INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Distributed multi-processing for high performance computing

by

Martin Algire

A Thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfilment of the requirements for the degree of

Master of Science

in the Department of Agricultural and Biosystems Engineering

McGill University, Montreal, August 2000

Abstract

Parallel computing can take many forms. From a user's perspective, it is important to consider the advantages and disadvantages of each methodology. The following project attempts to provide some perspective on the methods of parallel computing and indicate where the tradeoffs lie along the continuum. Problems that are parallelizable enable researchers to maximize the computing resources available for a problem, and thus push the limits of the problems that can be solved. Solving any particular problem in parallel will require some very important design decisions to be made. These decisions may dramatically affect portability, performance, and cost of implementing a software solution to the problem. The results gained from this work indicate that although performance improvements are indeed possible - they are heavily dependant on the application in question and may require much more programming effort and expertise to implement.

Résumé

L'exécution en parallèle représente la toute derniere tendance dans le milieu de l'informatique de haute performance. Les problèmes qui peuvent être traités en parallèle permettent aux chercheurs de maximiser les ressources informatiques, et ainsi étendre la complexité des problèmes à résoudre. Du point de vue de l'usager il est important de considérer les avantages et inconvénients de chaque solution technologique. Le projet présenté tente de faire le point sur les différentes méthodes disponibles pour le traitement en parallèle. Il tente aussi de définir les limites de chaque approche dans un contexte scientifique. La résolution d'un quelconque problème en exécution parallèle implique des décisions importantes et complexes. Ces décisions influence dramatiquement la compatibilité, la performance, et le coût de développement d'une solution logiciel. Les résultats obtenus démontrent le potentiel de l'exécution parallèle pour augmenter la vitesse de résolution de certains types de problèmes, mais indique aussi que l'application choisie peut être ou ne pas être appropriée pour l'exécution en parallèle, et peut souvent demander un effort de programmation et une expertise considérable.

Table of Contents

List of Figures

List of Tables

1. Introduction

The problems that are currently being worked on within the Department of Agricultural and Biosystems Engineering require significantly more computing resources than restricted research budgets can often provide. Parallelizing the problem is one way to maximize the computing resources that can be spent on such a problem. Parallel computing is implemented in two ways: either as a single computing machine with multiple processors working in parallel, or as multiple computing machines working in parallel over a local network connecting the machines. Computing machines with multiple processors have the advantage of very fast communication between processors, but generally suffer from a lack of scalability. Conversely, when using a number of computing machines working in parallel over a network, a gain is possible in the level of scalability, but the latency between processors increases dramatically. The emergence of high-speed networks coupled with reductions in hardware costs have created an opportunity for clusters of inexpensive personal computers (PCs) to perform parallel computing tasks with an excellent price/performance ratio. In theory, racks filled with Pentium III CPUs can provide equivalent or even more computing power than special-purpose supercomputers, at a lower monetary cost.

An example of a software application project developed within the Department of Agricultural and BioSystems Engineering, McGill University, that would benefit from increased computing resources is the EcoCyborg project. A component of the EcoCyborg project consists of an object-based model designed to represent complex features of ecosystems. The model consists of an object-based ecosystem model that is of significant breadth, including all of the major biotic and abiotic entities of such systems. The model is in many ways computationally bounded: Firstly, the number of different types of ecosystem entities that are included is much broader than that of most other ecosystem models, and key processes are represented at relatively fine granularity with respect to space and time. Secondly, it is entirely

object-based; every component within the system is represented as a distinct entity. The result is a model where each organism, or small group of organisms, is represented as a discrete individual object that exists in a well defined environment composed of cells arranged in a 2-D lattice. The model is written in American National Standards Institute (ANSI) C and is currently being executed on a Pentium-based computer running Windows 98. The model requires several weeks of computing time on a workstation class machine to get 50 yrs of data from the system (Parrot and Kok, 2000).

Unlike the general case where the computer central processing unit (CPU) speed is considered the single most important factor, processor speed in parallel computing is just one of several factors that will determine overall performance and efficiency. In order to run an application in parallel on multiple CPUs, it must be explicitly broken into concurrent parts. The distinction between "parallel" and "concurrent" needs to be made clear. The parts of a software program's source code that can be computed independently of each other are "concurrent", while "parallel" is used to refer to the concurrent software parts that are executing on separate processors at the same time. Therefore, concurrency is a characteristic of the program's source code, and parallelism is a characteristic of the program while executing on a specific instance of computing machinery. For example, software source code can be described with respect to it's concurrent parts, but in order to describe software with respect to its parallelism the computing machinery upon which the software is executing needs to be defined as well. In addition, a software application designed and written to run on a single CPU will not execute faster if executed in a multiple CPU environment, and a software application designed and written with several concurrent parts, will not execute faster if executed in a single CPU environment. Although there are some tools and compilers that can break up a program into its concurrent parts, developing concurrent parts of a software application is not a "plug and play" operation and depending on the application, can

be easy, extremely difficult, or in some cases impossible (for example, due to dependencies within an algorithm). A software application that has been designed and written such that the major paths of execution can be described concurrently will only demonstrate a performance improvement over the serial instance of the same software if the communication latency and overhead (required for the concurrent parts to communicate) do not become the bottleneck of the software in execution (it can occur that software applications written concurrently may actually cause the program to run more slowly, and thus offsetting any performance gains that might have been made in other concurrent parts of the program). A simple heuristic often used in the "parallel world" is that it is the task of the parallel application programmer to determine what concurrent parts of the program *should* run in parallel and which parts *should not.* The result of this tradeoff will determine the efficiency of the application.

2. Objectives

The overall objective of this project is to build, and test the performance of, a parallel computing machine from a cluster of networked PCs.

Within the overall objective, the following set of objectives with respect to distributed multi-processor parallel computing were identified:

- to investigate the methods of parallel computing and indicate what engineering tradeoffs affect the price/performance ratio the most.
- to build, and to describe how to build, a parallel computing machine from a network of PCs.
- to write prototype software for the parallel computing machine in order to examine the design decisions and their relative impact on the performance and efficiency.
- to test the parallel computing machine with several industry standard benchmarks.
- to generate information on using parallel computing in the form of a network of PCs to satisfy the department's need for increased computing power.

Note that this work does not include performing the port of an already existing software model or simulation from within the department to the parallel computing machine that was built.

3. Literature Review

3.1 The Need for Computing Speed

Establishing an engineering basis from which to make decisions for many environmental problems requires the ability to model the processes and technical solutions involved (i.e. contaminant transport in the subsurface, ecosystem modelling). In many cases, the tools, skills, and knowledge presently available are not adequate for effective decision making. One such inadequacy is the computing power required to effectively execute complex models in a period of time that is practical for research purposes. A model can define a system's behavior by mathematically expressing the relationships among internal and external variables in terms of expressions like difference or differential equations. Conventional approaches to modelling include the use of ordinary differential equations, partial differential equations, stochastic models, models for hierarchical and distributed systems, and so on. Engineers use such models as a formalism to represent the system they are trying to control (Gupta and Sinha, 1996). However, the development of mathematics in a certain period of time tends to reflect the state of the computational resources available at that time. Over the past three centuries, focus has been given to (1) models that are defined and well behaved in a continuum, (2) models that are linear, and (3) models entailing a small number of lumped variables. This focus does not represent the preference of complex natural systems, but rather the fact that the human brain, aided only by pencil and paper, can only handle a small number of symbolic tokens having substantial conceptual depth. Therefore, effort tends to be concentrated on problems that are likely to yield a symbolic, closed-form solution (Toffoli and Margolus, 1987). However, simulating efficient environmental clean-up of subsurface chemical spills, environmentally friendly oil recovery, safe containment of gases and fluids generated by underground nuclear tests, underground storage of nuclear waste, and accurate characterization of water-supply aquifers, to name a few, all require the

capability of simulating complex, non-linear systems (i.e. the flow of fluids through porous media) which require substantially more computing power than pencil and paper can provide.

Computer simulation is potentially ideal for working on such complex models. One abstracts a certain subset of biological reality into the model, plays out the scenario, and observes whether that subset is sufficient to predict or explain some natural phenomena. One of the challenges with the iterative approach to developing complex models in the realm of scientific computing is to harness sufficient computing power efficiently and affordably in order to generate meaningful results in a timely fashion. Many problems require simulations that would take hundreds of hours on single-processor systems. As with many other disciplines, parallel processing has become an integral part of simulating complex natural phenomena. For example, the simulations of severe thunderstorms and tornadoes were closely linked to the development of parallel processing supercomputers during the 1970s, as well as the ongoing explosion in microprocessor power over the last decade (Wicker, 1999). Another common application of parallel processing systems is in Computational Fluid Dynamics (CFD). Many of the environmental or energy-related issues faced today cannot possibly be confronted without detailed knowledge of the mechanics of fluids. Richard Feynman, the great Nobel Prize-winning physicist, called turbulence "the most important unsolved problem of classical physics." Its difficulty was wittily expressed by the British physicist Horace Lamb, who, in an address to the British Association for the Advancement of Science, reportedly said;

> "I an old man now, and when I die and go to heaven there are two matters on which I hope for enlightenment. One is quantum electrodynamics, and the other is the turbulent motion of fluids. And about the former I am rather optimistic".

The application of powerful computers to simulate and study turbulent fluid flows is a large part of the burgeoning field of CFD (Moin and Kim, 1997).

Simulations of such complex natural phenomena are commonly performed with mathematical models, which are generally solved on a computer using numerical approximations. As an example, many of the world's energy and environmental concerns can be addressed by modelling the flow of fluids and transport of dissolved materials beneath the earth's surface (e.g., oil and gas reservoirs, contaminated aquifers). The performance of these tasks requires the computational tools to quantitatively model flow and transport through porous/fractured media. At the Los Alamos National Laboratory (LANL), researchers have developed computer software that simulate the flow of air, water, and heat, as well as the transport of contaminants in both saturated and partially saturated porous and fractured media. In order to take advantage of these advanced software simulations they have over the past decade, and are continually developing new techniques to improve computational efficiency and take advantage of evolving computer architectures. Research on parallel computing implemented as clusters of PCs is performed at the Advanced Computing Laboratory at LANL. The first parallel machine built from a cluster of networked PCs started with the Peak cluster with 8 dual Pentium II (PII) processor PCs. Work advanced with the Blue Penguin cluster consisting of 64 dual processor PII PCs and research continues with the Rockhopper cluster with 128 dual PIII 500-MHz processors (LANL, 2000).

3.2 Addressing the Need

The study of high performance computing requires revisiting computer architecture. In order to optimize the performance of computer systems, it is necessary to understand what aspects of computer architecture have a direct impact on the system's performance.

