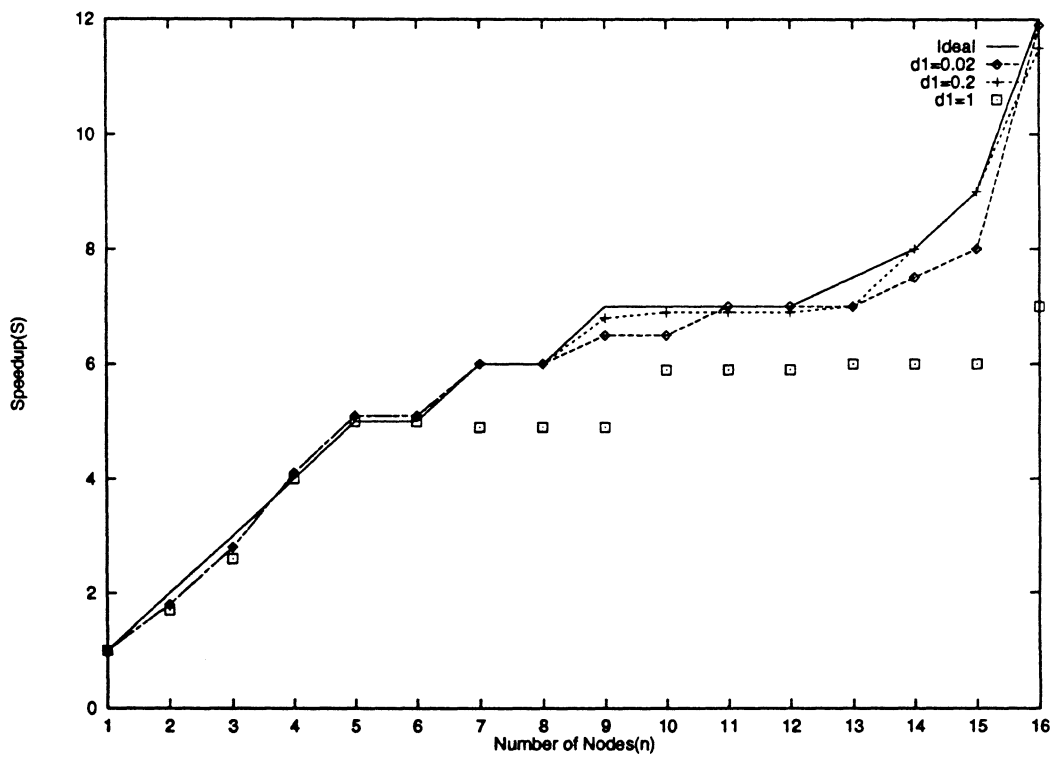


Figure 4.8: Speedup vs. number of nodes for m=25, d2=0.02, and k=16.

Chapter 5

SOFTWARE IMPLEMENTATION

5.1 Network Architecture and Client-Server Model

The parallel model and the scheduling algorithm discussed in Chapters 3 and 4 have been implemented in the Local Area Network at Stocker Engineering Center of Ohio University. The network structure of Stocker Engineering Center LAN is shown in Fig.5.1. This network consists of several Sun workstations and personal computers (PCs). For experimentation, Sun workstations Skinner, Maggie, Barney, and Lisa, which are connected to *Homer* station, are selected as the test bed. These machines are basically Sun SPARCstations 5/20 running Sun Solaris operating system (version 2.5) and share a common file server (Homer). The selection of the test bed does not take into account the delay due to Hub since the selected stations are in a distance that does not cross a hub.

TCP/IP (Transport Control Protocol/Internet Protocol) provides a peer to peer communication on this network. TCP/IP allows a user to establish communication between two application programs and pass data back and forth. In the peer to peer networking system, client-server paradigm is the basis for most computer communications application. The client-server model solves the rendezvous problem by asserting that in any pair of communicating applications, one side must start execution and wait indefinetely for the other side to contact it. Depending on whether the application waits for or initiates the communication, the communicating applications are divided into two types. An application that initiates the communication is

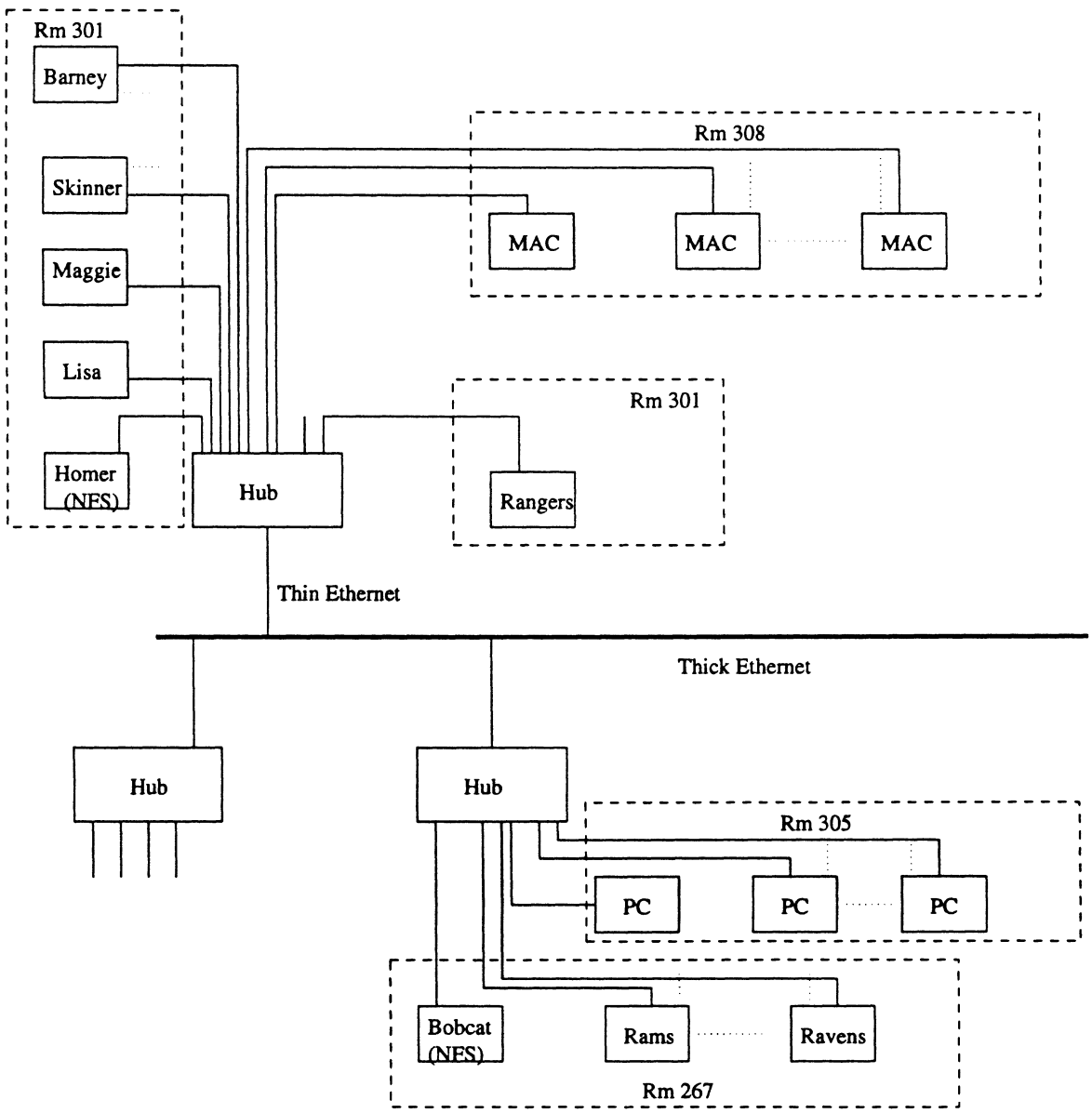


Figure 5.1: Network of workstations in the College of Engineering and Technology of Ohio University.

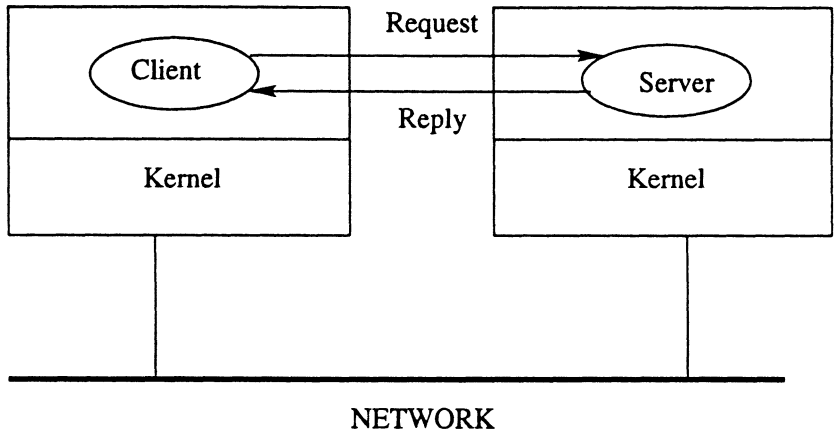


Figure 5.2: The Client-Server model.

known as *client* and an application that waits for incoming communication requests from *client* is called *server* [23]. The simple client-server model is depicted in Fig.5.2.

Whenever the *client* contacts a *server* for a particular application, it sends the request signal and waits for the response from the *server*. The *server* receives the incoming request from the *client*, processes it, and returns the desired result back to the client. After receiving the results, the *client* might continue to send requests or terminate the connection. The connection oriented approach in the client-server model makes programming easier. We have used this approach for our implementation. Most of the client-server applications are developed using TCP because of its reliabilty. Other option is connectionless Universal Datagram Protocol (UDP) [24].

The software implementation of the algorithm discussed in this thesis also heavily depends on Sun's *Network File System* (NFS) [24]. NFS is the mechanism that allows a computer to run a server that makes some or all of its files available for remote access, and allows applications on other computers to access those files. When an application accesses a file that resides on a remote machine, the operating

system invokes the client software that contacts a server on the remote machine and performs the requested operations on the file.

NFS accomplishes this goal by defining two client-server protocols [24]. The first NFS protocol handles mounting. Mounting is a process of attaching file system hierarchy to a given path. A client sends path name to a server and requests permission to mount the directory somewhere in its directory hierarchy. If the path is legal, the server returns a file handle to the client. The file handle contains fields uniquely identifying the file system type, the disk, and the security information.

