UNIVERSITY OF MINNESOTA

This is to certify that I have examined this copy of master's thesis by

Vishwas Raman

and have have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Dr. Richard Maclin

Name of Faculty Advisor

Signature of Faculty Advisor

Date

GRADUATE SCHOOL

Abstract

A common computing environment consists of many workstations connected together by a high speed network. These workstations have grown in power over the years, and if viewed as an aggregate they can represent a significant computing resource. Using such workstation clusters for distributed computing has thus become popular with the proliferation of inexpensive, powerful workstations. In this thesis, we present a method for building a software system that runs on a cluster of workstations and automates the execution of sequential jobs in these workstations. The objective is that the scheduling in this system must ensure that only the idle cycles are used for distributed computing and that local users, when they are operating, have full control of their machines. The unique feature of this system is that the checkpointing is user-initiated and that processes submitted to the system must have checkpointing code implemented in them. This ensures that *smaller* memory images of migrating processes are captured to restore the state of the process at a later stage, a distinguishing feature from existing traditional systems. Results of the various tests performed show that the system presented herein manages to make use of idle time on various workstations to run large processes across the network without creating large checkpoint images for interrupted processes while ensuring top priority for owners of individual machines.

Acknowledgements

I would like to thank my advisors, Dr. Richard Maclin and Dr. Masha Sosonkina Driver, for their help and guidance that has seen me through this thesis. I would like to express my gratitude to the other member of my examination committee, Dr. Robert McFarland, for his patience and support. I would also like to thank the entire computer science faculty and all my fellow graduate students for their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many computing environments consist of a number of workstations connected together by a high speed local area network. These networks of workstations have grown in power over the years, and if viewed as an aggregate they represent a significant computing resource. But in many cases, even though these workstations are owned by a single organization, they are dedicated to the exclusive use of individuals.

In examining the usage patterns of the workstations, we find it useful to identify three “typical” types of users [BLL91]. The first type of users are individuals who mostly use their workstations for sending and receiving mail or preparing papers. Theoreticians and administrative people often fall into this category. We identify many software development people as the second type of users. These people are frequently involved in the debugging cycle where they edit software, compile, then run it possibly using some kind of debugger. This cycle is repeated many times during a typical working day. These users sometimes have too much computing capacity on their workstations such as when editing, but then during the compilation and debugging phases they could often use more CPU power. Finally there are the third type of users who frequently do large numbers of simulations, or combinatoric searches. These people are almost never happy with just a workstation, because it really isn’t powerful enough to meet their needs. Another point is that most of the first two types of users leave their machines completely idle when they are not working, while the third type of users often keep their machines busy the entire day. Such users rely heavily on computing throughput. A computing environment that provides large amounts of computational power over a long period of time may be essential to solve the problems of such users. Such an environment is called a High Throughput Computing (HTC) Environment.

In a HTC environment, the available resources can be made use of in a more efficient manner by identifying idle machines and exploiting their availability to run jobs. Foreign jobs can be permitted to run on workstations when its owner is not using the machine assuming the owner has given prior permission to do so. This process of running jobs that do not belong to the workstation's owner is referred to as *cycle stealing*. It should also be possible for a user to submit many jobs at once to a system which in turn will find idle machines as they become available and submit the jobs to them. This way, large amounts of computation can be done with very little intervention from the user.

1.1 Motivation

Historically, users of computing facilities have been mainly concerned with the response time of applications while system administrators have been concerned with throughput. Users often judge the power of a system by the time taken to perform a fixed amount of work. Given this fixed amount of computing to perform, the question most users ask is: how long will I have to wait to get the results of this computation?

1.1.1 Motivating Example

Consider a situation in which there are 10 workstations and there are 10 users in a network - the owners of these workstations. Eight of these users work predominantly during the day, and leave their machines idle during the night. The remaining two users utilize their machines mostly during the night and leave them idle during the day. Every user runs his/her processes on his/her own machine. We find that the load on most of the machines is high primarily during the day. The number of processes run by the eight users during the day will tend to be high; there might even be processes in their machines waiting for the CPU when there are a couple of machines in the network which are lying idle. Similarly, during the night, the load on the couple of machines used by the night users is very high and there might be processes in these machines waiting for the CPU, when there are 8 machines lying idle in the network.

Figure 1.1 illustrates a similar case with a network of 5 workstations. It is clear that while there are processes waiting to execute on some machines, the CPU's of some other machines in the network are idle. The idle CPU cycles of these

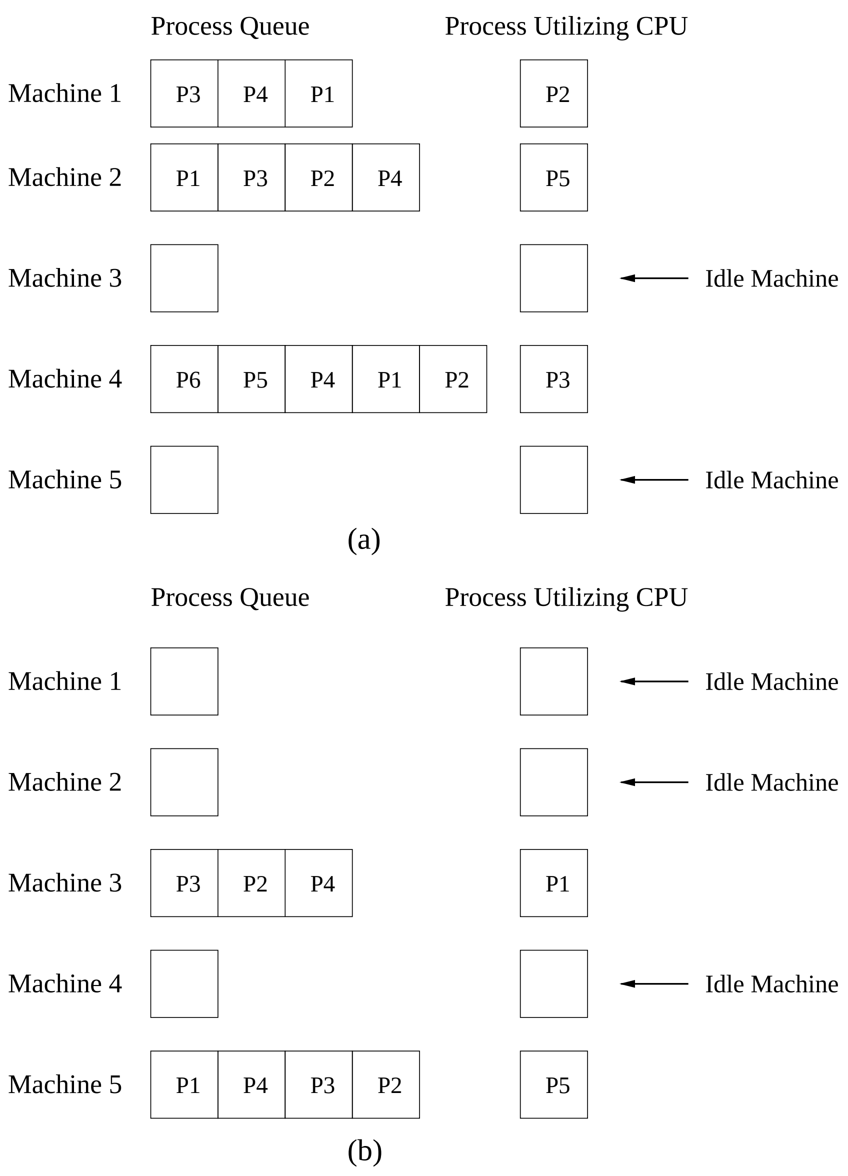

Figure 1.1: A network of 5 workstations shown (a) during the day time when 2 of the machines are idle while there are processes waiting to execute on other machines. (b) during the night time when the same 2 machines have a heavy load while the other machines are idle.

machines can in turn be used to execute these waiting processes. For example in situation (a) of Figure 1.1 it would benefit the users of machines 1, 2 and 4 if it was possible to turn such distributively owned collections of workstations into a HTC environment utilizing idle CPU cycles. The research herein focuses on building a software system that runs on a cluster of workstations to harness wasted CPU cycles.

1.2 Problem Description

For many scientists, their research is heavily dependent on computing throughput. It is not uncommon to find problems that require weeks or months of computation to solve. The computing needs of these throughput oriented users are satisfied by HTC environments as mentioned in the example above. These users are less concerned about the instantaneous performance of the environment (typically measured in Floating Point Operations per Second (FLOPS)), but are more interested in the amount of computing they can harness over a month or a year. It is an active area of interest among researchers to see how many jobs they can complete over a long period of time (high throughput of jobs).

The key to high-throughput is the efficient use of available resources. Instead of being concentrated in a single mainframe computer, an institution's computing power is "distributed" across many small computers [Fie93]. In an environment of distributed ownership, the total computational power of the institution may be high but the resources available to the individual users remains roughly the same. In such an environment, many machines sit idle for long periods of time [LM93]. A question that we might ask: Is it possible to use this idle time on the machines for a useful purpose? In fact, in order to improve the throughput of the system, it is very important that these idle cycles be used for processing.

1.2.1 A Possible Solution

Our Solution focuses on two issues. First, the available resources should be made use of in a more efficient manner by identifying idle machines and exploiting their availability to run jobs. Foreign jobs can be permitted to run on workstations when its owner is not using the machine (assuming the owner has given prior permission to do so).

Second, the resources available to a given user can be expanded by allowing the user to take advantage of idle machines to which they would not otherwise have access. This can be done by providing uniform access to the distributively owned resources through the use of a master process. This thesis is an effort towards building a software system that runs on a cluster of workstations to harness wasted CPU cycles.

The primary issue to be taken into consideration is the checkpointing of processes. In order to allow migrating processes to make progress, the system must write a checkpoint of the process's state before vacating a machine and putting it back in the job queue to be matched with another idle machine whenever one is available. When the job is restarted, the state contained in the checkpoint file is restored and the process resumes from where it left off. In this way, the job migrates from one machine to another. However, in traditional systems, the checkpointing of a process is done by saving the complete image of the process's virtual memory address space in a file. Only a part of this entire information is necessary to restart the process from the saved state. The checkpoint file ends up being very large and moreover, most of the information stored in it is not needed. Other issues to be taken into consideration are the amount of time taken by processes to quiesce themselves and the number of workstations that should be part of the cluster.

1.2.2 Proposed Thesis Work

Recent trends show that more and more computing environments are becoming fragmented and distributively owned, resulting in more and more wasted computing power. There are existing systems such as Condor [BLL91] that make use of wasted CPU cycles. But such systems use the traditional method of checkpointing by creating large checkpoint files for processes migrating from one machine to another.

Thesis Intent: Develop a system to automate the execution of sequential jobs in a cluster of distributively owned workstations. This system should store small checkpoint images for migrating processes as contrasted with currently existing systems.

This thesis is an effort to provide answers to the following questions:

- **Question 1:** *Can a parallel system be built that can manage a collection of distributively owned workstations having the potential to make use of idle time on various workstations to run large processes across the network but still ensuring top priority for owners of machines?*

- **Question 2:** *Can these large processes be run across the network without creating huge memory images (saved states of the processes) like most other systems do every time a process is interrupted?*
- **Question 3:** *Would jobs involving large scale computations perform better in such a system or not?*

1.3 What Follows

In Chapter 2, I will provide background material for the thesis which will include the concepts of parallel processing and distributed computing, the idea of cycle stealing and the desirable features of distributed computing environments such as these. Chapter 3 will describe the design and architecture of my system along with the issues taken into consideration. In Chapter 4, I will present the results of the experiments that were performed on the system along with their evaluations. In the final chapter, I will draw conclusions and explain how my thesis attempts to answer the proposed questions. I will also present directions for future research.

Chapter 2

Background

In this chapter, I will provide background material for this thesis. The first section will provide a brief introduction to the ideas of parallel processing and distributed computing. The second section will deal with the concept of cycle stealing in a network of workstations. The third section will provide the necessary background on Distributed Computing Environments (DCE) and Distributed Systems. It will describe some of the different features to be taken into consideration in distributed systems. The next section will provide short descriptions of some of the existing systems. The next section will cover the different types of jobs that we will use to test the work presented in this thesis. The final section will provide a brief introduction to CORBA - the middleware that is used in my system, its advantages and the reason it is used.

2.1 Parallel and Distributed Processing

Parallel processing has emerged as a key enabling technology in modern computers, driven by the ever-increasing demand for higher performance, lower costs, and sustained productivity in real-life applications. Concurrent events are taking place in today's high-performance computers due to the common practices of multiprogramming, multiprocessing, and multi-computing.

Parallel processing is the execution of program instructions by dividing them among multiple CPU's with the objective of running a program in less time. In the earliest computers, only one program ran at a time. A computation-intensive program that took one hour to run and a tape copying program that took one hour to run would take a total of two hours to run. Batch processing, in which non-interactive jobs are submitted to the system to be processed at appropriate times, is an important method of computing in a centralized environment. Although supporting only limited interactions with users (primarily through input and output files), batch processing has the advantages of allowing users to submit multiple jobs without having to wait for them to finish, and of giving the system the flexibility to schedule the jobs according to the jobs' priorities and as computing resources become available.

An early form of parallel processing allowed the interleaved execution of both programs together. The computer would start an I/O operation, and while it was waiting for the operation to complete, it would execute the processor-intensive program. The total execution time for the two jobs would be a little over one hour. The next improvement was multiprogramming. In a multiprogramming system, multiple programs submitted by users were each allowed to use the processor for a short time. Then the operating system would allow the next program to use the processor for a short time, and so on. To users it appeared that all of the programs were executing at the same time. Problems of resource contention first arose in these systems. Explicit requests for resources led to the problem of deadlocks.

The next step in parallel processing was the introduction of multiprocessing. In these systems, two or more processors shared the work to be done. The earliest versions had a master/slave configuration. One processor (the master) was programmed to be responsible for all of the work in the system; the other (the slave) performed only those tasks it was assigned by the master. This arrangement was necessary because it was not then understood how to program machines so they could cooperate in managing the resources of the system.

