

Department of Honors
University of South Florida
Tampa, Florida

CERTIFICATE OF APPROVAL

Honor's Thesis

This is to certify that the Honor's Thesis of
Bachelor of Science in Computer Science

ASHLEY HOPKINS

with a major in Computer Science has been approved
for the thesis requirement on April 18, 2003
for the Bachelor of Science in Computer Science degree.

Examining Committee:

Major Professor: Kenneth Christensen, Ph.D.

Member: Zornitza Genova Prodanoff

REMOTE++: A TOOL FOR AUTOMATIC REMOTE DISTRIBUTION OF
PROGRAMS ON WINDOWS COMPUTERS

by

ASHLEY HOPKINS

A thesis submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

May 2003

Major Professor: Kenneth Christensen, Ph.D.
Member: Zornitza Genova Prodanoff

ACKNOWLEDGEMENTS

I wish to thank my faculty advisor, Dr. Kenneth Christensen for his encouragement, his enthusiasm, and his support in writing this thesis. He explained things clearly and simply, but also made me think. Without his help and great ideas I would have been lost.

I also wish to thank my committee member Zornitza Genova Prodanoff for taking the time to read this thesis and provide valuable feedback. Additionally I would like to thank Bronwyn Thomas for her assistance in editing this work.

Thank you to the NSF for providing the REU grant which funded the research that this thesis was based upon. Without this grant I would not have been given the opportunity to be exposed to the research environment that is the graduate student's life.

Finally, I wish to thank my parents, Debbie Mahaney and Gary Hopkins, and Michael for always standing behind me. They have always encouraged me and guided me, never trying to limit my aspirations.

TABLE OF CONTENTS

LIST OF FIGURES	iii
ABSTRACT	v
CHAPTER 1 INTRODUCTION	1
1.1 Parallelization Methods	1
1.2 Parallel Independent Replications	2
1.3 Parallelization in Windows	3
1.4 Organization of Paper	4
CHAPTER 2 LITERATURE REVIEW	5
2.1 Review of Remote Shell (rsh) and Remote Execution (rexec)	5
2.2 Review of Remote Execution in UNIX	7
2.2.1 Xdistribute	7
2.2.2 Condor	9
2.2.3 Akaroa2	11
2.3 Review of Remote Execution in Windows	12
2.3.1 Condor for Windows NT	12
2.4 Review of GRID Computing	12
2.4.1 NetSolve	14
2.4.2 Condor-G	15
2.4.3 Grid Computing in Use	16
2.5 Distributed Operating Systems	18
2.5.1 Ameoba	19
2.5.2 Beowulf	19
2.6 Review of Existing REMOTE Tool	21
CHAPTER 3 GOALS OF REMOTE DISTRIBUTION	24
3.1 Remote Distribution System	24
3.2 Requirements of Distribution Tools	25
3.3 The Jobs	26
CHAPTER 4 REMOTE++DESIGN AND IMPLEMENTATION	28
4.1 REMOTE++ Components	28
4.2 Remote Distribution Structure in REMOTE++	31
4.3 REMOTE++ Input and Output	34

4.4 User View of REMOTE++	35
CHAPTER 5 EVALUATION OF REMOTE++	38
5.1 Overview of Queues	38
5.2 M/M/1 Queuing Systems	39
5.3 Evaluation of REMOTE++	41
CHAPTER 6 SUMMARY AND FUTURE WORK	44
REFERENCES	47
APPENDICES	50
Appendix A: REMOTE++ Code	51

LIST OF FIGURES

Figure 1: Main screen of Xdistribute tool	9
Figure 2: The SETI@home project's screen saver	17
Figure 3: The Folding@home project's screen saver	18
Figure 4: A Beowulf system diagram	21
Figure 5: Diagram of Remote distribution system	25
Figure 6: Help screen for REMOTE++	29
Figure 7: Sample execution of transfer command	30
Figure 8: Sample execution of run command	30
Figure 9: Flowchart of run function	32
Figure 10: Flowchart of run thread	33
Figure 11: Sample rsh and rcp command sequence	35
Figure 12: Sample joblist.txt file, hostlist.txt file, and status.txt file	37
Figure 13: M/M/1 queue	40
Figure 14: Execution times of M/M/1 simulation	42
Figure 15: Simulations time versus target utilizations for an M/M/1 queue	43
Figure 16: Order 6 polynomial growth trend line with results from M/M/1 queue	43

REMOTE++: A TOOL FOR AUTOMATIC REMOTE DISTRIBUTION OF
PROGRAMS ON WINDOWS COMPUTERS

by

ASHLEY HOPKINS

An Abstract

of a thesis submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

May 2003

Major Professor: Kenneth Christensen, Ph.D.

