Performance Of Multithreaded Computations On High-Speed Networks

BY

AJAY KIRIT PANDYA

A Thesis

Submitted to the Faculty of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

University of Manitoba

Winnipeg, Manitoba

Canada

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

PERFORMANCE OF MULTITHREADED COMPUTATIONS ON HIGH-SPEED NETWORKS

BY

AJAY KIRIT PANDYA

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Abstract

Inter-host communication has always been a performance bottleneck for Distributed Computing Systems exchanging large amounts of data between hosts. This is due to the the low speed, and low shared bandwidth provided by the networking technologies that are currently used, such as Ethernet and Token Ring. With the emergence of several high-speed and high-bandwidth networks like ATM and Gigabit Ethernet, fast inter-host communication for solving real time problems, has become possible.

Almost all software systems that support distributed concurrent computing use a process based computing model and a message-passing based communication model. However, recent research developments suggest that a thread based model for computing and scheduling has specific advantages over the traditional model.

In this thesis, we have implemented a thread based computing model and a thread based intra-host communication model and have tested two networking technologies namely ATM and Ethernet for supporting these models. We implement a recursive distributed matrix multiplication algorithm and the results are very encouraging, indicating the feasibility of developing portable and better performing thread based systems for distributed concurrent computing.

Acknowledgment

I would like to express my sincere gratitude to Dr. David Blight and Dr. Bob McLeod, my advisors, for their encouragement and help during my research. I would also like to thank Dr. Randal Peters, my external examiner, for his useful comments which were a valuable contribution for this thesis.

I thank Mr. Anindya Maiti for his useful ideas, and a part of this thesis was a joint effort between us. A special thanks goes to Mr. Guy Jonatschick for his help in setting up the network. I would like to acknowledge TR*Labs*, Winnipeg, for letting us use their network. I would also like to thank the Faculty of Graduate Studies for their financial support (in the form of the University of Manitoba Graduate Fellowship), and Dr. R. S. Azad, Department of Mechanical and Industrial Engineering for the financial support during the first year of my course.

I also thank all my friends who made my stay in Winnipeg a memorable one. Finally, I thank God, and my parents for their blessings in making this dream into a reality. This thesis is dedicated to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Internet, developed in the mid 1960s, has proved to be a benchmark of the information revolution that is leading us into the new millennium. The World Wide Web (WWW), as a result of the Internet, is the fastest growing ocean of information and knowledge that man has ever created. Internet, or "internetworking" deals with connecting a large number of same or different computers on a network, and can exchange information amongst themselves. One of the several advantages of connecting a number of computers together is that they can collectively work in a shared environment, with increased efficiency for distributed applications. In this thesis, we have examined the role of the Internet and computer networks for performing large-scale engineering and scientific computational tasks.

Problems such as weather forecasting, engineering simulations, numerical scientific computations, etc. require immense computational power and CPU time. Such computations are solved on Massively Parallel Processor (MPP) machines, or Supercomputers. MPPs are the most powerful computers in the world. These machines combine a few hundred to a few thousand CPUs in a single cabinet connected to hundreds of gigabytes of memory. As simulations become more realistic, the com-

putational power required to execute them grows rapidly. Thus, researchers on the cutting edge turn to MPPs and parallel processing in order to get the most computational power possible.

One of the major developments affecting scientific problem solving is *Distributed Computing*. Distributed computing is a method in which a set of computers connected by a network are used collectively to solve a single large problem. As more and more organizations have high-speed Local Area Networks (LANs) interconnecting many general-purpose workstations, the combined computational resources may exceed the power of a single high-performance computer. Although computers distributed on a network do not provide the raw computational power of a large MPP, they are able to solve problems that are several times larger than the ones for which a single workstation is designed. Thus, MPPs can be substituted by a collection of workstations scattered across a network. In this thesis, we have used a collection of workstations connected by a network to collectively solve a scientific problem, such as a parallel matrix multiplication computation.

Until now, distributed computing lacked the speed for interprocess communication between the cooperating processes, if the processes ran on hosts that were located at very large geographical distances from one another. However, with the emergence of several high-speed switch-based networks, such as the High Performance Parallel Interface (HIPPI), Fiber Channel, and Asynchronous Transfer Mode (ATM), the possibility of networks effectively supporting communication intensive parallel applications may soon become a reality. ATM has emerged as the most standard and widely used networking paradigm for high-speed, high bandwidth communication. So, if the workstations that are used for distributed computation are connected by an ATM network, the performance bottleneck of fast interprocess communication can be significantly reduced.

A common feature between MPP and distributed computing is *message passing*.

In all parallel processing systems, data must be exchanged between the cooperating tasks. Several paradigms have been tried including shared memories, parallelizing compilers, and message passing. The message-passing model has become the most popular model from the perspective of the number and variety of multiprocessors that support it, as well as in terms of applications, languages, and software systems that use it.

The Parallel Virtual Machine (PVM) [11] has been developed as a joint research effort by the Oak Ridge National Laboratory at the University of Tennessee at Knoxville, Emory University, and Carnegie Mellon University. It uses the message-passing model to allow programmers to exploit distributed computing across a wide variety of computer types, including MPPs. PVM makes a collection of computers appear like one large *virtual* machine. It has proved to be the *de-facto* standard for Heterogeneous Network Computing. A number of computers connected through a high-speed network, and running PVM can be made to act as a single parallel machine, that can be used to solve computationally intensive problems.

When a number of computers are connected to form a virtual machine using PVM, each machine is referred to as a *node*. In distributed computing, a single problem is divided up into several tasks and each task is given to a node for computation. In this case, the node where the subtasks are invoked is called a *master* node and the nodes that receive the subtasks are called the *slave* nodes. After the slaves have finished their work, they send back the results to the master. PVM helps in passing the tasks between the machines and also in passing messages between the tasks.

As mentioned above, the workstations available today such as Sun SPARC stations, SGI workstations, IBM RISC machines, etc. have one or more than one processors in them ranging from 1 processor to over 1000 processors mounted inside one cabinet [12]. So if a task is given to such a *multiprocessor machine*, only one of its processors, to which the task will be given by its operating system, will be utilized

[16]. The reason being that the individual task will always be in the form of a serial procedure. This feature of simple workload allocation in the form of serial tasks to individual nodes, which are multiprocessor machines, greatly reduces the actual amount of computational power that can be obtained from them. Therefore, it would be of great interest for researchers to come up with algorithms that not only divide the given problem into several tasks, but also to fully utilize the available resources, by parallelizing the individual tasks too. These tasks concurrently execute on individual slave nodes and also exchange information with other tasks, either by message passing, or by operating in a Distributed Shared Memory (DSM).

Multithreading is a relatively new approach in designing distributed applications. Modern Operating System (OS) platforms like Windows NT, OS/2, IBM's AIX, and several flavors of Unix, provide extensive library and system call support for multithreaded applications. Multithreading, unlike Multitasking or Multiprocessing[1] allows a single process to perform more than one task at the same time. Each task that is executing within a single process is called a *thread.* A thread is a *lightweight process* that resides in its parent process's address space. It is a lighter burden on the operating system compared to a process. When a process uses more than one thread, the process is said to be *multithreaded.* Multithreading allows a programmer to divide a process into a series of threads that can be executed at the same time. If the process is running on a computer that has several processors, each thread may be executed on a separate processor. This allows the programmer to implement tasks that can be executed in parallel within a single program.

This thesis deals with evaluating the performance of different high-speed networks for running computationally intensive distributed applications using PVM. The algorithm must be written in such a way that the application forks several threads at each

[1]Multitasking means an operating system can handle more than one tasks at the same time using context switching while Multiprocessing means that multiple processes can be executed simultaneously.

node, thereby parallelizing the subtasks (provided that that the subtasks are parallelizable). The improved performance of parallel multithreaded algorithms is studied against simple parallel algorithms, and thereby different networking paradigms are compared for running complex scientific and engineering simulations that are multithreaded in nature.

1.1 Organization of the thesis

The rest of the thesis is organized as follows. Chapter 2 gives a detailed review of the related research done in this area. Chapter 3 gives an overview of Cluster Computing, in which different networking technologies such as Ethernet, FDDI, ATM and Gigabit Ethernet are discussed, along with the feasibility of building Local Area Networks (LAN) using them. The PVM message passing library is also explained along with its extensions like the Migration based PVM (MPVM), and Lightweight Thread Based PVM (LPVM), and how they can be uses for performance improvement. The algorithms used in this thesis are discussed in Chapter 4 and their runtime results on ATM and Ethernet (10 base-T) are given and analyzed in Chapter 5. We finish off with the conclusions and possible future research directions in Chapter 6.

Chapter 2

Related Research

This chapter surveys past work related to Distributed Network Computing. We have done the review separately for each sub-area. In Section 2.1, we discuss the work done in evaluating network performance for distributed applications. Most commonly, we talk about using various high-speed networking technologies like ATM, HIPPI, and Fiber Channel for faster message passing. In Section 2.2, we talk about work done in improving the present PVM research, and how the PVM model has been modified by adding new features for improved performance. In Section 2.3 we discuss some research in parallel matrix multiplication that we have used for this thesis. Lastly, in Section 2.4, we discuss work done in distributed computing using other paradigms such as the Berkeley Network of Workstations (NOW) project.

2.1 High-speed Distributed Computing

The Distributed Multimedia Research Centre (DMRC) at the University of Minnesota is doing active research in improving performance of distributed systems running on high-speed backbones. Lin *et al* [21] have performed experiments with ATM networks to examine their performance for solving computationally intensive problems.

They have studied the end-to-end communication performance in terms of latency in an environment consisting of Sun workstations (with Fore Systems' ATM interface cards) connected through a Fore Systems' ASX-100 switch. They compare the performance of four different Application Programming Interfaces (API) such as Sun Microsystems' Remote Procedure Calls (RPC), BSD socket programming interface, PVM message passing library, and Fore Systems' ATM API. They conclude that BSD sockets and Sun RPC/XDR are not highly suitable for implementing high performance computing applications over a cluster of networked workstations. PVM is very efficient in developing these applications because of its features like data-type encapsulation, process group communication, remote process spawn, and dynamic process control. However, the PVM implementation on ATM (pvm-atm) [1] introduces more protocol overhead than the Fore Systems' API. Thus, the latter is most efficient for computationally intensive distributed applications.

Chang *et al* [6] have studied the performance of PVM over a local ATM network. They conclude that a high-speed network such as ATM, as opposed to a conventional network such as Ethernet, does provide increased communication bandwidth. They have achieved maximum bandwidth of 27.202 Mbps. at the application layer, which is far below the "raw" available bandwidth of 100 Mbps. provided by the TAXI interface cards.

Hsieh *et al* [14] at the DMRC have also extended PVM capabilities over a HIPPI LAN. They have compared the performance of PVM implementation over Ethernet with that over HIPPI. Similar to the Fore Systems' API for ATM networks, Hewlett Packard has the Link Level Application (LLA) programming interface. They have used the LLA API for implementing PVM over HIPPI and Ethernet, and thereby

[1] pvm-atm is an implementation of PVM for ATM networks so that PVM programs can run over ATM networks. It gives an option of message passing using either ATM AAL3/4, ATM AAL5, or original PVM (TCP/UDP), and is implemented on Fore Systems' API instead of the BSD socket interface. It is available at ftp://ftp.cs.umn.edu/users/du/pvm-atm/www.html

replacing the overhead of PVM's default protocol stack (namely UDP) for inter-daemon communication and TCP for inter-task communication. They have achieved superior performance because of the high-speed network medium offered by HIPPI, and due the replacement of PVM protocols by lower layer protocols (LLA API).

Huang *et al* [15] have presented the results of an investigation of collective communication operations for distributed computing across ATM networks. They have studied the performance of a thread-based software for ATM hardware features like ATM multicast channels. They propose a software framework based on reliable *multicast* connections, which are implemented on top of the unreliable ATM multicast (one-to-many) virtual channels. They have used *threads* within a single process, as against the conventional Unix process for implementing this. Hence they have combined the connection oriented nature of ATM and the ATM multicast virtual channels. This has been implemented using threads. They have incorporated this combination as a basic building block in a multicast virtual topology. Other related papers and technical reports of the Communications Research Group at the Michigan State University, which deal with aspects like hardware improvement of ATM multicast, or efficient I/O mechanisms for faster communication, are available at ftp://ftp.cs.msu.edu/pub/crg.

2.2 PVM Modifications

In this section, we discuss the work which has been done to improve the presently available PVM model for faster, and more efficient performance. Zhou and Geist [27] have developed a faster message passing route named *PvmRouteAtm* to exploit the bandwidth of an ATM network based on the socket-like application programming interface from Fore Systems. The comparison of this route is done with the standard route *PvmRouteDirect*. Test results show that *PvmRouteAtm* succeeds to reach a better bandwidth for large amounts of data but fails to gain any latency improvement over its counterpart route *PvmRouteDirect*. For this thesis, we have used *PvmRoute-*

Direct as it is inbuilt in the *pvm-atm* at DMRC. Implementation of *PvmRouteAtm* in the present *pvm-atm* will be our future work.

In another work, Zhou and Geist [26] have proposed a Light-weight process based implementation of PVM called LPVM, as explained later in section 3.2.3. The current PVM available has a process based computing model. LPVM has been designed to examine potential performance improvements by multithreaded message passing systems. The major disadvantage of LPVM model is its implementation only on a single SMP machine. Currently, research is being done at the Oak Ridge National Laboratory, Tennessee for extending this LPVM implementation to a cluster of SMPs that have completely disjoint address spaces. Another major drawback is the portability issue. Since LPVM is a modified version of PVM with added features like thread safety and with a different user interface, programs already written for PVM can not be transparently ported to LPVM systems.

Before the development of LPVM, a similar thread based message passing library was proposed by Ferrari and Sunderam [10] called TPVM. The major difference between TPVM and LPVM is that TPVM is built on top of the PVM system and no change is made to the underlying PVM system. This definitely is good for portability, but has a large added overhead. TPVM has been implemented over a cluster of SMPs, and threads operating in different processes communicate using explicit message passing. Subsequent work added the concepts of *Remote Memory* and *Data Driven Programming*. In TPVM, processes are spawned in the same way as PVM, but they are not themselves the computational units of parallelism and/or scheduling. The PVM system exports the entry points of the thread subroutines that decide the action of the threads. In this thesis, the thread model used is a combination of LPVM and TPVM in the sense that we have not built our system over PVM, but have kept the original PVM interface with sub-task based thread computations. This means that the multithreading occurs only on the slave processes and is transparent to the

master process and to the overall system. This approach does not impose any extra overhead, however it increases the programming complexity, because any distributed application has to be separately implemented to incorporate thread based parallelism. This is not a major drawback with the availability of easy and simple to use thread packages like Solaris, Posix, Java or Win32. Java and Posix specifications are being made portable so that they can be used on heterogeneous platforms.

2.3 Matrix Multiplication

We have implemented Strassen's Matrix Multiplication algorithm [17] and Block Matrix Multiplication algorithm [13] for doing sub-task computations, and the Scalable Universal Matrix Multiplication Algorithm (SUMMA) [25] for workload allocation. There has been some research for the parallel implementation of the Winograd's variant of Strassen's algorithm. Bjørstad *et al* [3] have recently come up with techniques to efficiently implement matrix multiplication algorithms on SIMD computers. They have chosen the cutoff dimension, n_0=128, and implemented it on a 8192 processor machine. They achieve superior performance of the algorithm for different size and shapes of matrices. Bailey [2] has also implemented Strassen's algorithm on a CRAY-2 and reported speedups of up to 2.01 for n=2048, where n is the matrix size. Compared to Bailey's results, Bjøorstad *et al* report that their algorithm was faster than the block methods for any levels of recursion on the one processor CRAY-2 machine using the CRAY MXM library. For our parallel implementation of Strassen's algorithm, we have parallelized the Strassen's algorithm according to the technique shown by Bjørstad *et al* [3].

Kleinman *et al* [18] from SunSoft Inc., Mountain View, California give a multithreaded implementation of matrix multiplication where each thread multiplies one row by one column in a cyclic fashion. Each iteration computes the results of one entry in the resultant matrix. They offer a flexible implementation whereby one

thread can compute a whole row of the resultant matrix, if the amount of work is not sufficient to justify the overhead of synchronization. We use this coding style for implementing our Block Matrix Multiplication algorithm.

An MPI implementation of SUMMA is done by Van De Geijn and Watts [25]. We use SUMMA for workload allocation at the master node, where a matrix is divided up into equivalent tasks according to the mesh configuration. After implementing SUMMA, the authors of [25] discuss some cases where SUMMA can be less efficient than its counterpart PUMMA (Parallel Universal Matrix Multiplication Algorithm). Regardless, we have used SUMMA as our task allocation algorithm and have implemented it in PVM using a similar scheme as the MPI implementation. SUMMA has been selected against PUMMA because it is competitive or faster, and given its simplicity and flexibility, warrants consideration even though it is slightly more sensitive to communication overhead compared to PUMMA..

Lastly, Huss-Lederman [17] show a parallel implementation of Strassen's algorithm that deals with matrices having odd numbered dimensions using either dynamic peeling and/or dynamic padding. They stop the recursions at an early stage and revert to a standard algorithm for more efficiency. We have implemented dynamic peeling based on their implementation.