3.2.1 Moore's Law, CPU and System Peripherals

Since the introduction of the microprocessor in the early 1970s, the semiconductor industry has approximately doubled the computer power each year for the last 20 years. This exponential growth in microprocessor power was predicted by Intel's Gordon Moore who in 1965 stated that the number of components on an integrated circuit would double every year for the next 10 years. A modified version of this prediction became known as Moore's Law which states that circuit complexity, defined by the number of transistors on a die, would double every 18 months (Intel, 2000). PC capabilities have indeed increased at a phenomenal rate consistent with Morre's Law. In 1978 the Intel 8086, which was featured in IBM's first PC, ran at 4.77MHz, had a bus width of 8 bits and was composed of 29,000 transistors. In 2000 the Intel Pentium 3 Xeon runs at 933 MHz, has a 64 bit bus and is composed of over 7.5 million transistors (Intel, 2000). Some of the increased performance associated with Moore's Law comes from increased clock rates. More significantly, microprocessor architectures are appropriating and innovating on techniques formally unique to supercomputers and large mainframes. They can execute four or more instructions at a time and are being combined to create very powerful multi-processing machines. Sometimes this performance increase is available just by porting the software code to a new machine. Other times the computer architecture imposes such severe constraints on the programmer that the porting effort becomes a rewriting effort.

However, in order to take real advantage of these capabilities, a significant amount of development of system peripherals is required. Efficient operation of a PC system requires that there be a balance between the CPU, main memory and system input and output capabilities (I/O). Amdahl's law states that the needs of these three components grow in proportion to each other (Dowd and Severance, 1998). Unfortunately, while processing capabilities have increased dramatically, main memory capabilities and I/O architectures have lagged behind. Even if the computational aspects of a processor were to become infinitely fast, the data and instructions still need to be loaded and stored from and to memory. Each new generation of processor runs at a higher frequencies in order to increase the number of available clock cycles but the rate that processors can now run at has quickly outstripped the ability of main memory to keep pace. Latency management is becoming a critical issue with respect to memory management. Because access time for memory devices has not improved very much relative to improvements in size (and there are no expectations of dramatic improvements at least in the near term) the gap between processor speed and memory speed is expected to worsen in the future and could be a major limitation to further increased PC performance (Elias, 1995).

As computers are getting faster, the size of problems they operate on tend to increase, and many of the interesting problems in high performance computing use a large amount of memory. In order to solve such complex computationally intensive problems at high speeds, a memory system that is large, yet at the same time fast, is required. This is very difficult to realize in practice (Dowd and Severance, 1998).

3.2.2 Multiple Processors

One way to increase the performance of computer systems independently of Moore's Law is to increase the number of processors in the system. Most systems to date are single-processor systems. However there is a trend for more powerful machines to have more than one processor in close communication, sharing the system bus, the clock, and sometimes memory and peripheral devices. These systems are referred to as multi-processor systems (Dowd and Severance, 1998). There are several advantages to building such systems. One advantage is increased throughput. By increasing the number of processors, more work can be done in less time. It is important to note that the speed-up ratio with n processors is not n however, but is rather less than n. When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping everything working together correctly, therefore Amdahl's law applies in multi-processor architecture as well.

Multi-processor computer system architectures can be divided into 3 types: symmetric multi-processor (also known as a Shared-Memory Multi-Processor (SMP) or Uniform Memory Access (UMA)); scalable multi-processor (or Scalable Parallel Processor (SPP) or Non-Uniform Memory Access (NUMA)); and distributed multiprocessor. It is important to consider the advantages and disadvantages of each when evaluating the requirements for a new system. The following section provides a brief overview of the three architectures.

Shared-Memory Multi-Processor (SMP)

Shared-memory multi-processor systems are typically composed of high performance processors designed to be easily inserted into a multiple-processor system with 2 to 64 CPUs. However, programming multiple processors to solve a single problem can add its own set of additional challenges to the programmer. The programmer of SMP systems must be aware of how the multiple processors will work together, and how the tasks can be most efficiently distributed among the

processors. Typically a workstation will have from 1 to 4 processors and a server system will have 4 to 64 processors (Dowd and Severance, 1998). Shared-memory multi-processor systems gain a significant advantage over other multi-processor systems because all processors share the same view of memory (UMA). This means that the memory is equally accessible to all processors with the same performance and the system is less complex for the developer to understand. The cost of a shared-memory multi-processor system can range from $4000 to $30 million. Some example systems include multiple-processor Intel systems from a wide range of vendors, SGI Power Challenge Series, HP/Convex C-Series, Dec AlphaServers, Cray vector/parallel processors, and Sun Enterprise systems. Among these systems, as the price increases, the number of CPU increases, and the memory performance increases.

Figure 3.1 shows the main system components of a SMP system and how the CPU and memory are conceptually organized.

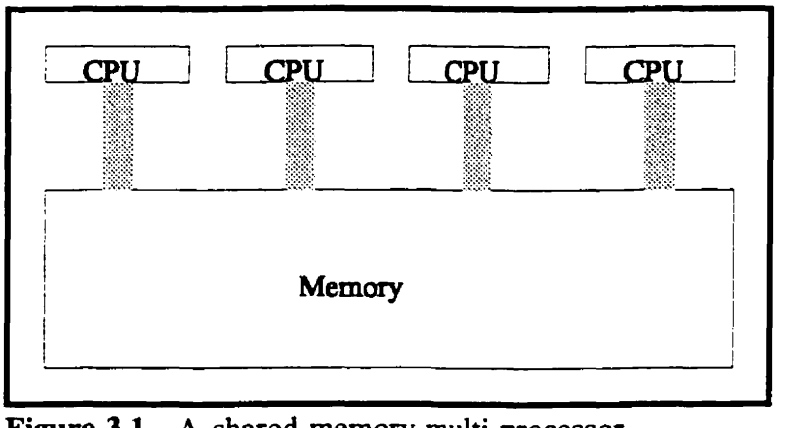

Figure 3.1 - A shared-memory multi-processor

Scalable Parallel Processor

A scalable parallel processing system is one that supports significantly more processors than a SMP based system. Another name for these systems is "Non-Uniform Memory Access" (NUMA) systems. To go beyond the hardware used to scale shared-memory multi-processor over 64 processors, typically the uniform memory access becomes too complex, so the uniform memory access is sacrificed and communication between the processors becomes the new design constraint, or bottleneck. In these machines the interconnect technology becomes the fundamental component (Dowd and Severance, 1998). The individual CPU components are the same ones as in SMP, but now it is the speed at which the CPUs can exchange information with each other that best describes the system. The resulting architecture is referred to as a "scalable parallel multi-processor".

Figure 3.2 shows the main system components of a scalable parallel processing

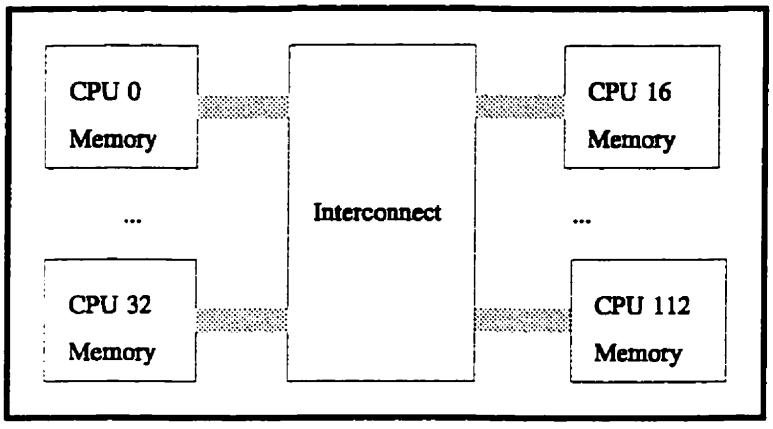

Figure 3.2 - Scalable Parallel Processing and Interconnect

system and how the CPU and memory are conceptually organized.

Distributed Multi-processor

A popular design in computer systems is to distribute computation among several processors connected via high performance networking interconnect. This approach is popular because it doesn't cost a lot to enter the high performance parallel computing market. An example of distributed multi-processor is a parallel computing machine built from a network of PCs. Creating parallel computers out of a collection of workstations is not a new idea: researchers have attempted to capitalize on the cost differential between workstation prices and those of supercomputers for 2 decades.

Throughout the 1980s and into the 1990s, there were a number of efforts to make parallel computing more practical. For example, through the use of programming libraries such as LINDA, developed by David Gelernter (Spector, 2000), programmers were able to develop machine-independent methods for breaking down problems into message-based systems that could be run over a large collection of idle machines. Experimental operating systems, such as Amoeba by Andrew Tannenbaum (Spector, 2000), would extend parallel constructs into the operating system proper. Although they found niches within the academic community, most of these workstation-based systems of the 1980s did not fide wide spread usage (Spector, 2000).

Unlike the other two multiprocessor architectures, distributed multi-processor systems are the most loosely coupled of the multiprocessor systems and do not share memory or a clock. Each processor has it's own local memory (Radajewski and Eadline, 1998). What differentiates a distributed multi-processor parallel computing machine from a local area network (LAN) is that a distributed multi-processor system is running a form of software interconnect on each machine in order for the PCs to be dedicated to some kind of computationally intensive task. Generally it is harder programmatically to get peak performance from these systems

because of slower communications, but because the hardware cost is so low, the extra programming effort is often warranted. Also, distributed multi-processor is a reasonable approach to work around the lagging performance in system peripheral (memory, I/O, storage) speeds which have been unable to keep up with CPU performance and have become the bottleneck. Keeping in mind that most applications require "out of cache memory access" and hard disk access, doing things in a distributed multi-processor system is one way to get around some of these limitations. Network-based parallelism can also provide greater fault tolerance. If a symmetric or scalable multi-processor system fails, the likelihood is high that the entire system will go down and will take all running processes with it. In a clustered environment, a node on the cluster failing typically means a loss of overall cluster performance, but applications can continue to run and there is much less chance of loss of data (Mehat, 2000).

An extreme example of the power of distributed parallel-processing is the SETI@home project, an ongoing search for extraterrestrials. The idea behind SETI@home is to take advantage of the unused processing cycles of personal computers. Interested computer owners download free software from SETI@home which when the owner's PC is idle will download a 300 kilobyte chunk of SERENDIP data for analysis. The results of this analysis are ultimately sent back to the SERENDIP team, combined with the crunched data from the many thousands of other SETI@home participants, and used to help in the search for extraterrestrial signals. As of February 2000, SETI@home has grown to encompass 1.6 million participants in 224 countries. The amount of computing time contributed since May, 1999 is equal to 165,000 years, averaging 10 Teraflops (about 10 times more than the largest supercomputer on the planet). It is the largest computation ever done, and has attracted the participation of thousands of groups such as schools and private companies (SETI, 2000).

Another example of the power of distributed multi-processor parallel computing systems was a project designed to win the RSA Data Security Inc. DES encryption key cracking challenge. In February of 1998 a global team of programmers called distributed.net broke a 56-bit Data Encryption Standard key in 39 days. The distributed.net team employed the same type of brute force that was used to break DES in the first challenge, utilizing the idle time of the computers of 22,000 participants throughout the world and linking more than 50,000 CPUs to plow through the 72 quadrillion possible key combinations (ZDNET, 1998).

Software Architectures for High Performance Parallel Computing

As indicated by the examples outlined above, specific software solutions can be created in order to solve a problem on a distributed multi-processor system implemented as a network of PCs. However, for the case of building a cluster for use in scientific computing, it is of more interest to consider a more general solution that can be used to solve a wider variety of problems. The most popular general software architectures for creating clusters for use in scientific computing are Beowulf, Mosix, and Parallel Compilers.