2.1.1 The Concept of Distributed Computing

Computing is said to be "distributed" when the computer programs and data that computers work on are spread out over more than one computer, usually over a network. Distributed computing allows different users or computers to share information. Distributed computing can allow an application on one machine to make use of processing power, memory, or storage on another machine. It is possible that distributed computing could enhance performance of a stand-alone application, but this is often not the reason to distribute an application.

2.2 Cycle Stealing

Studies have shown that up to three-quarters of the time workstations in a network are idle [RH95]. Systems such as Condor [LLM98], LSF [Com96], and NOW [AKPtNt95] have been created to use these available resources. Such systems define a “social contract” that permits foreign jobs to run only when a workstation’s owner is not using the machine. To enforce this contract, foreign jobs are stopped and migrated as soon as the owner resumes use of their computer.

We use the term *cycle stealing* to mean running jobs that do not belong to the workstation’s owner. The idle cycles of machines can be defined at different levels. Traditionally, studies have investigated using machines only when they are not in use by the owner. Thus, the machine state can be divided into two states: idle and non-idle. In addition to processor utilization, user interaction such as keyboard and mouse activity has been used to detect if the owner is actively using their machine. Acharya et al, [AES97] showed that for their definition of idleness, machines are in a non-idle state 50% of the time.

Two strategies have been used in the past to migrate foreign jobs: Immediate-Eviction and Pause-and-Migrate. In Immediate-Eviction, the foreign job is migrated as soon as the machine becomes non-idle. Because this can cause unnecessary, expensive migrations for short non-idle intervals, an alternative policy, called Pause-and-Migrate, that suspends the foreign processes for a fixed time prior to migration, is often used. The fixed suspend time should not be long because the foreign job makes no progress in the suspend state. We will use an Immediate-Eviction approach with minor variations in our system because giving the owner of the machine top priority and not affecting his/her interactive performance is of utmost importance to us.

2.3 Theory Behind Distributed Systems

A Distributed System (DS) consists of a collection of autonomous computers linked by a computer network and equipped with distributed system software. Distributed System software enables computers to coordinate their activities and to share the resources of the system - hardware, software and data. The users of a DS should perceive a single, integrated computing facility even though it may be implemented by many computers in different locations.

A DS must provide facilities for encapsulating resources in a modular and protected fashion, while providing clients with network-wide access. Kernels and

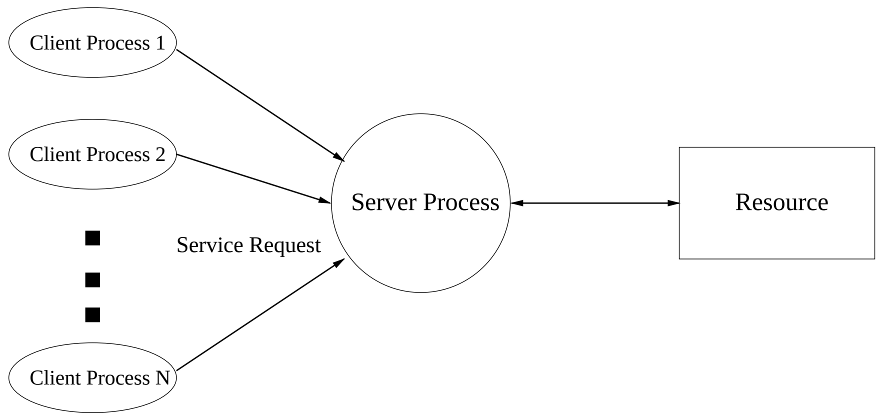

Figure 2.1: Distributed Computing: A Client/Server Model

servers are both resource managers. They contain resources, and as such they have to provide:

- *Encapsulation:* They should provide a useful service interface to their resources, that is, a set of operations that meet their clients' needs. The details of the management of memory and devices used to implement resources should be hidden from clients, even when they are local.
- *Concurrent Processing:* Clients may share resources and access them concurrently. Resource managers are responsible for achieving concurrent transparency.
- *Protection:* Resources require protection from illegitimate accesses - for example, files are protected from being read by users without read permission, and device registers are protected from application processes.

A typical Client-Server model in distributed computing is illustrated in Figure 2.1.

There are three major problems with the client/server model:

- Control of individual resources is centralized in a single server.
- Each single server is a potential bottleneck.
- To improve the performance, multiple implementations of similar functions must be used.

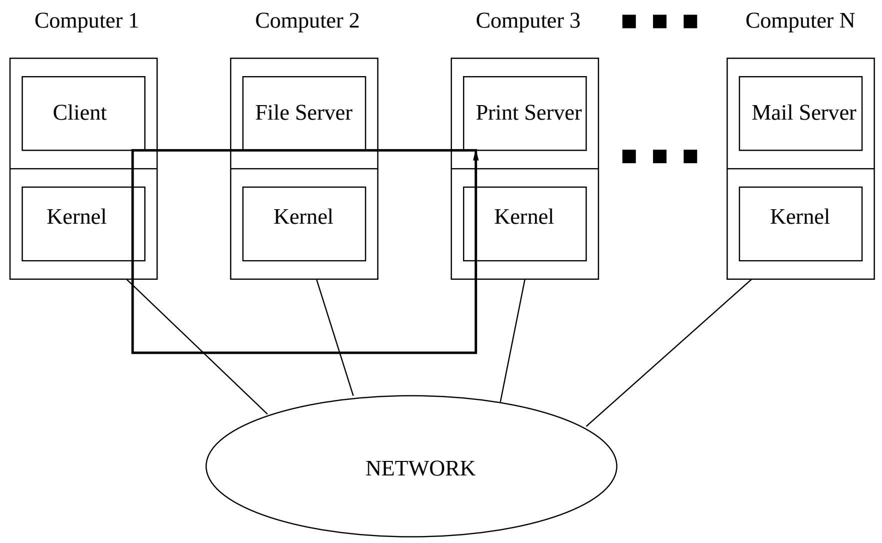

Figure 2.2: The Client/Server Model in a Distributed System

The deficiencies of the client/server model led to the development of an integrated model. According to this model, each computer's software is designed as a complete facility with a general file system and name interpretation mechanisms. This implies that each computer in a distributed system would run the same software. A distributed system that has been developed based on the integrated model can be easily made to look like a client/server based system if suitable configuration flexibility has been provided. A modified client/server model in a distributed system is shown in Figure 2.2.

With the dramatic improvement in microprocessor performance and networking technology, distributed computing is increasingly replacing mainframe and supercomputer based centralized computing. Across various industries, the production workload running on mainframes and supercomputers is being migrated to distributed server machines and workstations [WZAL93].

Distributed computing systems provide the user community with the ability to share resources on powerful workstations connected by a high speed network [TKDA96]. In a distributed system, the use of workstations varies. In many cases, workstations are utilized during the day-light hours to develop applications, and run services (e.g., word processing, database, electronic mail, etc.). Also, there are users who can always use additional workstations to satisfy their large-scale computing needs.

Distributed Computing Environment (DCE) is an industry-standard software technology for setting up and managing computing and data exchange in a system of distributed computers [Bur]. DCE is typically used in a larger network of computing systems that include different sized servers scattered geographically. DCE uses the client/server model. Using DCE, application users can use applications and data at remote servers. Application programmers need not be aware of where their programs will run or where the data will be located. Much of the DCE setup requires the preparation of distributed directories so that DCE applications and related data can be located when they are being used.

2.3.1 Important features of Distributed Systems

Certain features are essential for a Distributed Processing System (DPS) [TKDA96] to be an acceptable way of sharing computing resources. The local users need to be convinced that the DPS will not hurt their interactive performance. Also, the DPS must provide the owners priority in the use of their machines when they have jobs to execute.

The DPS services are needed for scheduling and load balancing. There are several issues: (a) hunting for idle workstations, (b) guaranteeing that the local users are least affected, (c) making efficient use of idle workstations, (d) ensuring that jobs are not lost, and (e) providing a fair use of the resources to multiple users.

Using workstation clusters for distributed computing has become popular with the proliferation of inexpensive, powerful workstations. Workstation clusters offer both a cost effective alternative to batch processing and an easy entry into parallel processing. However, a number of workstations on a network does not constitute a cluster. We can define a cluster to be a collection of computers on a network that can function as a single computing resource through the use of additional cluster management software. The following features can be identified as desirable for a DPS system to be efficient and acceptable in environments involving a network of workstations. These could also be used as evaluation criteria to facilitate the comparison of such systems. Note that the terms DPS and cluster are used interchangeably in the following criteria.

1. *Heterogeneous Support:* There are two types of cluster environments, homogeneous and heterogeneous. A homogeneous computing environment consists of a number of computers of the same architecture running the same operating system. A heterogeneous computing environment consists

of a number of computers with dissimilar architectures and different operating systems.

2. *Batch Support:* The cluster should allow users to submit multiple non-interactive jobs without having to wait until they terminate. A popular use of clusters is off-loading batch jobs from saturated supercomputers [AL93]. Clusters can often provide better turn around time than supercomputers for small (in terms of memory and CPU requirements) batch jobs.

3. *Parallel Support:* There is interest in moving to massively parallel processing machines via heterogeneous processing because the needs within an application might not be uniform. A cluster can serve as a parallel machine because workstations are inexpensive and easier to upgrade. A number of packages such as Parallel Virtual Machine (PVM) [GBD$^+$94] and Express [JK91] add parallel support for computers distributed across a network.

4. *Message Passing Support:* Message passing is the ability to pass data between processes in a standardized way. This inter-process communication allows several processes to work on a single problem in parallel. A large distributed application can be split across the many different platforms that exist in a heterogeneous environment. Some cluster management packages do not provide their own message passing support, choosing instead to rely on packages such as PVM and Linda [CGM93] to provide that feature.

5. *Checkpointing:* Checkpointing is a common method used by cluster management software to save the current state of the job [LS92]. In the event of a system crash, the only lost computation will be from the point at which the last checkpoint file was made. Because checkpointing in a heterogeneous environment is more difficult than on a single architecture, most current cluster management software that provides checkpointing does so with the following limitations:

   - Only single process jobs are supported (i.e., no fork(), exec(), or similar calls are allowed).
   - No signals or signal handlers are supported (i.e., signal(), sigvec(), and kill() are not supported).
   - No interprocess communication (i.e., sockets, send(), recv(), or similar call are not implemented).

- All file operations must either be read only or write only. These limitations will make checkpointing unsuitable for certain applications, including parallel or distributed jobs that must communicate with other processes.

6. *Process Migration:* Process migration is the ability to move a process from one machine to another machine without restarting the program, thereby balancing the load over the cluster. Process migration would ideally be used if the load on a machine becomes too high or someone logs on to the machine, thus allowing processes to migrate to another machine and finish without impacting the workstation owner.

7. *Local User Autonomy:* Local users should be guaranteed the top priority over the use of their workstations. The distributed processes should not adversely affect the interactive performance for local users. The DPS should use only the idle computing resources. If local users return to their workstations, the DPS must vacate the machines immediately and restart jobs at other idle workstations.

8. *Local Autonomy for Each owner:* Local autonomy for each workstation owner means that the participation in the cluster must be on a voluntary basis. The owners of workstations should have control to withdraw their machines from the pool when desired. A reconfiguration of the cluster system should be possible without having to reboot the system. The jobs running on other machines in the cluster should continue to run unaffected.

9. *Load Balancing:* Load balancing refers to the distribution of the computational workload across a cluster so that each workstation in the cluster is doing an equivalent amount of work. On a network, some machines may be idle while others are struggling to process their workload [AL93]

10. *Job Run-Time Limits:* A run time limit sets the amount of CPU time a job is allowed for execution. Providing a limit ensures that smaller jobs complete without a prolonged delay incurred by waiting behind a job that runs for an excessive period of time.

11. *Scalability:* Scalability of a cluster is defined as the effect on its performance when the number of workstations which are part of the pool is increased. The scalability of the workstation cluster may be limited by the existing software (e.g., network file system, authentication software, etc.) or the hardware

(network). However, the cluster management software should not introduce new bottlenecks that limit scalability. Also, the overheads of processing a job through the DPS should not grow as a result of adding more workstations to the pool.

12. *Ease of use:* It should be possible for users to run their programs through the cluster without requiring a major modification of their application programs. It should also be easy for new users to join the DPS pool. There should be facilities for submitting, monitoring and terminating jobs submitted through the system.

2.4 Existing Systems

In this section, I will provide a detailed description of the Condor High Throughput Computing System and the BATRUN Distributed Processing System which were a big motivation behind my thesis. I will also be providing short descriptions of some other existing systems such as Distributed Queuing System (DQS) [DGP96] and Computing in DIstributed Networked Environments (CODINE) [F93].

2.4.1 Condor High Throughput Computing System

Condor is a software system that runs on a cluster of workstations to harness wasted CPU cycles. A Condor pool consists of any number of machines, of possibly different architectures and operating systems, that are connected by a network. Condor is built on the principle of distributing batch jobs around this loosely coupled cluster of computers. Condor attempts to use idle CPU cycles that exist on some machines to provide more computational power to users who need them [AL93].

Several principles have driven the design of Condor. First is the principle that workstation owners should always have the resources of the workstation they own at their disposal. This is to guarantee immediate response, which is the reason most people prefer a dedicated workstation over access to a time sharing system. The second principle is that access to remote capacity must be easy, and should approximate the local execution environment as closely as possible. Portability is the third principle behind the design of Condor [LL90]. This is essential due to rapid developments in the workstations on which condor operates.

2.4.1.1 Mechanisms Used in Condor

Five mechanisms are basic to the operation of Condor. The first is a mechanism for determining when a workstation is not in use by its owner, and thus should become part of the pool of available machines. This is accomplished by measuring both the CPU load of the machine, and the time since the last keyboard or mouse activity. Individual workstation owners can customize the parameters that determine the state of that workstation - idle or non-idle.

