M.Tech. (Computer Science) dissertation Series

Simulation of Task Scheduling For Cluster Computing on One, Two and Three Processor Systems Using PVM : *Parallel virtual Machine*

a dissertation submitted in partial fulfilment of the requirements for the M.Tech. (Computer Science) degree of the INDIAN STATISTICAL INSTITUTE

By
G Surekha

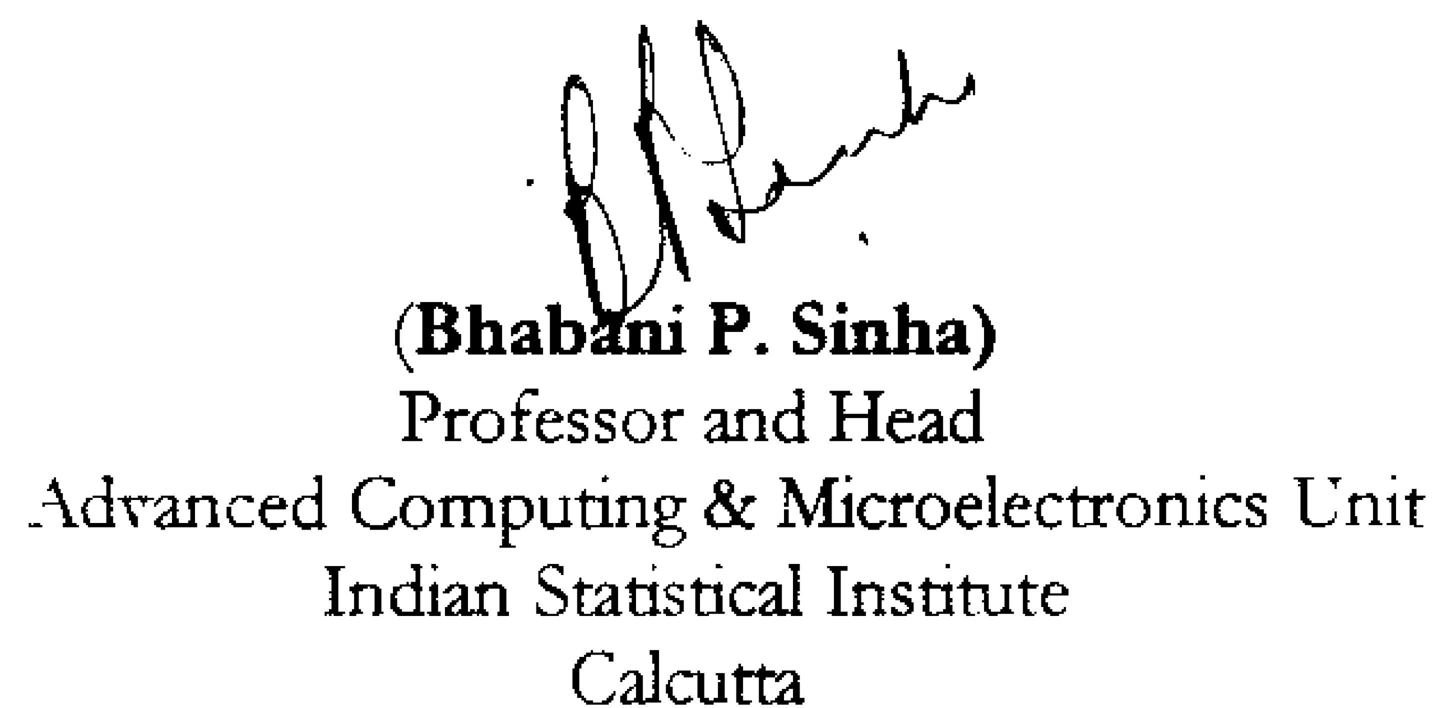
under the supervision of
Prof. Bhabani Prasad sinha
(Advance Computing and Electronics Unit)

INDIAN STATISTICAL INSTITUTE
203, Barrackpore Trunk Road,
Calcutta - 700 035

28th July , 1998

CERTIFICATE OF APPROVAL

This is to certify that the dissertation titled **Simulation of Task Scheduling for Cluster Computing on One, Two & Three Processor systems using PVM** submitted by **Ms. Guntur Surekha** to the **Advanced Computing and Microelectronics Unit, Indian Statistical Institute, Calcutta,** is a bonafide record of the work and investigation carried out by her under my supervision and guidance.



Date: 28h July, 1998
Indian Statistical Institute
Calcutta.

(Bhabani P. Sinha)
Professor and Head
Advanced Computing & Microelectronics Unit
Indian Statistical Institute
Calcutta

ACKNOWLEDGEMENTS

In the first place, I am very grateful to my supervisor, ***Dr. B. P. Sinha***, for his excellent guidance and constant support throughout my work. It was an enlightening experience to work under some of the outstanding educators and researchers at ISI.

I would like to thank ***Mrs. Sharbani Sengupta*** of Electronics Unit for providing her valuable suggestions.

I would like to acknowledge the members of Electronics Unit of ISI, for offering their valuable advice whenever I faced any problem.

I also thank all my classmates for helping me in various stages of my work.

I specially thank my classmates ***Balakrishna Reddy, Sudhakar Rao, Ishita De*** and my friend ***Sarat Jyothsna*** for helping me at various stages throughout my stay at I.S.I. and my senior ***Piyush Ranjan Kumar*** for his constant encouragement throughout the period of the project work.

I also thank my seniors ***Padma Mahalingam & Rekha Menon*** for providing their valuable suggestions.

Calcutta
July 28, 1998

G Surekha

Abstract

With the increased global competition and need for faster and efficient processing, a high demand for parallel computation is created. Even with the most sophisticated technology available so far, a parallel processing system usually involves a large investment. Also there is a need for efficient software support which makes parallel processing user-friendly.

A recent trend is to use Cluster Computing which involves computing on a set of workstations, which are interconnected in the form of a high speed net. This type of network facilitates the access of available resources from different sites connected in the network.

The present dissertation was mainly focussed on ***simulation*** of ***interconnection patterns for a Heterogeneous Network of Workstations*** on ***single, double*** and ***triple*** processors. To improve the efficiency of execution, optimum number of tasks into which a problem is to be divided, is identified for a given number of processors. All the tasks are distributed equally among the processors to ensure that all the processors complete execution at around the same time for effective utilization of resources. Then taking the load (number of ready processes) on each processor into consideration, the tasks are distributed among the processors, to further fasten the execution by giving more number of processes to the machine which has lesser load.

PVM software environment was used to study various operations on matrices on different types of interconnection patterns. The interesting results observed during this study were reported.

Contents

1. **Introduction**
   1.1 Need for parallel computation
   1.2 Design of a parallel algorithm
   1.3 Parallel Processing
   1.4 Distributed memory parallel processing
   1.5 Approach to parallize codes
      1.5.1 Semi-automatic parallel programming
      1.5.2 Data parallel programming
      1.5.3 Message Passing

2. **Heterogeneous processing**
   2.1 Software platform for heterogeneous resources
   2.2 Programming speed up in heterogeneous computing network
      2.2.1 Heterogeneous network model

3. **Software environment**
   3.1 Common parallel programming paradigms
      3.1.1 Crowd computation
      3.1.2 Tree computation
      3.1.3 Hybrid computation
   3.2 Work load allocation
      3.2.1 Data decomposition
      3.2.2 Function decomposition
   3.3 PVM user interface
      3.3.1 Process control
      3.3.2 Information
      3.3.3 Dynamic configuration
      3.3.4 Setting and getting options
      3.3.5 Message passing
   3.4 Dynamic process groups

4. **Program examples**
   4.1 Matrix multiplication on a torus
   4.2 Matrix multiplication on a meshinterconnection network
   4.3 Matrix inversion
   4.4 Computing eigen values of a given matrix

5. **Timing values**

6. **Conclusions**

Bibliography

Chapter 1

INTRODUCTION

1.1 Need for Parallel Computation

The need for ever faster computers has not ceased since the beginning of the computer era. Every new application seems to push existing computers to their limit. If the current and contemplated applications are any indication, our requirements in terms of computing speed will continue.

With the advent of inexpensive elements due to VLSI technology, it has become feasible to build computing machines with hundreds or even thousands of processors cooperating in solving a given problem. Computing machines with various types and degrees of parallelism built into their architectures are already available in the market, and many more are in various stages of development.

However, in practice the tremendous increase in the *raw computational power* available through these machines does not directly translate into a comparable increase in performance. In order to attain the maximum performance, one must keep as many processors active as possible. But this is critically dependent on the algorithm used. It could be that the problem itself is such that it does not admit an ***"efficient"*** parallel algorithm. Or the type of parallelism that the chosen algorithm exhibits may not readily fit into the architecture of the underlying machine. That is to say, not every parallel algorithm is ***"efficiently"*** mapped onto a realistic parallel machine, or the processors are all active all the time but perform a large amount of ***redundant computations***. Further, when two or more processors cooperate in solving a problem, there will definitely be a need for communication and coordination between them. Understanding the nature and extent to which each of these factors affects the performance of parallel algorithms and architectures constitutes one of the fundamental goals of the theory of parallel computation.

1.2 Design of a Parallel Algorithm

Parallelism is sure to change the way we think about and use computers. It promises to put within our reach solutions to problems and frontiers of knowledge never dreamed of before. The rich variety of architectures will lead to the discovery of novel and more efficient solutions to both old and new problems. It is important therefore to ask: How do we solve problems on a parallel computer? There are three ways to design a parallel algorithm to solve a problem.

1. One can detect and exploit any inherent parallelism in an existing sequential algorithm.
2. One can invent a new parallel algorithm or
3. One can adapt another parallel algorithm to solve the similar problem.

The main problem encountered in the development of parallel algorithms is the minimization of communication cost, ability to run on simple architectures, synchronization among processors, timing of data movement etc. There are many other performance criteria which must be kept in mind, e.g., speed-up achieved utilization of resources.

A recent trend in computer systems is to distribute computation among several physical processors. There are basically two schemes for building such systems. In a multiprocessor (tightly coupled) system, the processors share memory and a clock, and communication usually takes place through shared memory. In a distributed (loosely coupled) system, the processors do not share memory or a clock. Instead, each processor has its own local memory . The processors communicate with one another through various communication networks, such as high speed buses or telephone lines.

