Many different types of distributed batch scheduling systems have been developed in the last decade to take advantage of the decentralization of computers and the enormous investments that many companies and educational institutions have in desktop workstations. Based on the premise that the majority of desktop workstations are significantly underutilized, distributed batch systems allow users to submit and run jobs when these workstations are available. While simpler systems determine machine availability by time of day (e.g., 5:00 p.m. to 8:00 a.m.), more sophisticated systems determine availability dynamically, migrating tasks when the availability changes.

RingLeader is a distributed batch system currently under development at Hewlett Packard. Since meeting the objectives of a distributed system rely on the intelligent use of idle workstations, good resource determination and efficient utilization decisions are a high priority for such a system. System performance will depend heavily on the process of deciding where jobs should be run. This thesis explains the development of RingLeader's history based resource utilization scheme, and compares its performance to more simplistic algorithms.
Optimization of Machine Allocation in RingLeader

by

Jonathan B. King

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 6, 1996
Commencement June 1997
Master of Science thesis of Jonathan B. King presented on December 6, 1996.

APPROVED:

Redacted for Privacy

Major Professor, representing Computer Science

Redacted for Privacy

Head or Chair of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Jonathan B. King, Author
ACKNOWLEDGMENT

This project would not have been accomplished without help and encouragement from many people. In particular, Bob Clark and David Stuve from the Hewlett Packard company were invaluable. Bob was always willing to talk over an idea and provide insightful comments, sometimes bringing me back to reality. David Stuve has been a co-author of much of the RingLeader system; without his help RingLeader would not be working today. Thanks to Oregon State University, Dr. Saletore and Dr. Quinn for working with my somewhat unique situation. I hope that many further cooperative partnerships are formed between Hewlett Packard and Oregon State University in the years to come.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Related Work</td>
<td>3</td>
</tr>
<tr>
<td>Overview of RingLeader</td>
<td>7</td>
</tr>
<tr>
<td>RingLeader Basics</td>
<td>7</td>
</tr>
<tr>
<td>The Monitor</td>
<td>9</td>
</tr>
<tr>
<td>Determining Host Availability</td>
<td>10</td>
</tr>
<tr>
<td>Pausing and Resuming Jobs</td>
<td>12</td>
</tr>
<tr>
<td>Responding to MCP Commands</td>
<td>13</td>
</tr>
<tr>
<td>The Master Control Program</td>
<td>14</td>
</tr>
<tr>
<td>System Setup</td>
<td>14</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>15</td>
</tr>
<tr>
<td>User Response</td>
<td>17</td>
</tr>
<tr>
<td>The Job Submission Tool</td>
<td>17</td>
</tr>
<tr>
<td>The Gatekeeper</td>
<td>18</td>
</tr>
<tr>
<td>The Command Shell</td>
<td>19</td>
</tr>
<tr>
<td>Job Flow in RingLeader</td>
<td>20</td>
</tr>
<tr>
<td>Resource Allocation Design</td>
<td>22</td>
</tr>
<tr>
<td>Design Overview</td>
<td>22</td>
</tr>
<tr>
<td>Static Machine Information</td>
<td>23</td>
</tr>
<tr>
<td>Dynamic Machine Information</td>
<td>23</td>
</tr>
<tr>
<td>Job Information</td>
<td>25</td>
</tr>
<tr>
<td>Selection Algorithm</td>
<td>25</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------</td>
</tr>
<tr>
<td>Algorithm Performance</td>
<td>27</td>
</tr>
<tr>
<td>Experimental Design</td>
<td>27</td>
</tr>
<tr>
<td>Experimental Results</td>
<td>34</td>
</tr>
<tr>
<td>Conclusions</td>
<td>39</td>
</tr>
<tr>
<td>Bibliography</td>
<td>41</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>The RingLeader System</td>
<td>8</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Screen Shot of the Job Submission Tool</td>
<td>20</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Job Flow in RingLeader</td>
<td>21</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Algorithmic Hierarchy</td>
<td>28</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Historical Data Example</td>
<td>28</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Actual Machine Usage Example</td>
<td>30</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Actual Job Submission Rate</td>
<td>32</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Random Job Count Distribution</td>
<td>33</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Job Completion Time Ratio</td>
<td>36</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Job Length Comparison</td>
<td>37</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Job Throughput by Algorithm</td>
<td>38</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1: Distributed Batch Software Products</td>
<td>5</td>
</tr>
<tr>
<td>Table 2: Sample Job Submission Times</td>
<td>31</td>
</tr>
<tr>
<td>Table 3: Sample Simulation Output</td>
<td>34</td>
</tr>
<tr>
<td>Table 4: Average Completion Ratio (Time/Length)</td>
<td>35</td>
</tr>
</tbody>
</table>
DEDICATION

To Shirley
For Whom I Live
Optimization of Machine Allocation in RingLeader

Introduction

The notion of recovering unused computer cycles by distributing batch jobs to unused workstations is not a novel idea. Such systems have been in place since the early 1980’s, and their number has increased with the popularity of personal workstations and the continued decentralization of computing resources. Both commercial and research solutions are available to help balance workstation load and increase job throughput, effectively allowing users with computation needs to utilize colleagues’ unused resources.

Batch schedulers typically provide a single point of submission for running jobs on the system. Some mechanism is used to determine a “good” host machine on which to execute the task, and the job is started on that particular workstation. Some systems are capable of dynamic migration of processes, where jobs may be moved around at the system’s discretion. This means that a process may complete on a workstation different from the one on which it was started. Background and further information on batch schedulers can be found in the Related Work section.

RingLeader is a distributed batch scheduler developed at Hewlett Packard as an internal tool for increasing computer simulation and modeling throughput. It is tailored to an engineering environment in which literally hundreds of workstations across the site are available for computation. Typically, many of these workstations are available at any given time, and RingLeader makes these resources available to modeling engineers to run various applications, primarily computational fluid dynamics simulations. A brief introduction to the RingLeader system is given in Chapter 3, Overview of RingLeader.
Since each workstation in a distributed system normally has a single owner or primary user, accommodating enhanced CPU utilization typically means allowing the owner priority over their workstation, meaning that some method of allowing the owner to take control must take place whenever the workstation ceases to be available.