Second is a mechanism for "fair" allocation of these machines to users who have queued jobs. This task is handle by a centralized "machine manager". The manager allocates machines to waiting users on the basis of priority. The priority is calculated according to the *up-down* algorithm. This algorithm periodically increases the priority of those users who have been waiting for resources, and reduces the priority of those users who have received resources in the recent past. The purpose of this algorithm is to allow heavy users to do very large amounts of work, but still protect the response time for less frequent users.

Thirdly, Condor provides a remote execution mechanism which allows its users to run the same programs that they had been used to running locally after only a re-linking step. File I/O is redirected to the submitting machine, so that users do not need to worry about moving files to and from the machines where execution actually takes place.

The fourth mechanism is responsible for stopping the execution of a Condor job upon the first user activity on the hosting machine. As soon as the keyboard or mouse becomes active, or the CPU load on the remote machine rises above a specified level, a running Condor job is stopped. This provides automatic return of the use of the hosting workstation to its owner.

Finally Condor provides a transparent checkpointing mechanism which allows it to take a checkpoint of a running job, and migrate that job to another workstation when the machine it is currently running on becomes busy with non-Condor activity. This allows Condor to return workstations to their owners promptly, yet provide assurance to Condor users that their jobs will make progress, and eventually complete.

To meet the portability requirement, all these five mechanisms were implemented entirely outside the UNIX kernel. Even checkpointing is quite portable, but does depend on the specific formats of the "a.out" and "core" files for each system.

2.4.1.2 Architecture and Procedure

One machine in the pool, the “central manager” (CM) keeps track of all the resources and jobs in the pool. To monitor the status of the individual computers in the cluster, certain Condor programs called the Condor “daemons” must run all the time. One daemon is called the “master”. Its only job is to make sure that the rest of the Condor daemons are running. If any daemon dies, the master restarts it. If a daemon continues to die, the master sends mail to a Condor administrator and stops trying to start it. Two other daemons run on every machine in the pool, the *startd* and the *schedd*. The schedd keeps track of all the jobs that have been submitted on a given machine. The *startd* monitors information about the machine that is used to decide if it is available to run a Condor job, such as keyboard and mouse activity, and the load on the CPU. Since Condor only uses idle machines to compute jobs, the *startd* also notices when a user returns to a machine that is currently running and removes the job.

All of the schedds and startds of the entire pool report their information to a daemon running on the CM called the *collector*. The *collector* maintains a global view, and can be queried for information about the status of the pool. Another daemon on the CM, the *negotiator*, periodically takes information from the *collector* to find idle machines and match them with waiting jobs. This process is called a ”negotiation cycle” and usually happens every five minutes (See Figure 2.3).

Besides the daemons which run on every machine in the pool and the central manager, Condor also consists of a number of other programs. These are used to help manage jobs and follow their status, monitor the activity of the entire pool, and gather information about jobs that have been run in the past. These are commonly referred to as the Condor “tools” [Ove].

Every Condor job involves three machines. One is the submitting machine, where the job is submitted from. The second is the central manager, which finds an idle machine for that job. The third is the executing machine, the computer that the job actually runs on. In reality, a single machine can perform two or even all three of these roles. In such cases, the submitting machine and the executing machine might actually be the same piece of hardware, but all the mechanisms described here will continue to function as if they were separate machines. The executing machine is often many different computers at different times during the course of the job’s life. However, at any given moment, there will either be a single execution machine, or the job will be in the job queue, waiting for an available computer.

Every machine in the pool has certain properties: its architecture, operating

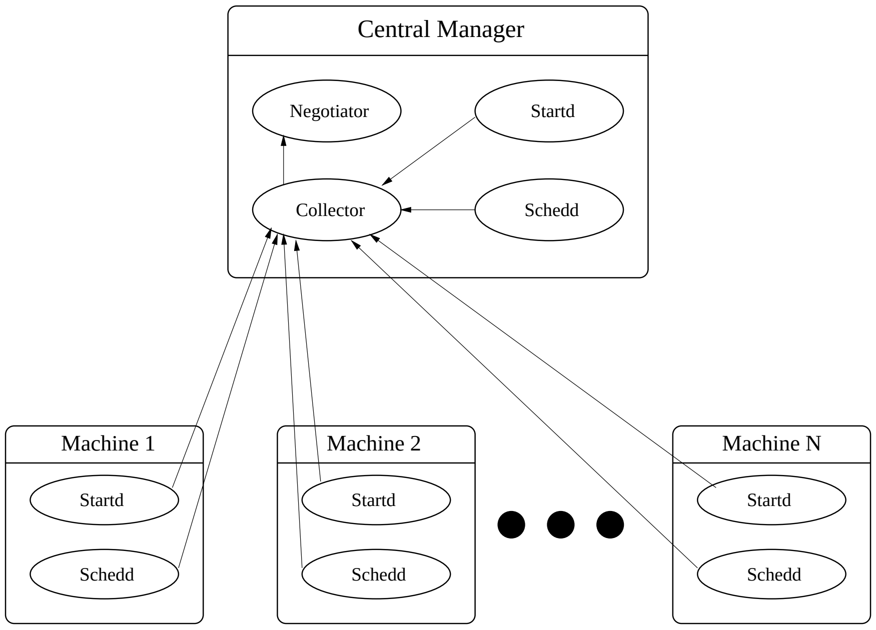

Figure 2.3: Architecture of a Condor Pool(With no jobs running)

system, amount of memory, the speed of its CPU, amount of free swap and disk space, and other characteristics. Similarly, every job has certain requirements and preferences. A job must run on a machine with the same architecture and operating system it was compiled for. Beyond that, jobs might have requirements such as how much memory they need to run efficiently, how much swap space they will need, etc. Preferences are characteristics the job owner would like the executing machine to have but which are not absolutely necessary. If no machines that match the preferences are available, the job will still function on another machine. The owner of a job specifies the requirements and preferences of the job when it is submitted. The properties of the computing resources are reported to the central manager by the startd on each machine in the pool. The negotiator's task is not only to find idle machines, but machines with properties that match the requirements of the jobs, and if possible, the job preferences.

When a match is made between a job and a machine, the Condor daemons on each machine are sent a message by the central manager. The schedd on the submitting machine starts up another daemon, called the ”shadow”. This acts as the connection to the submitting machine for the remote job, the shadow of the

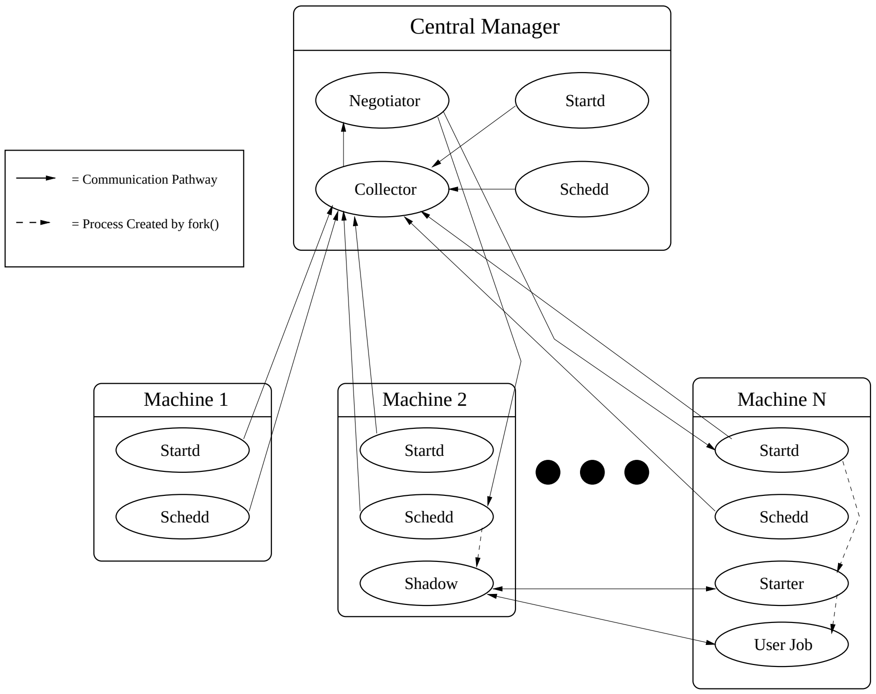

Figure 2.4: Architecture of a Condor Pool(With a job submitted on Machine 2 running on Machine N)

remote job on the local submitting machine. The startd on the executing machine also creates another daemon, the ”starter”. The starter actually starts the Condor job, which involves transferring the binary from the submitting machine (see Figure 2.4). The starter is also responsible for monitoring the job, maintaining statistics about it, making sure there is space for the checkpoint file, and sending the checkpoint file back to the submitting machine (or the checkpoint server, if one exists). In the event that a machine is reclaimed by its owner, it is the starter that vacates the job from that machine.

2.4.1.3 Limitations of Condor

Condor however has certain limitations. They are

- Only single process jobs are supported, (i.e., the fork(), exec(), and similar calls are not implemented).

- Signals and signal handlers are not supported, (i.e., the signal(), sigvec(), and kill() calls are not implemented).
- Interprocess communication(IPC) calls are not supported, (i.e., the socket(), send(), recv(), and similar calls are not implemented).
- All file operations must be idempotent, read-only and write-only file accesses work correctly, but programs which both read and write the same file may not.
- Each condor job has an associated "checkpoint file" which is approximately the size of the address space of the process. Disk space must be available to store the checkpoint file both on the submitting and executing machines [BLL91].
- Condor does a significant amount of work to prevent security hazards, but some loopholes are known to exist. One problem is that condor user jobs are supposed to do only remote system calls, but this is impossible to guarantee. User programs are limited to running as an ordinary user on the executing machine, but a sufficiently malicious and clever user could still cause problems by making local system calls on the executing machine.
- A different security problem exists for owners of condor jobs who necessarily give remotely running processes access to their own file system.

2.4.2 BATRUN Distributed Processing System

The BATRUN (Batch After Twilight RUNning) project was initiated to tap the resource of un-utilized CPU cycles, particularly during nighttime, for the execution of batch type jobs requiring no interprocessor communication [TKDA96]. The use of modern operating system features such as, multi-threading and remote procedure call, makes BATRUN DPS implementation simple and more flexible than its predecessors.

BATRUN is used to automate the execution of sequential jobs in a cluster of workstations where machines are owned by different groups of users. The objective is to use a general purpose cluster as one virtual computer for batch processing. In contrast to a dedicated cluster, the scheduling in BATRUN must ensure that only the idle cycles are used for distributed computing and that local users, when they are operating, have full control of their machines. BATRUN has two

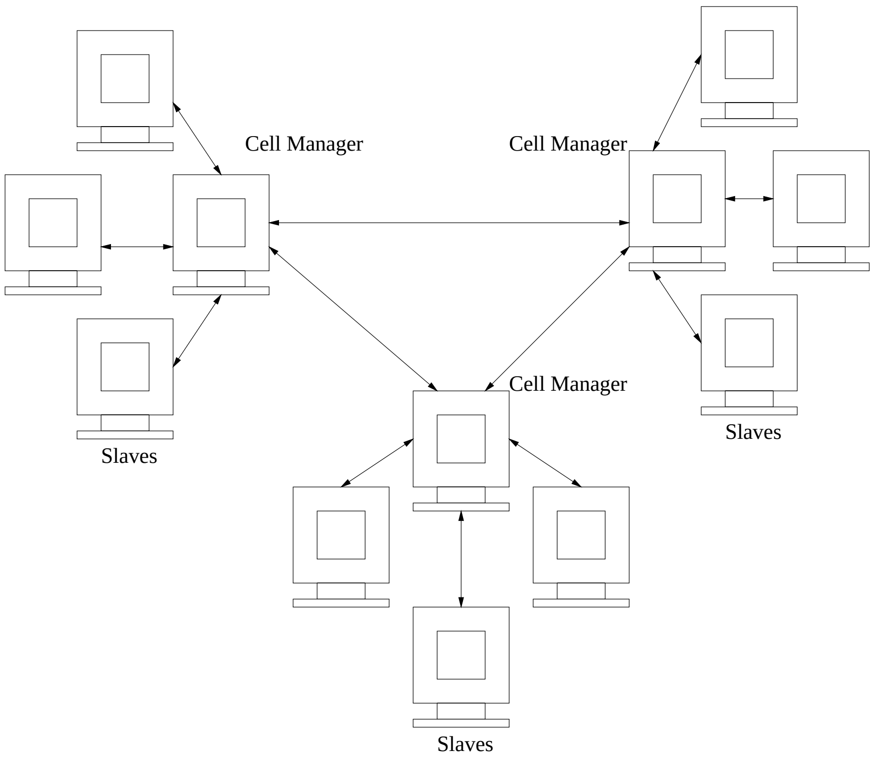

Figure 2.5: BATRUN Architecture

unique features: *ownership based scheduling policy* to ensure top priority for owners of machines, multi-cell distributed design to eliminate single point failure and to support scalability.

2.4.2.1 BATRUN Architecture

In BATRUN the workstation pool is divided into subsets called *cells*. Each cell may further be divided into subsets called *groups*. The cells and groups are formed on the basis of ownership of machines. Each cell consists of a *Cell Manager* and a number of slave machines, as shown in Figure 2.5. A cell and the groups within the cell are configured during the system installation time.

The Cell Manager consists of two daemons: *Machine Allocator* and *Resource Collector*. On each slave machine, there are three daemons: *Job Scheduler*, *Execd* and *Keyboard Monitor(Kbdd)*. The BATRUN components are shown in Fig-

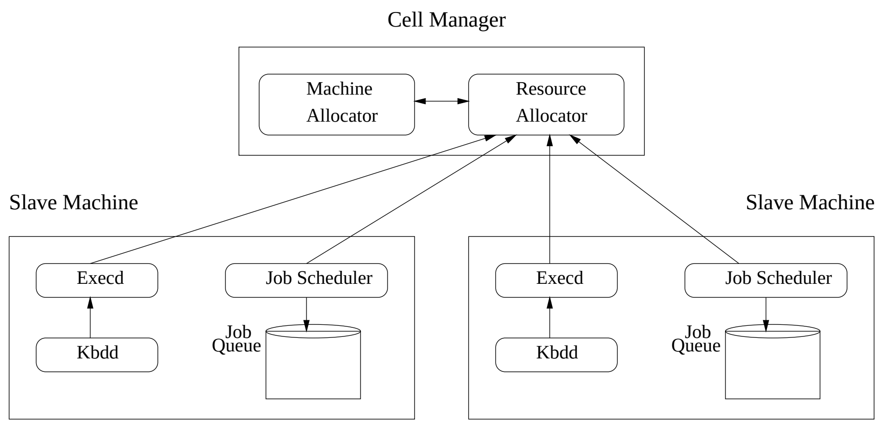

Figure 2.6: BATRUN Components

ure 2.6.