2.4 Other Related Work

The Network of Workstaions (NOW) [1] project at the University of California, Berkeley, also deals with connecting a cluster of workstations on a network for collective high-performance computation. The NOW project seeks to harness the power of clustered machines connected via high-speed networks. A special operating system, *GLUnix*, is used on the workstations participating in the LAN. *GLUnix* is built as a layer on top of existing operating systems and provides superior performance for both parallel and sequential applications. For this, it supports features like gang-scheduling

of parallel programs, identifying idle resources in the network, allowing for process migration to support dynamic load balancing (this is still under development), and providing support for fast inter-process communication for both the operating system and the user-level applications. NOW also uses a special programming language called *Split-C*, which is a parallel implementation of C and is also called the SPMD implementation of C. More details of NOW are available at http://now.cs.berkeley.edu.

Another major development at the NOW project is *Fast Sockets* [23]. *Fast Sockets* is a user-level library that provides the Berkeley Sockets API and facilitates high-performance communication. The main reason for developing this library is to get performance gains for networked applications over high speed networks like ATM and Myrinet. It solves the problem of *package processing overhead* like the time spent in preparing packets and receiving them off the network.

Chapter 3

Cluster Computing on High-Speed Networks

Communication between processors has long been the performance bottleneck of distributed network computing. However, the recent progress in switch-based high-speed LANs has opened up a new possibility to reduce this problem. Foremost amongst the emerging networking technologies is the Asynchronous Transfer Mode (ATM), which shows great promise to increase the performance in terms communication for Distributed Network Computing. The basic purpose of this thesis is to evaluate the performance of presently available high-speed networking technologies for performing large-scale scientific computations in a distributed environment. In this chapter, we overview some of the networking technologies available today that can be used for distributed network computing.

We also discuss the Parallel Virtual Machine (PVM) message passing library that we have used for explicit message passing between the cooperating tasks residing on the same or different hosts in the LAN. Some further extensions of PVM, namely MPVM and LPVM, are also discussed along with the new MPI specifications.

3.1 Networking Technologies

High-Performance Distributed Applications require communication media that can provide high data transfer rates at lower latencies, which are scalable, and have superior paradigm flexibility in the sense that the network should not only have high data shipment capability for applications such as visualization, it should also have multicasting capability for applications using distributed shared memory, which involves frequent changes in multiple data copies. In this section, we briefly discuss the various circuit-based and switch-based networking technologies available today and examine their feasibility to support large-scale distributed computations.

3.1.1 Ethernet

Ethernet [24] is a LAN technology that transmits information at speeds of 10 and 100 Mbps. Currently, the most widely used version is the 10 Mbps twisted-pair one. Ethernet was invented at Xerox corporation, Palo Alto, California. Formal specifications were published in 1980, which turned the experimental Ethernet into an open, production-quality Ethernet system that operates at 10 Mbps. The Institution of Electrical and Electronics Engineers (IEEE) standard was first published in 1985, with the title "IEEE 802.3 Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specification." The IEEE standard has since been adopted by the International Organization for Standardization (ISO), which makes it a worldwide networking standard.

A major criticism about Ethernet is that it is nondeterministic ie. you cannot guarantee bandwidth to a user of a shared Ethernet segment. This is one of the reasons why Ethernet is undesirable for large-scale simulations, as these applications usually require a large bandwidth with an extremely high Quality of Service. However, with the advent of switching, the problem of bandwidth allocation has been somewhat reduced. Each user is assigned dedicated ports, and has his/her own Eth-

ernet segment. This greatly improves the responsiveness of the network as there will be no collisions with other users on that segment. In the Department of Electrical and Computer Engineering, a 3Com switch is used.

Standard Ethernet provides 10 Mbps shared bandwidth, shared among all users on a common communication channel. In traditional LANs, this communication channel is called a *bus*, which is a segment of cable to which many devices connect along the way. In a switched environment, the physical topology used is a star configuration, where all devices connect back to a common connection point using a separate length of cable. A *hub* (or a repeater) can be used to fan out more ports from the switch port, but in this case, the devices connected on the hub ports will share the bandwidth, and the collision domain is presented by the number of active users at any one time attached to the hub. When a switched port is dedicated to a workstation, the user's collision domain is limited to the single workstation. Figure 5.1 shows an example of an Ethernet LAN installation.

Fast Ethernet

Fast Ethernet is essentially the same as Ethernet, but ten times faster in raw transmission speed. This is achieved by increasing the clock speed and using a different encoding scheme, both of which require a better grade of wire than the standard Ethernet. Fast Ethernet also provides full-duplex communication. Full-duplex Fast Ethernet provides 200 Mbps aggregate bandwidth - 100 Mbps in each direction.

One elegant feature of Fast Ethernet is *autonegotiation*. Autonegotiation is a scheme that facilitates automatic adaptation to the highest possible communication speed found at both ends of the cable. This results in easy migration from standard Ethernet to Fast Ethernet.

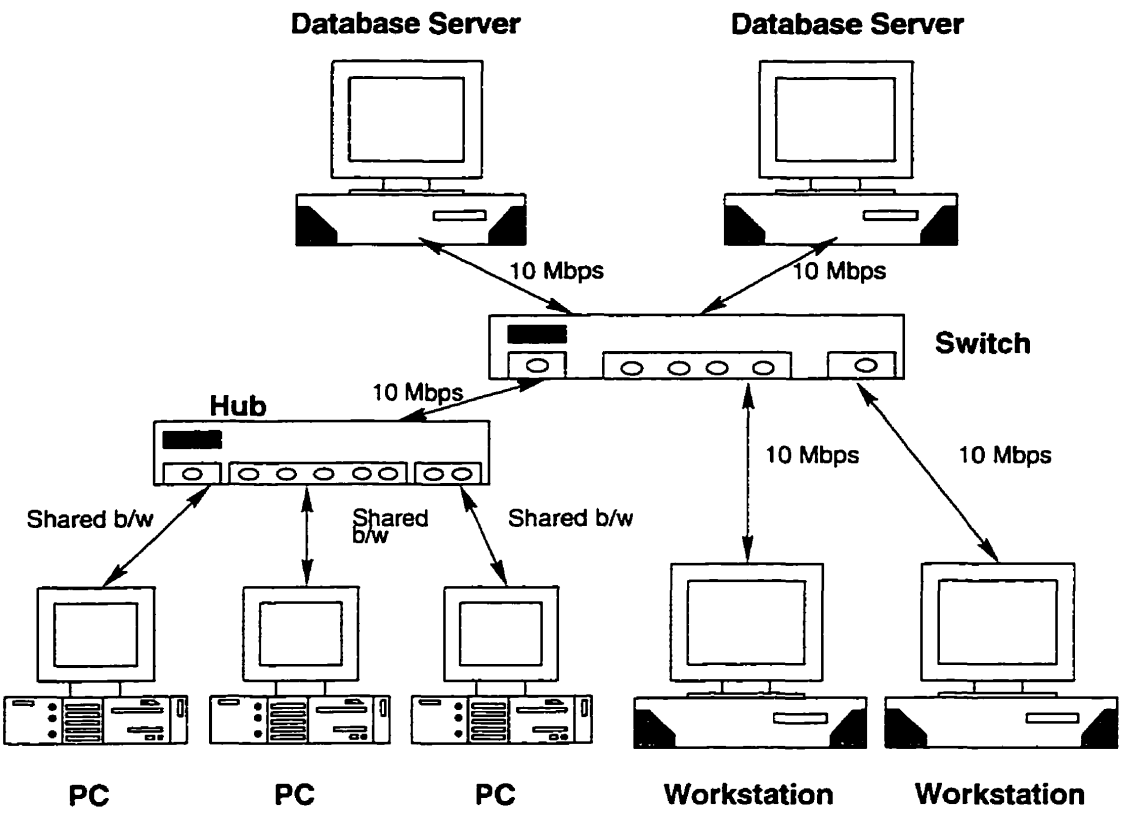


Figure 3.1: A Typical Ethernet LAN

3.1.2 FDDI

Fiber Distributed Data Interface (FDDI) [4] is a mature backbone technology that provides 100 Mbps communication. Unlike Ethernet, FDDI provides fault tolerance by incorporating a dual communication path scheme. This scheme includes two separate communication media (primary and secondary rings) that are run between devices. FDDI guarantees access by using a token passing-access method and contains a built-in network management. These features have made FDDI well suited as a backbone technology. FDDI is, however, much more expensive and significantly more complex to manage, and hence it is not widely deployed to the desktop.

Figure 3.2 shows the physical topology of a typical FDDI network. It uses a token to arbitrate communication, similar to a token ring network. Essentially, a token is created during ring initialization, and it continuously circulates around the ring. A

station must grab the token to communicate. Once it acquires a token, it puts an FDDI frame on the ring and reclaims it when it comes back to the station. By using a token-passing mechanism for communication, FDDI is able to provide a guaranteed bounding waiting time for transmission. This makes the technology more suitable than Ethernet for multimedia, since it can guarantee bandwidth for voice and video.

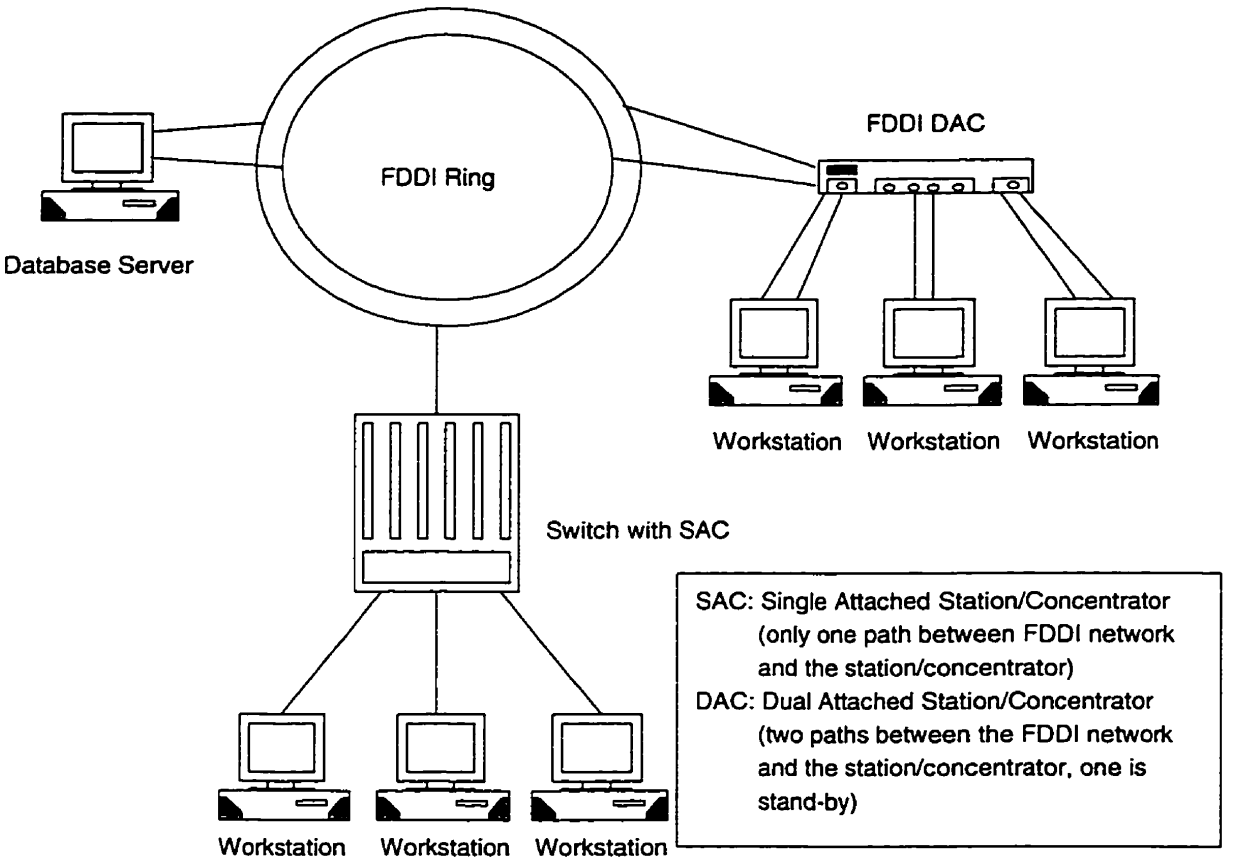


Figure 3.2: A Typical FDDI LAN

FDDI is considered to be very reliable and has low error rates. In addition to the flexibility of fault-tolerance attachments discussed above, FDDI provides a significant amount of self-management to achieve its reliability. This includes the Token Rotation Timer (TRT) to ensure that the ring is resilient to token loss, and Station Management (SMT) that includes a test for Link Quality (LCT) when a station attaches to the ring and periodic Monitoring of the Link Quality (LEM) for each station.

3.1.3 ATM

Asynchronous Transfer Mode (ATM) [5, 19] provides a common communication medium that simultaneously supports multiple types of data (multimedia) at high transmission rates across switched LAN or WAN backbones. Unlike Ethernet or FDDI, ATM is not a shared medium. The performance of the network does not degrade significantly as the number of users increase. ATM is connection oriented, meaning that an end-to-end connection is set up prior to communication, enabling ATM to provide a way to guarantee delivery with a negotiated set of parameters. This feature is called the Quality of Service (QoS).

ATM runs at different speeds. Three of the popular speeds are 52 Mbps, 155.52 Mbps (OC-3), and 622 Mbps (OC-12). It uses the notion of Permanent and Switched Circuits (PVCs and SVCs) to define communication paths within the LAN. Permanent Virtual Circuits are statically configured by the network manager for commonly used communication paths such as a highly used backbone. Switched Virtual Circuits are set up dynamically on an as-needed basis. When a user desires to establish a connection, he sends a message specifying the desired bandwidth and QoS. There is a fair amount of overhead with setting up a switched virtual circuit, especially if the connection is going to be there only for a short period of time.

LAN Emulation (LANE) was defined to incorporate ATM in existing networks consisting of Ethernet and Token Ring. LANE provides the normal connection-less service and multicast service characteristic to traditional LANs. LANE emulates the Media Access Control (MAC) protocol used by the connection-less technologies, enabling one to support legacy LAN technologies over ATM. LANE defines two major software components:The LAN emulation Client (LEC), which acts as a proxy ATM end-station for LAN stations, and the LAN emulation Server (LES), which resolves MAC address to ATM addresses. A typical ATM emulated LAN is shown in Figure 3.3.

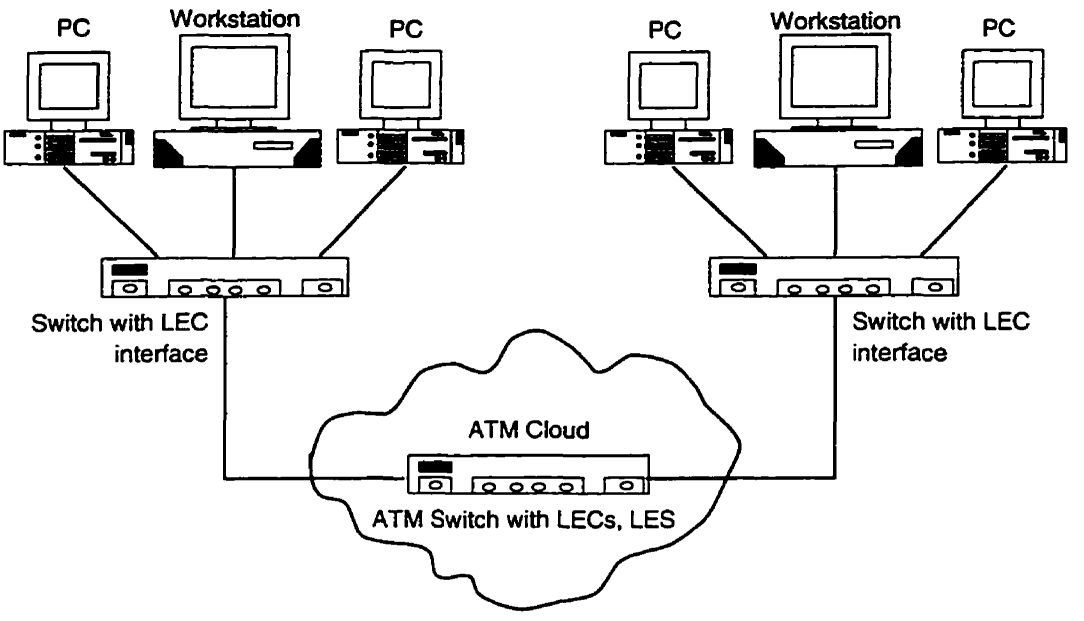


Figure 3.3: A Typical ATM LAN

Figure 3.4 shows the layered architecture of an ATM network. The ATM model is divided into three layers: the physical layer, the ATM layer, and the ATM Adaptation Layer (AAL). The physical layer defines a transport method for ATM cells between two ATM entities. It encodes and decodes the data into suitable electrical/optical waveforms for transmission and reception on the communication medium used. The ATM layer is responsible for cell relaying between ATM-layer entities, cell multiplexing of individual connections into composite flow of cells, cell demultiplexing of composite flows into individual connections, cell rate decoupling or unassigned cell insertion and deletion, priority processing and scheduling of cells, cell loss priority marking and reduction, cell rate pacing and peak rate enforcement, explicit forward congestion marking and indication, cell payload type marking and differentiation, and flow control access. Lastly, the purpose of the ATM Adaptation Layer (AAL) is to provide a link between the services required by higher network layers and the generic ATM cells used by the ATM layer. Four service classes have been defined based on three parameters: time relation between the source and the destination, constant or

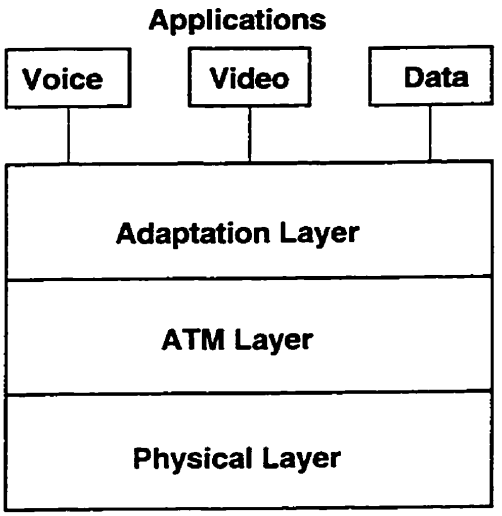


Figure 3.4: ATM architecture

variable bit rate, and connection mode. The classes are,

- *Class A*: A time relation exists between the source and the destination, the bit rate is constant, and the service is connection-oriented (eg. a voice channel). The class will use the AAL 1 protocol, defined by the ATM specifications.
- *Class B*: A time relation exists between the source and the destination, the bit rate is variable, and the service is connection-oriented (eg. a video or audio channel). Class B will use AAL 2 protocol.
- *Class C*: No time relation exists between the source and destination, the bit rate is variable, and the service is connection-oriented (eg. a connection oriented file transfer). This class will use either the AAL 3/4, or AAL 5 protocol.
- *Class D*: No time relation exists between the source and destination, the bit rate is variable, and the service is connection-less (eg. LAN interconnection and electronic mail). This class will use either the AAL 3/4, or AAL 5 protocol.