The researchers participating in the Earth and Space Sciences Project at the Goddard Space Flight Center have an almost unbounded need for computing power. To address the issue, Donald Becker and Thomas Sterling launched the Beowulf Project in 1994. Beowulf is a distributed parallel-processing computer system consisting of high-performance PCs built from off-the-shelf components. The PCs are connected via Ethernet, and run Linux as an operating platform. Ultimately, the goal of the Beowulf approach was to achieve supercomputer (gigaflop) performance at a fraction of the price (along with other members of his team, Becker was the recipient of the IEEE Computer Society 1997 Gordon Bell Prize for Price/Performance). The Beowulf project was the first of its kind and as such represents the ground-breaking work in clustering PCs to obtain supercomputer

levels of performance. Since it's inception, Beowulf-style clusters have become their own well-defined genre of high-performance computing systems. They have also made it into the upper levels of the "top 500 Supercomputers", which is published semi-annually and lists the achievements in high performance computing. As of October 1999, several of the systems within the top 200 supercomputers on the list were made up of Beowulf style cluster systems (Spector, 2000).

Beowulf as a system uses Parallel Virtual Machine (PVM) or Message Passing Interface (MPI) as the software interconnect (Radajewski and Eadline, 1998). PVM and MPI are message-passing environments. They typically consist of a library of function calls for C or Fortran that provide the programmer with a way to split an application for parallel execution. Data is divided and passed out to the processes as messages. The receiving processes unpack them, do some work, and send the results back or pass them along to other processes in the parallel computer. Typically a server node is designated to control the whole cluster, and acts as the master to the client nodes. Large Beowulf machines might have more than one server node, and possibly other nodes dedicated to particular tasks, for example consoles or monitoring stations. There are many software packages such as kernel modifications, PVM and MPI libraries, and configuration tools which make the Beowulf architecture faster, easier to configure, and much more usable.

Mosix is a software package that was specifically designed to enhance the Linux kernel with cluster computing capabilities. Mosix uses a number of adaptive resource management algorithms in order to determine when to migrate processes to other nodes on the cluster. Mosix will monitor several performance characteristics of each node (i.e free memory, CPU usage) as well as several performance attributes of the cluster as a whole (i.e. uneven load distribution among the nodes on the cluster) and will move processes from one node to another in an effort to maximize the efficient use of the cluster's resources. Mosix operates

silently and its operations are transparent to the applications. The operator does not need to know about where the processes are running. Shortly after the creation of a new process, Mosix attempts to assign the process to the best available node at that time. Mosix then continues to monitor the new process, as well as all the other processes, and will move any process among the nodes to maximize the overall performance of the cluster. All this is done without changing the Linux interface, so an operator can continue to see (and control) all processes as if they were running on a single node. The algorithms of Mosix are decentralized - each node is both a master for processes that were created locally, and a server for (remote) processes, that migrated from other nodes. This means that nodes can be added or removed from the cluster at any time, with minimal disturbances to the running processes. Another useful property of Mosix is its self tuning and monitoring algorithms, which detect the speed of the nodes, and monitor the load, available memory, as well as inter-process communication (IPC) and I/O rates of each process. This information is used to make near optimal decisions where to place processes.

Although Mosix has been used as a production system for many years on differnet versions of Unix, it has been recently ported to Linux on Intel platforms (Barak, 2000).

Initially compilers were tools that allowed programmers to write something more readable than assembly language. Today they border on artificial intelligence as they take high-level language source code and translate it into highly optimized machine language across a wide variety of single- and multiple- processor architectures. In the area of high performance computing, the compiler at times has a greater impact on the performance of the program than either the processor or memory architecture (Dowd and Severance, 1998).

There are tools available to the developer to create software that runs in parallel without having to program message passing into the application proper, like in the case of Beowulf/Message passing. Parallel compilers do exist for Fortran, C, and C++ and often provide integrated support for multiple message passing models (i.e. HPF, OpenMP, and MPI). Graphical debugging and parallel performance profiling tools are often bundled with such compilers. One advantage of using such a tool is that it is possible to build high performance applications for single, dual, or quad-processor workstations which can then run unchanged on workstation clusters, shared-memory servers, or high-end distributed-memory or NUMA supercomputers.

However, parallel compilers that help in determining the concurrent parts of a program are most common to the FORTRAN domain. Historically, FORTRAN has been used for the majority of number crunching applications and it's syntax is easier for parallelizing tools to analyze. Parallel compilers will often allow the user to provide some information about the concurrent nature of their application, but the compiler will then make all the decisions about how to execute the concurrency in parallel. Giving up this amount of control to the compiler in parallel software architectures rarely results in optimal implementations (Radajewski and Eadline, 1998). Parallel compilers also suffer because they are not appropriate for non-shared memory computers because there is no standard programming interface (e.g. compiler directives), and there are often severe restrictions placed on the architecture of the parallel solution.

3.3 Measuring Performance

The phrase "What is not measurable make measurable" attributable to Galileo Galilei is part of the folklore of measurement scientists (Fenton and Pfleeger, 1996). It suggests that one of the aims of science is to find ways to measure attributes of things in which we are interested. Strictly speaking there are two kinds of quantification: measurement and calculation. Measurement is a direct

quantification, as in measuring the height of a tree or the weight of a shipment of bricks. Some attribute of an entity is assigned a descriptor that allows it to be compared with others. Calculation is indirect, where measurements are made and combined into a quantified item that reflects some characteristic whose value we are trying to understand (Fenton and Pfleeger, 1996). Entities and their attributes are used interchangeably in everyday speech (the room is cold), but it is incorrect and unsuitable for scientific endeavours (the temperature of the room is cold). It is the attribute that can be measured.

"Benchmarking" with respect to computer systems refers to the measurement of computer system performance attributes. Benchmarks in the domain of high performance computing are typically used in order to market or sell a vendor specific system. The premise is that when a buyer will be spending somewhere between $10,000 and $30 million, they need to ensure that they get what they pay for. Benchmarks range from the very simple (30 lines of code) to very complex (100,000+ lines of code). Often a benchmark amalgamates the system performance in a single non-dimensional number. For example, a $200,000 system might be rated at 23.5 and an $85,000 system at 15.2 (Dowd and Severance, 1998). Comparisons based on benchmarks like these are not particularly useful in making informed purchasing decisions. Ultimately the only thing that matters after you buy the system is how well the system runs your applications under operating conditions that you expect in production (Collins, 1998).

Benchmarking computer systems allows comparisons to be made between the performance metrics generated by the benchmark tests. Benchmarking is a boring, repetitive task that requires attention to details. Although benchmarking is entirely objective and deals with facts and figures, often the results are not what one would expect and are subject to interpretation (Balsa, 1997).

3.3.1 Benchmark Implementations

There are three types of popular benchmark implementations: kernel, synthetic and application. Kernel benchmarks are founded upon the heuristic that in the majority of cases 90% of the time is spent in 10% of the code. A kernel benchmark is that 10% of the code extracted for measurement purposes. Livermore Loops and Linpack are examples of two popular kernel benchmarks. The fundamental shortcoming with kernel benchmarks is that they are usually small, fit in the processor's cache, are prone to optimization tricks by compilers, and measure only CPU performance (Dowd and Severance, 1998).

Synthetic benchmarks are designed to gather performance metrics on a specific subsystem (hardware, software, or combination of hardware and software). Dhrystone and Whetstone are examples of synthetic benchmarks.

From a user's perspective the best benchmark is the user's own application program. Of course, there are thousands of applications and many of them are proprietary. A benchmark suite with a large number of example user applications is also impractical because of difficulties in porting, evaluation, and long runtime (Dixit, 1992).

3.3.2 Industry Benchmarks

Industry benchmarks are typically developed by an independent organization that creates, maintains, distributes, and endorses a standardized set of application-oriented programs to be used as benchmarks. Industry benchmarks are viewed as one of the best sources for reliable computer performance information. The Systems Performance Evaluation Cooperative (SPEC) is the best known industry benchmark. SPEC is a nonprofit consortium of 22 major computer vendors whose common goal is to provide the industry with a realistic yardstick to measure performance of advanced computer systems (Dixit, 1992). Specmarks are published

measurements of how well computers performed on the benchmark suite.

Many vendors characterize system performance in millions of instructions per second (MIPS) and millions of floating-point operations per second (MFLOPS). These measurements are of the worst possible type to use to compare systems. Using MIPS and/or MFLOPS to determine relative performance of different computers is fundamentally flawed. A million instructions on one processor architecture does not accomplish the same "work" as a million instructions on another because instructions are not equal on different processor architectures. Since CISC machine instructions usually accomplish a lot more than those of RISC machines, comparing the instructions of a CISC machine and a RISC machine is similar to comparing Latin and Greek (Dixit, 1992). For this reason, using MIPS/MFLOPS to compare computer systems is analogous to determining the winner of a foot race by counting who used fewer steps.

Benchmarks are also devised for marketing purposes. Intel's ICOMP benchmark (Version 1.0) is an example of a benchmark program developed by a microprocessor manufacturer for measuring microprocessor performance. ICOMP 1.0 was a proprietary benchmark, unique to Intel. Intel did not publish the formula for ICOMP 1.0, and it didn't let the program be licensed by other objective third parties. Since the scientific method depends on a process of justification and on scientific results to be replicable (Dibona, Ockman, Stone, 1999), the results of the ICOMP 1.0 benchmark could not be verified and the benchmark architecture and code itself could not be reviewed by third parties for correctness. Intel has since abandoned ICOMP 1.0, replacing it with ICOMP 2.0, whose formula is published on the web, making it an open standard that can be independently verified.

The Numerical Aerospace Simulation (NAS) Parallel Benchmarks (NPB)

The Numerical Aerospace Simulation (NAS) Parallel Benchmarks (NPB) are the

standard for measuring the performance of distributed parallel-processing computer systems. The NAS Systems Division is part of the Information Sciences and Technology Directorate at NASA Ames Research Center, Moffett Field, California. The NAS Parallel Benchmarks (NPB) are a set of 8 programs designed to help evaluate the performance of parallel supercomputers. The benchmarks are derived from computational fluid dynamics (CFD) applications and consist of five kernels and three pseudo-applications. There are three benchmark sets that make up the NAS parallel benchmark. These sets are further described below:

NPB 1

This benchmark doesn't come with any source code, NAS publishes the problem definition describing the mathematics and computations to be performed and the implementor (typically vendors) are free to choose any language to solve the problem. The objective is to remove all bias or stunts by making them all explicitly legal, and the results represent the computer system in the best possible light. The results are verified by NAS and published in a periodic NAS report. NPB 1 implementations are generally proprietary.

NPB 2

This benchmark is a more traditional benchmark consisting of both kernel programs and application programs implemented and distributed by NAS. NPB 2 uses MPI as a message passing mechanism. The bechmarks are intended to be run with little or no tuning, and approximate the performance a typical user can expect to obtain for a portable parallel program. Because the NAS benchmark developer has already identified the parallelism, the data decomposition, and the communication pattern, it is a better test of the hardware and architecture performance than the compiler performance.

NPD 2-serial

These are single processor (serial) source-code implementations derived from the NPB 2 by removing all parallelism. The idea is to allow smaller single processor and multi-processor shared-memory systems to execute the same codes for comparison purposes. As benchmarks, they are intended to be run with little or no tuning.