Directory and file access is handled by the second NFS protocol. Client can send messages to the server to manipulate directories and to read and write files. They can also access file attributes, such as file mode, size, and time of last modification. Most of the UNIX system calls are supported by NFS.

5.2 Matrix Multiplication as a Distributed Task

Matrix multiplication is involved in many engineering applications such as image processing, robotics, signal processing, CAD/CAM applications, and in mathematical calculations such as simultaneous equations. A typical application in image processing is a ray tracing method to generate realistic 3D images [25]. Ray tracing proceeds by determining the visibility of the surfaces by tracing the imaginary rays of light from the viewer's eye to the objects in the screen. Since each point on the screen can be computed in parallel with other points, ray tracing is ideal for parallelization. For a $1,024 \times 1,024$ pixel image, 1,024 *chares* are created and can be distributed among the workstations. Calculation time of such a huge matrix multiplication can be drastically reduced by either using parallel algorithms or distributed computing.

Hence, matrix multiplication is one of the engineering problem that can be solved efficiently by using distributed computing.

Many research papers and texts [26-30] discuss the importance of parallelization of matrix multiplication problem. Each of them also suggests different algorithms taking into consideration available hardware and achievable throughput. The problem of parallelization of matrix multiplication problem among NOWs is proposed by Bhatkar et. al. [25]. The algorithm is explained below.

Suppose, we have two matrices A and B of size $m \times n$ and $n \times k$, respectively. The result of multiplication of these two matrices is another matrix C of size $m \times k$. This matrix multiplication involves *m.k.n* number of multiplications and *m.k.(n-1)* number of additions, resulting in a sequential time of

$$T_s = (m.k.n)t_{mult} + m.k.(n-1)t_{add} \tag{5.1}$$

where t_{mult} is the time to multiply an element of A and an element of B and t_{add} is the time to add these partial products.

We send each row of matix A to available remote node and the matrix B at each station. If we define each row of C, the resultant matrix, as one task then, we can map $\frac{m}{n}$ tasks among n idle workstations. The parallel execution time in this case can be given by

$$T_p = \lceil \frac{m}{n} \rceil .k.n.t_{mult} + \lceil \frac{m}{n} \rceil .k.(n-1).t_{add} + t_{p.sch} + t_{p.com} \tag{5.2}$$

To give a specific example, we consider matrix multiplication $A \times B$ shown in Fig.5.3.

To calculate $C = A \times B$, each row (a_{11}, a_{12}, a_{13} etc.) of the matrix A is sent to the corrosponding slave node (Skinner and Maggie) and the matix B is made available

$$A = \begin{bmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \end{bmatrix} \qquad B = \begin{bmatrix} b11 & b12 & b13 & b14 \\ b21 & b22 & b23 & b24 \\ b31 & b32 & b33 & b34 \end{bmatrix}$$

$$C = A \times B = \begin{bmatrix} c11 & c12 & c13 & c14 \\ c21 & c22 & c23 & c24 \end{bmatrix}$$

$$c11 = a11 \times b11 + a12 \times b21 + a13 \times b31$$
$$c12 = a11 \times b12 + a12 \times b22 + a13 \times b32$$
$$c13 = a11 \times b13 + a12 \times b23 + a13 \times b33$$
$$c14 = a11 \times b14 + a12 \times b24 + a13 \times b34$$

$$c21 = a21 \times b11 + a22 \times b21 + a23 \times b31$$
$$c22 = a21 \times b12 + a22 \times b22 + a23 \times b32$$
$$c23 = a21 \times b13 + a22 \times b23 + a23 \times b33$$
$$c24 = a21 \times b14 + a22 \times b24 + a23 \times b34$$

Figure 5.3: Example of matrix multiplication task distribution.

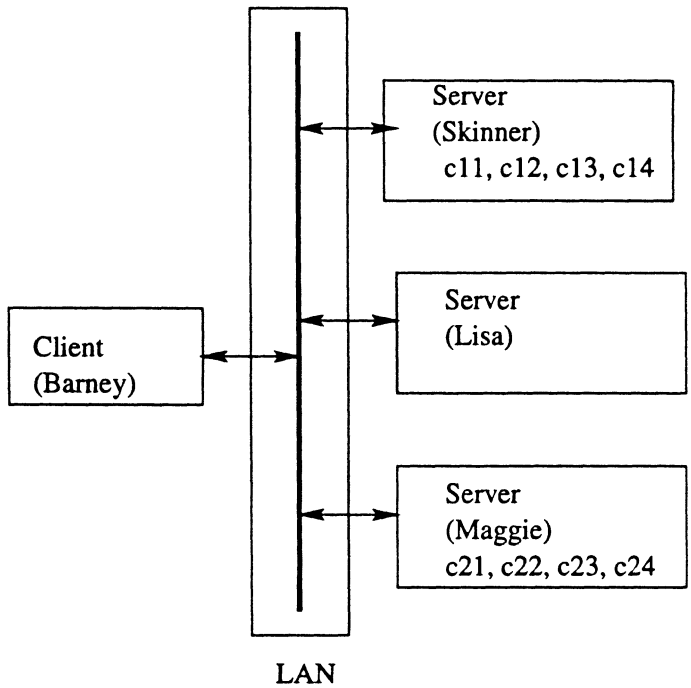


Figure 5.4: Network model used for matrix multiplication task.

at each slave node. Multiplication of each element of that row with elements of matrix B is performed as shown in dashed rectangle. As shown in Fig.5.4, workstation Barney is the client and workstations Skinner and Maggie are remote servers. First row of the matrix C is calculated at Skinner and second row is calculated at Maggie and results are sent back to client Barney. Here it is assumed that client Barney is heavily loaded and hence it is not executing any tasks.

5.3 Software Implementation

To implement the client-server model for distributed computing discussed earlier in this chapter, we have designed an interactive client and a concurrent server. The flowcharts of these programs are shown in Fig.5.5 and Fig.5.6.

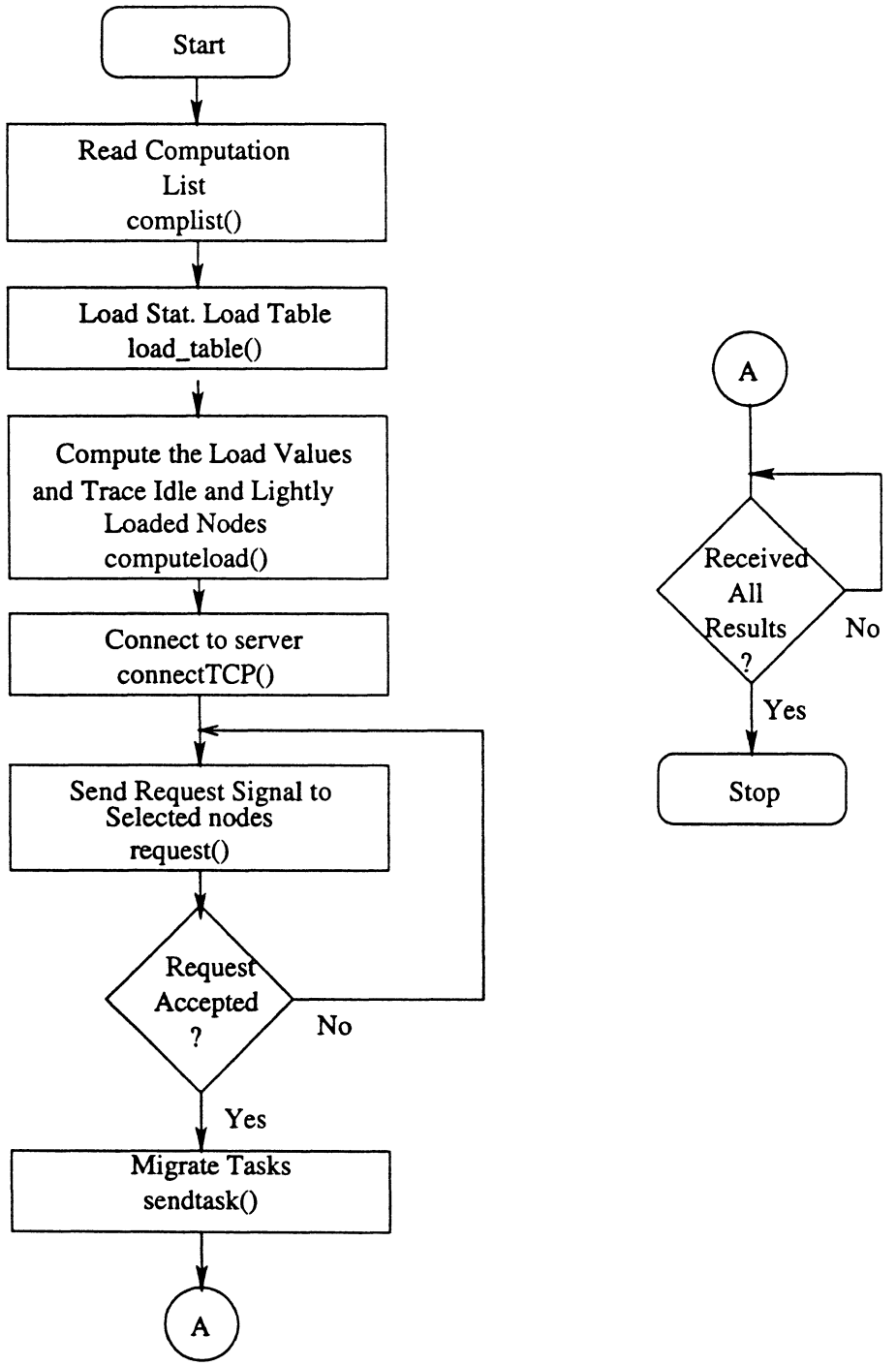


Figure 5.5: Flow chart of client program.