For this thesis, we have used Class C service, and have used both AAL 3/4 and AAL 5 protocols, given by the Fore Systems' specifications.

3.1.4 Gigabit Ethernet

Gigabit Ethernet [22] is an emerging standard that retains much of the simplicity of the traditional Ethernet. It uses CSMA/CD, provides full and half-duplex communication at 1000 Mbps, and retains the frame format/size. It is very easy to scale beyond Fast Ethernet, and since it is still Ethernet, troubleshooting and network management is similar. Gigabit Ethernet adds carrier extension and packet bursting. Carrier extension increases the number of bits that travel simultaneously through a connection without increasing the minimum frame length. Packet bursting allows end stations to send many frames at once, increasing bandwidth efficiency. Carrier bursting is a problem, specifically because of the carrier extension requirement.

Gigabit Ethernet is often compared with ATM because it is the first technology that rivals both 155 Mbps and 622 Mbps connections. Ethernet is more popular in the LAN because it has been around a lot longer. It is estimated at 80 percent of all desktops and servers within LANs use Ethernet. However, ATM's biggest strength is the built-in QoS. This enables ATM to offer performance guarantees when communication is established (as ATM is connection oriented), making it very attractive for real time data like multimedia. Ethernet has no guaranteed QoS; it must rely on the layers above to provide this kind of traffic management. Secondly, Ethernet allows variable-length frames (unlike fixed 53-bye ATM frames), thus making it difficult to regulate real-time flows that may get caught in the middle of a file transfer of consecutive 1500+ byte frames. Finally, Gigabit Ethernet is less expensive by about 50 percent, than the 622 Mbps ATM. This coupled with the fact that Ethernet is well understood and simple makes Gigabit Ethernet very attractive as an alternative to ATM in the LAN.

For this thesis, we used ATM because of its availability both at the Department of Electical and Computer Engineering, and at TR*Labs*, Winnipeg, Manitoba, and the availability of the pvm-atm package. However, in future we do plan to extend the

PVM implementation to Fast and Gigabit Ethernets and compare their performance with ATM.

3.2 PVM

In this section, we talk about the PVM system, its design architecture, and its routing schemes. We also address some of PVM's extensions like MPVM and LPVM, and the new MPI specifications.

3.2.1 The PVM System

For testing our algorithm in a distributed environment, we used the PVM [11] message passing library. PVM is a widely-used software system that allows a heterogeneous set of parallel and serial computers (running same or different operating systems) to be programmed as a single distributed-memory parallel machine. It is portable and runs on a wide variety of platforms. PVM is a mainstay of the Heterogeneous Network Computing research project, a collaborative venture between the Oak Ridge National Laboratory, the University of Tennessee, Emory University, and Carnegie Mellon University. We present a brief overview of PVM, its architecture, and its computing model. We also talk about MPVM (a Migration transparent version of PVM), which supports transparent process migration among the multiple hosts that constitute the virtual machine, and LPVM (Lightweight Thread-based PVM) which is another extension to PVM where heavyweight processes are replaced by lightweight processes.

PVM provides a unified computational framework for a network of heterogeneous computing resources. As mentioned before, computing resources may include workstations, multiprocessors and special purpose processors, and the underlying network may be a conventional Ethernet, the Internet, or may be a high-speed network such as

ATM. Computing resources are accessed by applications via a suite of PVM defined user-interface primitives. The PVM suite provides a standard interface that supports common parallel processing paradigms, such as message passing and shared memory. An application would embed well-defined PVM primitives in their procedural host language, usually C, C++, or FORTRAN. Recently a Java based PVM (JavaPVM), and Perl based PVM (Perl-PVM), have also been proposed, however they are still untested. The PVM suite provides primitives for such operations as point-to-point data transfer, message broadcasting, mutual exclusion, process control, and barrier synchronization. In most cases, the user views PVM as a loosely coupled, distributed memory computer with message passing capabilities, that is programmable in C, C++, or FORTRAN.

In a PVM "virtual machine" environment, there exists a support process, called a *pvmd*, or a daemon process, which executes on each host. These daemons execute independently from one another. During normal operation, they are considered equal peer processes. However, during startup, reconfigurations, or operations such as multicasting, there exists a master-slave relationship between *pvmds*. Each *pvmd* serves as a message passing router and a controller. They are used to exchange network configurable information, and dynamically allocate memory to store packets traveling between distributed tasks. They are also responsible for all application component processes (tasks) executing on their host.

Figure 3.5 depicts a network of three hosts. Each host has a local *pvmd* and a number of local tasks. Communication between hosts may occur as a task-task, task-pvmd-pvmd-task, or pvmd-pvmd interaction. Communication within a host, task-pvmd, occurs via Unix domain sockets.

As seen in Figure 3.5, ouzo has two tasks, task 6 and a console program. A console program may be used to perform tasks such as configuring the virtual machine, starting and killing processes, and checking and collecting status information of processes.

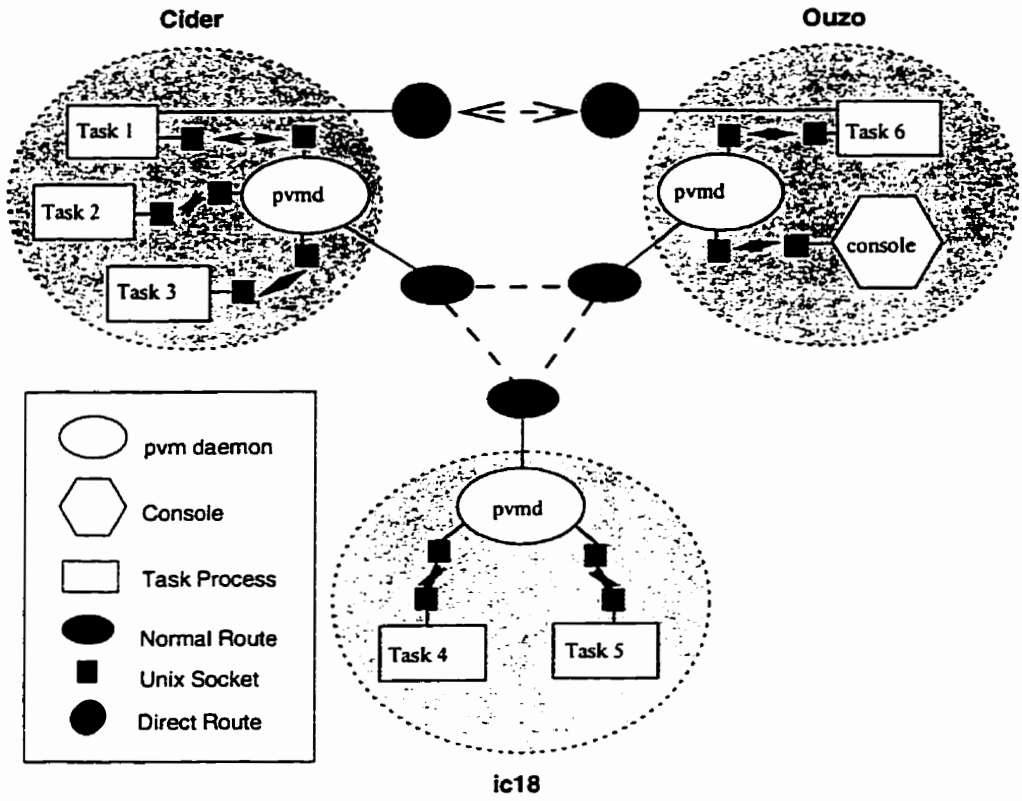


Figure 3.5: PVM task and daemon configuration (Adapted from Lin *et al* [21])

The network of independent PVM *pvmd*s form the basis for support of important features for a network-based computing environment. These features include dynamic reconfigurability, fault-tolerance and scalability.

PVM allows dynamic reconfigurability by allowing hosts to enter and exit the host pool via notification messages. PVM version 3 also supports the notion of dynamic process groups. Processes can belong to multiple named groups, and groups can be changed dynamically at any time during a computation. Functions that logically deal with groups of tasks such as broadcast and barrier synchronization use the user's explicitly defined group names as arguments. Routines are provided for processes to join and leave a named group. This dynamic reconfigurability ability also provides support for scalability and fault tolerance.

PVM provides two routing mechanisms for application messages; indirect and di-

rect routing. The choice of routing mechanism to use is controlled by the application code. By default, messages are routed indirectly. Using indirect routing, as illustrated in Figure 3.6, a message from task T2 to T3 passes through T2's local *pvmd* (*pvmd* on host 1), through T3's local *pvmd* (*pvmd* on host 2), and finally to T3. *Pvmd*-to-*Pvmd* communication uses UDP (User Datagram Protocol) socket connections while task-to-task communications uses TCP (Transmission Control Protocol) socket connection which is established during task start-up. In direct routing (PvmRouteDirect), a message from Task T2 to T4, also illustrated in Figure 3.6, uses a TCP socket connection between T2 and T4, by-passing the *pvmds* altogether. TCP connections between tasks are created "on-demand". A TCP connection is established only when a task has set its routing option to direct routing.

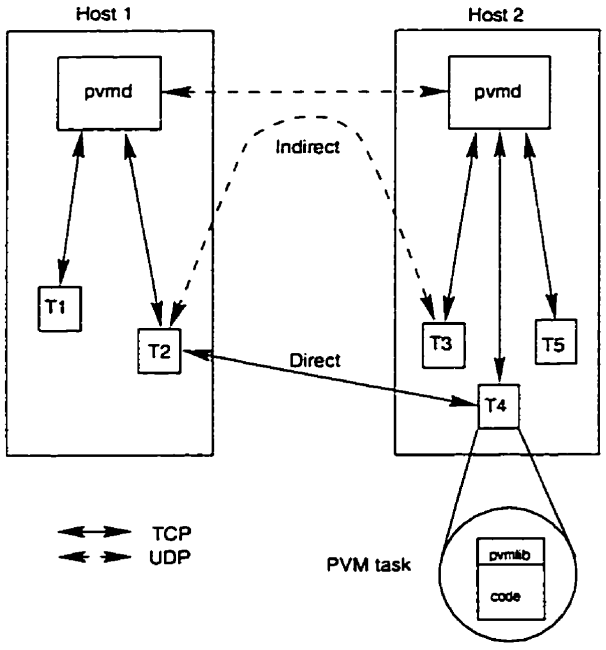


Figure 3.6: PVM Routing

In indirect routing (the default routing mechanism), the connection-less UDP sockets guarantee scalability, since a single UDP socket can communicate with any number of tasks (local or remote). Because the communication between tasks is routed

through the *pvmds*, however, messages need three hops to reach their destination. This is not very efficient. In the case of direct routing (task-to-task), connection-oriented TCP sockets are used for direct communication between the tasks. The use of TCP sockets tends to exhaust the limited number of file descriptors in a system. Since TCP connections establish a direct communication link between the tasks, however, messages reach their destination in a single hop.

In PVM 3.3.4 and above, it is possible to designate a special task as the resource manager. The resource manager, also called the global scheduler (GS), is responsible for decision making policies such as task-to-processor allocation for sensibly scheduling multiple parallel applications. Using a global scheduler makes it convenient to experiment with different scheduling policies. In MPVM, the interface between the *pvmds* and the GS has been extended to accommodate task migration, allowing the GS to use dynamic scheduling policies.

3.2.2 MPVM

MPVM is an extension of PVM, where tasks/processes running on one machine are allowed to be suspended, and then executed on another machine. MPVM makes this migration transparent to the user or the application programmer, whereby the user or the programmer does not know if a migration has occurred. MPVM is also fully compatible with PVM, in the sense that any applications written under PVM can be run under MPVM with minimal changes. MPVM is also fully portable.

Task migration may be required for the following reasons: excessively high machine load, a faster and more suitable machine becoming available, etc. When a task migrates, a major requirement is that the correctness of the task should not be destroyed. When a task needs to be migrated, it is first suspended on the machine where it is currently running, and then is reconstructed on another machine. The entire process, known as the migration protocol, is a four stage procedure. The

first stage addresses "when" the migration will occur, while the other three stages correspond to the state capture, transfer, and state-reconstruction of the task.

The Global Scheduler (GS) decides whether the task has to be migrated or not. If it decides in favor of migration, it sends a control message (CM) to the *pvmd* of the host on which the task is already running. CMs are invisible to the application code (eg. *SM_MIG*, in which *SM* stands for scheduler message, and *MIG* stands for migrate). Upon receipt of *SM_MIG*, the *pvmd* on the host currently running the task verifies the tid (task number) to ensure that the it is a locally running task. The migration initialization is divided into two components, that occur in parallel. The first component, executed locally involves flushing all the TCP socket connections from the *pvmd* to the task to avoid any loss of information that may be buffered. The second component is a skeleton process initialization on the remote host, that will run in the context of the original task. The state transfer of a task from one host to another involves capturing the task's state (ie. text, stack, data, etc.) on the local host, and transferring it to the remote host, where the skeleton process assimilates it in its own virtual address space. The state transfer is done via a TCP socket connection between the current task and the skeleton task. Before the skeleton task starts running, it called the *pvm_mytid()* routine to re-enroll in the PVM system.

The task migration, however, can occur only between homogeneous machines, and if the migration is implemented at user-level, then additional transparency is required if the current task uses Unix facilities like semaphores, mutexes, shared libraries, etc.

3.2.3 LPVM

LPVM (Lightweight-process based PVM) [26] system is an extension of PVM that supports the use of lightweight processes, or threads as the basic unit of parallelism. The basic idea was to improve the performance of the system by using multithreaded message passing systems, and study the effect of using multiple threads on SMP

(Symmetric Multiprocessors) in terms of latency. The idea of shared memory, and ease in thread management compared to process management, motivated the developers of PVM to implement a thread-based PVM.

Before LPVM was proposed, the designers of PVM had tried to build a subsystem that uses threads, called TPVM (Thread based PVM) [10]. TPVM was a system built on top of the PVM system, which did not require any change in the underlying PVM system. TPVM had a problem of portability, as most of the thread packages available today are not compatible with each other, and due to its CPU dependent feature (context switching depends on the handling of stack and frame pointed by a particular CPU), it is difficult to keep a thread package portable to all platforms, which makes thread-based software difficult to write if it is going to be used on distinct platforms. Recently, Posix has released the Posix.4a standard that has been implement on selected platforms.

As LPVM was developed as an extension to PVM, the user interface of both systems was approximately the same. However, PVM is not MT-safe (Multithreaded safe) [1], therefore LPVM interface was slightly modified to incorporate thread safety, but the changes were minimal. However, the current version of LPVM is designed for a single SMP machine with a shared memory, and so it cannot be used in a distributed environment with disjoint address spaces. The experimental LPVM was implemented on SMP systems because they are stable, multiprocessor (MP) safe, and MP efficient. The results showed improved latency compared to PVM and TPVM, as the threads ran in the same address space unlike TPVM, where the communication was still socket based.

For our work, we implemented a distributed thread based model which was MT-safe, and used threads for sub-task computations ie. the spawned processes export thread entry points describing the actions of the threads executing the routines, sim-

[1]MT-safe means that the data does not get corrupted when multiple threads operate on the same data structure

ilar to TPVM. However, unlike TPVM, which is built on top of PVM and thereby imposes an additional overhead, we implemented the multithreaded algorithm using the PVM interface.

3.3 MPI

The Message Passing Interface (MPI) [8] is the latest development in message passing systems and is reported to be more efficient than PVM. MPI is expected to be faster within a large multiprocessor. It has many more point-to-point and collective communication options than PVM. This can be important if an algorithm is dependent on the existence of special communication option. MPI also has the ability to specify a logical communication topology. The motivation behind developing MPI was that each MPP vendor was creating his own proprietary message-passing API. In this scenario, it was not possible to write a portable parallel application. MPI is intended to be a standard message-passing specification that each MPP vendor would implement on their system. MPI has the following main features,

- A large set of point-to-point communication routines.
- A large set of collective communication routines for communication among groups of processes.
- A communication context that provides support for the design of safe parallel software libraries.
- The ability to specify communication topologies.
- The ability to create derived datatypes that describe messages of non-contiguous data.