3.3.3 User Benchmarks

Because the raw hardware performance depends on many components: CPUs; floating-point units; I/Os; graphics and network accelerators; peripherals; and memory systems, traditional and popular benchmarks often fail to characterize system performance of modern computer systems. However, system performance as seen by the user depends on the efficiency of the operating system, compiler, libraries, algorithms and application software (Dixit, 1992). From a user's perspective the best benchmark is the user's own application program. Therefore there are several real benefits and practical reasons to running typical user code/applications on the computer system:

- the application may exercise some feature of the compiler or hardware that wasn't revealed in the industry benchmark.
- the program may require some support in the form of libraries or language features that aren't present on the machine.
- the application might run unexpectedly slowly
- the program might expose a debilitating bug.

3.3.4 Benchmark Metric

Elapsed time is a useful and simple measure with which to run a benchmark. As a program runs it switches back and forth between two fundamentally different modes: "user mode" and "kernel mode". The normal operating state is user mode. It might be enough to run in user mode for the duration of a program's execution, except that programs generally need other operating system

services, such as I/O, and these services require the intervention of the operating system kernel. A kernel service request made by your program causes a switch from user mode to kernel mode. Time spent executing in the two modes is accounted for separately as "user time" and "system time". The "user time" describes the time spent in user mode and the "system time" is a measure of the time spent in kernel mode. Each program running in user mode on the machine is accounted for separately (Johnson and Troan, 1998). Taken together, user time and system time are called CPU time. A third measure of time is the "elapsed time" which is a measure of the actual (wall clock) time that has passed since the program was started. For programs that spend the majority of their time computing, the elapsed time should be close to the CPU time. There are reasons why elapsed time might be greater than expected, which are:

- the application is timesharing the machine with other active programs
- the application performs a lot of I/O
- the application requires more memory bandwidth than is available on the machine.
- the program was paging or swapping.

(Dowd and Severance. 1998)

4. Material and Methods

Two different software architectures were implemented on a distributed multi-processor system: Beowulf and Mosix. This section describes the hardware and how it was setup and configured, the interconnect software used to implement each solution and how it was setup and configured, and the benchmark (industry and user application) software and how it was developed and the different modes of execution.

4.1 Beowulf

The following sections cover the architecture of the Beowulf-style parallel machine that was built from a cluster of PCs. Although there are several design decisions and tradeoffs that are required in implementing a production system, the system that was actually implemented is described here in detail, and the design alternatives are presented when they best aid in understanding this implementation.

4.1.1 Hardware Architecture

There are at least 4 ways to configure the persistent storage in the Beowulf cluster: diskless clients; fully local install; NFS (Network File System) and distributed file system. The tradeoffs between the 4 different methods are between price, performance and administration efforts. Diskless clients and NFS configurations were the most appropriate for the Beowulf-style cluster that was implemented. In the diskless client disk configuration none of the client machines in the client/server architecture require a hard disk at all. This is very, very time-saving and efficient from a network administration perspective. The way this works is that the Linux operating system kernel is compiled with configuration parameters that instruct the kernel when it is booted to use the Reverse Address Resolution Protocol (RARP) to get its Internet Protocol (IP) address from the server, and once it has it's IP address it uses the Network File System (NFS) to mount it's root file system from the server, which has a hard disk. Such a configured kernel can be built on the

server and copied to a floppy. An important point to note is that the kernel needs to be compiled with the network driver of the client, not of the server. If a client machine is then booted with the floppy that contains this specially configured Linux kernel it will start the bootstrap sequence, perform the RARP transaction with the server (get it's IP address) and then load it's root partition from the server. This is particularly convenient if there is a large number of machines with the same hardware configuration that are available but which are not running Linux and it would be prohibitively difficult or expensive to install Linux on each machine. The NFS configuration is the complete opposite from the diskless client paradigm, a full install of the Redhat 6.1 operating system is performed on all client and server machines. This is expensive from an administration point of view, but it is the most basic configuration and is therefore a useful way to get things working because of it's simplicity. A local install of the Redhat 6.1 Linux distribution was done for the Beowulf-style parallel computer. NFS is configured on the server machine to export its directories that contain the benchmark software, and the message passing software. This lightened the administrative load considerably, since the components that were changing most often did not have to be copied to each machine each time they were changed.

Although Linux runs on a large range of processing hardware, the choice of CPU should realistically be made from two families; Intel x86 compatible and DEC Alpha. There is just much better support online (mailing lists, FAQS, NNTP) for these more mainstream CPU architectures. SMP multi-processor boards are frequently used in Beowulf-style clusters. By going dual CPU across the entire cluster, it is possible to save on half of the network cards, cases, power supplies, and motherboards. The cost increase is the more expensive SMP motherboard but the savings typically outweigh this increase and the performance increase is huge (fast bus-style interconnect between half of the processors rather then Ethernet).

The Beowulf-style cluster was implemented with 4 CPUs: 2 Intel Pentium III running at 450 MHz and 2 running at 500 MHz. Although the size of the cluster was stunted at only 4 nodes, it was felt that it was a good first step that would provide the implementation experience and feasibility information required by the objectives.

For the networking hardware used to connect the machines in the cluster together, the fastest Ethernet available is always best from a performance point of view. The Beowulf-style cluster was implemented with 100BaseT. The cluster was built on a private subnet, so there was no contention (Ethernet is a broadcast protocol, and packet collisons are frequent) with other machines not participating in the cluster.

As in a serial computing machine, swapping of memory to disk will constrain the performance of software. Sufficient memory is required such that the applications that are executed on the parallel computer never (or very, very seldom) swap to disk. The Beowulf-style cluster was implemented with 256 Megs of memory on each machine for a total parallel machine memory size of 1 Gigabyte.

Figure 4.1 shows the resulting hardware architecture of the cluster.

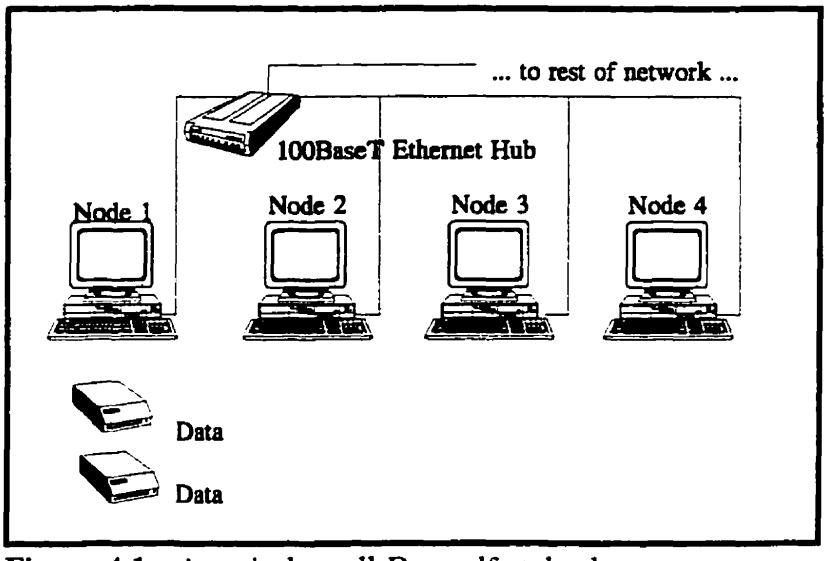

Figure 4.1 - A typical small Beowulf-style cluster

4.1.2 Software Architecture

There are two separate software packages that are required in addition to the hardware to build a Beowulf-style parallel computer; the GNU/Linux operating system and a message passing library.

Although there is no reason why non-Redhat Linux distributions would not work as the operating system for a Beowulf-style parallel computer, Redhat is a popular choice for which the most support is available within North America (Debian and SuSe are more popular in Europe).

The Beowulf-style cluster was implemented with the RedHat 6.1 Linux distribution. A full install of the contents of the installation media was performed to avoid having to install additional packages one at a time if required later.

There is some advanced Linux networking administration required to setup the Beowulf. All the nodes need to be able to bring up their Ethernet interfaces and

send/receive network packets with each of the other nodes in the cluster. Standard tools like *ping* are used to see if the interface and its IP configuration is operative, then *nslookup* and *traceroute* are used to ensure that the host names all resolve correctly. Depending on the size of the cluster, it may or may not be necessary to setup a Domain Name Server (DNS) for the cluster. Linux has a simple /etc/hosts file that can be used to specify IP/hostname mappings and is simple to administer as long as there are not too many changes to manage. Each machine in the cluster should have an identical copy of /etc/hosts. The other requirement with respect to hostname settings is that in order for the message passing software to function correctly, each machine needs to allow remote shells ("rsh") from any node to any node in the cluster. Because the cluster can be separated from the rest of the public Internet (there is no need for a physical route to any other networks) it is safe to lower the security settings to allow this. The names of the nodes that are allowed to execute and *rsh* without providing a password are described in /etc/hosts.equiv, which is a simple list of the hostnames that have this privilege. The hostname of each node in the cluster needs to be in the file and each node should have an identical copy of /etc/hosts.equiv.

The two most popular message-passing environments are "parallel virtual machine" (PVM) and "message-passing interface" (MPI). Most of the main features are available in either environment and once the concepts of message passing are mastered, it is usually not that difficult to go from one message-passing library to another.

MPI was chosen because it is available and in wide use on other Beowulf style clusters, because it is an industry standard, and because it is designed for and generally achieves high performance. PVM, while popular, undergoes major changes frequently, so that vendors rarely have optimized implementations of the latest version and codes require revision to conform. PVM was not designed for

high performance, and is more appropriate for loosely coupled, dynamic and fault-tolerant applications.

MPI was developed by a consortium of computer vendors, application developers, and computer scientists. The objective of the consortium was to specify a standard that would take the strengths of many of the existing proprietary message passing environments on a wide variety of architectures and come up with a common specification that could be implemented on a large range of architectures.

The Local Area Multicomputer (LAM) implementation of MPI grew out of the Trollius project from the Ohio Supercomputer Center. LAM/MPI provides a persistent run-time environment for MPI programs. Figure 4.2 shows how LAM daemons are launched on each machine in the cluster. The LAM daemons are mainly used for process control, an out-of-band communication channel for meta data, and a monitoring/debugging tool for user programs. Once the LAM daemons have been launched, MPI programs can be launched across the resulting parallel machine.

