Design and Analyses of a Cluster Computer

by

Damian Trybus

Faculty of Engineering Science
Department of Electrical and Computer Engineering

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario, Canada

Abstract

Parallel computing, as opposed to sequential processing, plays a growing role in solving increasingly complex computational problems. Traditionally mainframes and top of the line workstations were used for scientific (high power) computing. This dissertation investigates parallel computing by means of a variable computer cluster approach. An original variable cluster, based on PC-class computers, was produced and implemented for the purpose of this research, along with the necessary algorithms and computer codes. The processing performance of the variable cluster was evaluated in the case of different computing workloads provided by high incidence computational algorithms for one and two-dimensional FFT, as well as by Laplacian field (mesh) algorithm calculations. In order to allow for comparison with other studies (for instance, Amdahl's work), SpeedUp and Efficiency served as main concepts for the analysis of collected experimental data. Performance gain and reliability depend on the type of computing problem, amount of data transferred, number of machines participating in the computation, as well as on the physical characteristics of the machines and on infrastructure. A discrete model explaining the experimental data is proposed; an additional continuous model is also developed. "Resonance" workloads are to a certain extent predicted by our modeling, and the relation with computational performance is specified. Useful insights into the appropriate match between the computational algorithm and the cluster architecture are documented in our study.
The implemented computer cluster was found to be a robust platform that could be used for development of engineering applications requiring greater computing power than regular workstations can deliver. For selected cases the processing performance of the variable cluster scaled linearly with the number of nodes involved in the computation.

Keywords: Distributed Computer, Parallel Processing, FFT, Algorithms, Modeling

Acknowledgements

This project would have not been possible without the help and support I received from my advisor, Professor Z. Kucerovsky. The discussions we have had throughout the past six years have changed the way I think and opened my eyes to see things I never knew existed. He inspired me to go further than I believed I could and allowed me to make this project a reality.

I would like to thank my wife Dorothy and my children Daria and Daniel for their support. For a number of years they have put up with late nights, missed appointments, exams and other interruptions to their life so I could pursue my Ph.D. degree in Electrical Engineering. I owe them a great debt.

Last and certainly not least I would like to thank Adrian and Rodica Ieta for their help with the assembly of the thesis.

Contents

List of Tables

List of Figures

Nomenclature

Beowulf: Computing cluster based on PC hardware and Linux operating system

BOOTP: Bootstrap Protocol, IP management system for network hosts.

CM: Cluster Member.

CoW: Cluster of Workstations

CS: Cluster Server

CSMA/CD: Carrier Sense Multiple Access with Collision Detection. Medium access method for local area networks that employ a bus or tree topology.

DMS: Distributed Memory Systems

EDO: Extended Data Out. RAM type used in Pentium class computers.

Efficiency: A measure of parallel performance that is closely related to speedup. Efficiency is defined as: $Efficiency = \frac{SpeedUp}{N} \times 100\%$

FFT: Fast Fourier Transform.

IP: Internet Protocol, message passing protocol

LAN: Local Area Network

Linda: environment used for developing programs on DMS systems.

Linux: UNIX clone operating system

MIMD: Multiple Instruction Multiple Data.

MISD: Multiple Instruction Single Data.

MPI: Message Passing Interface, environment used for developing programs on DMS systems.

MPP: Massively Parallel Processor.

NFS: Network File System.

NIC: Network Interface Card.

NoW: Network of Workstations

NUMA: Non-Uniform Memory Access

OS: Operating System.

Parallel Computer: Computer with more than one processors capable of executing multiple instructions at the same time

PC: Personal Computer

POST: Power On System Test, self conducted computer tests prior boot.

PVM: Parallel Virtual Machine, environment used for developing programs on DMS systems.

RAM: Random Access Memory.

ROM: Read Only Memory

SDRAM: Synchronous Dynamic Random Access Memory.

SIMD: Single Instruction Multiple Data.

SISD: Single Instruction Single Data.

SMP: Symmetric Multi Processor

SMS: Shared Memory System

SpeedUp: $\frac{\text{Time required for one processor to compute a task}}{\text{Time required for N processors to compute a task}}$

TCP: Transfer Control Protocol, runs on top of IP.

TFTP: Trivial File Transfer Protocol

UDP: User Datagram Protocol, runs on top of IP.

UMA: Uniform Memory Access

x86: Processor architecture base on Intel x86 processor (i386–i686)

Chapter 1

Introduction

Sequential computers (computers that perform one task at a time) are quickly reaching a physical limit regarding the speed at which they process data, which cannot be increased through the use of faster components. The speed of light places an upper limit on the performance that can be achieved with a sequential architecture. For many computational problems, the time they take to obtain a solution using a sequential computer is unacceptably slow. One way out of this impasse is provided by parallel computation. On a parallel computer, several processors cooperate to solve a problem simultaneously in a fraction of the time required by one processor [4].

1.1 Why Parallel Computation?

Parallel computers are used primarily to speed up computations. A parallel algorithm can be significantly faster than the best possible sequential solution. There is a growing number of applications, in science, engineering and medicine that require computing speeds that cannot be delivered by any current or future conventional computers. These applications involve processing large amounts of data, or performing a large number of iterations, or sometimes both. The practicality of problem solution is often dictated by associated time constrains. The relevance of a 24 hour weather forecast may be questioned if it requires 36 hours to calculate [23, p 1]. Parallel computation is the only approach known today that would make these computations feasible.

1.2 Cluster Computers

Engineers and scientists are the primary users of high performance computers. Historically their options were limited to a few computing platforms. Large problems could be solved only on mainframe computers or state of the art, high performance workstations. Access to mainframes is usually not very convenient and state of the art workstations are expensive. Computer technology is changing in a rapid manner and machines become obsolete as soon as they are delivered to the user. It is also quite difficult and cumbersome to migrate data, applications and system settings from an old machine to a new one on a frequent basis.

Personal Computer (PC) revolution brought computers to the masses. PCs became common these days as calculators were several years ago. One can find networks of PCs in libraries, classrooms and laboratories. While those machines are usually used on a regular basis, they spend most of their time idling, doing nothing useful. Quite often these idling machines are connected to a Local Area Network (LAN) and they can also be accessed remotely.

While PCs idling in libraries are usually not high end machines, there are many of them and they are connected to a fast LAN. Traditional parallel computers have scaling problems and tend to be very expensive. It is possible to organize several workstations into a computing cluster and utilize their idling processors by scheduling and coordinating tasks that would be computed by the machines. Advantages are clear; there is no need to purchase new hardware (as existing resources would be utilized) and, with clever programming, problems that could be solved only by large machines would easily be solved by a cluster of workstations.

1.3 Scope of the Project

The project involved the design, implementation and analysis of a computing cluster. Several workstations were converted into cluster members for the experiment. A high end PC was assembled from standard, commercially available, parts and configured to administer the cluster. Several benchmarking applications were written and run on the cluster. The performance and applicability of the cluster were analyzed. The two most important performance aspects analyzed were *speedup* and *efficiency*.

1.4 Background of Thesis

Plentiful and inexpensive computer hardware together with free and powerful operating systems have led to the advent of distributed computing. Networks of Workstations (NoWs), Clusters of Workstations (CoWs) and computing clusters, such as the Linux based Beowulf cluster, have become very common and their applicability is the subject of studies in many research institutions. Flores [48] writes:

> "Research in parallel computing has traditionally focused on multicomputers and shared memory multiprocessors. Currently, net-

> works of workstations (NoWs) are being considered as a good alternative to parallel computers. That is due to there are high performance workstations with microprocessors that challenged custom-made architectures. This class of workstations is widely available at relatively low cost. Furthermore, these networks provide the wiring flexibility, scalability and incremental expansion required in this environment."

The majority of the Beowulf clusters in operation today run industrial benchmarks and the results of the benchmarks are compared against commercial supercomputers. Performance analysis of a high performance workstation conducted by Trybus [134] demonstrated several problems in SMP architectures. The main idea behind the conducted research was to build a distributed computing cluster and analyze its performance. The main emphasis was put on creating an open platform that could be used for conducting a variety of engineering experiments. Several applications were run on the cluster and its performance was evaluated.

1.5 Design Considerations

Several factors influenced the design of the cluster. The most notable factors include:

- Utilization of standard, commercially available hardware,
- Adaptation of standard software, operating system and networking software,
- Scalable and expandable architecture,

- High performance to price ratio,
- Flexibility and ease of configuration.

Researchers agree that such design decisions are difficult to make. Havick [61] writes:

> "One of the most difficult tasks in designing and commissioning a Beowulf cluster is considering the price/performance trade-offs from the multitude of possible configuration options. There are four crucial hardware parameters to choose in the design of a Beowulf cluster: the type of processor to use in the nodes; amount of memory installed in each node; the amount and type of disk installed in each node and the network infrastructure that is used to connect the nodes. The best options will depend on the particular application."

Havick [61] also states that Beowulf clusters are typically built from commodity computers, usually PCs with x86 processors or workstations based on RISC processors.

The thesis investigates the efficacy of the selected architecture in solving a range of scientific problems and determines the performance as well as the efficiency of the system. The thesis also demonstrates the importance of the match between the algorithm and the architecture in achieving maximum computational performance.

Chapter 2

Cluster Computing: Theory and Applications

This chapter introduces the theory of cluster computing, demonstrates known architectures and presents applications that could be run on a cluster computer. Parallel computing limitations are briefly described and Amdahl's law is discussed in some detail.

2.1 Cluster Computer Theory

In the late 1940's, a group of researchers at Princeton University proposed a design that ushered the modern computer era. Half a century later the overwhelming majority of computers in use follow this original design. In such a design, presented in figure 2.1, a computer consists essentially of a single processing unit, local memory and input/output devices. The processing unit executes a single sequence of instructions on a single sequence of data. Both instructions and data are stored in main the memory of the computer. The

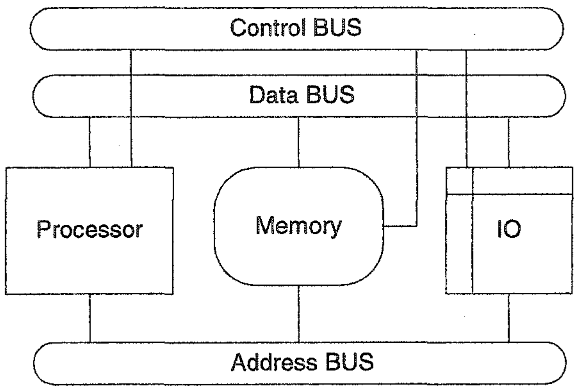


Figure 2.1: von Neumann Computer Architecture

sequence of instructions is the program, which tells the processor how to solve a certain problem. The sequence of data is an instance of that problem. Such a computer performs one instruction at a time. Such a model of computation is known as sequential (or serial, or conventional) computer [4, p. 2].
A parallel computer, by contrast has two or more processors. Such a computer is capable of processing more than one sequence of instructions on one or more sequences of data at the same time.

A cluster computer is a collection of off-the-shelf workstations connected by an off-the-shelf LAN [6, p. 475]. A typical cluster configuration is shown in figure 2.2.

The availability of inexpensive hardware and free sophisticated operating systems allowed researchers and developers to design and analyze PC clusters. Koski [73] writes:

> "During recent years clustered systems using off-the-shelf processors and standard Ethernet networks have been increasingly popular. The motivation has been primarily the cheap price of systems, but also the rapid development of standard processors. So-called Beowulf systems have spread around the world. This development

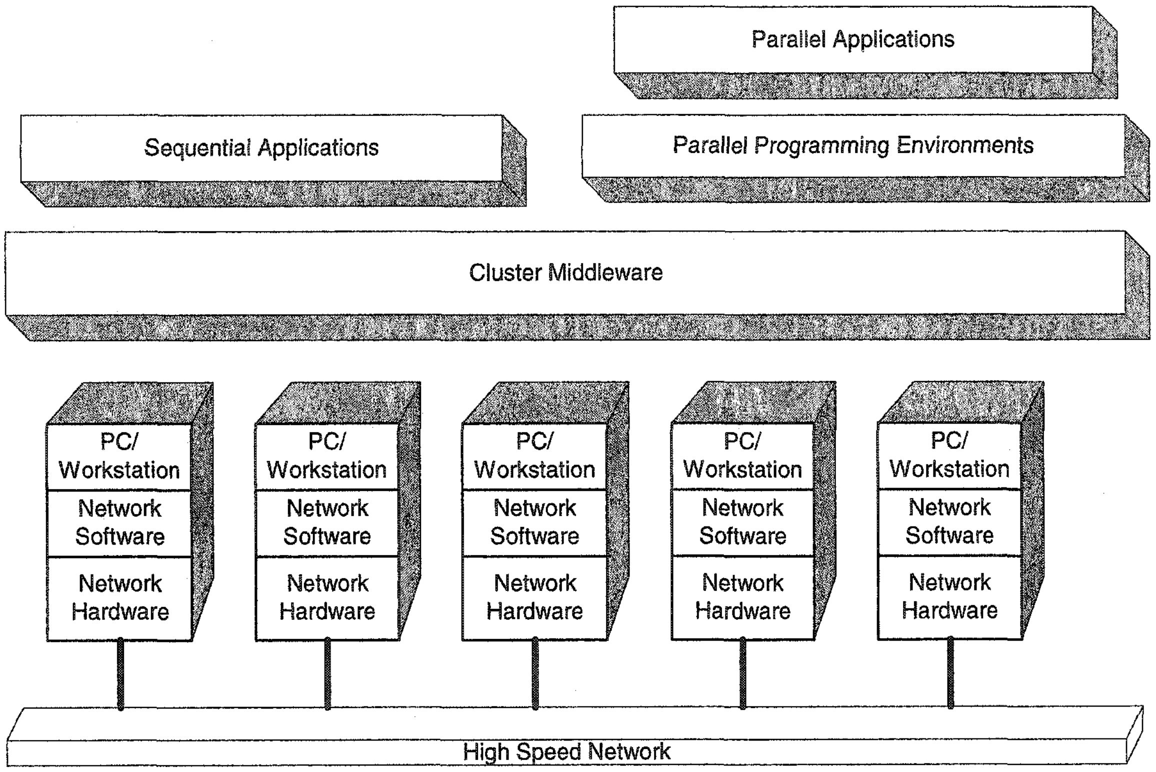


Figure 2.2: Cluster Computer Architecture

> is further accelerated by the Linux-boom which provides an ideal and free operating system for these clusters."

There are several reasons for designing and implementing cluster comupterse. Cluster environment can be used for:

- Fault Tolerance,
- Load Balancing,
- High Performance Computing.

Computer fault tolerance is quite often implemented by the means of identical or very similar systems where the backup system is aware of the state of the system it is protecting. The state of the cluster is usually preserved on a shared disk and the two systems have a private communication link that is used to determine the availability of the production system. Should the production system fail, the backup system will come on line with the same services that the production system offered and the state of those services will be the same as the one prior to the failure of the production server. Fault tolerance is usually implemented on mission critical database servers. Microsoft Wolfpack is an example of such technology [119].

Load balancing is a popular way of increasing the availability of a service by means of two or more systems providing the same static services or services that do not change with time. An example of such a system would be a web server serving web pages to clients. The web content can be replicated to multiple servers, possibly located in different parts of the world. Clients do not care where the information comes from as long the requested information is received. The only requirement that needs to be fulfilled is that the information be consistent among all participating cluster members.

High performance, cluster computing is driven by the following factors [109, p. 53]:

1. Solving large problems that ran too slowly even on the fastest supercomputers (simulations, scientific engineering applications).

2. Solving problems that were too large for any other available computer (multi-dimensional FFT's, operation on large matrices).

3. Cost saving computing of problems that could be solved on existing albeit more expensive hardware.

Researchers agree [142] that there is a renewed diffusion of parallel platforms from symmetric multiprocessors to PC clusters. Santoso [121] writes:

> "With the advent of large computing power in workstations and high speed networks, the high-performance computing is moving from the use of massively parallel processors (MPPs) to cost effective clusters of workstations."

Workstation based clusters have become a feasible alternative to expensive, commercially available systems.

2.1.1 Cluster Computer Architecture

A formal classification of computer architectures according to a macroscopic view of their principal interaction patterns relating to instruction and data streams was proposed by Flynn in 1972. What has become the so called Flynn's taxonomy is shown in figure 2.3 [23, p. 14]. A cluster computer

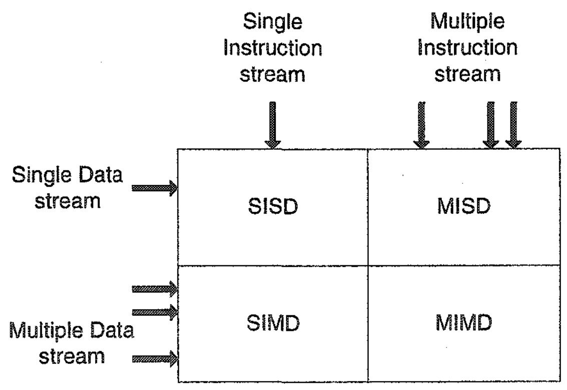


Figure 2.3: Flynn's Taxonomy for Processors

falls into the MIMD category as it is possible to run multiple instructions streams (MI) working on multiple data (MD) at the same time. There are two main architectures used for implementing cluster computers. The first architecture, UMA (Uniform Memory Access), uses processors that share common memory. A more common name used for UMA systems is SMS or Shared Memory Systems. The second popular architecture, NUMA (Non-Uniform Memory Access) uses processors that have private memory and communicate via a bus or a network. In this thesis NUMA systems are referred to as DMS or Distributed Memory Systems.

Shared Memory Systems

Shared memory systems utilize the most prevalent form of parallel architecture used in multiprocessors of small to moderate scale. This architecture provides a global physical address space and access to all of main memory from any processor [38, p. 269]. A generic view of the SMP architecture is shown in figure 2.4. Two variations of the generic implementation exist; they are shown in figures 2.5a and 2.5b. Figure 2.5a shows a multiprocessor computer where both the cache and the main memory are shared. Such architecture does not suffer from the cache coherence problems but if the combined speed of the CPUs is larger than the speed of the cache serious performance degradations are experienced. This approach has been used for connecting very small numbers of processors, usually 2–8. Such architectures were very popular in the mid-1980's. The architecture shown in figure 2.5b is the most common SMP architecture found in modern multiprocessor systems. Each processor has its own cache, where it stores instructions and data. Such architecture suffers from cache coherence problems [38, p. 273]; however, if properly implemented it delivers great performance. This architecture is used to implement medium scale multiprocessors consisting of 20–30 processors.

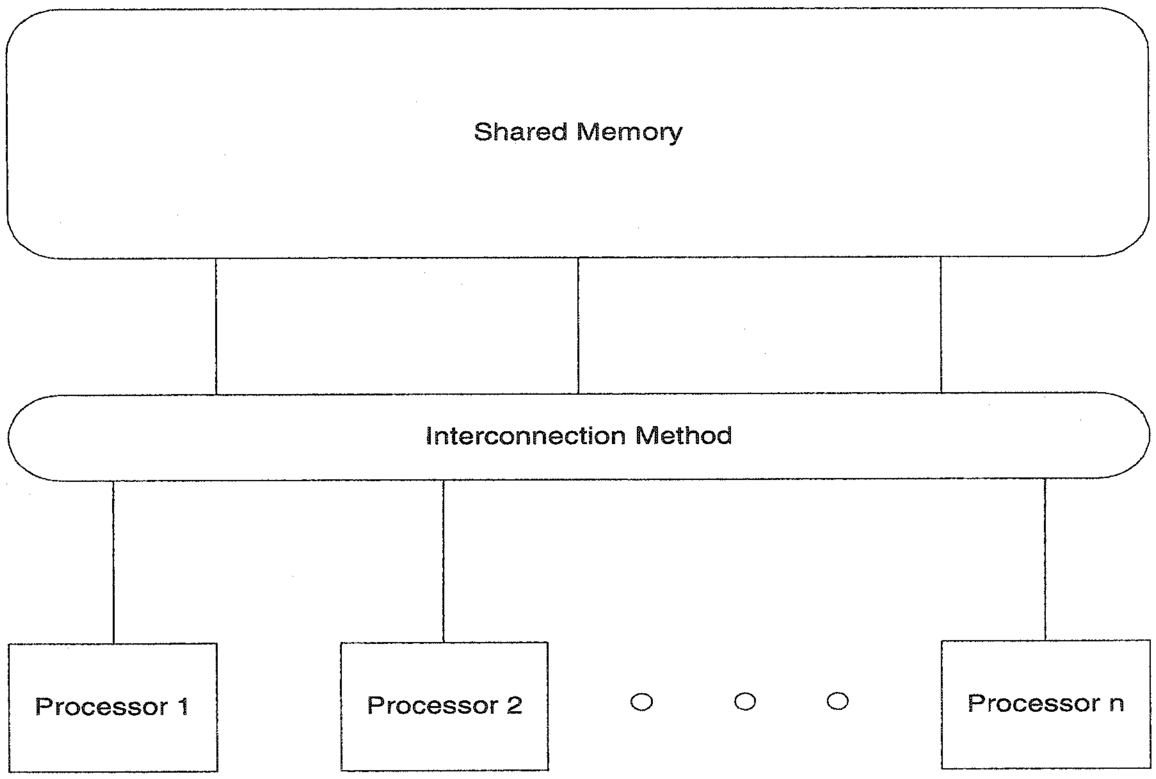
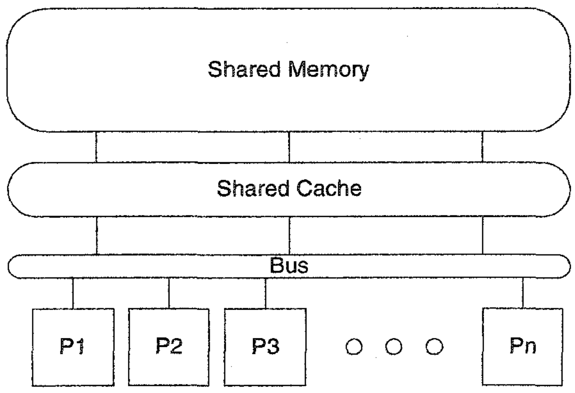


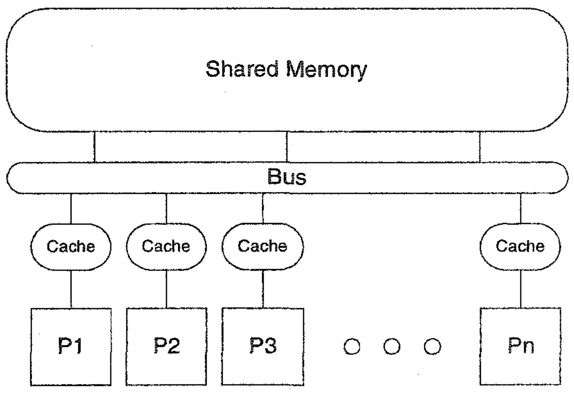
Figure 2.4: Shared Memory Cluster Computer

In the SMP architecture the shared space is supported directly by hardware. User processes can read and write shared virtual addresses, and these operations are realized by individual loads and stores of shared physical addresses. The operating system does not need to be involved in address translation because it is provided by the hardware.

Sharing the memory uniformly amongst all processors allows each processor equal access to all memory locations. The memory in UMA machines is typically implemented in a central location with the processors acquiring access across a high-speed interconnection mechanism such as a bus or crossbar switch. Communication and thus co-operation amongst processors is tightly



(a) Shared cache



(b) Dedicated cache

Figure 2.5: SMP architectures

coupled and occurs within the common memory via shared variables. Some arbitration mechanism is necessary to prevent simultaneous updates of these shared variables and to solve contention on the interconnection network [23, p. 29].

Scaling considerations. SMP systems use processors connected to one shared bus. A shared bus has a maximum length, and a fixed maximum bandwidth. These physical constraints limit expandibility of a SMP machine. In modern machines buses run at high speeds and the width of the connecting conductors is usually no longer than a few inches. The links become slower with length and every technology has an upper limit on length due to power requirements and signal-to-noise ratio[38, p. 455]. Chip-level integration technologies allow denser packaging and have been implemented by several vendors. The SMP systems available on the market today usually do not exceed 64 processor configurations.

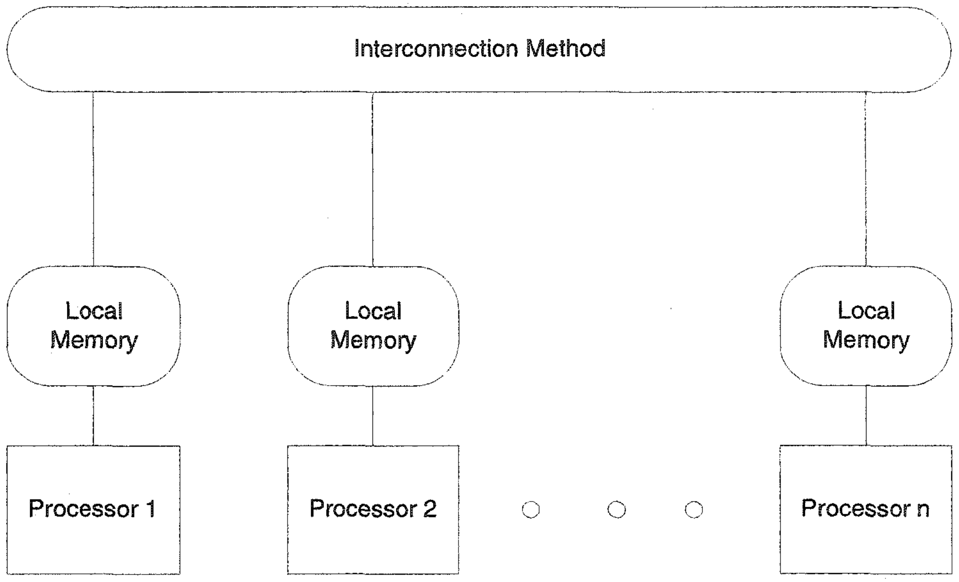


Figure 2.6: Distributed Memory Cluster Computer

Distributed Memory Systems

Implementing a virtual shared memory environment across all processors of a multiprocessor system introduces complex global communication patterns, as processing elements may need to fetch data items from anywhere within the system. Such communications place the heaviest strain on any system. An answer to this problem might be provided by NUMA systems. NUMA computers or distributed memory systems have memory that is physically distributed amongst the processors. The distributed memory is still accessible to all processors; however, the access time will differ depending on whether the requested memory address is local or remote to the requesting processor. A remote memory access requires communication across the interconnection network that links the processors and thus the distributed memory [23, p. 29]. A generic view of the DMS architecture is presented in figure 2.6.

Performance considerations. The performance of a distributed memory multiprocessor depends in large part on the efficiency of the message transfer system that provides the interface between the co-operating processors [23, p. 203]. Communications, by their nature, fall into the sequential processing category and thus can affect parallel systems' performance. Communication overheads inhibit overall system performance. Thus the efficiency of system communication plays a crucial role in reducing implementation penalties and in improving the scalability of the parallel solution of any problem.
Distributed memory systems use message passing mechanisms to communicate with member computers. Communications among member computers are the most critical points to support parallel applications in distributed memory systems [54]. The most widely used communication in computer clusters is message passing on an Ethernet network.

Ethernet Networks Ethernet technology was developed in late 1972 at Palo Alto Research Center (PARC) of Xerox Corporation. The design was successful and now Ethernet is the predominant LAN technology. The early Ethernet specifications contributed substantially to the work done by the IEEE of the 802.3 standard defining the CSMA/CD.
Data on an Ethernet network is transferred in Ethernet frames which are later encapsulated by TCP/IP frames. An Ethernet frame consists of the following sections: 1)preamble (8 bytes), 2) destination (6 bytes), 3) source (6 bytes), 4) type/length (2 bytes), 5) data (46-1500 bytes), 6) frame check sequence (4 bytes). An Ethernet frame can vary in size from 64 to 1518 bytes[86].

Message Passing

In figure 2.7 we can see a user level send/receive message passing abstraction as proposed by Culler [38, p. 39].

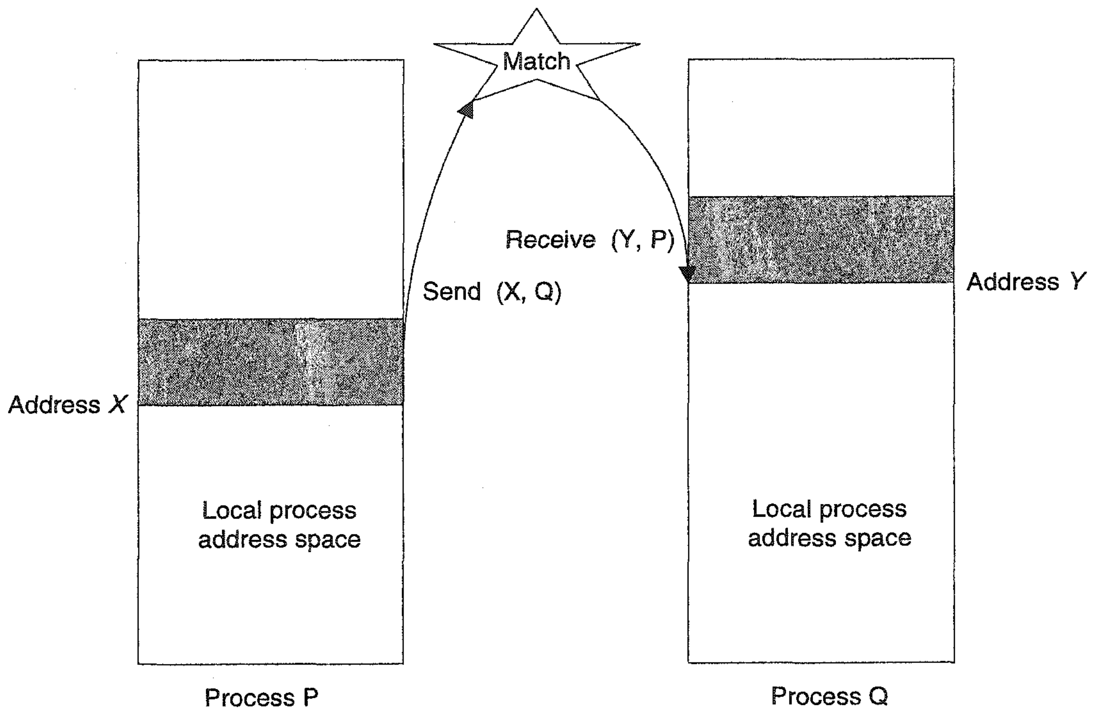


Figure 2.7: User-level send/receive message-passing abstraction

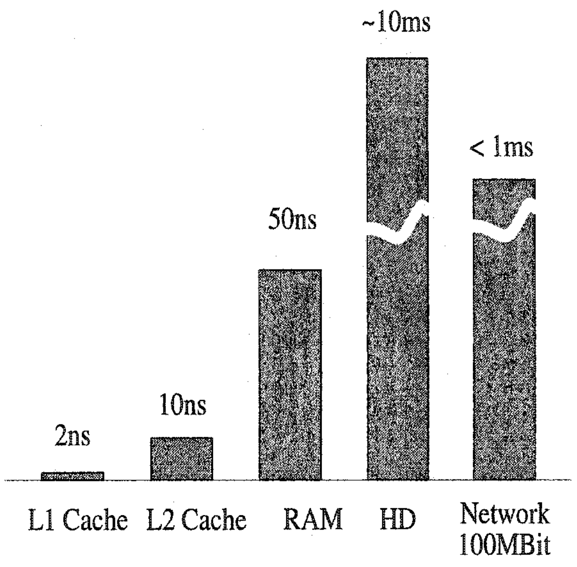
There are several techniques to accomplish member communications. There are widely used programming environment or toolkits for writing parallel programs to run on distributed memory MIMD hardware. Environments such as MPI, PVM and Linda provide constructs that allow a program to perform the three essential functions [23, p.s 41-54]:

- Define Parallel Execution
- Start and Stop Execution

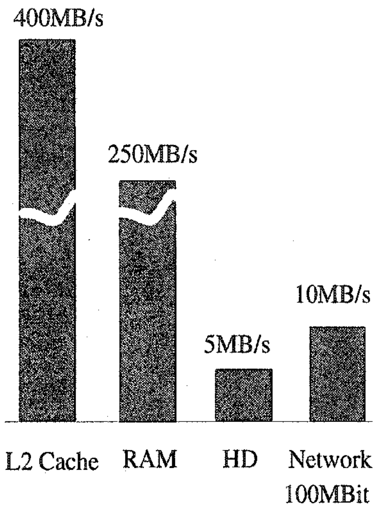
- Coordinate Parallel Execution

These environments have been implemented on many parallel architectures, and are particularly in demand as possible ways to obtain parallel execution on LAN-connected workstations. All of these implementations use IP and tend to give latencies in the millisecond range. [6, p. 249]. From Hobs [65] we learn that execution environments built on top of an operating system such as PVM introduce unnecessary overheads, since many of the services provided by the environment are also offered by the underlying operating system.
The performance of a programming environment needs to be balanced against ease of use, in particular in engineering applications. Rackl [115] has determined that in the CORBA's environment the data overhead is about 30% over plain TCP/IP. It has been determined that plain TCP/IP adds hundred to several thousand instructions per message [48]. However, even though TCP/IP introduces overhead and has large latency, its overall overhead is only about 4% of the total transfer [115]. Given the above observations, it was concluded that plain TCP/IP socket based communications will be used as the message passing mechanism in the cluster.