A new concept introduced by MPI is the *communicator*. The communicator can be thought as a binding of a communication context to a group of processes. Communication context allows library packages written in message passing systems to protect or mark their messages so that they are not received by the user's code incorrectly. Context is assigned by the operating environment and cannot be made a wild-card by any user program. When a program starts, all tasks are given a "world" communicator and a (static) listing of all the tasks that started together. When a new group (context) is needed, the program makes a synchronizing call to derive the new context from an existing one. This derivation of context becomes a synchronous operation across all the processes that are forming a new communicator. The advantages because of this are that no servers are required to dispense a context as the processes need only decide among themselves on a mutually safe context tag.

Unlike PVM, MPI does not have the concept of a virtual machine. However, MPI provides a higher level of abstraction in terms of message-passing topology. Communication among a group of tasks in MPI can be arranged in a specific logical interconnection topology. The communication thereafter, takes place inside that topology. This is in contrast to PVM in which the programmer is required to manually arrange the tasks into groups with the desired communication organization.

The fault-tolerance capability of MPI is lower than that in PVM, mainly because of the synchronous way that communicators are created and freed in MPI. The earlier version of MPI, MPI-1 did not even have any notification capabilities, like that available in PVM, whereby a task gets notified if the status of the virtual machine changes. The recent version of MPI, has added this capability.

Currently, the University of Tennessee and Oak Ridge National Laboratory are investigating the possibilities of merging PVM and MPI, and the project has been named PVMPI [9]. The idea is to access the virtual machine features of PVM and the message passing features of MPI. The duties that PVMPI is intended to perform are:

to use vendor implementations of MPI that are available of multiprocessors, to allow applications to access PVM's virtual machine fault tolerance and resource control, and to use PVM's network communication transparently for data transfer between different vendor's MPI implementations.

In summary, if an application is intended to be executed on a single MPP, then MPI is expected to give better communication performance. It would also be portable to other vendor's MPP. MPI also has a very rich set of communication functions, and therefore it is favored for applications requiring special communication modes that are unavailable in PVM, such as the non-blocking send. However, MPI lacks interoperability, in the sense that one vendor's MPI cannot communicate to another vendor's MPI. MPI also lacks fault tolerance. On the other hand, PVM is advantageous if the application is designed to run over a networked collection of heterogeneous hosts, because of its concept of a virtual machine. PVM also has resource management and fault tolerance, which makes it attractive for continuously running large applications even if hosts or tasks fail, or if loads change dynamically, which is very common in heterogeneous distributed computing.

For this thesis, the main reason for using PVM was its built-in fault tolerance capabilities, and for performing dynamic load scheduling based on the availability and load on the machines. Secondly, PVM has already been successfully implemented over Fore Systems' ATM API (pvm-atm), at the University of Minnesota, for evaluating the performance of an ATM LAN.

Chapter 4

Distributed Multithreaded Matrix Multiplication

In this chapter, we discuss the implementation details of the matrix multiplication algorithms that we have used for testing. The need for multithreading in such applications is discussed initially, followed by a brief summary of the concept of multithreading. We explain the Scalable Universal Matrix Multiplication Algorithm (SUMMA), which we have used for workload allocation. This algorithm is executed on the master processor and two other algorithms, namely Block Matrix Multiplication (BMM), and Strassen's algorithm with Winograd's variant are used for sub-task computations. Chapter 5 gives the performance of the algorithms on various networking media and compares them to a simple distributed algorithm and a serial algorithm (executed on a SUN SPARC Ultra) in terms of bandwidth and latency improvement.

4.1 Multithreading

Multithreading (MT) is a technique that allows one program to do multiple tasks concurrently. As an example, in case of a Graphical User Interface (GUI), one thread can download images, while a second thread can take care of the I/O, while a third thread can be responsible for doing some background calculations. A thread is a *lightweight process* compared to a Unix Process (*heavyweight process*), and uses the address space of the process in which it is running. Multithreading is a new approach in designing distributed applications where performance is the key aspect. Modern OS platforms like Windows NT, OS/2, and several flavors of Unix provide extensive library and system call support for multithreaded applications. With the ready availability of several thread packages (such as Posix, Java threads), multithreading has set a trend for efficient and easy concurrent computing.

Multithreaded programming offers several benefits over serial procedural programming. Primary amongst them are,

- Performance improvement for multiprocessor architectures.
- Better throughput as one blocked thread does not halt the entire application.
- Avoiding process-to-process communication which is a heavier burden on the operating system compared to thread-to-thread communication.
- Optimum use of system resources, for instance in an SMP machine, a multithreaded application will use the available computational power to its full capacity.
- Ability to use the inherent concurrency of distributed objects.

Computers with more than one CPU offer the potential for enormous application speedups. By making the application multithreaded, different threads can run

on different processors simultaneously with no extra effort from the programmer (ie. a multithreaded application written for a machine with one CPU also works for a machine with multiple CPUs). Most of the workstations available today are multiprocessor machines [20]. These machines, when given serial tasks to run, utilize only one processor and thereby give suboptimal performance, compared to when they are running concurrent tasks, where each thread is executed on a different processor. A schematic diagram is shown in Figure 5.1

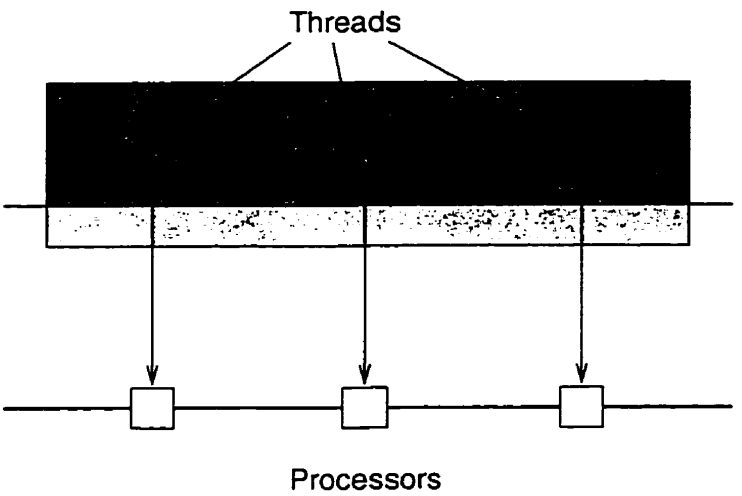


Figure 4.1: Different threads on different processors

For our matrix multiplication application, we have used this idea. The algorithm dynamically generates threads at runtime, according to the number of processors in the machine on which the task is spawned. Apart from that, the algorithm also detects the shape of the input matrices, and makes decisions as to how the threads will operate. A detailed explanation is given in the following sections. This approach is extremely efficient for SMP machines as the CPU idling time will decrease. This not only increases the speed of computation, but also improves the latency of the system.

4.1.1 What is a thread?

A thread is a *lightweight* process. Compared to a regular Unix process (also known as a *heavyweight* process, a thread is a lighter burden on the operating system to create, maintain, and manage, because very little information is associated with a thread. In case of a process, when one process is removed from the processor and another process is activated, a context switch occurs. When a context switch occurs from one process to another, the operating system must keep track of all relevant information needed to restart the process that was running. This information involves the pointer to the executable, the stack, and the memory for statically and dynamically allocated variables. This information is required by the processor when it again takes charge of the process. Therefore a process uses many system resources and causes a large overhead. Threads also have context. When preempted, a context switch must occur between the threads. But unlike a process, a thread does not have its own address space but uses the address space of the process in which it is running. Therefore, the information required to reinstate a thread is much less than that for a process. The information that is required is only a stack, a register set, and a priority that is given to each thread for execution. The text of the thread is contained in the text segment of its process. The data segment of the thread is shared with its process. A thread can read or write to the memory locations of its process and the process has access to the data. The stack of the thread is contained in the stack segment of the process. Threads can create other threads in the process and all the threads running in a process are called peers. All the threads share the resources and memory of the process, but do not own any of them, which make them very easy to handle. All they need is a thread ID, a set of registers that define the state of thread, and a priority. Threads are like a set of tenants living with a common host, which is a process.

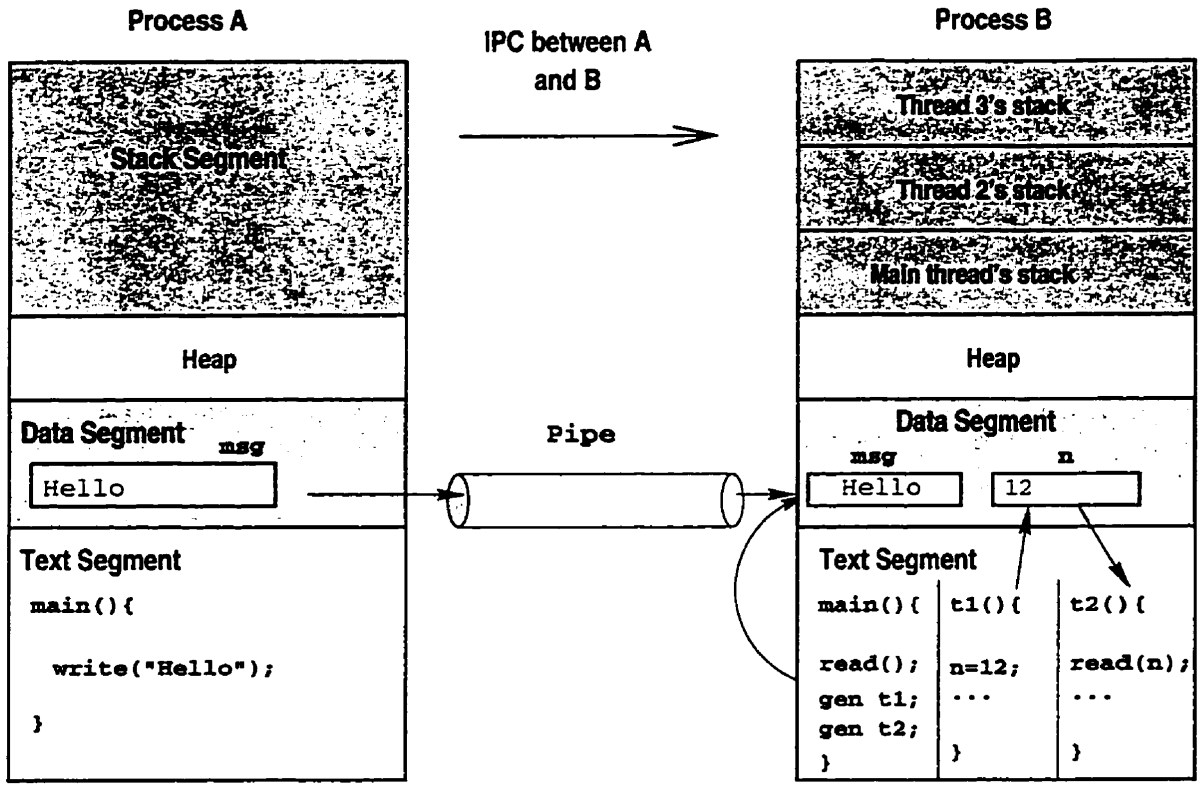


Figure 4.2: Communication between threads and processes

Similarities between Processes and Threads

Processes and threads have a fixed ID, a set of registers that maintain their state, and a priority. They both share resources with parent processes, and are independent entities when they are created. Processes and threads can exchange attributes after creation, and they can create new resources. Finally, they cannot directly access resources and memory of other unrelated processes or threads.

Dissimilarities between Processes and Threads

Threads unlike processes do not have an address space. For communication, parent and child processes use interprocess communication, such as sockets or pipes. In contrast, peer threads communicate by writing and reading data to the process variables. Child processes do not exercise control over other child processes from the same par-

ent process, while threads in a process are considered peers and have control over each other. Lastly, child processes do not have control over the the parent process, but a secondary thread can control the main thread, and thereby the entire process.

Figure 4.2 illustrates communication between processes and threads. As seen, if process A talks to process B, a pipe has to created. On the other hand, two threads t_1 and t_2 talk using common memory space, and thereby do not need any explicit communication medium.

4.2 Distributed Matrix Multiplication

Distributed Matrix Multiplication computation is a standard coarse-granular parallel application that is used for performance measurements of a distributed system. We have implemented a Distributed multithreaded matrix multiplication algorithm based on three algorithms. SUMMA is executed by the master node, and is used for workload division, and Block Matrix Multiplication and Strassen's multiplication algorithm, that are executed on the slaves, are used for sub-task computations. The three algorithms are explained in this section and their combined thread based parallel algorithm is discussed at the end of the section.

4.2.1 SUMMA

The Scalable Universal Matrix Multiplication Algorithm (SUMMA) [25] is a very simple algorithm for matrix multiplication, and we have implemented this algorithm for workload allocation. The master node uses SUMMA to divide the input matrices into blocks and the corresponding blocks of both matrices are then transferred to the slave nodes.

The machines in a PVM LAN are always assumed to form a mesh type architecture. The nodes of the LAN are the tasks that are spawned on the machines and not

the actual machines, so that there may be only three machines in the actual LAN, but these three machines can be running 2 tasks/processes each and thereby forming a 3×2 virtual mesh. The input matrices are divided according the mesh configuration, and SUMMA does it in such a way that the matrices on each node satisfy the row-column requirement for matrix multiplication.

The initial assumption is that the nodes of the parallel machine (formed by machines in the PVM pool) form a $r \times c$ mesh. The total number of nodes, denoted by $p = rc$, are indexed by their row and column index such that the *(i, j)* node will be denoted by $\mathbf{P}_{ij}$. The data decomposition for input matrices A and B, and output matrix C occurs as below;

$$X = \left(\begin{array}{c|c|c} X_{00} & \cdots & X_{0(c-1)} \\ \hline \vdots & & \vdots \\ \hline X_{(r-1)0} & \cdots & X_{(r-1)(c-1)} \end{array}\right)$$

If a given matrix X is of size $m \times n$, where $X \in \{A, B, C\}$, is to be divided on a mesh of $r \times c$ logical nodes, then the portion X_{ij} would be assigned to node $\mathbf{P}_{ij}$. Sub-matrix X_{ij} has dimensions ${m_i}^X \times {n_j}^X$, with $\sum {m_i}^X = m$ and $\sum {n_i}^X = n$.

For doing the multiplication, we require $m^A = m$, $n^A = m^B = k$, and $n^B = n$. If a_{ij}, b_{ij} and c_{ij} denote the *(i, j)* elements of the matrices respectively, then the elements of C are given by

$$c_{ij} = \sum_{l=1}^{k} a_{il} b_{lj}$$

As seen, the rows of C are calculated from the rows of A, and the columns of C are calculated from the columns of B. Therefore, the rows of A and C are assigned to the same row of nodes and columns of B and C are assigned to the same column of nodes. Hence, ${m_i}^C = {m_i}^A$ and ${n_j}^C = {n_j}^B$.

$$C_{ij} = \overbrace{\left(\begin{array}{c|c|c|c} A_{i0} & A_{i1} & \cdots & A_{i(c-1)} \end{array} \right)}^{\tilde{A}_i} \left\} \begin{array}{c} B_{0j} \\ \hline B_{1j} \\ \hline \vdots \\ \hline B_{(r-1)j} \end{array} \right. \tilde{B}^j$$

As seen, $\tilde{A}_i$ is entirely assigned to node row i, while $\tilde{B}^j$ is entirely assigned to node column j. By putting,

$$\tilde{A}_i = \left(\begin{array}{c|c|c|c} \tilde{a}_i^0 & \tilde{a}_i^1 & \cdots & \tilde{a}_i^{k-1} \end{array} \right) \text{ and } \tilde{B}^j = \left(\begin{array}{c} \tilde{b_0}^{jT} \\ \hline \tilde{b_1}^{jT} \\ \hline \vdots \\ \hline \tilde{b_{k-1}}^{jT} \end{array} \right)$$

we see that,

$$C_{ij} = \sum_{l=0}^{k-1} \tilde{a_i}^l \tilde{b_l}^{jT}$$

Hence the matrix-matrix multiplication can be formulated as a sequence of rank-one updates.

SUMMA can be improved by using the level-3 BLAS (Built-in Linear Algebra Subprograms) provided by major vendors of high performance microprocessors. An optimized version of the algorithm can do the matrix multiplication by accumulating several columns of $\tilde{A}_i$ and rows of $\tilde{B}_j$ before updating the local matrix. The advantage gained here is that it reduces the number of messages incurred, thereby reducing communication overhead. We have not implemented this possibility because of the unavailability of BLAS routines. Our main objective is to evaluate the network performance for multithreaded computations, and hence this feature of SUMMA is not addressed.

SUMMA is simpler than other algorithms that use broadcast-multiply-roll algorithm. However SUMMA is more flexible, its memory usage for work arrays is much lower and hence SUMMA was used for implementing the work-load division.

4.2.2 BMM

Block algorithms are very popular parallel algorithms for matrix computations. BMM (Block Matrix Multiplication) [13] algorithm is an efficient matrix multiplication algorithm, where the input matrices are divided into sub-blocks and then individual sub-block (or sub-matrices) from each input matrix are used for calculating the corresponding sub-block of the resultant matrix.

If two input matrices A and B (assuming square matrices with dimensions $n \times n$), are to be multiplied to form the resultant $n \times n$ matrix C, then the multiplication process can be devised in the following manner. Assuming that $n = Nl$, where N and l are positive integers, and where N is the number of sub-blocks of the matrices or dimension $l \times l$, such that,

$$C_{\alpha\beta} = \sum_{\gamma=1}^{N} A_{\alpha\gamma} B_{\gamma\beta}$$

where $\alpha = 1 : N$, and $\beta = 1 : N$. [1] The algorithm is shown on the next page.