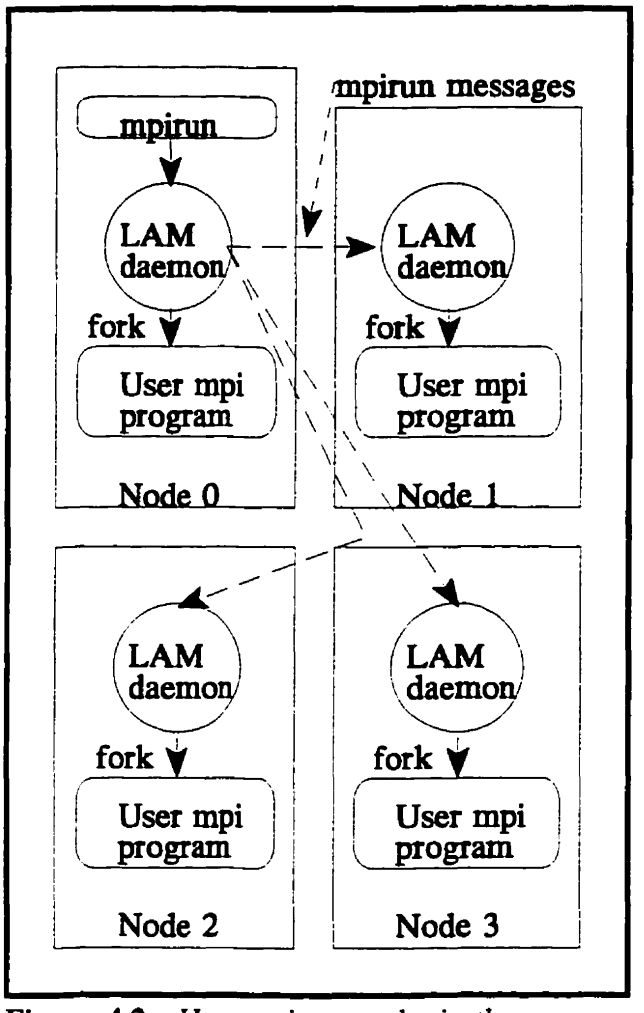

Figure 4.2 - How mpirun works in the LAM/MPI environment. Mpirun sends execution messages to the local LAM daemon, who, in turn, distributes them to the remote LAM daemons. Each daemon then starts up the user program.

One of the main functions of the MPI layer is to create and maintain communication queues. All send and receive communications within LAM/MPI are collectively known as requests. Unifying all types of communication under a single nomenclature allows the use of a uniform management system. For example, MPI SEND generates a request that contains information such as the buffer, count, data

type, tag, destination rank, and communicator of the message to be sent. If the queue is empty, the request is passed directly to the Request Progression Interface (RPI) layer to be processed immediately (this is known as the short-circuit optimization). If the queue is not empty, the new request is marked as blocking, and placed on the queue. The RPI is then invoked to progress the queue. Since the request was marked as blocking, the RPI will not return until the message has been fully sent. Figure 4.3 shows the conceptual organization of the MPI, RPI and Trollius layers (Trollius layer contains a set of infrastructure and communication libraries) which make up the LAM/MPI architecture.

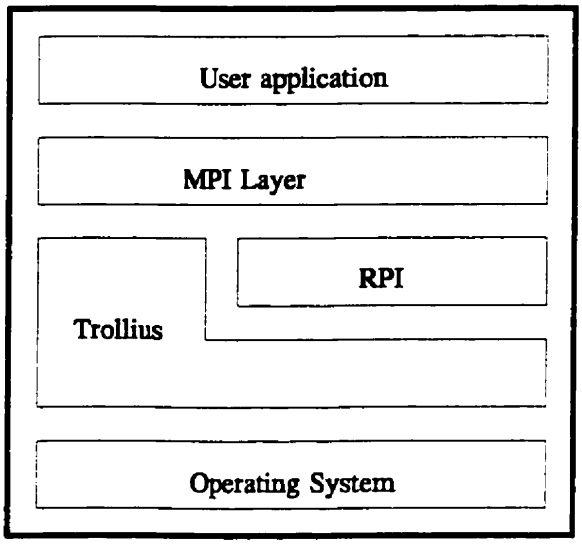

Figure 4.3 - Main Components of the LAM/MPI communication library

Requests are processed through the LAM request progression interface (RPI). The RPI is responsible for all aspects of communication with other MPI ranks. It progresses the communication requests that were formed and queued in the MPI layer. That is, the RPI is responsible for actually moving data from one rank to another. Once the RPI finishes a request, it marks the request as completed (the MPI layer will dequeue it).

The Beowulf-style cluster was implemented with the LAM/MPI 6.3.2. LAM/MPI is made freely available under the General Public License (GPL). Every machine on the cluster needs to have the same version of LAM/MPI installed. In the implementation, the LAM/MPI installation directory (/usr/local/lam-6.3.2) was NFS mounted by the clients from the server so that LAM had only to be installed once. Once installed there is a single configuration file that needs to be edited which contains a list of the hosts that are nodes on the cluster (/usr/local/lam-6.3.2.boot/lamhosts). LAM will use this information to launch the LAM daemon on each host at LAM boot time.

Software applications are compiled with special LAM wrappers for the respective compilers. Note that LAM does not replace or upgrade the version of the compiler, it simply wraps the call to the compiler so that it can adjust the runtime environment to support the compilation of the code that calls into the LAM libraries. Once the application is written and the calls into the message passing library are in place, the code can be compiled with the LAM wrapper compiler calls, hcc, hcp and hf77 respectively (wraps the gcc C compiler, the g++ C++ compiler, and the g77 FORTRAN77 compiler).

After the LAM compiled binaries are built, they need to be present in the same directory location on each machine in the cluster. This is straightforward if the binary is created in a NFS exported directory so that all nodes can see the binary without any additional manual copying of files. If NFS is not supported then the binary needs to be copied to each machine in the same directory, in order for "mpirun" (a program that is used to start the binary) to find it at runtime.

Before running the parallel application on the cluster, the LAM daemons need to be started. The LAM daemons are used to launch the parallel application on each of the nodes in the cluster. The LAM daemons are started by using the "lamboot"

command. Once all of the LAM daemons have been successfully started using lamboot, the user can execute their parallel program using the LAM program "mpirun". "mpirun" instructs the LAM daemon on the local machine to launch the LAM application and to notify all the other LAM daemons running to do the same. Once all of the applications have been started by the LAM daemon, they are free to start passing messages between each other and work on the computing task.

The software for testing the performance of the Beowulf-style cluster consists of the NPB-2 and NPB-2 serial as well as two user programs that were written in order to gain first hand experience in writing code in a parallel, message passing environment. All software is available in parallel and equivalent serial versions in order to make comparisons against the single serial computer.

The Benchmarks were run on 1, 2, 3 and 4 machine cluster configurations, wherever possible (several of the Benchmarks programs were constrained by specific CPU configuration requirements). This was intended to aid in determining an objective relationship between the specific properties of each software benchmark application and the parallel machine. An interpretation of the results and the qualitative aspects of the design and implementation of the two user benchmark software applications are discussed in section 5 (Results and Discussion).

In order to examine the design decisions required and the impact they have on the performance and efficiency of executing the software, two application programs for the parallel computing machine were developed, User Benchmark 1 (UB1) and User Benchmark 2 (UB2).

UB1 is the depth-first search algorithm, a standard search algorithm from within the domain of computer science. The strategy for choosing what to implement was

simply that applications should be representative of the kind of applications that might be implemented by the end-users of the cluster. Depth-first searching is one of the three blind-search control strategies (the other two are breadth-first search and uniform-cost search). Depth-first expands the search space downwards first. The algorithm was implemented with numbered blocks representing the solution space, and the control algorithm had to search it's way through the solution space from an initial condition to determine the path to the desired numbered block configuration. Depth-first searching through a large solution space is a computationally bound problem. The quantitative metric collected was CPU time (user and system times taken together).

Figure 4.4 describes in words the depth-first searching algorithm.

Put the start node on list OPEN (unexpanded nodes).
If start node is a goal node, show solution.

While searching for a solution
    If list OPEN is empty, no solution exists, quit.

    Remove first node n from list OPEN and place it
    on list CLOSED (expanded nodes)

    If depth of node n > maximum depth, quit.

    Expand node n.

    Place all successors of node n at beginning of list OPEN.

    If any successor is a goal node, show solution.
End while

Figure 4.4 - The depth-first control strategy algorithm

An example of a search tree that results from the depth-first search control strategy algorithm is described in Figure 4.5. The start node is N0 and the goal node is N11.

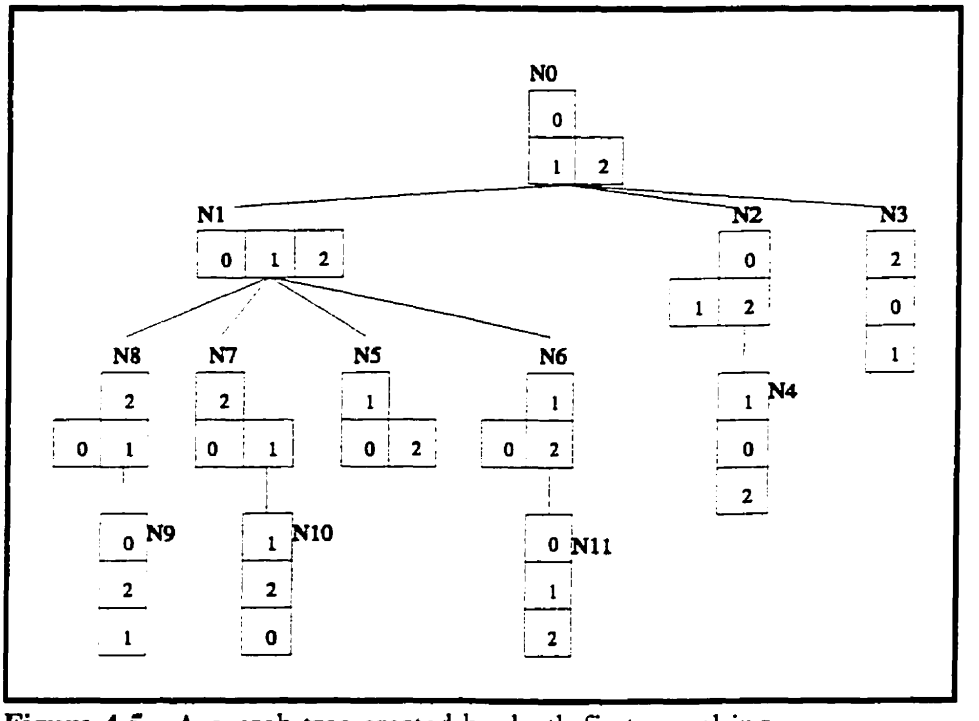

Figure 4.5 - A search tree created by depth-first searching

The second of the two user application software benchmarks is a floating point dense matrix multiplication application (UB2). The program was tested using matrices of varying sizes (from 10x10 to 25x25) and the contents were generated randomly. The storing, adding, and multiplying of large quantities of floating point numbers requires very large amounts of computing power and the management of large quantities of floating point numbers stored in matrices is a common implementation among potential end-users of the Beowulf style cluster.

The parallel version of the application was designed using a modified version of the

Single Instruction Multiple Data (SIMD) parallel decomposition. The traditional SIMD design pattern involves a large number of processors all performing the same operation at the same time step, yet on different data. Typically there exists a front end process (like a master in a master/slave parallel application decomposition) that broadcasts the instructions to the processors. UB2 uses a single front-end and each node has it own copy of the data. The single front end broadcasts to the nodes which row/columns to operate on and then collects the result into a product matrix. UB2 does not coordinate all of the calculations of the nodes on each time-step. The multiplication was written entirely asynchronously. A synchronous approach on the hardware architecture that was used for the tests would have the disadvantage of being constrained by the slowest processing unit on the cluster. It is often better in a heterogenous environment to operate asynchronously, if possible. The quantitative metric collected for the UB2 application benchmark was also CPU time (user and system times taken together).

A set of implementations of the NAS Parallel Benchmarks based on FORTRAN77 and the MPI message passing standard was executed on the parallel computer. These implementations, which are intended to be run with little or no tuning, approximate the performance a typical user can expect from a portable parallel program on a distributed memory computer. The implementations and the descriptions of the benchmarks were provided by the Numerical Aerospace Simulation (NAS, 2000). Although the software provided by NAS was originally intended for UNIX, it ported to Linux with little effort.