1.3 Parallel Processing

We formally define parallel processing as follows:
Definition: Parallel processing is an efficient form of information processing which emphasizes the exploitation of concurrent events in the computing process. Concurrency implies parallelism, simultaneity and pipelining. Parallel events may occur in multiple resources during the same time interval; simultaneous events may occur at the same time instant; and pipelined events may occur in overlapping time spans. These concurrent events are attainable in a computing system at various processing levels. It is a cost effective means to improve system performance through concurrent actives in computers.

1. 4 Distributed Memory Parallel Processing

A distributed system is a collection of loosely coupled processors interconnected by a communication network. From the point of view of a specific processor in a distributed system, the rest of the processor and their respective resources are remote, whereas its own resources are local.

The processors in a distributed system may vary in size and function. They may include small micro processors, workstations, minicomputers, and large general-purpose computer systems. These processors are referred to by a number of different names, such as sites, nodes, computers, machines, hosts, and so on, depending on context in which they are mentioned. We mainly use the term site, to indicate a location of machines and host to refer to specific system at a site.

If a particular computation can be partitioned into a number of subcomputations that can run concurrently, then the availability of a distributed system may allow us to distribute the computation among the various sites, to run the computation concurrently. In addition, if a particular site is currently overloaded with jobs, some of them may be moved to other, lightly loaded, sites. This movement of job is called load sharing.

The loosely coupled multiprocessor system is a collection of micro processors, workstations, minicomputers, and large general-purpose computer systems. Here each processor that make up he system has it's local memory and the local memory of any two processor is not shareable. The processors are connected through high or low speed data links. Data path may be either both serial or parallel bus connecting I/O of two computers or shared bus to which two or more computers are interconnected in various ways.

The general idea of parallelizing the programs for these machines is to decompose the domain of the application and to assign the parts to individual processors. These processors perform the computation on their subdomain and have to synchronize, if they need values computed by other processes.

The algorithm to achieve the objective is a distributed algorithm. The distributed algorithm is broken down into a set of algorithm, one of which is performed by each peer process. The algorithm performed by one process in a set of peers consists of carrying out various operation on available data, and at various points in the algorithm, either sending data to one or more peer processes or readind (or waiting for) data sent by another peer process.

In the simplest distributed algorithm, the order in which operations are carried out by various algorithm is completely determined. For example, one algorithm might perform several operations and then reliably send some data to other algorithm, which then carries out some operations and return some data. In more complex cases, several algorithms operate concurrent, but each still waits at predetermined points in the algotithm for predetermined messages from specific other algorithms. In this case, the overall distributed algorithm still operates in a deterministic fashion (given the input data to the peer processors), but the lockstep ordering of operations between different algorithms is removed.

In the most complex case , the order in which an algorithm performs its operations depends on the order in which data arrive.

1.5 Approach to Parallelizing Codes

Design of parallel programs depends a lot on the programming model that issued. There are three different ways to produce parallel codes.

Semi-automatic parallelizing
Data parallel Programming.
Message Passing Programming

1.5.1 Semi-automatic Parallel Programming

The main goal of automatic parallelizing is to generate, without any external intervention, a code appropriate for parallel computer from a sequential source program.

1.5.2 Data Parallel Programming

A program which performs identical computation over a large set of data points, is called Data Parallel Programming. The approach to parallelizing this type of program consists in distributing the data over the number of available processors. Then each processor independently performs the computation of it's subset of data. This is basically done through a Data-parallel Programming language. The programmer uses the data parallel programming support given in the language(i.e. compiler interpreted key words) to specify the data distribution code. On compilation the compiler generates the code for parallel programming.

1.5.3 Message Passing

In this programming model, a parallel program is considered as a set of cooperating processes, each solving a common task. Each processor owns a private set of data. If a processor needs data from other processors, an explicit request message has to be sent to the processor holding the desired data. The parallel program can be seen as a set of in instruction executing in parallel on different processors, which exchange message during execution, to send or receive certain values.

Developing a message passing program requires that the programmer take care to manually distribute both computations and data among the processors. Communication routine calls between processors are inserted explicitly into the source code. These programming features are straight forward to implement if the code has a intrinsic parallel structure with few and regular communications. If however . the code is difficult to parallelize, and the parallelizing process produces a large number of irregular communication between the processors, then the use of message passing library can be tidious.

Chapter 2

Heterogeneous processing

The main goal of the development of computing systems is to increase its performance. Basically , three different ways are used to achieve this. These are :

1. **Physical** enlarging of the clock frequency, capacity of memories, and the throughput of channels;
2. **Logical** exploiting of various methods of concurrent processing, and
3. **Functional** specialisation of software and hardware in order to achieve most effective realisation of different labour intensive functions.

Recently the aspiration for utilising the useful features of different architectures led to the emergence of the concept of Heterogeneous Computing(HC), which implies a distributed network system of several commercially available computers of diverse architectures. In such an environment , the user is able to vary flexibly the style of programming , in accordance with the characteristics of his problems. Heterogeneous processing (HP) addresses compute intensive applications and algorithms that are composed of fundamentally different sub-tasks. The goals of HP are to investigate and develop techniques and infrastructures for the execution of such applications, e.g., through building a virtual heterogeneous machine composed of diverse processing elements and through evolving appropriate programming and system methodologies. HP uses different types of parallel processors, processing components and/or connectivity paradigms to maximise performance, cost effectiveness and/or ease of development.

2.1 Software Platform for Heterogeneous Resources

In some applications collections of heterogeneous machines can be used to execute a parallel algorithm, thereby allowing the application to produce results at a higher speed than on a single machine. Facilitating the use of heterogenity for such purposes is the target of systems such as PVM, p4, APPL, MPI etc, which provide sophisticated software infrastructures that, in essence, turn a network of possibly heterogeneous workstations into a cost-effective parallel machine.

Here in this report we focus on PVM programming system. PVM (Parallel Virtual Machine) is a portable message-passing programming system, designed to link separate host machines to form a "virtual machine" which is a single, manageable computing resource.

The virtual machine can be composed of hosts of varying types, in physically remote locations. PVM applications can be composed of any number of separate processes, or components , written in a mixture of C, C++ and Fortran. The system is portable to a wide variety of architectures, including workstations, multiprocessors, supercomputers and PCs.

PVM is discussed at great length in the next chapter.

2.2 Program speedup in a Heterogeneous Computing Network

The purpose of heterogeneous computing network is to achieve speedup. Donaldson, Berman and Paturi [ ] proposed definitions and models for program speedup in a heterogeneous computing system, incorporating parameters relevant to the program and computations as well as the network.

Their model is based on the task graph with computational nodes and communication edges and assumes that tasks are atomic.

2.2.1 Heterogeneous network model

Their heterogeneous network model is a set $H = \{ M_1, M_2, \ldots, M_m \}$ of m distinct machines. Each machine can communicate with any other machine. The machines in H may include any combination of sequential, dataflow, vector or other processors. Their primary model of program execution on H assumes that each machine M_j may execute at most one program task at a time. In this case, we say the network is nondecomposable. If multiple tasks can be executed concurrently on an individual machine, we call the network decomposable.

Program Model :

The basis of their program model is the well known task graph. A task graph for a program P is a directed acyclic graph $G_p = (V,E)$. $V = \{t_1, t_2, \ldots, t_n\}$ is a partition of P consisting of n tasks. Tasks are constrained to be atomic- each task is executed to completion by exactly one machine without intermediate communication or synchronisation with other tasks. $E \subseteq VXV$. and $(t_i, t_k) \in E$ iff t_i must execute before t_k due to data dependence or task synchronisation requirements of P. Since G_p is acyclic , E may be viewed as imposing a partial ordering on the tasks in V. Without the loss of generality, we assume that G_p has unique entry and exit nodes identified with tasks t_1 and t_n. and for any task t_i, there is a path from t_1 to t_n .Figure. (a) . at the end of this chapter is an example of a task graph.

For a program consisting of n tasks and a network of m machines. there are m^n possible mappings of tasks to machines. A mapping of tasks to machines is a function $\Pi: V \rightarrow H$. If Π maps all n tasks to the same machine the mapping is sequential. If $n > m$. any mapping of tasks to machines must map multiple tasks to at least one machine.

The requirement that each machine may execute at most one task at a time can be enforced by augmenting the task graph. A scheduled task graph is a task graph with additional edges defining a total order on all tasks mapped to the same machine. If similar edges already exist in E, the additional scheduling edges are redundant. and are not added (the scheduled task graph is not a multigraph). Edges which are transtively redundant may also be deleted if desired. although when data communication times are considered. otherwise redundant edges may need to be retained. Figures. (b) and (c) at the end of this chapter are examples of scheduled task graphs.

Unless otherwise specified, we assume that G_p is a scheduled task graph, and that the mapping of Π on G_p is fixed.

Timing values are associated with the components of G_p. For each of the n tasks and each of the machines, T_{ti}^{Mj} is the time required to execute the task ti on machine M_j. T_{ti}^{H} is defined to be the value T_{ti}^{Mj} such that $\Pi(ti) = M_j$. We also define timing values for communication between tasks. For each edge (ti tk) $\in$E, $N_{ti,tk}$ is the time required for data communication (including data representation conversion if required) and synchronisation between task ti on machine Π(ti) and task t_k on machine $\Pi(tk)$. It is often assumed that if tasks t_i and t_k are mapped to the same machine, $N_{ti,tk} = 0$. This simplification can be justified by noting that relative to network times, communication times internal to individual machines in the network are typically negligible. In this discussion we assume zero communication time for intramachine communication.

The preceding model contains an additional simplification. In practice, both network times and task times will vary, depending on the state of the machines and the network. We make the common simplifying assumption that the computation times of individual tasks and the communication times between tasks are constant.

With these assumptions, we can define T_p^H, the time to execute program *P* on a heterogeneous network *H*, using a macro dataflow apporach. Let G_p be a scheduled task graph. Define est(t1), the earliest starting time of entry task t_1 to 0, and for all remaining tasks t_i, i=2,3,….,n define

$$est(t_i) = \max_{\substack{(t_k,t_i)\in E \\ t_k \neq t_i}} \{ est(t_k) + T_{tk}^H + N_{tk,ti} \}$$

Then

$$T_p^H = est(t_n) + T_{tn}^H.$$

We are also interested in T_p^{Mj}, the time required for each individual machine $M_j \in H$ to execute P. Since network *H* is non-decomposable and intramachine communication times are assumed to be negligible, for $j = 1,2,\ldots,m$, we define

$$T_p^{Mj} = \sum_{i=1,\ldots,n} T_{ti}^{Mj}$$

We assume that the program *P*, the network of machines *H*, the scheduled task graph G_p with timing values, and the mapping Π are fixed.