RingLeader uses UNIX signals to provide pause/resume capability for batch jobs, while also utilizing the nice facility. Since the premise of “polite” execution means that the process proceeds in a potentially non-continuous fashion, it is impossible to predict the final run time of a given job. At the same time, it is desirable to minimize the length of time that jobs run on the distributed system. Deciding which host a given job should run on is arguably the most important factor affecting job completion time. A decision to place a job on a frequently used or particularly slow workstation will result in increased time to result, and decreased throughput of jobs on the system. At the same time, it is not easy to determine when hosts are going to be available. A system available in one moment may not be the most desirable system in a few minutes. Also, making assumptions about times of availability (such as 6:00 p.m. to 7:00 a.m.) means that much of each day results in unused computers.

This thesis describes the development of the host selection algorithm to be used in the RingLeader system and presents a simulation which was run to determine the effectiveness of the algorithm. The algorithm is history based and attempts to determine the best machine selection based on past characteristics of machine usage.

---

1 The UNIX nice command allows a process to lower its priority relative to other processes on the system. The command is typically used to allow background processing to occur while minimizing the intrusiveness of the process on other, higher priority jobs. On a UNIX workstation, type “man nice” for more information.
Related Work

Work in the realm of distributed or cluster computing has a long history, and RingLeader is by no means the most advanced product on the market today. However, it is the selection of which machine on which to execute a job that most differentiates RingLeader from other programs. Since it would be impossible to cover each of these programs in depth, I will give a brief overview of the types of systems known to exist, and describe the important features of a few better-known systems.

The search for efficient use of distributed resources can be divided into three categories: distributed operating systems, cluster computing environments, and distributed batch systems. Distributed operating systems [Mullender 1987, 1993] are built from the ground up and include within the kernel everything required to utilize the capabilities of resources that may be physically separated. Examples of distributed operating systems, such as Amoeba [Tanenbaum et al., 1991], V [Cheriton, 1988], LOCUS [Walker, 1983], and THESIS [McLean, 1994][Kelly, 1994], are typically capable of transparent process migration, automatic load balancing, etc. Although distributed operating systems represent an important advancement to the field, and a valid means of progressing toward true machine independence, they do not fit into the same niche that RingLeader and other distributed batch systems are designed for: taking advantage of an already significant investment in hardware, software, and an environment in which the machines only use a fraction of their full potential.

The second category is cluster computing environments. These programs typically allow the user to build and execute a single application designed to run on multiple workstations in a local or distributed fashion. Such products, like PVM *(Parallel Virtual Machine)* [Geist et al., 1994], Charm [Kale et al., 1996], DOME [Áarbe et al., 1995] and others provide services that support the development, execution, and
maintenance of distributed applications. Although RingLeader does not fit into this category, it was built using the services provided by PVM, as described in more detail in Overview of RingLeader.

RingLeader fits into the third category of distributed computing implementations: distributed batch systems. In a distributed batch system the user writes a "standard" executable program and submits it to the system. The system then determines where, when, and how to run the given application. Although most of these environments support only sequential jobs, some systems also support parallel applications. Supporting parallel applications requires the system to allocate a set of hosts. Although some of the systems support checkpointing and migrating serial programs, none of the current systems support the migration of parallel tasks. According to the "Cluster Computing Review" [Baker, M., 1995], there are approximately 19 such environments, as shown in Table 1.

Of importance in a batch scheduler are features to ensure that jobs complete. Such fault tolerance can be achieved by either a restart mechanism, which will result in lost time, or by implementing some form of checkpointing1, which allows a program's state to be periodically saved for use in a restart. Also, since batch systems are designed to run in an environment where workstations are typically not communally shared, minimizing impact on the workstation owner is of primary importance. Minimizing impact can be achieved by three methods:

---

1 Checkpointing can be performed at either a high level, by capturing program state independently from the binary executable, or at a low level, representing a map of the process in memory. A high level checkpoint is typically more flexible and useful for heterogeneous process migration, however it puts severe constraints on program flow. See Árabe et al. for more information [Árabe et al., 1995]. Recompilation as a tool for heterogeneous process migration has also been proposed [Theimer et al., 1991]. System specific checkpointing can also be performed [Lim et al, 1991].
1. Using the system's *nice* facilities
2. Automatically pausing/resuming jobs with signals
3. Automatically migrating processes off of a newly loaded CPU

Support for these three methods is varied among the packages listed in Table 1, primarily because each of these packages is in a different stage of development. Package's use of the previous three methods are shown in the final column. Typically, the commercial packages represent more mature products that have been in use for as long as a decade or more.

<table>
<thead>
<tr>
<th>Commercial Packages</th>
<th>Vendor</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODINE</td>
<td>GENIAS GmbH, Germany</td>
<td>1,2,3</td>
</tr>
<tr>
<td>Connect:Queue</td>
<td>Sterling Corp.</td>
<td>1</td>
</tr>
<tr>
<td>Load Balancer</td>
<td>Unison Software</td>
<td>1,2,3</td>
</tr>
<tr>
<td>Load Leveler</td>
<td>IBM Corp.</td>
<td>1,2,3</td>
</tr>
<tr>
<td>LSF – Load Sharing Facility</td>
<td>Platform Computing, Canada</td>
<td>1,2,3</td>
</tr>
<tr>
<td>NQE – Network Queuing Environment</td>
<td>Craysoft Corp.</td>
<td>1,2</td>
</tr>
<tr>
<td>Task Broker</td>
<td>Hewlett Packard Corp.</td>
<td>1,2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Research Packages</th>
<th>Institution</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch</td>
<td>UCSF</td>
<td>1,2</td>
</tr>
<tr>
<td>CCS – Computing Centre Software</td>
<td>Paderborn, Germany</td>
<td>1,2</td>
</tr>
<tr>
<td>Condor</td>
<td>University of Wisconsin</td>
<td>1,2,3</td>
</tr>
<tr>
<td>DJM – Distributed Job Manager</td>
<td>Minnesota Supercomputing Center</td>
<td>1,2</td>
</tr>
<tr>
<td>DQS – Distributed Queuing System</td>
<td>Florida State University</td>
<td>1,2</td>
</tr>
<tr>
<td>EASY</td>
<td>Argonne National Laboratory</td>
<td>1</td>
</tr>
<tr>
<td>far</td>
<td>University of Liverpool, UK</td>
<td>1</td>
</tr>
<tr>
<td>Generic NQS</td>
<td>University of Sheffield, UK</td>
<td>1,2</td>
</tr>
<tr>
<td>MDQS</td>
<td>ARL</td>
<td>1,2</td>
</tr>
<tr>
<td>PBS – Portable Batch System</td>
<td>NASA Ames &amp; Lawrence</td>
<td>1,2</td>
</tr>
<tr>
<td>PRM – Prospero Resource Manager</td>
<td>Livermore National Laboratory</td>
<td>1,2</td>
</tr>
<tr>
<td>QBATCH</td>
<td>Vita Services Ltd.</td>
<td>1,2</td>
</tr>
</tbody>
</table>