While the benchmarks are implemented with MPI, they are not intended to test only MPI. They are holistic benchmarks, designed to measure the overall performance of a complex system of which MPI is one part.

Because these benchmarks measure wall clock time and are statically load balanced

and tightly synchronized they have to be run on completely dedicated systems, and with as few system processes as possible. Failure to do so may cause timing results to be in error.

Kernel Benchmark: Multigrid (MG)

The Multigrid Benchmark is based on four critical subroutines -- the smoother, psinv, the residual calculation, resid, the residual projection, rprj3 and the trilinear interpolation of the correction. This code requires a power-of-two number of processors. The partitioning of the grid onto processors occurs such that the grid is successively halved, starting with the z dimension, then the y dimension and then the x dimension, and repeating until all power-of-two processors are assigned.

Simulated CFD Application Benchmarks: Lower Upper (LU), Scalar Pentadiagonal (SP), Block Tridiagonal (BT)

The LU benchmark requires a power-of-two number of processors. A 2-D partitioning of the grid onto processors occurs by halving the grid repeatedly in the first two dimensions, alternately x and then y, until all power-of-two processors are assigned, resulting in vertical pencil-like grid partitions on the individual processors. Communication of partition boundary data occurs after completion of computation on all diagonals that contact an adjacent partition. It results in a relatively large number of small communications of five 8-bit words each. It is very sensitive to the small-message communication performance of an MPI implementation. It is the only benchmark in the NPB 2.0 suite that sends large numbers of very small (40 byte) messages.

The SP and BT algorithms have similar structures: each solves three sets of uncoupled systems of equations, first in the x, then in the y, and finally in the z direction. These systems are scalar pentadiagonal in the SP code, and block tridiagonal with 5x5 blocks in the BT code.

The NPB 2.0 implementations of SP and BT solve these systems using a multi-partition scheme because it provides good load balance and uses coarse grained communication. Additionally, the information from a cell is not sent to the next processor until all sections of linear equation systems handled in this cell have been solved. Therefore the granularity of communications is kept large and fewer messages are sent.

Both the SP and BT codes require a square number of processors. These codes have been written so that if a given parallel platform only permits a power-of-two number of processors to be assigned to a job, then unneeded processors are deemed inactive and are ignored during computation, but are counted when determining Mflop/s rates.

Kernel Benchmark: Conjugate Gradient (CG)

In this benchmark, a conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication and employs sparse matrix vector multiplication.

Kernel Benchmark: Embarrassingly Parallel (EP)

EP generates pairs of Gaussian random deviates according to a specific scheme and tabulates the number of pairs in successive square annuli. This problem is typical of many Monte Carlo simulation applications.

EP is the "embarrassingly parallel" benchmark, because the computation can be done independently on each processor.

Kernel Benchmark: Integer Sort (IS)

This benchmark tests a sorting operation that is important in "particle method" codes. This type of application is similar to "particle in cell" applications of physics, wherein particles are assigned to cells and may drift out. The sorting operation is used to reassign particles to the appropriate cells.

The following list of steps was used to ensure that the tests were executed in a controlled runtime environment:

- All tests were compiled with the compiler optimizations turned on.
- The memory size of each application was checked to ensure no swapping out of the process, which constrains the performance so much that the results are tainted beyond all usefulness.
- Several idle daemons were permitted to run in the background, but the system process table was checked before and after each test was executed in order to verify that no other processes were competing for CPU time.

4.2 Mosix

The following sections cover the architecture of the Mosix parallel machine that was built from a cluster of PCs. Although there are several design decisions and tradeoffs that are required in implementing a production system, the system that was actually implemented is described here in detail, and the design alternatives are presented when they best aid in understanding this implementation.

4.2.1 Hardware Architecture

The hardware architecture of the Mosix parallel machine is the same as that described in 4.1.1.

4.2.2 Software Architecture

In order to run Mosix on a cluster of networked machines at least two software

packages are required to be setup and configured; a specific version of the Linux kernel and the Mosix package itself. Unlike the Beowulf style cluster, there is not very much network configuration required. All the machines need to be able to send/receive packets to each other's IP interface.

The Redhat 6.1 Linux distribution was used in implementing Mosix on the parallel machine. The entire content of the distribution was installed during the installation since disk space was readily available and there is often a convenience of having all the packages on the machine.

However, Redhat 6.1 ships with Linux kernel version 2.2.14, and the latest version of Mosix (0.97.7) requires Linux kernel version 2.2.16. Therefore it was necessary as part of the installation of Mosix to have a 2.2.16 kernel source tree available. This package is available from various Linux mirror-sites all over the world.

The Mosix installation script recompiles the Linux 2.2.16 kernel with the Mosix configuration settings. There is a configuration file (/etc/mosix.map) that describes the IP addresses of the nodes on the cluster and assigns them a Mosix node number.

The Mosix 0.97.7 technology consists of 2 parts; a Preemptive Process Migration (PPM) mechanism, and a set of algorithms for adaptive resource sharing. Both parts are implemented at the kernel level, using a loadable module, thus they are completely transparent to the application level. The PPM can migrate any process, at any time, to any available node. Usually migrations are based on information provided by one of the resource sharing algorithms, but users may override any automatic system-decisions and migrate their processes manually.

The granularity of the work distribution in Mosix is the process. Users can run parallel applications by initiating multiple processes on one node, then allowing the

system to assign these processes to the best available nodes at the time. If during the execution of the processes new resources become available, then the resource sharing algorithms are designed to utilize these new resources by possible reassignment of the processes among the nodes.

Scalability is achieved by using randomness in the system's control algorithms, where each node bases its decisions on partial knowledge about the state of other nodes, and it does not even attempt to determine the overall state of the cluster or of any particular node.

The main resource sharing algorithms of Mosix are the load-balancing and the memory ushering. The dynamic load balancing algorithm continuously attempts to reduce the load differences between pairs of nodes, by migrating processes from higher loaded to less loaded nodes. The memory ushering (depletion prevention) algorithm is geared to place the maximum number of processes in the cluster-wide RAM, to avoid thrashing or the swapping out of processes. The algorithm is triggered when a node starts excessive paging due to shortage of free memory.

To implement the PPM, the migration process is divided into two contexts: the user context - that can be migrated, and the system context - which cannot be migrated. The user context contains the program code, stack, data, memory maps and registers of the process. The user context enscapsulates the process when it is running at the user level. The system context contains a description of the resources which the process is attached to, and a kernel stack for the execution of system code on behalf of the process. The system context enscapsulates the process when it is running in the kernel. It holds the site dependant part of the system context of the process, hence it must remain on the node in which the process was originally created. While the user context can migrate many times between different nodes, the system context is never migrated.

The interface between the user-context and the system context is well defined and it is therefore possible to intercept every interaction between these contexts, and forward this interaction across the network.

For the Beowulf parallel computing machine, user applications could be written specifically to test the design, implementation and performance of such programs in a message passing parallel computing environment. There is no application specifically ported to Mosix because the nature of Mosix is such that applications do not need to be modified in order to run. The implementation of Mosix was intended to be completely transparent. However, combining Mosix with the Beowulf software was tested in order to see what kind of benefits are possible. The hypothesis is that MPI will be used to do the original job allocation for parallel jobs. Mosix will then redistribute the processes as necessary to maximize performance. This combination should provide some interesting results that take advantage of the best properties of each parallel architecture.

5. Results and Discussion

The results and the analysis of executing the user (UB1 and UB2) and industry (MG, LU, SP, BT, CG, EP, IS) benchmarks on the Beowulf cluster is presented below. The metrics generated from combining Mosix with the Beowulf software are also provided.

5.1 UB1 - The depth-first search algorithm

Although parallel versions of the depth-first-search algorithm do exist, only the serial version of this program was implemented, since the serial version could not be modified to function in parallel while maintaining any resemblance to the original serial implementation. This betrayed the objective of the tests which was to allow single processor and multi-processor systems to execute the same (or very similar) codes for comparisons purposes. This is a lesson worth noting: when porting applications to a parallel machine, breaking it up may be trivial, difficult or impossible to do.

5.2 UB2 - Floating point dense matrix mulitplication application

UB2 is a floating point dense matrix multiplication application. Storing, adding, and multiplying large quantities of floating point numbers requires very large amounts of computing power. The management of large quantities of floating point numbers stored in matrices is a common requirement among potential end-users of the Beowulf style cluster.

In order to optimize the parallel efficiency of the code, the ratio of floating-point operations to communication transfers was maximized in the program. The Beowulf style cluster that was built has fast processors compared to the relatively slow interprocess communication. This Beowulf configuration resulted in the parallel version of UB2 that executed on all four nodes of the cluster to run almost twice as fast as the serial case. The results of the tests are presented in Table 5.2 and Figure 5.2.

# of nodes	CPU time (s)
1	*
2	16.36
3	8.47
4	7.46
Serial	12.8

Table 5.2 - UB2 benchmark test results

* The parallel version UB2 does not support single CPU

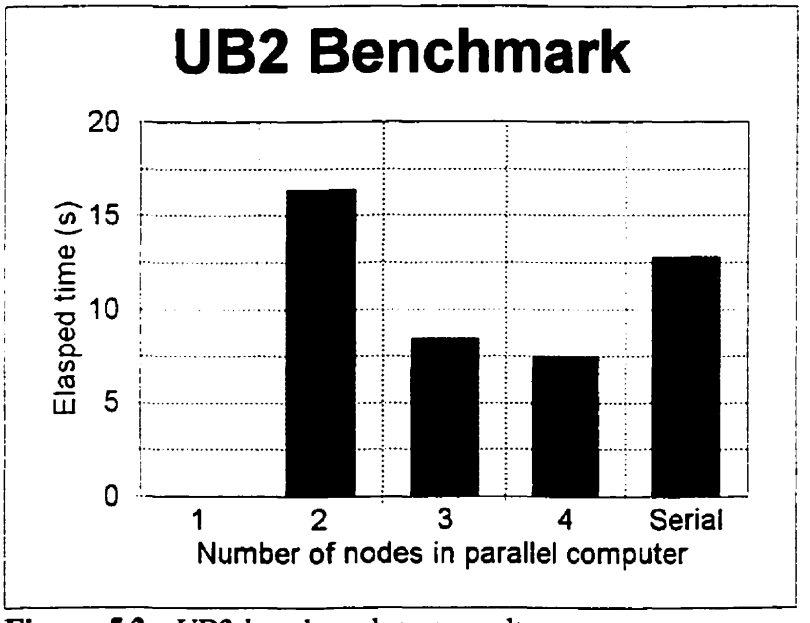

Figure 5.2 - UB2 benchmark test results

5.3 NPB-Block Tridiagonal

The Beowulf-style cluster completely failed to make any performance gains over the serial case when executing NPB-Block Tridiagonal (BT). Indeed, the Elapsed time increased by almost fourfold. The communication cost required to perform the transpose at each step makes BT appropriate only on extremely high bandwidth networks.

# of nodes	Elasped time (s)
1	131.43
2	*
3	*
4	529.19
Serial	130.41

Table 5.3 - BT benchmark test results

*BT requires that the number of nodes on the cluster be a square root.