Job Scheduler: The BATRUN Job Scheduler on each slave machine maintains a prioritized queue for jobs submitted through that machine. All the jobs submitted by users at this machine are spooled to the machine's job queue.

Machine Allocator: The Machine Allocator allocates idle slave machines to execute jobs. If all the slave machines in the Machine Allocator's cell are busy or do not have a resource match, it requests slave machines from the Machine Allocator of another cell.

Resource Collector: The Resource Collector is responsible for maintaining a list of all slave machines in a cell. For each machine, this list includes information about available resources and the current state of the machine. This list is sent to the Machine Allocator when it makes a request.

Execd and User Process Agent: The Execd is responsible for reporting the machine status to the Resource Collector. It invokes a User Process Agent (UPA), which is responsible for the execution of the job on the machine. The Execd is also responsible for sending a preemption signal to the UPA. The UPA starts a user process by forking itself. The UPA communicates with the user process using signals when it needs to do periodic checkpointing or when the user job has to be preempted from the local machine. If a job completes or if it is preempted, the UPA notifies the Job Scheduler of the machine where the particular job was submitted. The checkpointing mechanism used by the User Process Agent is adopted from Condor.

Keyboard Monitor (Kbdd): The Kbdd daemon monitors keyboard and mouse activities of the machine and reports the status to the Execd. These activities are monitored to detect the arrival of a local user. The BATRUN jobs are preempted after a fixed time interval if a local user is detected.

2.4.2.2 Job Execution and Related Policies

In the BATRUN environment a user can submit a job from any machine in the pool where he or she has login privileges. The machine where a user submits jobs is called the *submitting machine*. The machine that runs the job is called the *executing machine* which may or may not belong to the same cell as the submitting machine. When a job is submitted it is added to the queue on the submitting machine. The jobs submitted to BATRUN are processed as follows:

1. The Job Scheduler on a submitting machine selects the highest priority job from its local queue.

2. The Job Scheduler request a machine from the Machine Allocator.

3. The Machine Allocator gets information about availability of idle machines from the Resource Collector. There could be several submitting machines that may have jobs to execute. The Machine Allocator goes through a selection procedure to select jobs for execution. If all the slave machines in the Machine Allocator's cell are busy or do not have a resource match, it requests slave machines from the Machine Allocator of another cell.

4. The Machine Allocator assigns a machine to the Job Scheduler on a submitting machine to execute the job.

5. The Job Scheduler then contacts the Execd on an executing machine and requests permission to execute a job. The Execd grants the permission if the machine is still idle and has the required resources. Then the Job Scheduler sends the job information and requests the Execd to execute the job.

6. The Execd creates a User Process Agent (UPA) which will start and monitor the execution of the user job. The UPA informs the Job Scheduler on the submitting machines that the job is started. When the job completes, the UPA on the executing machine reports back to the Job Scheduler on the submitting machine.

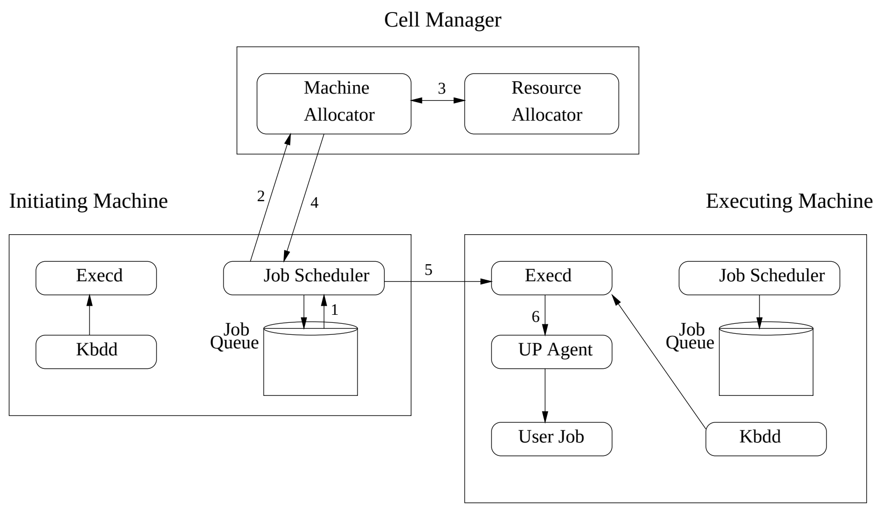

Figure 2.7: BATRUN job execution within one cell

In the preceding discussion, it was assumed that the idle machines with adequate resources are available within the same cell to execute jobs submitted through machines in that cell. This scenario is referred to as *local scheduling* (see Figure 2.7).

The other scenario, called *global scheduling*, deals with the situation when there are jobs to be executed and the Machine Allocator cannot find any slave machines in its cell that have the required resources. In this case, the job execution involves two different cells: the *submitting cell* where the job is submitted and the *executing cell* where the job is executed. In the global scheduling scenario, Step 3 in local scheduling is modified as follows. The Machine Allocator A from the submitting cell chooses an executing cell and requests its Machine Allocator B to commit itself. If Machine Allocator B decides to commit, it passes a list of idle machines to Machine Allocator A. The global scheduling is illustrated in Figure 2.8. The Machine Allocator may need to contact several cells before it finds the Machine Allocator B that is willing to commit.

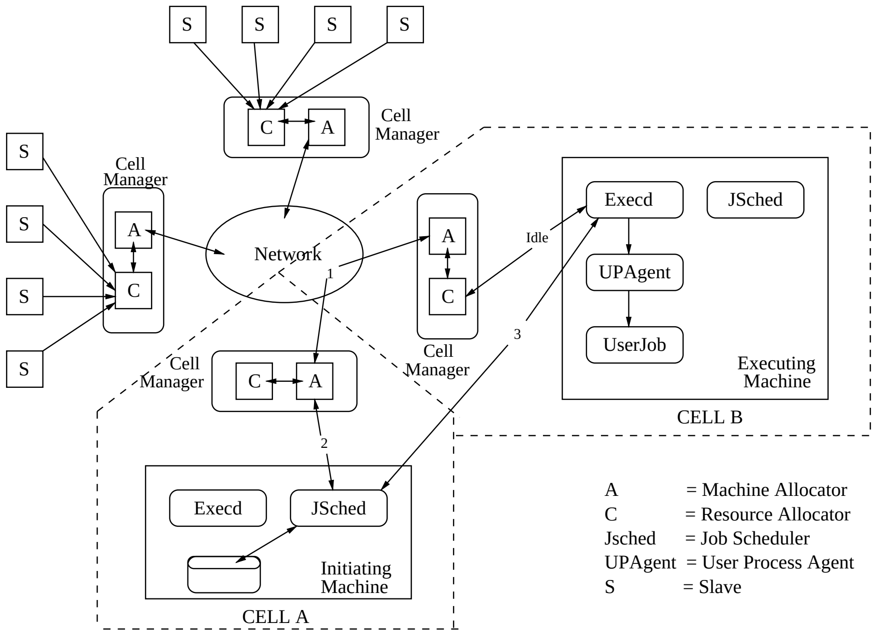

Figure 2.8: BATRUN job execution across two cells

2.5 Other Systems

2.5.1 COmputing in Distributed Networked Environments (CODINE)

CODINE, a merger of the queuing framework of DQS (Distributed Queuing System) and the checkpointing mechanism of Condor, is targeted for optimal utilization of the computer resources in heterogeneous networked environments.

Key features on CODINE are:

- *Resource Management:* CODINE allows an unlimited number of resources to be managed. For example CODINE allows you to manage memory space, disk space, software licenses, tape drives, and others. Each site can easily configure the system according to its own specific resources, thus improving the return of investment on the resources.

- *Distributed Computing in Heterogeneous Networks:* CODINE has been designed to work in a heterogeneous computer environment consisting of hosts with different operating systems as well as hardware/software configurations. CODINE provides the user with a single system image of all the available resources and simplifies the use of the resources.

- *Support for SMP:* CODINE provides special support for shared memory multiple processor machines to make full use of advanced technical possibilities and helps shortening development cycles and time-to-market.
- *Migration of checkpointed jobs:* CODINE supports the migration and restarting of user defined or transparent checkpointing jobs.
- *Job execution environment:* The job execution environment (user ID, umask, resource limits, environment variables, working directory, task exit status, signals, terminal parameters, etc.) is maintained when jobs are sent to execute remotely. This makes the use of CODINE completely transparent to the user.
- *Fault tolerance:* CODINE has no single point of failure and is operational as long as one host of the CODINE cluster is available, by providing a shadow master functionality. The CODINE internal communication protocol is fail-safe and works reliably in noisy networks with heavy traffic.
- *Application Programming Interface (API):* The CODINE API allows users or third party software vendors to develop distributed applications. All of CODINE's internal information is easily available from the CODINE database.
- *Job accounting information and statistics:* Various information on resource utilization is available to allow sites to plan future.

2.5.2 Distributed Queuing System

DQS is designed as a management tool to aid in computational resource distribution across a network. DQS provides architecture transparency for both users and administration across a heterogeneous environment, allowing for seamless interaction of multiple architectures. The goal of DQS is to provide an easy to use tool for the heterogeneous-distributed computing environment that maximizes resource utilization by providing a best fit between resources and job requirements. DQS provides GUIs for both use and administration of the queuing system, including an X-based accounting package. These GUIs further facilitate the maintenance and utilization of the queuing system as a network resource maximizer.

Some of the interesting features of DQS are:

- *Scheduling and Resource Allocation:* There are two methods of scheduling possible. The first is to schedule jobs according to the queue sequence number which means the first queue in the list receives the job for execution. The second method is to schedule by weighted load average within a group so that the least busy node is selected to run the job. The actual method used is selected at compilation time.

- *Job Dispatching:* The master scheduler will compile a list of all the queues that meet the user's resource request. These requests are sorted into two categories. The first is hard resources. All of these resources must be met before the job can begin execution. The second is soft resources. These should be allocated to the job if possible, but the job can still execute without them. The job is then dispatched based on which scheduling algorithm is being used.

- *Fault Tolerance:* A shadow master scheduler is provided to improve redundancy. If the master scheduler goes down, the system is instantly switched over to the shadow. If a machine crashes, the master scheduler will no longer submit jobs to it. When this machine comes back on-line, it will contact the master scheduler and its jobs will be rescheduled.

- *Queue Complexes:* Queue-Complexes are a major advancement in DQS. Queue-complexes are arbitrary resource definitions that once defined, can be associated with queues. These resource definitions can be combinations of available licenses, memory, architecture, available software, etc. Queue-complexes are used by the scheduler at job submission time to determine a best fit between requested and available resources. These resource specifications are completely arbitrary allowing for highly configurable systems.

- *Security:* DQS relies on the security provided by the Unix operating system.

- *Impact on machine owner:* Each machine is set to check for keyboard or mouse activity at the console and suspend any currently running job. The queues on the machine may also be set to restrict the number of jobs that can run on the machine.

- *Scalability:* DQS supports the concept of cells for the purpose of increased scalability. A cell is an instance of a cluster. For a single cell, the scalability of DQS is 300 to 400 machines depending on the power of the machines. Very large sites could potentially encompass several cells. DQS allows users to specify which cell to connect to at job submission time.

- *Parallel Make Utility:* Dmake (Distributed Make) can be used to greatly increase the productivity of workstations, by allowing large projects to be quickly compiled. DQS provides Dmake which allows multiple concurrent compilations. Given a list of N hosts, Dmake sends out M compilations in parallel. Concurrent compilation provides great time savings in the compilation of large projects. Dmake can be used in conjunction with DQS to get it's list of host machines and to takes advantage of DQS's load-balancing features.

- *Interactive Support:* DQS's ability to provide interactive support increases the cost-effectiveness of distributed computing. DQS queues can be configured as interactive, batch, or batch-interactive. The interactive queues, by default, would establish a session with the least loaded server of the specified group[cell] or to a specific queue[cell].

2.6 Types of Jobs in Distributed Systems

Several different types of jobs can be executed in distributed systems. In this section I will be introducing a couple of the types of jobs that will be used to test my system and their applicability in this situation.

2.6.1 Backpropagation

As neural networks are often used to solve problems which are not completely understood or which are hard to solve with the more traditional AI techniques, it is important to know how well a neural network can learn to solve such a problem. One category of problems that can be solved with a neural network, is the category of association problems; each input of the problem space has to be associated with a correct output. These association problems are often solved with so-called backpropagation (BP) networks. A BP network is trained with a set of inputs of the problem space and the correct outputs that are corresponding to these inputs. During the training session, the weights of the network will converge to a point in the network's weight space, in which the problem examples will be known to the network; when the network has reached this point in its state space, it can give the correct output for each example input. However, due to the nature of the network, it can not always learn the problem exactly; the output produced by the network when presented with an input problem, will sometimes only be an approximation of the exact output [Wul94].

2.6.1.1 The Algorithm

The Back Propagation algorithm learns the weights for a multi-layer network, given a network with a fixed set of units and interconnections. It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs. Considering networks with multiple output units, we define E to sum the errors over all of the network output units

$$E(w) \equiv \frac{1}{2}\sum_{d\epsilon D}\sum_{k\epsilon outputs}(t_{kd} - o_{kd})^2 \tag{2.1}$$

where *outputs* is the set of output units in the network, and t_{kd} and o_{kd} are the target and output values associated with the *k*th output unit and training example d.