Table 1: Distributed Batch Software Products
For a host selection policy, all of these packages base their machine choice by making a “resource match” of some type between hosts that are available and the resource requirements of the job, typically given as an input file. Thus they capitalize on static machine information such as available memory, disk, and processor type, combined with the current availability state of the machine. Some packages (CODINE, Load Leveler, NQE) tout a “configurable” host selection policy, the depth of which is not known since none of the packages have actually been installed and tested by the author. An implementation of the algorithm described here could be utilized in any of the above systems that support external plug-ins to implement selection policies.

Resource management for distributed batch systems is a very important line of research [Neuman and Rao, 1993][Wang et al., 1993], and generic scheduling algorithms have seen much research [Baker, D., 1992][Choy and Singh, 1993]. An example of a product in this area is the Stealth distributed scheduler [Krueger et al., 1991]. However, most work in this area has focused on how to ensure robustness by such techniques as co-scheduling [Atallah et al., 1991][Efe and Shaar, 1993] or other means of ensuring fault tolerance [Gait, 1985][Lee et al., 1991]. Scheduling for load balancing has also been addressed [Lin and Raghavendra, 1991]. All techniques described, however, utilize a very limited information set to make the allocation decision, choosing instead to focus on ways to limit the consequences of a bad decision [Shivaratri and Singhal, 1991].

It is my contention that although static machine information and instantaneous machine state (or load) is important in deciding which machine to choose, additional information about the usage characteristics of the machine can help to refine host selection. This thesis presents a different approach to host machine selection, by basing the decision not only on static machine characteristics and current machine state, but also on the past historical usage of the machine.
Overview of RingLeader

**RingLeader Basics**

The RingLeader system is designed to allow engineers at the Hewlett Packard company to approach a modeling experiment in a deterministic fashion. By making a number of machines available for simultaneous computation, engineers can take advantage of concurrency by starting multiple jobs on separate workstations. The ability to simultaneously execute many simulations means that the design space for a particular experiment may be modeled in greater deal, and an increased number of design parameters may be considered. RingLeader guarantees completion of each job as well as transparency to the workstation owner.

The RingLeader system has two primary goals: first, to achieve zero impact on the workstation owner, and second, to maximize use of available computer resources. Although complete accomplishment of the first prerequisite is not feasible, keeping this as the primary goal ensures that decisions regarding aggressive use of workstations are always made with the knowledge of how they will affect the owner. The second goal implies that we should not “waste” compute time by poor scheduling of jobs. Although on the surface this appears to be an easy and somewhat simplistic goal, implementation of such a system is non-trivial. For example, decisions such as when a job should be migrated to another machine to take advantage of faster speed or better availability are difficult and not easily implemented in an efficient fashion.
The Basic RingLeader design is shown in Figure 1. The MCP (Master Control Program) running on a central computer accepts, queues, and delegates jobs to various host workstations. A Monitor Daemon runs on each machine in the RingLeader system. The Monitor is responsible for system and job monitoring of its host workstation, and accepts and responds to commands issued by the MCP. Finally, the Job Submission Tool (JST) provides the user’s interface to the RingLeader system. The JST allows the user to do such operations as submit, abort, stop, and fetch jobs, and provides capabilities for monitoring jobs currently in progress and the status of hosts in the system.

RingLeader does not require any special permissions or execute privileges to run, depending solely on user accessible commands and privileges. Typically this means that a separate “user” account is set up for RingLeader, although a separate account is not necessary. Since each RingLeader user might or might not have an account on each remote machine, jobs are run underneath the RingLeader user ID. Remote execution among hosts must be enabled, however, to allow remote spawning of processes and the
use of various remote commands such as *rcp*. This means that the use of the *rhosts* facility must be enabled on the system that RingLeader uses.

There are two types of communication in the RingLeader system. RingLeader uses *PVM (Parallel Virtual Machine)*, a publicly available product from Oak Ridge National Laboratory, to provide remote spawning and communication between the *MCP* and the *Monitors*. A limitation of *PVM*, however, is that it cannot be used to communicate between two different accounts, meaning that cross-user communication is impossible. Since the primary purpose of the *JST* is to allow multiple users to attach to the *MCP* and run modeling tasks, communication between the *JST* and the *MCP* is performed using *TCP* sockets. The use of a socket does consume a file descriptor, however, so this places an upper limit (system defined) on the number of concurrent users that can connect to the *MCP*. This is one of the reasons *PVM* has been retained for communication between the *MCP* and the *Monitors*; since a typical RingLeader configuration has far more hosts than file descriptors, RingLeader would have to implement *UDP* socket communication to get around this obstacle. Implementation of a reliable *UDP* interface is more complicated than *TCP*, since *UDP* does not guarantee receipt of messages. *PVM* uses *UDP* sockets combined with a transmit/retry mechanism for default routing of host to host messages. This ensures receipt of all messages without unnecessary use of file descriptors [Stevens, 1990].

Each of the following sections describes in greater detail the three primary portions of the RingLeader system. The final section, *Job Flow in RingLeader*, gives a final overview of the steps taken in the completion of a job on RingLeader.

**The Monitor**

The monitor daemon is both the simplest and the most important of the three primary programs in RingLeader. Since a copy of the *Monitor* resides on every host in the system, and is never paused or "niced," it is essential that the *Monitor* be as small and
perform as few tasks as possible. Consequently, the Monitor performs very few operations and makes no decisions. All decision making in the system is done by the MCP. Thus the Monitor has the following duties:

1. Monitor the host's process table to determine current host status.
2. Pause/resume jobs as necessary.
3. Respond to commands from the MCP.

The following subsections cover each of these responsibilities.

**Determining Host Availability**

Determining when a host is available for computation is crucial to ensure that the workstation user is minimally impacted by running a simulation or model on that machine. Primarily, this involves detecting when a user starts to use the workstation, and secondarily, deciding when those operations on the workstation have been completed, and the machine is once again available for use by RingLeader. The first, detecting usage, is of paramount importance, since it is critical for the completion of the first directive of RingLeader: non-interference with workstation owners. The second, determining of availability, is not as important since the user will not be impacted, but it is necessary to identify a machine as being available as soon as possible to ensure that we are getting the most of the computer resources.