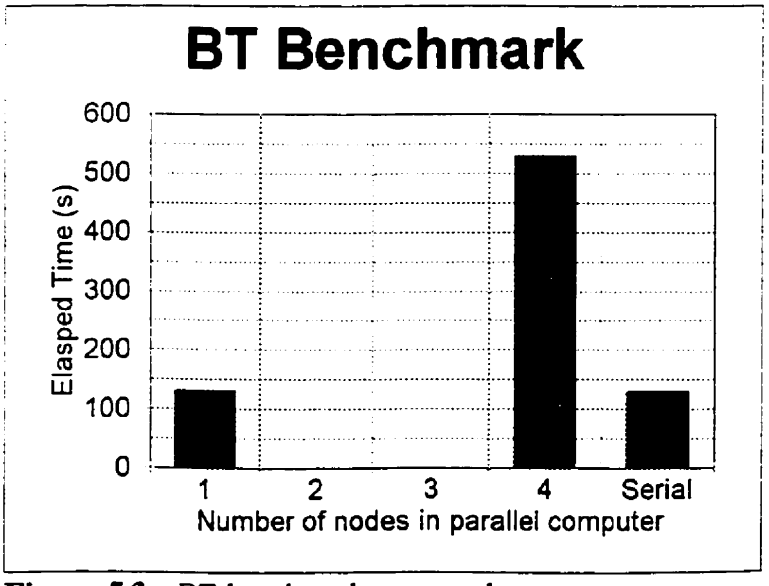

Figure 5.3 - BT benchmark test results

5.4 NPB-Conjugate Gradient

The purpose of the Conjugate Gradient Solver (CG) code is to set up a sparse matrix as posed by a typical conjugate gradient solver problem and solve it iteratively.

Noteworthy features of the code include the sparse storage scheme for the matrix which presents a problem for any parallel system. As shown in Table 5.4 and Figure 5.4, the performance improved by approximately 7% between the 2 node and serial case, but as in BT, the communication overhead of all 4 nodes attempting to work together caused the time to increase exponentially to 259.18 seconds in the 4 node cluster.

# of nodes	Elapsed time (s)
1	12.23
2	10.75
3	*
4	259.18
Serial	10.9

Table 5.4 - CG benchmark test results

* CG requires that the number of nodes on the cluster be a power of 2.

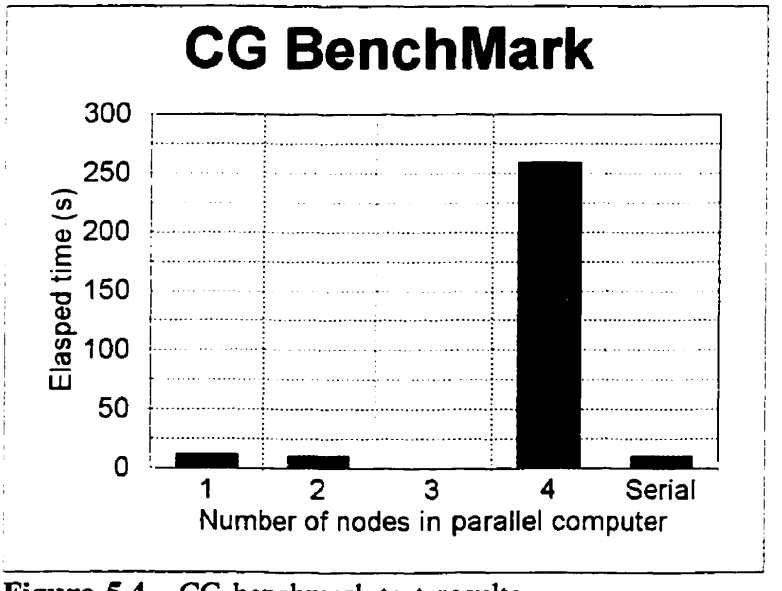

Figure 5.4 - CG benchmark test results

5.5 NPB-Embarrassingly Parallel

As indicated by its very name, this code is embarrassingly parallel, with little in the way of interprocessor communications.

As seen in Table 5.5 and Figure 5.5, there is a performance improvement for every additional CPU. This indicates that the cost of communication between the processes has little effect as the number of processors increases. Therefore, more processors could be added with an expected and measurable performance improvement, but at a diminishing rate.

# of nodes	User time (s)
1	61.75
2	33.67
3	22.4
4	16.93
Serial	61.78

Table 5.5 - EP benchmark test results

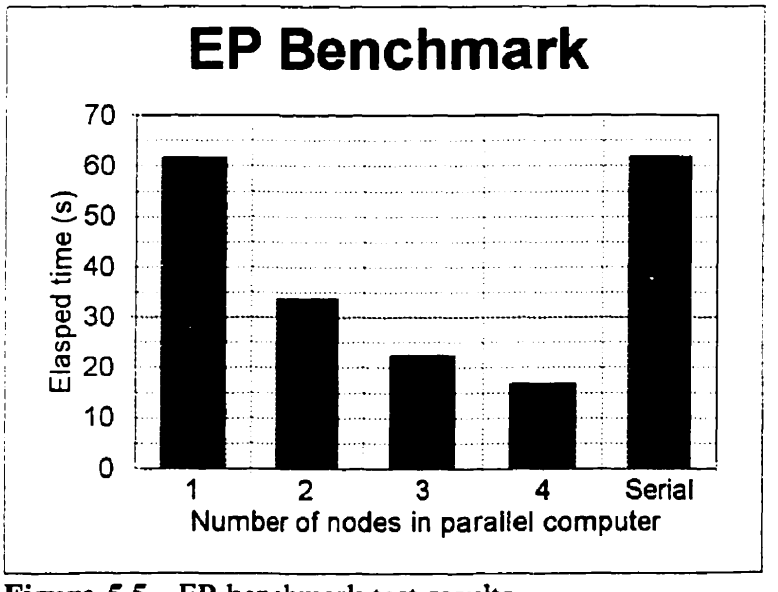

Figure 5.5 - EP benchmark test results

5.6 NPB-Integer Sort

The NPB-Integer Sort (IS) benchmark tests both integer computation speed and communication performance. As the results in Table 5.6 and Figure 5.6 indicate, the performance on the Beowulf cluster degrades with each processor added because of the poor communication performance of the cluster configuration.

# of nodes	Elasped time (s)
1	1.69
2	3.54
3	*
4	476.94
Serial	1.1

Table 5.6 - IS benchmark test results

* IS requires that the number of nodes in the cluster be a power of 2.

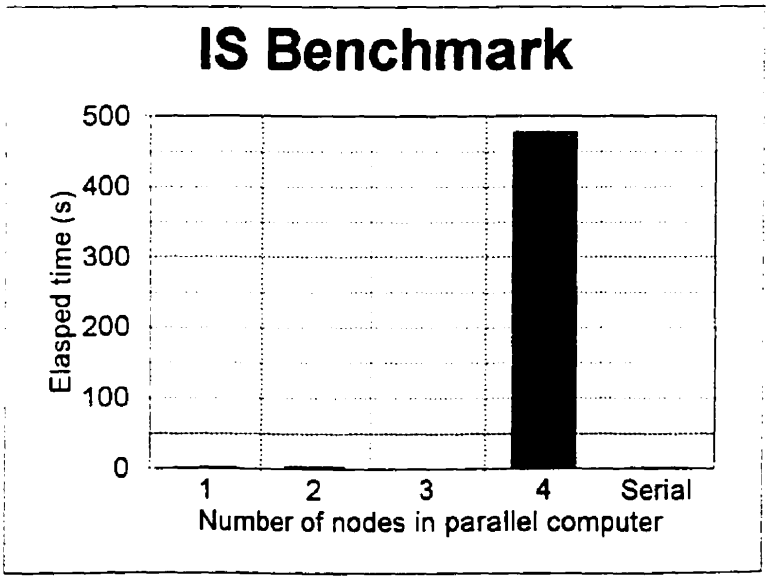

Figure 5.6 - IS benchmark test results

5.7 NPB-Lower Upper

The NPB-Lower Upper (LU) is very sensitive to the small-message communication performance of an MPI implementation. It is the only benchmark in the NPB 2.0 suite that sends large numbers of very small (40 byte) messages. As Table 5.7 and Figure 5.7 demonstrate, the performance improved with the 2 CPU cluster configuration, but degraded as the number of CPUs increased after that.

# of nodes	Elapsed time (s)
1	283.33
2	154.53
3	*
4	354.28
Serial	331.44

Table 5.7 - LU benchmark test results

* The code requires a power of 2 number of processors

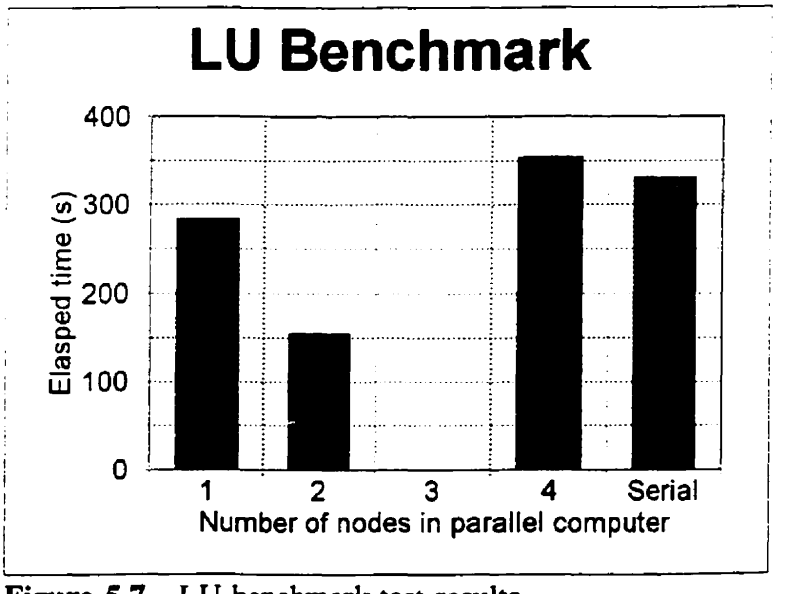

Figure 5.7 - LU benchmark test results

5.8 NPB-Multigrid

The NPB-Multigrid (MG) showed an improvement between 1 and 2 processors, but the performance degraded badly for 4 processors. Table 5.8 contains the results of the tests and Figure 5.8 shows the results in a graph.