Definition : The speedup S_p of heterogeneous network *H* on problem *P* under mapping Π is defined as …..

$$S_p = \min\{T_p^{M1}, T_p^{M2}, \ldots, T_p^{Mm}\} / T_p^H .$$

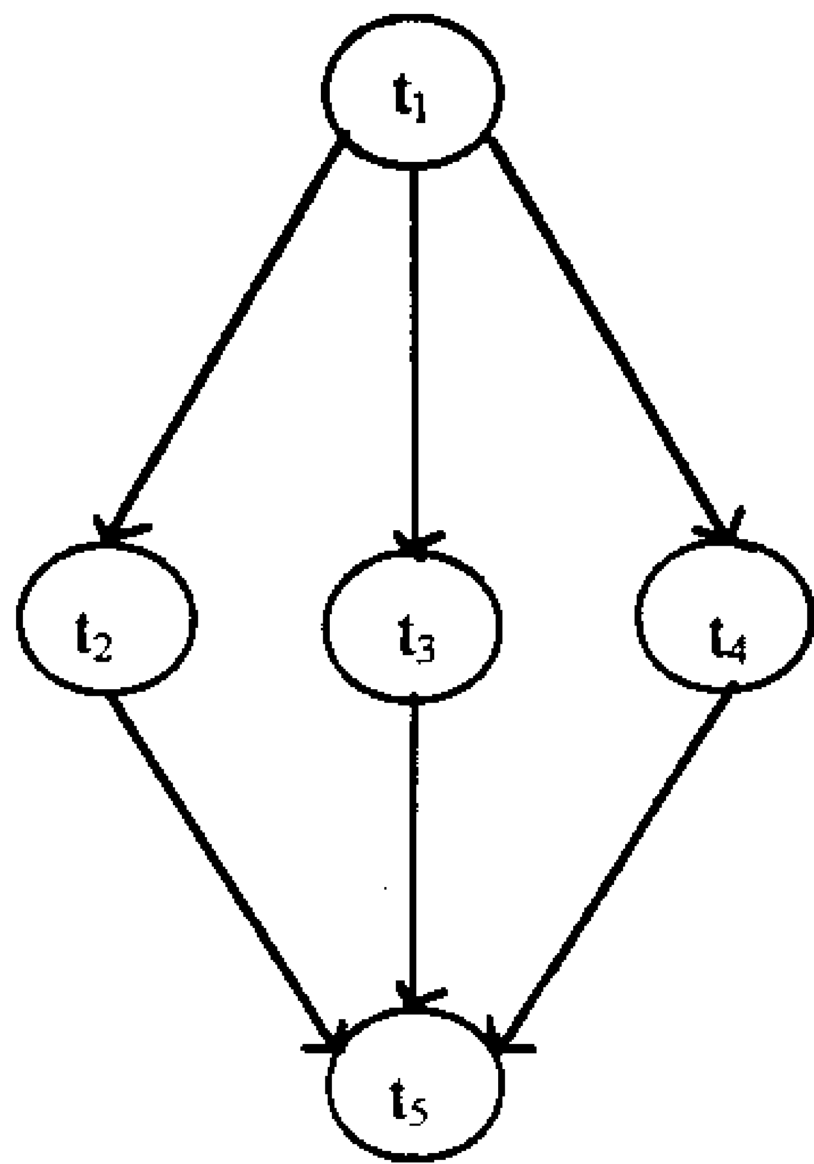


Figure . (a)

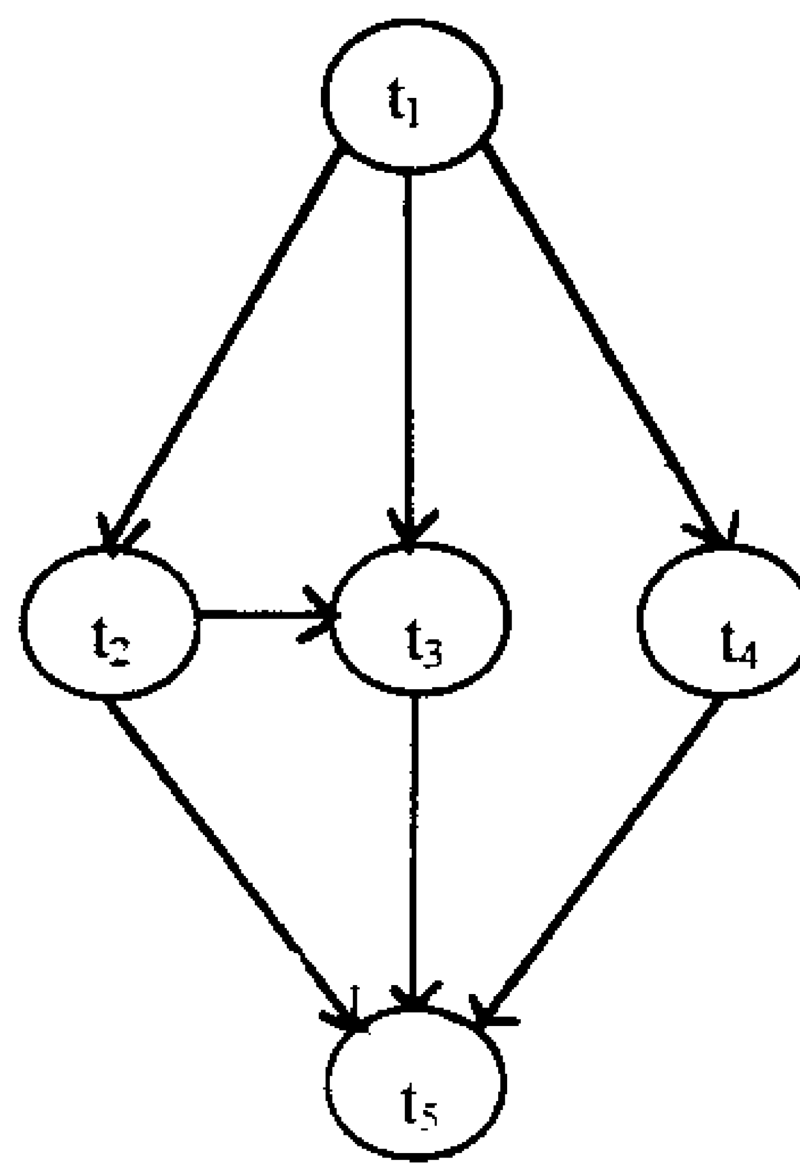


Figure . (b)

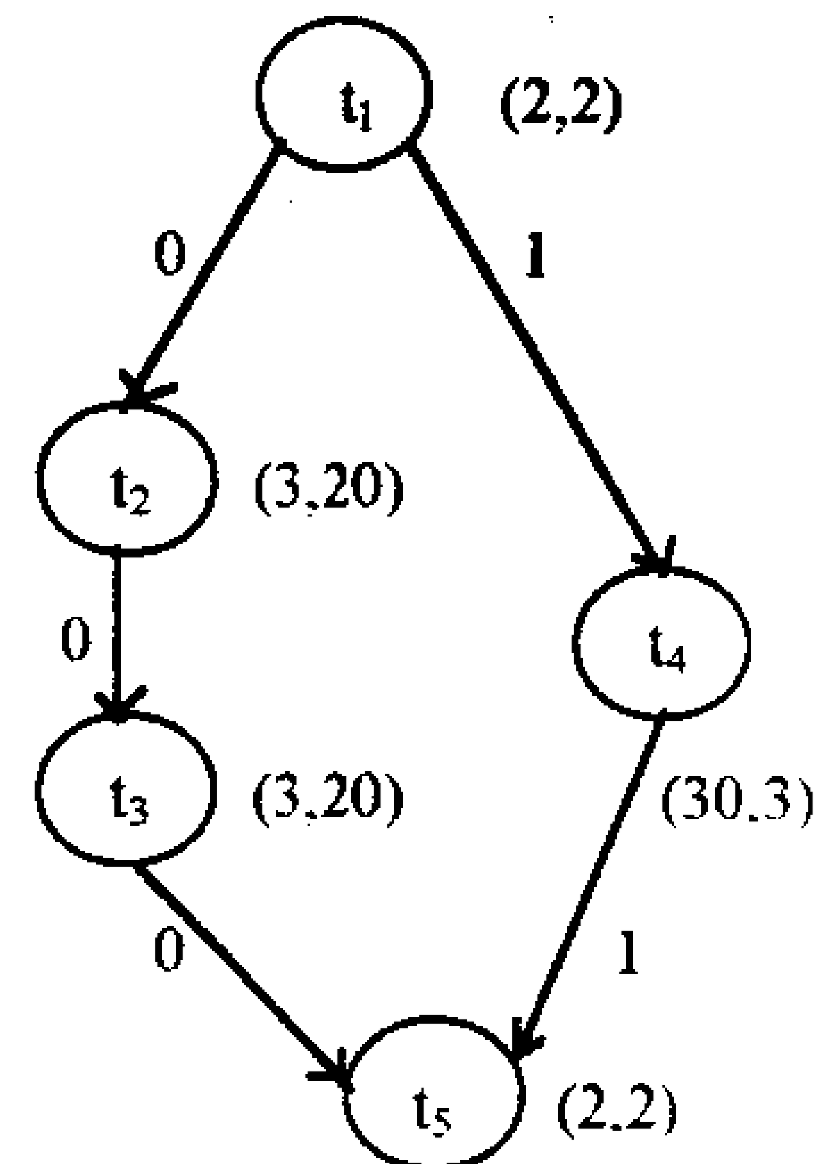


Figure . (c)

Figure (a) : Task Graph

Figure (b) : Scheduled task graph on two machines ;
Shaded tasks are mapped to one machine.

Figure (c) : Scheduled task graph without redundant edges. Timing values in paranthesis for task t_i are

$$(T_{ti}^{M1}, T_{ti}^{M2})$$

$$T_p^{M1} = 2+3+3+30+2 \quad = 40$$
$$T_p^{M2} = 2+20+20+3+2 \quad = 47$$

$$\min (T_p^{M1}, T_p^{M1}) = 40 \quad \text{(Sequential time)}$$

$$T_p^{H} = 2+3+3+2 \quad = 10 \quad \text{(Parallel time)}$$

Therefore, Speed up , $S_p = 40/10 = 4$.