[1] Block matrices are designated by the colon notation, whereby if A is an m $\times$ n matrix, and if 1 $\leq i_1 \leq i_2 \leq$ m, and 1 $\leq j_1 \leq j_2 \leq$ n, then $A(i_1 : i_2, j_1 : j_2)$ is the sub-matrix obtained by extracting rows i_1 through i_2 and columns j_1 through j_2.

for $\alpha = 1{:}N$

$i = (\alpha - 1)l + 1 : \alpha l$

for $\beta = 1 : N$

$j = (\beta - 1)l + 1 : \beta l$

for $\gamma = 1 : N$

$k = (\gamma - 1)l + 1 : \gamma l$

$C(i, j) = A(i, k)B(k, j)$

end

end

end

If $l = 1$, then $\alpha \equiv i, \beta \equiv j$, and $\gamma \equiv k$ and we revert to the standard inner dot-product matrix multiplication algorithm. For parallelizing this algorithm by forking multiple threads to compute individual sub-matrices, we use the following criteria,

1. **Number of Processors:** On a SMP machine, the most optimum performance is obtained if all the CPU's available are used for doing computational work concurrently. Our algorithm makes sure of this, and forks as many threads as there are number of processors in a machine. For a Unix workstation, this can be accomplished by calling the system configuration routines and checking the number of processors online, and for Windows NT workstations, a method called *GetSystemInfo()* which is included in the system routines, is called which returns the number of processors. The algorithm takes this value and dynamically forks equivalent number of threads.

2. **Shape of the input sub-matrices:** The shape of the input sub-matrices also play and important role in deciding on how to fork up threads and how to operate them on the matrices. If the two matrices are primarily rectangular, then only one of the two matrices is divided into sub-matrices. If both the

matrices are primarily square, then both the matrices are divided into sub-matrices.

Out of the two matrices A and B, A is always divided into sub-matrices regardless of the shape of A, if we are doing $A \times B$. By the first criteria, the number of threads to be forked is decided. The total number of rows of A are divided by the number of threads to be forked, and each resultant block is computed by one thread. If the number of rows in A are such that they cannot be divided equally amongst the threads, then the thread which becomes free first after doing its share of work takes up the computation of the last residual part that could not be allocated to any thread. In case if the matrices are primarily rectangular, then matrix B is not divided, and each thread multiplies one block of A by the entire B matrix. In this case, if the matrix A was divided into four parts, then ideally the algorithm will work four times faster then a simple multiplication, as there will be four concurrent multiplications going on at a time. On the other hand, if the matrices are primarily square in shape, then A is again divided in the same way, and B is also divided in the similarly, however, it is divided column-wise. In other words, the total number of columns of B are divided by the number of threads to be generated. Therefore a block of A is multiplied by the corresponding block in B by one thread. This type of dual partitioning will reduce the amount of work on the threads, and thereby increase the latency. Shown below are the sub-blocks of the two matrices.

$$C = AB = \left(\begin{array}{c} 1 \\ \hline 2 \\ \hline \vdots \\ \hline n \end{array}\right) \left(\begin{array}{c|c|c|c} & & & \\ 1 & 2 & \cdots & n \\ & & & \end{array}\right)$$

When performing the matrix multiplication, the main thread creates threads for each CPU. The main thread also sets up a counter of work available to do, and then uses a condition variable to signal the threads to start the computation. Each

individual thread acquires the mutex lock, and after doing its work, updates the counter, and releases the mutex lock. Hence the second thread can start from the point where the first thread stopped.

Whenever a condition arises when one thread depends on the data that is computed by some other thread, a condition variable (such as semaphores) is used for signaling purposes. Strassen's algorithm extensively uses data transfer between the worker threads, as explained in the next subsection, and these condition variables are used for that purpose.

4.2.3 Strassen's Algorithm

Strassen's algorithm is a fast recursive matrix multiplication algorithm. The advantage of this algorithm is that it uses fewer multiplications compared to any standard matrix multiplication algorithm. For multiplying two $m \times m$ matrices, m^3 scalar multiplications and m^3 - m^2 scalar additions are required, which results in a total arithmetic count of $2m^3$ - m^2 and a general algorithm complexity of $\Theta(m^3)$ [17]. This means that a 2×2 matrix will require 8 multiplications and 4 additions. Strassen's algorithm, proposed in 1969, multiplies two 2×2 matrices using 7 multiplications and 18 addition/subtractions. It has a general complexity of $\Theta(m^{2.807})$. Apart from that, the algorithm does not depend on the commutativity of the component multiplications, so it can be used for block matrices and used recursively. Thus, Strassen's algorithm reduces the number of multiplications, and increases the number of additions. However, in a large matrix multiplication problem, addition can be accomplished in parallel with more ease than multiplication. The total number of operations for multiplying 2×2 matrices whose elements are $m/2 \times m/2$ blocks will be,

$$7(2(m/2)^3 - (m/2)^2) + 18(m/2)^2 = (7/4)m^3 + (11/4)m^2$$

The ratio of this operation count to that of the standard algorithm, therefore, is,

$$\frac{\frac{7}{4}m^3 + \frac{11}{4}m^2}{2m^3 - m^2}$$

If m gets large, the operation count approaches 7/8, which means that for large matrices, one recursion of Strassen's algorithm gives an improvement of 12.5% over regular matrix multiplication. Therefore, if Strassen's algorithm is applied recursively to large matrices, the performance improvement is significant.

For this thesis, we have implemented Winograd variant of Strassen's algorithm (which is an improved version of Strassen's algorithm) that uses 7 multiplications and 15 additions/subtractions. The algorithm partitions the input matrices A and B into 2×2 blocks and computes C, as shown below.

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

The computation process in one recursion consists of four stages. Stages (1) and (2), as shown on the next page, which compute the S and T matrices are shown below. S and T are temporary matrices.

$$S_1 = A_{21} + A_{22}, \qquad T_1 = B_{12} - B_{11},$$
$$S_2 = S_1 + A_{11}, \qquad T_2 = B_{22} - T_1,$$
$$S_3 = A_{11} - A_{21}, \qquad T_3 = B_{22} - B_{12},$$
$$S_4 = A_{12} - S_2, \qquad T_4 = B_{21} - T_2.$$

Similarly, stages (3) and (4) compute the P and U matrices which are also temporary matrices. The two stages are shown on the next page.

$$P_1 = A_{11}B_{11}, \qquad U_1 = P_1 + P_2,$$
$$P_2 = A_{12}B_{21}, \qquad U_2 = P_1 + P_4,$$
$$P_3 = S_1T_1, \qquad U_3 = U_2 - P_5,$$
$$P_4 = S_2T_2, \qquad U_4 = U_3 - P_7,$$
$$P_5 = S_3T_3, \qquad U_5 = U_3 + P_3,$$
$$P_6 = S_4B_{22}, \qquad U_6 = U_2 + P_3,$$
$$P_7 = A_{22}T_4, \qquad U_7 = U_6 + P_6.$$

It can be seen that $C_{11} = U_1$, $C_{12} = U_7$, $C_{21} = U_4$, and $C_{22} = U_5$.

Strassen's algorithm can also be applied to rectangular matrices. As mentioned earlier, the recursion should be stopped when the component matrices reach a cutoff minimum size. At this point, the standard algorithm should be used, as it is more efficient for small matrices. It has been proved that for multiplying two matrices of dimensions $m \times n$, and $n \times k$, the cutoff dimensions are obtained by solving the following inequality,

$$1 \leq 4(1/n + 1/m + 1/k)$$

Therefore for a square matrix ($m = n = k$), the dimension should be less than or equal to 12. In other words, Strassen's algorithm should be used recursively until the component matrices are reduced to a size less than or equal to 12×12, after which the standard algorithm should be used for the optimum performance.

4.2.4 Our Algorithm

Our algorithm takes in matrices of any shape, size, and dimensions. It recursively uses the Strassen's algorithm until the cutoff dimensions (in our case, used 12 as the cutoff dimension) are reached, after which it switches to a standard block multiplication algorithm. Dynamic Peeling (as explained in the next subsection) is used for matrices that are rectangular, and have odd dimensions. In cases where the input matrices

are primarily rectangular, the algorithm uses BMM directly, and also decides on how to divide the matrices into sub-blocks. Finally, the algorithm detects the machine architecture on which the code is executing, as well as the number of processors present and forks the equivalent number of threads to perform the calculations. The operating system is also detected which calls a specific thread library (eg. for Unix machines, either Solaris or Posix thread packages are called, while for PCs running Windows NT, the Windows NT thread libraries are called.)

For initial work-load allocation, we use SUMMA, which uses the mesh topology, and divides up the rows and columns of the matrices accordingly. After this, the *pvm_pack()* routine is called, which packs the matrix data to be sent to the slave nodes. *pvm_send()* is called to transmit this data to the remote machine (slave), where *pvm_recv* and *pvm_upack* routines are called to receive, and unpack the data. At this point, the algorithm also detects the machine architecture of the slave, and spawns an equivalent number of threads. The shape and size of the matrices are also used to decide as to which algorithm should be used for sub-matrix computations ie. individual matrix multiplication at the nodes. If the matrices are primarily square (where the number of rows and columns are approximately the same), then we use Strassen's algorithm, and if they are primarily rectangular (where the number of rows or columns is greater than the other by more than twice), then we use BMM.

For matrices with an odd number dimension, either dynamic padding, or dynamic peeling is used. Dynamic padding involves adding an extra row or columns of zeros to the matrix to make the dimensions even. Dynamic peeling on the other hand deals with odd dimensions by stripping off the extra row and/or column, and adding their contribution to the final result later. For our algorithm, we have implemented the dynamic peeling method for two reasons. Firstly, dynamic padding increases the time of computation, as an extra row and/or column is added to the matrix, and this row/column is also multiplied along with the original matrix, which can cause a

performance overhead. Secondly, dynamic peeling method had not been previously tested through actual implementation, and we wanted to see the advantages of this method. Dynamic peeling is explained briefly below.

Dynamic Peeling

Let A be an $m \times n$ matrix, and B be a $n \times k$ matrix, where m, n, k are odd integers. Then the matrix division takes place as below,

$$A = \left(\begin{array}{c|c} A_{11} & a_{12} \\ \hline a_{21} & a_{22} \end{array}\right) \text{ and B} = \left(\begin{array}{c|c} B_{11} & b_{12} \\ \hline b_{21} & b_{22} \end{array}\right)$$

where A_{11} is a $(m-1) \times (n-1)$ matrix, a_{12} is a $(m-1) \times 1$ matrix, a_{21} is a $1 \times (k-1)$ matrix, and a_{22} is a 1×1 matrix. Similarly, B is also divided in the save way. The product $C = AB$ is computed as,

$$C = \left(\begin{array}{c|c} C_{11} & c_{12} \\ \hline c_{21} & c_{22} \end{array}\right) = \left(\begin{array}{c|c} A_{11}B_{11} + a_{12}b_{21} & A_{11}b_{12} + a_{12}b_{22} \\ \hline a_{21}B_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{array}\right)$$

Here, Strassen's algorithm is used for computing $A_{11}B_{12}$, and the other computations are added later. The Strassen's computation, then again starts recursively, and dynamically peels off rows and columns whenever an odd number is encountered, till the cutoff dimension (determined at runtime) is reached, at which point the standard algorithm is used. After this, the algorithm starts putting the extra row and column computations, and ends up with the final C matrix.

We have also implemented dynamic peeling for the case when only one dimension is odd, or when the odd dimension is greater than the even dimension, or when the odd dimension is smaller than the odd dimension. The peeling in this case involves

stripping off two rows or columns when the even dimension is smaller, and only one row or column if the odd dimension is smaller.

The complete algorithm is given below.

1. The input matrices, A and B are read by the algorithm along with the mesh configuration of the virtual machine (in the form of an $r \times c$ mesh. Based on the mesh configuration, the matrices are divided (A is divided row-wise and B is divided column-wise). The division is such that the column-row condition necessary for matrix multiplication is satisfied at each node.

2. The node which will run the PVM console will act as the master node, and it will spawn processes on the slave nodes. *pvm_spawn()* routine is used for this purpose. This routine will spawn a user defined process on the slave. In our test case, the process spawned will be *Matmult*.

3. PVM's internal runtime libraries will open connection with the specified hosts, and call the *pvm_pk*()* routines to pack the data to be sent to the hosts, which will be sent by the *pvm_mcast()* routine. This routine will spawn processes on the slaves in a round-robin fashion. In case of Ethernet as the underlying network, a socket based TCP/IP connection will be established between the hosts. On the other hand, if the underlying network in ATM, then Fore Systems' ATM API will be used to utilize the AAL 5 protocol, and thereby using Class 5 service. After packing and sending the data to the slaves, the master will wait for the results from the slaves.

4. At the slaves, the *pvm_recv()* routine will receive the data from the master, followed by *pvm_unpk*()* which will unpack the data items. At the slaves, the algorithm will detect two things.

    - Number of Processors: For a Unix machine, the routine *sysconf()* will be used, and for a Windows NT machine, *GetSystemInfo()* will be used. These

routines will determine the number of processors online, and the algorithm will create equivalent number of threads. The priorities of the threads can assigned in such a way so that each thread is running concurrently on each processor (We have not implemented this facility as the the machines on which we tested the algorithm were mostly single processor machines). If the machine has a uni-processor architecture, then by default, four threads will be created (this number can also be changed, although through trial-and-error, we concluded than for a uniprocessor machine, 4 threads gives the best performance).

- Shape of the Sub-matrices: This will determine how the sub-matrices have to be divided. As explained earlier, this will depend on whether the matrices are rectangular or square. In this case, if the number of rows/columns are more than twice the number of columns/rows, then the matrix is said to be rectangular, and only the sub-matrix of A will be divided. Otherwise, both sub-matrices will be divided as explained earlier.

5. Strassen's algorithm will be called recursively for multiplying the sub-matrices. This algorithm will dynamically peel off extra rows and columns should they be odd in number, and bring out the largest square matrix in the remaining part. If the dimensions of this square matrix are above the cutoff, a second recursion will take place where the same procedure will be repeated. As the dimension of the largest square reaches the cutoff, the standard algorithm will be used (a multithreaded block algorithm). After which, the algorithm will add the results of the extra rows and columns that were peeled off during each recursion, starting from the last recursion.

6. After the sub-matrix C has been verified, it calls the *pvm_pk*()*, and *pvm_send()* routines to pack and send the data back to the master and quits.

7. The master receives the data from the slaves, as they finish computation, and then assembles the final matrix C and quits.

This algorithm has been tested on different matrix sizes on two underlying networks namely ATM, and Ethernet. A correctness check is done at each node during the sub-matrix computation, and finally at the master node for the entire matrix. This accomplished by arbitrarily generating the elements of matrix A using the Unix *rand()* facility. The matrix B, is made an Identity matrix (where all the elements except the diagonal are zero, and the diagonal elements are all 1). This results in the matrix C to be exactly same as matrix A.

The Performance results of both the networks are shown and discussed in the next chapter.

Chapter 5

Network Performance - Results

The main objective of this thesis is to evaluate the performance improvement of different computer networks for supporting a thread based computing system and a thread based communication system. The performance is measured in terms of the latency, and the throughput. Two networks were tested as a part of this thesis. In the first case, an Ethernet LAN was constructed and a distributed matrix multiplication application was run on it. This was followed by running a multithreaded version of the distributed application. The performance of both the algorithms was examined. The latency of the network and the workstations, and the throughput of the network was studied. Following this, the same two algorithms were run on an ATM LAN, and the performance of the ATM network, the machines in the LAN, and the algorithms were evaluated. Lastly, we tested the performance improvement by using multithreading in a single machine and compared our algorithm with a serial matrix multiplication algorithm running on a single host

5.1 Test-bed Setups

We used the Ethernet testbed at the ECE Department and at TR*Labs*, Winnipeg, for running our distributed algorithm. For the ATM testbed, we used the existing OC-3 network in the ECE Department. A brief description of the experimental setup is given in the following subsections.

5.1.1 Ethernet

The standard 10 Mbps Ethernet, available in the Electrical and Computer Engineering (ECE) department was used for testing purpose. The network configuration, as shown in Figure 5.1, was used. The workstations in the Ethernet LAN were connected via a 3Com switch, and two hubs. PVM's round robin allocation scheme was used for workload allocation among the hosts. *PvmRouteDirect* was used as the default route between two daemons for faster message passing. Results and analysis of the runtime performance of our algorithm are given in the subsequent sections. The time measured anywhere during the experiments is the average time obtained after 3 or 5 runs of the algorithm.