# of nodes	Elasped time (s)
1	11.04
2	8.48
3	*
4	185.01
Serial	12.53

Table 5.8 - MG benchmark test result

* The code requires a power of 2 number of processors

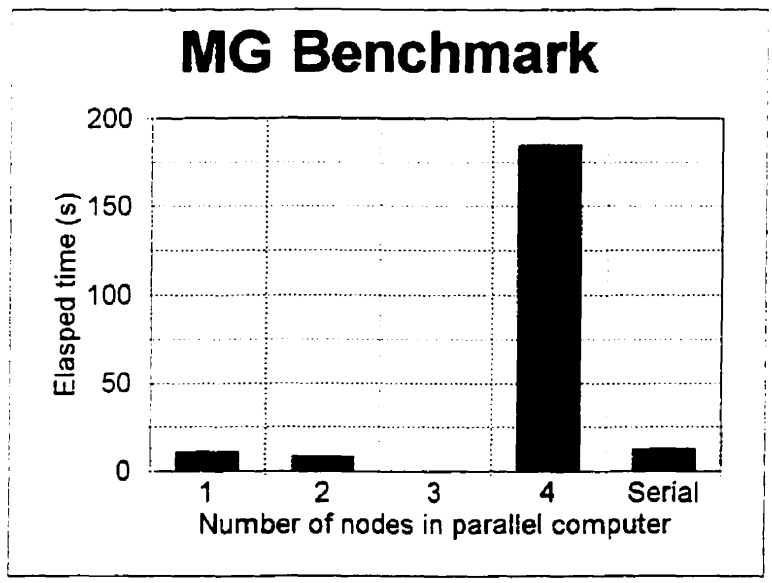

Figure 5.8 - MG benchmark test result

5.9 Mosix

One possible strategy to improve the performance of the cluster is to use Mosix to load-balance the running MPI applications. MPI was used to do the original job allocation for parallel jobs. Then Mosix redistributed the processes as necessary to maximize performance. This combination provided some interesting results that take advantage of the best properties of each parallel architecture. In order to test parallel applications running on Mosix, several of the NPB benchmarks (CG, EP, LU and MG) were executed in the 4 CPU cluster configuration in order to determine what kind of performance gains were possible when compared with Beowulf alone.

Benchmark, 4 nodes	**Elapsedtime (s) - Beowulf**	**Elasped time (s) - Beowulf and Mosix**
CG - 4	259.18	239.4
EP - 4	16.93	16.97
LU - 4	354.28	303.16
MG - 4	185.01	159.2

Table 5.9 - Mosix and Beowul benchmark test results

Table 5.9 shows that for applications with moderate amounts of communication between processes, the parallel architecture that combines Mosix with Beowulf shows an improvement over the Beowulf configuration. Mosix is also ideal for clusters with different speed nodes and/or memory sizes - because the adaptive resource allocation scheme of Mosix will allways attempt to maximize the performance.

6. Conclusion

As the performance of commodity computer and network hardware increase, and their prices decrease, it becomes more and more practical to build parallel computational systems from off the shelf components rather than buying time on a "supercomputer" or buying a dedicated shared-memory multi-processor system. However, because parallel computing can be implemented in a variety of ways, solving any particular problem in parallel will require some very important design decisions to be made. These decisions may dramatically affect portability, performance, and cost of implementing a software solution to the problem. In this project a synopsis of the software and hardware required to build a distributed multi-processor high performance computer was presented. A distributed multi-processor computer was designed and built using commodity PCs. Several industry standard parallel benchmarks were tested in order to generate performance metrics.

The results that were generated on the cluster indicate that applications that require even a modest amount of communication between the processes will suffer on the cluster. The latency between the nodes was too high and the cost of communication too important. Benchmarks like BT, CG, IS, MG all had intensive communication requirements and as such their performance decreased dramatically on the cluster. This indicates that applications within the domain of Agricultural and Biosystems Engineering that frequently need to communicate large chunks of data, are not suitable for implementation on the cluster. The Ecological Modelling application that was introduced in Section 1 is an example of an application that would not function more efficiently on the cluster unless it was rewritten and designed specifically to do so. The model has a large number of objects that can execute concurrently, but there are an exponential number of shared interactions between the objects that need to be passed between the nodes.

Other applications from the domain of Agricultural and Biosystems Engineering that

would have difficulty obtaining improved performance on the cluster are applications that use common algorithms or protocols from the field of artificial intelligence or artificial life. In this case, the problem is the same as that uncovered in UB1 - it can be prohibitively difficult to port many such algorithms and protocols to the parallel world. Making algorithms like the depth-first search algorithm in UB1 execute efficiently in parallel is a specialized area of research in itself, and in general would not be the efficient use an Agricultural and Biosystems engineer's time because of the depth of specialized knowledge required that is orthogonal to the domain of interest. It is also more esoteric to develop applications in a parallel programming environment, and so new tool-chains and environments often need to be explored adding to the overall cost of implementing the software in parallel.

There were however two benchmark tests that did exhibit significant performance gains. EP and UB2 both demonstrated strong performance increases. In these cases the communication between processes was minimal. This is a useful result. Applications that can be designed like UB2 in which a copy of the data can be provided to each process and for which there is not a large number of intermediate results required will execute efficiently on the parallel machine. Crunching through large sets of static data with computationally bound algorithms is such as example. Also applications that can be implemented similar to EP which are naturally parallel are a worthwhile port to the parallel computer. Potential applications include the same code of execution running a large number of times on separate data sets generating independant results.

Performance of the cluster could be improved by increasing bandwidth on the network by upgrading to a higher speed technology, but this approach requires modifying the LAN hardware infrastructure both at the user level (the NIC) and in the telecommunications closet or equipment room, and this can be prohibitively expensive (especially when the hardware is new on the market). A switched

ethernet infrastructure would speed things up. Legacy shared-architecture hubs operate using a bus architecture, whereby all users connected to the hub must share the hubs internal bus. Only one user (or hub port) has access to the bus at a given time. Therefore, while wire speed may be 100 Mbps, average throughput per user is more like 100 Mbps/N, where N is the number of users on the hub. Switches, on the other hand, do not have a bus. In a switched architecture each port is directly linked to every other port through a switching matrix or fabric. Full wire speed is then possible on each port since multiple transmissions can occur simultaneously. If architected properly, LAN switching can provide an instantaneous increase in total network bandwidth by a factor equal to the number of switch ports. In current PCs the host CPU is still required to mediate all of the demands of the I/O subsystems and interface cards. A heavily loaded CPU can spend up to 30% of its available clock cycles on overhead functions. This is extremely critical in server applications. Asynchronous interrupts, as from network card traffic, can severely impact efficiency.

Another route to better performance is to improve the software. Careful examination and algorithmic redesign of key subroutines could yield order-of-magnitude increases in performance in some cases. Whether running on a parallel computer or not, nearly every program has a mixture of parts. Some parts run in serial and other parts may run in parallel. Amdhal's Law describes how much of the program can be run in parallel and how much must be run using only a single processor. For example, if an application executes in 120 minutes, and it could run in parallel for 100 of the 120 minutes, the ratio of parallel to serial time puts an upper bound on the possible speedup for the application using multiple processors. This communication is an overhead that would not be present if the problem were run on a single processor. Therefore even during the parallel portions of the code, there are overhead factors that limit the speedup. Generally, as the number of processors increases, the relative impact of the overhead also increases and the returns continue

to diminish with increasing processors.

The language that the application is implemented in will also have a large impact on the performance of the application. Most computer science students and professionals use C and C++ or some other language focused on data structures or objects. To get the peak performance across a wide range of architectures, FORTRAN is the only practical language. The fundamental reason that data structure oriented languages are unsuitable for high performance computing is the extensive use of pointers. The problem with pointers is that the effect of a pointer operation is known only at execution time when the value of the pointer is loaded from memory. When an optimizing compiler processes a pointer, all optimizations are lost. It cannot make any assumptions about the effect of a pointer operation at compile time. It must generate conservative (less optimized) code that simply does exactly the same operation in machine code that the high-level language described.

There are other practical difficulties with the development of parallel software besides that it is challenging from a software design and implementation perspective. As the software and hardware architectures get more complicated, a new problem emerges. Managing and monitoring a cluster of machines running specialized software requires a large amount of system administration effort. Configuration must be managed across tens and potentially hundreds of machines. The need to execute jobs in an autonomous fashion and the ability to control how those jobs are run and what kind of resources they may use all needs to be managed in some controllable fashion. On traditional, large systems, batch processing systems are set up to allow users to submit jobs for execution with varying characteristics, such as how many CPU hours/minutes/seconds the jobs may use, and what time of day the job should or should not run. This kind of system management functionality is not currently possible on a cluster of PCs.

The continued development of computational tools, both hardware and software, is vital to increasing and improving the models of many environmental problems. Some applications will be good candidates for parallel computing.

7. References

Bailey, David, Tim Harris, Rob van der Wijngaart, William Saphir, Alex Woo and Maurice Yarrow. 2000. "The NAS Parallel Benchmarks"
Available: http://www.nas.nasa.gov/Software/NPB/Specs/npb2_0/npb2_0.html

Balsa, André D. "Linux Benchmarking HOWTO". v0.12, 15 August 1997.
Available: http://www.linuxdoc.org/HOWTO/Benchmarking-HOWTO.html.

Barak, Amnon. 2000. "MOSIX: Scalable Cluster Computing for Linux"
Available: www.mosix.org

Collins, Robert R. "Benchmarks: Fact, Fiction, or Fantasy?" Dr. Dobb's Journal, March 1998.
Available: http://www.ddj.com/articles/1998/9803/9803c/9803c.htm

DiBona, C. and S. Ockman and Stone. "Open Sources: Voices from the Open Source Revolution". Oreilly. 1st Edition January 1999

Dixit, Kaivalya M. "Overview of the SPEC Benchmarks".1992. IBM Corporation. Available:
http://www.benchmarkresources.com/handbook/chapter9.pdf

Dowd, Kevin and Charles Severance. "High Performance Computing". Oreilly. 1998: 247, 47, 193, 194, 249, 81, 33, 333.

Elias, Doug. "Fundamentals of Distributed Memory Programming". 1995
Available: http://www.tc.cornell.edu/Edu/Talks/DistMemProg/revision-notes.html

Fenton NE and Pfleeger SL. "Software Metrics: A Rigorous and Practical

Approach". International Thomson Computer Press, 1996

Gupta, M. M. and N. K. Sinha. "Intelligent Control Systems: Theory and Applications". IEEE Press, Piscataway, NJ, 1996.

Johnson, M. and E. Troan. "Linux Application Development". Addison-Wesley, 1998: 368.

Kersetter, Jim. 1998. "RSA's encryption challenge solved in 39 days". ZDNET. Available: http://www.zdnet.com/zdnn/content/pcwo/0226/288730.html

Parrott, L. and R. Kok. "Use of an object-based model to represent complex features of ecosystems". Dept. of Agricultural & Biosystems Engineering, McGill University, Montreal, CA

Mehat, San. "Network Montoring for Custers". Journal of Linux Technology. 2000. vol. 1, #1: 8

Moin, P. and K. John. "Tackling Turbulence with Supercomputers". Scientific American, January 1997: 63.

Radajewski, J. and D. Eadline. "BeoWulf Howto". v1.1.1, 22 November 1998. Available: http://www.linuxdoc.org/HOWTO/Beowulf-HOWTO.html

Spector, David. "Managing Beowulf Clusters". Journal of Linux Technology. 2000. vol. 1, #1: 18.

Wicker, Luios J. "Simulating Severe Weather". Dr. Dobb's Journal, March

1999, #297: 18.

"Cluster Computing". LANL. 2000.
Available: http://www.lanl.gov/worldview/science/features/cluster.html

Technical Specifications from the "Processor Hall of Fame". INTEL. 2000.
Available: http://www.intel.com/intel/museum/25anniv/hof/moore.htm

Technical Specifications from the "Processor Hall of Fame". INTEL. 2000.
Available: http://www.intel.com/intel/museum/25anniv/hof/tspecs.htm

"Use Your PC at Home for Seti?". 2000.
Available: http://www.seti-inst.edu/science/setiathome.html