Chapter 3

Software Environment

In this chapter we describe the software environment used for executing parallel applications. The software system used is - **PVM** that permits a heterogeneous collection of Unix networked together to be viewed by a user's program as a single parallel computer. PVM is the mainstay of the heterogeneous network computing research project, a collaborative venture between Oak Ridge National Laboratory, the University of Tennessee, Emory University and Carnegie Mellon University.

PVM supports a straight forward but functionally complete message passing model. PVM is designed to link computing resources and provide users with a parallel platform , for running their applications , irrespective of the number of different computers they use and where the computers are located.

In this chapter we discuss the architecture of the PVM system and discuss its computing model: the programming interface it supports; auxiliary facilities for process groups; and some of the internal implementation techniques employed.

3.1 Common Parallel Programming Paradigms

Parallel computing may be approached from three fundamental viewpoint, based on the organization of computing tasks. Plainly we can say these three classifications are made on the basis of process relationships. Within each these there classes, different workload allocation strategies are possible .

Crowd Computation
Tree Computation
Hybrid Computation

3.1.1 Crowd Computation

The first and most common model for parallel application can be termed as crowd computing: a collection of closely related processes, typically executing the same code, perform computations on different portions of the workload, usually involving the periodic exchange of intermediate result. This paradigm can be further subdivided into two categories:

The master-slave(or host-node) model in which a separate "control" program termed the master is responsible for process spawning, initialization, collection and display of results, and timing functions. The slave program perform the actual computation involved; they either allocated workload by their master (statically or dynamically) or perform the allocation themselves.

The node only model where multiple instances of a single program execute, with one process(typically the one initiated manually) taking over the noncomputational responsibilities in addition to contributing to the computation itself.

Crowd computation typically involve three phases. The first is the initialization of process group; in the case of node-only computation , in the case of node-only computations, dissemination of group information and problem parameters, as well as workload allocation, is typically done within this phase.

The second phase is computation. The third phase is collection results and display of output; during this phase, the process group is disbanded or terminated.

3.1.2 Tree Computations

The second model is tree computation. In this model processes are spawned (usually dynamically as the computation progresses) in a tree like manner, there by establishing a tree-like , parent-child relationship (as opposed to crowd computations where a star like relationship exists). This paradigm, although less commonly used, is an extremely natural fit to applications where the total work load is not known prior to the computation.

3.1.3 Hybrid Computations

The third model which we term "hybrid," can be thought of as a combination of the tree and crowd model. Essentially , this paradigm possesses an arbitrary spawning structure: that is, at any point during application execution, the process relationship structure may resemble an arbitrary and changing graph.

3.2 Workload Allocation

In the preceding section, we discussed the common parallel programming paradigms with respect to process structure . In this section we address the issue of workload allocation. subsequent to establishing process structure, and describe some common paradigms used in distributed memory parallel computing. Two general methodologies are commonly used:

Data decomposition
Function decomposition

The first, termed data decomposition or partitioning, assumes that overall problem involves applying computational operations or transformations on one or more data structures and , further, that these data structures may be divided and operated upon. The second, called function decomposition ,divides the work based on different operations or functions.

3.2.1 Data Decomposition

As a simple example of data decomposition , consider the addition of two vectors, A[1..N] and B[1..N], to produce the result vector, C[1..N]. If we assume that P processes are working on this problem, data partitioning involves the allocation of N/P elements of the resulting vector. This data partitioning may be done either "statically," where each process knows prior (at least in terms of the variables N and P) its share of the workload, or " dynamically," where a control process (e.g., the master process) allocates subunits of the workload to processes as and when they become free. The principal difference between these two approaches is "scheduling". With static scheduling individual processes work load are fixed; with dynamic scheduling, they vary as the computation progresses.

In a real execution of even this trivial vector addition problem, an issue that cannot be ignored is input and output. In other words, how do the processes described above receive their workloads, and what do they do with the result vectors? The answer to these questions depends on the application and the circumstances of particular run, but in general:

1. Individual processes generate their own data internally, for example, using random numbers or statically known values. This is possible only in very special situation or for program testing purposes.

2. Individual processes independently input their data subsets from external devices. This method is meaningful in many cases, but possible only when parallel I/O facilities are supported.

3. A controlling process sends individual data subsets to each process. This is the most common scenario. especially when parallel I/O facilities do not exist. Further this method is also appropriate when input data subsets are derived from previous computation within the same application.

The third method of allocating individual workloads is also consistent with dynamic scheduling in applications where interprocess interactions during computations are rare or nonexistent. However nontrivial algorithms generally require intermediate exchanges of data values, and therefore only the initial assignment of data partitions can be accomplished by these schemes. In order to multiply two matrices A and B , a group of processes is first spawned , using the master-slave or node-only paradigm. This set of processes is considered to form a mesh; the matrices to be multiplied are divided into subblocks, also forming a mesh. Each subblock of matrices A and B is placed on the corresponding process, by utilizing one of the data decomposition and work load allocation strategies listed above. During computation , subblocks need to be forwarded or exchanged between processes. At the end of the computation, however, result matrix subblocks are situated on the individual processes, in conformance with their respective positions in the process grid, and consistent with a data partitioned map of resulting matrix C.

3.2.2 Function Decomposition

Parallelism in distributed memory environments may also be achieved by partitioning the overall workload in terms of different operations. The most obvious example of this form of decomposition is with respect to the three stages of typical program execution. namely . input processing. and result output. In function decomposition, such an application may consist of three separate and distinct programs . each one dedicated to one of three phases. Parallelism is obtained by concurrently executing the three programs and by establishing a "pipeline" (continuous or quantized) between them. Note. however. that in such scenario. data parallelism may also exist between within each phase.

Although the concept of function decomposition is illustrated by the trivial example above. the term is generally used to signify partitioning and workload allocation by function within the computational phase. Typically, application computations contain several different subalgorithms- sometimes on the same data (the MPSD or Multiple Program Single Data scenario). sometimes in a pipelined sequence of transformations. and sometimes exhibiting an unstructured pattern of exchanges.

3.3 PVM User Interface

In this section we give a brief description of the routines in the PVM3 user library. All the routines are clearly explained in the book ***PVM: A user's guide and Tutorial for Networked Parallel Computing.***

3.3.1 Process Control

1. **int tid = pvm_mytid(void)**

The routine **pvm_mytid()** returns the TID of the process and can be called multiple times. It enrolls this process into PVM if this is the first PVM call.

2. **int info = pvm_exit(void)**

This routine **pvm_exit()** tells the local pvmd that this process is leaving PVM.

3. **int numt = pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids)**

The routine **pvm_spawn** starts up **ntask** copies of an executable file task on the virtual machine. **argv** is a pointer to an array of arguments to task with the end of the array specified by **NULL.** If the task argument takes no arguments, then **argv** is NULL. To find out about the flag argument refers to, see the PVM manual.

4. **int info = pvm_kill(int tid)**

The routine **pvm_kill()** kills some other PVM task identified by TID.

5. **int info = pvm_catchout(FILE *ff)**

The routine **pvm_catchout** causes the calling task to catch output from tasks subsequently spawned.

3.3.2 Information

6. **int tid = pvm_parent(void)**

The routine **pvm_parent( )** returns the TID of the process that spawned this task or the value of **PvmNoParent if not created by pvm_spawn( ).**

7. **int dtid = pvm_tidtohost(int tid)**

The routine **pvm_tidtohost(int tid)** returns the dtid of the daemon running on the same host as TID.

8. **int info = pvm_config(int *nhost, int *narch, struct pvm_hostinfo **hostp)**

The routine **pvm_config()** returns information about the virtual machine including the number of hosts, **nhost,** and the number of different data formats, **narch. hostp** is a pointer to a user declared array of **pvmhostinfo** structures.

9. int info = pvm_tasks(int which, int *ntask, struct pvmtaskinfo **taskp)

The routine **pvm_tasks()** returns information about the PVM tasks running on the virtual machine. The integer **which** specifies which tasks are returning information about.

3.3.3 Dynamic Configuration

10. int info = pvm_addhosts(char **hosts, int nhost, int *infos)
11. int info = pvm_delhosts(char **hosts, int nhost, int *infos)

These routines add or delete a set of hosts in the virtual machine.

3.3.4 Setting and Getting Options

12. int oldval = pvm_setopt(int what, int val)
13. int val = pvm_getopt(int what)

The routine **pvm-setopt( )** is a general purpose function that allows the user to set or get options in the PVM system. In PVM3, **pvm_setopt( )** can be used to set several options. including automatic error message printing, debugging level. and communication routing method for all subsequent PVM calls.

3.3.5 Message Passing

Sending a message comprises three steps in PVM. First, a send buffer must be initialized by a call to **pvm_initsend()** or **pvm_mkbuf()**. Second, the message must be "packed" into this buffer using any number and combination of **pvm_pk*()** routines. Third, the completed message is sent to another process by calling the **pvm_send()** routine or multicast with **pvm_mcast()** routine.

3.3.5(i) Message Buffers:

14. int bufid = pvm_initsend(int encoding)

The routine **pvm_initsend()** clears the send buffer and creates a new one for packing a new message.

15. int bufid = pvm_mkbuf(int encoding)

The routine **pvm_mkbuf()** creates a new empty send buffer and specifies the *encoding* method used for packing messages.

16. int info = pvm_freebuf(int bufid)

The routine **pvm_freebuf( )** disposes of the buffer with identifier **bufid**. This should be done after a message has been sent and is no longer needed.

3.3.5(ii) Packing Data :

17. **int info = pvm_pkbyte(char *cp, int nitem, int stride)**
18. **int info = pvm_pkcplx(float *xp, int nitem, int stride)**
19. **int info = pvm_pkdcplx(double *zp, int nitem, int stride)**
20. **int info = pvm_pkdouble(double *dp, int nitem, int stride)**
21. **int info = pvm_pkfloat(float *fp, int nitem, int stride)**
22. **int info = pvm_pkint(int *np, int nitem, int stride)**
23. **int info = pvm_pklong(long *np, int nitem, int stride)**
24. **int info = pvm_pkshort(short *np, int nitem, int stride)**
25. **int info = pvm_pkstr(char *cp)**

Each of the above C routines packs an array of the given data type into the active send buffer. The arguments for each of the routines are a pointer to the first item to be packed, **nitem** which is the total number of items to pack from this array, and **stride** which is the stride to use when packing. A stride of 1 means a contiguous vector is packed, a stride of 2 means every other item is packed and so on. An exception is **pvm_pkstr( )** which by definition packs a NULL terminated character string and thus does not need **nitem** or **stride** arguments.