For the tests, we used only four dedicated machines as we wanted to compare the performance of the Ethernet LAN with ATM LAN, and only these machines had the Fore Systems' ATM Interface cards, although PVM allows any number of machines to be added to the host pool. All the tests were conducted at three different times of the day, namely in the early morning, when the network traffic is not very high, in the late morning, when the network is fully loaded, and in the evening, when again the network is not running at its full capacity. This was done to get the mean test results. For each case, the latency and throughput of the network was calculated. Apart from that, we also tested our algorithm in terms of CPU usage, speed of computation, and how it reduces the latency of the SMP machines. For measuring the network load, we used a simple bandwidth allocation ratio, which was the ratio

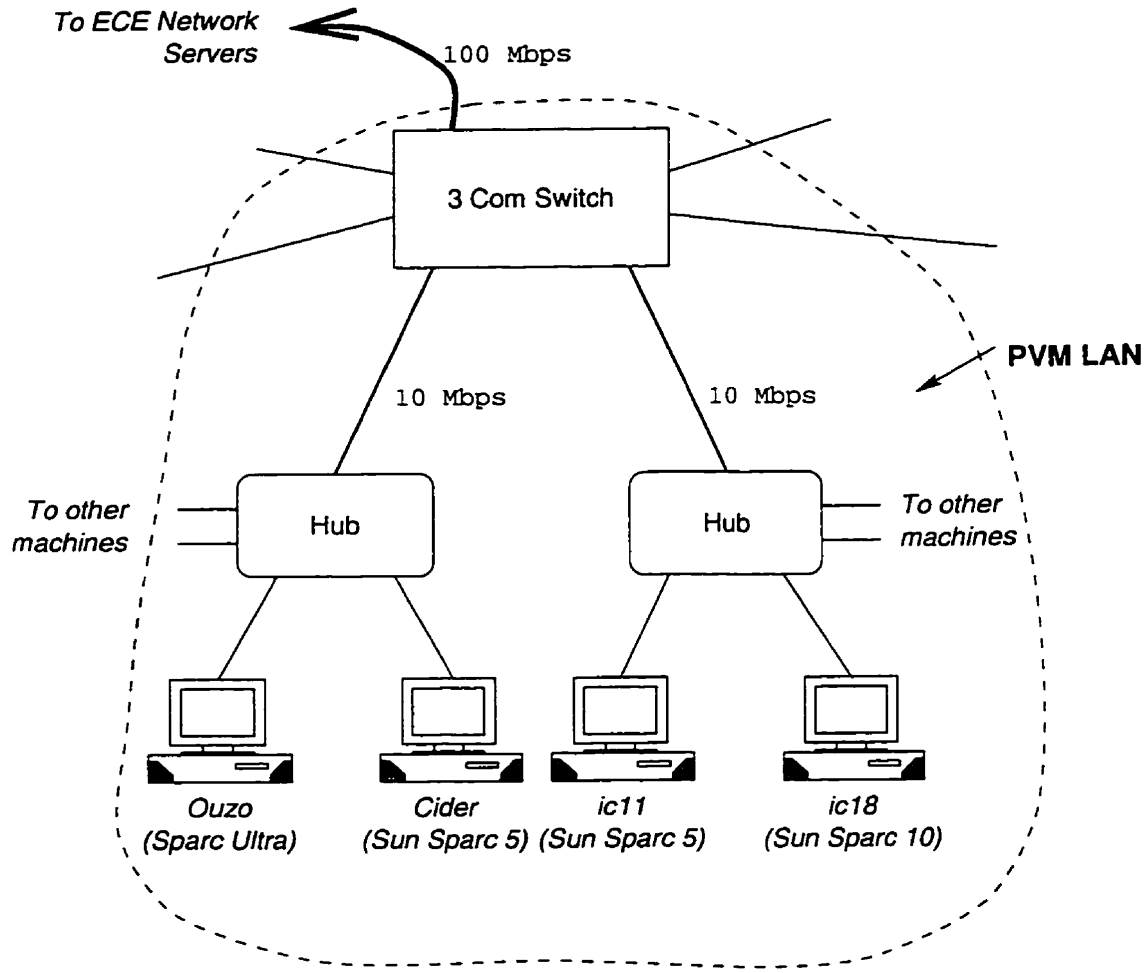


Figure 5.1: Workstations on the Ethernet LAN at the ECE Dept.

of the allocated bandwidth to the achieved bandwidth. Later, we discovered the availability of SNMP on the workstations, and the results extracted after "sniffing" the network yielded approximately the same network loads.

The communication between hosts occurs in two ways: (i) when a master process spawns a slave process on a remote host, the communication occurs between the two PVM daemons in the form of message passing, and (ii) when a thread in one process communicates with a thread in another process. This happens when a thread running in one process requires a data item computed by some other thread in some other process. This is also accomplished by explicit message passing for this research, and is done several times during the execution of the algorithm. Algorithms with finer granularity are supposed to give sub-optimal performance compared to course granular algorithms. The main reason being the usage of the interconnecting network between the hosts should be minimum. By using high-speed, high-bandwidth networks, and using thread spawning and inter-thread communication, as opposed to process spawning, and interprocess communication, this drawback can be reduced. Hence in this thesis, we evaluate the performance of the networks for both these cases, namely single-threaded execution and multithreaded execution. We have tried to determine the maximum achievable throughput, and the end-to-end communication latency for each case, and the results are plotted and detailed discussion of the results is given in the Section 5.7.

5.1.2 ATM

We use the existing ATM testbed in the ECE department, which is an OC-3 (155 Mbps) network operating on a Fore Systems' ASX-200 switch. Four machines are connected in the LAN through the switch, each having Fore Systems' ATM API and using SBA-200 Sbus adapter boards from Fore Systems. The machines, namely Ouzo, Cider, ic18, and ic11, are connected to the ATM network. Their ATM cards

have different IP addresses, such as 204.112.157.*, than the regular ECE domain addresses of 130.179.8.*. Figure 5.2 shows the network layout for the test. PVM was implemented on the API by using the *pvm-atm* package, developed by The Distributed Multimedia Research Centre (DMRC) at the University of Minnesota, which directly uses the Fore Systems' AAL 3/4 and AAL 5 protocols. We tested both the protocols for this thesis.

5.2 Network Performance

We evaluate the network performance in terms of end-to-end communication latency, and maximum achievable throughput. In the following Sections, we describe the tests and the echo programs used for latency measurements.

5.3 End-To-End Communication Latency

The *communication latency* for sending a M-byte message can be estimated as half of the round trip time required for sending and receiving this data from one host to another. We used a standard available echo program for doing this, which is provided with the PVM 3.3.11 package. In the echo program, a client sends a M-byte (M ranges from 20 bytes to 250 Kbytes) message to the server and waits to receive the M byte message back. This client/server interaction iterates N times, and we note the timing for each iteration, and determine the average timing for each value of M. We repeat this procedure four to five times and select the best three observations corresponding to different network loads during different times of the day. The start-up latency is also an important performance parameter for network performance. It is the time required to send extremely short messages. The start-up latency for Ethernet and ATM is used as a performance comparison parameter and is discussed in Section 5.7.

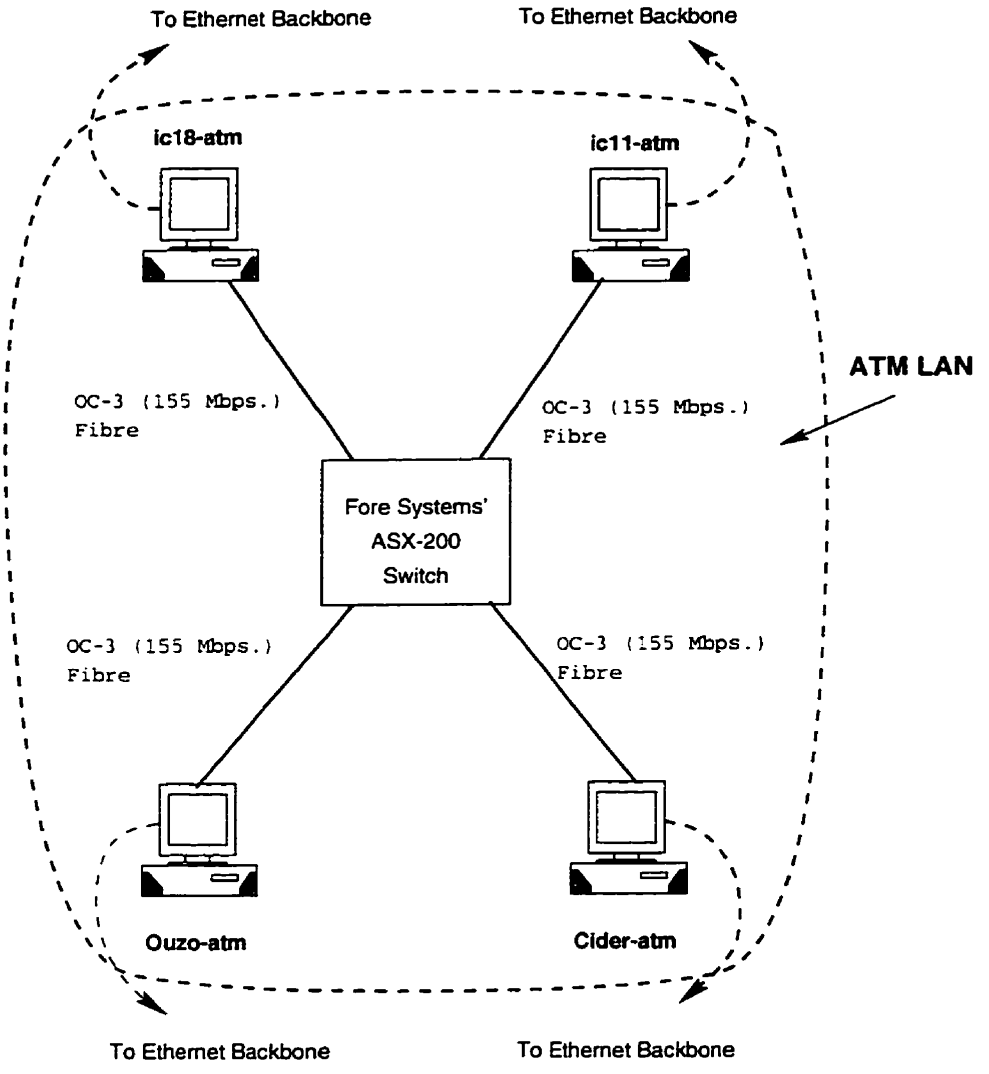


Figure 5.2: Workstations on the ATM LAN at the ECE Dept.

5.3.1 Ethernet

We varied the data size from 20 bytes to 204 kbytes for the echo program, and observed the Round-Trip timing (RTT) for different Network Loads. We took 5 to 7 trial runs and selected the best three runs for network loads that were distributed uniformly, namely 90% load, 50% load, and 20% load. The underlying protocol stack used was the default TCP/IP and UDP/IP. The results are plotted in Table 5.3. A detailed discussion of all the results is given in Section 5.7.

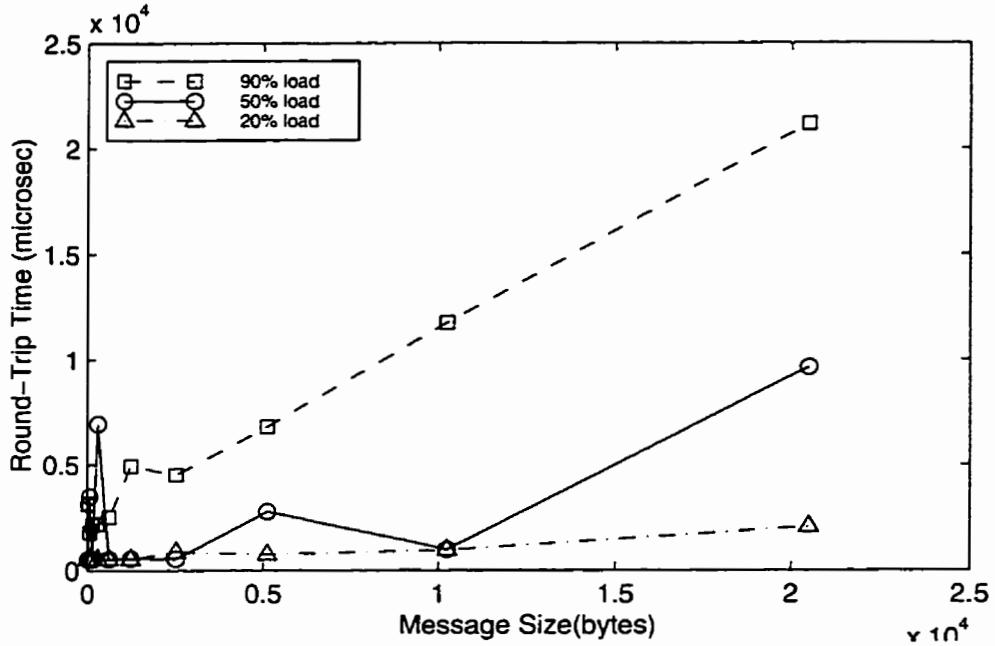


Figure 5.3: Communication Latency for PVM over Ethernet

5.3.2 ATM AAL 3/4

We used ATM AAL 3/4 protocol as well as the ATM AAL 5 protocol as the underlying protocol on our tests on the ATM network. The results obtained for three network loads are plotted in Figure 5.4.

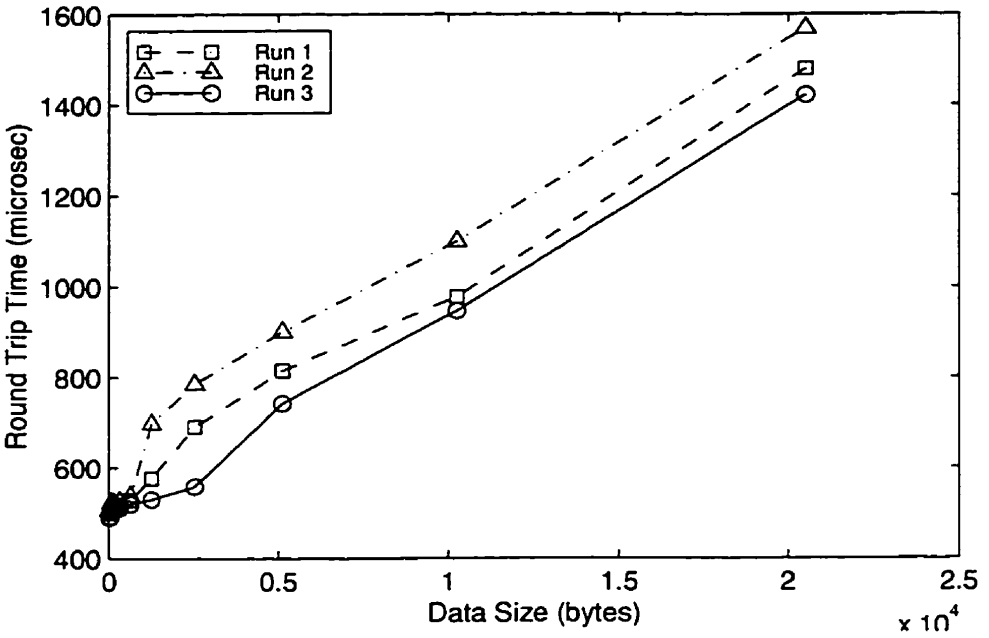


Figure 5.4: Communication Latency for PVM over ATM AAL 3/4

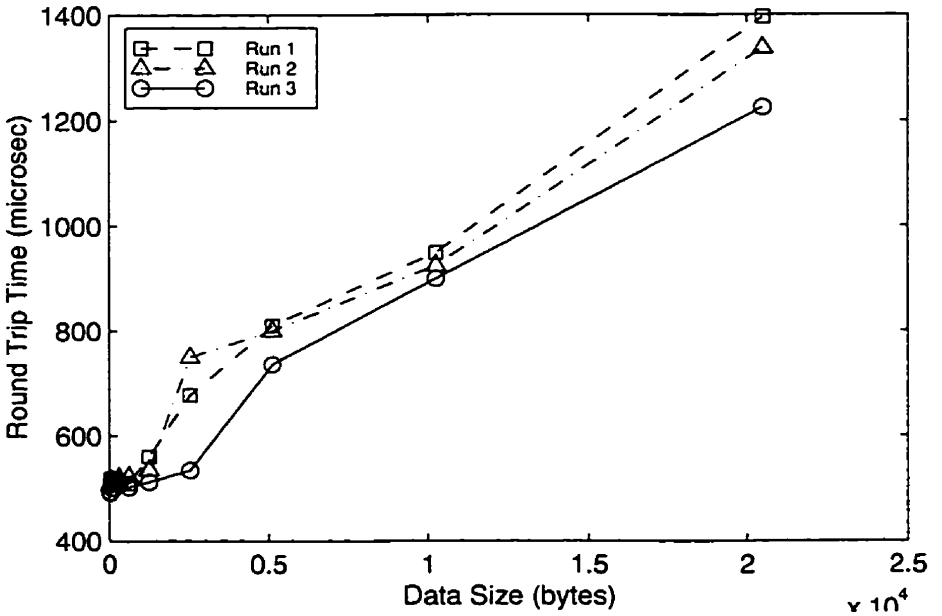


Figure 5.5: Communication Latency for PVM over ATM AAL 5

5.3.3 ATM AAL 5

ATM AAL5 was also tested in the similar fashion for three different runs of the echo program. The results of the runs are shown in Figure 5.5. There is not much difference in the performance of PVM-AAL 3/4 and PVM-AAL 5 as clearly seen in Figure 5.6, however AAL5 provides slightly lower latency. Therefore, ATM AAL 5 has a superior latency characteristic compared to either Ethernet or ATM AAL 3/4.

For comparison purpose, we also plotted the results of the runs for Ethernet (PVM/TCP/UDP) with the ones with ATM (PVM/AAL 3/4 and PVM/AAL 5) together, to see the difference in values. Figure 5.6 shows the combined plots.