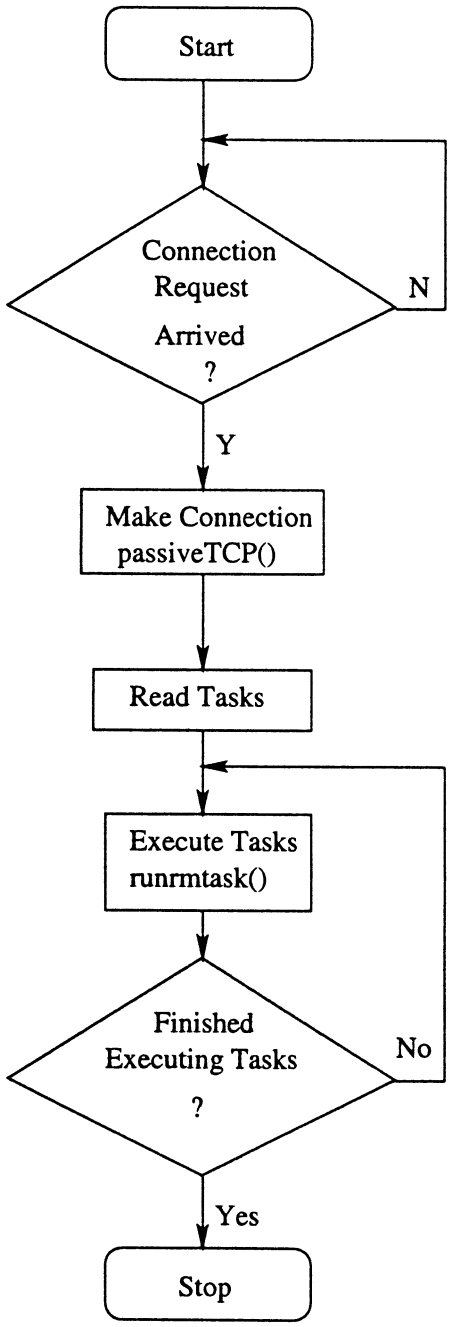


Figure 5.6: Flow chart of server program.

The server waits indefinitely for a request from the client. Whenever it receives a request from the client, connection is established between the client and the server using TCP/IP. TCP uses three way handshaking to establish a connection between two machines by specifying application type and the port number. Once the connection is established, the server reads the type of application the client wants to run. In our case, application type is the remote execution of the tasks sent by client to the server. After the client migrates the tasks, the server performs the required operations on the migrated data and sends the results back to the client.

When started, the client creates the computation list. These lists are depicted in Fig.5.7 and Fig.5.8, respectively. This statistical load table and the threshold values of loads (L_i and L_l) are used to determine available number of idle and lightly loaded nodes. After finding suitable number of idle or lightly loaded nodes, connection is established and the request for task migration is sent to the remote nodes in packets. Upon acceptance of request, tasks are migrated to the remote node and master node waits for the results.

Task	Execution Time
T25	0.9E
T24	0.7E
T23	0.9E
T22	E
T21	0.65E
T20	0.9E
T19	0.7E
T18	0.9E
T17	0.7E
T16	0.65E
T15	0.65E
T14	E
T13	0.65E
T12	0.7E
T11	0.9E
T10	0.7E
T9	E
T8	0.9E
T7	E
T6	0.65E
T5	0.65E
T4	0.9E
T3	0.7E
T2	0.7E
T1	E

E is unit time for each task (multiplcation + addition)

Figure 5.7: Computation list for matrix multiplication.

Workstation	Time Interval	% of CPU Utlilazation
Lisa	8:00-8:10	75%
	8:10-8:20	65%
	.....	.....
Skinner	.....	.....
	17:00-17:10	91%
	17:10-17:20	85%
	17:20-17:30	81%
	.....	.....
Maggie	.....	.....
	10:10-10:20	42%
	10:20- 10:30	50%
	10:30-10:40	64%
	.....	.....

Figure 5.8: A sample statistical load table.

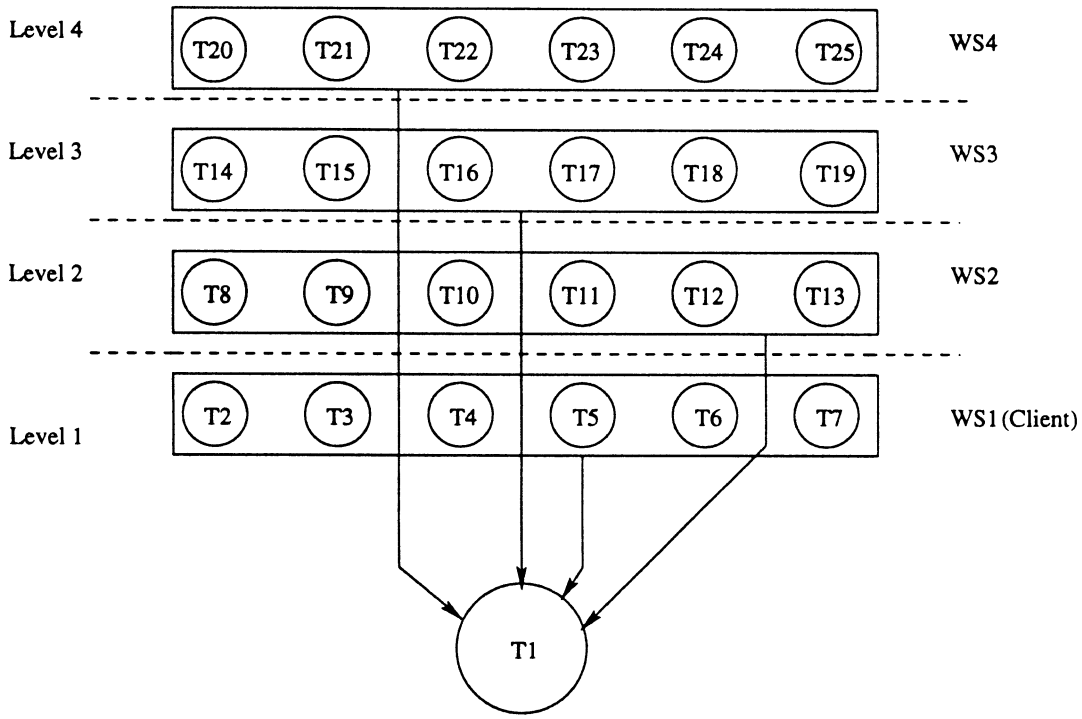
Chapter 6

EXPERIMENTAL RESULTS

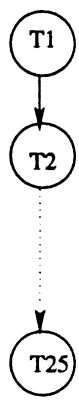
In this chapter the speedup results obtained in the experimentation of parallel execution of the matrix multiplication task are discussed. They are compared with serial execution of the program on a single machine. These results are obtained by using Sun workstations Barney, Skinner, Lisa, and Maggie of Fig.5.1

6.1 Speedup

The task graph of matrix multiplication used for speedup evaluation is shown in Fig.6.1. Matrices used for multiplication are of size 25×25. Two matrices of size 25×25 generate a resultant matrix of size 25×25. Each task corrosponds to each row of the resultant matrix obtained by multiplication of two matrices. Ideally we need minimum 25 machines to execute our 25 tasks. Since we have only four machines (Barney, Skinner, Lisa, and Maggie), we divide 25 tasks among four machines as shown in Fig. 6.1. The method used to allocate these tasks among four available machines is explained in Chapter 4. The plot shown in Fig.6.2 indicates that speedup is different for different subtask execution time. Results also show that there is no speedup gain if the single execution time is less than 44.44 ms.



(a) Parallel Execution



(b) Serial Execution

Figure 6.1: Task graphs used for (a) parallel and (b) serial execution.

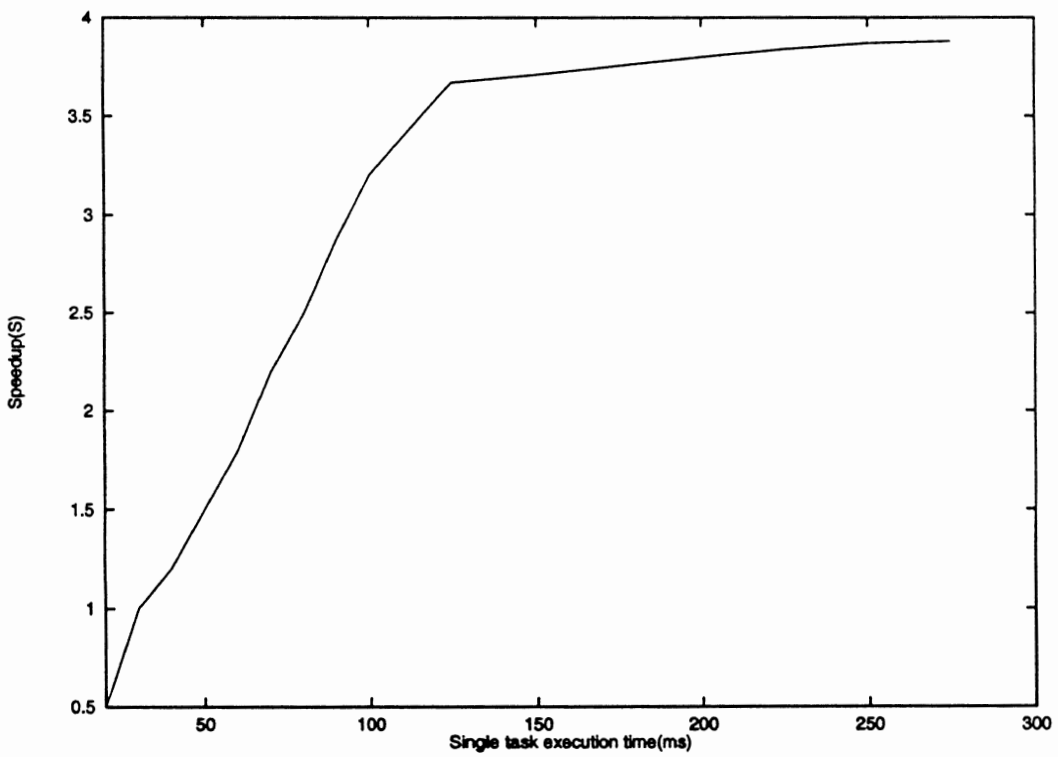


Figure 6.2: Plot of speedup vs. single task execution time(ms) for four workstations.