3.3.5(iii) Sending and Receiving data :

26. **int info = pvm-send(int tid, int msgtag)**
27. **int info = pvm_mcast(int *tids, int ntask, int msgtag)**

The routine **pvm_send()** labels the message with an integer identifier **msgtag** and sends it immediately to the process TID. The routine **pvm_mcast( )** labels the message with an identifier **msgtag** and broadcasts the message to all tasks specified in the integer array **tids** (except itself) . The tids array is of length **ntask**.

28. **int info = pvm_recv(int tid, int msgtag)**

The blocking routine will wait until a message with label **msgtag** has arrived from TID. A value of -1 in **msgtag** or **TID** matches anything (wildcard).

3.3.5(iv) Unpacking Data :

29. **int info = pvm_upkbyte(char *cp, int nitem, int stride)**
30. **int info = pvm_upkcplx(float *xp, int nitem, int stride)**
31. **int info = pvm_upkdcplx(double *zp, int nitem, int stride)**
32. **int info = pvm_upkdouble(double *dp, int nitem, int stride)**
33. **int info = pvm_upkfloat(float *fp, int nitem, int stride)**
34. **int info = pvm_upkint(int *np, int nitem, int stride)**
35. **int info = pvm_upklong(long *np, int nitem, int stride)**
36. **int info = pvm_upkshort(short *np, int nitem, int stride)**
37. **int info = pvm_upkstr(char *cp)**

The above routines unpack data types from the active receive buffer. In an application they should match their corresponding pack routines in type, number of items, and stride. **nitem** is the number of items of the given type to unpack.

3.4 Dynamic Process Groups

The dynamic process group functions are built on top of the core PVM routines. A separate library libgpvm3.a must be linked with user programs that make use of any of the group functions.

38. int inum = pvm_joingroup(char *group)
39. int info = pvm_lvgroup(char *group)

These routines allow a task to join or leave a user named group. The first call to **pvm_joingroup()** creates a group with the name **group** and puts the calling task in this group. This function returns the instance number (**inum**) of the process in this group. Instance numbers range from 0 to the number of group members minus 1.

40. int tid = pvm_gettid(char *group, int inum)
41. int inum = pvm_getinst(char *group, int tid)
42. int size = pvm_gsize(char *group)

the routine **pvm_gettid()** returns the TID of the process with a given group name and instance number. The routine **pvm-getinst()** returns the instance number of TID in the specified group. The routine **pvm-gsize()** returns the number of members in the specified group.

43. int info = pvm-barrier(char *group, int count)

On calling **pvm_barrier()** the process blocks until **count** number of members of a group have called **pvm_barrier.**

44. int info = pvm_bcast(char *group, int msgtag)

pvm_bcast() labels the message with an integer identifier **msgtag** and broadcasts the message to all tasks in the specified group except itself (if it is a member of the group).

45. int info = pvm_reduce(void (*func)(), void *data, int nitem, int datatype, int msgtag, char *group, int root)

pvm_reduce() performs a global arithmetic operations across the group, for example , global sum or global max.

These are some of the routines in the PVM3 user library. An alphabetical listing of all the routines is given in Appendix B of the book *PVM: Parallel Virtual Machine, A user's Guide and Tutorial for Networked Parallel computing.*

Chapter 4

Program Examples

In this chapter we discuss several parallel algorithms which are implemented for solving many matrix problems, such as finding the product of two matrices, computing the inverse, transpose and eigen values of a matrix. These algorithms demonstrate how to implement parallel algorithms and how to schedule the work load evenly among the processes for such matrix operations. Since here we have a network of three unix workstations, all the processes were spawned and run on three workstations. We have tried to simulate different interconnection schemes using **PVM** library calls. At the end we have discussed some interesting results that we came across in the process of exploring **PVM.** We make a comparative study of the different times for completion of a computation for different number of processors for a given topology. The interconnections topologies between tasks that we have considered are torus and mesh. We have chosen different operations on matrices and observed the timing values by varying number of processes which are distributed ***equally*** among the processors present in the network. The problems are also studied taking into consideration the "***load***" on each processor. The load on a workstation is taken as the number of ***ready processes*** at the time of execution of a program. When the tasks are created this load is taken into consideration and processes are distributed accordingly among different processors.

4.1 Matrix multiplication on a torus

This program uses node only computation for matrix multiplication. The algorithm is as follows..

```
# Matrix multiply program using Pipe-Multiply-Role Algorithm #

# processor 0 starts up other processes #

if(<process-number> = 0) then
        for I = 0 to Meshdimension*Meshdimension
                spawn the process;
        end for
end if

forall processes Pij, 0 <= I,j < Meshdimension
        for k := 0 to Meshdimension - 1
                # Pipe #
                if row = (column+k) mod Meshdimension
                        Send A to all Pxy, x = row, y != column;


                else
                        receive A;
                end if

                # Multiply : Running totals are maintained in C #
                Multiply(A,B,C);

                # Roll matrix B subblocks #
                Send B to Pxy, x = row -1, y = column; #i.e.sending the blocks upwards#
                Receive B; # i.e. receive from down #
```

```
        end for
end for
```

This algorithm shown in fig 1 multiplies matrix subblocks locally, and uses row-wise multicast of matrix A in conjuction with column-wise shifts of matrix B subblocks.

4.2 Matrix multiplication on a mesh interconnection network

This program computes the product of two $n \times n$ square matrices. This algorithm uses mxm processors arranged in the form mesh. Here n is equal to $m \times block\ size$, where m is the number of processors in each row and *block size* is the dimension of each subblock into which the original matrices are broken up into. E.g. 300 x 300 matrix can be broken up into 9 subblocks each of size 100 x 100.

Here we have used the *master-slave* model in which a separate "control" program termed the master is responsible for process spawning, initialisation, collection and display of results. The slave program performs the actual computations involved.

Assume that we have a grid of $m \times m$ tasks. These tasks have all been created by a master process. Mash rows are numbered 0....(m-1) and columns are numbered 0....(m-1). Matrices A and B are fed to the *boundary processors* in column 1 and row 1 respectively, as shown in the figure at the end of this chapter.

At the end of the algorithm, block Cij of the product matrix C resides in the processor Pi,j. Initially Cij is zero. Subsequently, when Pi,j receives two input blocks a and b, it

1. multiplies them,
2. adds the result to Cij,
3. sends a to $Pi,j+1$ unless $j = m$, and
4. sends b to $Pi+1,j$ unless $I = m$.

Here the two matrices a and b are actually two submatrices. Note that processor $P1,2$ lags one time unit behind processor $P1,1$ (as shown in the figure 2), because it waits for $P1,1$ to be over with its submatrix multiplication of $B11$ and $A11$ before receiving $A11$. The other processes also lag behind accordingly, depending on their row and column positions. Elements Amm and Bmm take 3m-2 steps from the beginning of the computation to reach Pm,m. Since Pm,m is the last processor to terminate, this many steps are required to compute the product.

4.3 Matrix Inversion

The best known sequential algorithm for *matrix inversion* is the $O(n^x)|_{2<x<2.5}$ algorithm. According to this algorithm, the inverse can be computed as follows. We begin by writing

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} I & 0 \\ A_{21}A_{11}^{-1} & I \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & B \end{bmatrix} \begin{bmatrix} I & A_{11}^{-1}A_{12} \\ 0 & I \end{bmatrix}$$

where Aij are $(n/2) \times (n/2)$ submatrices of A, and $B = A_{22} - A_{21}A_{11}^{-1}A_{12}$. The $(n/2) \times (n/2)$ matrices I and 0 are the identity matrix and zero matrix respectively. The inverse of A is then given by the matrix product

$$A^{-1} = \begin{bmatrix} I & A_{11}^{-1}A_{12} \\ 0 & I \end{bmatrix} \begin{bmatrix} A_{11}^{-1} & 0 \\ 0 & B^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -A_{21}A_{11}^{-1} & I \end{bmatrix}$$

where A_{11} and B are computed by applying the same process recursively. This requires two inversions two inversions, six multiplications and two additions. In sequential computation the time required to compute the inverse of an $n \times n$ matrix using the above algorithm matches, up to a constant multiplicative factor, the time required to multiply two n x n matrices.

Since the algorithm for finding the inverse reduces to multiplication of matrices, the parallel implementation of this algorithm is nothing but writing a parallel program for matrix multiplication. Any parallel algorithm for matrix multiplication can be used. Here for this purpose we have used matrix multiplication on mesh.

4.4 Computing Eigen Values Of A Given Matrix

The algebraic eigen values problem derives its importance from its relation to the problem of solving a system of n simultaneous linear differential equations of first order with constant coefficients. Such a system is written as

$$dx/dt = Ax$$

where A is an n x n matrix and x is an n x 1 vector. For some vector $u \neq 0$, $x=ue^{\lambda t}$ is a solution of the preceding system if and only if $\lambda u = Au$. Here, λ is called an eigen value and u an eigen vector. The algebraic eigen value problem is to determine λ and u.

For an n x n matrix B and an n x 1 vector y, if we apply the transformation $x = By$ to the system of differential equations, we get

$$dy/dt = (B^{-1}AB)y.$$

The eigen values of $B^{-1}AB$ are same as those of A. We therefore chose B such that the eigen values of $B^{-1}AB$ are easily obtainable. For, example if $B^{-1}AB$ is a diagonal matrix (i.e. all the elements are zero except on the diagonal), then the diagonal elements are the eigen values. One method of transforming a symmetric matrix A to diagonal form is *Jacobi's algorithm*. The method is an iterative one, where the kth iteration is defined by

$$A_k = R_k A_{k-1} R_k^T \quad \text{for } k = 1,2,\ldots,$$

with

$$A_0 = A.$$

The n x n matrices R_k are known as plane rotations. Let a_{ij}^k denote the elements of A_k. The purpose of R_k is to reduce the two elements of a_{pq}^{k-1} and a_{qp}^{k-1} to zero (for some $p<q$ depending on k). In reality, each iteration decreases the sum of the squares of the nondiagonal elements so that A_k converges to a diagonal matrix. The process stops when the sum of the squares is sufficiently small, or more specifically, when

$$d_k = (\Sigma_{i=1}\Sigma_{j=1}(a_{ij}^k))_{i\neq j}^2 < c$$

for some small tolerance c. At that point, the columns of the matrix $R_1^T R_2^T \ldots R_k^T$ are the eigen vectors.