The learning problem faced by Backpropagation is to search a large hypothesis space defined by all possible weight values for all the units in the network. As in the case of training a single unit, gradient descent can be used to attempt to find a hypothesis to minimize E [Mit97].

The algorithm as described here applies to layered feed-forward networks containing two layers of sigmoid units, with units at each layer connected to all units from the preceding layer. This is the incremental, or stochastic, gradient descent version of Backpropagation.

2.6.1.2 Applicability to the System

The weight update loop in Backpropagation may be iterated thousands of times in a typical application. A variety of termination conditions can be used to halt the procedure. One may choose to halt after a fixed number of iterations through the loop, or once the error on the training examples falls below some threshold, or once the error on a separate validation set of examples meets some criterion. The choice of termination criterion is an important one, because too few iterations can fail to reduce error sufficiently, and too many can lead to over-fitting the training data [Mit97].

Table 2.1: The stochastic gradient descent version of the BACKPROPAGATION algorithm for feedforward networks containing two layers of sigmoid units.

Backpropagation(*training_examples*, η, n_{in}, n_{out}, n_{hidden}**)**

Each training example is a pair of the form ($\vec{x}$, $\vec{t}$), where $\vec{x}$ is the vector of network input values and $\vec{t}$ is the vector of target network output values.

η is the learning rate, n_{in} is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output units.

The input from unit i into unit j is denoted x_{ji}, and the weight from unit i into unit j is denoted w_{ji}.

1. Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.

2. Initialize all network weights to small random numbers.

3. Until the termination condition is met, Do

   - For each ($\vec{x}$, $\vec{t}$) in *training_examples*, Do

     (a) Propagate the input forward through the network: Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

     (b) For each network output unit k, calculate its error term δ_k

     $$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

     (c) For each hidden unit h, calculate its error term δ_h

     $$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}\delta_k$$

     (d) Update each network weight $w_{j,i}$

     $$w_{j,i} \leftarrow w_{j,i} + \Delta w_{j,i}$$

     where

     $$\Delta w_{j,i} = \eta \delta_j x_{j,i}$$

The entire process of training a layered feed forward network using the gradient descent version of Backpropagation depends on a few important parameters. The most important are the number of iterations through the training examples also called as *epochs* and the modified weights of all the connections in the network after each training iteration. When the number of epochs for which a network has to be trained as well as the training set size are huge, the process of training such networks to learn successfully usually takes a long time. When such a process is running on a single machine, it needs a lot of uninterrupted processing time to train the network. It is usually very difficult to get enough uninterrupted processing power to execute the training process without affecting the user of the machine to a very great extent.

However, if such a process runs on a distributed system such as the one I have built, the process can run unattended until completion of execution. The process migrates from one idle machine in the system to another thereby running only on machines which are not being used by their owners. Moreover, the state of execution is saved every time the process migrates from one machine to another. This is done by storing just the parameters that control the training of the network such as the number of epochs completed and the network weights at that stage. The amount of time taken to checkpoint the state of the process as well as the saved checkpoint file is small. Hence, once a training job is submitted to my system, it finds idle machines and gets executed to completion.

2.6.2 Code Breaking

Another testing example that we will use in our system is the code breaking procedure. The driving forces behind encryption policy and technology are served by two opposing functions: code making and code breaking [Ros97]. The term “code making” is used here loosely to refer to the use and development of encryption products. Code making serves several purposes, including protecting proprietary information from corporate and economic espionage, protecting individual privacy, including private communications and personal records, and protecting military and diplomatic secrets from foreign espionage, and information relating to criminal and terrorist investigations from those being investigated.

The term “code breaking” is also used here loosely; in this case to mean acquiring access to the plain-text of encrypted data by some means other than the normal decryption process used by the intended recipient(s) of the data. Code breaking is achieved either by obtaining the decryption key through a special key recovery service or by finding the key through cryptanalysis (e.g., brute force search). It can be employed by the owner of encrypted data when the decryption key has been lost or damaged, or by an adversary or some other person who was never intended to have access [Den97].

When code breaking is done using brute force search, it takes a lot of processing time and power since the number of possible combinations to be tried is very large. This again makes it possible to run in a distributed system such as mine, migrating between idle machines and completing to execution. This process would also run unattended. In this case, the combinations to be examined are split into a number of packets. These packets are examined sequentially for the presence of the code. Therefore as in the previous case, the execution of such a process depends on a few important parameters, such as the number of packets examined and the packet that is being examined. By saving these parameters when the process migrates from one machine to another, the execution of the process can be started from the previously saved state.

2.7 Distributed Objects and CORBA

In this section, I will introduce the concept of distributed objects and how they function in a client-server environment. I will also give a basic introduction to CORBA [Cor], the middleware that is to be used in my system to achieve any form of communication between the clients and the server.

A distributed object is an object that can be accessed remotely. This means that a distributed object can be used like a regular object, but from anywhere on the network. An object is typically considered to encapsulate data and behavior. The location of the distributed object is not critical to the user of the object. A distributed object might provide its user with a set of related capabilities. The application that provides a set of capabilities is often referred to as a service. In our system we will be using this concept of distributed objects for establishing communication and message passing between the individual client machines and the central server.

Applications are developed with distributed objects for a number of reasons:

1. Distributed objects can be used to share information across applications or users.

2. Distributed objects can be used to synchronize activity across several machines.

3. Distributed objects can be used to increase performance associated with a particular task.

4. Distributed objects are a way to distribute computing power across a network of computers, which makes it easier to accommodate unpredictable growth. Centralized approaches frequently fail in such environments.

The Common Object Request Broker Architecture (CORBA) is a standard for distributed objects. CORBA is designed to support the distribution of objects implemented in a variety of programming languages. This is achieved by defining an Interface Definition Language (IDL) that can be mapped to a number of existing languages. IDL is used to define the services offered by a particular distributed object. CORBA defines a wire protocol for making requests to an object and for the object to respond to the application making the request.

In CORBA a server is defined as an Interface. Data passing between the client and the server is defined as IDL structures, sequences, etc. The IDL is compiled with an IDL compiler and the generated code is included within the client and server processes. The server implements a particular interface. The implementation is the distributed object. Clients communicate with the object through an object reference. When an operation is performed on the object reference, network communication occurs, operation parameters are sent to the server, and the actual distributed object executes the operation. It then returns any appropriate data to the client.

CORBA objects are defined as interfaces in IDL. They support operations that take and return simple and complex IDL types. A CORBA object obeys certain rules and which can be accessed via a particular protocol. A CORBA Object is frequently also a distributed object, but it does not have to be. A distributed object is not necessarily a CORBA Object. A distributed object might be a C++ object that can be accessed via a socket, RPC, or telephony. In order for a distributed object to be a CORBA Object, it must be declared in IDL. The object can be implemented in a variety of programming languages.

Some advantages of using CORBA as the middleware are:

1. CORBA supports many existing languages. CORBA also supports mixing these languages within a single distributed application.

2. CORBA supports both Distributed Processing and Object Orientation.

3. CORBA is an industry standard. This creates competition among vendors and ensures that quality implementations exist. The use of the CORBA stan-

dard also provides the developer with a certain degree of portability between implementations.

4. CORBA provides a high degree of interoperability. This insures that distributed objects built on top of different CORBA products can communicate.

CORBA communication is inherently asymmetric. Request messages originate from clients and responses originate from servers. The important thing to realize is that a CORBA server is a CORBA Object and a CORBA client is a CORBA stub. A client application might use object references to request remote service, but the client application might also implement CORBA Objects and be capable of servicing incoming requests. Along the same lines, a server process that implements CORBA Objects might have several object references that it uses to make requests to other CORBA Objects. Those CORBA Objects might reside in client applications. By implementing a CORBA Object within an client application, any process that obtains its object reference can “notify” it by performing an operation on the client-located object [Cor].

CORBA has been used in a variety of industries on real-world systems. The problem domain that CORBA addresses, i.e., distributed computing, is a complex domain. Any real-world distributed computing system is complex, possibly address issues of fault tolerance, availability, transactions, messaging, persistence, performance, and scalability, just to mention a few. CORBA provides infrastructure, services, and tools to develop solutions in this complex domain. We will look more closely at how CORBA helps in implementing the client-server architecture of my system in the next chapter.

Chapter 3

My Method

In this chapter, I will describe the design and the architecture of my system in detail. In addition, I will also sketch the assumptions that were made while building the system and the various issues such as checkpointing, process migration, quiescing of processes and the scheduling of jobs and how they are handled. I will also mention the various parameters on which my system is dependent.

My system is going to focus primarily on the following issues:

1. Hunting for idle workstations.
2. Guaranteeing that the local users are least affected.
3. Making efficient use of idle workstations.
4. Ensuring that jobs are not lost.
5. Providing a fair use of the resources to multiple users.

3.1 Architecture

I have developed a system to automate the execution of large processing jobs in a pool of UNIX workstations. A pool is a cluster of workstations connected via a network that is watched over by one of the machines in the network called the *central server* (see Figure 3.1). This system allows the utilization of otherwise idle CPU cycles in this pool of workstations.

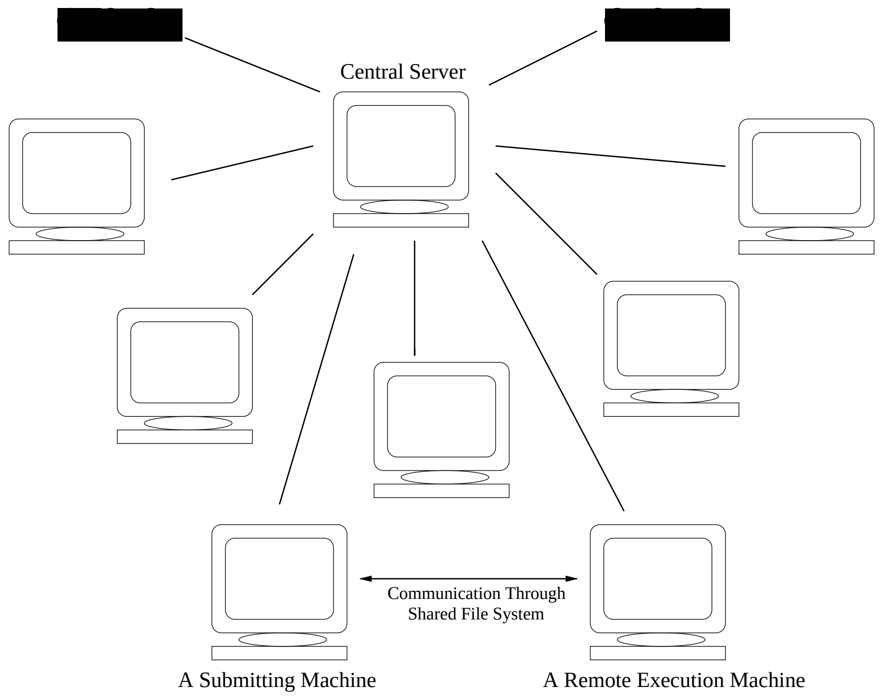

Figure 3.1: Basic System Framework

3.1.1 Basic Components

One of the machines in the pool of workstations is the central server which is responsible for storing and maintaining the priority queue of jobs. All the other machines in the pool are treated as client machines. Jobs that are to be run in this distributed system can be submitted from these machines. The submitted jobs are then passed on by the central server to one of the idle client machines in which it runs.

The central server consists of four daemons: *Job Scheduler*, *Execution Server*, *Machine Allocator* and an *Alarm Sender*. On each client machine, there are two daemons: *Executing Client* and *Load Monitor*. The system components are shown in Figure 3.2.

- **Job Scheduler**
  The Job Scheduler on the central server maintains a prioritized queue for jobs that were submitted to the system through the different client machines.

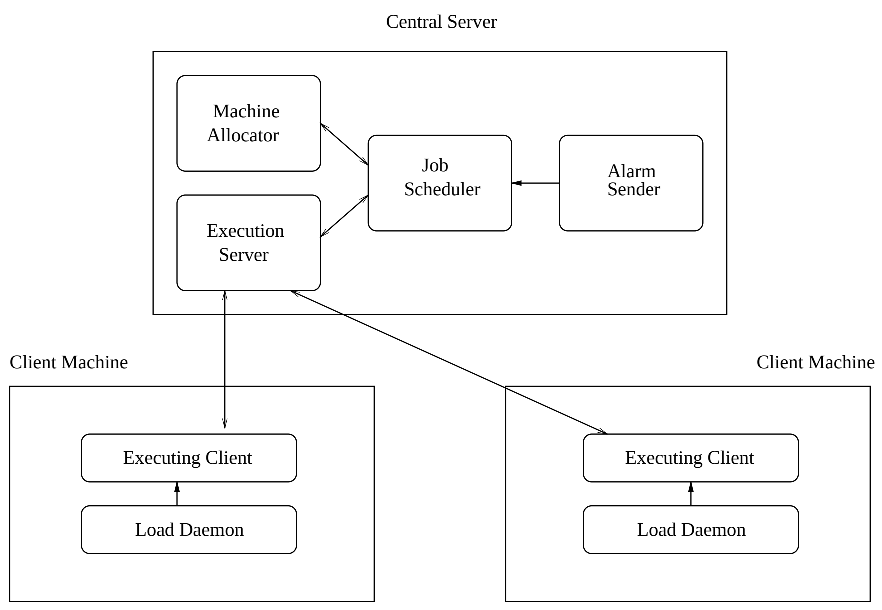

Figure 3.2: System Components

It also holds information about the various users of the system such as the number of active processes they have running on the system and the amount of their CPU utilization in the system as a whole.

- **Execution Server**
  The Execution Server is also run on the central server machine. It is the interface between the Central server machine and the Client machine. The execution server passes the details of the job to execute that it gets from the Job Scheduler to the Client Machine. It also reports the status of the returned job to the Job Scheduler.