Determining when a machine is being used can be done in two fashions, either by an interrupt driven mechanism, or by polling the workstation. An interrupt-driven mechanism requires that a daemon catch or otherwise detect keyboard or mouse usage. When the mouse is moved or a key depressed, the monitoring daemon would send an interrupt to the Monitor. However, since in a distributed environment a user of the machine need not be physically at the machine (i.e., can be remotely logged on), such an interrupt-driven daemon would be unable to determine if a logged on user was actually using the machine, or simply idly logged on. Consequently, RingLeader uses a polling
mechanism to determine whether a process on the workstation has become "active."
Both the polling time and the activity level allowed are independently configurable by
machine.

The activity level allowed determines what background "noise" on the
workstation that RingLeader will ignore, and continue using the machine. Since each
workstation can have a variable number of background and daemon processes even while
idle, an availability mechanism based on the sum of CPU time used or count of active
processes is difficult to tune for a multitude of workstations. A reasonable background
noise on one workstation may permanently preclude another workstation from ever
getting selected. Individually determining background CPU levels would be cumbersome
and hard to maintain. However, upon examination of what happens when a workstation
becomes active, it is observed that one (or more) processes begin to consume CPU time.
If the user starts or activates a process, that process should begin to consume CPU cycles.
Upon polling, the Monitor watches the CPU utilization of all processes on the system,
and when any one process begins using more than the activity threshold percent of the
workstation CPU, the workstation is determined to be busy. All RingLeader jobs running
on the system are ignored in this calculation, and the system nice facility is used to ensure
that any user job will have a higher priority than all RingLeader jobs (all RingLeader jobs
are nice to 39, the maximum allowed by the system).

Determining machine availability in this fashion ensures that any user process that
utilizes CPU cycles beyond the normal background level will be detected by the Monitor.
By choosing a small enough activity level, we are reasonably certain that any user
program will exceed the threshold. This threshold is also independently configurable,
meaning that for faster machines it can be set to a lower level (typically less background
process noise). Default settings for poll time and activity level are 3 seconds and 3%
respectively. These settings do not represent absolute or optimized numbers, but can be
configured per machine. In setting the values, it is important that the poll time not be too
small, since that would result in constant traversal of the process table, while at the same
time if the value is too large, there will be a long detection period between the time a
process begins to consume cycles and the time the Monitor daemon notices the process in
the process table. For the process load, the setting must not be so low that background
system processes trigger the Monitor, but low enough that any reasonable user process
will exceed the threshold rapidly. Also, since any violation of this threshold determines
the usability of the machine, the process table traversal can be short circuited upon
finding a single process that exceeds the threshold. Thus the entire process table need
only be traversed when the machine is inactive.

**Pausing and Resuming Jobs**

In order to ensure that a workstation responds quickly to the user upon
determination of activity, all RingLeader jobs are controlled using signals. When a
machine is determined to be busy, a SIGSTOP is sent to all child processes. The
SIGSTOP signal is one of only two signals in UNIX which is uncatchable by a process.
SIGSTOP allows for immediate stopping of a process regardless of current state, and is
provided to allow the administrator of a UNIX system to stop any job on the system. The
use of SIGSTOP in the suspension of child processes ensures that the jobs are
immediately suspended. Upon determination that the machine is again available, a
SIGCONT is sent to all processes. The SIGCONT signal is guaranteed to work on a job
that has been stopped using SIGSTOP or another pausing signal.

There are two problems with implementing transparency using signals. The first
is that a process that is paused continues to consume memory resources. Although the
job will be swapped out of main memory if it remains paused for a sufficient period of
time, it will continue to use swap space. The combination of a large job on a low
memory or low swap space machine can be disastrous, and result in the user noticing a
serious degradation of machine performance. Consequently, one of the parameters in the
machine selection algorithm deals with amount of memory consumed. If a machine is determined to not have a sufficient amount of memory for the job, it is not available to run that particular task.

The second, and more insidious drawback has to do with the use of signals on a program that might not have been designed or implemented in a signal-robust fashion. Although this has never yet been an issue with RingLeader to date, it is a potential pitfall that warrants mentioning. A crash or program malfunction due to signals is most likely to occur in a blocking system call, such as a read or a write. Such system calls can be interrupted by signals, and if interrupted, typically return an invalid response, and set the errno flag to EINT. It is the programmer's responsibility to check for this occurrence, and re-call the function if it was interrupted. In code that does not fully check and handle all return values from system functions, this exception may not be handled correctly, resulting in invalid errors or incorrect execution. There is no solution to this problem with signals, other than to warn programmers to beware of this side effect.

Responding to MCP Commands

Finally, the Monitor is responsible for responding to MCP requests. The MCP may issue one of several commands to the Monitor; the primary commands are:

- **START** job – accept and begin execution of the given job
- **STOP** job – stop and return files of the given job
- **STATUS** – return status of all the jobs to the MCP
- **ESTIMATE** job – estimate time to complete the given job

These commands are processed in a straightforward fashion. The MCP prefaces the command message with a flag that indicates message type, and the Monitor need only process a switch statement and perform the appropriate activity. Jobs are started by processing a fork and exec pair, spawning a child that executes a command line specified in a job definition file. Since each spawned job is a child of the monitor, the monitor uses
the receipt of the SIGCHLD signal as notification of job completion, and maintains a list of child pids which the monitor uses to send the SIGSTOP and SIGCONT signals to depending on the availability of the system. Another possible caveat enters here: RingLeader processes cannot execute the fork command, since RingLeader does not currently handle the problem of grandchildren. If a RingLeader process fork's and exec's another process, that new process would not run within the RingLeader system.

Terminating a job sends a SIGKILL to the child task, to ensure that the child process cannot catch and ignore the SIGINT signal. Status messages to the MCP contain current information on each child task, including current state (paused or running), total elapsed run time, and total elapsed pause time. The MCP combines this information with basic job information retained on the server to present a detailed status report to the user.

Estimation of job completion is a task required by the machine selection algorithm, and is covered in detail in Resource Allocation Design.