One needs to carefully design a distributed application in order to benefit from the cluster's combined power. Contrary to SMS machines, the communications of the DMS machines are very expensive. Matsuda [89] shows that the memory bandwidth of a current PC is at least two orders of magnitude greater than the bandwidth of a 10MBit Ethernet network. A summary of latency and bandwidth of common I/O devices is listed in figures 2.8a and 2.8b. Because of these constrains it is generally assumed that only embarrassingly parallel applications (that is, applications that almost never communicate) can make use of workstation clusters [11].



(a)Latency



(b)Throughput

Figure 2.8: Latency and bandwidth of I/O devices

2.2 Parallel Processing

Almasi [6, p. 5] defines the parallel processor as:

> **"A large collection of processing elements that can communicate and cooperate to solve problems fast."**

The author quickly adds that this definition raises more questions than it answers. In figure 2.9 we see a generic scalable parallel processor organization as proposed by Culler [38, p. 51].

2.2.1 Computations

Multiprocessor hardware delivers greater power than single processor equivalents only when it is properly utilized [134]. The two most common techniques used on multiprocessor architectures are *processes* and *threads.* Quite often a combination of both techniques is used.

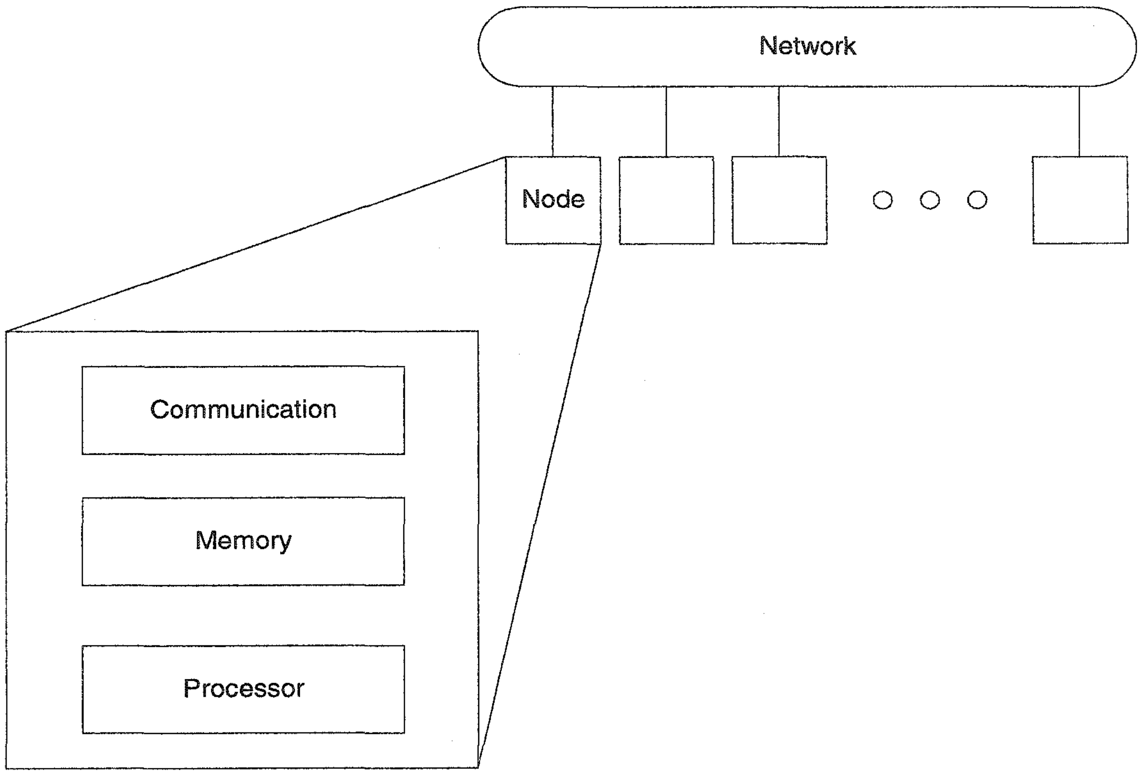


Figure 2.9: Generic scalable multiprocessor organization.

Processes

A process is an independent program with its own memory for local variables and a stack for procedure calls. Multitasking operating systems can run multiple processes simultaneously. All running processes are distributed evenly among all processors available in the system. In general, the creation of a process is an expensive task. The operating system has to allocate memory space for the process, load the process into memory and start executing it. Once the process is started, it is difficult and expensive to communicate with it. In order for distinct processes to communicate with each other they have to use interprocess communication techniques involving pipes and system calls. Processes have been extensively employed on multiuser, time sharing machines, to utilize the hardware efficiently. Multi processor SMS machines can benefit from multiple processes running concurrently. An example of such an efficient utilization is a server computer running a Web server process and a Database process. Clients connect to the server via the Web and update data in the database. The server can run both processes concurrently provided it has enough processors to run them on.

Threads

A thread is an independent procedure running inside a process. A process can have many working threads. Threads are inexpensive to create and have full access to process data. This means that a program can have multiple threads communicating with each other via shared variables, similar to the way procedures communicate with each other. Thread oriented operating systems are capable of assigning program threads to separate processors. This phenomenon is very beneficial, since it is possible to develop multithreaded programs and take advantage of multiprocessor hardware. The operating system, running a multithreaded process, would schedule the processes' threads to run on distinct processors, and thus higher throughput could be achieved [134].

Threads are commonly used on SMS machines. There have been numerous attempts to extend the thread programming paradigm to the DMS environment; however, so far none has been very successful [124].

Threads vs. Processes. Industry studies show that it is much more expensive to create, and context switch a process than a thread. A new UNIX process takes about 11 times more time to create than a new thread on the same computer. The same studies show that it takes 5 times more time to switch between UNIX processes than to switch between threads belonging to a common execution environments. The context switch cost is most important because it is incurred many times in the lifetime of a program [32].

2.2.2 Communications

Distributed applications running on a cluster computer need to communicate with each other and access global data. In section 2.1.1 we learn that cluster computers fall into the MIMD category. Such machines cannot use shared variables for communications. SMS systems could use inter-processes communications as described above. DMS machines, however run processes on physically distinct machines. The most common way of communications on such systems are remote procedure calls, which are commonly implemented via sockets [20].

Data Transfer

A network computer or computer cluster is heavily dependent on the interconnecting infrastructure. Such machine communicates with cluster members using network. In order to evaluate performance several concepts need to be identified and measured.

Data Transfer Time. The time required for a data transfer operation is generally described by a linear model:

$$\text{Transfer Time} = T_0 + \frac{n}{B}[s]$$

where n is the amount of data (usually in bytes), B is the transfer rate of the medium (in comparable units to n, usually bytes/sec) and T_0 is the start-up cost. This model is used widely to describe a diverse collection of operations, memory accesses, bus transactions, and message passing. Culler also notices that the bandwidth of a data transfer operation depends on the transfer size; as transfer size increases, it approaches the asymptotic rate of B, which is sometimes referred to as r_∞. How quickly it approaches this rate depends on the start-up cost. It is easily shown that the size at which half of the peak bandwidth is obtained, the *half-power point*, is given by:

$$n_{\frac{1}{2}} = \frac{T_0}{B}$$

Communication Time Communication time is the time that is required to establish communication and to transfer data between cluster members. The following model is used to describe this operation:

$$\text{Communication Time(n)} = \text{Overhead} + \text{Occupancy} + \text{Network Delay}$$

where *Overhead* is the time spend by the processor preparing the message and initiating the transfer, *Occupancy* is the time it takes for the data to pass through all components on the communication path (hubs, switches, routers) and *Network Delay* is the remaining communication time (access to media, collisions, etc.)

Communication Cost From the performance point of view the most important fact is the Communication Cost. The following model was proposed by Culler to define the communication costs:

$$\text{Communication Cost} = \text{Frequency} \times (\text{Communication Time - Overlap})$$

where the *Frequency* is the number of communication operations per unit of work in the program and the *Overlap* is the portion of the communication operation that is performed concurrently with other useful work (computations or other communications) [38, pp. 60-63].

System overhead

We realize that network communication involves more than just the transport medium. Several factors have been identified that contribute to the overall performance and capabilities of a network communication layer [68].

Context Switching. The difference between special purpose HPC systems and cluster workstations begins to disappear, as programs are now running in a multiprocess environment. This makes context switching overhead hard to avoid. This is particularly visible when one overlaps communication with computation.

System Call Overhead. In an operating system such as Linux it is the kernel's task to access the actual networking hardware via a system call, shielding the hardware from the application for portability and security reasons. The approach also allows for sharing of the hardware between different applications. If direct access to the hardware by an application is allowed a significant speed up can be obtained. This is however not acceptable for a large number of application domains that use the network.

Interrupt/Signal Latency: Network interface cards communicate with CPU through interrupts signaling finishing sending or receiving data. Handling interrupts causes overheads. Quite often those interrupts are propagated to the user space and further degrades the performance.

Semantics. The semantics expected by programs or interface definitions often do not match the most optimal way of sending or receiving data. Convenience does not match efficiency. While near optimal communication speed can be reached for simple operations, this will not be possible in practice for more sophisticated operations such as multicasting and non-blocking operations.

Reliability. Software communication layers must deal with packet loss on the hardware level. TCP/IP implements this by using sequence numbers and acknowledgments which in turn lead to overhead.

2.3 Parallel Processing Examples

The previous section outlined several problems that a cluster designer must face when designing a cluster. The following section gives two examples of thinking in parallel.

2.3.1 Emptying a swimming pool using pails.

Openshaw [109] maps parallel processing to tasks occuring in everyday world using the following example. Consider the problem of emptying a pool using a pail. If one worker would need T time to empty the pool then quite likely ten workers would empty the pool in approximately $T/10$ time and N persons could empty the pool in T/N time. We know that the pool is of finite size and that it contains X pails of water. While this is a clear parallel task (each person could carry a pail of water independently), it is very unlikely that the pool would be emptied in one unit of time if X workers were used. There are several possible aspects that can affect the time required to complete the operation. For example, there might not be enough space to accommodate more than W workers, adding more than W workers would cause collisions and serious congestion in gaining access to the pool (shared resource). We can also observe that a possible saturation has occured when the workforce reached W workers. Introducing additional workers would mean that some of the workers already working would have to be retired or slowed down.

2.3.2 Assembling a hard disk using a pipeline.

One major disk manufacturer produces hard drives that are assembled sequentially in various plants around the world. Casing is produced in the USA. The casing is then shipped to the UK where the motor is mounted. The platters are mounted in Malaysia and the head assemblies are mounted in Taiwan. The finished product is sent to the USA. As long as the pipeline is filled and the flow can be sustained, all plants are working together to produce the product. It is often the case that a flaw is discovered in a batch of components. The entire pipeline is affected by the problem. If the problem is discovered late in the process, it might take weeks before the flow of the finished products is restored. The process can not be sped up by adding additional factories, as they will not help the process.

The above examples illustrate typical problems that a developer must face while designing and implementing a parallel processing machine. Installing more processing entities than there is work will not help solving the problem.

Moreover communications between the processing entities should be considered and kept to the minimum, especially if the processing entities do not share memory.

2.4 Parallel Performance

Quite often FLoating point OPerations (FLOPS) are used to describe the computing power of a computer. While it is possible to use FLOPS to describe the theoretical computing power of a cluster computer, it is quite often an unrealistic figure that does not describe the performance of a cluster computer. If one were to use FLOPS as a measure of the computing power of a cluster, one could use the following formula:

$$Power = \sum_{i=1}^{N} Machine_i \text{ FLOPS}$$

Summing the FLOPS of all machines participating in the cluster would give the theoretical performance of a cluster in FLOPS. That figure could only be used if all machines could work continously on a given problem at their peak speed. That formula does not consider inter processor communications, job scheduling overhead and processor synchronization problems. Figure 2.10 illustrates how those factors impair overall system performance. This figure shows two systems. The first one is a single processor machine computing in a sequential manner. 80% of the computing time is spent performing 'busy-useful' operations and 20% of the time is spent on accessing data. The second system consists of four machines computing in parallel. First we note that the total execution time of the second machine is not four times shorter than the first. The second system has several aspects it needs to deal with. The busy-useful fraction is close to 20% but we also have factors such as busy overhead (instructions not needed in sequential program), data remote (time spent waiting for remote data) as well as interprocess synchronization issues [38, p. 157].

In order to describe accurately the cluster's performance it is proposed that two terms be used: *SpeedUp* and *Efficiency*. SpeedUp describes how much performance gain is achieved using the cluster. Efficiency shows how efficient the cluster participants utilized in the cluster are.

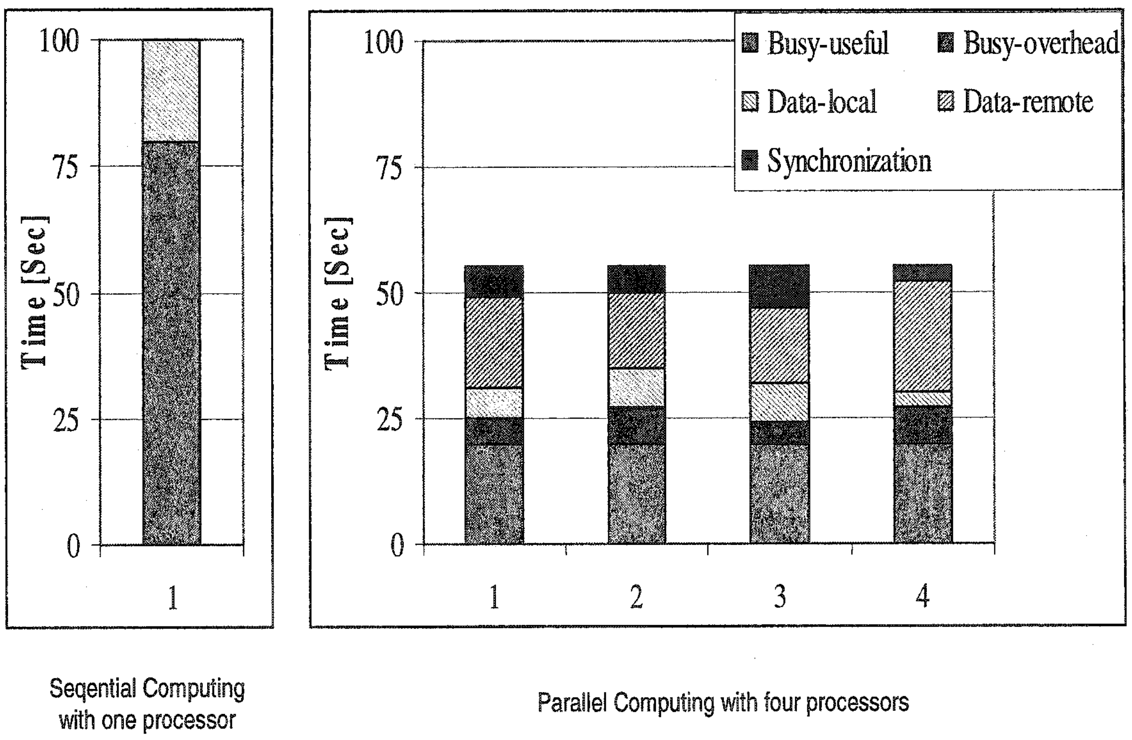


Figure 2.10: Components of execution time from the perspective of an individual processor.

2.4.1 SpeedUp

SpeedUp is defined as:

$$SpeedUp = \frac{\text{Time required for one processor to compute a task}}{\text{Time required for } N \text{ processors to compute a task}}$$

or we could also write this in the following way:
$SpeedUp =$

$$\frac{Busy(Time_{1processor}) + Data_{local}(Time_{1processor})}{Busy_{useful}(N) + Data_{local}(N) + Synch(N) + Data_{remote}(N) + Busy_{overhead}(N)}$$

SpeedUp can be a number from 0 to N, where N is the number of processors present in the system. Ideally we would like to achieve the so called perfect SpeedUp, which is a linear function: $SpeedUp = N$ i.e. If a problem takes T time to compute on one processor, the same problem run on N processors would be computed in $\frac{T}{N}$. Perfect speedup is rarely achieved in practice. Algorithms that achieve linear SpeedUp are called completely parallelizable, and not surprisingly, are highly desired [6, p. 195].

Superlinear SpeedUp

Superlinear SpeedUp is defined as a speedup that is greater than the number of processors used. Superlinear Speedup is achieved when a large sequential problem can be mapped efficiently on a set of processors participating in the experiment. The data and the code of a large problem quite often would not entirely fit into memory and cache. Given multiple processors a problem could be divided in such way that every processor computes only on a fraction of the entire dataset. Then each processor could utilize its cache and registers more efficiently and superlinear speedup could be achieved. Superlinear speedup usually indicates that the sequential problem had cache miss or page fault problems.

2.4.2 Efficiency

Efficiency is a measure of parallel performance that is closely related to speedup. We define efficiency as:

$$Efficiency = \frac{SpeedUp}{N}$$

One could define efficiency as the average speedup per processor. In a computing cluster it is not very likely that every processor will devote 100% of its time to cluster computations. Efficiency measures the fraction of time the processors are being useful. The range of efficiency lies between 0 and 1. When efficiency is equal to 1 this corresponds to perfect speedup of:

$$SpeedUp = N$$

Efficiency Example

Consider the problem of multiplying a vector of 100 elements by a scalar S. The pseudocode to perform such operation would be written as follows:

```
For $i = 1$ To 100
$X_i = X_i * S$
Next $i$
```

If this operation is performed on a single processor and the time required to perform one iteration is t then the computation will take

$$SingleProcessorTime = 100t$$

If the operation is performed on a computer with eight processors, the following strategy could be employed. Each of the eight processors would perform equal number of multiplications and the remaining multiplications would be performed by a single processor. In our case seven processors would perform twelve multiplications and one processor would perform twelve multiplications together with the seven processors and then it would perform four multiplications. The total execution time would then be $12t + 4t = 16t$
The SpeedUp is then calculated as:

$$\text{SpeedUp}_{8CPU} = \frac{100t}{16t} = 6.25$$

The Efficiency is calculated as:

$$\text{Efficiency}_{8CPU} = \frac{6.25}{8} = .78125 \text{ or } 78.125\%$$

The less than perfect SpeedUp is due to load imbalance.

2.4.3 Amdahl s Law

There is considerable skepticism regarding the viability of massive parallelism; the skepticism centers around Amdahl's law, an argument put forth by Gene Amdahl in 1967 [7] according to which even when the fraction of serial work in a given problem is small, s, the maximum SpeedUp obtainable from even an infinite number of parallel processors is only $1/s$.

If N is the number of processors, s is the fraction of time spent (by a serial processor) on serial parts of a program and p is the fraction of time spent (by a serial processor) on parts of the program that can be done in parallel, then Amdahl's law says that speedup is given by

$$SpeedUp = \frac{1}{s + \frac{p}{N}}$$

where $s + p = 1$

The range of s lies between 0 and 1 (0% and 100%). When $s = 0$, then $SpeedUp = N$ and perfect parallelism is achieved. When $s = 1$, then $SpeedUp = 1$, and there is no benefit from parallelism. SpeedUp is limited by the fact that not all parts of our code can be run in parallel. Even if an infinite number of processors is used, the SpeedUp is still limited by $1/s$ [53, pp 24-26]. The sequential fraction s has a strong effect on SpeedUp. This explains the need for large problem sizes. As the problem size increases the opportunity for parallelism grows, and the sequential fraction decreases and reduces its importance for SpeedUp.

Quinn [114, pp 45-47] reevaluates Amdahl's law. He states that if a large fraction of sequential code is identified it should be performed by the fastest participating processor and it should preferably be overlapping other code that could be done in parallel. Quinn also adds that parallel computers will be able to compete with supercomputers only if they have at least one processor capable of extremely fast sequential operation or if they execute algorithms with virtually no sequential component.

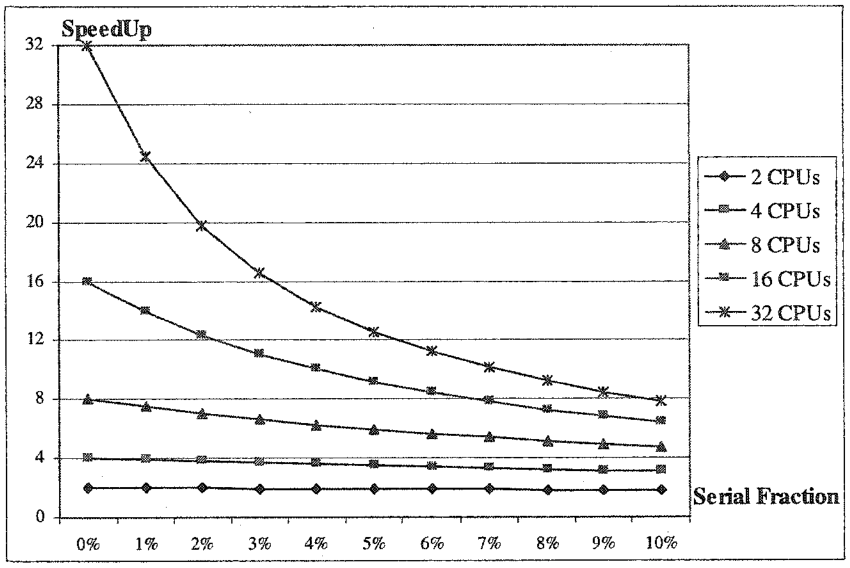


Figure 2.11: Amdahl's Law SpeedUp

2.4.4 SpeedUp Limitations

In practice quite often one encounters problems that cannot efficiently be solved on parallel architectures. The SpeedUp can be affected by several aspects. Richardson [118] identifies the following SpeedUp limitations:

- I/O
- Memory Contention
- Algorithm
- Problem Size
- Load Imbalance
- Sequential Code
- Parallel Overhead

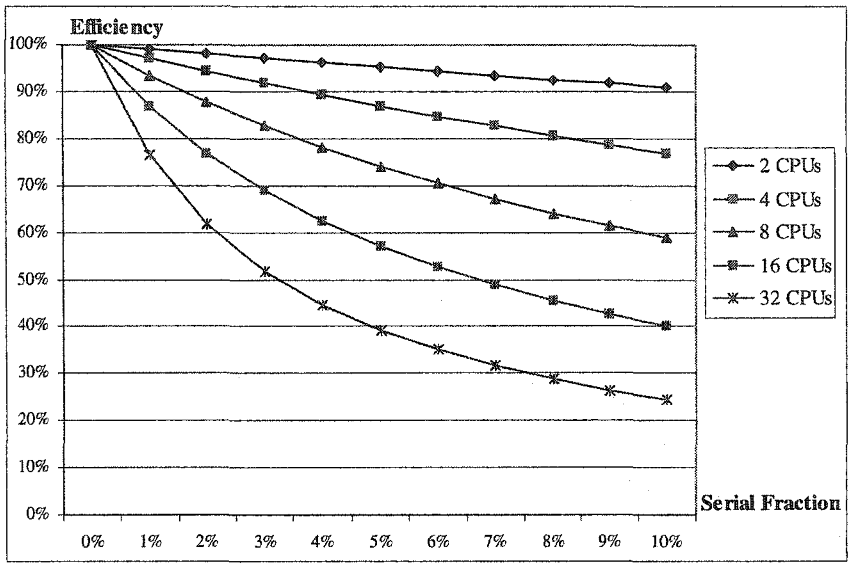


Figure 2.12: Amdahl's Law Efficiency

There are many problems involving continuous I/O operations. If a problem is I/O bound the slow (and very likely sequential) operations take more time compared to the amount of computation.

In the majority of cases computer algorithms deal with any problem in a sequential manner that is not suitable for parallel computers. Parallel versions of sequential algorithms need to be designed and implemented to utilize parallel hardware. SpeedUp is almost always an increasing function of problem size. The size of the problem can affect the way it can be solved. If a problem is trivial or too small to take best advantage of a parallel computer then it cannot be computed efficiently on a large parallel system. In other words, if there is not enough work to be done by the available processors, the system will show limited speedup. By the same token, if a problem size is fixed and it can be solved with a given set of processors, it will not benefit from additional hardware. Adding more processors will not reduce to computation time and in some cases it even increases computing time. Figure 2.13 shows two curves. The 'optimum time' curve shows the execution time as a function of the number of processors present in the system. Such result is what one would like to expect from a parallel computer. A more realistic curve is the 'actual time' curve. The curve shows an initial decrease in the time taken by the example problem on the parallel system up to a certain number of processing elements. Beyond this point, adding more processors actually leads to an increase in computation time [23, p. 78].

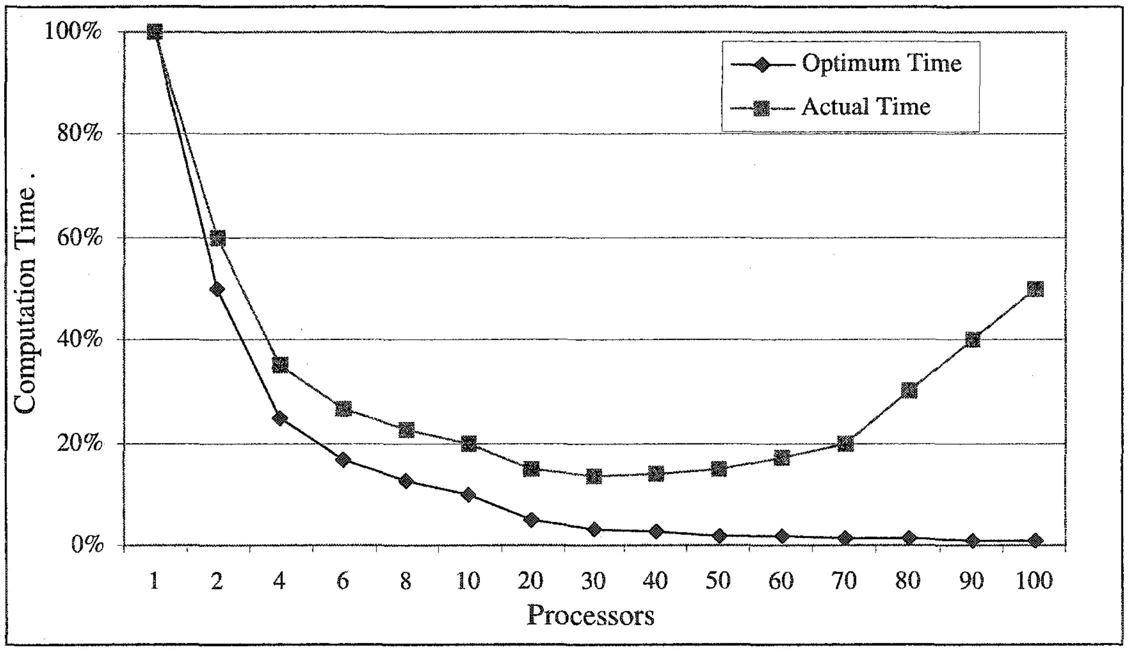


Figure 2.13: Optimum and actual parallel implementation times

In section 2.4.2 it was demonstrated how load imbalance can affect SpeedUp and overall efficiency. A designer will attempt to map a given problem onto parallel hardware, but quite often the processors will have unequal workloads. This causes some processors to idle as they wait for other processors to finish their work. Amdahl's law demonstrates the effects of the sequential part of the code on the overall performance. In general, most computer programs have a sequential nature. This limits speedup as shown by Amdahl's Law. Figures 2.11 and 2.12 show how even a small fraction of the sequential code can affect the SpeedUp. Parallel programming introduces additional overhead. Parallel algorithm is almost always larger and more complicated than a sequential equivalent. Additional processor cycles are required to create parallel regions, threads, synchronizing threads, and spin/blocking threads.

2.5 Performance Evaluation of the Cluster

A first step in evaluating a real machine is to understand its basic performance capabilities–that is, the performance characteristics of the primitive operations provided by the programming model, communication abstraction and hardware/software interface. The two most common ways of evaluating system performance are *microbenchmarks* and *workloads* [38, pp 215-217].

2.5.1 Microbenchmarks

Microbenchmarks are small, specially written programs designed to isolate performance characteristics such as latencies, bandwidth, overhead, etc.
Five types of microbenchmarks are used in parallel systems:

1. *Processing microbenchmarks* measure the performance of the processing capabilities of the machine.

2. *Local memory microbenchmarks* determine the organization, latencies, and bandwidths of the levels of the memory hierarchy within the local node and measure the performance of local read and write operations.

3. *Input-output microbenchmarks* measure the characteristics of I/O operations, such as disk reads and writes of various strides and lengths.

4. *Communication microbenchmarks* measure data communication operations such as message sends and receives or remove reads and writes of different types.

5. *Synchronization microbenchmarks* measure the performance of different types of synchronization operations, such as locks.

The developed cluster had distributed memory; therefore, only results from microbenchmarks 1, 3, 4 and 5 will be analyzed and discussed.
For measurement purposes, microbenchmarks are usually implemented as repeated sets of primitive operations (e.g. 1000 floating point operations on data in a row). They often have simple number of parameters that can be varied to obtain fuller characterization. For example, one can vary the amount of data to calculate, or change the number of processors processing the data.

2.5.2 Workloads

Workloads can be divided into three classes:

1. Kernels

2. Multiprogrammed workloads

3. Complete applications

Kernels are well-defined parts of real applications but are not complete applications themselves. Kernels usually provide computing facilities for applications but do not support any communications, or vice versa. Multiprogrammed workload tests involve running multiple applications on the same system simultaneously. The overall performance of the system is observed. The cluster discussed in this thesis is providing computing facilities for specialized engineering calculations. The main objective of the workload handling capability should be the performance evaluation of the machine when running engineering programs. Three popular applications were implemented and run on the cluster and the performance of the cluster was then observed. The three applications developed were: matrix multiplication, two-dimensional FFT and, finally, electric field approximator.

Matrix Multiplication

Matrix multiplication is a fundamental part of many complex science and engineering applications [23, p. 25]. The algorithm is relatively computationally intensive and is very often used to assess the performance of computer systems [10], [23], [51], [56], [70], [96].
The product of two matrices is represented as $[C] = [A][B]$, where the elements of $[C]$ are defined as [24]:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k}B_{k,j}$$

A sequential matrix multiplication algorithm might be implemented in the following way:

```
For $i = 1$ To m
    For $j = 1$ To l
        $C_{i,j} = 0$
        For $k = 1$ To n
            $C_{i,j} = C_{i,j} + A_{i,k} \times B_{k,j}$
        Next $k$
    Next $j$
Next $i$
```

Such code is well suited to a highly parallel MIMD design with processors powerful enough to carry out substantial computations on their own [6, p. 307]. Matrix multiplication is an inherently parallel algorithm with well-defined points of synchronization and is thus well suited to implementation on a cluster computer. Consider the multiplication of two $n \times n$ matrices performed by p processors. A balanced workload is achieved by allocating each processor a sub-problem of computing the $(\frac{n}{p} \times n) \times (n \times \frac{n}{p})$ submatrix of the problem. These operations may be carried out in parallel by p processors. The best solution of the multiplication of the two $n \times n$ matrices on $p < n^2$ processors is achieved in $O(\frac{n^3}{p})$, provided that communication time is much smaller than computation time [23, p. 32].
A distributed version of the matrix multiplication algorithm listed above was implemented and used to evaluate the cluster's performance.

FFT

Fourier transform is a powerful tool for many problems, and especially for solving various differential equations of interest in science and engineering [50, p 1]. The Fourier transform algorithm has a complexity of $O(n^2)$. The most popular implementations of the transform are based on an algorithm proposed by Cooley and Tukey. The so called fast Fourier transform, or FFT as we will refer to it, has a lower computational complexity of only $O(nlog(n))$. Similar to matrix multiplication the FFT algorithm is frequently used to measure the performance of computer systems [6], [51], [70], [101], [117].

2D-FFT. The two-dimensional Fourier transform is required in applications that involve two-dimensional data sets, such as image processing and geophysical analyses [44].

Let $[A]$ be an $L \times M$ 2-dimensional complex matrix. The $L \times M$ 2-dimensional transform of $[A]$ denoted by $\mathcal{F}(A_{L,M})$ is the $L \times M$ 2-dimensional array [B] defined by:

$$B_{r,s} = \sum_{m=0}^{M-1} \sum_{l=0}^{L-1} A_{l,m} e^{2\pi irl/L} e^{2\pi ism/M}$$

which can be written in a compact matrix notation:

$$[B] = \mathcal{F}(L)[A]\mathcal{F}(M)$$

This method is called *row-column* because it computes $[B]$ by a sequence of 1-dimensional finite Fourier transforms of the rows of $[A]$ followed by a sequence of 1-dimensional finite Fourier transforms of the resulting columns. The matrix $[B]$ is computed in two stages. First an intermediate matrix of Fourier transforms of the rows is computed, then a second series of Fourier transforms of the columns is performed on the resulting matrix. Computation of an $N \times N$ -point 2-D Fourier transform requires $2N$ complete, 1-D FFT calculations. The cost of such a computation would be $2N(Nlog(N))$ plus

the communication overhead. Communication during the distributed computation requires four transfers of the $N \times N$ matrix.
Consider the computation of 2-dimensional FFT on a $N \times N$ data matrix computed by p processors. A balanced workload is achieved by allocating each processor a sub-problem of computing the $(\frac{N}{p} \times N)$ submatrix of the problem. These operations may be carried out in parallel by p processors. The best solution of the computation is achieved in $O(\frac{2N(Nlog(N))}{p})$, plus the communication costs.