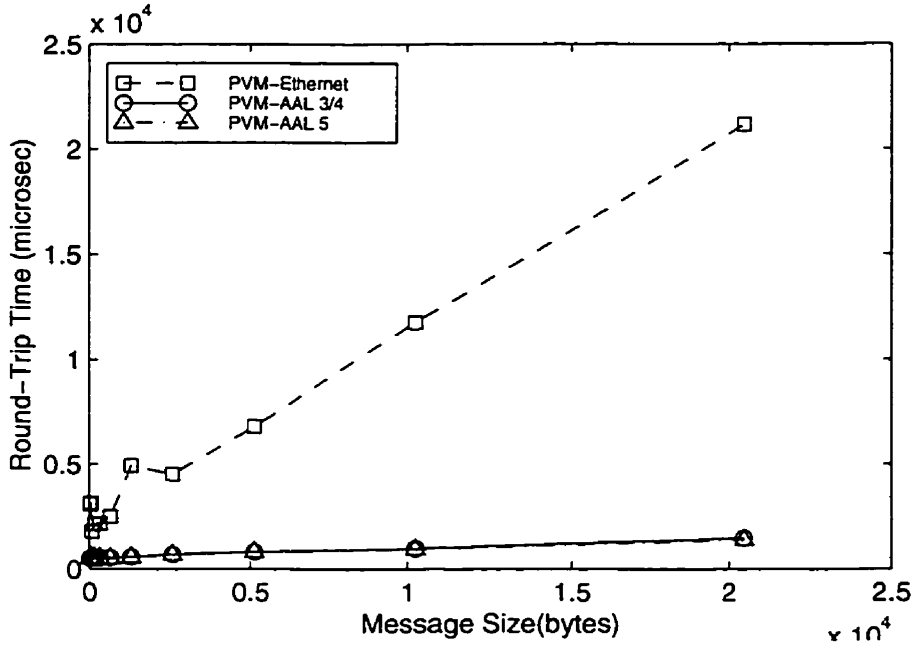


Figure 5.6: Communication Latency for PVM over Ethernet and ATM

5.4 Performance of the Serial Algorithm

We also ran a unthreaded matrix multiplication program on a single workstation, for comparing the speed of computation of a serial multiplication against a multi-

threaded matrix multiplication, and the gain in latency that can be obtained by using concurrent thread based computing. We ran both the algorithms ie. a simple matrix multiplication algorithm and our algorithm (implemented for a single host in this case) on Ouzo (SPARC Ultra), and we saw the computation times of both the algorithms for different matrix sizes, and with two, four and eight spawned threads. As clearly seen, the serial code was the slowest as compared to a threaded code. The maximum number of threads that the machine we used were 4, hence the computation time was minimum for 4 threads. When 8 threads were imposed, the thread management overhead became very high, and thereby the time of computation increased. We have plotted the results in Figure 5.7.

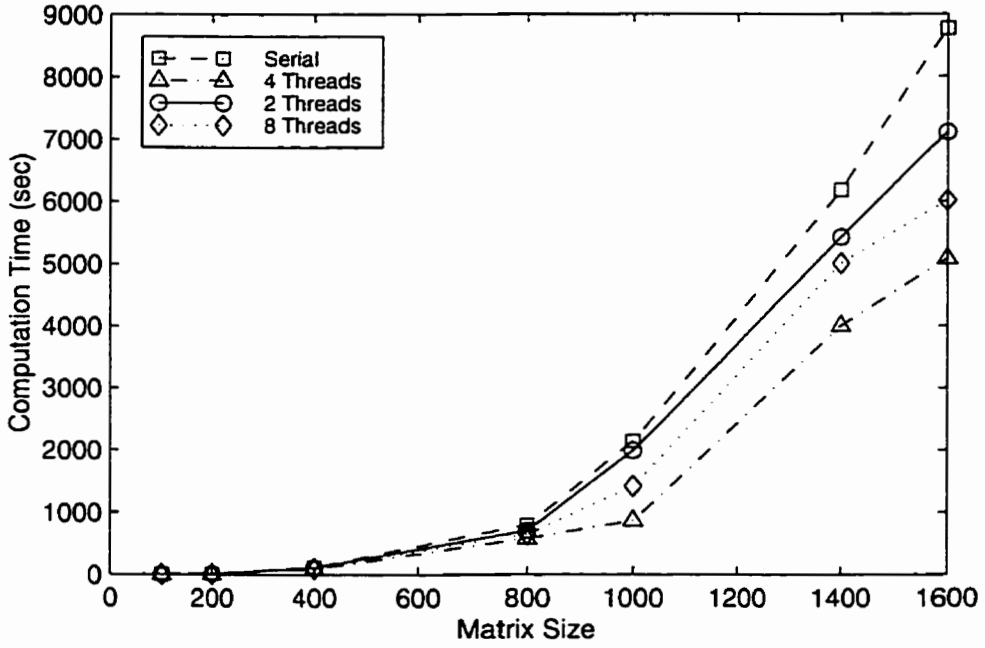


Figure 5.7: Communication Latency for a Sun SPARC Ultra for serial and multithreaded codes

As clearly seen, a multithreaded algorithm performs better than a serial algorithm. The workstation, Ouzo, is a single processor workstation, so the threads are context switched. If however, the workstations has a multiprocessor architecture (like an

SMP), then threads can be concurrently executed on individual processors, and the performance of the algorithm would improve further. By using advanced features like priority scheduling, and functions like *thr_setconcurrency()* in implementing the algorithm, a better performance can be achieved.

Thread Spawning against Process Spawning

For mission critical applications, the time required to spawn a process either locally on the same host or remotely on another host is an important performance parameter. We measured the times required to spawn a process and a thread. The Unix *fork()* utility was used to spawn a process, and the total time for a spawn was measured. For threads, we used the *thr_create()* function, and measured the time. The results are shown in Table 5.1 and plotted Figure 5.8.

It can be clearly seen that the time required to spawn threads is much less than the time required to spawn a process. For our case, we spawned threads only on a single host, however, results from TPVM over PVM [10], and LPVM [26] show that spawning remote threads also requires much less time than spawning remote processes. This adds a lot to improving the latency of the overall system.

Thread and Process Communication Times

We also determined the communication time required for two processes to talk with each other, and two threads to talk with each other. The threads resided on processes that were running on the same host. This was accomplished using the the *pvm_send()* and *pvm_recv()* functions for processes, and by using inter-process condition variables in case of threads. The time for communication is shown in Table 5.2 and plotted in Figure 5.9.

As clearly seen, the time for message passing between threads is significantly less compared to processes. The time for remote threads for communication has also

Number Spawned	*Time (msec)*	
	PVM Processes	*PVM Threads*
1	57.75	1.75
4	110.24	3.78
8	200.27	5.67
16	852.54	20.94

Table 5.1: Spawning Times for a Process and Thread

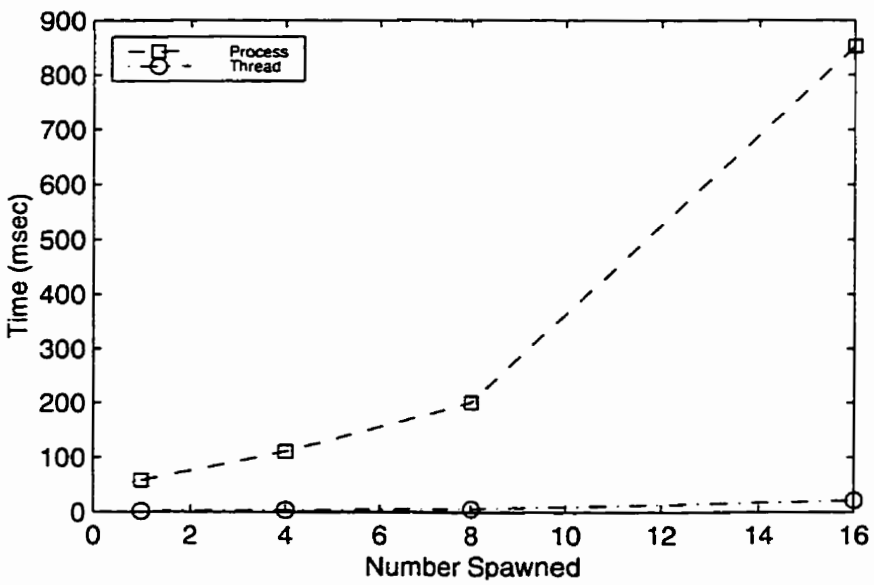


Figure 5.8: Spawning Times for Process and Thread

Message Size	*Communication Time (msec)*				
	Processes				*Threads (local)*
	Ethernet	*AAL 3/4*	*AAL 5*	*Local*	
1 B	5.53	1.10	1.11	1.04	1.01
1 KB	9.24	4.09	4.12	2.92	1.09
32 KB	120.69	72.29	68.67	39.27	15.25
1 MB	3267.32	1477.75	1256.79	897.17	238.44

Table 5.2: Communication Times for Processes and Threads

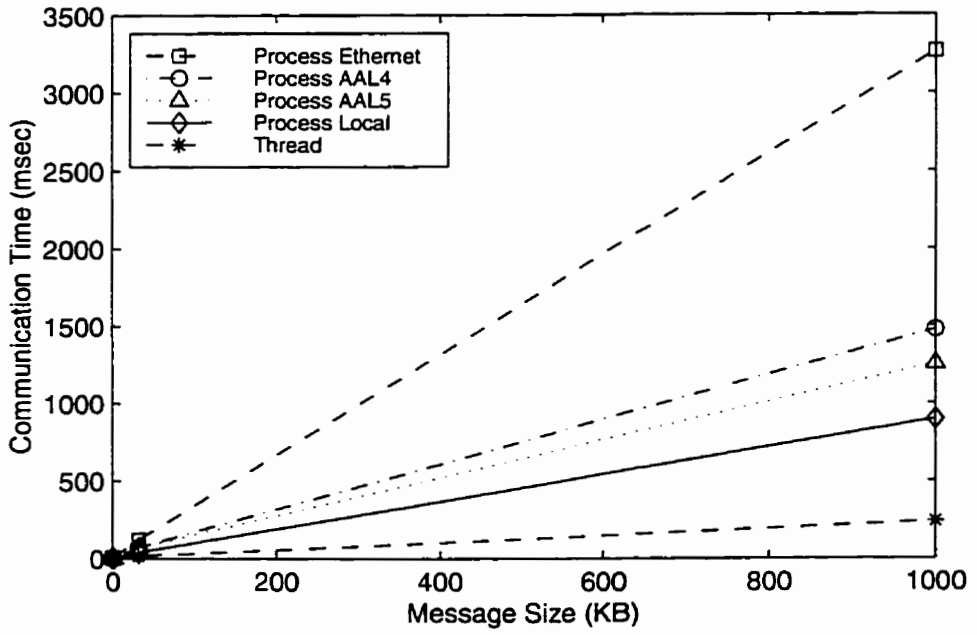


Figure 5.9: Communication Times for Processes and Thread

been proved to be less in TPVM, although usage of a Distributed Shared Memory can improve the performance in case of remote thread communication. The reason being, the threads do not need to use the network every time for accessing a remote data item, if it is stored in the DSM, and if the DSM uses the hosts memory map.

5.5 Maximum Achievable Throughput

The *maximum achievable throughput* (r_{max}) is obtained by transmitting very large messages. It is an important parameter for applications that require large amounts of data transfer between the hosts and is a performance metric for networks. We ran our algorithm for different matrix sizes on a 2×3 mesh (total 6 nodes), and noted the bandwidth. Like before, we conducted our tests at different network loads.

5.5.1 Ethernet

For measuring the communication bandwidth, and thereby the maximum achievable throughput, we measured the round trip time required for sending messages of fixed sizes between two hosts. This was similar to the echo program that we used for measuring latency, wherein a message is sent from one machine to another, and received back repeatedly N number of times. The average communication time for a fixed message size is determined, and this gives the bandwidth that the network can operate for that particular message size. This is repeated for different message sizes, and the bandwidths achieved for each message size is plotted. The maximum bandwidth or the maximum achievable throughput is the peak of the curve obtained. As the time measured is the average time, the maximum throughput is the average maximum achievable throughput.

We have tabularized (Table 5.3) the observations for Ethernet, ATM AAL 3/4 and ATM AAL 5 and have plotted them in Figure 5.10.

Message Size (KB)	*Bandwidth (Mbits(sec)*		
	Ethernet	*ATM AAL 3/4*	*ATM AAL 5*
1	6.41	9.94	11.78
80	8.53	23.39	27.26
320	9.12	27.22	29.17
1280	9.12	27.75	29.26
5120	8.98	27.01	28.85

Table 5.3: Bandwidth for PVM over Ethernet and ATM for different message sizes

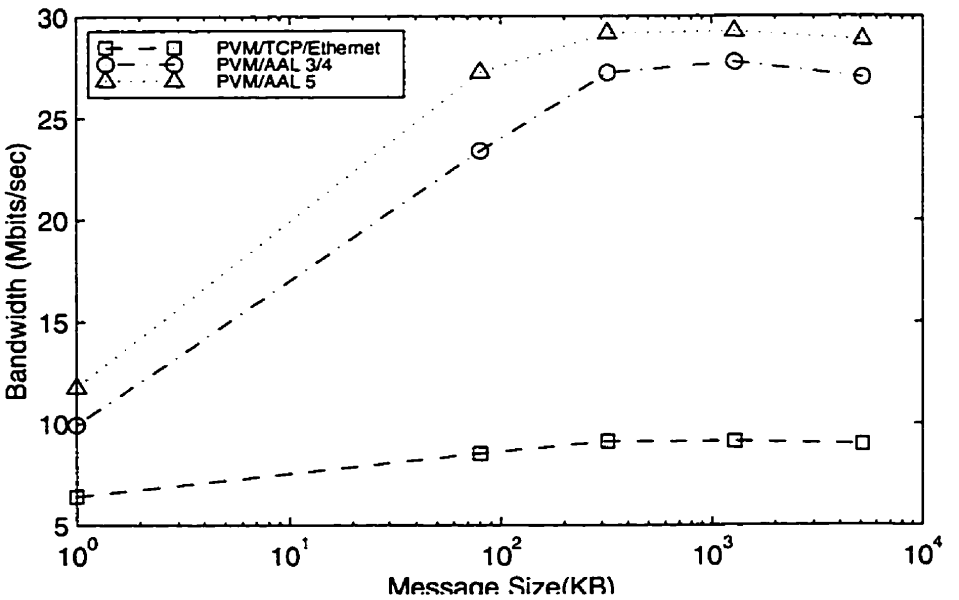


Figure 5.10: Bandwidth and Maximum Throughput for Ethernet and ATM networks

5.5.2 ATM AAL 3/4

The same procedure was repeated for an ATM network with AAL 3/4 protocol. The bandwidths obtained for AAL 5 for different message sizes was higher than the bandwidths for Ethernet. Table 5.3 and Figure 5.10 show the results.

5.5.3 ATM AAL 5

Similar runs were given for the ATM AAL 5 network, and the results are shown in Table 5.3 and are plotted in Figure 5.10. It can be clearly seen that ATM AAL 5 gives the highest throughput among the three networks.

5.6 Overall Real-Time Performance

We also measured the overall real-time performance of a distributed multithreaded matrix multiplication algorithm (our algorithm) against a distributed unthreaded matrix multiplication algorithm. The overall real-time included the total communication time, computation time, packing and unpacking time, and the time for doing I/O. The time was measured using the *high-resolution time* facility in Unix, that gives the real-time in nano-seconds. We did this performance analysis on an Ethernet network, and then on an ATM network with AAL 3/4 and AAL 5 protocols. The results are shown in Table 5.4 and are plotted in Figure 5.11.

5.7 Analysis of Results

The previous sections and subsections give plots and tables of the results that we obtained by running the algorithms mentioned in this thesis in different environments under different conditions.

As we can clearly see, the ATM AAL 5 protocol gives the most superior perfor-

Matrix Size	*Real-Time (sec)*					
	Ethernet		*ATM AAL 3/4*		*ATM AAL 5*	
	Serial	MT	Serial	MT	Serial	MT
100	0.23	0.31	0.18	0.17	0.17	0.15
200	0.95	0.92	0.95	0.89	0.96	0.88
400	7.08	6.23	7.01	6.11	6.99	6.02
800	57.68	50.5	53.47	42.80	50.25	39.28
1600	657.19	519.14	601.93	401.17	532.28	320.67

Table 5.4: Real-time speedup of a Multithreaded (MT) Algorithm over a Serial Algorithm on Ethernet and ATM networks

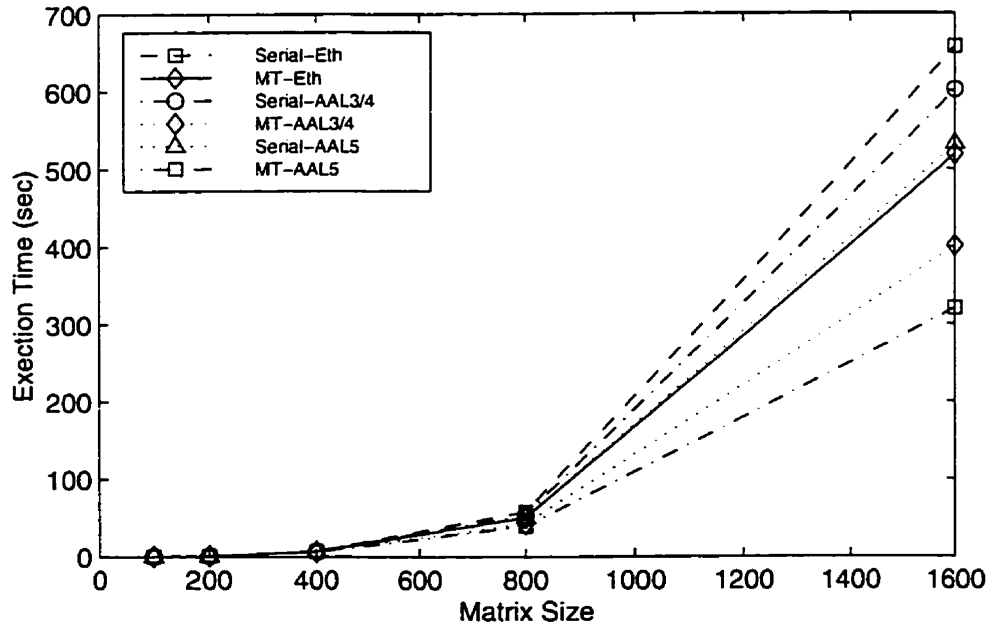


Figure 5.11: Total Execution Time for a MT and Serial Algorithm on Ethernet and ATM

PVM Environment	t_0 μsec
PVM/TCP/Ethernet	1595
PVM-ATM on AAL 3/4	1910.3
PVM-ATM on AAL 5	1915.4

Table 5.5: Start-up Latency for PVM Environments

mance as far as latency is concerned. The AAL 3/4 protocol is highly efficient but gives a slightly higher latency than ATM, while Ethernet gives the highest latency amongst the three. However, the start-up latency t_0 of Ethernet is the minimum followed by ATM AAL 3/4 and AAL 5 respectively as seen in Table 5.5 The t_0 is measured by a 4 byte message from one host to another and calculating the round trip time required.