The Master Control Program

The MCP is the centralized controller of the RingLeader system. As such, it performs all tasks related to accepting, maintaining, delegating, and controlling job flow, and adding, removing, and restarting monitors. The MCP’s duties, are therefore quite varied; since a detailed description of the MCP’s duties would be too lengthy for this document, I have included a few sections that cover some specifics necessary for a basic understanding of how the MCP operates. These sections follow, and include system setup, fault tolerance, and user response.

System Setup

On system startup, the MCP starts a PVM virtual machine automatically, and reads a host configuration file, which provides host specific information, such as host name, speed, and configuration, allowing each host to be configured differently. Each
host is added both to an internal host table and to the PVM virtual machine; once a machine has been added successfully, the monitor program is spawned and a configuration message sent to the newly created monitor. Once the monitor is verified to have been successfully added and started, the next monitor is started. Currently this results in a fairly lengthy startup process, which is being optimized. Hosts that are not successfully started are marked with an error tag. Each evening, as described in the next section, the MCP attempts to re-add hosts that have failed during the day or which were not able to be restarted the previous evening.

Once the hosts in the host file have been added, the MCP checks its job directories for jobs that were in process when the MCP was shut down or failed. Jobs that were pending or in progress are restarted, and jobs that were otherwise completed are given the appropriate status. This is described in more detail in the next section.

After hosts and jobs have been added, the system is ready to accept commands. In contrast to the Monitor, the MCP accepts messages both from the user and from the various monitors. Message tags are used to distinguish between a COMMAND message and a CONTROL message. Examples of COMMAND messages are JOB_START, JOB_STOP, JST_STATUS, etc. User interface commands are described in The Job Submission Tool, but are typically handled in only one of two fashions: the message is either immediately responded to with information available on the MCP, or a message is passed to the appropriate host. Depending on the type of command, the MCP may either wait for a response from the host, or immediately return a response to the user.

Fault Tolerance

RingLeader is designed to run on multiple workstations, and to run jobs whose lengths may be measured at best in hours, normally in days, and at worst in weeks. With such long, continuous jobs, it is important that the MCP be able to handle the failure of individual machines and be able to recover and restart without human intervention. The
MCP accomplishes this by using the notification facilities of PVM. PVM provides notification messages on specified events, such as host or program failure (synonymous, from the MCP perspective). On being notified that a task has failed, the MCP marks the host with an error in its host table, and checks to see if any jobs were running on the remote host. If a job had been delegated to the failed host, the MCP re-delegates the job to a different host. In order to allow programs to provide their own restart capability, jobs can be tagged with a special restartable flag, which indicates to the MCP that files from the remote host are to be copied back to the MCP before the job is sent to a new host. The job will remain in a special restarting state until the files have been obtained.

We are currently looking into the potential of integrating a third party checkpoint/restart mechanism into the RingLeader system that will allow jobs to be checkpointed and migrated at will. Such a facility would allow even better job scheduling, allowing a job in progress to be migrated off of a busy machine to a currently available workstation. Since this will be a feature in the future, the machine selection algorithm described in this paper has been designed with this additional capability in mind – specifically, allowing the MCP to vary the predicted run time to coincide with optimal algorithmic performance, as described in Resource Allocation Design.

Since RingLeader follows a centralized design, the MCP represents a single point of failure for the RingLeader system. The system has been designed to allow individual host and monitor failure, job failure, and JST failure, but a crash of the MCP does cause all currently running jobs to stop. In order to avoid the loss of job information, the MCP is stateless [Tanenbaum, 1992]. All information pertaining to the jobs on the system are contained in job files on the server. The MCP distinguishes jobs by directory, and maintains a pending, working, and complete job directory. Underneath these directories, each job is stored in its own subdirectory given by its unique job identification number. Each job directory contains a job file, created by the user and appended to by the MCP that gives RingLeader information on the user, initial directory, files to send, files to
receive, and how to run the program. Thus on a restart, the MCP is able to re-determine all job identification numbers and pertinent job information by scanning all of the job directories. Although a copy of all the job files is always maintained on the server, in the event of a system crash the most recent job information is actually found on the remote systems, so the monitors and the MCP coordinate to restart the jobs.

**User Response**

As described in System Setup, the MCP accepts and processes two types of messages: user commands and control messages. The messages are processed in a sequential manner; no multi-threading is currently implemented, although this may be a future improvement. PVM takes care of ensuring delivery and provides ordering of messages to ensure that they arrive in the proper order (for multi-message commands). To ensure that user response is quick, and to help prevent visible “hangs” of the system, as many messages as possible are processed with information contained on the MCP. Types of information that can be returned are host lists, job lists, detailed job information. Commands that can be immediately processed on the MCP include starting and deleting jobs. However, some commands must be forwarded to the appropriate host, such as stop and abort. Also, the status command requires that each host be queried for the current job status. The MCP takes the appropriate action and in all cases returns a success or error string to the user.

**The Job Submission Tool**

The final major portion of the RingLeader system is the JST. Since there is no shell level interface to the rest of the system, the only way that users (and administrators) interact with the RingLeader system is through the JST. However, since multiple users can simultaneously connect to the MCP, and PVM is inherently single user, the Job Submission Tool was broken up into two parts. The first, the Gatekeeper, is a server
program that resides on the MCP host, accepting multiple client accesses. The shell interface (also referred to as the JST) is what the clients actually use.

**The Gatekeeper**

The Gatekeeper's primary responsibility is to translate client requests from multiple TCP/IP streams into single-user PVM messages. It is a simple program adapted from examples in Steven's book *TCP/IP Programming* [Stevens, 1990], and consists of a single server program that accepts requests from clients. On receipt of a client request, the server forks a child, the child and the client interact, and the server waits for another request. In this manner, multiple clients can simultaneously connect to the Gatekeeper, and each Gatekeeper child is dedicated to handling and processing messages from only one client. Again, since PVM messages can only be sent within a single user environment, the Gatekeeper translates multiple user requests, which come in the form of TCP/IP message streams, into equivalent PVM messages to pass to the MCP.