We can mathematically support this fact as follows:
If we use four nodes (n=4) in the network to distribute our 25 tasks (m=25), we can have four level task graph (k=4) as shown in Fig.6.1. Thus, for $S < 1$ we can write from Eq.4.22

$$\frac{25}{7 + 16d1 + d2} < 1 \tag{6.1}$$

Hence, $25 < 7 + \frac{16D_{max}+t_c+t_{1p.sch}}{E}$. By neglecting $t_{1p.sch}$ compared to t_c, we have $E < \frac{16D_{max}+t_c}{18}$. The value of t_c (communication time) is obtained by measuring the connection time between master (Barney) and any one of the slave (Skinner, Maggie, and Lisa) using *tcptrace* programs. This connection time was found to be 160 ms. The value of D_{max} was approximately 40 ms between these nodes. Thus $E < \frac{16\times40+160}{18} = 44.44$ms, which means that there will not be any speedup gain if the single task execution time is less than 44.44 ms. All the communication parameters discussed here are measured using *tcpdump* and *tcptrace* [31] programs. *tcptrace* uses binary file produced by *tcpdump*. A typical output of *tcptrace* is given in Appendix A.

6.1.1 Effect of Execution Time on Speedup

The plot of speedup vs. execution time is shown in Fig.6.2. In this experiment execution time is changed by changing matrix size. With these practical results we can again assert that speedup improves when task execution time increases. In our case this means that speedup improves if we use matrices of larger size for multiplication. This is due to the fact that the effect of communication overhead becomes smaller as the task execution time increases. Comparision of the serial and parallel execution times is depicted in Fig.6.3. This plot shows that total execution time increases almost exponentially as single task execution time increases. This also means that if we carry out matrix multiplication on one machine, total execution time increases as

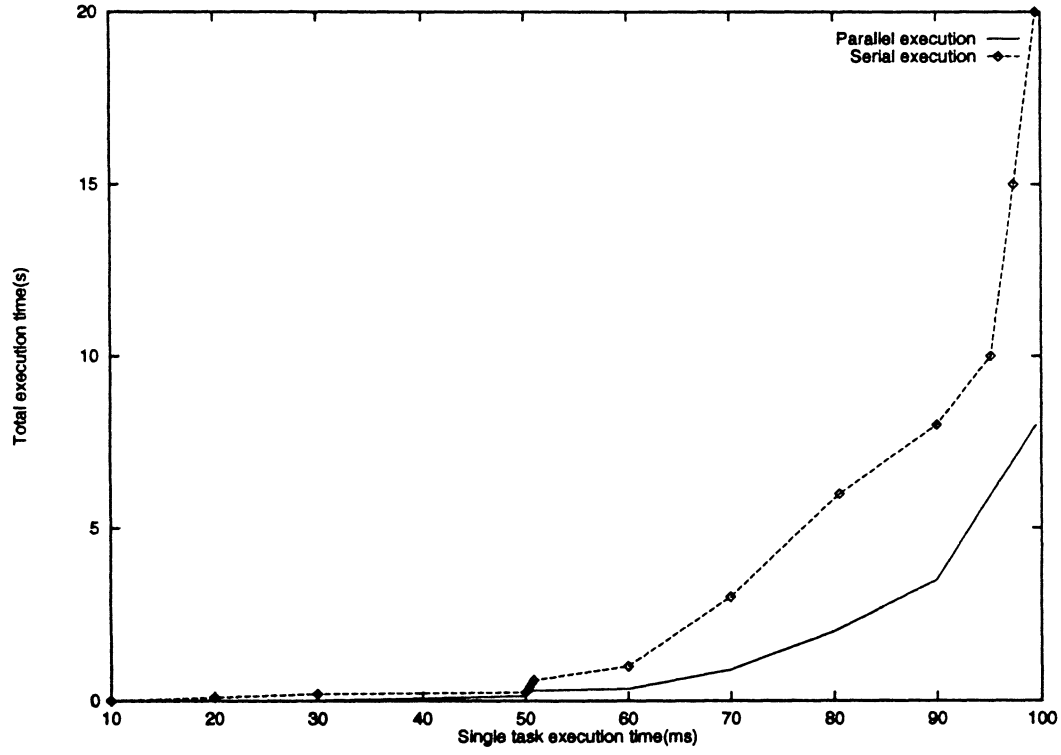


Figure 6.3: Plot of total execution time vs. single task execution: serial execution on Skinner, parallel execution on Barney, Maggie, Lisa and Skinner for k=4.

we increase size of matrices. On the other hand, if we distribute tasks among available idle machines, total execution time is decreased resulting in increased speedup. For this experimentation serial execution of matrix multiplication is carried out on Skinner which is one of the slowest machine among the four stations used in the experiment.

6.1.2 Effect of Slave Node's Load on Speedup

As we have discussed in Chapter 4, the load status of the selected nodes for distributed computing affects the system performance. We have studied this issue by changing the load values of slave nodes. The local load is generated by running matrix multiplication task. We changed load by changing the size of matrices. We found that matrix multiplication of two matrices of size 75×75 takes 10% of the total CPU time and matrices of size 100×100 take 20% of the CPU time on SPARCstation 5 (Maggie, Lisa, and Skinner). Thus by increasing the load factor by 10%, 20% and 40%, the speedup vs subtask level curves for different load values are plotted using equation 4.23 and shown in Fig.6.5. These load values corrospond to matrix sizes of 100×100, 105×105, and 115×115, respectively. We can see from these plots that as the load values of the slave nodes increase, the speedup drops. Hence heavily loaded nodes are not suitable for remote task execution. Load measurement in this experiment is done by monitoring CPU idle time using System Activity Reporter (SAR) program. The general syntax of SAR program is *SAR n t*; i.e., SAR samples the CPU activity counter in the operating system at n intervals of t seconds. A sample output of SAR for Barney is shown in Fig. 6.4. In this figure, %usr, %sys, %wio, and %idle are portion of CPU time running in user mode, running in system mode, idle with some process waiting for block I/O, and otherwise idle, respectively.

17:45:47	%usr	%sys	%wio	%idle
17:45:57	50	4	0	46
17:46:07	16	2	0	83
17:46:17	12	2	0	86
17:46:27	60	5	0	35
17:46:37	11	2	0	87
17:46:47	60	3	0	37
17:46:57	27	3	0	70
17:47:07	43	4	0	53
17:47:17	0	0	0	100
17:47:27	0	0	0	100
Average	28	2	0	70

Figure 6.4: Sample output of SAR for t=10(sec), n=10 for Barney.

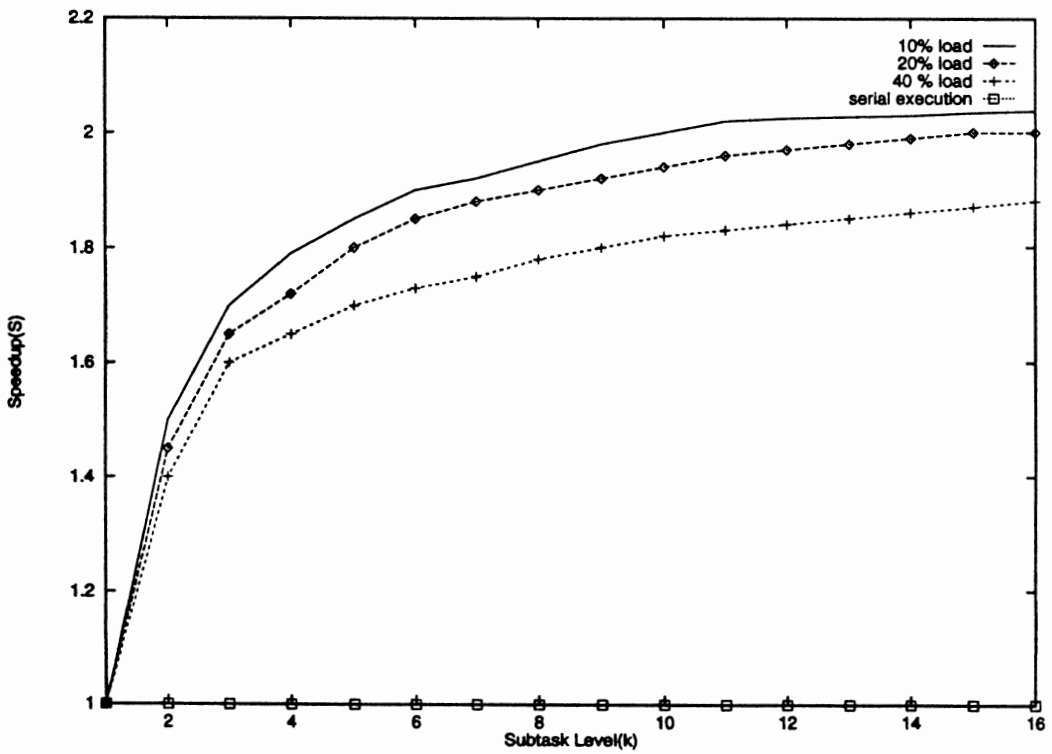


Figure 6.5: Plot of speedup vs. subtask level for m =25, n= 4, d1=d2=0.1 and different loads.

6.1.3 Effect of Subtask Level on Speedup

The effect of subtask level on the speedup has been studied by changing the number of slave nodes and the number of task distribution among them. The result is shown in Fig.6.6. Speedup drastically drops if we increase the number of subtask levels. This is due to the communication overhead for increased number of slave nodes and delays in interprocess communication.

6.2 Communication and Scheduling Delays

The delays involved in scheduling and information transfer among different nodes play important role in the performance of the distributed system. Since we are using LAN with TCP/IP as a communication protocol, we study the behavior of the system using this protocol.

In our case, scheduling delay consists of time required to search the idle node table, receive a response from servers, and prepare tasks for migration. Server response time and time required to migrate a task are negligible compared to time required for searching the statistical load table. The size of the statistical load table is proportional to the number of nodes. Hence the scheduling delay is measured by changing the size of statistical load table and the number of nodes. The results obtained are shown in Fig.6.7. As can be seen from the plot, the scheduling delay increases with increase in number of nodes. The worst case time was approximately 5.2ms.