The plane rotations are chosen as follows. If a_{pq}^{k-1} is a nonzero off-diagonal element of A_{k-1}, we wish to define R_k so that $a_{pq}^k = a_{qp}^k = 0$. Denote the elements of R_k by r_{ij}. We take

$$r_{pp}^k = r_{qq}^k = \cos\theta_k$$

$$r_{pq}^k = -r_{qp}^k = \sin\theta_k$$

$$r_{ij}^k = 1 \text{ for } i \neq p \text{ or } q,$$

$$r_{ij}^k = 0 \text{ otherwise,}$$

where $\cos\theta_k$ and $\sin\theta_k$ are obtained as follows. Let

$$\alpha_k = (a_{qq}^{k-1} - a_{pp}^{k-1})/2a_{pq}^{k-1}$$

and

$$\beta_k = 1/[\text{sign}(\alpha_k)][\text{abs}(\alpha_k) + (1 + \alpha_k^2)^{1/2}],$$

where sign(α_k) is +1 or -1 depending on whether α_k is positive or negative, respectively. Then

$$\cos\theta_k = 1/(1 + \beta_k^2)^{1/2} \text{ and } \sin\theta_k = \beta_k\cos\theta_k.$$

The only question remaining is : which nonzero element a_{pq}^{k-1} is selected for reduction to zero during k^{th} iteration? Many approaches are possible, one of which is to choose the element of greatest magnitude since this would lead to the greatest reduction in d_k.

This algorithm converges in $O(n^2)$ iterations. Since each iteration consists of two matrix multiplications, the entire process takes $O(n^5)$ time, assuming that the (sequential) procedure for *matrix multiplication* is used.

Jacobi's method itself lends to parallel implementation. It is seen clearly that for the parallel implementation, parallel procedure for *matrix multiplication* is used. Any parallel algorithm for matrix multiplication can be used. We have used *matrix multiplication* on *mesh* for this purpose. The algorithm is as follows.....

procedure **EIGENVALUES(*A*,*c*)**

while $d > c$ **do**

1. Find the off-diagonal element in A with largest absolute value.
2. Create R.
3. $A \leftarrow RA$.
4. Create R^T.
5. $A \leftarrow AR^T$.

end while.

Analysis : Since matrix multiplication on mesh takes $O(n)$ time , the time per iteration is $O(n)$. Since convergence is attained after $O(n^2)$ iterations, the overall running time is $O(n^3)$.

First 4 steps of Pipe-Multiply-Roll on a 3x3 Mesh-Connected Machine.

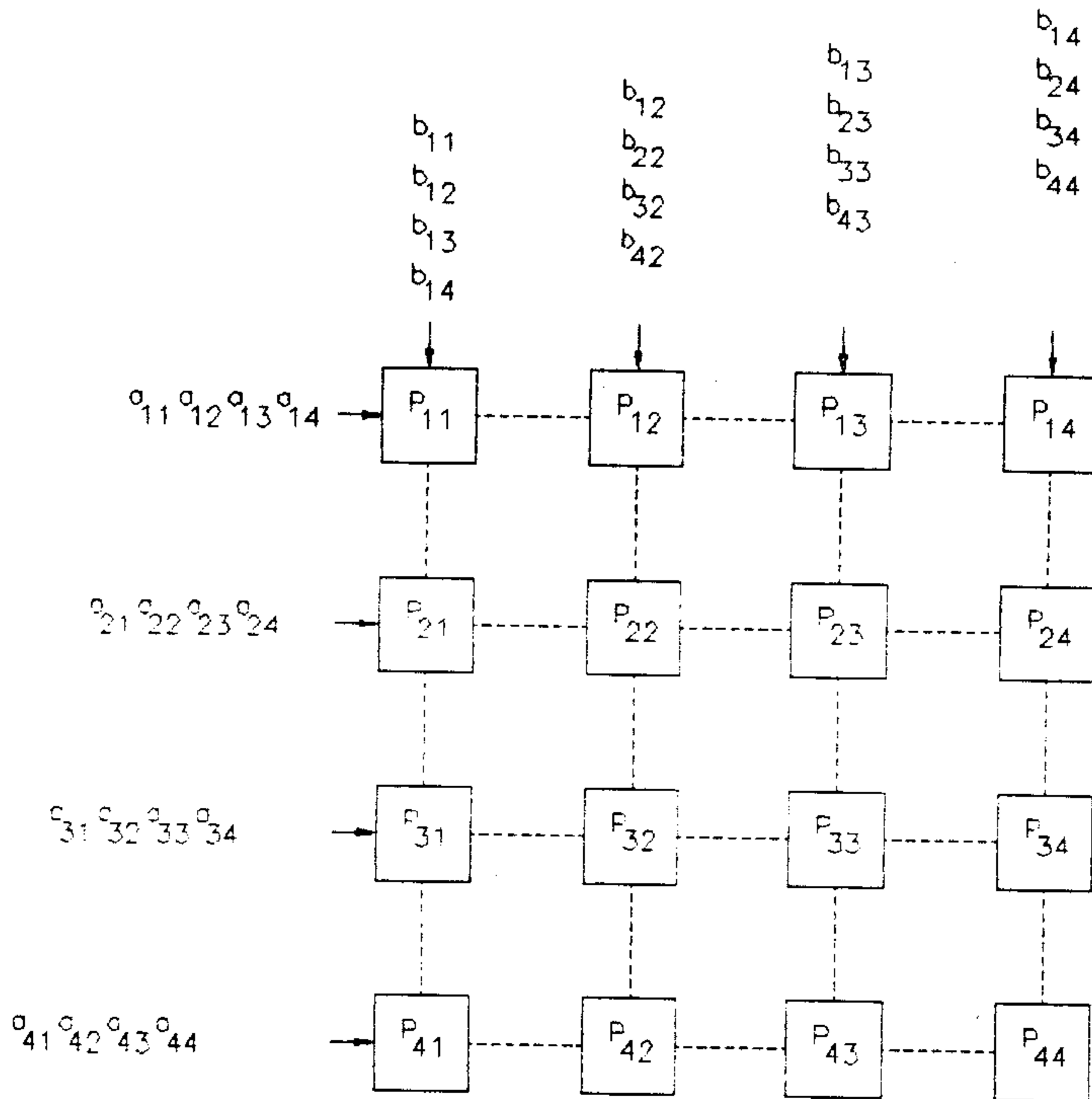
Step 1: First "pipe"

Step 2: Multiply temp matrix and matrix B

Step 3: First "roll"

Step 4: Second "pipe"

General crowd computation.



Two matrices to be multiplied, being fed as input to mesh of processors.

Chapter 5

Timing Values

In this chapter, we compare the timing values of various programs if they are executed on *single* processor, *two* processors and *three* processors. The programs are executed on dec ALPHA machines. Here we have a network of three Digital ALPHA Station255 with Digital UNIX. The programs are executed on one, two and three processors and the results are compared. The tasks created are equally distributed on all the processors.

The timing values are given in number of clock ticks. Number of clock ticks for our dec ALPHA machine is 60 ticks per second. When we measure the execution time of a process, we see that Unix maintains three values for a process.

- clock time
- user CPU time
- system CPU time

The *clock time* is the amount of time the process takes to run, and its value depends on the number of other processes being run on the system.

The *user CPU time* is the CPU time that is attributed to the user instructions.

The *system CPU time* is the CPU time that can be attributed to the kernel, when it executes on behalf of the process.

The sum of user CPU time and the system CPU time is often called the CPU time.

In this chapter we have noted the real time of completion of execution of parallel program. We have also noted the maximum CPU time taken by any process.

The execution time for different matrix sizes and different number of processors are also shown graphically at the end of this chapter.

Matrix Size(row x col)												
Number of Processes = 1												
Time (60 x secs)	120 x 120			240 x 240			360 x 360			480 x 480		
No.of Processors	1			1			1			1		
Real Time	15			116			468			1364		
Max CPU time	14			114			463			1349		
Number of Processes = 2 x 2												
No.of Processors	1	2	3	1	2	3	1	2	3	1	2	3
Real Time	26	22	22	138	114	103	154	375	313	1281	875	740
Max CPU time	3	2	2	21	22	22	79	81	82	186	189	189
Number of Processes = 3 x 3												
No.of Processors	1	2	3	1	2	3	1	2	3	1	2	3
Real Time	42	28	22	155	103	90	446	271	204	1125	709	536
Max CPU time	3	3	3	15	16	15	46	49	47	125	121	122
Number of Processes = 4 x 4												
No.of Processors	1	2	3	1	2	3	1	2	3	1	2	3
Real Time	65	37	26	165	98	96	486	245	201	1032	562	486
Max CPU time	3	2	2	9	10	9	27	28	28	63	64	63
Number of Processes = 5 x 5												
No.of Processors	1	2	3	1	2	3	1	2	3	1	2	3
Real Time	93	42	31	193	116	114	510	273	221	1089	557	435
Max CPU time	2	1	2	7	7	6	18	20	20	40	41	43
Number of Processes = 6 x 6												
No.of Processors	1	2	3	1	2	3	1	2	3	1	2	3
Real Time	-	63	42	-	148	121	-	299	251	-	614	454
Max CPU time	-	2	2	-	5	4	-	15	14	-	28	29

Table 5.1: Execution Time for the Matrix Multiplication on Torus Interconnection network (Pipe-Multiply-Role Algorithm)

" - " indicates given number of processes can not be created on given number of processors.

Real time : real time taken for completion of the program.
Max CPU time : Maximum CPU time taken by any process in the parallel program.