Primarily, this design was implemented to keep the MCP's message handling process very simple. The MCP doesn't have to care or worry about various clients, forking to allow another client access, etc. All the MCP knows or cares about is the PVM generated task id (tid) of the source. Once the message processing is complete, PVM returns the result to the appropriate tid, and returns to process another message. Each Gatekeeper, after being created from the server, is assigned a new tid, so each tid individually represents a single client. The Gatekeeper receives the message back from the MCP, translates it into a TCP message stream, and returns the result to the client. So long as the client remains alive, the appropriate gatekeeper child remains to provide services for that client. The use of TCP means that there is a finite limit (typically ~32) to the number of concurrent client accesses, since each open TCP socket requires a file descriptor, and the operating system typically limits the number of open file descriptors that a single process can consume [Stevens, 1990, 1992].
The *Gatekeeper* is also used for transferring files between the *MCP* system and the client’s machine. Early versions of RingLeader transferred files with *rcp* and automated anonymous ftp. Using *rcp*, however, requires that the user have their account set up properly and that each user have an account on both the submitting and server machine. This was an assumption that we did not want made, since the server is typically a limited access machine. Anonymous ftp worked, but required the files to be vulnerable for a period of time before the *MCP* moved them out of the public directory. The final solution was to use the socket already set up for communication between the gatekeeper and the client shell. This solution provides a secure link for file transmission and eases the burden of transferring files between the clients and the *MCP*.

**The Command Shell**

The command line shell that the user employs to execute *MCP* commands is very simple. It can be run in two modes, either single command or shell mode. To execute a single command, the user need only append the appropriate command to the execution line. That command is executed, and the program exits. This facility was provided to allow easy scripting of *JST* commands to automate experiment creation and submission. When the user enters shell mode, a different prompt is simply provided to the user. Each *JST* command typed at the prompt is executed by passing the appropriate message to the *MCP* (through the gatekeeper) and waiting for the result. The shell is the first line of defense against invalid commands: the *JST* ensures valid command and syntax before forwarding the message. Since the *JST* cannot perform argument checking (e.g., check whether or not a given job ID is valid), the *MCP* ensures validity of arguments. The shell remains open until the user explicitly exits with a “quit” command. A screen shot showing the output of the ‘help’ command is shown in Figure 2.
As the last section in this overview of RingLeader, it is important that the operation and stages through which a job progresses be clear. Figure 3 shows the process flow that a job follows in the RingLeader system.
Figure 3: Job Flow in RingLeader

The user submits a job to the JST, which coordinates with the MCP to get a job identification number. The JST then transfers the files into the appropriate directory on the MCP (given by the job ID), and notifies the MCP that the job is ready to be executed. The MCP places the job in a job queue, determines the appropriate machine to which the job should be assigned, and directs the Monitor on that host to execute the given job. The Monitor then copies the files to its local drive (to reduce network traffic) and executes the given job. Once the job finishes executing, the Monitor notifies the MCP and copies the result files back to the MCP. The job then shows up as complete, and the user is able to retrieve their results through the JST.
Resource Allocation Design

Design Overview

In a distributed batch system such as RingLeader, the decision of which machine to run a particular task on is a complex problem. There is no "optimal" solution, since a perfect solution would schedule machines with foreknowledge of both future machine usage and future job submissions. The algorithm described here, which I am calling the historical or history-based algorithm, makes the assumption that job submission rates are entirely random and cannot be predicted, but relies on the fact that the machines being used in RingLeader are part of a network where the usage patterns may be predictable. By taking into account both static information, such as machine speed (determined through benchmarking), disk space and physical memory as well as dynamic information such as the current status of usage on the system and a historical pattern of usage, I have attempted to build an algorithm that makes better machine allocation decisions than can be made when only the static machine information is considered. As described in the Related Work section, current products use only static machine information (resource requirements) with limited dynamic information (e.g., current load). In brief, this historical algorithm uses a static table of host information combined with past and current usage to determine which host to use. There are two types of machine information used in the selection of an appropriate host, which, when coupled with specific job information, allow the algorithm to operate. These information types are addressed in the following three sections.
Static Machine Information

There are three static pieces of information used in the determination of a machine on which to run a job. These are:

1. Processing power (speed) of machine
2. Available physical memory
3. Available hard disk space

The speed of each machine in the test configuration (made up of 111 HPUX workstations) was determined by running a “typical” computational fluid dynamics problem, and is normalized, with the ranking of 1.0 arbitrarily assigned to the most common machine in the test configuration, a HP 735/100. The speed of the machine is used in the selection algorithm to attempt to estimate the time to complete the job. The second two pieces of static information used in machine selection act as filters to ensure that for a given job, only machines “suitable” to run that job will get selected. Any machine that does not meet the minimum requirements for memory or disk space is not eligible to run a given job. In the rare case of a “tie” between machines (from the dynamic information), the machine with the better static attributes (i.e., more memory and/or more hard disk space) is chosen.

Dynamic Machine Information

The dynamic information used in the host selection algorithm is twofold: current usage (i.e., is the machine busy right now), and historical usage (i.e., what has the “typical” usage been for this day of the week). As described above in Determining Host Availability, each Monitor program continuously polls the process table in an attempt to determine if the machine is being used. Instead of discarding this information, the results of each poll is saved so that a historical measure of activity can be reconstructed. Since actually storing every single load level at the typical poll interval (3 seconds) would
eventually consume an enormous amount of disk space, each day (Sunday through Saturday) is broken up into intervals (called “windows”) of time. In each given window the mean availability for that time period is stored, as well as information necessary to compute the standard deviation. For all of the experimentation described in this thesis, the window interval is ten minutes. To track this usage information, each time the monitor checks the load it increments two counters: how many checks were executed, and how many of the checks returned busy. The Monitor then sets an alarm signal to go off every window interval, at which time it stores the information for the past window and resets the flags.

Enough windows are stored so that each day can be treated separately, and each window information is combined with past traversals through that window. For example, every Monday from 8:40-8:50 a.m. would fall into the same window. For the current setup of 10-minute windows, storage space is needed for 1008 windows (144 per day), not an unreasonable amount of disk space. This file is also guaranteed not to grow in size as the length of history stored increases.

The premise behind saving this information is that the machines that RingLeader will be used on (and has been used on for several months) are business machines. Most of the machines have a single owner/user, which means that the usage of a given workstation should track closely with that person’s duties and schedule. In a business environment like Hewlett Packard, there are several ways that predictability can occur. Weekly meetings and daily schedule provide for a majority of the predictability achievable through a history gathering mechanism. Most people (at Hewlett Packard, at least) tend to work the same schedule each day, and the usage patterns of each machine can reveal which users are “power” users, meaning that their machine exhibits a higher than normal usage pattern.
Job Information

The final pieces of information used in the algorithm are job specific. Each job submitted to the RingLeader system has attributes associated with it, including owner, send files, return files, restartability, as well as estimated job length, estimated memory required, and estimated disk space required. These last three pieces of information are needed for the selection algorithm. Default values are used if any of the three values are not supplied, and were selected for “typical” applications, so they need to be modified only by unusual applications.