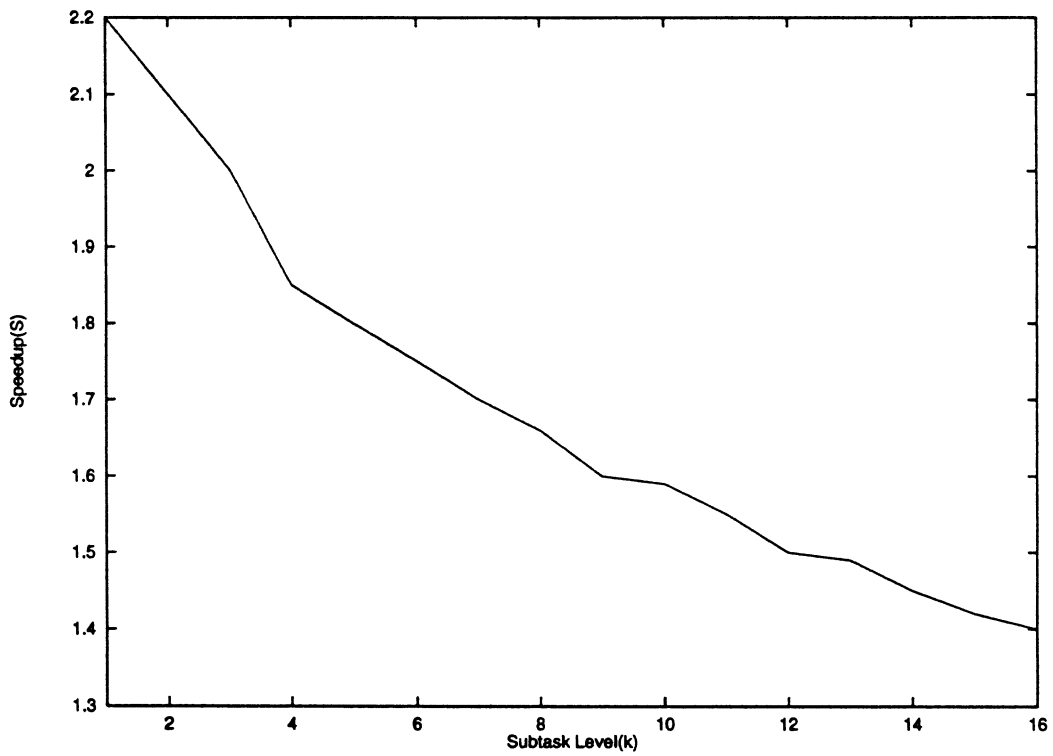


Figure 6.6: Plot of speedup vs. subtask level for m =25, n=4, d1=d2=0.5.

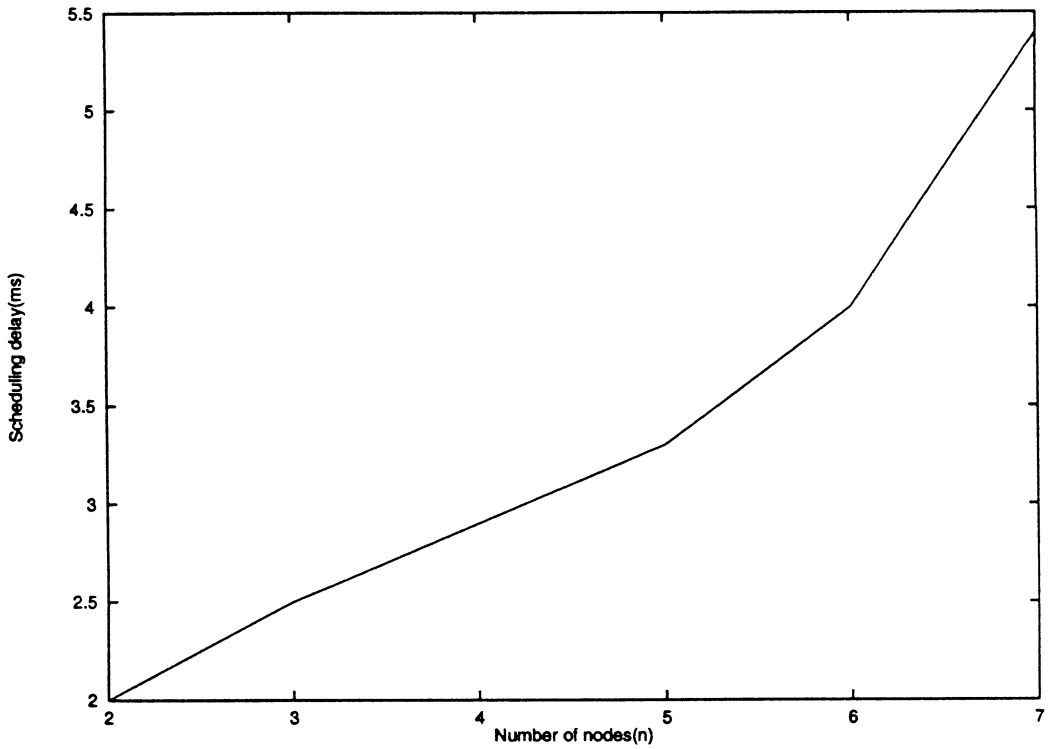


Figure 6.7: Plot of scheduling delay vs. number of nodes.

The major portion of the communication delay is caused by TCP connection time. The connection time is measured using *tcpdump* and *tcptrace* programs. The connection time between client (Barney) and any server machine was approximately 180ms. This large value is due to the three way handshaking mechanism of TCP/IP [23].

Chapter 7

CONCLUSIONS AND FURTHER RESEARCH

In this research we have attemted to use the idle workstations for distributed computing. We have designed a scheme to locate idle or lightly loaded nodes in the network. Depending upon the load condition of a particular workstation, the scheduling scheme described in this research distributes the tasks among the pool of idle workstations.

We have studied the performance of LAN of Sun workstations by evaluating the speedup in different conditions. We have plotted various graphs of speedup vs. level of subtask, number of nodes, and various other communication parameters. We can summarize these results as follows:

- Speedup is strongly influenced by communication and scheduling delay.
- Speedup drops as we increase the level of subtasks.
- As long as number of subtask level is large, increase in number of nodes does not improve the speedup.
- Since the TCP connection overhead is significant (approximately 180 ms), the execution time of a single task must be approximately 44.44 ms to expect a speedup gain from the network for distributed task execution.

- Speedup improves as the single task execution time is increased because the communication overhead becomes insignificant compared to execution time.

Considering our results, we can imply that there are several issues which need to be addressed in this research area. They can be summarized as follows:

1. We have determined that the TCP connection overhead is the major bottleneck in achieving the high throuhput. Designing a new protocol to minimize the connection overhead is necessary to achieve practical feasibility of distributed computig in a network of workstations.

2. The existing operating systems do not support distributed computing. Developing new operarting system which can be effective in a client-server model is a promising topic.

3. In the complex applications, parallelizing the tasks is a major challenge. In this regard, a new programming language for parallel and distributed computing is a topic of further study.

4. Using connectionless protocol for such applications could be another topic of further study.

Bibliography

[1] G.J. Popek and B. J. Walker, "The locus distributed system architecture," Journal of the Audio Eng. Soc., Vol.34, pp.153-166, March 1986.

[2] D.L. Black, "Scheduling support for concurrency and parallelism in mach operating system," IEEE Computer, pp.35-43, May 1990.

[3] M.F. Kaashoek, H.E. Bal, and A.S. Tanenbaum,"Orca: A language for parallel programming of distributed systems," IEEE Trans. on Software Engineering. Vol. SE-18, No.3, pp.190-205, March 1992.

[4] N. Carriero, S. Ahuja, and D. Gelernter, "Linda and Friends," IEEE Computer, Vol.19, pp.26-34, Aug. 1986.

[5] S. Zhou, "A trace-drive simulation study of dynamic load balancing," IEEE Trans. on Software Engineering, Vol. SE-14, pp.1327-1341, Jan. 1987

[6] V. Rego, J.C. Gomez, and V.S. Sunderam, "Efficient multithreaded user space transport for network computing: Design and test of the trap protocol," Journal of Parallel and Distributed Computing, Vol.40, pp.103-117, Nov. 1997.

[7] G. Cabillic and I. Puaut, "An environment for parallel programming on network of hetrogeneous worstations," Journal of Parallel and Distributed Computing, Vol.40. pp.65-80, Nov. 1997.

[8] D.A. Nichols, "Using idle workstations in a shared computing environment," Proc. Eleventh ACM Symp. Oper. Syst. Principles, pp.5-12, 1987.

[9] V.S. Sundaram, "PVM: A framework for parallel programming on a network of hetrogeneous workstations," Concurrency: Practice and Experience, Vol.2, No. 4, pp.315-339, Dec. 1990.

[10] H. Clark and B. McMillin, "DWAGS: a distributed computer server utilizing idle workstations," Journal of Parallel and Distributed Computing, Vol.14, pp.175-186, June 1992.

[11] D.R. Cheriton, "The V distributed system", Comm. ACM, Vol.31, pp.314-333, Oct. 1988.

[12] C. Kesselman, S. Tuecke, I. Foster, and J. Geisler, "Managing multiple communication methods in high performance networked computing systems," Journal of Parallel and Distributed Computing, Vol.40, pp.35-48, Nov. 1997.

[13] J. Reidl, J. Andreson, D. Johnson, and D. Lilja, "Low cost high performance barrier synchronization on networks of workstations," Journal of Parallel and Distributed Computing, Vol. 40, pp131-137, Nov. 1997.

[14] K. Lantz and M. Theimer, "Finding idle machine in a workstation based distributed system," IEEE Trans. on Software Engineering, Vol.5, pp.1444-1457. Nov. 1989.

[15] M.W. Mutka, "Estimating capacity for sharing in a privately owned workstation environment," IEEE Trans. on Software Engineering, Vol. SE-18, No.4, pp.319-328, April 1992.

[16] Y. Wang, "Distributed parallel processing in network of workstations," Ph.D. thesis, Ohio University, 1994.

[17] S.H. Bokhari, "Dual processor scheduling with dynamic reassignment," IEEE Trans. on Software Engineering, Vol. SE-5, No.4, pp341-349, July 1979.

[18] F.C.H. Lin and R.M. Keller, "The gradient model load balancing method," IEEE Trans. on Software Engineering, Vol. SE-13, No.1, pp.32-38, Jan. 1987.

[19] L.M. Ni, "A Distributed load balancing algorithm for point local computer networks," Proceedings of COMPCON Computer Networks, pp.116-123, 1982.

[20] E. Lazowska, D. Eager, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," IEEE Trans. on Software Engineering, Vol. SE-12, No.5, pp.662-675, May 1986.

[21] J.W.S. Liu, C. Gao, and M. Railey, "Load balancing algorithms in homogeneous distributed systems," Proc. of Conference on Parallel Processing, pp.302-306, 1984.

[22] M. Celenk and Y. Wang, Distributed computation in local area networks of workstations," Parallel algorithms and applications, Vol.5, pp.79-106, April 1994.