A popular FFT program suite [90] has been adapted and modified to compute the two dimensional FFT on the cluster computer and to evaluate the cluster's performance.

Electric Field Approximator

Numerous problems that arise in engineering can be visualized as a 2-dimensional grid where the values of the individual elements vary over time in response to the values of neighbouring elements. Examples of such problems include electric field intensity, thermal conduction, oceanographic simulation, and atmospheric modeling. Partial Differential Equations (PDE's), having well known solution techniques, can be expressed in a data parallel fashion using arrays to store a discretized representation of the problem [34]. Grid or mesh techniques are frequently used to approximate the states of continuous entities that behave in a wave-like or fluid fashion. Problems where each point in the grid has the same computational requirement are quite often called *uniform.* Partial Differential Equations are commonly used to solve uniform grid problems. Laplace's equation governs steady-state distribution of electrical potential on a plane [116]:

$$\nabla^2 u \equiv \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{2.1}$$

The derivatives can be replaced by the finite difference approximation:

$$f''(x) \approx \frac{1}{h^2}[f(x+h) - 2f(x) + f(x-h)] \tag{2.2}$$

Which yields the formula:

$$\nabla^2 \approx \frac{1}{h^2}[u(x+h,y) + u(x-h,y) + u(x,y+h) + u(x,y-h) - 4u(x,y)] \tag{2.3}$$

Setting $\nabla^2 = 0$ and $h = 1$ (grid granularity) produces an algorithm that can be used to calculate any value on the grid whose dimensions are $[x+1] \times [y+1]$:

$$u(x,y) \approx \frac{1}{4}[u(x+1,y) + u(x-1,y) + u(x,y+1) + u(x,y-1)] \tag{2.4}$$

A sequential mesh calculation algorithm might be implemented in the following way:

```
For $y = 1$ To m
    For $x = 1$ To l
        $A_{x,y} = (A_{x+1,y} + A_{x-1,y} + A_{x,y+1} + A_{x,y-1})/4$
    Next $x$
Next $y$
```

Depending on the size of the grid the above calculation might have to be repeated several times in order to achieve accurate results. The number of iterations required will vary from one grid problem to another.

Such code is frequently implemented on SISD machines, since the values of all data points depend on the values of all elements of the grid. The computation of $N \times N$ grid points requires N^2 floating point divisions and $3N^2$ floating point additions, and the execution times become very long when N is sufficiently large. The computational complexity of the algorithm $O(n^2)$ makes it an attractive problem to be implemented in a distributed environment.

Distributed implementation of this algorithm requires partitioning of the grid and assigning the partitions to every computer participating in the computation. This partitioning and assignment of the data is usually done by one machine, which is aware of all the machines participating in the computations. Since the computed data reside on machines physically distinct from each other, additional communications are also required in order to ensure correct grid values at the partition boundaries. The communications can either take place among the participating machines or they can be performed between the participants and the machine acting as a server. The later approach was chosen, as it is the server that assigns data to the participants. The server is also aware of the boundaries resulting from the partitioning of data. Communications can be performed either in a synchronous or an asynchronous manner. Since all participants had the same CPU and the number of data points required to compute, the grid values, at the boundaries is only $4N$ per participant, the synchronous type of communication was chosen and implemented.

Consider the calculations of the grid values of an $n \times n$ matrix performed by p processors. A balanced workload is achieved by allocating each processor a sub-problem of computing the $(\frac{n}{p} \times n)$ submatrix of the problem. These operations may be carried out in parallel by p processors. The best solution of the calculations of the grid values of $n \times n$ matrix on $p < n$ processors is achieved in $O(\frac{n^2}{p})$, provided that communication time is much smaller than

computation time.
A distributed version of the mesh algorithm listed above was implemented and used to evaluate the cluster's performance.

2.6 Modeling

The physical principles underlying the behaviour of most electronic devices are fairly complex, although the actual electrical behaviour of the device may be quite straightforward. Rather than attempt to relate physical effects of the device directly to network analysis, an intermediate step can be undertaken. This step is generally represented by the behaviour of the device by voltage-current or other appropriate components such as resistors, voltage or current sources or other ideal elements. In the latter case, the device is easily analyzed in terms of circuit theory.
For classical analysis the standard approach has traditionally been to apply an equivalent circuit to linear (small-signal) problems. Either a graphical analysis or piecewise linear analysis is applied to the solution of large signal circuits. Large-signal or nonlinear networks are often too difficult to analyze, and it is not uncommon to resort to several simplifying assumptions to obtain an approximate solution. Frequently, problems might require a large volume of calculations if a high degree of accuracy is to be maintained. Thus with manual analysis it is almost always necessary to simplify the device model to reduce the complexity of the overall circuit/system [31, p 221].
An attempt was made to produce several models that characterize/resemble the behaviour of the designed system. Modeling in this sense is the process that represents the electrical properties of a device or interconnected device by means of mathematical equations, circuit representation or tables. Complex

devices and large scale systems are characterized by *macromodels* that reflect their behaviour. Modeling at both levels, device and terminal, is equally important. Device level models are used for accurate analysis and design of smaller networks. Eventually, if these networks represent typical building blocks in larger systems, macromodeling is used to simplify the representation and speed up the analysis. Frequently, device information/behaviour will be obtained through a series of experiments and then the designer is faced with the task of implementing and constructing a model of the system from measured data. Physical device models usually involve a number of mathematical equations. Typical timing studies have shown that the major problem in analysis is in evaluating these complicated relationships. Further, most analysis methods also require derivatives of the model equations–a cumbersome and error-prone task for the designer. For these reasons, increasing use is being made of approximations of the model equations [136, p. 308].

2.6.1 Linear Model

Since resistors, capacitors, inductors, switches and ideal sources can be analyzed in an orderly manner, frequently an attempt is made to relate devices such as active circuits to these elements. The basic elements have known voltage-current characteristics that can be characterized by constant, time-invariant parameters. Many important applications require the device to operate over only a small area of the possible operating region within which the characteristics will be approximately linear. A disadvantage of this method is that once the device is modeled by constant-parameter elements, the area over which linear operation takes place is not apparent. Thus a given input signal, applied to the actual circuit, may be large enough to cause a highly distorted output, while the circuit model will predict a non distorted output. Some attention is then required in prescribing the limits over which a given model is valid, especially when automatic analysis programs are utilized [31, p 223].

2.6.2 Nonlinear Model

Simple circuits can be used to model complicated systems only when the system's operating region is small. Frequently the nonlinearities inherent in the device characteristics begin to distort the response of the actual system. In order to describe the system's nonlinearities one must resort to sophisticated modeling techniques. One method for analyzing nonlinear circuits is that of piecewise linear approximation. The nonlinear characteristics are averaged over the swing of interest and represented approximately by linear characteristics. A linear circuit model yielding the linearized characteristics can then be proposed. For devices passing from one operating region to another, a different, linearized equivalent circuit can be proposed for each region [31, p 229].

2.6.3 Discrete Systems

Discrete systems are difficult to model in a linear fashion. Frequently a transformation of the discrete system needs to take place before continuous modeling techniques can be used. In order to perform continuous modeling of the response of a system, one needs to analyze it as a system that changes in time. Example of such transformation can be a transformation of a computational task involving a series of computations whose length or complexity increases when the calculations of the last computation are complete. The total time required to perform the computations is the sum of the computation times of the varied sized problems. The system response is recorded at the end of each iteration and the data is plotted. The intervals at which the response is recorded increase with the increase of the data on which the system computes. Frequently only selected regions of the system response can be modeled with a satisfactory level of accuracy using one modeling technique. Often multiple models need to be devised to accurately model the entire response of a complex system.

Modeling the performance of distributed systems requires identification of critical phenomena affecting the response of the system. In distributed environments, the performance of the I/O and floating point components plays an important role. In this thesis we will attempt to identify and study the performance of the above identified, critical cluster components. Later we will attempt to develop a discrete and a continuous model of the designed system.

Chapter 3

Apparatus

The previous chapters specify that one of the objectives of the experiment was to develop a machine that would have a high performance to price ratio. In order to accomplish the task the cluster needed to be implemented using commodity commercially available parts. The following hardware components were identified as the absolute minimum:

- Cluster Server
- Several Cluster Members
- Network Connections

Adaptation of standard software, operating system and networking software was also identified as one of the factors that influenced the development of the cluster. Linux operating system was chosen as the development platform. All popular Linux distributions come with development tools such as compilers, debuggers and related literature. For details and history of Linux please refer to the appendix. The networking details of the implemented cluster will be addressed in the next chapter.

3.1 Server

The server computer was the only computer assembled from new parts. While it was not crucial to build a fast machine to coordinate cluster activities, it had to meet several requirements. The following services had to be provided by the server:

- Development platform (all code was compiled on the server)
- Network management:
  1. NFS server
  2. TFTP server
  3. BOOTP server
  4. DHCP server
  5. Telnet server
- Cluster coordination
- Experiment data collection and management

Several observations should be made at this point. Firstly, we realize that the server might potentially be required to perform several tasks simultaneously. A multiprocessor architecture, while not required, would help ensure the accuracy of the experiment results being recorded. Secondly, we observe that the server will be providing file services for several cluster members as well as for itself. A dedicated disk should be allocated to each task if possible. Lastly, we note that the server will be providing a variety of network services to the cluster members participating in the experiment. A fast Ethernet network card, possibly multiple cards, should be installed in the server.

3.1.1 Server Hardware

The server was assembled using commercially available parts which were purchased at a local computer store. The following is a list of components used to assemble the server:

- Dual Processor Pentium II/Pentium III Motherboard
- Two Intel Pentium III processors
- 128MB of SDRAM memory
- Two Ultra 2 SCSI hard discs
- AGP Video Card
- Fast Ethernet Network Card
- Tower Case

3.1.2 Server Software

The RedHat distribution of Linux was the operating system of choice. The RedHat flavour of Linux comes with several development tools. This distribution also has all networking software that was needed to set up a network management system required for the experiment.

```
make config
make dep
make clean
make bzImage
make lilo
make modules
make modules_install
```

Figure 3.1: SMP support kernel compilation

Kernel Configuration

Most Linux distributions do not provide a kernel that is multiprocessor aware. In order to enable the SMP support one needs to compile a custom kernel and enable several configuration options. The following options need to be selected while compiling [94]:

- Processor Type and Features:
  MTRR (Memory Type Range Register) support: ENABLED
  Symmetric multi-processing support: ENABLED

- General Setup:
  Advanced Power Management BIOS support: DISABLED
  RTC (Real Time Clock) support: ENABLED

Like most UNIX kernels, Linux kernel, is monolithic but it is possible to use kernel modules for device drivers. It is necessary to recompile all modules to enable the SMP support. Figure 3.1 lists the commands that need to be issued to compile and activate the SMP support on RedHat Linux operating system.

```
processor  :  0                      processor  :  1
vendor_id  :  GenuineIntel           vendor_id  :  GenuineIntel
cpu family :  6                      cpu family :  6
model      :  7                      model      :  7
model name :  Pentium III (Katmai)   model name :  Pentium III (Katmai)
stepping   :  3                      stepping   :  3
cpu MHz    :  451.026194             cpu MHz    :  451.026194
cache size :  512 KB                 cache size :  512 KB
...        :  ...                    ...        :  ...
bogomips   :  450.56                 bogomips   :  448.92
```

(a) Processor 1 (b) Processor 2

Figure 3.2: Listing of /etc/proc file

The multiprocessor kernel operation can be determined in two ways. Firstly, one can examine kernel messages at the boot when the kernel tries to detect all processors and activate them. Secondly, the information about the system's CPUs can be found by examining the contents of the /proc/cpuinfo virtual file. The information obtained from the /proc/cpuinfo file is shown in figure 3.2.

Development Software

RedHat 6.1 Linux comes bundled with program development software (compilers and libraries) as well as the documentation useful for programmers and system developers. A list of packages installed on the server can be found in the appendix.

3.2 Cluster Member

In order to built an inexpensive cluster one could use computers that have been previously deployed and are reaching the end of their productive life cycle. Such machines are often found in computer laboratories or libraries. Several Pentium class computers were obtained and modified to participate in the experiments.

3.2.1 Hardware Configuration

The hardware requirements for a cluster participant were very modest. It was determined that a computer consisting of the parts listed below would fully suffice:

- CPU: Pentium class
- Memory: 16MB or more
- Floppy Drive: 3.5", 1.44MB
- NIC: Ethernet 10MBit or 100MBit
- Hard Drive: optional
- Video: optional
- Keyboard: optional

All cluster member computers came with local hard disks, video cards and keyboards but these items were not directly utilized. None of the computers contained a local copy of the operating system and the computers booting were not booting of the local hard drive. All machines were booting of the server via network. However, the hard drive was utilized. In order to minimize network traffic and avoid system swapping via network, the swap partition of each cluster member was mounted on its local hard drive. The remaining peripherals (video cards and keyboards) were not used after the initial setup was performed. Most modern computers can be "told" to operate without a video card or a keyboard by modifying settings in the computer's BIOS.

3.2.2 Software Configuration

As stated in the previous section, each cluster member was booting of the cluster server via the network. In order to accomplish this task two issues need to be addressed. Firstly, the booting computer has to be told, and be able, to use the network card as its booting device. Secondly, a customized version of the operating system needs to be available to the booting computer at the time of the boot.

Network Boot

The first task can be accomplished via the means of a BOOT ROM in the network card of a cluster member. The second approach is to provide the computer with an image of the operating system on a floppy disk. Each approach has its benefits, but it also has some drawbacks.

Creating BOOT ROMS requires that one obtain BOOT ROM images of each network card used in the cluster. Such images are often subject to copyright agreements and are in general difficult to obtain. The second problem with such an approach is that one needs to physically remove the ROM chip from the network interface card when one does not want to boot from the network. On a developmement system one quite often needs to modify the kernel image of a cluster member. Each such a modification would require the creation of a new set of boot floppies; booting from a floppy drive is also much slower than booting from a hard drive or via the network.

A feasible compromise would be to provide the BOOT ROM code to the computer on a floppy disk. Several free (Linux based) software packages provide BOOT ROM images that could be burned into an eeprom and then used for network booting. The etherboot package allows developers to test BOOT ROM images prior to EEPROM burning. The etherboot software package allows for the creation of boot floppy disc containing only the BOOT ROM code (8KB). This was a perfect compromise between a commercial BOOT ROM and a fully blown OS image on a floppy. The BOOT ROM code is loaded in less than a second and then the network boot takes place. Changes to the cluster member's kernel can be made in one central location and they will be picked up by booting cluster members. If a computer participating in the experiment for some reason needs to be used for other tasks, it can be used without the need to open the case and remove the BOOT ROM.

The boot process can be divided into the following steps:

1. Power On System Test (POST),
2. Boot device identification,
3. Loading boot code,
4. Location of the Operating System files,
5. Loading of system files and mounting file systems.

The boot is accomplished in the following manner: first a boot floppy is located and the BOOT ROM code is loaded; next the booting computer broadcasts requests for an image of the OS files. When such a broadcast is detected by the cluster server, the server tells the client where it can locate the image of the kernel. After the kernel image is loaded, the control of the boot processes is passed to the kernel. The kernel identifies the hardware configuration of the machine and recognizes the fact that it needs to finish the boot process using the network. Another broadcast request is sent inquiring about the location of the system files and remote file systems. After this information is provided by the server, the cluster member finishes loading system files, mounts the swap partition on the local hard drive, remote file systems on the server, and the boot is complete. Any files required by the computer after the boot are loaded from file systems mounted from the server.

Kernel Configuration

A custom kernel needs to be built in order to support diskless configuration of a cluster member. The following options were specified during the kernel configuration procedure:

- Filesystems:
  Second extended fs support
  Network File Systems:
  - NFS filesystem support: ENABLED
  - Root file system on NFS: ENABLED
- Network Device Support:
  Ethernet (10 or 100Mbit):
  - Cluster Member Network Card Type: COMPILED-IN (not as module)

Diskless Client

As stated in the previous section, none of the computers participating in the experiment had a local copy of the operating system. The operating system was loaded via network and file systems were mounted on the server. The listing below shows all file systems mounted by a cluster member:

1. `asus2p3:/tftpboot/cm3 on / type nfs (rw,rsize=8192,wsize=8192,timeo=14,intr)`
2. `none on /proc type proc (rw)`
3. `none on /dev/pts type devpts (rw,gid=5,mode=620)`
4. `asus2p3:/tftpboot/usr on /usr type nfs (rw,rsize=8192,wsize=8192,timeo=14,intr,addr=192.193.1.250)`
5. `asus2p3:/home/development on /development type nfs (rw,rsize=8192,wsize=8192,timeo 14,intr,addr=192.193.1.250)`

The first entry shows that the root of computer CM3 is mounted on computer named asus2p3 (the server) in the /tftpboot/cm3 directory. The next two entries apply to virtual file systems that are not mounted physically. The fourth entry shows a common /usr file system that is shared among all cluster participants. Finally, the fifth entry shows that the working directory of the currently logged user is mounted on asus2p3 in the /home/development directory.

3.3 Network Connection

The most common type of networking technologies used today is Ethernet. Ethernet is also the least expensive networking hardware available today. While Ethernet technology does not scale and reaches its peak efficiency at 60% medium utilization, it works well on medium sized networks [128]. There are three types of Ethernet hardware available on the market today. The first and most common type is 10MBit Ethernet. 10MBit Ethernet hardware runs at 10MHz and delivers transfer rates around 1MB/sec. The second type, so called "Fast Ethernet" runs at 100MHz and delivers transfer rates around 10MB/Sec. The newest type of Ethernet is "Gigabit" Ethernet. Gigabit Ethernet runs at 1GHz and delivers transfer rates close to 100MB/sec. Gigabit hardware is still very expensive and the distance between nodes cannot be very large [25, p 132].
The conducted research examined the applicability of Ethernet and fast Ethernet technologies in cluster topology.

3.3.1 NICs

Two types of Network Interface Cards (NICs) were used. Initially each cluster member had a 10MBit Ethernet card. Later on tests were conducted on 100MBit Ethernet.

3.3.2 Hubs

The 10MBit topology was implemented using a 10MBit hub. The 100MBit topology was implemented using a 100MBit. The use of a switch was considered. An Ethernet switch allows for creation of virtual connections between two machines exchanging information and that conversation is isolated (filtered) from the rest of the computers present on the network. The performance of such a network is greater than that of a hub based network because there are fewer packet collisions. A hub based implementation does not allow for creation of isolated virtual circuits. Any broadcasting machine is "heard" by all computers connected; when more than two computers are exchanging information, packet collisions contribute to overall network performance [86]. The cluster's network topology is illustrated in figure 3.4. We see that the server has only one network connection. All information sent to cluster members is carried through that connection. It would be impossible to create multiple isolated circuits between the server and cluster members. Thus the cluster would not benefit greatly from the use of a switch.

3.3.3 Network Topology

Ethernet based networks are implemented in two fashions. The original coax based Ethernet was implemented using bus topology. All participating computers connected to a common bus and broadcast information on the bus. While still common, the coax based Ethernet is being replaced by twisted pair based Ethernet, which is implemented using star topology. In star topology all network participants are connected to a hub or a switch using Category 3 or higher twisted pair cables. One issue worth noting is the fact that the coax based Ethernet is limited to 10MHz, hence it cannot be used with 100MBit or faster network interface cards. The star topology was chosen to implement the cluster's network infrastructure. The bus and star topologies are illustrated in figures 3.3 and 3.4 respectively.

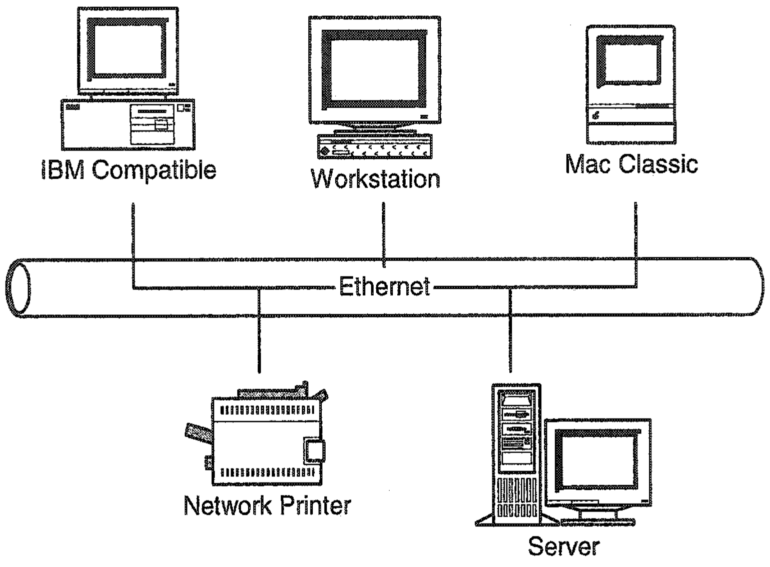


Figure 3.3: Bus Topology

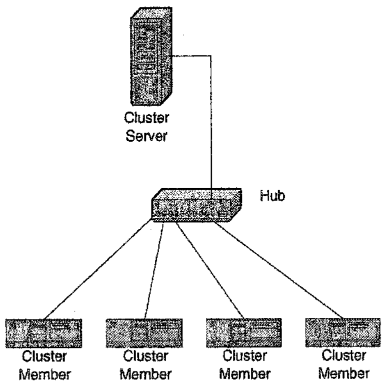


Figure 3.4: Star Topology

3.4 Scalability

It was important to design a cluster that could scale, or whose performance would increase with the number of nodes present. While Ethernet networks benefit from switched technologies, the only communications that take place in the cluster are the communications between the server and each cluster member. In order to create N isolated circuits the server would need N network cards. The current PC architecture imposes a limit on how many expansion slots can be present in a PC. Usually a PC will have four expansion slots on one bus. Some high end servers have two buses, but that would still put a limit on the number of network cards present in a system.

In order to avoid that physical limitation, it was decided to use one network card in the server and to observe its scalability.

Chapter 4

Cluster Network Implementation

The previous chapter illustrated the topology of the implemented cluster. The following chapter will provide the reader with additional implementation details.

4.1 Network Connectivity

The main idea behind the implemented cluster is to utilize the individual computing facilities of machines that can be accessed remotely via network. The network is the only connection that exists between the cluster members and the server.

4.1.1 Client–Server Computing

As stated previously, the computers participating in the cluster are not aware of each other. They are not even aware of the cluster server. As far as the cluster member is concerned it only provides computing facilities to anybody that requests them. In the implemented cluster, cluster members are actually computing servers that can perform some computations on the data sent to them. The results of the computations are sent back to the computer the data originated from (client). This model of computing is called Client–Server computing. Some clarification is needed at this point. Cluster member computers are computing servers. The cluster server is a clever client that divides its computing problem evenly among computing servers. The clever client is capable of dividing and coordinating the activities of the servers. As far as the client is concerned, it can send its data to one or more servers and collect the results. The computing servers do not care where the data comes from. As long as the client(s) follow a protocol of the server, the server will receive the data and perform computations on them. The following is the protocol designed for cluster matrix multiplication:

Cluster Member	**Cluster Server**
(Server)	**(Client)**
Wait for connection	Send Matrix Dimensions
Receive Matrix Dimensions	Wait for Confirmation of Matrix Dimensions
Wait for Matrix 1	Send Matrix 1
Receive Matrix 1	Wait for Confirmation
Wait for Matrix 2	Send Matrix 2
	Wait for Confirmation
	Wait for Results
Send Results	Receive Results

Similar protocol was designed for cluster FFT:

Cluster Member (Server)	**Cluster Server (Client)**
Wait for connection	Send Matrix Dimensions
Receive Matrix Dimensions	Wait for Confirmation of Matrix Dimensions
Wait for Matrix	Send Matrix
Receive Matrix 1	
	Wait for Results
Send Results	Receive Results

The client–server architecture is very common. Quite often servers are powerful computers performing computations on behalf of less powerful clients. It is common to see servers serving multiple clients simultaneously. In our case we have several computing servers utilized simultaneously by one client. Figures 4.1 and 4.2 illustrate both concepts clearly.

4.1.2 OS Support

One of the desired features of cluster computing is the fact that it can be performed on machines that are completely independent. Cluster members do not have to have the same hardware architecture or run the same operating system. There are a few requirements that need to be satisfied.

Network Support

Each machine participating in the cluster needs to be able to communicate with the cluster server. The operating system needs to provide a means for conducting communications. It was decided that the commonly used TCP/IP protocol should be used as the lingua franca of the cluster. Any computer run-

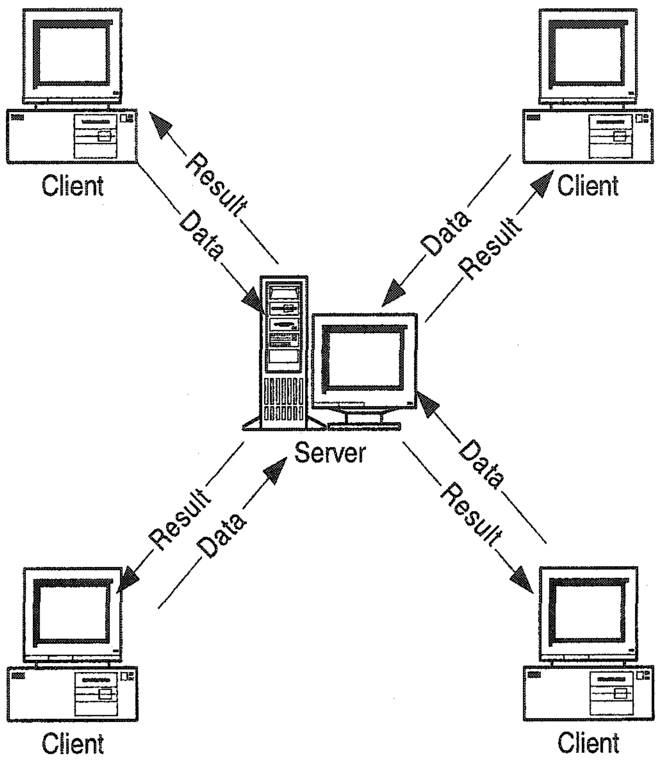


Figure 4.1: Typical Client–Server

ning an operating system that provides support for TCP/IP communications can be used to participate in the cluster.

Binary Compatibility

ANSI C programming language was used to develop the code and socket communications were used to pass messages between the cluster participants. GNU C compiler was used to compile the code for cluster members and the server. GNU C compiler has been ported to many operating systems. The cluster member code should run without modifications on any platform to which the GNU C compiler has been ported.

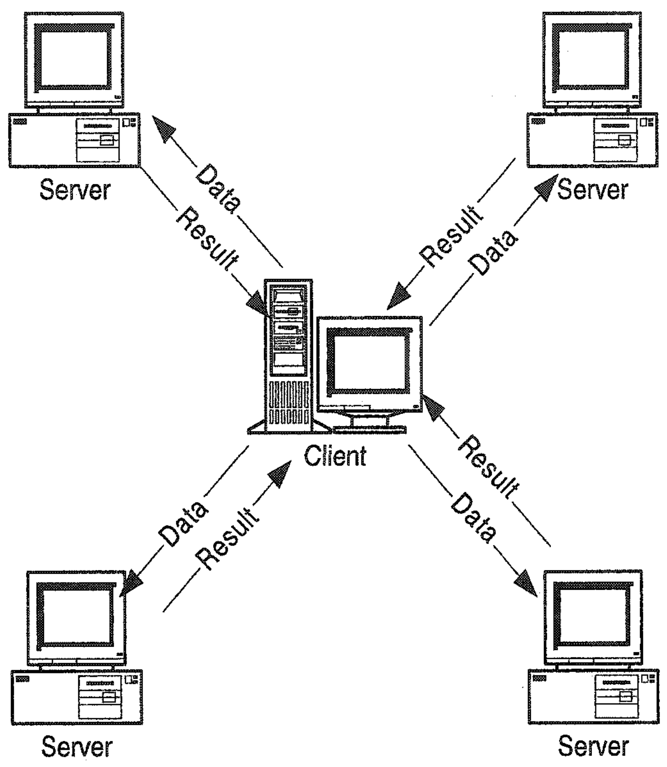


Figure 4.2: Implemented Cluster Client–Server

4.2 Network Services

The previous chapter addressed the network configurations that needed to be performed on the client side. The following sections will address the implementation details on the server side. The server needs to provide several services for cluster participants. While it is not absolutely necessary that all of these services be implemented, the services listed below allowed seamless client addition and automated the cluster administration. The following services were configured:

- DHCP and BOOTP: IP management
- TFTP: Network Boot
- NFS: Network File System

The server provides three basic services for the clients. Firstly and most importantly, it gives them an identity. Secondly, it tells them where to load the image of the operating system from and finally, it provides them with a working file system. Each of the services is explained in the sections below.

4.2.1 DHCP and BOOTP

The software used to perform network IP management was Internet Software Consortium DHCP Server, *dhcpd.* The software implements Dynamic Host Configuration Protocol (DHCP) and Internet Bootstrap Protocol (BOOTP). The DHCP protocol allows a host unknown to the network administrator to be automatically assigned a new IP address out of a pool of IP addresses for its network. In order for this to work, the network administrator allocates address pools in each subnet and enters them into the dhcpd.conf file. On startup, *dhcpd* reads the dhcpd.conf file and stores a list of available addresses on each subnet in memory. When a client requests an address using the DHCP protocol, *dhcpd* allocates an address for it. Each client is assigned a lease, which expires after an amount of time chosen by the administrator. Before leases expire, the clients to which leases are assigned are expected to renew them in order to continue to use the addresses. Once a lease has expired, the client to which that lease was assigned is no longer permitted to use the leased IP address [138].

The *dhcpd* software needs to be configured before it can serve clients. The configuration settings for *dhcpd* are stored in the /etc/dhcpd.conf file:

```
subnet 192.193.1.0 netmask 255.255.255.0 {
range 192.193.1.80 192.193.1.90;
default-lease-time 36000;
max-lease-time 72000;
}
group{
filename "/tftpboot/eeprokernel";
server-name "asus2p3";
next-server 192.193.1.250;
option domain-name-servers 129.100.2.12;
option domain-name "uwo.ca";
host cm1 {
hardware ethernet 00:D0:B7:BD:49:8A;
fixed-address 192.193.1.71;
option host-name "cm1";
}
⋮
host cm6 {
hardware ethernet 00:D0:B7:BD:90:4D;
fixed-address 192.193.1.76;
option host-name "cm6";
}
}
```

The first part of the configuration file contains a range of IP addresses that it can give out to any client that requests them. In that section one also specifies the lease time of the IP addresses given out. Before the lease expires the client

will need to renew the IP address it leases or obtain a new one.
The second section specifies global data for a set of hosts. In our case these data belong to cluster members participating in the experiment (CM1–CM6). The first line tells clients where to find an image of the kernel file to be loaded. The second line tells clients the name of the server. The remaining options in the global section tell clients additional information they might need.
The third section contains information organized in groups for each and every host participating in the cluster. Each host has a network card with a unique hardware address assigned to it by the card's manufacturer. The first line in every group identifies the hardware address of the cluster participant. The IP address of the participant is found on the second line. Finally, its name is listed on the third line.
Consider a computer attempting to boot using network facilities. The computer loads the boot code from its BOOT ROM and then it attempts to find a server that contains an image of the OS the client needs to load. The client broadcasts requests to DHCP servers present on the network. If a DHCP server is present on the network, it will answer and offer the client an IP address together with the information specifying the location of the kernel image. If the client accepts the offered IP, it sends an acknowledgment to the server confirming the acceptance of the lease.