Since the memory and disk space estimates are simply used as filters, setting a default value high enough to fit the “typical” range of applications means that only programs that require particularly low or high values should specify these. The estimate of job length, however, is another matter. One of the planned improvements to the RingLeader system in the near future is the addition of task migration. The job length is currently a user specified parameter, but in the context of a migratable job, the minimal length of time that a machine should be scheduled is dependent not on the total run length but the amount of time that would be necessary to overcome the costs of job migration. Thus, in a system capable of dynamic process migration, the minimum scheduling unit is not the entire job length, but the shortest amount of time that would compensate for the price of migration. However, in the current RingLeader system only the total job length is used for scheduling of jobs.

Selection Algorithm

The algorithm for host selection then, is as follows: when a job is submitted to the MCP, the MCP sends a “bid” request to each machine that a) meets the minimum memory requirements for the job, and b) does not currently have a job running on it. Each Monitor then checks available disk space, and returns a host-speed modified
estimated job completion time, computed by traversing the history table on the given host. The MCP retrieves each of the results and chooses the lowest bidder.
Algorithm Performance

Experimental Design

This history based algorithm was verified by simulating its performance with real world data gathered from a network of workstations at Hewlett Packard. Since the purpose of a more "intelligent" algorithm is to optimize run time by minimizing the average wall clock time of jobs on the system, my algorithm was compared against two other more simplistic algorithms: a random selection, and a "fastest available machine" algorithm. For this simulation, all machines were assumed to have sufficient memory and disk capacity to meet the static machine requirements for any job submitted. Each of these three algorithms then represents the use of additional dynamic information in the selection process. The random information uses no machine specific information to decide which machine to use, the fastest machine uses only a normalized machine speed, and my selection algorithm adds historical usage information to optimize machine choice. Each algorithm, then is a superset of the previous, as shown in Figure 4.

In each algorithm, should the differentiating information be removed from the calculation, the algorithm becomes equivalent to the next most simplistic method. For example, if all the machines were of speed 1.0, there would be no differentiation between the fastest machine available method and a random selection. The same applies for the difference between my historical selection algorithm and the fastest machine method. If all of the machines had zero usage, or an identical historical pattern, then the machine chosen would be the fastest available machine. The historical algorithm relies on both difference and repetition to be able to predict host availability.
Figure 4: Algorithmic Hierarchy

Historical Machine Usage on Mondays before Monday August 19

Figure 5: Historical Data Example
In order to test the performance of my history-based selection algorithm, it was necessary to gather historical data to use in the simulation. The test began by running the history gathering mechanism on 111 workstations for a period of approximately five weeks. A sample of the data from one machine, for one day of the week (Monday) is shown in Figure 5. Although this simulation was run on a relatively small historical data set, the history-based mechanism has remained in place since this study took place, which should allow a future test to be run, to verify that accuracy increases as the historical database gains references.

In order to simulate realistic machine usage, once the historical data was gathered, on each of the workstations another program was run that collected maximum process load (percent of CPU use) at five second intervals, for a period of one week. This data was used to test each algorithm’s machine selections, and represents actual machine usage on all 111 workstations for the third week in August, 1996. A sample of this data is shown in Figure 6, and corresponds to the same day of the week (Monday) and machine whose history prior to this day was shown in Figure 5.
In an attempt to model actual user submission rates, engineer job submissions for the month of August were tracked, with the results shown in Figure 7. A third program was written that generated approximately 200 random job start times over a single week, following the daily submission ratios shown in Figure 7. Given the typical working schedule of Hewlett Packard engineers, the jobs were assumed to be submitted in a uniform random distribution from 7:00 a.m. to 6:00 p.m. Since typical usage of RingLeader at HP has shown that many users submit multiple jobs simultaneously, each job submission time contains a random number of jobs to be submitted. In order to give a

---

1 In Figure 6, the percent utilization represents the maximum usage of any single process within a one minute period. It does not represent the total percent system utilization. The 3% line illustrates the default “busy” limit in RingLeader. Also note the early morning utilization, most likely caused by the nightly backup and maintenance scripts.
non-uniform distribution to the number of jobs submitted, the probability of a job of length $x$ is given by $f(x) = 1/(x+1)$, normalized over the range $x=[0,9]$. The count was then given by $x+1$, yielding a range of job count from 1-10, with emphasis on the lower end of the range. A histogram of the number of occurrences of each job count is shown in Figure 8. Finally, each job was given a random length, from 30 minutes to 24 hours.

Results were placed into files similar to Table 2. This file was then sorted by submission time so that the jobs could be submitted to the simulator in chronological order.

<table>
<thead>
<tr>
<th>Day</th>
<th>Window</th>
<th>Count</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>38</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>43</td>
<td>1</td>
<td>97</td>
</tr>
<tr>
<td>1</td>
<td>50</td>
<td>2</td>
<td>116</td>
</tr>
<tr>
<td>1</td>
<td>57</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>63</td>
<td>4</td>
<td>102</td>
</tr>
<tr>
<td>1</td>
<td>65</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>68</td>
<td>2</td>
<td>40</td>
</tr>
<tr>
<td>1</td>
<td>69</td>
<td>2</td>
<td>28</td>
</tr>
<tr>
<td>1</td>
<td>72</td>
<td>3</td>
<td>107</td>
</tr>
</tbody>
</table>

Table 2: Sample Job Submission Times

---

1 In Table 2, Day and Window represent the day of the week and the 10 minute window of the day. Count gives the number of jobs to be submitted at the given time, and Length is the length of the job in ten minute intervals.
Number of Jobs Submitted During August

![Bar chart showing job submission rate over a week with counts for each day of the week ranging from 14 to 142 jobs.]

**Figure 7: Actual Job Submission Rate**

Taking the previous machine history, actual usage for a week, and a random table of job submission times as input, a simulation program was created that would "run" each of the jobs in the jobfile on the 112 workstations used as hosts, simulating a week of actual job submissions to RingLeader. The simulation iterated through three algorithms:

- Random – randomly chooses an available machine
- Fastest – chooses fastest (from benchmark) available machine
- Historical – chooses “best” available machine according to history-based mechanism

---

1 It is interesting to note the almost bimodal distribution of jobs over the course of a week. An interesting extension to this work might be a human factors study to determine if the pattern continues, and if so the causes of such a pattern.
Each simulation run produces three files, one for each selection algorithm. As shown in Table 3, each file details job number, algorithm, time, machine details, expected length of the job, and how long the job actually took to run. Analysis of the data generated is described in the Experimental Results section.