- **Machine Allocator**
  The Machine Allocator contains information about the various machines in the system and their state (i.e., whether they are idle or busy). It decides the machine in the pool which is going to run the particular job that has been submitted to the system.

- **Alarm Sender**
  A daemon that runs on the central server and sends an alarm to the Job

Scheduler periodically. Upon receiving the alarm, the Job Scheduler updates the details of all the users of the system. The overall usage of each user is updated and the charges for that period is reset to zero.

- **Executing Client**
  The Executing Client runs on each client machine and reports the state of the machine periodically to the central server. It receives a process that it has to execute from the Job Scheduler through the Execution Server only when they are idle. Upon receiving the process the Executing Client spawns a child to execute the process. When the executing process is either interrupted or executes to completion, the Executing Client reports the status of the job back to the Execution Server.

- **Load Monitor**
  The Load Monitor daemon monitors the load on each client machine and reports the status to the Executing Client. The load on the machine is monitored to detect the arrival of a local user. The distributed job that is running on the machine is preempted if the load is detected to be above a certain threshold. A parameter, provided in the system configuration file, is used to select the load threshold.

3.2 Job Execution

In my system, a user can submit a job from any machine in the pool where he or she has login privileges. The machine in the pool from which the job is submitted is called the submitting machine. The machine that eventually runs the job is called the executing machine. Through its lifetime, a job could run in one or more machines in the pool before executing to completion.

When a job is submitted from a client machine to the Job Scheduler in the central server it is added to the priority queue of jobs maintained by the Job Scheduler. The jobs submitted are processed as follows: (see Figure 3.3)

1. The client in the submitting machine submits either a single job or a batch of jobs to be executed in the system. The job/jobs are submitted to the Job Scheduler in the Central Server.

2. The Job Scheduler requests a machine from the Machine Allocator.

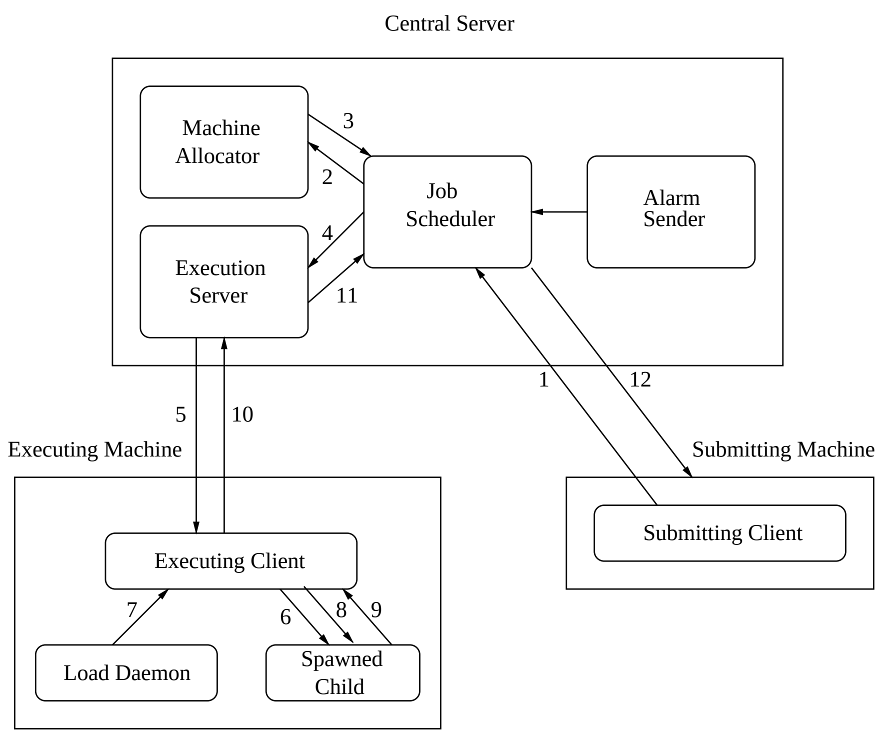

Figure 3.3: Job Execution

3. The Machine Allocator stores the information of the status of machines in the pool. Upon receiving a request from the Job Scheduler, it assigns the idle machine that is at the head of the idle machine queue to execute the job. After allocating that idle machine it then removes it from the idle machine queue. This allocated machine will now be called the executing machine for the job that is to be run.

4. The Job Scheduler selects the highest priority job from its job queue. It then passes the process details as well as the information of the machine in which the job has to be executed to the Executing Server.

5. The Executing Server then contacts the Executing Client in that executing machine and requests permission to execute a job. The Executing Client grants the permission if the machine is still idle. Then the Executing Server sends the job information and requests the Executing Client to execute the job.

6. The Executing Client spawns a child which will start and monitor the execution of the user job. The Executing Client then waits in a loop either for the child to return on completion or till the load on the machine goes above the upper threshold.

7. If the load on the machine goes above a certain threshold, the load daemon indicates that to the Executing Server.

8. When the load on the machine is high, the Executing Server interrupts the child immediately and forces it to checkpoint itself.

9. The child dies either due to completion of execution or because of interruption due to heavy load. The status of the job is extracted by the Executing Client.

10. The Executing Client reports the status of the job to the Executing Server. If the process has been interrupted, then the process details are passed back to the Executing Server.

11. The Executing Server in turn reports the above details to the Job Scheduler. If the job has been interrupted, the Job Scheduler inserts the job back into the Priority Queue after adjusting its priority based on the amount of CPU utilized by it.

12. Finally, if the process has executed to completion, the Job Scheduler passes the information to the submitting machine along with the published results.

3.3 Algorithms Implemented

This section describes the algorithms that are chosen to implement the different features of the system. There are a number of issues that have to be handled in this system such as checkpointing of processes, allocation of machines to processes, and the scheduling of jobs by the Job Scheduler in the central server. We will examine each of these issues in detail and describe the algorithms implemented to handle these issues.

3.3.1 Checkpointing

In simple words, checkpointing involves saving all the work a job has done up until a given point. Normally, when performing large-scale computations, if a machine crashes, or must be rebooted for some reason, all the work that has been done is lost. The job must be restarted from scratch, which can mean days, or even weeks of wasted computation. Checkpointing ensures that you only loose the computation that has been performed since the last checkpoint. A command can be issued to asynchronously checkpoint a job on any given machine, or you can even call a function within your code to perform a checkpoint as it runs. Checkpointing also happens when a job is moved from one machine to another which is known as "process migration" [Dob95].

Traditional distributed systems such as Condor and BATRUN accomplish checkpointing by saving all the information about the state of the job to a file. This includes all the registers currently in use, a complete memory image, and information about all open file descriptors. This file called a "checkpoint file", is written to disk. The file can be quite large, since it holds a complete image of the process virtual memory address space. Such systems therefore have to make sure, before writing to the checkpoint file, that there is enough space to store the checkpoint file in the machine.

In most processes the execution of the job depends on a few important parameters. The execution of the process can be resumed after interruption if the values of these important parameters are known at the time the process was interrupted. The above space constraint for the checkpoint file can be avoided in my system by ensuring that the checkpointing is part of the executing process itself. Every process that is submitted to my system therefore needs to have checkpoint code incorporated in it. This code would be invoked upon receiving an interrupt from the Executing Client and will contain code that saves just the parameters essential to restart the process later from the point where it was interrupted.

As explained earlier, upon receiving the process details from the Central Server, the Executing Client on the idle executing machine spawns a child to execute the process. The load daemon running on the executing machine monitors the load on the machine. When the load daemon detects that the load on the machine goes above a certain threshold because of the user returning to the machine, it forces the Executing Client to send a kill signal to the child that it spawned (i.e., the running job is interrupted). The child catches the interrupt signal and executes the checkpoint code to save the essential state of the process in a file and then dies. The Executing Client then reports the status of the interrupted child to the Job Scheduler which inserts it back into the job queue with a changed priority. Therefore, the requirement of my system is that any job submitted to it has to have its own checkpoint code incorporated in it (i.e., upon receiving an interrupt, it would write a checkpoint file that it understands rather than having to deal with more complex issues such as saving register values).

3.3.2 Machine Allocation

To implement a machine allocation algorithm, the system supports the following states: the REQUESTING state to identify all the machines in the pool that have jobs to be executed, the IDLE state to identify the machines that can be allocated, the BUSY state to identify the machines that are executing the jobs. Note that a machine can be in REQUESTING and IDLE state at the same time.

At any given point in time, there can be several REQUESTING and also several IDLE machines. Note that the Job Scheduler selects one job per REQUESTING machine. The purpose of the machine allocation algorithm is to decide how to allocate the IDLE machines given that there may be several REQUESTING machines.

Implemented Algorithm: There are two separate queues maintained, one for REQUESTING machines and the other for IDLE machines. When a job is submitted from a machine, that machine is added at the end of the REQUESTING queue. As and when REQUESTING machines are allocated to an idle machine to execute the job submitted, they are removed from the front of the REQUESTING queue. Similarly, when a machine from the IDLE queue is allocated to execute a job, it is removed from the IDLE queue. Also, when a machine completes execution of a process and is still idle it is added to the end of the IDLE queue.

3.3.3 Scheduling of Jobs

One of the critical requirements of a scheduler is that it be *fair*. Traditionally, this has been interpreted in the context of processes and has meant that schedulers were designed to share resources fairly between processes [EL68], [L.K70], [B.W68]. More recently, it has become clear that schedulers need to be fair to *users* rather than just processes. This is reflected in work such as [G.J84] and [C.M86].

A fairly typical scheduler was inadequate in our environment for a number of reasons:

1. It gave more of the machine to users with more processes, which meant that users could increase their share of the system simply by submitting more processes to the system.

2. It did not take into account the long-term history of a user's activity. Thus, if a student used the system heavily for approximately two hours, the same

system share was allocated as to a student who had not used the machine for some time.

3. When one user or a set of users have a lot of jobs to run and they submit numerous jobs to the system, the system responds poorly to all other users.

4. If someone needed a good response from the system, it was difficult to ensure that they would get that response without denying all other users access to the system.

In our system, we will try and address these problems by using the *charging* mechanism. Typically, charging systems involve allocation of a budget to each user, and as users consume resources, they are charged for them. We might call this the fixed-budget model, in that each user has a fixed-size budget allotment [JP88]. Then, as the resources are used, the budget is reduced, and when it is empty, the user cannot use the system at all. This fixed-budget model is refined in order to suit our system requirements. Every user has a usage and charge associated with him/her. Every time a process is run by a user and executes to completion or returns to the central server due to interruption, the charges of the user that submitted that user is updated. The usage of a user is updated periodically by adding all the charges accrued by all the processes submitted by that user since the last update. The priority of the individual processes submitted are also adjusted when they return from the executing machine based on the charges they accrued. The charges that my system uses may be defined in terms of the percentage of the CPU used by a process when it is executing on a remote machine.

3.3.3.1 Implementation

As one might expect, conceptually there are two main components, one at the user level and the other at the process level.

3.3.3.1.1 User-Level Scheduling Every user has a *usage*, which is a decayed measure of the work the user has done in the system. The usage of a user has to be updated periodically, so that no user is allowed to dominate the system. The following are the steps to be taken:

- Update usage for each user by adding charges incurred by all their processes since the last update and decaying by an appropriate constant.

- Update accumulated records of resource consumption for planning, monitoring and policy decisions.

The user-level scheduler is invoked every $t1$ seconds. The value of $t1$ defines the granularity of changes in a user's usage as he or she uses the system. Since usage is very large compared to the resources consumed in a second, $t1$ can be of the order of a few seconds without compromising the fairness of the scheduler.

The first component of the user-level scheduler decays each user's usage. This ensures that usages remain bounded and the value of the constant K1 in combination with $t1$ defines the rate of decay. We generally consider the effect of K1 in terms of the half-life of usage. At a conceptual level, this step is performed for all users. In fact, the effect of the calculation is computed as each user logs in, and the actual calculation need only be performed for active users [JP88]. The next part of user-level scheduling involves updating the usage of active users by the charges they have incurred in the last $t1$ seconds and resetting the charges tally.

At this point, the system computes the charges due to a user for the resources the user has consumed during the last cycle of the user-level scheduler. This part of the scheduler does not need to run frequently because usage generally changes slowly.

For each user,

decay usage and update with costs incurred in last t1 seconds:

$$usage_{user} = usage_{user} * K1 + charges_{user} \tag{3.1}$$

reset cost tally:

$$charges_{user} = 0 \tag{3.2}$$

3.3.3.1.2 Process-Level Scheduling The remainder of the scheduler operates at the process level. Each process has a priority, and the smaller its value, the better the scheduling priority. The priority defines the order in which processes are entitled to be allocated CPU resources in idle machines. We also introduce the term *active process* to describe any process that is ready to run which means it is in the job queue held by the central server. Accordingly, there are two types of activities at the process level performed by the scheduler:

1. It schedules or allocates idle machines to the process with the smallest priority, which corresponds to the process being at the head of the queue; and

2. It increases the priority of a process each time it is allocated CPU time in an executing machine. This can be viewed as putting the process further down in the job queue.

Process Activation

When a process returns from an executing machine, *Update costs incurred by the current process*

$$charges_{curr-u} = charges_{curr-u} + cost_{execution} \tag{3.3}$$

Next is the adjustment to the priority of the current process, which defines the resolution of the scheduler. This ensures that the CPU use of the current process increases (worsens) its priority.

Priority Adjustment

When a process returns from an executing machine unfinished, (i.e., it has been interrupted due to heavy load on the executing machine), *Increase the priority of current process in proportion to the user's usage, shares, and number of active processes*

$$priority_{curr-p} = priority_{curr-p} + \frac{usage_{curr-u} * active - processes_{curr-u}}{shares^2_{curr-u}} \tag{3.4}$$

My system pushes the current process down the queue by an amount proportional to the usage and the number of active processes of the process's owner, and inversely proportional to the square of that user's shares. Processes belonging to higher usage (more active) users are pushed further down the queue than processes belonging to lower usage (less active) users. This means that a process belonging to a user with high usage takes longer to drift back up to the front of the queue. (The priority needs longer to decay to the point that it is the lowest).