Demonstration of conversation between the server and the client (CM4):

```
[root/@asus2p3 /root]# dhcpd -d
Internet Software Consortium DHCP Server 2.0
Copyright 1995, 1996, 1997, 1998, 1999 The Internet Software
Consortium.
All rights reserved.

Please contribute if you find this software useful.
For info, please visit http://www.isc.org/dhcp-contrib.html

Listening on LPF/eth0/00:90:27:77:41:8a/192.193.1.0
Sending on LPF/eth0/00:90:27:77:41:8a/192.193.1.0
Sending on Socket/fallback/fallback-net
DHCPDISCOVER from 00:d0:b7:bd:90:4d via eth0
DHCPOFFER on 192.193.1.74 to 00:d0:b7:bd:90:4d via eth0
DHCPREQUEST for 192.193.1.74 from 00:d0:b7:bd:90:4d via eth0
DHCPACK on 192.193.1.74 to 00:d0:b7:bd:90:4d via eth0
```

4.2.2 TFTP

When the client receives a valid IP address together with the information where to find the kernel image, it needs to load and execute it. The protocol used to load the kernel is TFTP or Trivial File Transfer Protocol. TFTP is a light version of the File Transfer Protocol or FTP. TFTP is not a secure protocol and it does not provide authentication. TFTP runs on top of User Datagram Protocol (UDP) instead of Transmission Control Protocol (TCP). UDP was chosen instead of TCP for simplicity. The implementation of UDP is much simpler than that of TCP and the code can fit easily on a BOOT ROM. Because UDP is a block oriented, as opposed to a stream oriented, protocol, the transfer is performed block by block. A typical conversation between a cluster member and the server is illustrated in the dialogue below:

```
CM: Give me block 1 of /tftpboot/eeprokernel
CS: Block 1 of /tftpboot/eeprokernel
CM: Give me block 2 ...
```

The conversation is carried on until the entire image of the kernel is transferred. Handshaking is a simple acknowledgment of each block scheme, and packet loss is handled by retransmit on timeout. When all blocks have been received, the network boot ROM hands control to the operating system image at the entry point [39].

4.2.3 NFS

When the OS kernel boots it needs to mount a root file system. The cluster was implemented in such a way that each cluster member mounted a root file system from the server. Thus it always had updated binaries and all cluster member files were up to date. The protocol used to provide root file systems for cluster members was Network File System or NFS. After the kernel is loaded and the root over NFS option is compiled into the kernel (see section 3.2.2) the booting computer can mount a file system residing on the server. The server hosts a separate OS image for each cluster member. The server also allows each cluster member to mount a common directory, which is used to host binaries of programs run by cluster members. The list below contains all server directories that can be accessed using NFS [74].

```
[damian@asus2p3 damian]$ cat /etc/exports
/home/development *.uwo.ca(rw)
/tftpboot/cm1 *.uwo.ca(rw,no_root_squash)
/tftpboot/cm2 *.uwo.ca(rw,no_root_squash)
/tftpboot/cm3 *.uwo.ca(rw,no_root_squash)
/tftpboot/cm4 *.uwo.ca(rw,no_root_squash)
/tftpboot/cm5 *.uwo.ca(rw,no_root_squash)
/tftpboot/cm6 *.uwo.ca(rw,no_root_squash)
/tftpboot/usr *.uwo.ca(ro,no_root_squash)
```

The first entry specifies a common directory that is accessible freely by anybody. The next four entries are unique to each cluster member participating in the experiment. They contain root file systems of a particular computer (CM1–CM6). Finally, the last entry lists a common /usr folder that is mounted as "read only". The /usr folder on a Linux system contains system files that do not need to be modified by users.

Chapter 5

Cluster Applications

Let us consider the following problem. Suppose we want to evaluate the value of e^x using the following formula:

$$e^x = 1 + \frac{x}{1!} + \frac{x}{2!} + \frac{x}{3!} \cdots + \frac{x}{n!} \tag{5.1}$$

Given the values of x (power) and n (desired accuracy) we could evaluate the value of e^x using the algorithm listed in figure 5.1. We can easily see that the

```
E=1
For i=1 To n
  E=E+ x/i!
Next i
```

Figure 5.1: Exponent evaluation serial algorithm

value of e^x is the sum of independently calculated discrete fractions of x and $i!$. We could easily distribute the task among remote computers and collect their individual results to produce the value of e^x. Figure 5.2 demonstrates how we can rewrite the serial algorithm in such a way that it could be used to calculate any part of the series. This program could run on any computer

```
E=0
For i=Min To Max
  E=E+x/i!
Next i
```

Figure 5.2: Parallel exponent evaluation client

participating in the computations. We would then need some coordinating computer that would schedule those computation on remote computers. A program run by the coordinator is listed in figure 5.3. The computing task

```
E=1
For i=1 To NumberOfClients
  E[i]=CalculateE(Min(i), Max(i))
Next i
For i=1 To NumberOfClients
  E=E+E[i]
Next i
```

Figure 5.3: Parallel exponent evaluation server

would be performed in parallel by all (N) computers participating in the computations. The overall computing time would be reduced and, depending on the nature of the problem, a potential speed-up of N could be achieved.
In order to evaluate the functionality and applicability three engineering applications were developed and run on the cluster. Implementation details will follow in the sections below.

5.1 Matrix Multiplication

Matrix multiplication algorithm is a CPU intensive task, hence a good candidate for performance evaluation of shared and distributed memory parallel computers [51].
In section 2.5.2 the product of two matrices was defined as $[C] = [A][B]$ and

the elements of $[C]$ were defined as:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}$$

where n is the column dimension of $[A]$ and the row dimension of $[B]$. That is, the C_{ij} element is obtained by adding the product of individual elements from the i^{th} row of the first matrix $[A]$ by the j^{th} column from the second matrix $[B]$ [24, pp 206–207]. The above definition states that the multiplication of two matrices can only be performed if the first matrix has as many columns as the number of rows in the second matrix. Thus, if $[A]$ is an $m \times n$ matrix $[B]$ could be an $n \times l$ matrix. The resulting $[C]$ matrix would have dimension of $m \times l$.

5.1.1 Sequential Algorithm

A sequential matrix multiplication algorithm was presented in section 2.5.2. We reproduce it here for reference. The algorithm uses three nested loops that traverse each row of matrix $[A]$ and each column of matrix $[B]$. The algorithm is illustrated in the pseudocode listed in figure 5.4.

```
For i=1 To m
  For j=1 To l
    For k=1 To n
      C[i][j]=C[i][j] + A[i][k] x B[k][j]
    Next k
  Next j
Next i
```

Figure 5.4: Sequential matrix multiplication algorithm

5.1.2 Parallel Algorithm

The sequential algorithm described above performs $m \times l \times n$ independent multiplications. The order in which each set of multiplications takes place does not affect the final result. The result could be obtained by performing the multiplications in parallel by one or more independent processors. Each of the processors would only need to have access to the particular row of matrix $[A]$ and to the corresponding column of matrix $[B]$ as well the location where the result should be stored. Obviously, this is not an optimal way to perform a matrix-matrix product in parallel; however, it results in a good illustration of the concept. The data are replicated to all participating processors, as it is quite often the case in many parallel calculations that some data items are needed in all processors. Replication of this data is more efficient than inter processor communications [71].

Consider two 2×2 matrices:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

The result of $[A] \times [B]$ could be obtained by performing computations on two independent processors:

Processor 1 **Processor 2**

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

The results of the independent computations can be then combined into one resulting matrix:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

This characteristic can be utilized to implement a cluster matrix multiplication algorithm.

5.1.3 Cluster Implementation

Consider the following cluster infrastructure. There exist N independent computing entities *(Cluster Members or CM)* capable of performing matrix multiplications on arbitrarily sized matrices $[A]$ and $[B]$.

There exists a supervising computing entity *(Cluster Server or CS)* which is coordinating any computing activities in the cluster. The CS is aware of each and every CM available for computations. The CS divides the computational task evenly among all CM's. This means that data are partitioned and sent to all CM's.

Each CM is waiting for data to compute on; when it receives the data (two matrices), it performs the multiplication of the two matrices. The results of the computation are sent to the computer where the data originated from (CS).

The CS receives all results and combines them into one logical entity that could be stored for later analysis.

Cluster Server Pseudo Code:

```
Read Matrix $[A]_{n,n}$
Read Matrix $[B]_{n,n}$
For $i = 1$ To N
    Connect to Cluster Member $i$
    Send $[A]_{n/N,n}$
    Send $[B]_{n,n}$
    Disconnect from Cluster Member $i$
Next $i$
For $i = 1$ To N
    Connect to Cluster Member $i$
    Receive $[C]_{n/N,n}$
    Disconnect from Cluster Member $i$
Next $i$
Store $[C]_{n,n}$
```

Cluster Member Pseudo Code:

```
Do
    Listen for Connection from Cluster Server
    Connect to Cluster Server
    Read Matrix $[A]_{m,n}$
    Read Matrix $[B]_{n,n}$
    Multiply $[A]_{m,n}[B]_{n,n}$
    Connect to Cluster Server
    Send Result $[C]_{m,n}$
    Disconnect from Cluster Server
End Do
```

To demonstrate the above algorithms we will perform a multiplication of two 4×4 matrices on a cluster with four CM's. Consider two matrices:

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} B = \begin{bmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ B_{31} & B_{32} & B_{33} & B_{34} \\ B_{41} & B_{42} & B_{43} & B_{44} \end{bmatrix}$$

The CS needs to partition the data and send it to the participating CMs. Each CM will receive one row of $[A]$ and the entire matrix $[B]$:

CM 1:

$$A1 = \square \quad A_{11} \quad A_{12} \quad A_{13} \quad A_{14} \Big], B = \begin{bmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ B_{31} & B_{32} & B_{33} & B_{34} \\ B_{41} & B_{42} & B_{43} & B_{44} \end{bmatrix}$$

CM 2:

$$A2 = \square \quad A_{21} \quad A_{22} \quad A_{23} \quad A_{24} \Big], B = \begin{bmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ B_{31} & B_{32} & B_{33} & B_{34} \\ B_{41} & B_{42} & B_{43} & B_{44} \end{bmatrix}$$

CM 3:

$$A3 = \square \quad A_{31} \quad A_{32} \quad A_{33} \quad A_{34} \Big], B = \begin{bmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ B_{31} & B_{32} & B_{33} & B_{34} \\ B_{41} & B_{42} & B_{43} & B_{44} \end{bmatrix}$$

CM 4:

$$A4 = \square \quad A_{41} \quad A_{42} \quad A_{43} \quad A_{44} \Big], B = \begin{bmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ B_{31} & B_{32} & B_{33} & B_{34} \\ B_{41} & B_{42} & B_{43} & B_{44} \end{bmatrix}$$

Each CM will then multiply the matrices it has received and then send the results to the CS:

CM 1:

$$C1 = \overset{\square}{} \; C_{11} \;\; C_{12} \;\; C_{13} \;\; C_{14} \Big]$$

CM 2:

$$C2 = \overset{\square}{} \; C_{21} \;\; C_{22} \;\; C_{23} \;\; C_{24} \Big]$$

CM 3:

$$C3 = \overset{\square}{} \; C_{31} \;\; C_{32} \;\; C_{33} \;\; C_{34} \Big]$$

CM 4:

$$C4 = \overset{\square}{} \; C_{41} \;\; C_{42} \;\; C_{43} \;\; C_{44} \Big]$$

The CS will assemble the individual results into one matrix:

$$\left(\begin{array}{cl} \square & \\ & C1 \\ \square & \\ & C2 \\ \square & \\ & C3 \\ \square & \\ & C4 \end{array} \right] \Bigg) \Longrightarrow \begin{bmatrix} C_{11} & C_{12} & C_{13} & C_{14} \\ C_{21} & C_{22} & C_{23} & C_{24} \\ C_{31} & C_{32} & C_{33} & C_{34} \\ C_{41} & C_{42} & C_{43} & C_{44} \end{bmatrix}$$

5.1.4 Concluding Remarks

The algorithm has been implemented in the C programming language (see the listings in the Appendix). The correctness of its operation was tested by running several multiplications of a randomly generated matrix by an identity matrix. We know that the result of any matrix multiplied by an identity matrix is the original matrix [67, Page 713]:

$$[A][I] = [A]$$

Where $[A]$ is any matrix and $[I]$ is a square matrix, all of whose elements are 0 except for the diagonal elements which are 1:

$$I_{4,4} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The results of the multiplication were then compared to the original random matrix. The distributed matrix multiplication worked correctly in all cases.

5.2 2DFFT

We stated in section 2.5.2 that the 2-dimensional finite Fourier transform can be written as a two dimensional tensor product whose factors are 1-dimensional finite Fourier transforms. Let $[A]$ be an $L \times M$ 2-dimensional complex matrix. The $L \times M$ 2-dimensional transform of $[A]$ denoted by $\mathcal{F}(A_{L,M})$ is the $L \times M$ 2-dimensional array [B] defined by:

$$B_{r,s} = \sum_{m=0}^{M-1} \sum_{l=0}^{L-1} A_{l,m} e^{2\pi i r l/L} e^{2\pi i s m/M}$$

Which can be written in a compact matrix notation:

$$[B] = \mathcal{F}(L)[A]\mathcal{F}(M)$$

This method is called *row-column* because it computes $[B]$ by a sequence of 1-dimensional finite Fourier transforms of the rows of $[A]$ followed by a sequence of 1-dimensional finite Fourier transforms of the resulting columns. The matrix $[B]$ is computed in two stages. First an intermediate matrix of Fourier transforms of the rows is computed, then a second series of Fourier transforms of the columns is performed on the resulting matrix.

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{bmatrix} \Rightarrow \begin{bmatrix} F1 : A_{11} & A_{12} & \cdots & A_{1n} \\ Fn : A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ Fn : A_{n1} & A_{n2} & \cdots & A_{nn} \end{bmatrix} \Rightarrow \begin{bmatrix} F1: & F2: & \cdots & FN: \\ A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{bmatrix}$$

5.2.1 Sequential Algorithm

2-dimensional FFT is quite often implemented using a brute force sequential method. The target matrix is handled on row by row and column by column basis. The algorithm uses two simple loops that traverse every row and every column of the matrix. Such an implementation is illustrated in the pseudocode below:

```
Read Matrix $[A]_{n,m}$
For $i = 1$ To n
    $FFT(A_{i,m})$
Next $i$
For $i = 1$ To m
    $FFT(A_{n,i})$
Next $i$
Store $[A]_{n,m}$
```

5.2.2 Parallel Algorithm

This algorithm is very simple and works very well on a single processor. In order to implement this algorithm in a parallel manner several issues need to be addressed. Computers store data in memory in a sequential manner. Multidimensional data structures such as arrays are always mapped onto a continuous set of memory locations. For example, a 4×4 array will be stored in sixteen consecutive memory locations. If we assume that each array element requires one byte of storage and that the first array element is stored at memory location M, then the first element of the second row is stored at $M + 4$. The first element of the third row is stored at $M + 8$ and the first element of the fourth row is stored at $M + 12$. The following formula is used to translate high level transcript notation of the element $a[i][j]$ or matrix $A_{x,y}$:

$$a[i][j] = \text{Mem Location}(a[0][0]) \times (x \times i) + j$$

where $a[i][j]$ is the value we want to access and $a[0][0]$ is the memory location of the first element of the $A_{x \times y}$ matrix. This of course presents a problem when a set columns is sent to a remote machine with its own local memory. The set of columns would be mapped into a set of rows resulting in computations on the wrong data.

In order to avoid this problem the following solution is proposed. The data resulting from row FFT computations would be "rotated" in such a way that columns would become rows and vice versa:

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{bmatrix} \mathit{rotate} \Rightarrow \begin{bmatrix} A_{n1} & \cdots & A_{21} & A_{11} \\ A_{n2} & \cdots & A_{22} & A_{12} \\ \vdots & \vdots & \vdots & \vdots \\ A_{nn} & \cdots & A_{2n} & A_{1n} \end{bmatrix}$$

When data are rotated we can use the same algorithm to perform the FFT on both rows and columns without compromising the integrity of the data.

We recognize that the final result depends on the FFT's performed on all rows or columns of the matrix; however, FFT's of each row or column can be performed independently from each other. This observation leads us to believe that we could perform FFT's on distinct rows or columns simultaneously, independently from each other. Consider a 2-dimensional Fourier transform of a 2×2 matrix $[A]$:

$$\mathcal{F}(A_{2\times 2}) = \mathcal{F}\left(\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}\right)$$

Suppose we could use two independent processors to perform the transform $\mathcal{F}(A_{2\times 2})$. First we could use each processor to perform 1-dimensional Fourier transform on one distinct row of the matrix $[A]$:

Processor 1

$$\mathcal{F}\left(\begin{bmatrix} A_{11} & A_{12} \end{bmatrix}\right)$$

Processor 2

$$\mathcal{F}\left(\begin{bmatrix} A_{21} & A_{22} \end{bmatrix}\right)$$

Before computing FFT on the columns of $[A]$ we need to collect the results and combine them into an intermediate matrix $[A']$. The intermediate result matrix $[A']$ has to be then rotated in such a way that the columns become rows:

$$Rotate\left(\begin{bmatrix} A'_{11} & A'_{12} \\ A'_{21} & A'_{22} \end{bmatrix}\right) \Longrightarrow \begin{bmatrix} A'_{11} & A'_{21} \\ A'_{12} & A'_{22} \end{bmatrix}$$

Then we could use each processor to perform 1-dimensional Fourier transform on one distinct row of the matrix $[A']$:

Processor 1

$$\mathcal{F}\left(\begin{bmatrix} A'_{11} & A'_{21} \end{bmatrix}\right)$$

Processor 2

$$\mathcal{F}\left(\begin{bmatrix} A'_{12} & A'_{22} \end{bmatrix}\right)$$

We again collect the results and combine them into one matrix $[A'']$. We could then leave the results in that form or rotate the result matrix $[A'']$ back to the original form:

$$Rotate\left(\begin{bmatrix} A''_{11} & A''_{21} \\ A''_{12} & A''_{22} \end{bmatrix}\right) \Longrightarrow \begin{bmatrix} A''_{11} & A''_{12} \\ A''_{21} & A''_{22} \end{bmatrix}$$

The resulting matrix $[A'']$ would then contain the result of a 2-dimensional Fourier transform of $[A]$.

5.2.3 Cluster Implementation

Consider the following cluster infrastructure. There exist N independent computing entities *(Cluster Members or CM)* capable of performing 1-dimensional fast Fourier transform (FFT) on arbitrarily sized matrix $[A]$ whose dimensions are a power of 2.

There exists a supervising computing entity *(Cluster Server or CS)* which is coordinating any computing activities in the cluster. The CS is aware of each and every CM available for computations. The CS divides the computational task evenly among all CM's. This means that data are partitioned and sent to all CM's.

Each CM is waiting for data to compute on; when it receives the data (set of rows) it performs an FFT on every row of the matrix. The results of the computation are sent to the computer where the data originated from (CS).

The CS receives all results, combines them into one logical entity, and reorganizes the results in such a way that another series of 1-dimensional FFT's could be performed. The reorganized matrix is partitioned and each partition (set of rows) is sent to all CM's for computations.

The results are again combined into one logical entity and stored for later analysis.

Cluster Server Pseudo Code:

```
Read Matrix [A]_{n,m}
For i=1 To N
  Connect to Cluster Member[i]
  Send [A]_{n/N,m}
  Disconnect from Cluster Member[i]
Next i
For i=1 To N
  Connect to Cluster Member[i]
  Receive [A']_{n/N,m}
  Disconnect from Cluster Member[i]
Next i
Rotate [A']_{n,m}
For i=1 To N
  Connect to Cluster Member[i]
  Send [A']_{n/N,m}
  Disconnect from Cluster Member[i]
Next i
For i=1 To N
  Connect to Cluster Member[i]
  Receive [A'']_{n/N,m}
  Disconnect from Cluster Member[i]
Next i
Store [A'']_{n,m}
```

Cluster Member Pseudo Code:

```
Do
  Listen for Connection from Cluster Server
  Connect to Cluster Server
  Read Matrix [A]_{n,m}
  For i=1 To n
    FFT([A]_{i,m})
  Next i
  Connect to Cluster Server
  Send Result [A']_{n,m}
  Disconnect from Cluster Server
End Do
```

To demonstrate the above algorithms we will perform a 2-dimensional FFT on one 4×4 data matrix using a cluster with four CM's. Consider the matrix $[A]_{4\times4}$:

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix}$$

The CS needs to partition the data and send them to the participating CM's. Each CM will receive one row of $[A]$:

CM 1:

$$A1 = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \end{bmatrix}$$

CM 2:

$$A2 = \begin{bmatrix} A_{21} & A_{22} & A_{23} & A_{24} \end{bmatrix}$$

CM 3:

$$A3 = \begin{bmatrix} A_{31} & A_{32} & A_{33} & A_{34} \end{bmatrix}$$

CM 4:

$$A4 = \begin{bmatrix} A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix}$$

Each CM will then perform FFTs on the rows it has received and send the results to the CS:

CM 1:

$$\mathcal{F}(A1) = \begin{bmatrix} A1'_{11} & A1'_{12} & A1'_{13} & A1'_{14} \end{bmatrix}$$

CM 2:

$$\mathcal{F}(A2) = \begin{bmatrix} A2'_{21} & A2'_{22} & A2'_{23} & A2'_{24} \end{bmatrix}$$

CM 3:

$$\mathcal{F}(A3) = \begin{bmatrix} A3'_{31} & A3'_{32} & A3'_{33} & A3'_{34} \end{bmatrix}$$

CM 4:

$$\mathcal{F}(A4) = \begin{bmatrix} A4'_{41} & A4'_{42} & A4'_{43} & A4'_{44} \end{bmatrix}$$

The CS will assemble the individual results into one matrix:

$$\left(\begin{bmatrix} A1' \\ A2' \\ A3' \\ A4' \end{bmatrix} \right) \Longrightarrow \begin{bmatrix} A'_{11} & A'_{12} & A'_{13} & A'_{14} \\ A'_{21} & A'_{22} & A'_{23} & A'_{24} \\ A'_{31} & A'_{32} & A'_{33} & A'_{34} \\ A'_{41} & A'_{42} & A'_{43} & A'_{44} \end{bmatrix}$$

The CS will then rotate the resulting matrix, so that a series of 1-dimensional FFT's can be performed on the columns of $[A']$:

$$Rotate\left(\begin{bmatrix} A'_{11} & A'_{12} & A'_{13} & A'_{14} \\ A'_{21} & A'_{22} & A'_{23} & A'_{24} \\ A'_{31} & A'_{32} & A'_{33} & A'_{34} \\ A'_{41} & A'_{42} & A'_{43} & A'_{44} \end{bmatrix}\right) \Longrightarrow \begin{bmatrix} A'_{11} & A'_{21} & A'_{31} & A'_{41} \\ A'_{12} & A'_{22} & A'_{32} & A'_{42} \\ A'_{13} & A'_{23} & A'_{33} & A'_{43} \\ A'_{14} & A'_{24} & A'_{34} & A'_{44} \end{bmatrix}$$

The resulting matrix $[A']$ will then again be partitioned and its rows sent to participating CM's:

CM 1:

$$A1' = \square \; A'_{11} \quad A'_{21} \quad A'_{31} \quad A'_{41} \; \Big]$$

CM 2:

$$A2' = \square \; A'_{12} \quad A'_{22} \quad A'_{32} \quad A'_{42} \; \Big]$$

CM 3:

$$A3' = \square \; A'_{13} \quad A'_{23} \quad A'_{33} \quad A'_{43} \; \Big]$$

CM 4:

$$A4' = \square \; A'_{14} \quad A'_{24} \quad A'_{34} \quad A'_{44} \; \Big]$$

Each CM will then perform an FFT on the rows it has received and then send the results to the CS:

CM 1:

$$\mathcal{F}(A1') = \begin{bmatrix} A''_{11} & A''_{21} & A''_{31} & A''_{41} \end{bmatrix}$$

CM 2:

$$\mathcal{F}(A2') = \begin{bmatrix} A''_{12} & A''_{22} & A''_{32} & A''_{42} \end{bmatrix}$$

CM 3:

$$\mathcal{F}(A3') = \begin{bmatrix} A''_{13} & A''_{23} & A''_{33} & A''_{43} \end{bmatrix}$$

CM 4:

$$\mathcal{F}(A4') = \begin{bmatrix} A''_{14} & A''_{24} & A''_{34} & A''_{44} \end{bmatrix}$$

The CS will assemble the individual results into one matrix:

$$\begin{pmatrix} \begin{bmatrix} A1'' \end{bmatrix} \\ \begin{bmatrix} A2'' \end{bmatrix} \\ \begin{bmatrix} A3'' \end{bmatrix} \\ \begin{bmatrix} A4'' \end{bmatrix} \end{pmatrix} \Longrightarrow \begin{bmatrix} A''_{11} & A''_{21} & A''_{31} & A''_{41} \\ A''_{12} & A''_{22} & A''_{32} & A''_{42} \\ A''_{13} & A''_{23} & A''_{33} & A''_{43} \\ A''_{14} & A''_{24} & A''_{34} & A''_{44} \end{bmatrix}$$

5.2.4 Concluding Remarks

The algorithm has been implemented in the C programming language (see the listings in the Appendix). The correctness of its operation was tested by running a series of 2-dimensional FFT's on several matrices whose dimensions were a power of two. The matrices contained the data of a 2-dimensional pulse function. It is known that the values of all elements of a 1-dimensional FFT of a pulse function are close to zero, except for the value of the first element, which is close to the sum of all elements of the original data matrix:

$$[A]_{1\times 4} = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \Longrightarrow \mathcal{F}([A]) = \begin{bmatrix} 4 & 0 & 0 & 0 \end{bmatrix}$$

If we then perform a series of 1-dimensional FFT's on the rows of a square matrix $[A]$ we will obtain a matrix whose entries are all 0 except for the entries in the first column:

$$[A]_{4\times 4} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \Longrightarrow \mathcal{F}_{rows}([A]) = \begin{bmatrix} 4 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 \end{bmatrix}$$

Performing a series of 1-dimensional FFT's on the columns of the matrix will result in a matrix whose elements are all 0 except for the value of $A_{1,1}$ which will again be the sum of all elements of the first column:

$$[A]_{4\times4} = \begin{bmatrix} 4 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 \end{bmatrix} \Longrightarrow \mathcal{F}_{columns}([A]) = \begin{bmatrix} 16 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

We could use this property of the Fourier transform to test the correctness of the computed results. After each computation of the 2-dimensional FFT of a square pulse function we test if the value of the first element is equal or very close to the product of the matrix's dimension. The values of the rest of the elements should be close to 0.

The distributed 2-dimensional FFT worked correctly during all tests conducted.

5.3 Electric Field Approximation

Section 2.5.2 illustrated the algorithm for electric field approximation that can be performed on a digital computer.

Let $[A]$ be an $n \times m$ 2-dimensional matrix representing a plate on which we want to calculate the electric field. The potential values to which the plate is subjected are stored in the first (top), last (bottom) rows and first (left) and last (right) columns. The initial values of the grid matrix are set to the average value of the potentials the plate is exposed to:

$$A_{i,j} = \frac{P_{top} + P_{bottom} + P_{left} + P_{right}}{4} \tag{5.2}$$

After the matrix has been initialized one can proceed and calculate the grid values using the algorithm presented in section 2.5.2:

$$A_{i,j} = \frac{A_{i+1,j} + A_{i-1,j} + A_{i,j+1} + A_{i,j-1}}{4} \tag{5.3}$$

5.3.1 Sequential Algorithm

Mesh calculation algorithm is frequently implemented in a sequential manner using three nested loops. The target matrix is handled on an element by element basis. The outer loop of the algorithm is used to perform a number of iterations required for satisfactory convergence of the values of the grid elements. The two inner loops traverse every row and every column of the matrix and allow for the calculation the values of the grid elements. Such an implementation is illustrated in the pseudocode below:

```
Initialize Matrix [A]_{n,m}
For i=1 To MaxIterations
  For y=1 To n-1
    For x=1 To m-1
      A_{x,y}=(A_{x+1,y}+A_{x-1,y}+A_{x,y+1}+A_{x,y-1})/4
    Next x
  Next y
Next i
Store [A]_{n,m}
```

5.3.2 Parallel Algorithm

The sequential algorithm illustrated in section 5.3.1 works very well on a single processor (NUMA) computer; however, in order to implement this algorithm in a parallel manner several issues need to be addressed. Distributed implementation of this algorithm requires partitioning of the grid and assigning the partitions to every computer participating in the computation. This partitioning and assignment of the data is usually done by one machine, which is aware of all the machines participating in the computations.

Suppose we could use two independent processors to perform the mesh calculations on a $n \times n$ matrix A. First we would divide the data evenly and then we would allocate the data to both processors to perform the mesh calculations:

Processor 1

$$\begin{matrix} A_{1,1} & \cdots & A_{1,n} \\ \vdots & \cdots & \vdots \\ A_{n/2,1} & \cdots & A_{n/2,n} \end{matrix}$$

Processor 2

$$\begin{matrix} A_{n/2+1,1} & \cdots & A_{n/2+1,n} \\ \vdots & \cdots & \vdots \\ A_{n,1} & \cdots & A_{n,n} \end{matrix}$$

Then we could use each processor to perform one iteration of the mesh calculation on the data it has access to:

Processor 1
For every element calculate:
$A_{i,j} = \frac{A_{i+1,j}+A_{i-1,j}+A_{i,j+1}+A_{i,j-1}}{4}$

Processor 2
For every element calculate:
$A_{i,j} = \frac{A_{i+1,j}+A_{i-1,j}+A_{i,j+1}+A_{i,j-1}}{4}$

We would need to repeat the calculations several times in order to obtain a satisfactory convergence of the grid values. We then collect and combine the results into the result matrix A'.

The problem with the algorithm is that the mesh values at the boundaries (rows n/2 and n/2+1) will not be calculated as there are no data required to calculate them. Since the data reside on machines physically distinct from each other, additional communications are required in order to ensure the correct grid values at the partition boundaries. The communications can either take place among the participating machines or they can be performed between the participants and the machine acting as a server. A distributed algorithm that produces correct grid values of the matrix and the boundaries is listed below:

Processor 1
```
For every element calculate:
$A_{i,j} = \frac{A_{i+1,j}+A_{i-1,j}+A_{i,j+1}+A_{i,j-1}}{4}$
SendRows($A_1, A_2, A_{n/2-1}, A_{n/2}$)
ReceiveRows($A_1, A_2, A_{n/2-1}, A_{n/2}$)
```

Processor 2
```
For every element calculate:
$A_{i,j} = \frac{A_{i+1,j}+A_{i-1,j}+A_{i,j+1}+A_{i,j-1}}{4}$
SendRows($A_{n/2}, A_{n/2+1}, A_{n-1}, A_n$)
ReceiveRows($A_{n/2}, A_{n/2+1}, A_{n-1}, A_n$)
```

In order to simplify the cluster member algorithms and minimize the delays caused by the computations of the grid values at the boundaries, the server-participant type of communications has been implemented. The participant's communications algorithm has been simplified, as it is the server that assigns and coordinates the data flow to and from the participants. The server is also aware of the boundaries resulting from the partitioning of data. Communications can be performed either in a synchronous or an asynchronous manner. Since all the participants had the same CPU and the number of data points required to compute the grid values at the boundaries is only $4N$ per participant, a synchronous type of communication was chosen and implemented. Each processor sends the two top (A_1, A_2) and bottom (A_{n-1}, A_n) rows to the computer that assigned the data to them. That computer performs the calculations of the grid values at the boundaries. The computed grid values are sent back to the computers they originated from.

5.3.3 Cluster Implementation

Consider the following cluster infrastructure. There exist N independent computing entities *(Cluster Members or CM)* capable of performing grid values approximations on an arbitrarily sized matrix $[A]$.

There exists a supervising computing entity *(Cluster Server or CS)* which is coordinating any computing activities in the cluster. The CS is aware of each and every CM available for computations. The CS divides the computational task evenly among all CM's. This means that the data are partitioned and sent to all CM's.

Each CM is waiting for data to compute on; when it receives the data (set of rows) it computes the values the grid elements. After the computations are complete CM sends the values of the boundary rows (*1* and n or top and bottom) together with the values of the neighbouring rows (*2* and *n-1*) to the cluster server for 'mending'. The mended rows are used in the next round of computations. The computations are repeated a predetermined number of times specified by the CS. Finally, the results of the computation are sent to the computer where the data originated from (CS).

The CS receives all results and combines them into one logical entity representing the electric field values on the given plate.