[23] D.E. Comer and D.L. Stevens, **Internetworking with TCP/IP Vol. III: Client Server Programming and Applications.** Prentice Hall , Englewood Cliffs, New Jersey, second edition, 1996.

[24] A.S. Tanenbaum, **Distributed Operating Systems.** Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1995.

[25] V. Bhatkar, L. Patnaik, R. Madan, and N. Rao, "Parallel and distributed signal and image integration problems," Proc. of Indo-US Workshop, 1993.

[26] D.P. Bertsekas and J.N. Tsitsiklis, **Parallel and Distributed Computation Numerical Methods.** Prentice Hall, Englewood Cliffs, second edition, 1989.

[27] M. Cosnard and D. Trystram, **Parallel Algorithms and Architectures.** International Thomson Computer Press, UK, first edition, 1995.

[28] M.J. Quinn, **Designing Efficient Algorithms for Parallel Computers.** McGraw Hill, New York, first edition, 1987.

[29] K. Hwang, **Advanced Computer Architecture.** McGraw Hill, New York, 1993.

[30] V. Kumar, A. Grama, A. Gupta, and G. Karypis, **Introduction to Parallel Computing.** Benjamin/Cummings, California, 1994.

[31] S. Ostermann, tcptrace homepage, http://jarok.cs.ohiou.edu/software/tcptrace.

Appendix A

A SAMPLE TCPTRACE OUTPUT

For *tcptrace* program we have used, tcpdumpop file is created by *tcpdump* program. To create this file we run *tcpdump* as **tcpdump host homer and maggie.** This will print the traffic between *homer* and *maggie.* To run *tcptrace* program we type **tcptrace -l tcpdumpop.** Following is the typical ouput.

```
1 args remaining, starting with '/home/homer/1/b/kore/thesis/sw/tcpdumpop'
Ostermann's tcptrace -- version 4.1.0 -- Fri Aug 22, 1997

Running file '/home/homer/1/b/kore/thesis/sw/tcpdumpop'
1 connection traced:
11 packets seen, 11 TCP packets traced
connection 1:
host a:         maggie:576
host b:         homer.ece.ohiou.edu:2049
complete conn: no (SYNs: 0)  (FINs: 0)
first packet:  Tue Feb 10 17:40:53.312131
last packet:   Tue Feb 10 17:40:53.371235
elapsed time:  0:00:00.059104
total packets: 11
   a->b:     b->a:
     total packets:       6          total packets:     5
     ack pkts sent:       6          ack pkts sent:     5
     unique bytes sent:   560        unique bytes sent  580
     actual data pkts:    5          actual data pkts:  5
     actual data bytes:   560        actual data bytes  580
     rexmt data pkts:     0          rexmt data pkts:   0
     rexmt data bytes:    0          rexmt data bytes:  0
```

outoforder pkts:	0	outoforder pkts:	0
SYN/FIN pkts sent:	0/0	SYN/FIN pkts sent	0/0
mss requested:	0 bytes	mss requested:	0 bytes
max segm size:	112 bytes	max segm size:	116 bytes
min segm size:	112 bytes	min segm size:	116 bytes
avg segm size:	111 bytes	avg segm size:	115 bytes
max win adv:	8760 bytes	max win adv:	8760 bytes
min win adv:	8760 bytes	min win adv:	8760 bytes
zero win adv:	0 times	zero win adv:	0 times
avg win adv:	8760 bytes	avg win adv:	8760 bytes
initial window:	112 bytes	initial window:	0 bytes
initial window:	1 pkts	initial window:	0 pkts
throughput:	9475 Bps	throughput:	9813 Bps
ttl stream length:	NA	ttl stream length:	NA
missed data:	NA	missed data:	NA

Appendix B

EXPERIMENTAL PROCEDURE AND SAMPLE OUTPUTS

To experiment serial and parallel execution of matrix multiplication, Sun workstations Barney, Skinner, Lisa, and Maggie are used. *clientsim* and *serversim* are the client and server programs which are installed on these workstations. In the following discussion **bold** phrase represents a command and *italic* phrase represents the user output.

A) Serial Execution: Suppose we need to see the serial execution of multiplication of two matrices of order 10×10. After login, at the system prompt user types **mat 10**. Here, 10 is the size of the matrix. Following is the output of this program. To get different size of matix, user types **mat n**, where **n** is the desired matrix size.

```
Matrix A =

38 13 51 10 12 49 84 25 89 37
66 95 67 31 82 24 94 51 54 61
28 85 67 65 62 53 44 62 22 57
79 51 70 48 29 6 38 14 16 81
43 33 60 71 36 60 29 75 14 88
2 34 82 86 6 51 73 93 48 20
56 66 93 91 26 85 10 52 20 39
85 25 14 76 43 19 23 65 73 13
99 13 26 46 36 78 73 71 52 91
22 28 63 41 75 63 44 13 61 45
```

```
Matrix B =

58 15 27 19 86 67 60 43 83 66
78 11 54 45 36 5 2 67 53 96
88 43 6 3 34 20 3 18 27 45
83 94 59 57 6 90 16 62 25 34
22 36 79 1 61 5 44 93 39 21
71 17 15 53 74 48 34 10 10 49
62 97 10 99 5 82 0 65 80 4
75 65 71 97 97 87 49 56 1 57
67 98 11 66 88 26 87 17 88 83
44 85 59 76 78 85 89 11 26 96

C = A x B =

26953 26751 10084 24066 23451 21465 17074 14429 21967 21744
39170 34100 24735 31304 33265 29323 21897 30752 30444 34064
36032 28778 23723 27288 28456 26764 17646 25764 20211 30140
27810 23463 16849 19421 22785 23336 16587 18224 20109 25849
33526 29118 22197 27503 29504 30443 20311 20781 16632 28817
36472 32627 17736 30361 24010 29187 14321 21229 17753 24132
38316 26125 20645 24349 28672 27521 17020 21745 18422 30430
28479 25882 18310 22552 27029 24611 19639 21002 21192 24162
36531 34219 21971 33798 37036 36532 27118 23182 25795 32485
28300 25888 16462 20681 24884 20113 17612 19223 19662 23331


Execution time: 0.007072s
```

B)Parallel Execution: For parallel execution, we open four terminal windows on the workstation. We telnet to three remote workstations in three windows to run server and use one window on local machine to run client. We run *serversim* on remote machines. After running *serversim*, the server waits for connection request

indefinetely. We run *clientsim* on local machine. After running *clientsim* program on local machine, we follow the system prompts and enter data. Following is the example.

```
 %lisa serversim

 %maggie serversim

 %skinner serversim


 %barney clientsim


Please enter the number of nodes you need 2
 enter host name: maggie.ece.ohiou.edu
enter service ID:3000
 enter host name: skinner.ece.ohiou.edu
enter service ID:3000

 How many matrix elemants in matrix A ? 6

 Enter the matrix elements and task ID:
1 1

 Enter the matrix elements and task ID:
3 2

 Enter the matrix elements and task ID:
4 3

 Enter the matrix elements and task ID:
2 4

 Enter the matrix elements and task ID:
0 5

 Enter the matrix elements and task ID:
1 6
```

```
The element 0 is E = 1, ID number = 1
The element 1 is E = 3, ID number = 2
The element 2 is E = 4, ID number = 3
The element 3 is E = 2, ID number = 4
The element 4 is E = 0, ID number = 5
The element 5 is E = 1, ID number = 6


How many levels in the task graph ?2



How many tasks in level 0 ? 3


Please enter task 0 ID in level 0: 1

Please enter task 1 ID in level 0: 2

Please enter task 2 ID in level 0: 3

How many tasks in level 1 ? 4


Please enter task 0 ID in level 1: 5

Please enter task 1 ID in level 1: 6

Please enter task 2 ID in level 1: 3

Please enter task 3 ID in level 1: 2


The computation table is

level 0:  1  2  3
level 1:  5  6  3  2
```

```
***You connect to the following hosts ***
maggie.ece.ohiou.edu
skinner.ece.ohiou.edu

The total execution time 0.034765 us
```