Matrix Size(row x col)												
Number of Processes = 1												
Time (60 x secs)	120 x 120			240 x 240			360 x 360			480 x 480		
No.of Processors	1			1			1			1		
Real Time	17			197			883			2752		
Max CPU time	14			114			458			1359		
Number of Processes = 2 x 2												
No.of Processors	1	2	3	1	2	3	1	2	3	1	2	3
Real Time	30	23	28	160	108	121	612	412	433	1483	880	842
Max CPU time	6	4	5	31	29	30	109	110	110	234	240	249
Number of Processes = 3 x 3												
No.of Processors	1	2	3	1	2	3	1	2	3	1	2	3
Real Time	55	37	35	171	120	112	531	319	288	1342	814	666
Max CPU time	4	3	3	110	14	14	46	44	45	115	116	115
Number of Processes = 4 x 4												
No.of Processors	1	2	3	1	2	3	1	2	3	1	2	3
Real Time	92	60	54	205	120	105	541	295	249	1189	653	597
Max CPU time	3	1	1	10	8	8	27	25	28	60	59	60
Number of Processes = 5 x 5												
No.of Processors	1	2	3	1	2	3	1	2	3	1	2	3
Real Time	144	96	68	257	164	130	594	320	285	1252	726	708
Max CPU time	3	2	1	7	6	6	18	17	17	40	39	39
Number of Processes = 6 x 6												
No.of Processors	1	2	3	1	2	3	1	2	3	1	2	3
Real Time	231	125	72	318	155	147	669	348	285	1320	862	820
Max CPU time	2	1	1	6	3	4	13	17	17	29	27	28

Table 5.2 : Execution Time for the Matrix Multiplication on Mesh Interconnection network

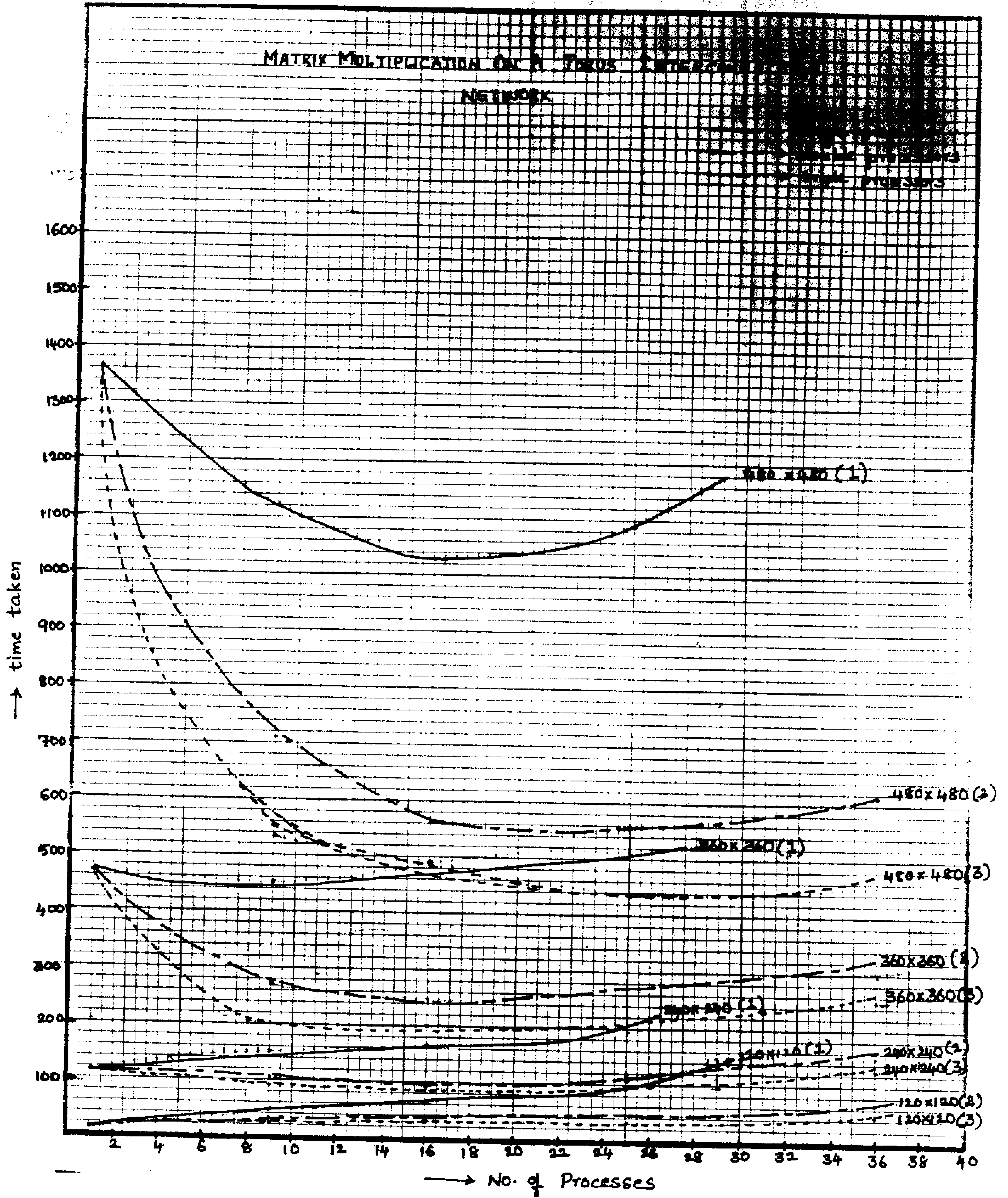
Real time : real time taken for completion of the program.
Max CPU time : Maximum CPU time taken by any process in the parallel program.

Matrix Size(row x col)												
Number of Processes = 1												
Time (60 x secs)	**32 x 32**			**64 x 64**			**128 x 128**			**256 x 256**		
No.of Processors	**1**			**1**			**1**			**1**		
Real Time	6			27			87			316		
Max CPU time	2			9			23			80		
Number of Processes = 2 x 2												
No.of Processors	1	2	3	1	2	3	1	2	3	1	2	3
Real Time	5	6	7	66	59	58	159	162	148	498	523	479
Max CPU time	3	3	3	13	12	11	32	33	32	97	100	104
Number of Processes = 4x 4												
No.of Processors	1	2	3	1	2	3	1	2	3	1	2	3
Real Time	8	6	6	246	156	113	525	364	257	1248	905	695
Max CPU time	3	3	3	38	30	26	83	73	70	194	187	182
Number of Processes = 8 x 8												
No.of Processors	1	2	3	1	2	3	1	2	3	1	2	3
Real Time	-	8	7	-	755	505	-	1877	1136	-	4234	2543
Max CPU time	-	3	4	-	118	108	-	307	287	-	710	674

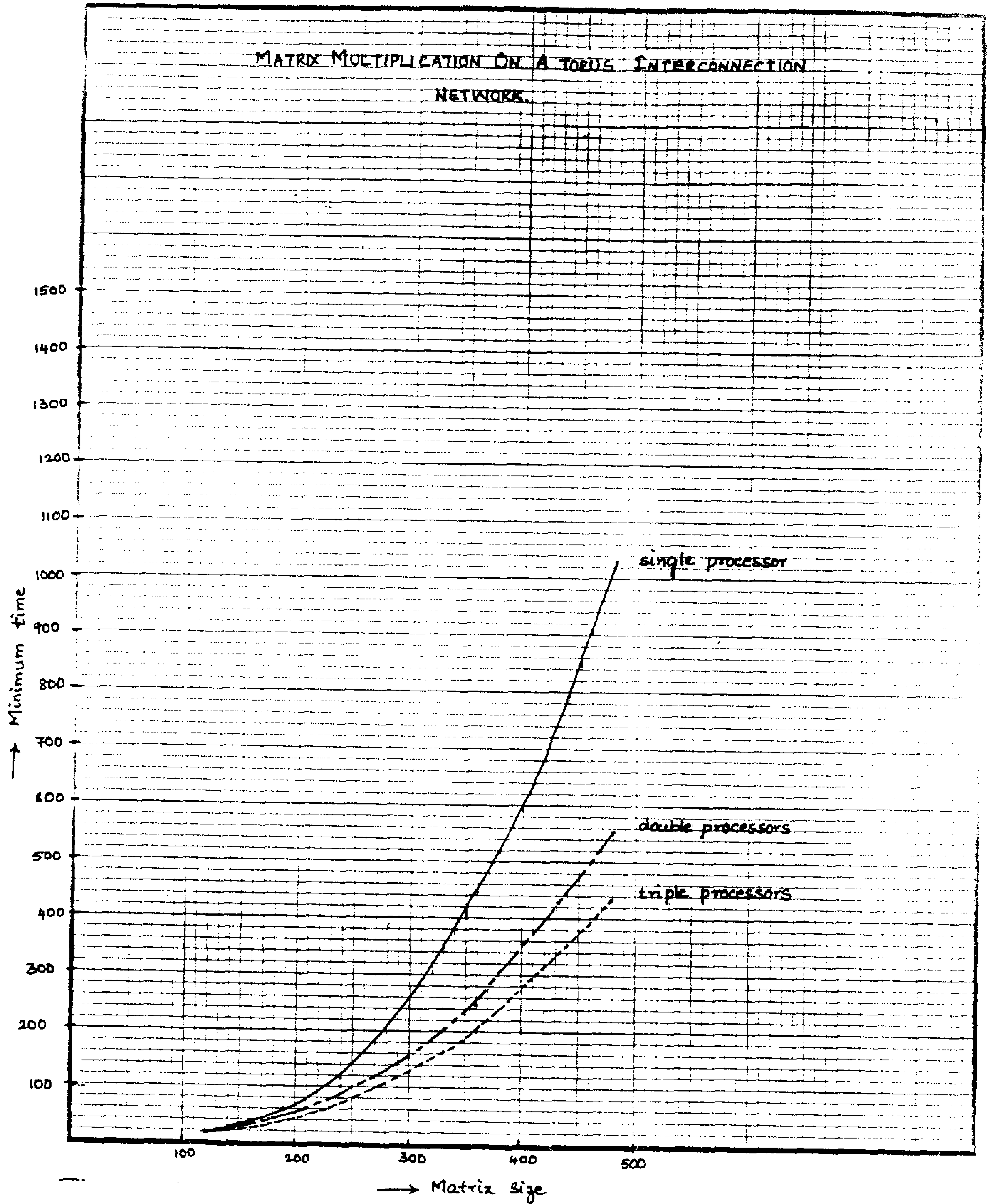
Table 5.3 : Execution Time for the Matrix Inversion