Cluster Server Pseudo Code:

```
Read Matrix [A]_{n,m}
For i=1 To N
  Connect to Cluster Member[i]
  Send [A]_{n/N,m}
  Disconnect from Cluster Member[i]
```

```
Next i
For i=1 To NumOfIterations
  Connect to Cluster Member[i]
  Receive [B]_{n/4,m}
  Disconnect from Cluster Member[i]
  Mend [B]
  Connect to Cluster Member[i]
  Send [B]_{n/4,m}
  Disconnect from Cluster Member[i]
Next i
For i=1 To N
  Connect to Cluster Member[i]
  Receive [A]_{n/N,m}
  Disconnect from Cluster Member[i]
Next i
Store [A]_{n,m}
```

Cluster Member Pseudo Code:

```
Do
  Listen for Connection from Cluster Server
  Connect to Cluster Server
  Read Matrix [A]_{n,m}
  For i=1 To NumberOfIterations
    Calculate Grid(A)
    Connect to Cluster Server
    Send [A]_{n/4,m}
    Disconnect from Cluster Server
    Connect to Cluster Server
    Receive [A]_{n/4,m}
    Disconnect from Cluster Server
  Next i
  Connect to Cluster Server
  Send Results [A]_{n,m}
  Disconnect from Cluster Server
End Do
```

To demonstrate the above algorithms we will calculate the potential values of a 16×16 grid matrix using a cluster with two CM's. Consider the matrix $[A]_{16\times16}$:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,15} & A_{1,16} \\ A_{2,1} & A_{2,2} & \dots & A_{2,15} & A_{2,16} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ A_{16,1} & A_{16,2} & \dots & A_{15,16} & A_{16,16} \end{bmatrix}$$

The CS needs to partition the data and send them to the participating CM's. Each CM will receive eight rows of $[A]$:

CM 1:

$$A1 = \begin{bmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,15} & A_{1,16} \\ A_{2,1} & A_{2,2} & \dots & A_{2,15} & A_{2,16} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ A_{8,1} & A_{8,2} & \dots & A_{8,15} & A_{8,16} \end{bmatrix}$$

CM 2:

$$A2 = \begin{bmatrix} A_{9,1} & A_{9,2} & \dots & A_{9,15} & A_{9,16} \\ A_{10,1} & A_{10,2} & \dots & A_{10,15} & A_{10,16} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ A_{16,1} & A_{16,2} & \dots & A_{15,16} & A_{16,16} \end{bmatrix}$$

Each CM will then perform one iteration of the calculations on the rows it has received:
CM 1:

$$ComputeGridValues(A1) = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{115} & A_{116} \\ A_{21} & A'_{22} & \dots & A'_{215} & A_{216} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ A_{71} & A'_{72} & \dots & A'_{715} & A_{816} \\ A_{81} & A_{82} & \dots & A_{815} & A_{816} \end{bmatrix}$$

CM 2:

$$ComputeGridValues(A2) = \begin{bmatrix} A_{9,1} & A_{9,2} & \dots & A_{9,15} & A_{9,16} \\ A_{10,1} & A'_{10,2} & \dots & A'_{10,15} & A_{10,16} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ A_{15,1} & A'_{15,2} & \dots & A'_{15,15} & A_{15,16} \\ A_{16,1} & A_{16,2} & \dots & A_{15,16} & A_{16,16} \end{bmatrix}$$

After every iteration the CM's will send the boundary rows containing the intermediate results of the calculations to the CS for adjustment:
CS:

$$MendBoundaries(TempA) = \begin{bmatrix} A_{71} & A'_{72} & \dots & A'_{715} & A_{816} \\ A_{81} & A'_{82} & \dots & A'_{815} & A_{816} \\ A_{9,1} & A'_{9,2} & \dots & A'_{9,15} & A_{9,16} \\ A_{10,1} & A'_{10,2} & \dots & A'_{10,15} & A_{10,16} \end{bmatrix}$$

The corrected boundaries are then sent to the CM's for the next round of computations. The operation is repeated N times (number of iteratation or until a satisfactory convergence of the results is obtained). After all iterations are completed the CS will assemble the individual results into one matrix:

$$\left(\begin{matrix} \square & \\ & A1^n \\ \square & \\ & A2^n \end{matrix} \right] \Bigg) \Longrightarrow \begin{bmatrix} A^n_{1,1} & A^n_{1,2} & \dots & A^n_{1,15} & A^n_{1,16} \\ A^n_{2,1} & A^n_{2,2} & \dots & A^n_{2,15} & A^n_{2,16} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ A^n_{16,1} & A^n_{16,2} & \dots & A^n_{15,16} & A^n_{16,16} \end{bmatrix}$$

5.3.4 Concluding Remarks

The algorithm has been implemented in the C programming language (see the listings in the Appendix). The correctness of its operation was tested by running a series calculations of potential distribution on a plate by one computer and by multiple computers and then comparing the results.
The distributed version of the algorithm worked correctly during all tests conducted. A sample output of the computed results by the cluster is plotted in figure 5.5.

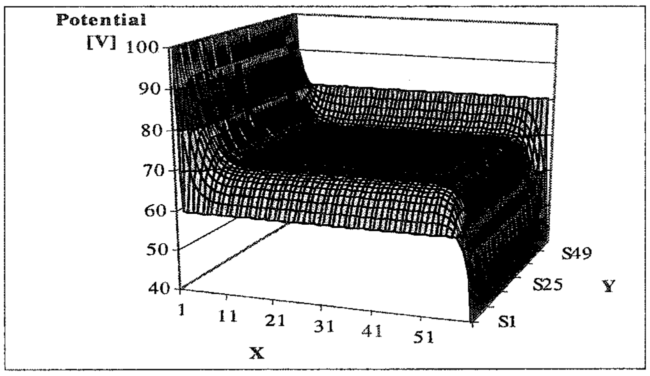


Figure 5.5: Mesh calculations

5.4 Multitreaded Server Applications

The Cluster Server coordinates all computations in the designed cluster. The server is the only computer aware of all cluster members and thus capable of utilizing their resources. During the design particular care was paid to the development of an environment that would not be restricted to a specific configuration. The distribution of work is determined at the run time. The server, depending on the number of participating cluster members, creates a working thread for each cluster member participating in the computation. The thread is given a fraction of the data to be computed on and then independently conducts communications with the assigned cluster member. The number of working threads is restricted only by the memory constrains of the server. When all cluster members finish the computations, their results are collected and the performance of the cluster is stored in a database for future analysis. No fault tolerance has been implemented in the development system. The server, however, is capable of recognizing the fact that a cluster member is not responding. Upon discovery of a problem the server notifies the operator about the cluster member causing a problem.

5.4.1 Matrix Multiplication

In section 5.1.3 a pseudocode for the cluster server was described. A clarification is needed at this point. Without the use of threading techniques the parallel algorithm performance would be impaired if it were performed by the server in a sequential manner as listed below.

Cluster Server Sequential Pseudo Code

The two loops responsible for sending and receiving data are sequential by their nature. The algorithm should be implemented in a more efficient manner:

```
For i=1 To N
  Connect to Cluster Member[i]
  Send [A]{n/N,n}
  Send [B]{n,n}
  Disconnect from Cluster Member[i]
Next i
For i=1 To N
  Connect to Cluster Member[i]
  Receive [C]{n/N,n}
  Disconnect from Cluster Member[i]
Next i
```

Figure 5.6: Sequential Server Code

Cluster Server Multithreaded Pseudo Code:

```
For $i = 1$ To N
    Create Thread $i$ Responsible for Communicating
    with Cluster Member $i$
Next $i$
For $i = 1$ To N
    Wait for Thread $i$ to Finish
Next $i$
```

This algorithm will attempt to communicate with all participating cluster members simultaneously and the throughput of the server will increase.

5.4.2 2D-FFT

The 2-D FFT parallel algorithm listed in section 5.2.2 suffers from the same problem as the matrix multiplication algorithm described in the previous section. A multithreaded version was developed in order to enhance the performance of the server.

5.4.3 Shared Memory Access

The use of threaded techniques improves the throughput of the server. However, communication with multiple clients simultaneously complicates memory management, as simultaneous accesses to shared variables can take place. We know that on a shared memory computer each CPU can access any memory location. It is possible that the running threads might attempt to update the shared memory areas simultaneously. The usage of locks was considered for synchronizing access to the shared memory. Such a protection is always expensive [91]. It can be seen from the server program listing that each thread works only the memory area it was assigned to work on. Any updates in that area would only be performed by one thread at a time; hence it is safe to allow the threads access to the shared memory at any time.

Chapter 6

Experimental Data and Results

The primary objective of the conducted experiments was to determine the cluster's functionality and applicability. Several synthetic and practical applications have been developed and used to obtain the cluster's characteristics. Synthetic applications were used to obtain the cluster's I/O characteristics and dependencies. In particular the system latency and the I/O throughput were determined. Practical applications were used to obtain the raw performance (wall time clock SpeedUp) of the system. The following sections demonstrate sample results of all conducted experiments

6.1 System Latency

System latency has been defined in section 2.1.1 as the amount of time required for the system to setup computations. The implemented cluster is interconnected using an Ethernet network, hence its latency is strongly dependent on (related to) the latency of the interconnecting medium. The latency of the system was determined experimentally by recording the data transfer val-

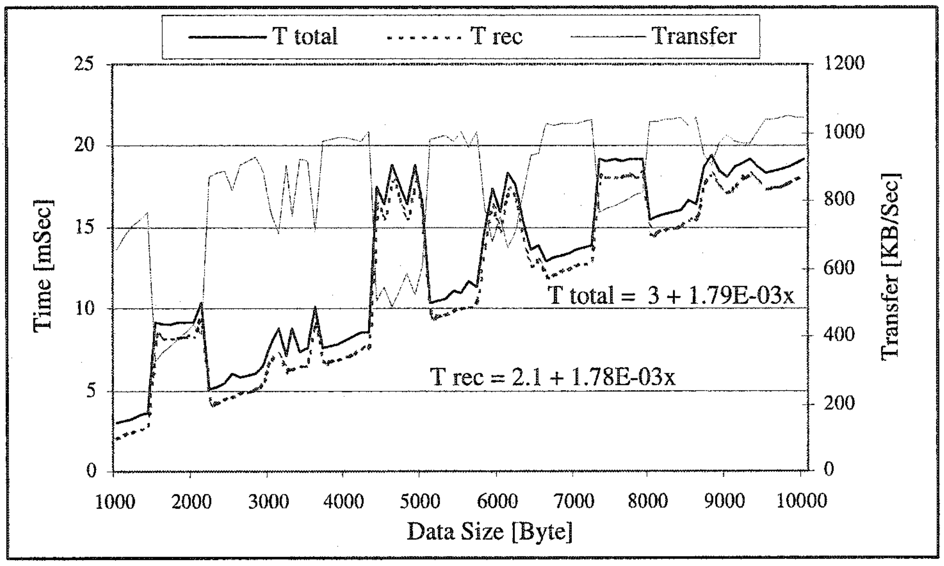


Figure 6.1: Machine latency on 10MBit network

Network	Latency $[ms]$
10 Mbit	4
100 Mbit	3

Table 6.1: Network latency

ues of various batches of data. The data were sent from the cluster server to a cluster member. The amount of data was increased until the transfer rate reached its maximum for the given Ethernet technology; 0.97[MB/s] and 8[MB/s] for 10MBit and 100MBit Ethernet networks, respectively. The slope of the curve was approximated and the results were interpolated to determine network latency (figures 6.1 and 6.2).

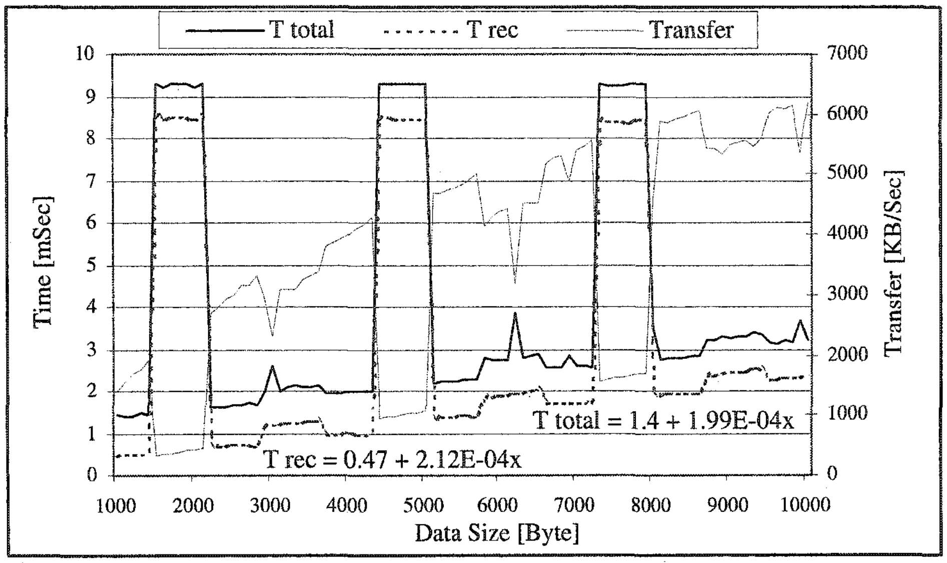


Figure 6.2: Machine latency on 100MBit network

6.2 Data Transfer

The computers participating in the experiment are fully independent machines interconnected via an Ethernet network. It is obvious that the performance of the cluster will depend on its network performance and data transfer capabilities. Applications that process a lot of data will be subject to the network performance of the cluster. Applications that perform a lot of processing locally will be subject to the CPU performances of the cluster participants.

The data required for cluster based computations can be transferred in either raw or marshaled format. Raw format is simpler to implement; however, the Marshalled format is safer and works regardless of the hardware architecture of cluster participants.

6.2.1 Raw Data Transfer

Raw format implementation does not translate the data in transfer into a format that is hardware architecture independent. The receiver will receive and interpret the received data the same way the sender sends, it provided both the receiver and the sender run on the same hardware architecture (Ix86 to Ix86, SUN to SUN, etc.)
This method is relatively safe, provided the designer uses only one type of hardware, or if the hardware architecture implementation is the same on all machines participating in the cluster.

6.2.2 Marshalled Data Transfer

In order to ensure that the data in transfer will always be interpreted correctly, regardless of the hardware architecture of the sender and receiver, one would need to convert the data to be transferred to a common network format. The sender converts the data from its hardware format to the network format. The data then are sent to the receiver which in turn will convert the data from the network format to its native architecture format. Each transfer requires additional processing of both the sender and the receiver.

6.2.3 Cluster Data Transfers

The star infrastructure that was used to implement the cluster is subject to Ethernet technology performance. The Ethernet technology does not handle simultaneous accesses linearly; however, for the six cluster member configuration, its performance does not degrade drastically.
Several experiments were conducted in order to determine if the network utilization had any impact on overall performance of the cluster.

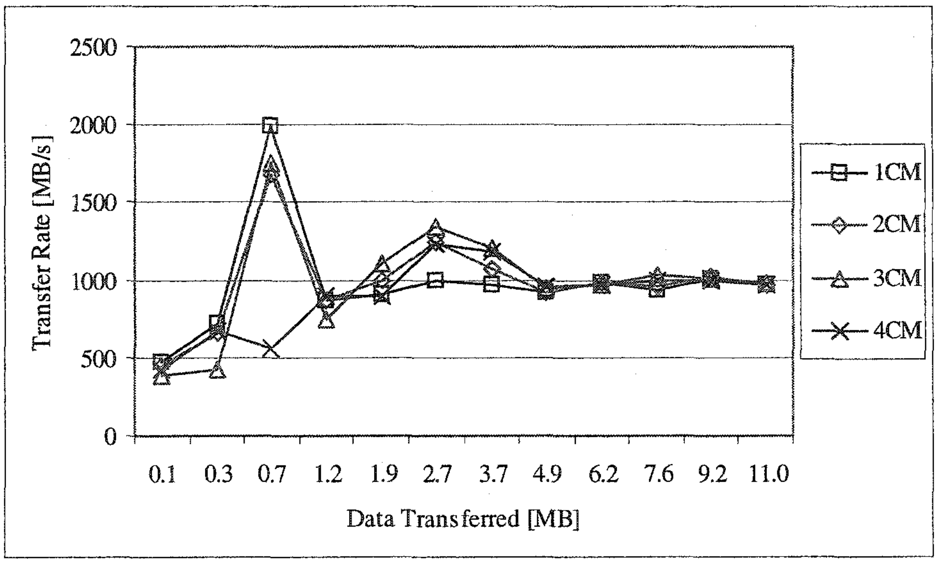


Figure 6.3: Transfer Rate on 10MBit Network

Figures 6.3 and 6.4 show execution times and effective transfer rates of four cluster configurations connected via a 10MHz and 100MHz Ethernet network.

6.3 Matrix Multiplication

The distributed matrix multiplication program described in section 5.1 has been run on the cluster and the execution times for various problem sizes have been recorded. The SpeedUp of the cluster has been calculated and the results are shown in figures 6.5 and 6.6.

6.4 2D FFT

A popular engineering application, namely the 2D-FFT was chosen for the second performance evaluator of the cluster. The algorithm used for computing

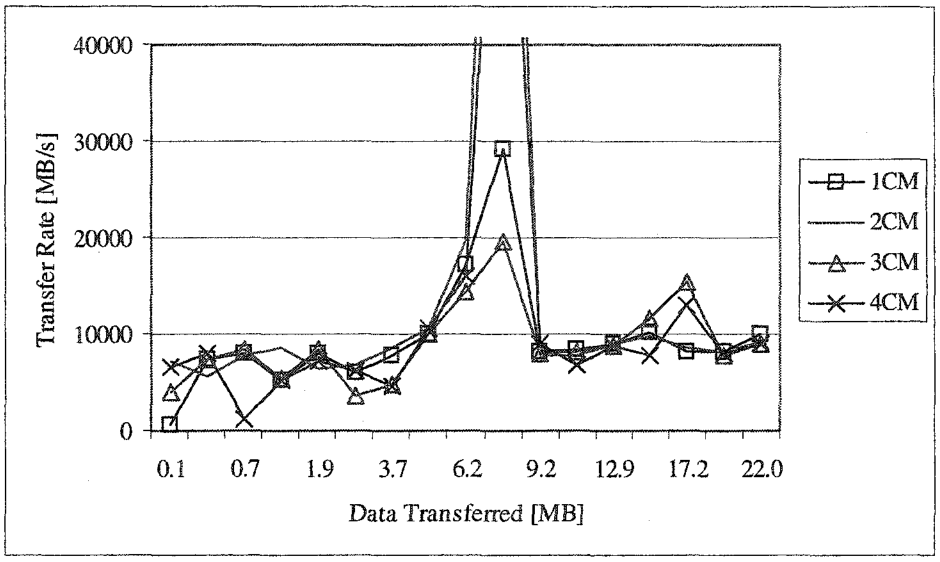


Figure 6.4: Transfer Rate on 100MBit Network

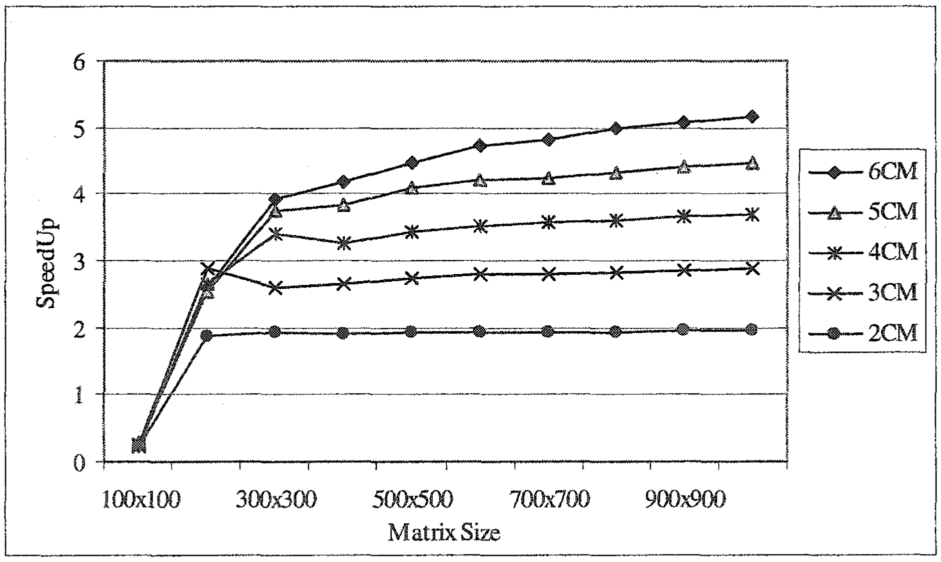


Figure 6.5: Matrix multiplication speedup on 10MBit network

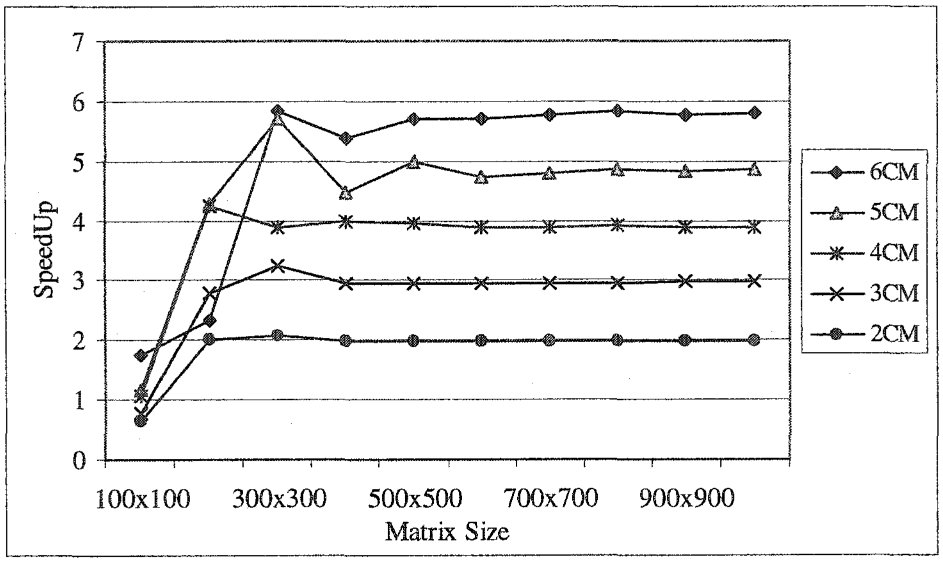


Figure 6.6: Matrix multiplication speedup on 100MBit network

the 2D-FFT in a distributed manner was described in section 5.2. The program was on the cluster and the execution times for various problem sizes were recorded. The SpeedUp of the cluster was calculated and the results are shown in figures 6.7 and 6.9.

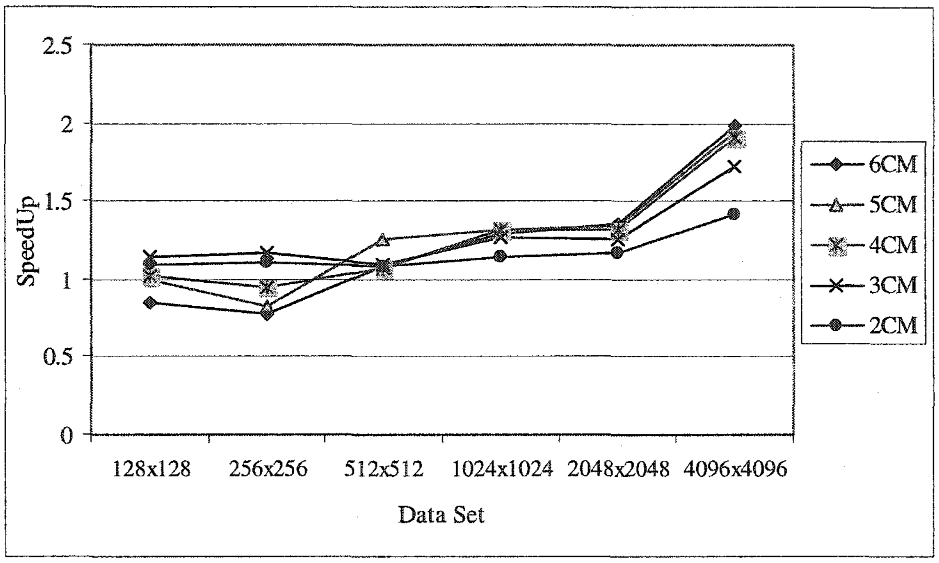


Figure 6.7: 2D-FFT speedup on 10MBit network

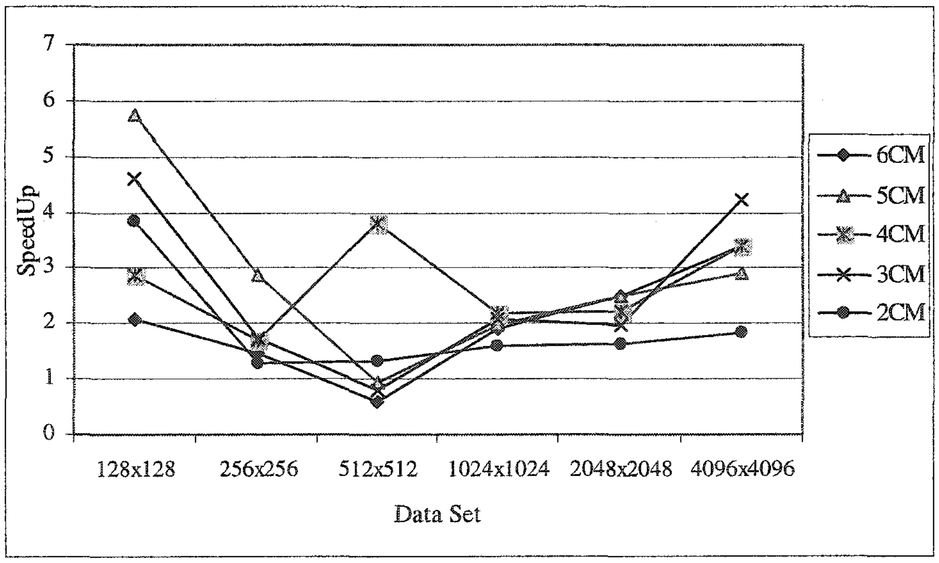


Figure 6.8: 2D-FFT SpeedUp on 100MBit network

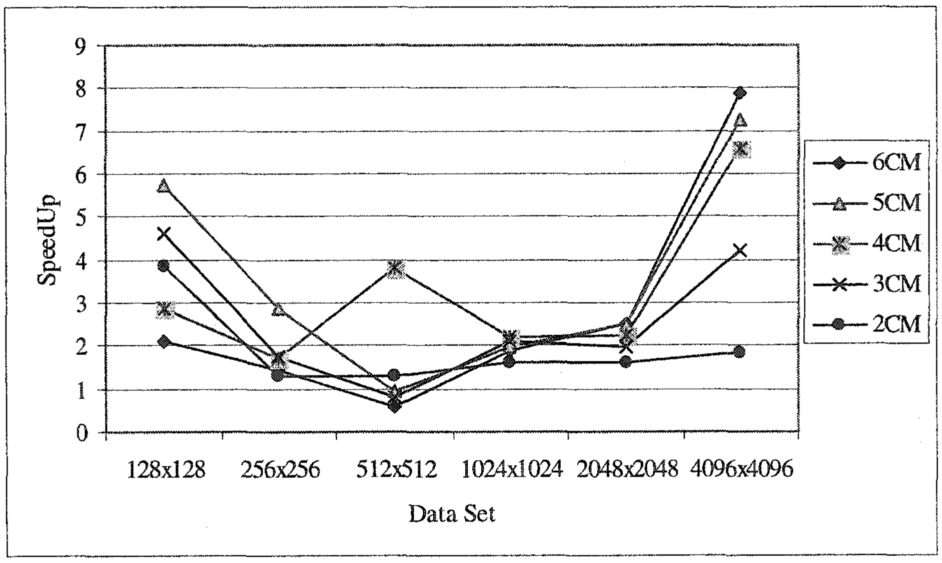


Figure 6.9: Large memory 2D-FFT SpeedUp on 100MBit network

6.5 Mesh Calculations

The third engineering application run on the cluster computed the grid values of a 2-dimensional mesh. The algorithm used for computing the values of the grid in a distributed manner was described in section 5.3. The program was run on the cluster and the execution times for various problem sizes were recorded. The SpeedUp of the cluster was calculated and the results are shown in figures 6.10 and 6.11.

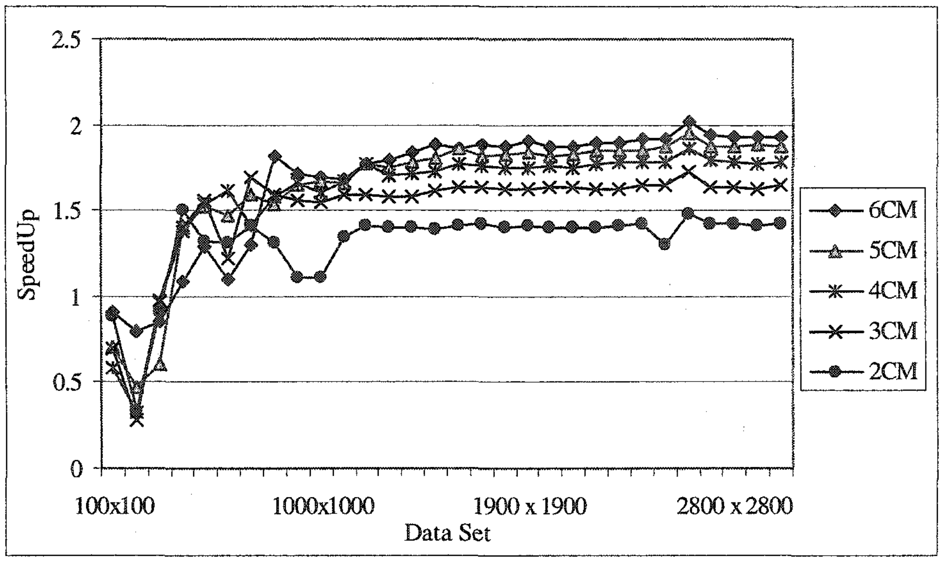


Figure 6.10: Mesh calculations SpeedUp on 10MBit network

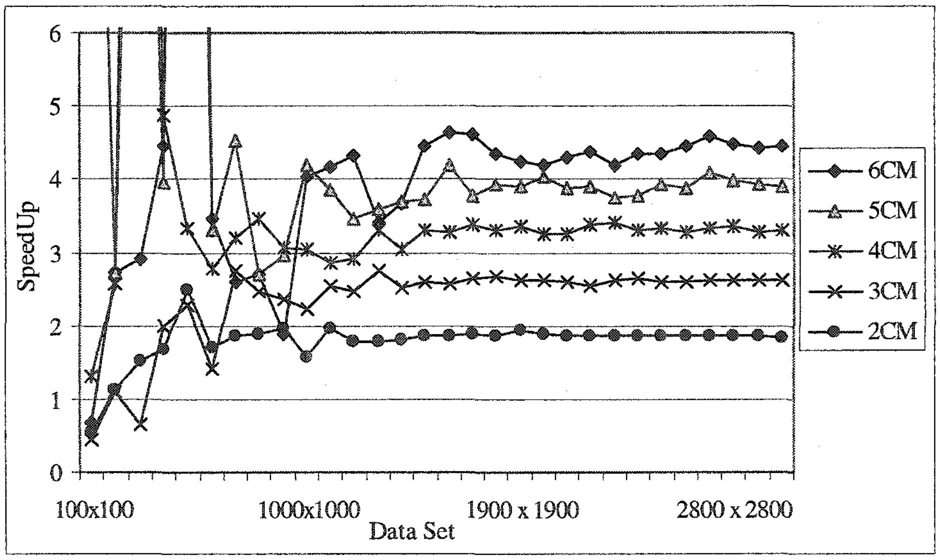


Figure 6.11: Mesh calculations SpeedUp on 100MBit network

Chapter 7

Discussion

Several analyses of the collected data were performed. These analyses are presented in the sections to follow.

7.1 Performance and Scalability

From section 6 we see that the cluster SpeedUp stops oscillating when the size of the data set becomes large enough (over 50% of the time is spent on computations, as opposed to I/O operations. In order to determine the maximum possible SpeedUp of the system, the execution times of the cluster configurations for the largest data sets were analyzed.

7.1.1 Distributed Matrix Multiplication

The SpeedUp of distributed matrix multiplication for the largest data set is shown in figure 7.1. The SpeedUp for both 10Mbit and 100Mbit configurations is a linear function of the number of cluster members.

The SpeedUp of the 10Mbit configuration can be regressed as a linear function

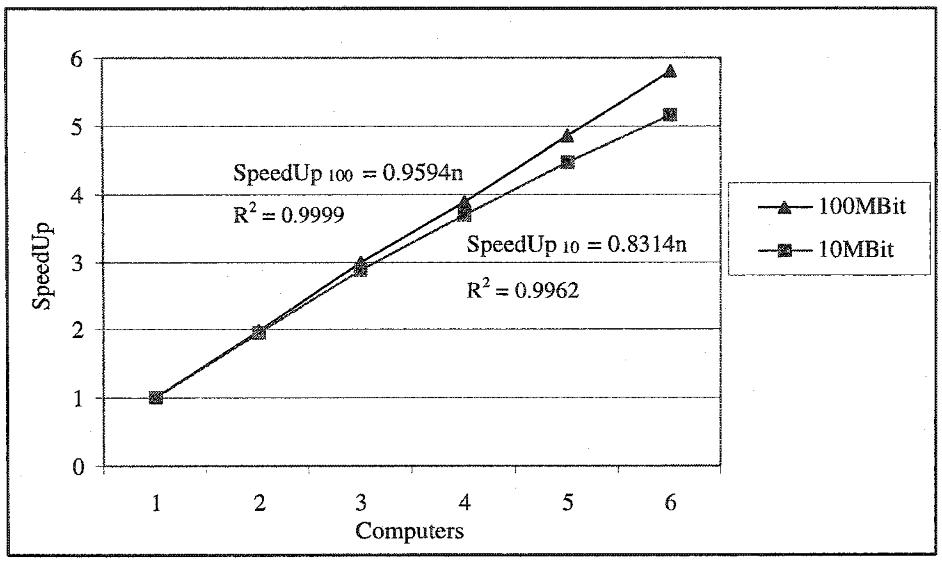


Figure 7.1: Large data matrix multiplication SpeedUp

as follows:

$$SU(n) = 0.8314n \tag{7.1}$$

Similarly, the SpeedUp of the 100Mbit configuration can be regressed according to the following expression:

$$SU(n) = 0.9594n \tag{7.2}$$

7.1.2 Distributed 2DFFT

The SpeedUp of distributed calculation of 2DFFT for the largest data set is shown in figure 7.2. The SpeedUp for both 10Mbit and 100Mbit configurations is a logarithmic function of the number of cluster members.