Appendix C

SOURCE CODE

```
/* ---------------------------------------------------------------
    Matrix Multiplication Sequentially.
    This program allows you to size of matrix at command and generates
    two random matrices of given size. Then it does the matrix
    multiplication of these random matrices. It also calculates
        execution time.
    ---------------------------------------------------------------
*/

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <string.h>
#include <sys/times.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/types.h>


#define BOUND 100
#define BUFFSIZE 100000

#define TYPE int
#define DEMO 1

double clock1, ucpu1, scpu1;
double clock2, ucpu2, scpu2;
double clock3, ucpu3, scpu3;
```

```
int size;
char *buff;

int My_Id, Num_Procs;
int Dim;

void print(TYPE **matrix, int num);

int main( argc, argv )
int     argc;
char    *argv[];
{
TYPE    **A, **B, **C;
int     i, j, k;
double  start, stop, total_time;

 /* --- find out dimension --- */
Dim = atoi( argv[argc-1] );

/*start timer*/

gettimeusage(&clock1, &ucpu1, &scpu1);
/* --- allocate memory --- */
if( (A=(TYPE**)malloc(Dim*sizeof(TYPE))) == NULL )
{
printf("Cannot allocate memory\n");
}
if( (B=(TYPE**)malloc(Dim*sizeof(TYPE))) == NULL )
        {
 printf("Cannot allocate memory\n");
}
if( (C=(TYPE**)malloc(Dim*sizeof(TYPE))) == NULL )
{
printf("Cannot allocate memory\n");
}
/* --- allocate memory --- */
for( i = 0; i < Dim; i++ )
{
A[i] = malloc(Dim*sizeof(TYPE));
```

```
B[i] = malloc(Dim*sizeof(TYPE));
C[i] = malloc(Dim*sizeof(TYPE));
}
/* --- constructing two random matrices --- */
for( i = 0; i < Dim; i++ )
for( j = 0; j < Dim; j++ )
{
A[i][j] = (rand() % BOUND) / DEMO;
B[i][j] = (rand() % BOUND) / DEMO;
C[i][j] = 0 / DEMO;
}

printf("\n\nMatrix A =\n\n");
print(A,Dim);
printf("Matrix B =\n\n");
print(B,Dim);
 /* --- do matrix multiplication --- */
for( i = 0; i < Dim; i++ )
for( j = 0; j < Dim; j++ )
{
for( k = 0; k < Dim; k++ )
C[i][j] += A[i][k] * B[k][j];
}

gettimeusage(&clock2, &ucpu2, &scpu2);
clock3 = clock2-clock1;
ucpu3 = ucpu2-ucpu1;
scpu3 = scpu2-scpu1;


printf("C = A x B is\n\n");
print(C, Dim);

printf("-----------------------------------------------\n");
        printf("  Clock time    User CPU time   System CPU time\n");
        printf("-----------------------------------------------\n");
        printf("   %lf  %lf        %lf\n",
                        clock3, ucpu3, scpu3);
/* --- free memory --- */
for( i = 0; i< Dim; i++ )
```

```
{
free(A[i]);
free(B[i]);
free(C[i]);
}
free(A); free(B); free(C);
}
void print(TYPE **matrix, int num)
{
int     i, j;
for( i = 0; i < num; i++ )
{
for( j = 0; j < num; j++ )
printf("%d\t", matrix[i][j]);
printf("\n");
}
printf("\n");
}


void gettimeusage(clk, ucpu, scpu)
double *clk, *ucpu, *scpu;
{
struct rusage rusage_now;
struct timeval tnow;
struct timezone tz;
gettimeofday(&tnow, &tz);
getrusage(RUSAGE_SELF, &rusage_now);
*clk = tnow.tv_sec + tnow.tv_usec/1000000.0;
*ucpu=rusage_now.ru_utime.tv_sec +
rusage_now.ru_utime.tv_usec/1000000.0;
*scpu=rusage_now.ru_stime.tv_sec +
rusage_now.ru_stime.tv_usec/1000000.0;
return;
}
```

```
/*--------------------------------------
connectTCP.c
connect to the specified TCP service
on a specified host
--------------------------------------
*/

int connectTCP(host, service)
char *host;
char *service;
{
return connectsock(host, service, "tcp");
}


/*---------------------------------
 connectsock.c
 Connect a socket using TCP or UDP
  ---------------------------------
  */


  #include <sys/types.h>
  #include <sys/socket.h>
  #include <netinet/in.h>
  #include <netdb.h>
  #include <arpa/inet.h>

  #ifndef INADDR_NONE
  #define INADDR_NONE     0xffffffff
 #endif

  extern int      errno;
  extern char     *sys_errlist[];

u_short htons();
u_long inet_addr();


int connectsock(host, service, protocol)
```

```
char    *host;
char    *service;
char    *protocol;
  {

 struct hostent  *phe; /* host info pointer */
  struct servent *pse;  /* service info pointer */
  struct protoent *ppe; /* protocol info pointer */
  struct sockaddr_in sin; /* internet endpoint address */
  int s, type;            /* socket descriptor and type */

   bzero((char *) &sin, sizeof(sin));
   sin.sin_family = AF_INET;

   /* map service name to port number */

   if (pse = getservbyname(service, protocol))
   sin.sin_port = pse->s_port;
   else if ((sin.sin_port = htons((u_short)atoi(service))) == 0)
   errexit("can't get \"%s\" host_entry\n", service);
   if(phe = gethostbyname(host))
    bcopy(phe->h_addr, (char *) &sin.sin_addr, phe->h_length);
    else if ((sin.sin_addr.s_addr = inet_addr(host))==INADDR_NONE)
    errexit("cant get \"%s\" host entry\n", host);

    /* map protocol name to protocol number */

   if((ppe = getprotobyname(protocol)) ==0)
     errexit("can't get \"%s\" protocol entry\n", protocol);

 /* Use protocol to choose a socket type */

  if(strcmp(protocol, "udp") == 0)
  type = SOCK_DGRAM;
   else
   type = SOCK_STREAM;

   /* Allocate a socket */

  s = socket(PF_INET, type, ppe->p_proto);
```

```
 if(s < 0)
if(s < 0)
 errexit("can't create socket: %s\n", sys_errlist[errno]);

 /*connect the socket */

 if(connect(s, (struct sockaddr *)&sin, sizeof(sin)) <0)
 errexit("can't connect to %s.%s: %s\n", host, service,
 sys_errlist[errno]);
 return s;
  }


/*-------------------------------------
 passivesock.c
 Allocates and binds a server
 using TCP or UDP
  -------------------------------------
  */

 #include <sys/types.h>
  #include <sys/socket.h>

 #include <netinet/in.h>

  #include <netdb.h>

  extern int errno;
  extern  char *sys_errlist[];

  u_short htons(), ntohs();

  u_short portbase = 0;

  int passivesock(service, protocol, qlen)
  char    *service;
  char    *protocol;
  int     qlen;
  {
    struct servent *pse;
```

```
    struct protoent *ppe;

    struct sockaddr_in sin;
    int s, type;
    bzero((char *)&sin, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl(INADDR_ANY);

    /*Map service name to port number */

    if(pse = getservbyname(service, protocol))
     sin.sin_port = htons(ntohs((u_short)pse->s_port)
                     + portbase);
   else if ((sin.sin_port = htons((u_short)atoi(service))) ==0)
     errexit("can't get \"%s\" service entry\n", service);

     /*Map protocol name to protocol number */

     if((ppe = getprotobyname(protocol)) == 0)
     errexit("can't get \"%s\" protocol entry\n", service);

     /* Use protocol to choose a socket type */

     if(strcmp(protocol, "udp") == 0)
     type = SOCK_DGRAM;
     else
     type = SOCK_STREAM;

     /* Allocate a socket */

     s = socket(PF_INET, type, ppe->p_proto);
     printf("This is the value of s -->%d", s);
     if (s<0)
    errexit("can't create socket: %s\n", sys_errlist[errno]);

/*Bind socket */

if(bind(s, (struct sockaddr *)&sin, sizeof(sin)) <0)

errexit("can't bind to %s port: %s\n", service,
```

```
        sys_errlist[errno]);
if(type == SOCK_STREAM && listen(s, qlen) < 0)
errexit("can't listen on %s port: %s\n", service,
       sys_errlist[errno]);
return s;

 }

/*----------------------------
passiveTCP.c
creats a passive socket
----------------------------
*/
int passivesock(const char *service,
const char *transport, int qlen);

int passiveTCP(service, qlen)

char *service;
int qlen;
{
 return passivesock(service, "tcp", qlen);

   }

/*-----------------------------------
errexit.c
prints an errors message and quits.
-----------------------------------
*/

#include <varargs.h>
#include <stdio.h>

errexit(format, va_alist)
char *format;
va_dcl
  {
   va_list args;
   va_start(args);
```

```
   _doprnt(format, args, stderr);
   va_end(args);
   exit(1);
    }

  /*-----------------------------------------------------------------
    clientsim.c  This is the client program for matrix multiplication
    -----------------------------------------------------------------
*/

#include "sim.h"

#define TEST2
/*#define TEST1 */
/*testtime(double *); */

struct table_1 *FindIdleTable();
double clock1, ucpu1, scpu1;
double clock2, ucpu2, scpu2;
double clock3, ucpu3, scpu3;



main(argc, argv)
int argc;
char *argv[];
{
union wait *status;
int childpid;
struct cmlist cmtable[MAXNODE];
int pipe1[2], pipe2[2];
int context;
char message[MAXBUFF];
int cptable[MAXTASK][MAXTASK], level[MAXTASK], totallevel;
char ans[1], cptable2[MAXTASK][MAXTASK][MAXNUM];
int nodenum, diftasknum;
char tasknum[MAXTASK][MAXNUM];
int taskID[MAXTASK];
struct table_1 *idletable;
char *host[MAXNODE], *service[MAXNODE];
```

```
int socknum[MAXNODE];

double overhead;
struct cmlist *ptr;
int host_num, i,j,k;

printf("\n\n Have you created the statistical load table? (y/n)");

scanf("%ls", ans);
if((strcmp(ans, "n")==0)|| (strcmp(ans, "N") == 0)){
printf("\n\n Create a statistical load table for idle nodes(y/n)?");
scanf("%ls", ans);
if((strcmp(ans, "y") == 0)||(strcmp(ans,"Y") ==0))
createcmtable(&nodenum, cmtable);
}

diftasknum = gettask(tasknum, taskID);
printf("\n\n PLease create computation list \n\n");
totallevel = createcptable(cptable, level);

for(j=0; j<totallevel; j++)
  for(i=0; i<level[j]; i++)
for(k=0; k<diftasknum; k++) {
    if(cptable[j][i] == taskID[k]) {
strcpy(cptable2[j][i], tasknum[k]);
break;
}
   }

if(k == diftasknum)
printf("level %d task %d ID number %d is
incorrect\n", j,i, cptable[j][i]);

nodenum = 2; /*debug*/
#ifdef TEST2
/*testtime((double *)0);*/
#endif

printf("nodenum = %d\n", nodenum);
if(nodenum==0)
```

```