Occurrence of Job Count

![Random Job Count Distribution](image)

Figure 8: Random Job Count Distribution
Table 3: Sample Simulation Output

Experimental Results

The first experiment ran through the simulator was used to determine whether or not there was a significant improvement between the three algorithms. Since the measure of “goodness” is wall clock time, a ratio of job completion time vs. job length was computed, giving the average completion ratio for each of the three algorithms. In order to compare the algorithm’s significance, each algorithm was run on thirty (30) separate random data sets. Since the Central Limit Theorem [Ross, 1990] asserts that the sum of a large number of independent random variables has approximately a normal distribution, this allows the mean and standard deviation to be computed from the results, and to be treated in a normal fashion. The results, showing 95% confidence intervals with the error bars, are shown in Table 4 and graphed in Figure 9.
Table 4: Average Completion Ratio (Time/Length)

The results do show that each of the three algorithms is considerably different from the others, within their 95% confidence intervals, which demonstrates appreciable performance within the context of the experiment. The real question that this simulation is trying to answer is “does using additional information in host machine selection increase the probability of lowering job completion time.” The random allocation method has basically no information about the machines from which it is selecting a candidate, and thus does the worst. The fastest machine algorithm adds only one data point, that of a relative machine speed, and by choosing the fastest free machine, hopes to optimize run time. My historical algorithm adds additional information in the form of past historical machine usage. Since average completion time improved by a factor of approximately 15% over just choosing the fastest machine available, the additional history information was valuable in the selection decision. This is evident because of the superset relationship each of the algorithms has with the others.

Figure 9 represents an actual completion ratio of the time to complete a given job, compared to the length of the job on the standard, or speed = 1.0 machine. This is an important point because it means that the completion time is modified for host speed. A random selection of a 0.5 speed machine will never be able to achieve a better than 2.0 ratio of completion time to job length. However, this distinction is important since in any actual installation the speed and capabilities of the available workstations will vary. The particularly bad showing that the random allocation demonstrates is due in large part to

<table>
<thead>
<tr>
<th></th>
<th>Avg.</th>
<th>+/- 95% c.i.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>1.74</td>
<td>0.096</td>
</tr>
<tr>
<td>Fastest</td>
<td>1.29</td>
<td>0.072</td>
</tr>
<tr>
<td>Historical</td>
<td>1.15</td>
<td>0.016</td>
</tr>
</tbody>
</table>
the selection of machines with less than a speed of 1.0. Both the historical and fastest machine algorithms take the speed of the machine into account when choosing a machine.

Average Job Completion Time

![Figure 9: Job Completion Time Ratio](image)

By extracting out jobs of various lengths, the run time to job length ratio was also computed for jobs of varying sizes. Figure 10 shows the performance of each algorithm as the job length scaled up to 24 hours of run time. What is interesting and exciting about this graph is that it demonstrates that as the run time increases, the decision making process becomes less of a factor. This is most likely due to an averaging effect, wherein over the course of a given 24 hour period, each of the machines are used approximately
the same amount. Also, the ability to predict usage becomes less effective as the computed time grows, meaning that the historical based algorithm has less of an advantage over the fastest machine algorithm. Note that the best performance of the historical algorithm was actually achieved in jobs of less than one hour, which is exactly the same time frame that both the random and fastest machine algorithm posted the worst overall ratio.

### Average Completion Time vs. Job Length

![Average Completion Time vs. Job Length](image)

**Figure 10: Job Length Comparison**

The ratio showed in Figure 10 can be inverted to demonstrate the effect that the decision algorithm would have on the throughput of jobs in the RingLeader system. Figure 11 demonstrates the number of jobs that each algorithm would be able to complete per machine in one week's time. Especially for the shorter job lengths, the history based
algorithm's superiority has a dramatic effect on the number of jobs that could be completed within a given period of time.

Job Throughput

![Job Throughput by Algorithm](image)

**Figure 11: Job Throughput by Algorithm**
Conclusions

This simulation was conducted to verify the premise that utilizing historical usage characteristics of a machine helps predict the future usage patterns of that workstation. Since such information usage was shown to be of benefit in this sample environment, it is worthwhile to implement such a history based algorithm for at least a subset of computational tasks. Results showed a decrease of approximately 15% over the simple selection of the fastest available machine. For scheduling tasks of less than 12 hours, the historical algorithm provides significant improvements over the fastest machine algorithm. However, for tasks of length greater than 12 hours, the benefits of applying additional information to the host selection algorithm provides minimal benefits.

Although the results from this simulation are specific to the environment in which they were conducted, namely in a production CAD environment similar to that at the Hewlett Packard Corvallis site, the results should be similar in any comparable “real world” surroundings in which workstation usage may follow a predictable pattern. This is due in large part to a normally regular work schedule, and limited use beyond the machine owner. Applicability to a shared or laboratory environment may be limited, due to the heavy and unpredictable usage of the workstations. However, shared environments reduce or eliminate complications arising from individual ownership of workstations, perhaps allowing a more aggressive utilization scheme.

This study has many possible offshoots, since it delves into both human factors and algorithmic areas of computer science. For the human portion, it would be interesting to conduct a further, more in depth study of the workstation use patterns. It would also be valuable to study job submission rates and usage of a distributed system such as RingLeader. Interesting usage patterns, such as the bimodal distribution shown in Figure 7, could be studied in further depth. Algorithmically, extensions toward a faster,
centralized algorithm would be valuable, as would improvements in the prediction process.

Finally, I note that I would anticipate that the history should be based on a "sliding window" of history, rather than the entire history. Work should be done to determine an optimal number of days/weeks/months to track the machine usage. The reasoning would be that any actual patterns would eventually get drowned in the noise of the number of historical readings.

This thesis has described a problem, that of determining which machine on which to run a batch job in a distributed system, and a possible history-based solution. The algorithm was implemented and a simulation run against two more simplistic algorithms, and was shown to outperform even the choice of the fastest available machine, which is essentially the same as the most commonly used algorithm among related products. Due to its success, the algorithm will be folded into the RingLeader system within the next few months.
Bibliography