The SpeedUp of the 10Mbit configuration can be regressed using the following equation:

$$SU(n) = 0.2021\ln(n) + 1 \tag{7.3}$$

Similarly, the speedup of the 100Mbit configuration can be regressed using the following equation:

$$SU(n) = 0.8721\ln(n) + 1 \qquad (7.4)$$

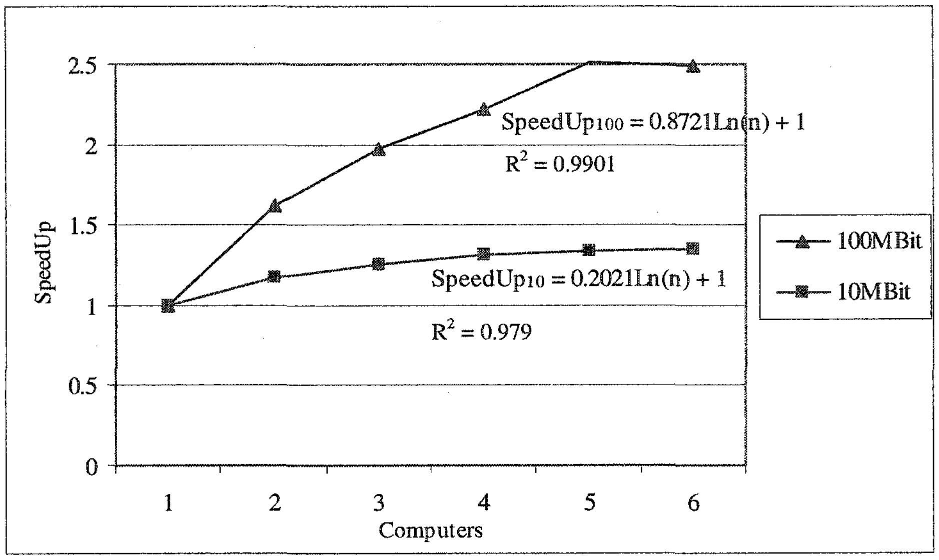


Figure 7.2: Large data 2D-FFT SpeedUp

The SpeedUp for 2D-FFT was observed to be substantially lower for the largest data sets on the implemented cluster by comparison to the low data sets. Analyses of the problem determined that some of the cluster members did not have enough RAM to handle the calculations without extensive swapping. The amount of RAM in the cluster member computers was doubled and the experiment involving the calculations of 2D-FFT for the largest data set was conducted again. The new results of the experiment are shown in figure 7.3. A super SpeedUp was achieved in the new configuration with a SpeedUp of 7.9 on a six machine cluster. The reason for the super SpeedUp was the fact that the base (reference) tests for the largest data set were conducted on a cluster member whose memory was barely adequate to store the data on which it computed. Some minor swapping occurred, which was compensated by the swapping of the cluster members equipped with less memory. When the memory of all cluster members was upgraded, in order to eliminate swapping, super SpeedUp was achieved.

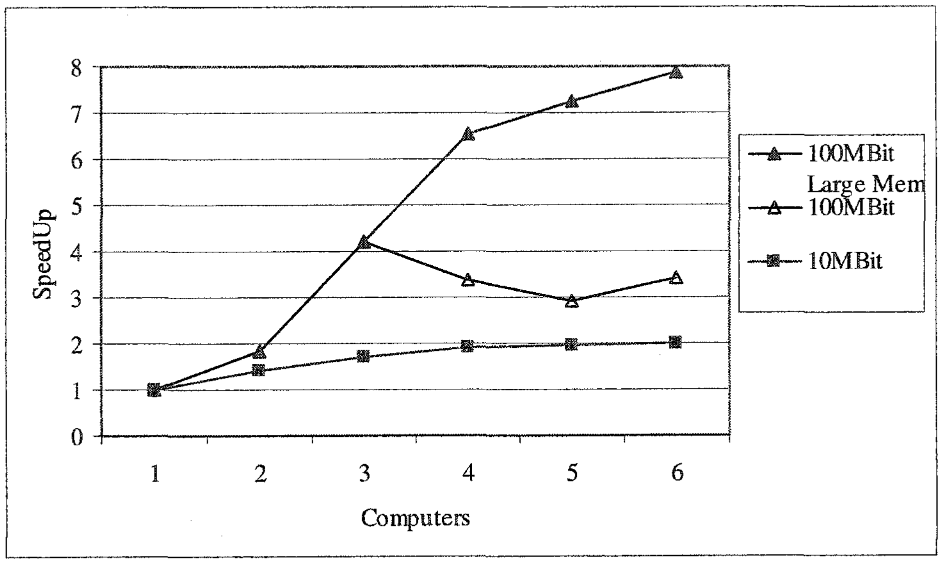


Figure 7.3: Large data 2D-FFT SpeedUp (Super SpeedUp)

7.1.3 Distributed Grid Calculation

Grid calculations fall into medium I/O category of cluster calculations. The speedup of distributed matrix multiplication for the largest data set is shown in figure 7.4. The speedup model for the 10Mbit configuration is a logarithmic function of the number of cluster members. The speedup of the 10Mbit configuration can be calculated using the following equation:

$$SU(n) = 0.5255 \ln(n) + 1.0337 \tag{7.5}$$

The speedup model for the 100Mbit configuration is a quadratic function of the number of cluster members. The speedup of the 100Mbit configuration can be accurately regressed on the following quadratic expression:

$$SU(n) = -0.0373n^2 + 0.9531n + 0.0829 \tag{7.6}$$

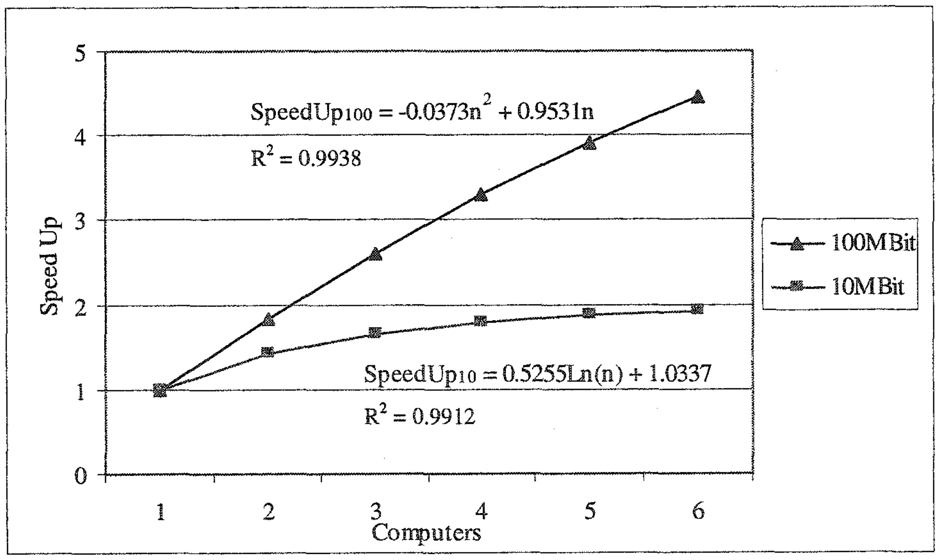


Figure 7.4: Large data grid calculation SpeedUp

7.2 Distributed Matrix Multiplication Modeling

The performance modeling of any computer system is a complex, application and data specific, task. The sections below discuss the developed models for the cluster's SpeedUp while performing matrix multiplication.

7.2.1 Discrete Model

The discrete model appears to be well suited for the cluster performing the computations on various (discrete) data sets.

IO Performance

The implemented cluster uses Ethernet network for member communications. From section 2.1.1 it is known that data on an Ethernet network are transferred in Ethernet frames that are later encapsulated by TCP/IP frames. The developed model includes an I/O component whose analysis is included below.

Matrix Multiplication I/O Analysis

Distributed matrix multiplication requires relatively low I/O. In order to multiply two matrices of size $N \times N$ the following amount of data needs to be transferred:

$$I/O(n, N) = SOF(\{N^2 + \frac{N^2}{n}\}n + N^2) \tag{7.7}$$

where SOF is the machine size of a floating point number, n the number of cluster members, and N the number of rows and columns of a $N \times N$ square matrix.

On a network with a finite packet size, equation (7.7) needs to be rearranged in order to calculate the number of packets required to send the data over the network:

$$Packets(n,N) = \{\frac{SOF(N^2)}{MPS}\}n + \{\frac{SOF(N^2/n)}{MPS}\}n + \{\frac{SOF(N^2/n)}{MPS}\}n \quad (7.8)$$

Since the fractional packets cannot be combined, equation (7.8) needs to be modified to allow the calculation of the actual amount of data sent over the network. In order to determine the number of packets required to transfer that amount of data the following calculation is performed:
$ActualPackets(n,N) =$

$$= RUp(\{\frac{SOF(N^2)}{MPS}\})n + RUp(\{\frac{SOF(N^2/n)}{MPS}\})n + RUp(\{\frac{SOF(N^2/n)}{MPS}\})n \quad (7.9)$$

where RUp is a round up or ceiling function and MPS is the maximum packet size for the medium. We could define *I/O Performance* as

$$I/O\ Perf(n,N) = \frac{Packets(n,N)}{ActualPackets(n,N)} \quad (7.10)$$

and after the expansion we obtain:

$$I/O\ Perf(n,N) = \frac{\{\frac{SOF(N^2)}{MPS}\}n + \{\frac{SOF(N^2/n)}{MPS}\}n + \{\frac{SOF(N^2/n)}{MPS}\}n}{RUp(\{\frac{SOF(N^2)}{MPS}\})n + RUp(\{\frac{SOF(N^2/n)}{MPS}\})n + RUp(\{\frac{SOF(N^2/n)}{MPS}\})n} \quad (7.11)$$

We also define *Cluster's I/O Performance* by comparing the number of packets required to send the data to one and n cluster members.

In order to calculate the number of packets required to send three $N \times N$ matrices (multiplicands and results) to one cluster member we need to perform the following calculation:

$$Packets(1,N) = 3RUp(\{\frac{SOF(N^2)}{MPS}\}) \tag{7.12}$$

The *Cluster's I/O Perf* would then be:

$$Cluster's\ I/O\ Perf(n,N) = \frac{Packets(1,N)}{ActualPackets(n,N)} \tag{7.13}$$

after expansion:

Cluster's I/O Perf(n,N) =

$$= \frac{3RUp(\{\frac{SOF(N^2)}{MPS}\})}{RUp(\{\frac{SOF(N^2)}{MPS}\})n + RUp(\{\frac{SOF(N^2/n)}{MPS}\})n + RUp(\{\frac{SOF(N^2/n)}{MPS}\})n} \tag{7.14}$$

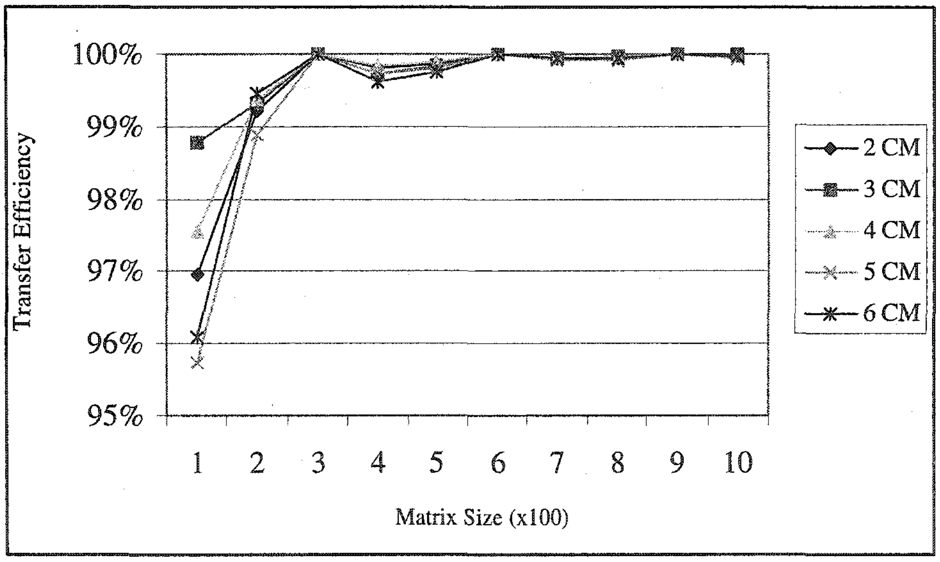


Figure 7.5: Cluster I/O Performance for a distributed matrix multiplication

CPU Performance

The implemented matrix multiplication algorithm illustrated in figure 7.6 requires both integer and floating point operations. In order to multiply two $N \times N$ matrices the processor has to perform the following operations:

- N^3 floating point multiplications [FPM]
- $3N^3 + N^2$ integer multiplications [IntM]
- N^3 floating point additions [FPA]
- $6N^3 + 2N^2$ integer additions [IntA]

```
int MultiplyMatrix(float *a, int aRow, int aCol, float *b, int bRow, int bCol, float *c)
{
  int x, y, z;
  for(z=0;z<aRow;z++){
      for(y=0;y<bCol;y++){
          *(c+(z*bCol+y))=0;
          for(x=0;x<aCol;x++)
            *(c+(z*bCol+y)) += *(a+(z*aCol+x)) * *(b+(x*bCol+y));
        }
    }
  return z*y*x;
}
```

Figure 7.6: Matrix multiplication algorithm

Cluster based, or distributed, matrix multiplication requires partitioning of the data among all of the participating cluster members. The data partitioning algorithm is illustrated in figure 7.7.

The simple algorithm allocates $\frac{N}{n}$ rows of matrix A as well as N rows of matrix B to each cluster member. The last cluster member is assigned either $\frac{N}{n}$ rows of matrix A or $RUp(\frac{N}{n})$ rows in the instance when N does not evenly

```
offset = 0;
for(membercount = 0; membercount<n; membercount++){
  cmData[membercount].matrixA = matrixA + offset; //move pointer to desired row of A
  cmData[membercount].matrixB = matrixB; //move pointer to the first row of B
  if(membercount < n-1) //allocate number of rows of array A
    cmData[membercount].arrdims[0] = N/n;  //integer division of the array
  else //if data does not divide evenly allocate the reminder to the last machine
    cmData[membercount].arrdims[0] = N - (N/n*(n-1)); //reminder
  cmData[membercount].arrdims[1] = N; //allocate number of columns of array A
  cmData[membercount].arrdims[2] = N; //allocate number of rows of array B
  cmData[membercount].arrdims[3] = N; //allocate number of columns of array B
  cmData[membercount].result = resultmatrix + offset; //move pointer to desired row of C
  offset += N*(N/n); //increment pointer offset for next cluster member
}
```

Figure 7.7: Data partitioning algorithm

divide by n. The number of CPU intensive operations performed by the cluster will then be:

$$CPUOps(N) = FPM(N) + IntM(N) + FPA(N) + IntA(N) \tag{7.15}$$

The maximum number of operations each cluster member will perform will then be:

$$MaxCPUOps(n,N) = FPM(RUp(\frac{N}{n}))+IntM(RUp(\frac{N}{n}))+FPA(RUp(\frac{N}{n}))+IntA(RUp(\frac{N}{n})) \tag{7.16}$$

The theoretical CPU SpeedUp of the cluster will then be:

$$CPU\ SpeedUp(n,N) = \frac{CPUOps(N)}{MaxCPUOps(n,N))} \tag{7.17}$$

and after expansion:

$CPU\ SpeedUp(n,N) =$

$$\frac{N^3_{FPM} + (3N^3 + N^2)_{IntM} + N^3_{FPA} + (5N^3 + 2N^2)_{IntA}}{(RUp(\frac{N}{n})^3_{FPM} + (3(RUp(\frac{N}{n})^3 + (RUp(\frac{N}{n})^2)_{IntM} + (RUp(\frac{N}{n})^3_{FPA} + (6(RUp(\frac{N}{n})^3 + 2(RUp(\frac{N}{n})^2)_{IntA}} \tag{7.18}$$

Cluster Performance

The cluster's performance is a function of several variables: cluster size, data size, setup time or latency, communications or I/O performance, and CPU performance. I/O performance and CPU performance have been determined in the previous sections. The discrete model of the system's performance while performing matrix multiplications can be determined using the following relation:

$$ClusterPerf(n,N,IO,CPU) = \frac{\text{1 CM Execution Time(N)}}{\text{n CM Execution Time(N)}} - SystemLosses(n,N) \tag{7.19}$$

where

$$\text{1 CM Execution Time(N)} = \text{1CM IO Time(N)} + \text{1CM CPUTime(N)} \tag{7.20}$$

$$\text{1 CM IO Time(N)} = \text{Packets(1,N) Packet Transfer Time} \tag{7.21}$$

1 CM CPU Time(N) =

FPM(N)FPMT + IntM(N)IntMT + FPA(N)FPAT + IntA(N)IntAT [1]

(7.22)

and

n CM Execution Time(N) = nCM IO Time(N) + nCM CPUTime(N) (7.23)

n CM IO Time(N) = ActualPackets(n,N) Packet Transfer Time (7.24)

After expansion:

n CM CPU Time(N) = $FPM(RUp(\frac{N}{n}))FPMT + IntM(RUp(\frac{N}{n}))IntMT$

$$+ FPA(RUp(\frac{N}{n}))FPAT + IntA(RUp(\frac{N}{n}))IntAT \tag{7.25}$$

and

$$SystemLosses(n, N) = C\frac{n^2}{N} \tag{7.26}$$

also

$$C = C_1C_2 \tag{7.27}$$

where C_1 is a network speed constant and C_2 is a dataset constant, both obtained experimentally.

[1]FPMT: Floating Point Multiplication Time, IntMT: Integer Multiplication Time, FPAT: Floating Point Addition Time, IntAT: Integer Addition Time.

Table 7.1: C_1 and C_2 Values

Network/Constant	C_1	C_2
10MBit	0.1	20000
100MBit	0.01	40000

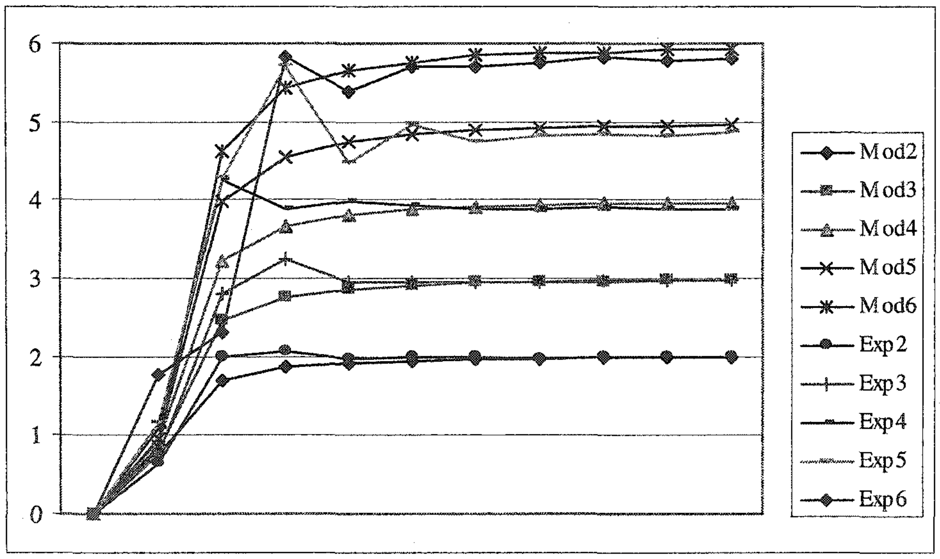


Figure 7.8: Matrix multiplication discrete model

7.2.2 Continuous Model

In order to perform continuous modeling of the response of the system one needs to analyze the cluster as a system that changes in time. The following considerations could be made when continuous modeling methods are to be applied.

Let us examine the computational task involving a series of matrix multiplications. The sizes of the matrices increase when the calculations of the last computation are complete. The total time required to perform the computations is the sum of the computation times of the varied sized matrices. The system response is recorded at the end of each iteration and the data is plotted, as in figures 7.11 and 7.12. The intervals at which the response is recorded increase with the increase of the data on which the system computes.

Data Transfer

The speed at which the system receives the data required for the computation plays a critical role in the cluster's performance. Figure 7.9 illustrates the average rate at which the cluster receives data. Since the designed cluster used shared Ethernet network, the transfer rate was decreasing as more machines were added.

CPU Utilization

Experimental data show that with the increase of cluster size the time spent on calculations decreases. This is mainly due to system overhead and to the increased complexity of the scheduling and assignment of the tasks to cluster members. The CPU utilization was calculated as the ratio of the processing

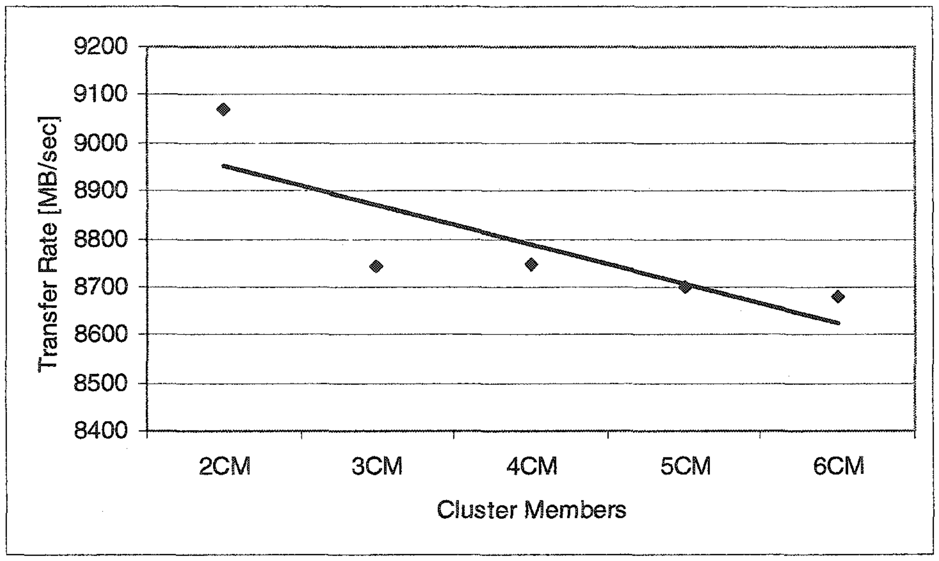


Figure 7.9: Cluster transfer rate

time to the total time required for the computation.

$$CPU_{util} = \frac{T_{CPU}}{T_{Total}} \tag{7.28}$$

Examination of the experimental data shows that, the response of the cluster performing matrix multiplication often resembles the forced response of an overdamped system. The overdamped system response can be calculated as the solution of the second order differential equation:

$$\lambda \frac{d^2SU}{dt^2} + \rho \frac{dSU}{dt} + \frac{1}{\gamma} SU = 0 \tag{7.29}$$

where λ, ρ, and γ are now considered as cluster parameters modeling the characteristics and SU is the system SpeedUp.

Equation (7.29) is as a homogenous second-order linear differential equation with constant coefficients (λ, ρ, γ). The characteristic polynomial associated to (7.29) is:

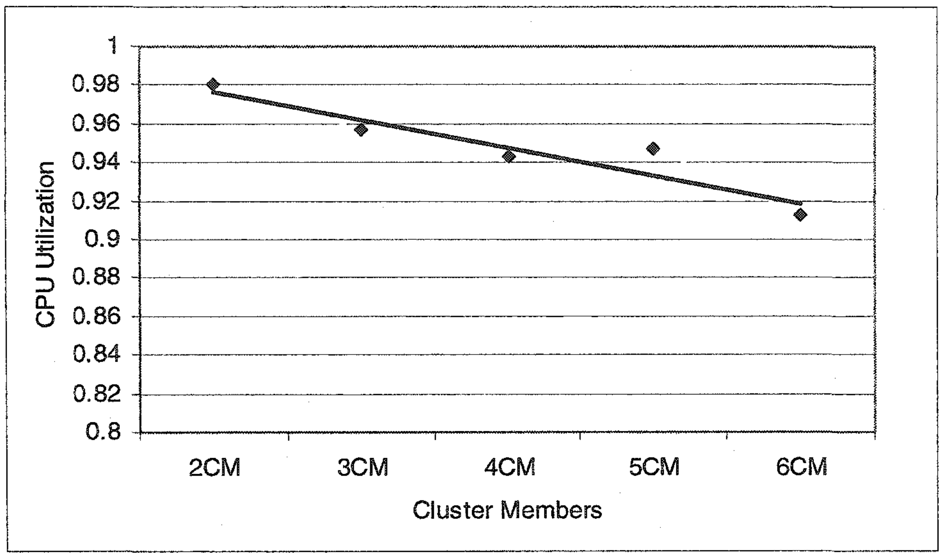


Figure 7.10: Cluster CPU utilization

$$P(s) = \lambda s^2 + \rho s + \frac{1}{\gamma} \tag{7.30}$$

with roots:

$$s_{1,2} = -(\frac{\rho}{2\lambda})^2 \pm \sqrt{(\frac{\rho}{2\lambda})^2 - \frac{1}{\lambda\gamma}} \tag{7.31}$$

The forced response of the overdamped system characterized by (7.29) is of the form [63]:

$$SU(t) = Ae^{s_1 t} + Be^{s_2 t} + F \tag{7.32}$$

where A and B are constants derived from initial conditions, namely:

$$SU(0+) = A + B \text{ and } \frac{dSU(0+)}{dt} = s_1 A + s_2 B \tag{7.33}$$

and $SU(0+) = -F$ and $\frac{dSU(0+)}{dt}$ are obtained experimentally.
Examples of the regression of experimental data on a relation of the type (7.32) are shown in figures 7.11 and 7.12. The close match of the curve with our data suggests that the computational cluster may indeed undergo damp oscillations during its operation. Although this may not always be the case, a significant number of the experimental plots suggests that. An immediate conclusion to this observation is that the performance of the computer cluster may be at times highly dependent on the "response" frequency of the system when processing different computational loads. As many experimental data cannot be regressed with sufficient accuracy on the solution of a damped oscillation, it follows that normally homogenous equation (7.29) may be too simple to capture the entire range of observed system response. We only want to point out that occasionally the simple modeling presented here appears to be appropriate and that it signals the oscillatory properties of the cluster.

Figure 7.13 gives the values of λ, ρ and γ obtained from the best fit regression on relation 7.32. A closer analysis of the data revealed some interesting facts related to parameters λ, ρ and γ.

1. It has been observed that the linear increase in γ is directly proportional to the increase of memory in the system

$$Memory(n) = K_1\gamma(n) \tag{7.34}$$

2. The linear decrease in ρ is directly proportional to the decrease of the effective data transfer rate of the system.

$$TransferRate(n) = \rho(n) \tag{7.35}$$

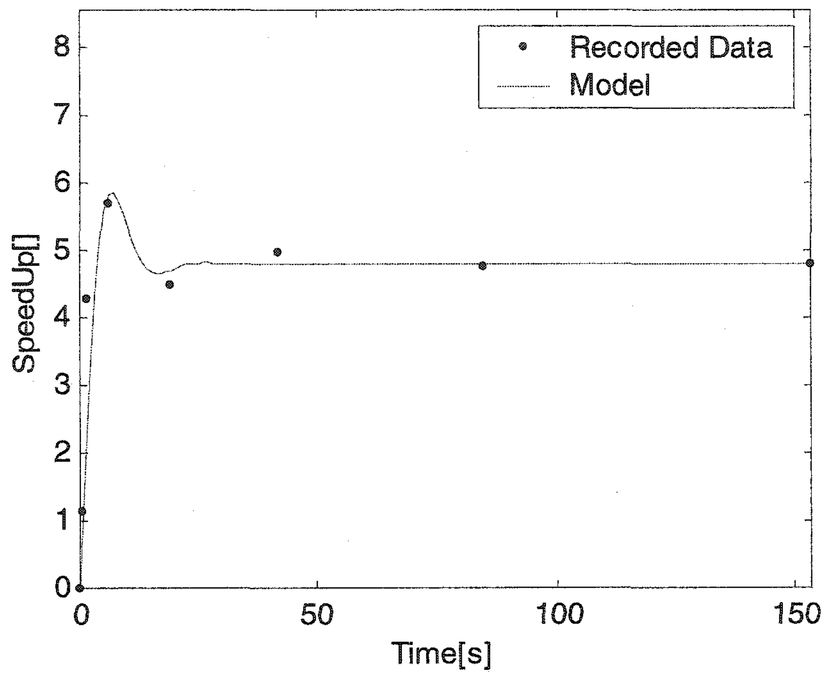


Figure 7.11: Matrix multiplication 5 machine SpeedUp model 100MBit

3. The decrease in λ is directly proportional to the decrease in CPU utilization of the system. The following relation for λ and CPU_{util} has been observed:

$$CPU_{util}(n) = \lambda(n) \tag{7.36}$$

The performance increase of the system (SpeedUp) is closely related to all of those parameters. The model demonstrates that there are areas when it is possible to predict, with reasonable accuracy the system SpeedUp, as a function of time, using the observed characteristics.

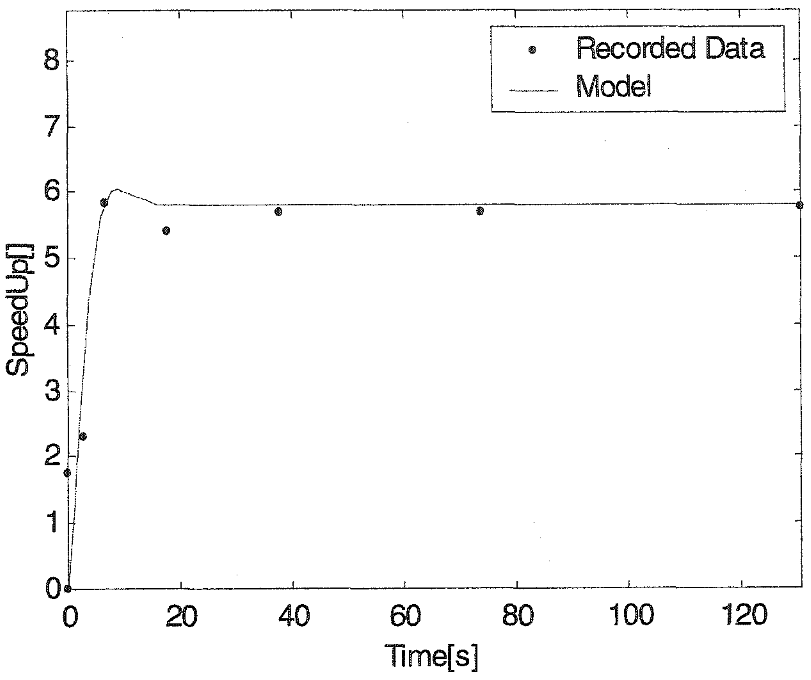
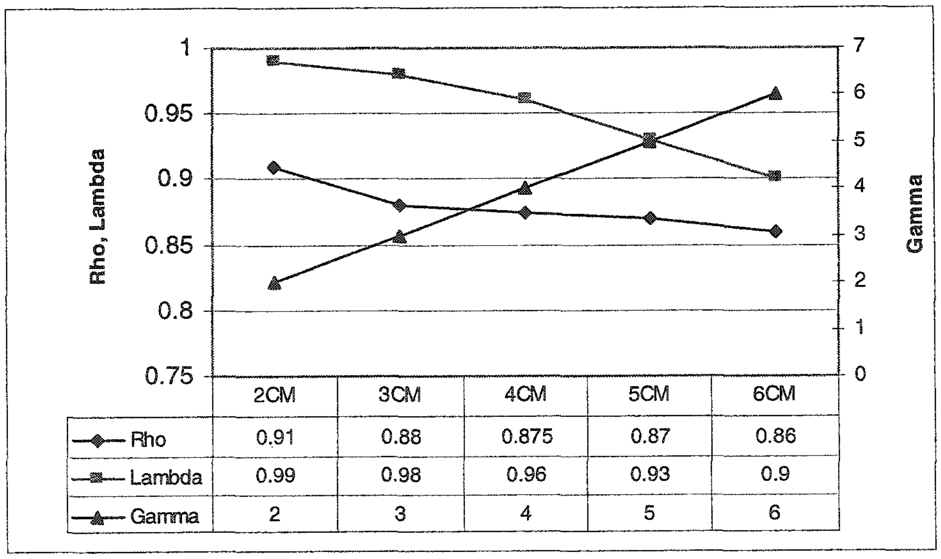


Figure 7.12: Matrix multiplication 6 machine SpeedUp model 100MBit



	2CM	3CM	4CM	5CM	6CM
—◆— Rho	0.91	0.88	0.875	0.87	0.86
—■— Lambda	0.99	0.98	0.96	0.93	0.9
—▲— Gamma	2	3	4	5	6

Figure 7.13: λ, ρ, γ values for Matrix Multiplication

Chapter 8

Summary and Conclusions

Following the investigation into parallel computing by means of a variable computer cluster conducted and presented in this dissertation some final comments are required. Parallel programming is much more difficult than sequential programming. Programming for good performance requires much work, especially in determining a good parallelization. Significant amount of labour is required to implement and orchestrate parallel programs and debugging such programs is not a trivial task. The task is difficult because of the interactions among multiple processes with their own program orders, and because of sensitivity of timing. Depending on when events in one process happen to occur relative to events in another process, a bug in the program may or may not manifest itself at run time in a particular execution.