The basic purposes of such a fair share scheduling system are:

- It is fair to all the users of the system and does not end up favoring users submitting a large number of jobs to the system. It does not allow users to cheat the system;
- Gives a good prediction of the response that a user might expect; and
- Gives meaningful feedback to users on the cost of various services.

3.4 Assumptions Made and Associated Limitations

Both the execution server and the Job scheduling server run on a single machine (the central server) in the cluster. Both these servers are CORBA servers and therefore deadlocks can occur between them. However dead-locks can only occur under certain conditions. First of all, a cyclic or looping relationship must exist between the servers. For example: one server must make invocations on the second server which is in turn implemented to make an invocation upon the first server. This situation is referred to by some as nested call-backs. While cyclic relationship between servers might seem easy to avoid, it often arises, if CORBA is being used to support the combination of client/server request/response and server/client notification. If a cyclic relationship exists and a remote invocation blocks a process, a dead-lock will occur.

The load detecting daemon runs on each machine in the cluster and continuously reports the status of the machine to the central server. If the load daemon on a machine fails for any reason, then the status of the machine is not known to the central server. After waiting for a specific time-interval, the central server makes the machine permanently busy until the load daemon in that machine is corrected to report its status.

Only single process jobs are supported by the system. This means that any job that is to be executed by the system cannot have the fork(), exec() and similar calls present in them. Also all file operations must be idempotent. This means that read-only and write-only file accesses work correctly, but programs which both read and write the same file may not.

The distributed system assumes a shared file system. Hence a different security problem exists for owners of jobs who essentially give remotely running processes access to their own file system. Each job submitted to the system has an associated “checkpoint” file. This file is stored in a common place accessible by all machines in the cluster. Interprocess Communication (IPC) calls are not supported which means that any of the jobs executing cannot implement the socket(), send(), recv() and similar calls.

3.5 Bells and Whistles

There is very minimal interaction between the clients and the server as well as between the clients themselves. The only form of interaction between the clients and the server is in the form of the IDL structure representing the job to be run. The submitting machine passes the structure representing the process to the central server and the job enters the job queue in the server. The central server extracts the highest priority job and passes the structure representing it to one of the idle machines which then becomes the executing machine for that process. Other than that the interaction is only in the form of reporting the status of machines. Such a minimal interaction reduces the network traffic considerably and the throughput of the system increases considerably.

A user can submit one or more jobs from a machine in the cluster. If a user wants to submit a number of jobs to the system, he/she can do so through writing the details of the job/jobs to a file. All the jobs whose details are present in the file are passed on as structures to the central server. My system also provides the capability to remove the jobs from the job queue in the central server. However, only the user who has submitted the job is allowed to remove the job from the queue. This is especially useful, when a user has submitted a single bad job multiple times with different parameters. If the user is not able to remove all these bad jobs from the queue then a lot of the processing time of the individual machines would be wasted in executing these bad jobs.

Chapter 4

Experiments

In this chapter, we present the results of experiments we performed to test our system. We used two types of test problems: backpropagation processes and code-breaking processes. We contrast our results with similar processes that run on the Condor High Throughput Computing System.

4.1 Problems Used in Testing

Our system is built for processes that have large amounts of computation and hence require a large amount of processing power in order to execute to completion. My system provides such a computational environment with large amounts of computational power over a long period of time which is essential to solve such problems.

Another requirement of my system is that the user who submits a process to the system has to write code that is already capable of checkpointing. The user includes the checkpointing code himself. The process on interruption executes that checkpointing code and migrates from that machine back to the server. The main purpose of user-initiated checkpointing is to avoid saving all the information about the state of a job, such as the registers currently in use, a complete memory image, and information about all file descriptors, to a file. Instead, the checkpointing code in each process would save just the information essential to restart the process from the previously interrupted state. One important example of such a process would be a neural network training problem using the backpropagation algorithm. The training at any stage is dependent on only a few parameters such as the number of epochs that the network has been trained until then as well as the weights of the connections between the nodes at the interrupted stage. Saving just these parameters is enough to restart the process from the previously interrupted state.

If the standard method of checkpointing was used (i.e., dumping a core image with all information about the state of a process), then the size of the checkpoint file ends up being quite large. The file is large because it holds a complete image of the process's virtual memory address space. In fact some existing systems such as Condor have to check that there is enough space to store the checkpoint file every time they write the checkpoint file to the checkpoint server.

Another kind of problem which would be tested in our system is the code-breaking problem. For this, the client provided by *distributed.net* was down-loaded which uses a trial and error method to examine numerous codes before arriving at the right one. Here again, the amount of computational power required to examine all the codes is extremely high and it requires long uninterrupted amounts of computational power to do so.

4.2 Basic Performance Tests

A number of tests were performed to verify that the system was working as expected. Tests were run to check that running a process on a single processor as well as running it over a number of workstations in my system yielded the same results. Tests were also performed to verify the interruption of executing processes on machines when the machines are no longer idle (i.e., when the load on them becomes heavy). Experiments were also done to prove that the average time of execution of a process is much better in my system than running it in a single processor sequentially.

4.2.1 Correctness of the System

In order to test that the system was working as it was expected to, backpropagation tests were run on it. Different variations of a command file were given as input to the backpropagation code and the results (i.e., the training set correctness as well as the test set correctness of the network) were obtained both when it was run on a single processor as well as when it was submitted to my system. Table 4.1 illustrates these results.

The goodness or the correctness of my system can be defined as follows:

Table 4.1: Results of backpropagation tests when submitted to my system and when executed on a single processor. Tests were run on different command files and their correctness of prediction on the training and testing set were calculated both in my system when the process gets interrupted on heavy load and outside of it where it runs on a single processor uninterrupted.

Command Files	Correctness with Interruptions		Correctness on single processor	
	Training Set	Test Set	Training Set	Test Set
let(7 epochs)	0.730	0.705	0.730	0.705
let(8 epochs)	0.742	0.711	0.742	0.711
let(10 epochs)	0.758	0.709	0.758	0.709
let(12 epochs)	0.769	0.749	0.769	0.749
let(18 epochs)	0.787	0.768	0.787	0.768
let(20 epochs)	0.791	0.766	0.791	0.766
let(22 epochs)	0.796	0.754	0.796	0.754
let(24 epochs)	0.798	0.776	0.798	0.776

$$correctness = \frac{Number\ of\ Tests\ with\ Same\ Result}{Total\ Number\ of\ Tests\ Performed} \tag{4.1}$$

From Table 4.1 we can find that the results obtained by executing a process on a single processor as well as when the process is submitted to the system and executes on multiple processors (idle machines) are the same every time. This effectively means that the correctness of our system is 1. The same tests were run using Condor and they yielded the exact same results.

4.2.2 Interruption of Processes and their Quiesce Time

Tests were performed to indicate that processes running on idle machines did migrate back to the server when the load on the machine went above a certain threshold. In other words, when the load on an executing machine goes above the upper threshold defined by the system, the process running on that machine would receive an interrupt signal from the client which actually spawned the child process in the executing machine. The time taken for a process to quiesce itself and checkpoint was recorded for a number of backpropagation tests as shown in Table 4.2.

Table 4.2: Time taken by different backpropagation processes to quiesce themselves upon interruption. The processes are interrupted many times due to heavy load on executing machines and the time taken to come out of the machine was recorded after each interrupt.

Command-File	Quiesce Time (sec) Interrupt1	Quiesce Time (sec) Interrupt2	Quiesce Time (sec) Interrupt3	Quiesce Time (sec) Interrupt4	Quiesce Time (sec) Interrupt5
let(7 epochs)	40	15	–	–	–
let(8 epochs)	8	–	–	–	–
let(10 epochs)	24	27	–	–	–
let(12 epochs)	25	8	8	55	20
let(18 epochs)	33	39	13	6	9
let(20 epochs)	35	42	45	5	4
let(22 epochs)	58	37	9	7	55
let(24 epochs)	19	61	52	4	2

The time taken to do so should ideally be as little as possible, since the process is supposed to evacuate the machine as soon as the machine is indicated to have a heavy load.

For the code-breaking tests, the quiesce time was less than 1 second for all interruptions. This is because, the code-breaking processes were designed to perform periodic-checkpointing. Hence the process on interruption does not execute any special code but just kills itself. The next time the process is restarted it resumes from the previously checkpointed state.

4.2.3 Checkpointing Results

When a process is interrupted while execution, the interrupt handler in the process executes the code to write the essential information to a checkpoint file. The size of the checkpoint file written depends on the number of parameters whose values have to be saved in order to restart the process from the interrupted state. In case of the backpropagation, it is sufficient to store the number of epochs that the network has been trained until then as well as the weights of the connections in the network at that stage. Table 4.3 shows the average size of a checkpoint

Table 4.3: Size of the checkpoint files written by various backpropagation processes during each run

Command-File	Number of Interruptions	Average Size of Checkpoint File (Bytes)
let(7 epochs)	2	21840.5
let(8 epochs)	1	21811
let(10 epochs)	2	21834.5
let(12 epochs)	7	21847.5
let(18 epochs)	5	21888.2
let(20 epochs)	11	21878.3
let(22 epochs)	10	21873.3
let(24 epochs)	11	21876.1

file created for various backpropagation tests. The different backpropagation tests were performed on the same command file with the number of epochs that the network is trained being different every time.

The same checkpointing tests were performed on Condor. For the same command files Condor yielded a checkpoint file of size 7.31 MegaBytes on an average. The maximum size of a checkpoint file was 10.20 MegaBytes. In contrast, our system yields a checkpoint file of size 21.85 KiloBytes on an average. This tells us the effect user-initiated checkpointing has in drastically reducing the size of a checkpoint image created.

4.2.4 Events on a Workstation

The various events that happen on a machine which is a part of my system is based on the load on the machine. When the load on a machine goes below the lower threshold and stays under it for a certain period of time, it is viewed as idle. The machine then gets the top priority process from the server and starts executing it. The machines continues running the process either until completion or until the load on the machine goes above the upper threshold. At that point, the process evacuates the machine after writing to a checkpoint file. When the server does not have any more jobs to run, the client idle machines wait in a loop till a process is submitted to the server.

All the machines in the system execute the above mentioned cycle. Table 4.4

Table 4.4: The significant types of events happening on machines in the system based on load :- (a) The machine is idle (i.e., load on machine < 1.0) (b) Machine gets the top priority process from the server when it has been idle for 15 minutes or more (c) Process writes the current state to a checkpoint file and evacuates the machine, when the machine is no longer idle (i.e., load on machine > 2.3)

Time (min)	Load ”moe”	State of Processor ”moe”	Time (min)	Load ”curly”	State of Processor ”curly”
0	0.43	(a) idle	0	0.089	(a) idle
5	0.71	idle	5	0.094	idle
10	0.92	idle	10	0.092	idle
15	0.85	(b) idle - gets process	15	0.123	(b) idle - gets process
20	1.59	Process Running	20	2.06	Process Running
25	2.17	Process Running	25	2.21	Process Running
26	2.28	(c) Process Evacuates	30	2.22	Process Running
28	1.90	Process Migrated	33	2.26	Process Running
32	1.55	Load Decreasing	34	2.28	(c) Process Evacuates
35	1.36	Load Decreasing	35	2.13	Process Migrated
40	1.10	Load Decreasing	40	0.85	(a) Load Decreasing - idle
45	0.97	(a) idle	45	0.016	idle
50	0.84	idle	50	0.036	idle
55	0.84	(b) idle - gets process	52	0.038	(b) idle - gets process
59	1.48	Process Running	59	1.78	Process Running
65	1.90	Process Running	65	2.09	Process Running
70	2.25	(c) Process Evacuates	70	2.14	Process Running
71	2.19	Process Migrated	74	2.29	(c) Process Evacuates
75	1.28	Load Decreasing	75	1.93	Process Migrated
80	0.81	idle	80	1.33	Load Decreasing

shows this cycle of events on a couple of machines ”curly” and ”moe” which were part of my system. The upper and lower load threshold of the system were set to 2.30 and 1.0 respectively for these results.

Table 4.4 is also plotted as a graph of the load on various machines versus time (Graph 4.1). The graph highlights the various events that take place on the machines in the system.