printf("read: file error\n");
#ifdef TEST3
testtime((double *)0);
#endif
idletable = FindIdleTable(cmtable, nodenum);
gettimeusage(&clock1, &ucpu1, &scpu1);
findhost(idletable, host, service, nodenum);
        for(i=0;i<nodenum; i++)
socknum[i] = connectTCP(host[i], service[i]);

    for(i=0;i<nodenum; i++)
if(Request(socknum[i]) <0)
   printf("Host %s is busy! \n", host[i]);
    for(j=0; j<totallevel; j++){
for(i=0; i<level[j]; i++) {
   if(fork() ==0){
exit(runtask(socknum[i], cptable2[j][i]));
}
}
for(i=0; i<level[j]; i++)
   wait(&status);
}

#ifdef TEST2
/*testtime(&overhead);*/
gettimeusage(&clock2, &ucpu2, &scpu2);

printf("The total execution time %lf us\n", (ucpu2-ucpu1));
#endif

}

int computeload(cmtable)
struct cmlist *cmtable;
{
register int i, j;
FILE *fp;
int nodenum;
if((fp = fopen("cmtable", "rb")) == NULL) {
perror("open file");
```

```
return -1;
  }
fscanf(fp, "%3d", &nodenum);
for(i=0; i<nodenum; i++){
fscanf(fp, "%10s%10s", cmtable[i].name, cmtable[i].nodeID);
for(j=0; j<INTERVAL; ++j)
fscanf(fp,"%2d", &cmtable[i].percent[j]);
}
fclose(fp);
return nodenum;
}


testtime (overhead)
double *overhead;
{
static struct timeval begin;
struct timeval now;
double start, stop;
if(gettimeofday(&now, (struct timezone *)0) < 0)
perror("gettomeofday");
if(!overhead) {
begin = now;
return;
}
start = ((double) begin.tv_sec) * 1000000.0
+ begin.tv_usec;
stop = ((double) now.tv_sec) * 1000000.0
+ now.tv_usec;
*overhead = (stop - start);
}

void gettimeusage(clk, ucpu, scpu)

        double *clk, *ucpu, *scpu;
        {
struct rusage rusage_now;
struct timeval tnow;
struct timezone tz;
gettimeofday(&tnow, &tz);
```

```
getrusage(RUSAGE_SELF, &rusage_now);
*clk = tnow.tv_sec + tnow.tv_usec/1000000.0;
*ucpu=rusage_now.ru_utime.tv_sec +
rusage_now.ru_utime.tv_usec/1000000.0;
*scpu=rusage_now.ru_stime.tv_sec +
rusage_now.ru_stime.tv_usec/1000000.0;
return;
        }


load_table(nodenum, cmtable)
struct cmlist *cmtable;
int nodenum;
{
register int num, i, j;
int seeds;
FILE *fp;
char Name[10];

seeds = SEED;

printf("Please enter the number of nodes you need");
scanf("%d", &nodenum);
fp = fopen("cmtable", "wb");
fprintf(fp, "%-3d", nodenum);
for (i=0; i<nodenum; i++) {
  printf(" enter host name: ");
  scanf("%s", cmtable[i].name);
  fprintf(fp, "%-10s", cmtable[i].name);
  printf("enter service ID:");
  scanf("%s", cmtable[i].nodeID);
  fprintf(fp, "%-10s", cmtable[i].nodeID);

  for(j=0; j<INTERVAL; ++j){
cmtable[i].percent[j]=(ranl(&seeds));
fprintf(fp, "%2d", cmtable[i].percent[j]);
  }
    }
fclose(fp);
} /*end load_table */
```

```
int complist(taskID, levelnum)
int taskID[MAXTASK][MAXTASK];
int levelnum[MAXTASK];
{
int i, j, mj, k;
printf("\n How many levels in the task graph ?");
scanf("%d", &k);
printf("\n\n");
for(j =0; j<k; j++){

 printf("\n How many tasks in level %d ? ", j);
 scanf("%d", &mj);
 printf("\n");
 levelnum[j] = mj;
 for(i=0; i<mj; i++){

 printf(" \n Please enter task %d ID in level %d: ", i,j);
 scanf("%d", &taskID[j][i]);
}
     }

printf("\n\n The computation table is \n\n");

for(j =0; j<k; j++){
  printf(" level %d: ", j);
   for(i=0; i<levelnum[j]; i++)
    printf(" %d ", taskID[j][i]);
printf("\n");
    }
printf("\n\n");
return k;
}

int sendtask(NUM, ID)
char NUM[MAXTASK][MAXNUM];
int *ID;

{
```

```
int i,n;
    printf("\n How many matrix elemants in matrix A ? ");
scanf("%d", &n);

for(i=0; i<n; i++){
 printf("\n Enter the matrix elements and task ID:\n");
 scanf("   %s  %d", NUM[i], &ID[i]);
 }
 printf("\n\n");
 for(i=0; i<n; i++){
    printf(" The element %d is E = %s, ID number = %d \n",
i, NUM[i], ID[i]);
}
return n;
     }
int Request (s)
int s;
{
char message[LINELEN+1];
int outchars, n;
outchars = strlen(RES_SIG);
(void) write(s, RES_SIG, outchars);

bzero(message, sizeof(message));
n = read(s, message, sizeof(message));
if (n < 0)
errexit("socket read failed: %s\n",
sys_errlist[errno]);
else
 if (strcmp(message, "Available") ==0)
    return 0;
else
   return -1;
}

int runtask(s, buf)
int  s;
char *buf;
{
int  n;
```

```
int outchars, inchars;
char message[LINELEN+1];
double overhead;
printf("Inside runtask now...\n");

#ifdef TEST1
testtime((double *)0);
#endif
/****send task name*****/

outchars = strlen(buf);
(void) write(s, buf, outchars);
bzero(message, sizeof(message));
n = read(s, message, sizeof(message));
if (n<0)
 errexit("socket read failed: %s\n", sys_errlist[errno]);

#ifdef  TEST1
testtime(&overhead);
printf("Run %s overhead is %lf us\n", message, overhead);
#endif
return 0;
}

/*-------------------------------------------------------------
  serversim.c This is the server program which runs on remote nodes
  -------------------------------------------------------------
 */

#include <sys/types.h>
#include <sys/signal.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/times.h>
#include <sys/resource.h>
#include <sys/wait.h>
#include <sys/errno.h>
#include <netinet/in.h>

#include <stdio.h>
```

```
#define QLEN 5
#define BUFSIZE 4096
#define RESPONSE "Remote execution successful"
#define AVAI_SIG "Available"
#define MAXTASK 100
#define EXEC

extern int errno;
extern char *sys_errlist[];

int reaper();
int pid;

double clock1, ucpu1, scpu1;
double clock2, ucpu2, scpu2;
 double clock3, ucpu3, scpu3;


int main()
       /*
int argc;
char *argv[];
       */
{
char host[10]="maggie";

char *service;
struct sockaddr_in fsin;
int alen;
int msock, ssock;
/*
switch(argc) {
case 1:
break;

case 3:
service =argv[1];
break;
default:
```

```
errexit("usage:server [port number] [node's name]\n");
}
*/
service="3000"; /*debug*/

printf("\n");
printf("Now waiting to run the task requested by master node!\n");
printf("\n");

msock = passiveTCP(service, QLEN);

(void) signal(SIGCHLD, reaper);

while (1){

alen = sizeof(fsin);
ssock = accept(msock, (struct sockaddr *)&fsin, &alen);
if(ssock < 0){
if(errno ==EINTR)
continue;
errexit("accept failed: %s\n", sys_errlist[errno]);
}
/*
pid = fork();
switch (pid){
case 0:
(void) close(msock);
        */

exit(runrmtask(ssock,host));
/*

default:
(void) close(ssock);
break;
case -1:
errexit("fork: %s\n", sys_errlist[errno]);
}
       */
}
```

```
   }


int runrmtask (fd, hname)
int fd;
char *hname;
{
union wait *status;
char buf[BUFSIZE], message[30];
int cc, i,k,j;
char *name, *malloc();
double overhead;

bzero(buf, sizeof(buf));

while (cc =read(fd, buf, sizeof(buf))){
if(cc <0)
errexit("echo read: %s\n", sys_errlist[errno]);
if(strcmp(buf, "Request") ==0)
write(fd, AVAI_SIG, sizeof(AVAI_SIG));
else {
name = malloc(cc);
bzero(name, cc);
strncat(name,buf,cc);
printf("Server received %s-command\n", name);
#ifdef EXEC
  /*testtime((double *)0);*/

gettimeusage(&clock1, &ucpu1, &scpu1);
#endif
k=atoi(name);

gettimeusage(&clock2, &ucpu2, &scpu2);
   j+=task(k);
gettimeusage(&clock3, &ucpu3, &scpu3);
                printf("atoi overhead= %lf s \n",ucpu2-ucpu1);
printf("Execution time= %lfs\n", (ucpu3-ucpu2)-(ucpu2-ucpu1));
printf("Sum of i's is %d\n",j);
#ifdef EXEC
```

```
  /*testtime(&overhead);*/
        /*
gettimeusage(&clock2, &ucpu2, &scpu2);
printf("------------------------------------------------\n");
printf("  Clock time    User CPU time   System CPU time\n");
printf("------------------------------------------------\n");
printf("   %lf  %lf        %lf\n",
                        clock2-clock1, ucpu2-ucpu1, scpu2-scpu1);
        */

  /*printf("Overhead= %lf\n", overhead);*/
#endif
if(write(fd, hname, sizeof(hname)) < 0)
errexit("echo write: %s\n", sys_errlist[errno]);
free(name);
   }
bzero(buf, sizeof(buf));
}
return 0;
 }



void testtime(overhead)
double *overhead;
{
static struct timeval begin;
struct timeval now;
double start, stop;

if(gettimeofday(&now, (struct timezone *)0) < 0)
perror("gettimeofday");
if(!overhead){
begin = now;
return;
}

start = ((double) begin.tv_sec) *1000000.0
+ begin.tv_usec;
stop  = ((double) now.tv_sec)*1000000.0
```

```
+ now.tv_usec;
*overhead = (stop - start);
return;
}



int reaper(){
union wait    *status;


while(wait3(status, WNOHANG, (struct rusage *) 0) >=0)
;
 }

void gettimeusage(clk, ucpu, scpu)
double *clk, *ucpu, *scpu;
{
struct rusage rusage_now;
struct timeval tnow;
struct timezone tz;
gettimeofday(&tnow, &tz);
getrusage(RUSAGE_SELF, &rusage_now);
*clk = tnow.tv_sec + tnow.tv_usec/1000000.0;
*ucpu=rusage_now.ru_utime.tv_sec +
rusage_now.ru_utime.tv_usec/1000000.0;
*scpu=rusage_now.ru_stime.tv_sec +
rusage_now.ru_stime.tv_usec/1000000.0;
return;
        }
/*-------------------------------------------------------
sim.h    header file used in programs
-------------------------------------------------------
*/
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <time.h>
#include <sys/time.h>
#include <sys/times.h>
```

```
#include <sys/resource.h>

#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <sys/time.h>

extern int errno;
extern char *sys_errlist;

#define RES_SIG "Request"
#define LINELEN 10/* original value=128 */
#define INTERVAL 144
#define IDLE_L1 70
#define LIGHT_L2 80
#define MAXBUFF 1024
#define COMPILER   1
#define LINKER 2
#define EXECUTION 3
#define RUN 4
#define LOGIN 5
#define INTERRUPTER 6
#define LOCAL 31
#define REMOTE 32

#define MAXTASK 100
#define MAXNODE 100
#define MAXNUM 20
#define SEED -100
#define DEBUG
#define EXEC
struct cmlist{
char name[25];
char nodeID[10];
int percent[INTERVAL];
};

struct state_1{ /*structure for state table */
char name[25];
char nodeID[10];
```

```
int state;
struct state_1 *next;
};

struct table_1 {
int number;
struct state_1 *next;
};
```