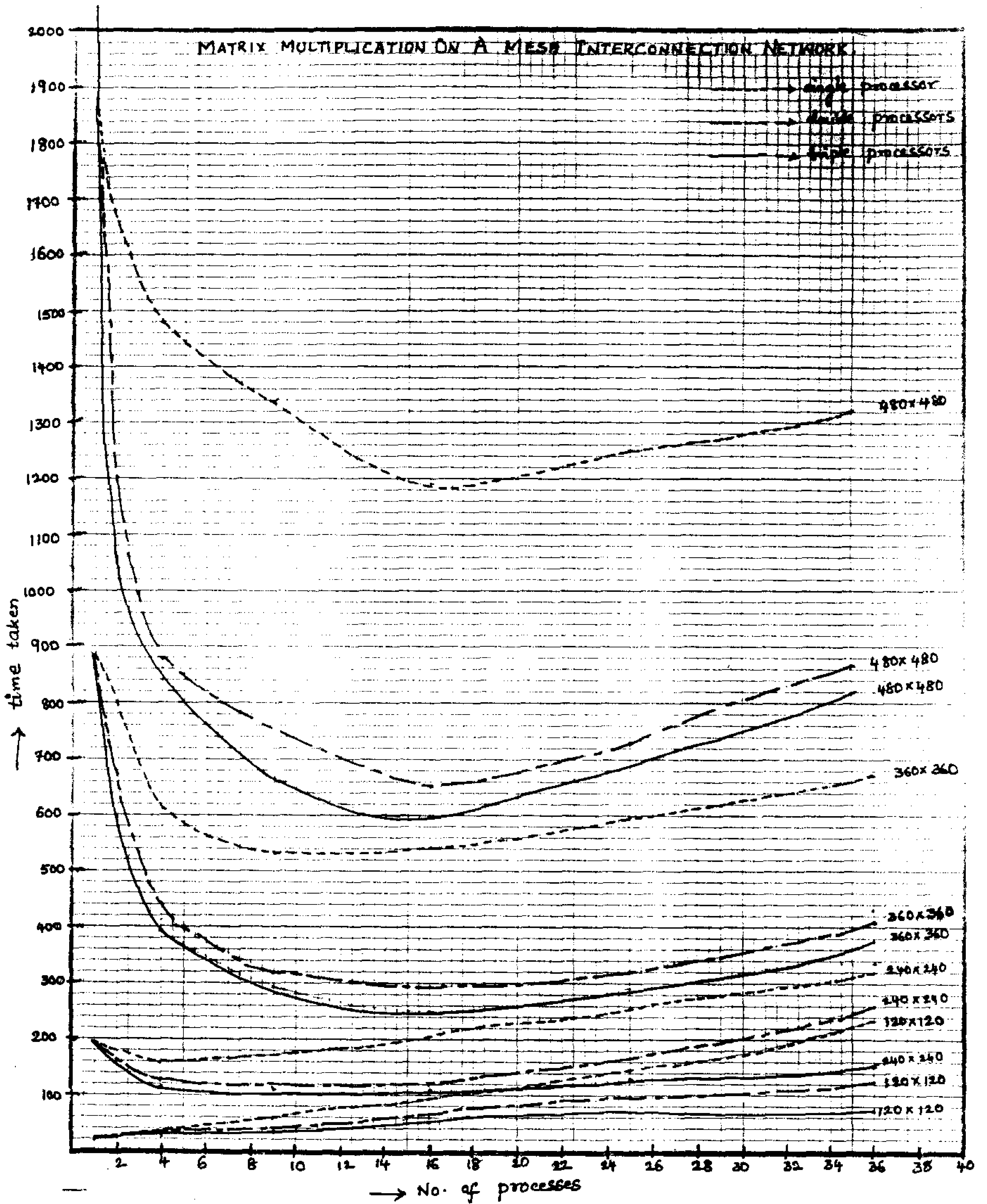
" - " indicates given number of processes can not be created on given number of processors.

Real time : real time taken for completion of the program.
Max CPU time : Maximum CPU time taken by any process in the parallel program.



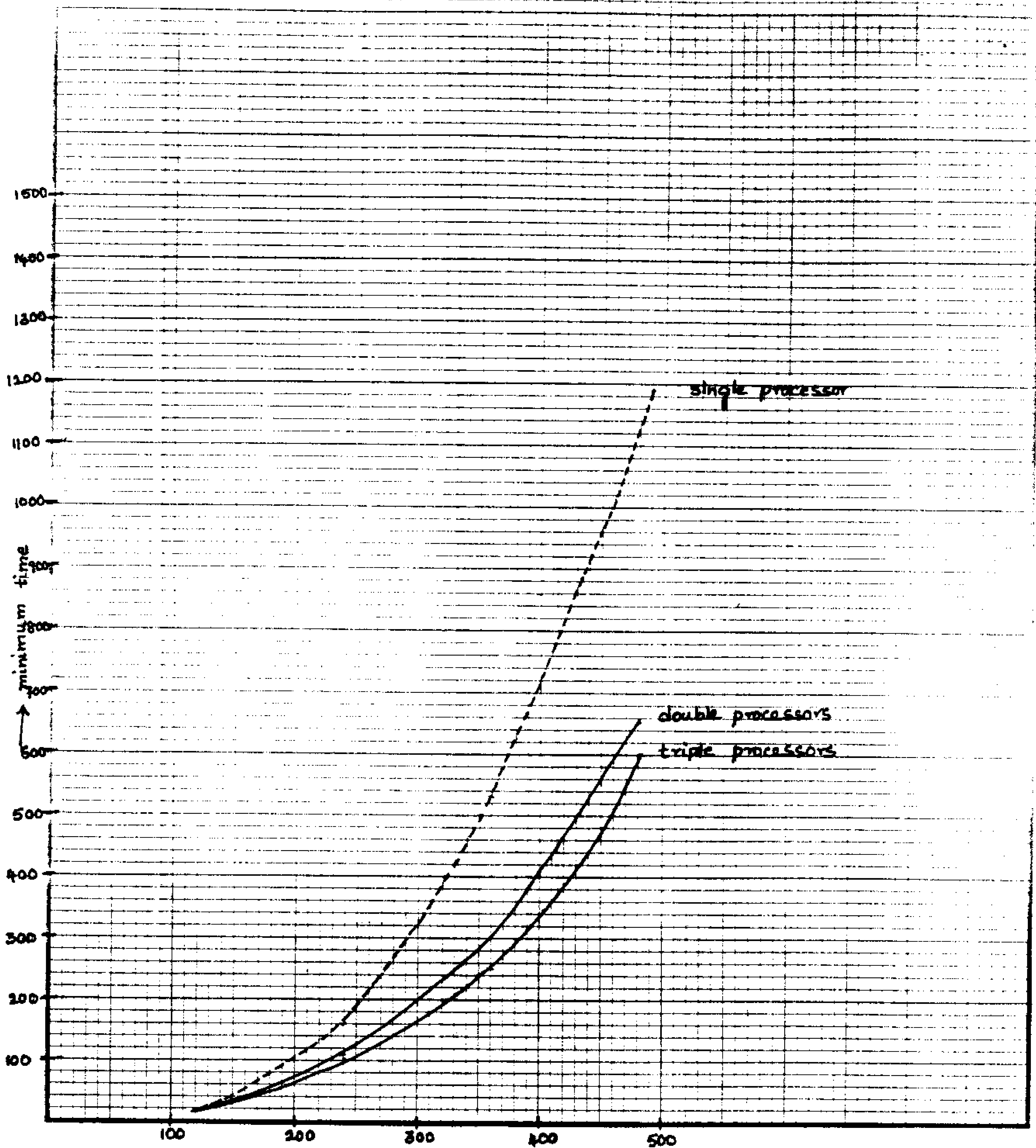
MATRIX MULTIPLICATION ON A TORUS
NETWORK
1600
1500
1400
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
→ time taken
480 x 480 (1)
480x480 (2)
360x360 (1)
480x480 (3)
360x360 (2)
360x360 (3)
240x240 (2)
240x240 (3)
120x120 (2)
120x120 (3)
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
40
→ No. of Processes


MATRIX MULTIPLICATION ON A TORUS INTERCONNECTION NETWORK.
single processor
double processors
triple processors
1500
1400
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
→ Minimum time
100
200
300
400
500
→ Matrix size



MATRIX MULTIPLICATION ON A MESH INTERCONNECTION NETWORK
processor
processors
processors
2000
1900
1800
1700
1600
1500
1400
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
480 x 480
480 x 480
480 x 480
360 x 360
360 x 360
360 x 360
240 x 240
240 x 240
120 x 120
240 x 240
120 x 120
120 x 120
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
→ time taken
→ No. of processes

MATRIX MULTIPLICATION ON A MESH INTERCONNECTION NETWORK



Chapter 6

Conclusions

In the previous chapter we have given a comparison of the time taken by different programs by varying number of processors. Here we have a network of three Digital ALPHA UNIX workstations. So all the programs are executed on one, two and three processors and the time taken to complete execution of programs are compared. We have implemented different operations on matrices and the timing values are noted(as given in the previous chapter). It is obvious from the tables that *as the size of the matrices increases, the parallel programs take lesser time to complete execution, than the corresponding sequential programs.*

From the graphs given in the previous chapter, it is obvious that as the matrix size increases, dividing a program into number of sub-tasks results in faster execution of a program. But just dividing a program into any number of sub-tasks will not yield faster computation as it is obvious from the graphs given in the previous chapter. For small size matrices, dividing a program into number of sub-tasks involves overhead in the form of process switching. So the time increases as number of sub-tasks increases for smaller size matrices. But for a bigger problem dividing the program into number of sub-tasks involves less computation time. As is shown in the graphs, for a larger size matrices as the number of tasks increase, up to some point, the time decreases and then time taken increases as we increase the number of processes further. This is because of the overhead involved in process switching between tasks. That point gives the optimum value in terms of number of processes for a given size of the matrix.

For a given size matrix,

We have implemented different operations on matrices and the details are given in chapter 4. All the programs are executed on *one, two* and *three* processors. When multiple tasks are run on a single processor a substantial amount of overhead is incurred in the form process switching. But when the programs are run on more than one processor, *message passing time*, and not the process switching time will contribute to the maximum overhead time.

So if we run a program on two processors, the time is not reduced to half as can be thought of because of message passing overhead time. Suppose if we have *n* number of processors and we create *n* number of tasks, distributing each task to one processor, the time is not reduced by *n* because of the message passing time involved. So it is not the only criteria of dividing the a problem into number of tasks as the number of processors available, but we also have to take into account the message passing time involved. So we have to find the optimum number of tasks that a given problem can be divided into and optimum number of processors onto which the created tasks are distributed.

The timing values are noted by distributing the number of tasks equally among the number of processors present. These are given in the previous chapter. the timing values are also observed taking into consideration the load on each processor. The load on each processor is taken as the number of ready processors present at the time of distribution of tasks. More number of tasks are given to the processor which has less load. There is comparable decrease in the time values when the tasks are distributed according to the load. At the same time we have to take into consideration message passing time while distribution of tasks. There are many other factors which have to be taken into consideration like processor speed, network distance etc. Since all the machines in the network here are of same type, the speed of each processor is same. If we have a environment where we have machines of different

architectures, this factor becomes important. We can assign a complicated task to the machine which has faster speed. So by taking all these into consideration one can make a task scheduler.

Here we have dealt with some problems on matrices, but many other problems like sorting, solving partial differential equations, etc. can be parallelized in a like manner, by breaking up into sub-tasks.