Our research indicates that computer clusters are viable alternatives to mainframes for computation intensive applications. Applications that require little I/O are especially suited for distributed memory clusters, such as the one that has been designed. The biggest challenge posed by the developed machine was the process of mapping data onto the nodes. Ideally the data would be evenly distributed so that the whole machine participates in the computations. At the same time, it is important to position data "close" to other data it participates into, because communication is very expensive. At any rate, it takes a fair amount of manual intervention and custom crafting to develop a code that can run in parallel. Parallelism in an application is often expressed serially in a fashion that obscures whatever parallelism once existed. Converting a sequential algorithm to a parallel equivalent involves hard work and hand tuning. The system designer has to coordinate the activities of the different processors explicitly, usually through message passing.

The main idea behind the conducted research was to design and build a distributed computing cluster and to analyze its performance. The emphasis was put on creating an open platform that could be used for development of engineering applications requiring greater computing power than regular workstations can deliver. Several factors influenced the design of the cluster. The most notable factors include utilization of standard, off-the-shelf hardware, adaptation of standard operating system and networking software, scalability and expendability, high performance to price ratio, and flexibility and ease of configuration. By building an initial implementation of the distributed computing cluster, hands-on experience has been acquired, which shows that a first phase distributed system can be built with an acceptable level of functionality. However, implementing a distributed computing cluster is a challenging task. The obtained results show that this computing concept is feasible and that it can be implemented efficiently on low cost hardware. The developed variable cluster can be used to run engineering applications that require great processing power. Computing kernels for matrix multiplication, 1-D and 2-D FFT, and electric field calculator were designed and implemented. While clusters are built on a regular basis, little research has been done in the modeling of their performance. Data collected during the experiments were used to develop models of the cluster while performing matrix multiplication in discrete and continuous domains. Accurate models were developed and compared with the collected data.

Clusters offer great performance at a low cost. The research indicates that it is important to match a problem to a machine. Distributed computing requires partitioning of the problem and orchestration of the computations. It was observed that I/O intensive problems do not benefit from cluster technologies. A simple formula

$$T_{I/O} < T_{FloatingPointOperations}$$

is proposed for a quick assessment of the applicability of the designed cluster to a given problem. Implementations where more than 50% of time is spent on I/O do not benefit from the designed cluster architecture. The ideal candidate for a cluster application has a computational complexity of $O(n^2)$ or greater. Sample applications include: matrix operations (imaging operations) and grid operations (simulations).

The collected results obtained from several applications run on the cluster allowed for the analysis of its performance. Data were used to calculate system SpeedUp and selected sets of cases served to develop models of the experimental system. Two cluster models, discrete and continuous, were advanced. The close match of the developed models with our data suggests that the computational cluster may undergo damp oscillations during its operation. Although this may not always be the case, a significant number of the experimental plots suggests that. An immediate conclusion to this observation is that the performance of the computer cluster may at times be highly dependent on the "response" frequency of the system when processing different computational load. The model inherently signals the oscillatory properties of the cluster.

PC clusters are commonly used for conducting scientific calculations. The absolute performance of such clusters is not attractive compared to massively parallel processors, because the performance of interconnecting networks is not good enough, especially with communication intensive applications. However, a good cost to performance ratio can be achieved in these clusters. Such systems are interesting as research prototypes, but none of them has been accepted as a common platform. Distributed memory parallel machines are the only vehicle for applying many processors to an individual problem. However, quite often the performance of systems employing multiple processors does not scale or increase at a satisfactory rate with the number of processors available for computations. There are many advantages of these systems that can be custom tailored to an application. The designer is not restricted to generic implementations available on the market. A custom tailored system can be used to process data available in any form and anywhere. Computations can also be scheduled at times when computers are idling. Since the cluster server is aware of all available cluster members, it can assign the data and collect the results of computations when they become available. If failure of a cluster member is detected, it would be possible to reassign the failed cluster member's data to a member that has finished computations. With multiple cluster members a high degree of redundancy can be achieved. Cluster computing does not come without a price. In order to benefit from the cluster's power one needs to develop programs that utilize the hardware efficiently. Frequently it is difficult, and sometimes impossible, to convert a sequential program into a parallel equivalent. From our study it follows that problems that require much communication are not well suited for a cluster implementation.

8.1 Recommendations for Future Work

The developed system performed at a satisfactory level. Several aspects could be improved or optimized to increase the overall performance of the system. The sections below address the most notable ones.

The cluster does not utilize the cluster server during computations. The primary role of the server, aside from cluster management and task allocation, was to record accurate measurements of execution times during experiments. The server of course could be utilized to perform computations on a set of data. The communications with the cluster members would be reduced and the overall performance would certainly increase.

The I/O operations are synchronous. The computation is not started unless all data are received. Since the data on which the computations are performed are stored in consecutive memory locations, it would be possible to start computations as soon as a set of data is received. In addition, the partially computed results could be sent to the server as soon as they are available. Such optimization would especially benefit the 2D-FFT application, where a large portion of the execution time is devoted to I/O.

The experiment illustrated the applicability of a distributed computing cluster to perform computations of the selected engineering application. Very little optimization has been performed. The primary concern was the correctness of the results. It would be possible to tune the code, especially when it comes to memory references during matrix multiplication operation. Multi-threaded routines could also be added for computations and the overlapped I/O, as discussed above.

Basic fault tolerance has been implemented in the experimental system. The server is capable of recognizing a crashed cluster member. When such a problem is detected, the server continues to run and collects results of the computation from the running cluster members. The server then notifies the operator about the cluster member that failed and the problem can be addressed by the operator. However, such failures cause the whole computation to fail, as there are no results from the machine to which the computation was assigned. A possible improvement would involve an assignment of the data belonging to the faulty cluster member to the first cluster member to finish its assigned computations.

Cluster management tasks, such as cluster member registration and computational power assessment, are performed manually. The operator must also know how many cluster members will participate in the experiment/calculations before he/she schedules any computations on the cluster. Such tasks could be automated. Cluster members could be added and removed dynamically to and from a database maintained by the server. Machines willing to participate in the cluster could be given a pre-registration assessment test whose results would be used to rank the computational power of the participant. By the same token, the removal of cluster members could be automated. For example, any failure detected during computations would cause de-registration of the cluster member.

Appendix A

Cluster Program Listings

The cluster code consists of three parts. The first part contains library functions utilized by both the server and cluster member. The second part contains the client code and finally the third part contains the server code. For the sake of brevity only the 2D-FFT code for the server and the members has been included in this appendix. The code for latency, datatrasfer and matrix multiplication is very similar to the one listed below.

A.1 Cluster Libraries

In order to simplify the development of the cluster server and the cluster memebers several auxiliary libraries have been implemented. The library functions are responsible for handling socket communications, matrix operations and database connectivity.

A.1.1 Socket Library

socket.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
int readbuffer( int socket, void * buffer, int bytes );
int writebuffer( int socket, void * buffer, int bytes );

// Define an Internet address given a host and port.
setaddr(sp, host, port)
  struct sockaddr_in *sp;
  char *host;
  int port;
{
  struct hostent *hp;
  hp = gethostbyname(host); /* searches /etc/hosts */
  if (hp == NULL) {
     fprintf(stderr, "%s: unknown host\n", host);
     exit(1);}
  sp->sin_family = AF_INET;
  bcopy(hp->h_addr, &sp->sin_addr, hp->h_length);
  sp->sin_port = htons(port);
}

// Create a stream socket and bind it to the given port number.
int streamsocket(int port)
{
  int s;
  struct sockaddr_in sin;

  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = INADDR_ANY; /* shorthand for 'this host' */
  /* htons() converts the port number to network byte order */
  sin.sin_port = htons(port);
  s = socket(AF_INET, SOCK_STREAM, 0);
  if (s < 0)
     error("socket");
  if (bind(s, &sin, sizeof sin) < 0)
     error("bind");
  return s;
}

// System call failed: print a message and give up.
error(char *msg;)
{
  extern char *myname; /* program name */

  fprintf(stderr, "%s: ", myname);
  perror(msg);
  exit(1);
}

// Function readbuffer ensures that the entire expected data has been read
int readbuffer( int socket, void * buffer, int bytes )
{
  int count=0;
  int br;

  while (count < bytes) { /* loop until full buffer */
    if ((br = read(socket ,buffer, bytes-count)) > 0) {
      count += br; /* increment byte counter */
      buffer += br; /* move buffer ptr for next read */
    }
    if (br < 0) /* signal an error to the caller */
      return(-1);
  }
  return(count);
}

// Function readbuffer ensures that the entire expected data has been sent
int writebuffer( int socket, void * buffer, int bytes )
{
  int count=0;
  int br;

  while (count < bytes) { /* loop until full buffer */
    if ((br = write(socket ,buffer, bytes-count)) > 0) {
      count += br; /* increment byte counter */
      buffer += br; /* move buffer ptr for next read */
    }
    if (br < 0) /* signal an error to the caller */
      return(-1);
  }
  return(count);
}
```

A.1.2 Database Library

sqllib.h

```
#define MSGSIZ 1
#define BUFFER 1024
#include <stdio.h>
#include <stdlib.h>
#include <mysql/mysql.h>

void exiterr( int exitcode );  // MySQL error handling function
int OpenDB( char *DB );        // Open Database DB
int CloseDB();                 // Close Open Database
int CreateTable( char *name );     // Create Table name
int InsertData( char *table, int CPU, int Ether, float Data, float Time, float CPUTime, float IOTime  );
int ShowTable( char * table);  // Show Table table

MYSQL mysql;
MYSQL_RES *res;
MYSQL_ROW row;
```

sqllib.c

```
#include "sqllib.h"

// Create a table for an experiment in the research database
int CreateTable( char *name )
{
  char sqlStr[1024];
  char definition[1000];
  strcpy(sqlStr,"CREATE TABLE ");
  strcat(sqlStr, name);
  strcat(sqlStr, "\n (ExpID INT NOT NULL AUTO_INCREMENT,\n");
  strcat(sqlStr, "Date TIMESTAMP(14),\n");
  strcat(sqlStr, "HostCPU INT NOT NULL,\n");
  strcat(sqlStr, "Ethernet INT NOT NULL,\n");
  strcat(sqlStr, "DataSet FLOAT (10,2) NOT NULL,\n");
  strcat(sqlStr, "RunTime FLOAT (6,2) NOT NULL,\n");
  strcat(sqlStr, "CPUTime FLOAT (6,2) NOT NULL,\n");
  strcat(sqlStr, "IOTime FLOAT (6,2) NOT NULL,\n");
  strcat(sqlStr, "PRIMARY KEY (ExpID))\n");

  if (mysql_query(&mysql, sqlStr))
    exiterr(3);

  return 0;
}
// Print an SQL error code
void exiterr( int exitcode )
{
  fprintf( stderr, "%s\n", mysql_error(&mysql) );
  exit (exitcode);
}

// Insert a record into a given table
int InsertData( char *table, int HostCPU, int Ethernet, float DataSet, float RunTime, float CPUTime, float IOTime )
{
  char sqlStr[1024];
  char values[1000];

  sprintf(sqlStr, "%s%s", "INSERT INTO ", table);
  strcat(sqlStr, "( HostCPU, Ethernet, DataSet, RunTime, CPUTime, IOTime )\n");
  sprintf(values, "VALUES (%d, %d, %f, %f, %f, %f )", HostCPU, Ethernet, DataSet, RunTime, CPUTime, IOTime);
  strcat(sqlStr, values);

  if (mysql_query(&mysql, sqlStr))
      exiterr(3);

  return 0;
```

```
}

// Show all records in a given table
int ShowTable( char *table )
{
  char sqlStr[1024];
  int i;

  sprintf(sqlStr,"%s%s", "SELECT * FROM ", table);

  if (mysql_query(&mysql, sqlStr))
    exiterr(3);
  if (!(res = mysql_store_result(&mysql)))
    exiterr(4);

  while(( row = mysql_fetch_row(res)))
    {
      for (i=0; i<mysql_num_fields(res); i++)
         printf("%s  ", row[i]);
      printf("\n");
    }

  if (!mysql_eof(res))
    exiterr(5);

  mysql_free_result(res);
  return 0;
}

// Open a database
int OpenDB( char * DB )
{
  if ( !(mysql_connect(&mysql,"asus2p3","root","")) )
    exiterr(1);
  if (mysql_select_db(&mysql, DB))
    exiterr(2);
}

// Close a database
int CloseDB()
```

```
{
  mysql_close(&mysql);
  return 0;
}
```

A.1.3 System Library

system.c

```
#include <stdio.h>

int getCPUInfo( float *);
int getSwaps( int *swapout, int *swapin);

int getCPUInfo(float *mhz)
{
  FILE *procfile;
  char buffer[80];

  *mhz = -1;
  procfile = fopen("/proc/cpuinfo","r");
  if(procfile == NULL )
    return(-1);

  while(fgets(buffer, 80, procfile))
    if (strncmp(buffer, "cpu MHz", 7)==0){
        sscanf(&buffer[11],"%f", mhz);
        break;}
  fclose(procfile);
  return(0);
}

int getSwaps( int *swapout, int *swapin)
{
  FILE *procfile;
  char buffer[80], temp[80];

  *swapout = *swapin = -1;
  procfile = fopen("/proc/stat","r");
  if(procfile == NULL )
```

```
    return(-1);

  while(fgets(buffer, 80, procfile))
    if (strncmp(buffer, "swap", 4)==0){
       sscanf(buffer,"%s %d %d", temp, swapin, swapout);
       break;}
  fclose(procfile);
  return(0);
}
```

A.1.4 Matrix Library

arrayops.h

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

void PrintMatrix (float *M, int a, int b);
void CreateRandomMatrix (float *M, int a, int b);
void CreateIdentityMatrix (float *M, int a, int b);
void CreateOnesMatrix (float *M, int a, int b);
void RotateMatrix (float *M, int rows, int cols);
void PrintMatrix (float *M, int a, int b);
int CompareMatrix (float *M1, float *M2, int a, int b);
int PopulateMatrix (float *M1, float *M2, int a, int b);
int MultiplyMatrix(float *, int, int, float *, int, int, float *);
int GetRows( float *Source, int SRows, int SCols, int StartRow, int EndRow, float *Dest);
int GetCols( float *Source, int SRows, int SCols, int StartCol, int EndCol, float *Dest);
void SortMatrix( float *M, int rows, int cols);
float ExpTime( struct timeval, struct timeval );
```

arrayops.c

```
#include "arrayops.h"
// Function for sorting elements of a matrix
void SortMatrix (float *M, int rows, int cols)
{
  int count, i, j;
  float temp;

  count = rows*cols;

  for(i=0;i<count;i++)
    for(j=i;j<count;j++)
      if( *(M+i) > *(M+j) ){
         temp = *(M+i);
         *(M+i) = *(M+j);
         *(M+j) = temp;}
}
// Function for "rotating" a matrix, rows become cols
void RotateMatrix (float *M, int rows, int cols)
{
  float *temp;
  int offset, rowcount, colcount,n,i=0;

  if( (temp = (float *)malloc(rows*cols*sizeof(float))) == NULL ){
     printf("Cannot allocate mem for rotating matrix");
     exit(-1);}

  for(colcount=0; colcount<cols; colcount++){
     offset = cols*(rows-1) + colcount;
     for(rowcount=0; rowcount <rows; rowcount++){
       *(temp+i++) = *(M+offset);
       offset -= cols;}
     }
  n = rows*cols;
  for(i=0;i<n;i++)
    *(M+i) = *(temp+i);
  free(temp);
}
// Function of printing elements of a matrix in a human readable form
void PrintMatrix (float *M, int a, int b)
{
  int i, j;
  for( i=0;i<a;i++ ){
    for( j=0;j<b;j++ )
      printf("%.2f ", *(M+(i*b+j)));
    printf("\n");
    }
}
// Function for multiplying two arbitrarily sized matrices, no error checks
int MultiplyMatrix(float *a, int aRow, int aCol, float *b, int bRow, int bCol, float *c)
{
  int x, y, z;

  for(z=0;z<aRow;z++){
    for(y=0;y<bCol;y++){
      *(c+(z*bCol+y))=0;
      for(x=0;x<aCol;x++)
        *(c+(z*bCol+y)) += *(a+(z*aCol+x)) * *(b+(x*bCol+y));
    }
  }
  return z*y*x;
}
// Function for populating a matrix with random data
void CreateRandomMatrix (float *M, int a, int b)
{
  int i, number;

  number = a*b;
  srand(time(NULL));

  for(i=0;i<number;i++)
    *(M+i) = rand();
}
// Function for populating a matrix with 1's
void CreateOnesMatrix (float *M, int a, int b)
{
  int i, number;
  number = a*b;

  for(i=0;i<number;i++)
    *(M+i) = 1.0;
}
// Function for creating an Identity matrix
void CreateIdentityMatrix (float *M, int a, int b)
{
  int i, number, offset;
  number = a*b;
  offset = a+1;
  *M = 1;
  for(i=1;i<number;i++)
    if(i==offset)
      *(M+i) = 1; offset += a+1;
    else
      *(M+i)=0;
}
// Function for comparing the contents of two matrices
int CompareMatrix (float *M1, float *M2, int a, int b)
{
  int i, count;

  count = a*b;
  for(i=0;i<count;i++)
    if(*(M1+i) != *(M2+i))
      return -1;
  return 0;
}
// Function for copying a matrix
int PopulateMatrix (float *M1, float *M2, int a, int b)
{
  int i, n;

  n = a*b;

  for(i=0;i<n;i++)
    *(M1+i) = *(M2+i);
  return n;
}
//Function GetRows assigns rows of data from matrix source to matrix dest
//It returns number of assigments performed
int GetRows( float *Source, int SRows, int SCols, int StartRow, int EndRow, float *Dest)
{
  int offset, counter, end;

  offset = StartRow*SCols;
  end = EndRow*SCols + SCols;
  for( counter=offset; counter<end; counter++ )
    *(Dest + (counter-offset)) = *(Source + counter);
  return (counter-offset);
}
//Function GetCols assigns columns of data from matrix source to matrix dest
//It returns number of assigments performed
int GetCols( float *Source, int SRows, int SCols, int StartCol, int EndCol, float *Dest)
{
  int offset, OffCounter, ElCounter, endElCounter;
  int destcount = 0;
  float t;

  endElCounter = EndCol - StartCol;
  offset = 0;

  for( OffCounter = 0; OffCounter < SRows; OffCounter++ ){
    offset = StartCol + SCols * OffCounter;
    for( ElCounter = 0; ElCounter <= endElCounter; ElCounter++ )
      t = *(Dest + destcount++) = *(Source + (offset + ElCounter));
    }
  return destcount;
}
// Function ExpTime returns expired time between starttv and endtv events
float ExpTime( struct timeval starttv, struct timeval endtv )
{
  float ETime=0;
  float fraction=0;
  ETime =  endtv.tv_sec - starttv.tv_sec;
  fraction = endtv.tv_usec - starttv.tv_usec;
  fraction /= 1000000;

  if (fraction < 0){
      fraction = -fraction;
      ETime = ETime - 1 + fraction;
    }
  else
    ETime = ETime + fraction;
  return (ETime);
}
```

A.2 2D-FFT Code

A.2.1 Server

clusterserver.c

```
/*
 * clusterserver port
 * Cluster Server.
 * CS generates data and sends it to clustermembers for computations.
 * CS collects the results and records the execution time in DB.
 *    Assumption: all cluster memebers have the same computing power.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/time.h>
#include <unistd.h>
#include <ctype.h>
#include <pthread.h>
#include <signal.h>

#include "arrayops.h"
#include "sqllib.h"

#define REPETITIONS 10
#define STARTSIZE 128
#define ENDSIZE 4096

int MAXMEMBERS;
char *myname, *port;

void *computeMM(void *arg);  // Thread function
int ComputeFFT(float *matrix, int size, float * timeStats);
void signal_handler(int signal);

typedef struct{
  int socket;             // destination socket
  float *matrix;          // start address of data matrix
  int arrdims[2];           // AX, AY
  float volatile *result;  // start address of the result matrix
  pthread_mutex_t *lock;   // lock for locking the access to timeStats
  float volatile *timeStats;  // IO, CPU
  int threadID;            // Thread ID
} threadData;

int *s;          // Communication socket variables, allocated dynamically
struct sockaddr_in *sint;

char *cmName[4] = {"cm4","cm3","cm2","cm1"}; //Computer names of cluster members

pthread_t * cmThread; //Thread variables
threadData * cmData;
pthread_mutex_t lock;

main(argc, argv)
char *argv[];
{
  int arrdims[2];
  int n, zero, rval, counter, membercount, offset, repeat;
  struct timeval starttv, endtv;
  struct timezone tz;
  char  table[80];
  float *matrix;
  float expTime[REPETITIONS], expTimeAve;
  float *timeStat, IOTimeAve, CPUTimeAve;
  signal(SIGPIPE, signal_handler); // Try to catch CM fault signals

  myname = argv[0];

  if (argc < 3) {
    fprintf(stderr, "usage: %s port members [table]\n", myname);
    exit(1);}
  port = argv[1];
  MAXMEMBERS = atoi(argv[2]);

  if (MAXMEMBERS > 0 && MAXMEMBERS < 5){
    for(n=0;n<MAXMEMBERS;n++)
    printf("%s ",cmName[n]);
  printf("machines that will participate in the experiment\n");}
else{
  printf("Currently only 1 to 4 machines can participate in the experiment\n");
  exit(-1);}
s = (int *) calloc(MAXMEMBERS, sizeof(int)); // Initialize Communication sockets
if(s == NULL){
  fprintf(stderr,"Cannot allocate memory for communication sockets!");
  exit(2);}
sint = (struct sockaddr_in *) calloc(MAXMEMBERS, sizeof(struct sockaddr_in));
if(sint == NULL){
  fprintf(stderr,"Cannot allocate memory for communication structs!");
  exit(2);}
timeStat = (float *) malloc(2*REPETITIONS*sizeof(float));
if(timeStat == NULL){
  fprintf(stderr,"Cannot allocate memory for time stats!");
  exit(2);}
cmThread = (pthread_t *)calloc(MAXMEMBERS, sizeof(pthread_t));
if(cmThread == NULL){
  fprintf(stderr,"Cannot allocate memory for threads!");
  exit(2);}
cmData = (threadData *)calloc(MAXMEMBERS, sizeof(threadData));
if(cmData == NULL){
  fprintf(stderr,"Cannot allocate memory for thread data");
  exit(2);}
if (argc == 4){
  OpenDB( "research" );
  strcpy(table, argv[3]);
  printf("Results will be stored in research.%s table.\n",table);
  CreateTable( table ); }
else
  printf("Results will not be recorded\n");

for(counter = STARTSIZE;counter<=ENDSIZE;counter = counter*2){
  for(repeat = 0;repeat<REPETITIONS; repeat++){
    *(timeStat + 2*repeat) = *(timeStat + 2*repeat +1) = 0;
    arrdims[0] = arrdims[1] = counter;
    matrix = (float *) malloc(arrdims[0]*arrdims[1]*sizeof(float));
    CreateOnesMatrix(matrix, arrdims[0], arrdims[1]);
    gettimeofday(&starttv, &tz);
```

```
      ComputeFFT(matrix, counter, (timeStat + 2*repeat));
      RotateMatrix(matrix, arrdims[0], arrdims[1]);
      ComputeFFT(matrix, counter, (timeStat + 2*repeat));
      gettimeofday(&endtv, &tz);
      *(timeStat + 2*repeat) /= MAXMEMBERS;  //Normalize Stats
      *(timeStat + 2*repeat+1) /= MAXMEMBERS;

      // Correctnes check.  Element 0,0 should be ~ A*B
      printf("M[0][0]: %f Should be close to: %f\n",*matrix, (float)arrdims[0]*arrdims[1]);
      free (matrix);     // free memory for next round of computations
      expTime[repeat] = ExpTime( starttv, endtv );
      printf("Run: %d Time: %.2f CPU %.2f IO: %.2f\n",repeat+1,expTime[repeat],*(timeStat+2*repeat+1),
                                                   *(timeStat+2*repeat));
    }
    SortMatrix(expTime, 1, REPETITIONS);
    PrintMatrix(expTime, 1, REPETITIONS);
    RotateMatrix(timeStat, REPETITIONS, 2);
    SortMatrix(timeStat, 1, REPETITIONS);
    SortMatrix((timeStat+REPETITIONS), 1, REPETITIONS);
    PrintMatrix(timeStat, 2, REPETITIONS);
    expTimeAve = IOTimeAve = CPUTimeAve = 0;
    for(n=1;n<REPETITIONS-1;n++){
      expTimeAve += expTime[n];
      IOTimeAve += *(timeStat + n);
      CPUTimeAve += *(timeStat + REPETITIONS + n );}
    if(REPETITIONS>2){
      expTimeAve /= (REPETITIONS-2);
      IOTimeAve /= (REPETITIONS-2);
      CPUTimeAve /= (REPETITIONS-2);}
    if(argc == 4) // record result in db if required
      InsertData( table, 120, 10, (float)arrdims[0]*arrdims[1]*sizeof(float), expTimeAve, CPUTimeAve, IOTimeAve);
    printf("Average Time expired: %.2f CPU: %.2f IO: %.2f\n",expTimeAve, CPUTimeAve, IOTimeAve);}
    if(argc == 4){ // Show results recorded in db
      ShowTable(table);
      CloseDB();}
   exit(0);
}

void *computeMM(void *arg)
{
  threadData td = *(threadData *) arg;
  float timeStats[3];  //temp buffer for stats

  if (writebuffer(td.socket, td.arrdims, sizeof(td.arrdims)) < 0)
    error("writing array dimmensions");
  td.arrdims[0] = td.arrdims[1] = 0;
  if (readbuffer(td.socket, td.arrdims, sizeof(td.arrdims)) < 0)
    error("reading arrdims confirmation");
  if (writebuffer(td.socket, td.matrix, td.arrdims[0]*td.arrdims[1]*sizeof(float)) < 0)
    error("writing first array");
  if (readbuffer(td.socket, td.result, td.arrdims[0]*td.arrdims[1]*sizeof(float)) < 0)
    error("reading result array");
  if (readbuffer(td.socket, timeStats, sizeof(timeStats)) < 0)
    error("reading time stats");
  pthread_mutex_lock(td.lock); // obtain lock for shared data
  *(td.timeStats) += timeStats[0] + timeStats[1]; // IO in plus IO out
  *(td.timeStats+1) += timeStats[2];              // CPU Time
  pthread_mutex_unlock(td.lock);  //release lock
  printf("[%d] Finished, CPU: %.2f, IO: %.2f\n",td.threadID,*(td.timeStats+1),*(td.timeStats));
}

int ComputeFFT(float *matrix, int size, float *timeStats)
{
  int membercount, offset = 0;
  printf("Connecting on port %s\n", port);
  for(membercount = 0; membercount<MAXMEMBERS; membercount++){
   s[membercount] = streamsocket(0); /* port 0 means "any port" */
   setaddr(&sint[membercount], cmName[membercount], atoi(port));

   /* connect a socket using port specified by the command line */
   if (connect(s[membercount], &sint[membercount], sizeof(sint[membercount])) < 0) {
     error("connecting stream socket");
     exit(1);}

   printf("Offset: %d\n", offset);
   cmData[membercount].socket = s[membercount];
   cmData[membercount].matrix = (matrix + offset);
   if(membercount != (MAXMEMBERS-1))
      cmData[membercount].arrdims[0] = size/MAXMEMBERS;
   else
cmData[membercount].arrdims[0] = size - (size/MAXMEMBERS)*(MAXMEMBERS-1);

   cmData[membercount].arrdims[1] = size;
   cmData[membercount].result = (matrix + offset);
   cmData[membercount].lock = &lock;
   cmData[membercount].timeStats = timeStats;
   cmData[membercount].threadID = membercount;

   pthread_create( &cmThread[membercount],
   NULL,
   computeMM,
   &cmData[membercount] );

   offset += size*(size/MAXMEMBERS);
 }
 for(membercount = 0; membercount<MAXMEMBERS; membercount++){
   pthread_join(cmThread[membercount], NULL);    // wait for threads to finish
   close(s[membercount]);
 }
 return 1;
}
void signal_handler(int sig)
{
  printf("A communication error has occured.\n");
}
```

A.2.2 Cluster Member

```
/*
 * FFT Cluster Member
 * Internet stream server.
 * Receives vectors of floats and sends FFT of them
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <signal.h>
```

```
#include "arrayops.h"

#define MSGSIZ 1

void signal_handler(int);

char *myname;
int msgs; //socket descriptor

main(argc, argv)
char *argv[];
{
  struct sockaddr_in from;
  int s, n, fromlen, rval;
  int arrdims[2];
  float timeStats[3]; //*timeStats;
  struct hostent *hp;
  char buf[BUFSIZ];
  float *matrix;
  struct timeval startComp, endComp, startIO, endIO;
  struct timezone tz;
  float CPU;
  int swapOutStart, swapInStart, swapOutEnd, swapInEnd;

  myname = argv[0];
  if (argc < 2) {
    fprintf(stderr, "usage: %s port\n", argv[0]);
    exit(1);}
  signal( SIGPIPE, signal_handler ); // Try to catch I/O faults
  timeStats[0] =  timeStats[1] =  timeStats[2] = 0;
  s = streamsocket(atoi(argv[1]));
  fromlen = sizeof(from);
  if (getsockname(s, &from, &fromlen)) {
    error("getting socket name");
    exit(1);}
  printf("Socket has port #%d\n",ntohs(from.sin_port));
  listen(s, 5);
  printf("Raw Transfer Rates\n");
  printf("HostCPU | Ethernet | DataSet | RecIOTime | SendIOTime | CPUTime SwapOut SwapIn\n");
  for (;;) {
    msgs = accept(s, 0, 0); /* start accepting connections */
    if (msgs == -1)
      error("accept");
    getSwaps(&swapOutStart, &swapInStart);
    bzero(arrdims, sizeof(arrdims));
    if (readbuffer(msgs, arrdims, sizeof(arrdims)) < 0){
      printf("Error reading arrdims\n");
      goto end;}
    if( writebuffer(msgs, arrdims, sizeof(arrdims)) < 0 ){
      printf("Error writing arrdims\n");
      goto end;}
    matrix = (float *)malloc(arrdims[0]*arrdims[1]*sizeof(float));
    bzero(matrix, sizeof(matrix));
    gettimeofday(&startIO, &tz);
    if (readbuffer(msgs, matrix, arrdims[0]*arrdims[1]*sizeof(float)) < 0){
      printf("Error reading matrix\n");
      goto end;}
    gettimeofday(&endIO, &tz);
    timeStats[0] = ExpTime(startIO, endIO);
    gettimeofday(&startComp, &tz);
    FFT_Matrix(matrix, arrdims[0], arrdims[1]);
    gettimeofday(&endComp, &tz);
    timeStats[2] = ExpTime(startComp, endComp);

    gettimeofday(&startIO, &tz);
    if (writebuffer(msgs, matrix, arrdims[0]*arrdims[1]*sizeof(float)) < 0){
      printf("Error writing result.\n");
      goto end;}
    gettimeofday(&endIO, &tz);
    timeStats[1] = ExpTime(startIO, endIO);
    if (writebuffer(msgs, timeStats, sizeof(timeStats)) < 0)
      printf("Error writing time stats\n");
    end:
    getCPUInfo(&CPU);
    getSwaps(&swapOutEnd, &swapInEnd);
    printf("%f  100  %d  %f  %f  %f %d %d\n", CPU, arrdims[0]*arrdims[1]*sizeof(float),timeStats[0], timeStats[1], t
                                            swapOutEnd-swapOutStart, swapInEnd-SwapInStart);
    free(matrix);
  }
}
```

```
void signal_handler(int sig)
{
  printf("\nI/O error has occurred (Broken pipe).\nAttempting to resume normal operation.\n");
  signal(SIGPIPE, signal_handler);
}
```

Bibliography

[1] Marha L. Abell and James P. Braselton. *Differential Equations with Maple V.* Academic Press, 2 edition, 2000.

[2] Vikram S. Adve. *Analyzing the behavior and performance of parallel programs.* Ph.D. Thesis, University of Wisconsin-Madison, 1993.

[3] Ahmad Afsadi and Nikitas J. Dimopoulos. **Efficient Communication Using Message Prediction for Cluster of Multiprocessors**. In Babak Falsafi and Mario Lauria, editors, *Network-Based Parallel Computing*, volume 1797 of *Lecture Notes In Computer Science.* Springer, 2000.