Start-up latency is half of the round trip time required for sending a small message from one host to another and then receiving it back. This is a performance metric for extremely short messages. The overhead in terms of latency for the ATM network can be due to two reasons namely,

- The device driver for ATM is considered to be slower than the one for Ethernet, and this can be costly when sending short messages. The firmware for Ethernet has been optimized for better communication latency [7].

- There is also an overhead in case of ATM for preparing packages for transmission and receiving them off the network, or in other words the ATM incurs a *package processing overhead.*

When multithreading is used for computing, the latency of the workstation decreases by nearly half the value. As seen in Figures 5.7 and 5.11, using multiple threads for computing makes the program faster, and thereby decreasing the latency

of the machine as well as reducing the overall *real time* for the execution of the algorithm. By using more than one thread on a workstation, the number of *virtual slaves* are increased, and thereby the algorithm is executed concurrently. However as noted in Figure 5.7, when the number of threads are changed from 1 to 2, the time for computation decreases. If the number of threads are further increased to 4, then the algorithm still computes faster. If we increase the number of threads to 8, then as seen, the speed of computation decreases. The main reason is that by increasing the number of threads beyond a certain limit increases the thread management overhead on the operating system. The machines that we used for our experimentation were all uniprocessor machines that gave an optimum performance for 4 threads (and therefore we have used the default number of threads as 4 in our implementation Strassen's algorithm), but if the number of processors are more (as in an SMP), then by increasing the number of concurrent threads, the speed of execution can increase by a large magnitude [16].

We also observed that the spawning of a process is more expensive than spawning a of a thread. For this thesis, we implemented only an intra-host thread spawn program (that spawns threads only within a single host), but experimental results show that even remote thread spawning (using a Thread-server, and remote memory) is less expensive than spawning a process [10]. This, along with the fact that high-bandwidth networks such as ATM can support very frequent communications between hosts, suggest that thread based distributed applications can be extremely performance efficient.

Lastly, we discuss the maximum achievable throughput (r_{max}). Table 5.6 shows the maximum available throughput for Ethernet and ATM networks.

As clearly seen, ATM AAL 5 gives the maximum throughput of 29.26 Mbits/sec indicating its high-bandwidth characteristic, while Ethernet provides a maximum throughput of 9.12 Mbits/sec. This is a very important difference for large computa-

PVM Environment	r_{max} *Mbits/sec*
PVM/TCP/Ethernet	9.12
PVM-ATM on AAL 3/4	27.75
PVM-ATM on AAL 5	29.26

Table 5.6: Maximum Communication Throughput for Ethernet and ATM networks

tionally intensive problems like complex simulations, number crunching, etc. where large amounts of data transfer is required between hosts.

Chapter 6

Conclusions and Future Research Directions

Until now, Distributed Network Computing was used mostly for those applications that require a minimum use of the interconnecting network between the participating hosts (ie., Distributed Network Computing was suitable for course-granular problems.) Due to the low bandwidth and high latency characteristics of existing networks like Ethernet, interprocessor communication across a network was avoided. However, with the recent developments in the area of high-speed networks, and with the emergence of highly efficient and fast networks like ATM, Gigabit Ethernet, etc., we are seeing a whole new horizon for large-scale computing across a network with hosts separated by large geographical distances, such as on the Internet. In this thesis, we explored the possibility of solving problems with extensive inter-processor communication using underlying high-speed networks and thread based communication instead of task based communication. The results are positive and encouraging.

Our main aim was to improve the real-time performance of a distributed computing system. We used multithreading for this purpose as thread management is much

easier and simpler compared to process management, thereby reducing the overhead on the operating system and the CPU. We implemented a thread based computation model and a intra host communication model. We observed a marked improvement in the latency of the workstations due to the thread based computing model. Our second aim was to show that high-speed, high-bandwidth network technologies like ATM can be used without any concerns about its maximum achievable throughput for applications/computations that use extensive message passing between tasks. Our implementation of the Strassen's algorithm proved this premise. Hence an ATM LAN can be used for large simulations with fine-granularity and give excellent performance over other regular networks like Ethernet or Fiber Channel. The main reason ATM is very efficient in large scale data transfers is that it is not a shared medium (ie. the bandwidth allocated to any segment is not shared). Therefore, if desired, an important task can be executed without any obstructions from other network traffic. The second reason is the inbuilt QoS in ATM which is a very attractive feature for mission critical applications where extremely accurate results are desired. Hence, multithreaded programming implemented on an ATM backbone can be a very effective combination for Distributed Scientific Computing.

6.1 Ideas for Future Research

Our work can be extended in several ways. Some of the possible directions that we envision are listed below.

- We have used the PVM implementation on ATM (pvm-atm) for process management and synchronization on ATM. Further improvement in the results can be achieved if the application is developed using Fore Systems' ATM API directly. This can give better performance because the protocol overhead of pvm-atm (as it is implemented on PVM 3.3.2) is eliminated.

- During our workload allocation, we export thread entry points to the slaves from the master, when the master spawns a process. Therefore, threads are used for subtask computations on the slaves. This was shown to have several benefits over process based subtask computations. However, additional performance improvement can be obtained if the master can spawn threads on the slaves directly instead of spawning processes. This idea has been proposed in the LPVM standards [26], but LPVM has still not been developed for a distributed environment, and is used only for SMP machines. TPVM on the other hand *does* utilize indirect thread spawning by calling the *tpvm_spawn()* routine. However, this routine in turn calls the *pvm_spawn()* routine, which in reality spawns processes on the slaves, although it appears to the user that it spawns threads on the slaves. The TPVM implementation on ATM can prove to be significant work, if TPVM's *remote memory* model (*remote memory* model maps one memory space into an another disjoint memory space) is developed further.

  For our implementation, we used used explicit message passing for inter-thread communications. For this, if there is a Distributed Shared Memory (DSM), which can be a common memory for all the machines participating in the PVM LAN, then the threads can communicate by writing and reading variables in the DSM, and synchronize with each other using mutexes, semaphores, condition variables, etc. This can significantly improve the communication overhead on the network. The reason being, threads do not have to use the network each time for accessing a data object, as the DSM may be using the local hosts memory as a part of the Global Shared Memory.

- We tested our algorithm on a network that consists only of homogeneous hosts, ie. Sun SPARC stations running Solaris. In an actual corporate environment, a network/LAN usually consists of heterogeneous hosts with completely different platforms, and running distinct operating systems. Hence this algorithm can

also be tested by introducing other hosts like PCs (running Windows 95 or NT), or a Macintosh machine, and the performance of the network should be studied for servicing a hybrid network. As far as we can envision, the performance will be slightly inferior compared to a homogeneous network because of the added overheads of data-type matching, address mapping, etc.

- Lastly, this algorithm can also be tested on the High Performance Network (HPCNet), which is a high-speed ATM backbone connecting several Supercomputers across Canada. Testing this sort of network topology would give a more broader view of the problems that Network Engineers are facing today, and possibly come up with solutions to solve them.

Bibliography

[1] Anderson, T. Culler, D. Patterson, D. and the NOW Team. The Case for Network of Workstations. *IEEE Micro.*, Volume 15, Number 1:54–64, February 1995.

[2] Bailey, D. H. Extra high speed matrix multiplication on the CRAY-2. *SIAM Journal of Science and Statistical Computing*, Volume 9:603–607, 1988.

[3] Bjørstad, P. Manne, F. Sørevik, T. and Vajteršic, M. Efficient Matrix Multiplication on SIMD Computers. *Siam Journal of Matrix Analysis and Applications*, Volume 13, Number 1:386–401, January 1992.

[4] Black, D. *Managing Switched Local Area Networks - A Practical Guide.* Addison-Wesley, 1998.

[5] Boudec, J. The Asynchronous Transfer Mode: A Tutorial. *Computer Networks and ISDN Systems*, Volume 24:279–309, 1992.

[6] Chang, S. L. Du, H. C. Hsieh, J. Lin, M. and Tsang, R. Enhanced PVM Communications over a High-Speed Local Area Network. In *Proceedings of First International Workshop on High-Speed Network Computing*, Santa Barbara, California, April 1995.

[7] Clark, D. Jacobson, V. Romkey, J. and Salwen, H. An Analysis of TCP Processing Overhead. In *IEEE Communications Magazine*, pages 38–44, June 1989.

[8] Dongarra, J. Otto, S. Snir, M. and Walker, D. An Introduction to the MPI Standard. In *University of Tennessee Technical Report CS-95-274*, January 1995.

[9] Fagg, G. and Dongarra, J. PVMPI: An Integration of PVM and MPI Systems. *Calculateurs Paralleles*, Volume 8, Number 2:151–166, 1996.

[10] Ferrari, A. and Sunderam, V. S. TPVM: Distributed Concurrent Computing with Lightweight Processes. In *Proceedings of the 4th High-Performance Distributed Computing Symposium*, pages 211–218, Washington, DC, August 1995.

[11] Geist, A. Beguelin, A. Dongarra, J. Jiang, W. Manchek, R. and Sunderam, V. *PVM:Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[12] Golub, G. and Ortega, J. *Scientific Computing: An Introduction with Parallel Computing.* Academic Press, San Diego, CA, 1993.

[13] Golub, G. and Van Loan C. *Matrix Computations.* John Hopkins University Press, 3rd edition, 1996.

[14] Hsieh, J. Du, H. C. Troullier, N. J. and Lin, M. Enhanced PVM Communications over HIPPI Networks. In *Proceedings of The Second International Workshop on High-Speed Network Computing (HiNet '96)*, pages 20–32, Honolulu, April 1996.

[15] Huang, C. Huang, Y. and McKinley, P. K. A Thread-Based Interface for Collective Communication on ATM Networks. In *Proceedings of the 1995 International Conference on Distributed Computing Systems*, pages 254–261, Vancouver, British Columbia, May 1995.

[16] Hughes, C. and Hughes, T. *Object-Oriented Multithreading Using C++.* John Wiley & Sons, Inc., 1997.

[17] Huss-Lederman, S. Jacobson, E. Johnson, J. Tsao, A. and Turnbull, T. Implementation of Strassens's Algorithm for Matrix Multiplication available at http://www.bib.informatik.th-darmstadt.de/SC96/JACOBSON. In *Proceedings of the International Conference on High Performance Computing and Communications*, Pittsburg, PA, November 1996.

[18] Kleinman, S. Smaalders, B. Stein, D. and Shah, D. Writing Multithreaded Code in Solaris. In *Solaris Threads White Papers*, Mountain View, California, 1993. SunSoft Inc. Press.

[19] Lea, C. What should be the goal for ATM? *IEEE Network*, Volume 6, Number 5:60–66, September 1992.

[20] Lewis, B. and Berg, D. *Multithreaded Programming with Pthreads.* Sun Microsystems Press, 1997.

[21] Lin, M. Hsieh, J. Du, H. C. Thomas, J.P. and MacDonald, J. A. Distributed Network Computing over Local ATM Networks. *IEEE Journal on Selected Areas in Communications, Special Issue of ATM LANs: Implementations and Experiences with an Emerging Technology*, Volume 13, Number 4:733–748, May 1995.

[22] Roberts, E. Gigabit Ethernet: Fat Pipe or Pipe Bomb? In *Data Communications*, pages 30–42, May 1997.

[23] Rodrigues, S. Anderson, T. and Culler, D. High-Performance Local Area Communication with Fast Sockets. In *Proceedings of USENIX'97*, Anaheim, California, January 1997.
Available at http://now.cs.berkeley.edu/Fastcomm/usenix.ps.

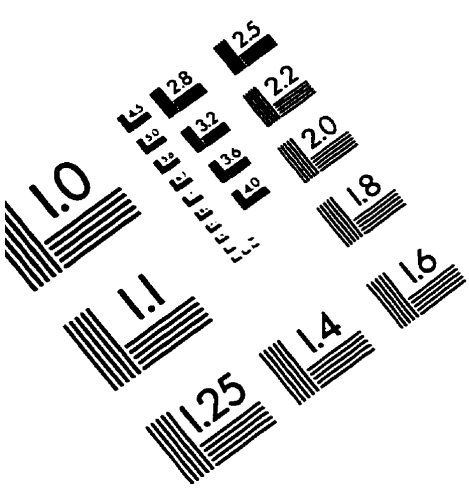
[24] Tanenbaum, A. *Computer Networks.* Prentice Hall, 3rd edition, 1996.

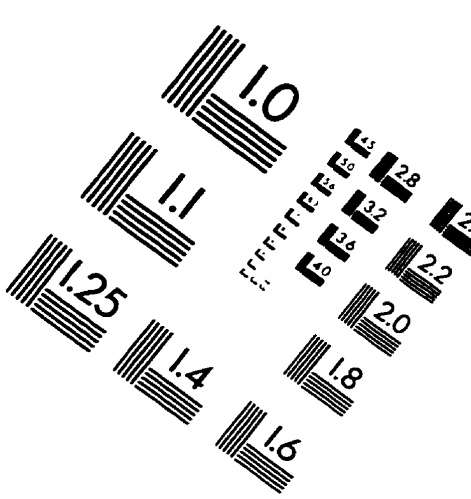
[25] Van De Geijn, R. and Watts, J. SUMMA:Scalable Universal Matrix Multiplication Algorithm. *Concurrency: Practice and Experience*, Volume 9, Number 4:255–274, April 1997.

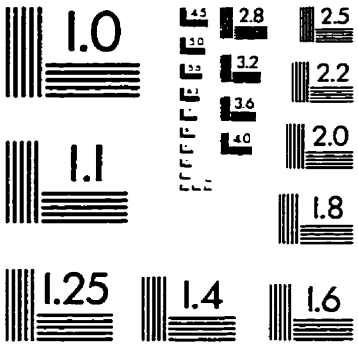
[26] Zhou, H. and Geist, A. LPVM: A Step Towards Multithreaded PVM. *Technical Report, Oak Ridge National Laboratory*, July 1995. Available at http://www.epm.ornl.gov/~zhou/lpvm.ps.

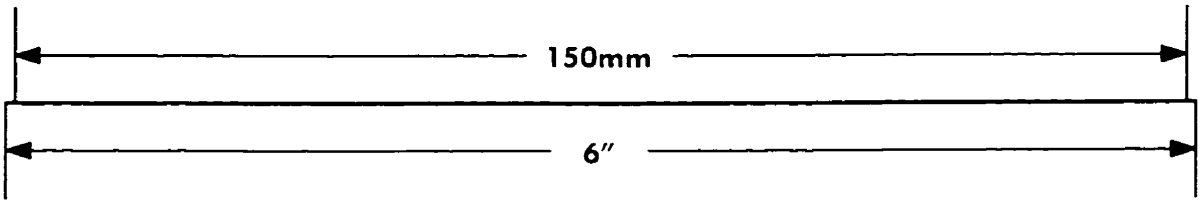
[27] Zhou, H. and Geist, A. Faster (ATM) Message Passing in PVM. In *Proceedings of 9th International Parallel Processing Symposium: Workshop on High-Speed Network Computing*, Santa Barbara, California, April 1995. Available at http://www.epm.ornl.gov/~zhou/patm.ps.

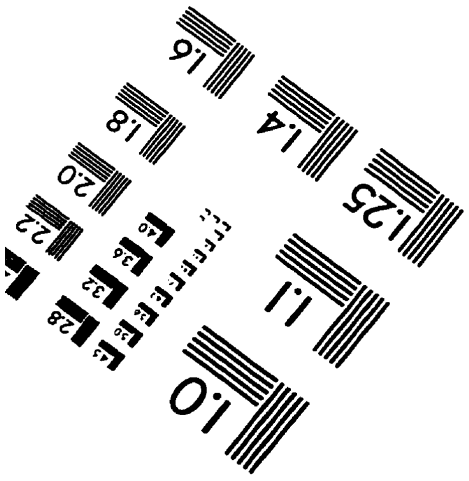
IMAGE EVALUATION
TEST TARGET (QA-3)

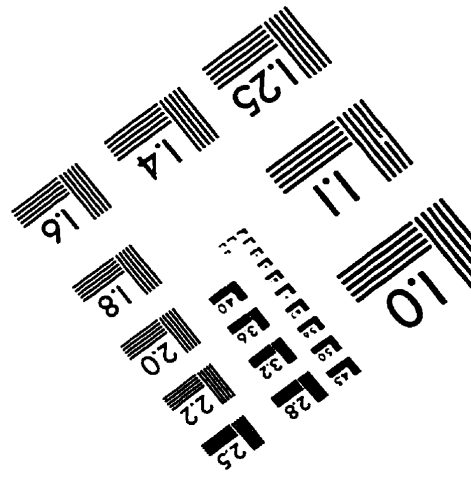












APPLIED IMAGE . Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989