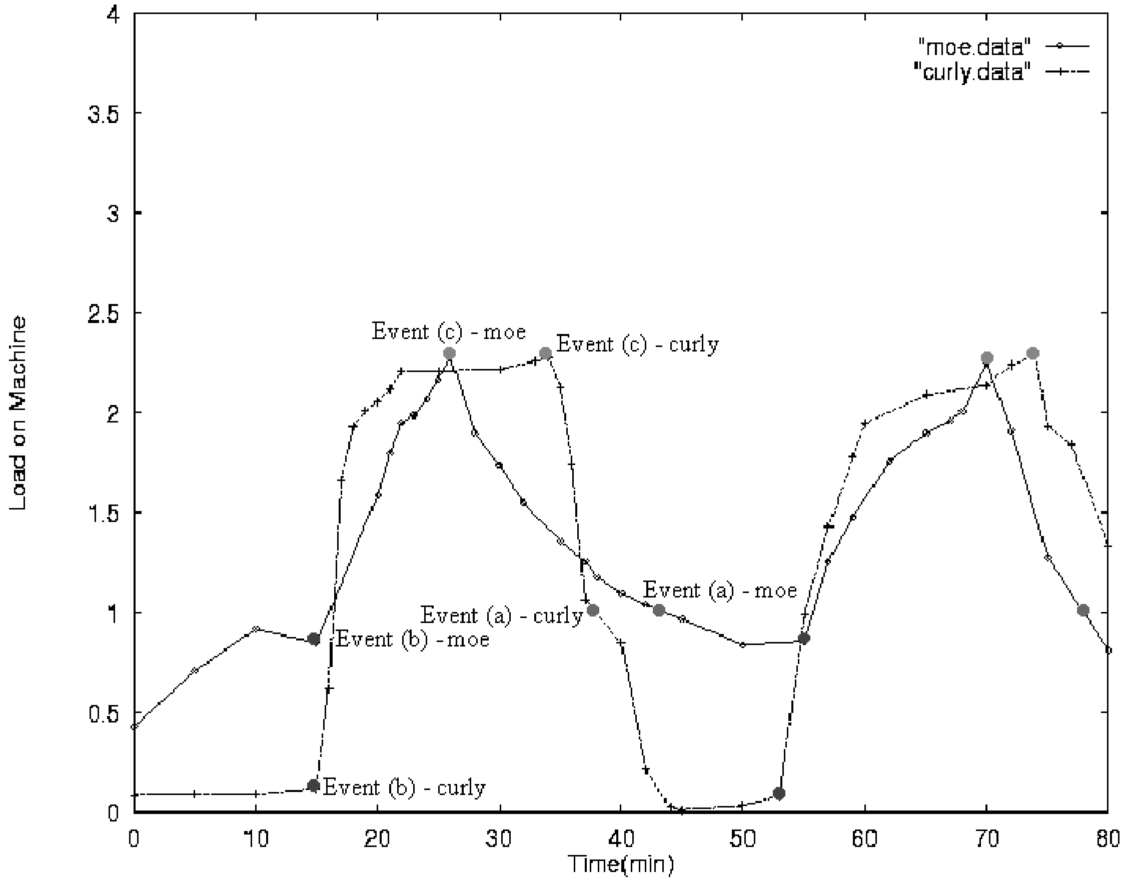

Figure 4.1: The Load on a machine vs Time : Graph highlighting the various events that take place on the machines in the system

4.3 Discussion of Results

Table 4.1 shows that the results obtained by executing a process on a single processor as well as when the process is submitted to the system and executes on multiple idle machines are the same. This clearly indicates that the system works exactly as it was expected to in terms of producing identical results with that of uninterrupted execution of processes. The execution of the same processes in the Condor system also yielded the same results. Clearly from the results tabulated in Table 4.1 we can see that the accuracy of my system is very good and is comparable with the Condor system.

The time taken by different backpropagation processes to quiesce themselves upon interruption were recorded in Table 4.2. The processes were interrupted many times due to heavy load on machines in which they were running and the time taken to come out of the machine was recorded after each such interruption. The quiescing time for the same processes on interruptions were recorded for Condor and it was noted to be over 1 minute on an average. On the other hand the quiescing time for almost all interruptions in our system was less than 1 minute. The quiescing time in effect depends on the amount of checkpointing code written and in Condor and other such systems, which examine the core file dumped by a process to save its state, the time taken to extract all information from the core image dumped is high.

The next issue that we dealt with while recording results was the size of the checkpoint images created by our system. Table 4.3 shows the average size of the checkpoint files written by our system during each run of various backpropagation tests. The average size of the checkpoint file created by Condor was also recorded for the same tests. As mentioned in Section 4.2.3 the size of the Condor checkpoint file on an average was 7.31 MegaBytes whereas the average size of the checkpoint file created by our system was only 21.85 KiloBytes. This clearly indicates that the size of the checkpoint file created is far less than that of Condor. This is primarily because Condor saves the complete image of the process's virtual memory address space in the checkpoint file whereas our system saves only the parameters essential for restarting the process from the saved state. This method that we have adopted to checkpoint migrating processes allows us to overcome the limitation of Condor in creating huge checkpoint memory images.

Finally, in order to test the involvement of individual workstations which are part of the cluster, we monitored the load on the various machines in the cluster and recorded the various events that happen in a machine in the system with respect to the load on it. This test was performed on a couple of machines in the system and significant events such as the machine being idle, a process moving into the machine from the server and the quiescing of a process when the load overshoots the upper threshold defined in the system, were identified.

All the results recorded above give us further evidence that the different features of our system such as the load monitoring code, the quiescing of processes and the checkpointing of migrating processes with small checkpoint images are all working correctly.

Chapter 5

Conclusions

In this thesis I implemented a new distributed processing system to satisfy large-scale computational needs by using idle cycles in a network of workstations. I evaluated the results obtained from my system and then compared its performance with Condor in terms of the throughput of jobs, size of the checkpoint file created and the quiescing time of processes on individual machines. I believe that my system has demonstrated that it is a viable method for addressing the computing needs of users.

Experiments performed on the system indicate that the questions put forth in Chapter 1 have been answered. In the preliminary tests, I first determined the basic correctness of the system. Tests were performed to check if my system yielded the same results for a process as that when it was run on a single machine uninterrupted. Tests were also performed to verify the interruption of processes on machines when the machines are no longer idle. These results confirmed the correctness of our system.

The main contribution of this thesis is to investigate alternatives to the idea of storing large checkpoint files in memory when processes running on individual machines are interrupted as experienced in currently existing systems. My thesis proposes user-initiated checkpointing instead of saving the entire virtual address space of the process.

The questions from Chapter 1 that provided the motivation for this thesis are restated here along with the conclusions that can be drawn from answering them in light of the experiments conducted:

Question 1: Can a parallel system be built that can manage a collection of distributively owned workstations having the potential to make use of idle time on various workstations to run large processes across the network but still ensuring top priority for owners of the machines?

From the results of the initial correctness experiments we see that the system that I have built produces the exact same results for processes as when it is executed sequentially in a single processor. The system makes use of only the idle time on the various workstations in the cluster to execute the submitted jobs, thus giving highest priority to owners of workstations. It ensures this by setting an upper load threshold on machines and forcing processes to migrate from a machine as soon as the load on the machine goes above the threshold.

Question 2: Can these large processes be run across the network without creating large memory images (saved states of the processes) like most other systems do every time a process is interrupted?

In order to overcome this inherent problem with the standard checkpointing method adopted by existing systems, we propose a new technique of checkpointing with user-initiated checkpointing. Here, instead of using the core file dumped by the process on interruption to recover the state of the process, the user is asked to insert the checkpointing code in the process himself. This strategy is particularly useful for those processes that are dependent on a few essential parameters for their execution. Upon receiving an interrupt the process executes the checkpointing code and writes the essential parameters to a file. This file is much smaller than that created with the information extracted from the core dump. The results shown in Section 4.2.3 show that the sizes of the checkpoint images created in our system are much smaller than those created in Condor for the same processes.

Question 3: Would jobs involving such large scale computations perform better in such a system or not?

The results in section 4.2.1 indicate again that jobs involving large scale computations such as the backpropagation processes produce the same results in my system as they would outside of it. The advantage in using my system would be when a batch of such jobs have to be executed. Such a batch of jobs can be submitted from a client machine and the system in turn queues the jobs in the server and uses the idle time present on the individual machines to execute a number of these jobs at the same time in different idle machines. This in turn improves the throughput of the system and is much better than having the whole set of processes executing sequentially on a single machine.

The principle conclusion of this thesis is that in a cluster of workstations it is possible to use only the idle cycles for executing processes involving large-scale computations. This can be done without storing the entire memory image of the process as is done in most existing systems. Instead, the suggested method of user-initiated checkpointing to restart the process from its saved state, proves vital in drastically reducing the size of the checkpoint image.

5.1 Future Work

Preliminary testing of the scheduling policy which provides a fair share to all users of the system, yielded results that were expected. But the different features of the scheduling strategy that were implemented such as the user-level scheduling, the process-level scheduling and the decay of users' shares as they use the system, have not been tested rigorously and there is need for a more detailed analysis of the scheduling policy. The scalability of the system has also not been tested rigorously. Most of the experiments that I performed were on a small cluster of 5 workstations and all the features that I have implemented worked correctly in such a cluster. The scalability of my mechanism cluster requires further testing and analysis.

The system assumes the presence of a single central server which runs the job scheduler and the executing server daemons. If this central server goes down, then the whole system goes down. Mechanisms to cope with the loss of the central server have to be devised. One possible solution would be to start a shadow process which would monitor the central server. As soon as the central server goes down, the shadow process spawns a new master process and the entire control is transferred to that new process. The feasibility of such a solution has to be analyzed as well. In addition to this, further testing has to be done to measure the capability of the system to deal with missing slaves.

At this point, the system assumes a homogeneous collection of workstations (i.e., all the machines in the cluster are SUN Solaris workstations with UNIX). Further attempts can be made at making the system heterogeneous to support workstations in different environments. Moreover, the requirement that the user checkpoints his/her own code might make it too complex for some processes. Simple mechanisms can be identified to automate the checkpointing to an extent such as the user specifying the important parameters of the process to be saved. The values of these parameters are then automatically saved every time the process is interrupted. Finally, the system also assumes that all the workstations in the cluster have the capability to provide the necessary resources for all submitted jobs. However, this might not be true in some complex cases. These cases have to be analyzed and the idle machine resources have to be checked before processes are sent to them.

Bibliography

[AES97] A. Acharya, G. Edijlali, and J. Saltz. The Utility of Exploiting Idle Workstations for Parallel Computation. *SIGMETRICS '97*, pages pp. 225–236, May, 1997.

[AKPtNt95] Thomas E. Anderson, David E. Keller, David A. Patterson, and the NOW team. A Case for NOW (Networks of Workstations). Technical report, University of California, Berkeley, February 1995.

[AL93] Joseph A.Kaplan and Michael L.Nelson. A Comparison of Queueing, Cluster and Distributed Computing Systems. Technical report, NASA Langley Research Center, October, 1993.

[BLL91] Allan Bricker, Michael Litzkow, and Miron Livny. Condor Technical Summary. Technical report, Computer Sciences Department, University of Wisconsin – Madison, 1991.

[Bur] Ted Burghart. Distributed Computing Overview. http://www.quoininc.com/quoininc/dist_comp.html.

[B.W68] B.W.Lampson. A Scheduling philosophy for Multiprocessor Systems. In *Communications of the ACM*, volume 11, pages pp. 347–360, 1968.

[CGM93] Nicholas Carriero, David Gelernter, and Tim Mattson. Experience with the Linda Coordination Language and its Environment. Technical report, Yale University - Department of Computer Science, April 1993.

[C.M86] C.M.Woodside. Controllability of Computer Performance Tradeoffs Obtained using Controlled-Share Queue Schedulers. In *IEEE Transaction Software Eng.*, volume 10, pages pp. 1041–1048, Oct, 1986.

[Com96] Platform Computing. LSF MultiCluster: Software for Global Load Sharing. December 1996.

[Cor] The CORBA FAQ. http://www.aurora-tech.com/corba-faq.

[Den97] Dorothy E. Denning. Encryption Policy and Market Trends. http://www.cs.georgetown.edu/ denning/crypto/Trends.html, 1997.

[DGP96] Dennis W. Duke, Thomas P. Green, and Joseph L. Pasko. Research Toward a Heterogeneous Networked Computing Cluster: The Distributed Queuing System. Technical report, Florida State University, Tallahassee, Florida, 1996.

[Dob95] Dr. Dobbs. Checkpointing and Migration of UNIX Processes in the Condor Distributed Processing System. *Dr. Dobbs Journal*, 1995.

[EL68] E.G.Coffman and L.Kleinrock. Computer Scheduling Methods and Their Countermeasures. In *In Proceedings of the Spring Joint Computer Conference*, volume 32, pages pp. 11–21, 1968.

[F93] Ferstl. F. CODINE Technical Overview. Technical report, Genias Software, April 1993.

[Fie93] Scott Fields. Hunting for Wasted Computing Power. *Research Sampler*, 1993.

[GBD$^+$94] Al Geist, Adam Beguelina, Jack Dongarra, Weicheng Jiang Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing*. 1994.

[G.J84] G.J.Henry. The Fair Share Scheduler. *Bell Systems Technical Journal*, 63:pp. 1845 – 1857, Oct, 1984.

[JK91] J.Flower and A. Kolawa. Parallel Programming with EXPRESS. Technical report, ParaSoft Corporation, 1991.

[JP88] J.Kay and P.Lauder. A Fair Share Scheduler. In *Communications of The ACM*, volume 31, pages pp. 44–55, Jan, 1988.

[L.K70] L.Kleinrock. A Continuum of Time-Sharing Scheduling Algorithms. In *In Proceedings of the Spring Joint Computer Conference*, volume 36, pages pp. 453–458, 1970.

[LL90] Mike Litzkow and Miron Livny. Experience With The Condor Distributed Batch System. In *IEEE Workshop on Experimental Distributed Systems*, 1990.

[LLM98] M. Litzkow, M. Livny, and M.W. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages pp. 104–111, 1998.

[LM93] M. Livny and M.W. Mutka. The Available Capacity of a Privately Owned Workstation Environment. *Performance Evaluation*, vol. 12, no.4:269–284, 1993.

[LS92] Michael Litzkow and Marvin Solomon. Supporting Checkpointing and Process Migration Outside the UNIX Kernel. In *Proceedings of the Usenix Winter Conference*, 1992.

[Mit97] Tom M. Mitchell. *Machine Learning*. 1997.

[Ove] Overview of The Condor High Throughput Computing System. http://www.cs.wisc.edu/condor/overview/.

[RH95] Kyung Dong Ryu and Jeffrey K. Hollingsworth. Linger Longer: Fine-Grain Cycle Stealing for Networks of Workstations. *SC98, Orlando, Florida*, June, 1995.

[Ros97] Shawn J. Rosenheim. *The Cryptographic Imagination*. The Johns Hopkins Univ. Press, 1997.

[TKDA96] Fredy Tandiary, Suraj C. Kothari, Ashish Dixit, and E. Walter Anderson. BATRUN Distributed Processing System(DPS): Utilizing Idle Workstations for Large-scale Computing. In *IEEE Parallel and Distributed Technology*, volume 4, Summer 1996.

[Wul94] Alex Wulms. On the Dynamic Behavior of Back Propagation Networks. Project Study on Neural Networks, 1994.

[WZAL93] Jingwen Wang, Songnian Zhou, Khalid Ahmed, and Weihong Long. LSBATCH: A Distributed Load Sharing BAtch System. Technical report, Computer Systems Research Institute, University of Toronto, 6 King's College Road, Toronto, Ontario, Canada, MFS 1A1, April 1993.