[4] Selim G. Akl. *Parallel computation: models and methods.* Prentice-Hall, Inc., 1997.

[5] A.M. Alkindi, D.J. Kerbyson, and G.R. Nudd. **Run-Time Optimization Using Dynamic Performance Prediction**. In Roy Williams Marian Bubak, Hamideh Afsarmanesh and Bob Hertzberger, editors, *High Performance Computing and Networking*, volume 1823 of *Lecture Notes In Computer Science.* Springer, 2000.

[6] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing.* The Benjamin/Cummings Publishing Company Inc., 2 edition, 1994.

[7] Gene Amdahl. **Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities**. In Group, editor, *AFIPS'67.* AFIPS, 1967.

[8] Françoisce André, Christine Morin, and Maria-Teresa Segarra. **Mechanisms for Global Processor and Memory Management on a NoW**. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes In Computer Science.* Springer, April 1998.

[9] Cosimo Anglano. **Predicting Parallel Applications Performance on Non-dedicated Cluster Platforms**. Internet, 1998. *http://citeseer.nj.nec.com/anglano98predicting.html.*

[10] Kubota Atushi, Tatsumi Shogo, Tanaka Toshihiko, and Mori Shin-ichiro. **A Technique to Eliminate Redundant Inter-Process Communications on Parallelizing Compiler TINPAR**. In J. Harmanis G. Goos and J. van Leeuwen, editors, *High Performance Computing, ISHPC'97*, Lecture Notes In Computer Science. Springer, 1997.

[11] Matthias Brune Axel Keller and Alexander Reinefeld. **Resource Management for High-Performance PC Clusters**. In Alfons Hoekstra Peter Sloot, Marian Bubak and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1593 of *Lecture Notes In Computer Science.* Springer, 1999.

[12] F. Baiardi, P. Becuzzi, P. Mori, and M. Paoli. **Load Balancing and Locality in Hierarchical N-body Algorithms on Distributed Memory Architecture**. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes In Computer Science.* Springer, April 1998.

[13] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. **The NAS Parallel Benchmarks**. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[14] Mark Baker and Rajkumar Buyya. **Cluster Computing at a Glance**. In Buyya Rajkumar, editor, *High Performance Cluster Computing*, volume 1, chapter 1, pages 3–47. Prentice Hall Inc, 1999.

[15] Manjunath Bangalore and Anand Sivasubramaniam. **Remote Subpaging Across a Fast Network**. In Dhabaleswar K. Panda and Craig B. Stunkel, editors, *Network-Based Parallel Computing*, volume 1362 of *Lecture Notes In Computer Science.* Springer, February 1998.

[16] Pierrick Beaugendre and Thierry Priol. **A Client/Server Approach for HPC Applications within a Networking Environment**. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes In Computer Science.* Springer, April 1998.

[17] J. Błasiak and W. Dzwinel. **Visual Clustering Multidimensional and Large Data Sets Using Parallel Environments**. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes In Computer Science.* Springer, April 1998.

[18] Mathias Brune, Jörn Gehring, and Alexander Reinefeld. **A Lightweight Communication Interface for Parallel Programming Environments**. In Bob Hetzberger and Peter Sloot, editors, *High-Performance Computing and Networking, HPCN'97*, volume 1225 of *Lecture Notes In Computer Science.* Springer, 1997.

[19] Marian Bubak, Włodzimierz Funika, and Jacek Mościński. **Performance Analysis Environment for Parallel Applications on Networked Workstations**. In Bob Hetzberger and Peter Sloot, editors, *High-Performance Computing and Networking, HPCN'97*, volume 1225 of *Lecture Notes In Computer Science.* Springer, 1997.

[20] Robin Burk, Martin Bligh, and Thomas Lee. *TCP/IP Blueprints.* Sams Publishing, 1 edition, 1997.

[21] Duncan K.G. Campbell. **A Survey of Models of Parallel Computation**.

[22] Eddy Caron, Olivier Cozette, Dominique Lazure, and Gil Utard. **Virtual Memory Management in Data Parallel Applications**. In Alfons Hoekstra Peter Sloot, Marian Bubak and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1593 of *Lecture Notes In Computer Science.* Springer, 1999.

[23] Alan Chalmers and Jonathan Tidmus. *Practical Parallel Processing.* International Thomson Computer Press, 1996.

[24] Steven C. Chapra and Raymond P. Canale. *Numerical Methods For Engineers.* McGraw-Hill Book Company, 1988.

[25] Helen Chen and Pete Wyckoff. **Simulation Studies of Gigabity Ethernet Versus Myrinet Using Real Application Cores**. In Babak Falsafi and Mario Lauria, editors, *Network-Based Parallel Computing*, volume 1797 of *Lecture Notes In Computer Science.* Springer, 2000.

[26] Hsin-Chu Chen, Alvin Lim, and Nazir A. Warsi. **Multilevel Master-Slave Parallel Programming Models**. In Joxan Jafar and

Roland H.C. Yap, editors, *Concurrency and Parallelism, Programming, Networking and Security*, volume 1179 of *Lecture Notes In Computer Science*. Springer, December 1996.

[27] Wai-Kai Chen. *The Circuits and Filters Handbook*. CRC Press Inc., 1995.

[28] Eleanor Chu and Alan George. *Inside the FFT black box. Serial and Parallel Fast Fourier Transform Algorithms*. CRC Press, 2000.

[29] M. Clement and M. Quinn. **Analytical Performance Prediction on Multicomputers**. Internet, May 1993. *http://citeseer.nj.nec.com/clement94analytical.html*.

[30] Albert Cohen. **Parallelization via Constrained Storage Mapping Optimization**. In A Fukuda C Polychronopoulos, J. Kazuki and S Tomita, editors, *High Performance Computing, ISHPC'99*, Lecture Notes In Computer Science. Springer, 1999.

[31] David J. Comer. *Computer Analysis of Circuits*. International Textbook Company, 1971.

[32] George Coulouris, Jean Dollimore, and Tim Kinderberg. *Distributed Systems Concepts and Design*. Addison-Wesley, 4 edition, 1996.

[33] Michel Courson, Alan Mink, Guillaume Marçais, and Benjamin Traverse. **An Automated Benchmarking Toolset**. In Roy Williams Marian Bubak, Hamideh Afsarmanesh and Bob Hertzberger, editors, *High Performance Computing and Networking*, volume 1823 of *Lecture Notes In Computer Science*. Springer, 2000.

[34] Phyllis E. Crandal and Michael J. Quinn. **Block data decomposition for data parallel programming on a heterogeneous workstation network**. In *Proceedings of the Second International Symposium on High Performance Distributed Computing*, July 1993.

[35] Phyllis E. Crandall, Eranti V. Sumithasri, Johann Leichtl, and Mark A. Clement. **Toward Massive Dual-Level Parallelism in Cluster Computing**. Internet. *http://citeseer.nj.nec.com/239483.html*.

[36] Paolo Cremonesi, Claudio Gennaro, and Roberto Marega. **I/O Performance in Hybrid MIMD+SIMD Machines**. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing*

and Networking, volume 1401 of *Lecture Notes In Computer Science.* Springer, April 1998.

[37] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. **LogP: Towards a realistic model of parallel computation**. Internet, May 1993. *http://citeseer.nj.nec.com/culler93logp.html.*

[38] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture.* Morgan Kaufmann Publishers, Inc., 1999.

[39] Hans de Goede. **Root over nfs clients & server Howto**. Internet, March 1999. *http://www.linux.org/docs/ldp/howto/Diskless-root-NFS-HOWTO.html.*

[40] Frank Dehne and Siang W. Song. **Randomized Parallel List Ranking for Distributed Memory Multiprocessors**. In Joxan Jafar and Roland H.C. Yap, editors, *Concurrency and Parallelism, Programming, Networking and Security*, volume 1179 of *Lecture Notes In Computer Science.* Springer, December 1996.

[41] Jos Derksen and Harry Van den Akker. **Parallel Simulation of Turbulent Fluid Flow in a Mixing Tank**. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes In Computer Science.* Springer, April 1998.

[42] C.H.Q Ding, P.M. Lyster, J.W. Larson, J. Guo, and A. da Silva. **Atmospheric Data Assimilation on Distributed-Memory Parallel Supercomputers**. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes In Computer Science.* Springer, April 1998.

[43] Kevin Dowd. *High Performance Computing.* O'Reilly & Associates, Inc., first edition edition, 1993.

[44] D.E. Dudgeon and R.M. Mersereau. *Multidimenisional Digital Signal Processing.* Prentice-Hall Inc., 1984.

[45] Tzilla Erald, Baoling Sheen, and Novak V. Nastasic. **CHESSBOARD: A Synergy of Object Oriented Concurrent Programming and Program Layering**. In Joxan Jafar and Roland H.C. Yap, editors, *Concurrency and Parallelism, Programming, Networking and Security*,

volume 1179 of *Lecture Notes In Computer Science*. Springer, December 1996.

[46] Ertel. **On the Definition of Speedup**. In *PARLE: Parallel Architectures and Languages Europe*. LNCS, Springer-Verlag, 1994.

[47] Dror G. Feitelson. **Scheduling Parallel Jobs on Clusters**. In Buyya Rajkumar, editor, *High Performance Cluster Computing*, volume 1, chapter 21, pages 519–533. Prentice Hall Inc, 1999.

[48] A. Flores and J.M. Garcia. **Improving the Performance of Scientific Parallel Applications in a Cluster of Workstations**. In Erik Elmroth Bo Kågström, Jack Dongara and Jerzy Waśniewski, editors, *Applied Parallel Computing*, volume 1541 of *Lecture Notes In Computer Science*. Springer, 1998.

[49] Giuliana Fogaccia. **Parallel Implementation of a Lattice Boltzman Algorithm for the Electrostatic Plasma Turbulence**. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes In Computer Science*. Springer, April 1998.

[50] Gerald B. Folland. *Fourier Analysis and Its Applications*. Brooks/Cole Publishing Company, 1992.

[51] Bertil Folliot and Pierre Sens. **Load Sharing and Fault Tolerance Manager**. In Buyya Rajkumar, editor, *High Performance Cluster Computing*, volume 1, chapter 22, pages 535–552. Prentice Hall Inc, 1999.

[52] Robert Frank and Helmar Burkhart. **Application Support by Software Reuse: The ALWAN Approach**. In Bob Hetzberger and Peter Sloot, editors, *High-Performance Computing and Networking, HPCN'97*, volume 1225 of *Lecture Notes In Computer Science*. Springer, 1997.

[53] T.L. Freeman and C. Phillips. *Parallel Numerical Algorithms*. Prentice Hall, 1992.

[54] Antonio Augusto Fröhlich, Gilles Pokam Tientcheu, and Wolfgang Schröder-Preikschat. **EPOS and Myrinet: Effective Communication Support for Parallel Applications Running on Clusters of Commodity Workstations**. In Roy Williams Marian Bubak, Hamideh Afsarmanesh and Bob Hertzberger, editors, *High Performance*

Computing and Networking, volume 1823 of *Lecture Notes In Computer Science*. Springer, 2000.

[55] Xiadong Fu, Hua Wang, and Vijay Karamcheti. **Transparent Network Connectivity in Dynamic Cluster Environments**. In Babak Falsafi and Mario Lauria, editors, *Network-Based Parallel Computing*, volume 1797 of *Lecture Notes In Computer Science*. Springer, 2000.

[56] John Gilbert and Donald Kershaw. *Large-Scale Matrix Problems and the Numerical Solution of Partial Differential Equantions*. Oxford University Press, 1994.

[57] Frank R. Giordano and Maurice D. Weir. *Differential Equations a Modeling Approach*. Addison-Wesley Publishing Company, Inc., 1991.

[58] R. Grindley, T. Abdelrahman, S. Brown, S. Caranci, D. DeVries, B. Gamsa, A. Grbic, M. Gusat, R. Ho, and P. McHardy. The NUMAchine multiprocessor. Department of Electrical and Computer Engineering, University of Toronto, 2000.

[59] John L. Gustafson. **Reevaluating Amdahl's Law**. Internet, August 2000. *http://www.scl.ameslab.gov/Publications/AmdahlsLaw/Amdahls.html.*

[60] Issam Hamid and Ferhat Khendek. **A Dynamic Evolution for the Specifications of Distributed Systems**. In Joxan Jafar and Roland H.C. Yap, editors, *Concurrency and Parallelism, Programming, Networking and Security*, volume 1179 of *Lecture Notes In Computer Science*. Springer, December 1996.

[61] K.A. Havick, D.A. Grove, P.D. Coddington, and M.A. Buntine. **A Beowulf Cluster for Computational Chemistry**. In Roy Williams Marian Bubak, Hamideh Afsarmanesh and Bob Hertzberger, editors, *High Performance Computing and Networking*, volume 1823 of *Lecture Notes In Computer Science*. Springer, 2000.

[62] K.A. Hawick, H.A. James, C.J. Patten, and F.A. Vaughan. **DISCWorld: A Distributed High Performance Computing Environment**. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes In Computer Science*. Springer, April 1998.

[63] William H. Hayt and Jack E. Kemmerly. *Engineering Circuit Analysis.* McGraw-Hill, Inc., 1993.

[64] Bruce Hendrickson and Tamara G. Kolda. **Partitioning Sparse Rectangular Matrices for Parallel Computations of Ax and A^Tv*.** In Erik Elmroth Bo Kågström, Jack Dongara and Jerzy Waśniewski, editors, *Applied Parallel Computing*, volume 1541 of *Lecture Notes In Computer Science.* Springer, 1998.

[65] M. Hobs and A. Goscinski. **Remote and Concurent Process Duplication for SPMD Based Parallel Processing on COWs**. In Alfons Hoekstra Peter Sloot, Marian Bubak and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1593 of *Lecture Notes In Computer Science.* Springer, 1999.

[66] Nayeem Islam. **Dynamic Partitioning in Dierent Distributed-Memory Environments.** Internet. *http://citeseer.nj.nec.com/86642.html.*

[67] Thomas K. Jewell. *Computer Applications for Engineers.* John Wiley & Sons, Inc., 1991.

[68] Ersin Cem Kaletas, A.W. van Halderen, Frank van der Linden, and Hamideh Afsarmanesh. **Evaluation of RCube-Based Switch Using a Real World Application**. In Roy Williams Marian Bubak, Hamideh Afsarmanesh and Bob Hertzberger, editors, *High Performance Computing and Networking*, volume 1823 of *Lecture Notes In Computer Science.* Springer, 2000.

[69] M. Kandemir, A. Choudhary, and J. Ramanujam. **Restructuring I/O-Intensive Computations for Locality**. In Alfons Hoekstra Peter Sloot, Marian Bubak and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1593 of *Lecture Notes In Computer Science.* Springer, 1999.

[70] Richard M. Karp. **Parallel Combinatorial Computing**. In Jill P. Mesirov, editor, *High Performance Cluster Computing*, volume 1, chapter 15, pages 221–238. Capital City Press, 1991.

[71] M.A. Kartawidjaja and A.G. Hoekstra. **Memory Efficiency of Parallel Programs and Memory Bounded Speedup**. In Peter Sloot,

Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes In Computer Science*. Springer, April 1998.

[72] JunSeong Kim and David J. Lilja. **Characterization of Communication Patterns in Message-Passing Parallel Scientific Application Programs**. In Dhabaleswar K. Panda and Craig B. Stunkel, editors, *Network-Based Parallel Computing*, volume 1362 of *Lecture Notes In Computer Science*. Springer, February 1998.

[73] Kimmo Koski, Jussi Heikonen, Jari Miettinen, and Jussi Rahola. **Results of the One-year Cluster Pilot Project**. In Roy Williams Marian Bubak, Hamideh Afsarmanesh and Bob Hertzberger, editors, *High Performance Computing and Networking*, volume 1823 of *Lecture Notes In Computer Science*. Springer, 2000.

[74] Nicolai Langfeldt. **NFS Howto**. Internet, October 1999. *http://www.linux.org/docs/ldp/howto/NFS-HOWTO.html.*

[75] Pascale Launay and Jean-Louis Pazat. **A Framework for Parallel Programming in Java**. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes In Computer Science*. Springer, April 1998.

[76] Xavier Leroy. **Linux Threads**. Internet, 1997. *http://pauillac.inria.fr/ xleroy/linuxthreads/index.html.*

[77] Th. Lippert, N. Petkov, and K. Schilling. **BLAS-3 for the Quadrics Parallel Computer**. In Bob Hetzberger and Peter Sloot, editors, *High-Performance Computing and Networking, HPCN'97*, volume 1225 of *Lecture Notes In Computer Science*. Springer, 1997.

[78] Th. Lippert, K. Schilling, F. Toschi, S. Trentmann, and R. Tripiccione. **Transpose Algorithm for FFT on APE/Quandrics**. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes In Computer Science*. Springer, April 1998.

[79] Lennart Ljung and Torkel Glad. *Modeling of Dynamic Systems*. Prentice Hall, 1994.

[80] Paul A. Lynn and Wolfgang Fuerst. *Introductory Digital Processing with Computer Applications*. John Wiley and Sons LTD., November 1996.

[81] J.M. MacLaren and J.M. Bull. **Lessons Learned when Comparing Shared Memory and Message Passing Codes on Three Modern Parallel Architectures**. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes In Computer Science*. Springer, April 1998.

[82] Qusay H. Mahmoud. **The Web as a Global Computing Platform**. In Alfons Hoekstra Peter Sloot, Marian Bubak and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1593 of *Lecture Notes In Computer Science*. Springer, 1999.

[83] Ursula Maier and Greg Stellner. **Distributed Resource Management for Parallel Applications in Networks of Workstations**. In Bob Hetzberger and Peter Sloot, editors, *High-Performance Computing and Networking, HPCN'97*, volume 1225 of *Lecture Notes In Computer Science*. Springer, 1997.

[84] Ofer Maor. **NFS-Root-Client Mini-HOWTO**. Internet, February 1999. *http://www.linux.org/docs/ldp/howto/mini/NFS-Root-Client-mini-HOWTO/index.html*.

[85] Evangelos P. Markatos, Manolis G.H. Katevenis, and Penny Vatsolaki. **The Remote Enqueue Operation on Networks of Workstations**. In Dhabaleswar K. Panda and Craig B. Stunkel, editors, *Network-Based Parallel Computing*, volume 1362 of *Lecture Notes In Computer Science*. Springer, February 1998.

[86] James Martin and Kathleen Kavanagh Chapman. *Local Area Networks*. Prentice-Hall, 1989.

[87] Jeremy Martin and Alex Wilson. **A Visual BSP Programming Environment for Distributed Computing**. In Babak Falsafi and Mario Lauria, editors, *Network-Based Parallel Computing*, volume 1797 of *Lecture Notes In Computer Science*. Springer, 2000.

[88] Norman Matloff. **Analysis of a Programmed Backoff Method for Parallel Processing on Ehternets**. In Dhabaleswar K. Panda and Craig B. Stunkel, editors, *Network-Based Parallel Computing*, volume 1362 of *Lecture Notes In Computer Science*. Springer, February 1998.

[89] Motohiko Matsuda, Yoshiko Tanaka, Kazuto Kubota, and Mitsuhisa Sato. **Network Interface Active Messages for Low Over-**

head Communication on SMP PC Clusters. In Alfons Hoekstra Peter Sloot, Marian Bubak and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1593 of *Lecture Notes In Computer Science.* Springer, 1999.

[90] Ron Mayer. **Performance and accuracy benchmarking for FFT**. Internet, 1993. *http://www.geocities.com/ResearchTriangle/-8869/fftsummary.html.*

[91] Arnold Meijster and Fred Wubs. **Towards an Implementation of a Multilevel ILU Preconditioner on Shared-Memory Computers**. In Roy Williams Marian Bubak, Hamideh Afsarmanesh and Bob Hertzberger, editors, *High Performance Computing and Networking*, volume 1823 of *Lecture Notes In Computer Science.* Springer, 2000.

[92] J. Meira. **Modeling Performance of Parallel Programs**. Internet, June 1995. *http://citeseer.nj.nec.com/meira95modeling.html.*

[93] C. Mendes. *Performance Scalability Prediction on Multicomputers.* Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1997.

[94] David Mentré. **Linux SMP-Howto**. Internet, January 2000. *http://www.linux.org/docs/ldp/howto/SMP-HOWTO.html.*

[95] Chan Wai Ming, Samuel Chanson, and Mounir Hamdi. **The Design of a Parallel Programming System for a Network of Workstations: An Object-Oriented Approach**. In Dhabaleswar K. Panda and Craig B. Stunkel, editors, *Network-Based Parallel Computing*, volume 1362 of *Lecture Notes In Computer Science.* Springer, February 1998.

[96] Jagdish J. Modi. *Parallel Algorithms and Matrix Computation.* Oxford University Press, 1988.

[97] J. Mohan. *Performance of Parallel Programs: Model and Analyses.* Ph.D. Thesis, Carnegie Mellon University, 1984.

[98] F. Munz, T. Stephan, U. Maier, T. Ludwig, A. Bode, S. Ziegler, S. Nekolla, P. Bartenstein, and M. Schwaiger. **Improved Functional Imaging through Network Based Parallel Processing**. In Dhabaleswar K. Panda and Craig B. Stunkel, editors, *Network-Based Parallel Computing*, volume 1362 of *Lecture Notes In Computer Science.* Springer, February 1998.

[99] Bhavana Nagendra and Lars Rzymianowicz. **High Speed Networks**. In Buyya Rajkumar, editor, *High Performance Cluster Computing*, volume 1, chapter 9, pages 204–245. Prentice Hall Inc, 1999.

[100] Hironori Nakajo, Hidekazu Tanaka, Yoshinori Nakanishi, Masaki Kohata, and Yukio Kaneda. **Distributed Shared-Memory for a Workstation Cluster with a High Speed Serial Interface**. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes In Computer Science*. Springer, April 1998.

[101] Shimizu Naohiko and Watanabe Takehiko. **High Peformance Parallel FFT on Distributed Memory Parallel Computers**. In J. Harmanis G. Goos and J. van Leeuwen, editors, *High Performance Computing, ISHPC'97*, Lecture Notes In Computer Science. Springer, 1997.

[102] T. D. Nguyen, R. Vaswani, and J. Zahorjan. **Maximizing Speedup Through Self-Tuning Processor Allocation**. Technical Report TR-95-09-02, 1995.

[103] Jaechun No, Jesús Carretero, and Alok Choudhary. **High Performance Parallel I/O Schemes for Irregular Applications on Clusters of Workstations**. In Alfons Hoekstra Peter Sloot, Marian Bubak and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1593 of *Lecture Notes In Computer Science*. Springer, 1999.

[104] Masato Oguchi and Masaru Kitsuregawa. **Dynamic Remote Memory Acquiring for Parallel Data Mining on PC Cluster: Preliminary Performance Results**. In Alfons Hoekstra Peter Sloot, Marian Bubak and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1593 of *Lecture Notes In Computer Science*. Springer, 1999.

[105] Masato Oguchi, Takahiko Shintani, Takayuki Tamura, and Masaru Kitsuregawa. **Characteristics of a Parallel Data Mining Application Implemented on an ATM Connected PC Cluster**. In Bob Hetzberger and Peter Sloot, editors, *High-Performance Computing and Networking, HPCN'97*, volume 1225 of *Lecture Notes In Computer Science*. Springer, 1997.

[106] Hitoshi Oi and N. Ranganathan. **Utilization of Cache Area in On-Chip Multiprocessor**. In A Fukuda C Polychronopoulos, J. Kazuki

and S Tomita, editors, *High Performance Computing, ISHPC'99*, Lecture Notes In Computer Science. Springer, 1999.

[107] H. Öksüzoğlu and A.G.M van Hees. **A Barotropic Global Ocean Model and Its Parallel Implementation on Unstructured Grids**. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes In Computer Science*. Springer, April 1998.

[108] Kunth Omang. **Performance of a Cluster of PCI Based Ultra-Sparc Workstations Interconnected with SCI**. In Dhabaleswar K. Panda and Craig B. Stunkel, editors, *Network-Based Parallel Computing*, volume 1362 of *Lecture Notes In Computer Science*. Springer, February 1998.

[109] Stan Openshaw and Ian Turton. *High-Performance Computing and the Art of Parallel Programming*. Routledge, 2000.

[110] P. Palazzari, P.D. Atanasio, and F. Ragusini. **Simulation of Pattch Array Antennas through the Implementation of Finite-Difference Time-Domain (FD-TD) Algorithm on Distributed Memory Massively Parallel Systems**. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes In Computer Science*. Springer, April 1998.

[111] David A. Patterson and John L. Hennessy. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers Inc., second edition, 1996.

[112] Miguel Paz and Victor Gulias. **Cluster Setup and its Administration**. In Buyya Rajkumar, editor, *High Performance Cluster Computing*, volume 1, chapter 2, pages 48–67. Prentice Hall Inc, 1999.

[113] A.J.H. Peddemors and L.O. Hertzberger. **A high Performance Distributed Database Systrem for Enhanced Internet Services**. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes In Computer Science*. Springer, April 1998.

[114] Michael J. Quinn. *Designing Efficient Algorithms For Parallel Computers*. McGraw-Hill Inc., 2 edition, 1987.

[115] Günther Rackl, Filippo de Stefani, Francois Héran, Anotonello Pasquarelli, and Thomas Ludwig. **Airport Simulation Using CORBA and DIS**. In Alfons Hoekstra Peter Sloot, Marian Bubak and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1593 of *Lecture Notes In Computer Science.* Springer, 1999.

[116] Simon Ramo, John R. Whinnery, and Theodore Van Duzer. *Fields and Waves in Communication Electronics.* John Wiley & Sons, 1993.

[117] Myoung An Richard Tolimeri and Chao Lu. *Mathematics of Multidimensional Fourier Transform Algorithms.* Springer, 2 edition, 1997.

[118] Beth Richardson. **Parallel Performance Analysis**. Internet, September 1998. *http://www.ncsa.uiuc.edu/SCD/HPCTraining/materials/html/parperf/sld001.htm.*

[119] Mark Russinovich. **Inside Microsoft Cluster Server**. *Windows NT Magazine*, pages 57–62, February 1998.

[120] Volker Sander, Dietmar Erwin, and Valentina Huber. **High-Performance Computer Management Based on Java**. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes In Computer Science.* Springer, April 1998.

[121] J. Santoso, G.D. van Albada, B.A.A. Naziel, and P.M.A. Sloot. **Skel-BSP: Performance Portability for Skeletal Programming**. In Roy Williams Marian Bubak, Hamideh Afsarmanesh and Bob Hertzberger, editors, *High Performance Computing and Networking*, volume 1823 of *Lecture Notes In Computer Science.* Springer, 2000.

[122] Daniel Savarese and Thomas Sterling. **Beowulf**. In Buyya Rajkumar, editor, *High Performance Cluster Computing*, volume 1, chapter 26, pages 625–645. Prentice Hall Inc, 1999.

[123] J. Schopf. **Structural prediction models for high-performance distributed applications**. Internet, 1997. *http://citeseer.nj.nec.com/schopf97structural.html.*

[124] Martin Schulz. **SISCI-Pthreads SMP-like Programming on an SCI-cluster**. In Peter Sloot, Marian Bubak, and Bob Hertzberger,

editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes In Computer Science.* Springer, April 1998.

[125] P.A. Smeulders. *A Reconfigurable Multicomputer System: Implementation and Performance.* Ph.D. Thesis, University of Western Ontario, 1992.

[126] E. Smirni and E. Rosti. **Modeling speedup of SPMD applications on the Intel Paragon**. *Lecture Notes in Computer Science*, 919:94–??, 1995.

[127] Jörg Stadler. **Industrial Applications of High Performance Computing The Experiences from HWW**. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes In Computer Science.* Springer, April 1998.

[128] William Stallings. *Data and Computer Communications.* Macmillan Publishing Company, 4 edition, 1994.

[129] Nenad Stankovic and Zhang Kang. **A Parallel Programming Environment for Networks**. In A Fukuda C Polychronopoulos, J. Kazuki and S Tomita, editors, *High Performance Computing, ISHPC'99*, Lecture Notes In Computer Science. Springer, 1999.

[130] Frank L. Stasa. *Applied Finite Element Analysis for Engineers.* CBS College Publishing, 1985.

[131] P. Strating. **Parallel Efficiency of a Bundary Integral Equation Method for Nonlinear Water Waves**. In Bob Hetzberger and Peter Sloot, editors, *High-Performance Computing and Networking, HPCN'97*, volume 1225 of *Lecture Notes In Computer Science.* Springer, 1997.

[132] Nanri Takeshi, Sato Hiroyuki, and Shimasaki Masaaki. **Cost Estimation of Coherence Protocols of Software Managed Cache on Distributed Shared Memory System**. In J. Harmanis G. Goos and J. van Leeuwen, editors, *High Performance Computing, ISHPC'97*, Lecture Notes In Computer Science. Springer, 1997.

[133] David Taniar. **A High Performance Object-Oriented Distributed Parallel Database Architecture**. In Peter Sloot, Marian Bubak, and

Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes In Computer Science*. Springer, April 1998.

[134] Damian Trybus. Analysis of a high performance workstation. M.E.Sc. Thesis, University of Western Ontario, August 1998.

[135] Damian Trybus. **Electric Field Approximation Using Mesh Techniques in a Distributed Environment**. In *Proceedings of the Electrical and Computer Engineering Graduate Research Symposium*, May 2001.

[136] Jiri Vlach and Kishore Shinghal. *Computer Methods for Circuit Analysis and Design*. Van Nostrand Reinhold Company, 1983.

[137] Heribert Vollmer. **Relations Among Parallel and Sequential Computation Models**. In Joxan Jafar and Roland H.C. Yap, editors, *Concurrency and Parallelism, Programming, Networking and Security*, volume 1179 of *Lecture Notes In Computer Science*. Springer, December 1996.

[138] Vladimir Vuksan. **DHCP mini-HOWTO**. Internet, July 2000. *http://www.linux.org/docs/ldp/howto/mini/DHCP/index.html.*

[139] Chen Wang and Yong Meng Teo. **A Framework for Exploiting Object Parallelism in Distributed Systems**. In Roy Williams Marian Bubak, Hamideh Afsarmanesh and Bob Hertzberger, editors, *High Performance Computing and Networking*, volume 1823 of *Lecture Notes In Computer Science*. Springer, 2000.

[140] Y. Yan, X. Zhang, and Y. Song. **An effective and practical performance prediction model for parallel computing on non-dedicated heterogeneous NOW**. Internet, Oct 1996. *http://citeseer.nj.nec.com/yan96effective.html.*

[141] Akiyama Yutaka, Kentaro Onizuka, Tamotsu Noguchi, and Makoto Ando. **Biological- and Chemical- Parallel Applications on a PC Cluster**. In A Fukuda C Polychronopoulos, J. Kazuki and S Tomita, editors, *High Performance Computing, ISHPC'99*, Lecture Notes In Computer Science. Springer, 1999.

[142] Andrea Zavanella. **Skel-BSP: Performance Portability for Skeletal Programming**. In Roy Williams Marian Bubak, Hamideh Afsarmanesh

and Bob Hertzberger, editors, *High Performance Computing and Networking*, volume 1823 of *Lecture Notes In Computer Science*. Springer, 2000.

[143] Sijun Zeng and Sivarama P. Dandamudi. **Centralized Architecture for Parallel Query Processing on Networks of Workstations**. In Alfons Hoekstra Peter Sloot, Marian Bubak and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1593 of *Lecture Notes In Computer Science*. Springer, 1999.

[144] X. Zhang, Z. Xu, and L. Sun. **Semi-empirical multiprocessor performance predictions**. Internet, 1995. *http://citeseer.nj.nec.com/95872.html*.

Index

Vita

NAME:	Damian Trybus
PLACE OF BIRTH:	Wroclaw, Poland
YEAR OF BIRTH:	1969
POST-SECONDARY EDUCATION AND DEGREES:	University of Western Ontario London, Ontario 1993-1997 B.E.Sc.
	University of Western Ontario London, Ontario 1997-1998 M.E.Sc.
HONOURS AND AWARDS:	OGGST scholarship recipient 1999, 2000, 2001, 2002
RELATED WORK EXPERIENCE:	Systems Designer Ernst & Young. 1997 - 2002
	Lecturer University of Western Ontario 1999 - 2002
	Research Assistant University of Western Ontario 1997 - 2003

Publications:
Performance Analysis of a Dual Processor Workstation Trybus, D., Kucerovsky, Z., Ieta, A., Doyle, T. IEEE CCECE02 Proceedings; ISBN: 0-7803-7514-9; volume 2.
Distributed Electric Field Approximation. Cluster Base Computations Trybus, D., Kucerovsky, Z., Ieta, A., Doyle, T. IEEE HPCSA02, Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications. Moncton, Quebec, Canada; volume 1.
Pressure Dependent Corona Discharge in Selected Hydrocarbons Ieta, A., Kucerovsky, Z. Greason W., Trybus D. Proceedings of the 4th International Power Systems Conference. Timisoara, Romania