Execution of simulation programs requires large amounts of CPU resources and therefore takes many hours to execute. At the same time, many single-user computers spend much of their time sitting idle. Parallel Independent Replications (PIR) is one method of reducing simulation run time by enabling time-based parallelization of simulations on distributed machines. Most existing PIR systems are designed for Unix.

REMOTE is a Windows-based program for distributing executables to idle Windows PCs. This thesis describes the development of REMOTE++, a new program that builds on the previous REMOTE version. REMOTE++ replaces the complex sockets interface used by the original REMOTE tool with standard remote shell (rsh) and remote copy (rcp) services. These services enable remote program execution and the transfer of input and output files to and from the remote machines. To enable execution of REMOTE++, each remote computer need only run an rsh/rcp daemon.

A single master computer maintains a job list and host list and distributes jobs (from the job list) to remote hosts (from the host list). The execution results are on the master computer at the completion of a remote execution.

The REMOTE++ program was used to investigate run time trends of the simulation time needed for steady state simulation of an M/M/1 queue at utilizations approaching 100%. It was found that as the utilization approaches 100%, the simulation time grows at a rate slightly faster than order 6 polynomial growth.

Abstract Approved: _____

Major Professor: Kenneth Christensen, Ph.D.

Associate Professor, Department of Computer Science and Engineering

Date Approved: _____

CHAPTER 1 INTRODUCTION

Simulations are used for modeling in many fields of the sciences and engineering. Many of these simulations require extensive CPU processing, and therefore, take many hours to execute on a single machine. At the same time, many computer resources are underused. Computers sit idle for many hours in the evenings, on weekends and while computer owners are completing other tasks. Remote distribution and parallelization of programs can be used to reduce execution time of large programs, including simulations, by harnessing idle computer resources.

1.1 Parallelization Methods

Parallelization of programs can be used to reduce the execution time of an experiment by enabling segments of the experiment to be executed in parallel. There are two methods of parallelization. Space parallelization involves splitting a single process into segments to be executed. Time parallelization applies to experiments that requires multiple executions of a single process. Both methods can be used to reduce execution time, but they apply to different types of applications. This thesis discusses one program which implements time parallelization.

Space parallelization involves splitting a single very large program into independent pieces that can be executed at the same time with little to no sharing of data required between them. Research into the reduction of execution time has focused on space parallelization. However, it is only applicable to programs that can be easily split into independent processes. There is no reliable method for automatically splitting programs, so each application requires manual division of source code before execution is possible.

Time parallelization is used to execute relatively small programs, which can be executed on a single machine but require multiple runs to complete a project or simulation. Many simulations fall into this category because multiple input values must be investigated to complete a single experiment. Simulations may also need to be executed multiple times to reduce the occurrence of evaluation errors. For example, the simulation of a queue requires multiple runs with different input parameters and control variables to determine its behavior. Time parallelization would enable several instances of this program to be executed on several computers at the same time, each with different input values to reduce the overall run time of the simulation. Also, because time parallelization involves executing each instance of the program in full, there is generally no need to modify the original program. In effect, time parallelization is applicable to many processes that are not well suited for space parallelization.

1.2 Parallel Independent Replication

Parallel Independent Replications (PIR) uses time parallelization of programs. PIR is used to distribute programs to multiple machines to be run in parallel. Typically, PIR is used to distribute multiple instances of the same program to a group of machines, using different input parameters as described above. PIR can also be used to distribute different executables to each machine in the group to run in parallel. This allows multiple parameters to be held constant. PIR enables distribution of any set of executables that are independent processes. This method does not reduce the execution time of any single process, but will reduce the time needed to complete the set of processes. Through parallel execution of these programs, a greater number of input and control variables can be evaluated, which provides increased output, and ultimately more accurate results.

1.3 Parallelization in Windows

Program distribution tools have primarily been developed for Unix platforms. There exist few PIR tools that enable distribution for Windows PC's. One reason for this may be that research has traditionally been done on Unix platforms, and therefore many of the processor intensive programs run in Unix. Unix also has standard components that readily enable remote execution. However, Windows machines have become the predominant computer resource. Many of these machines are underused and thus have many hours of idle CPU cycles. Recent advances in the memory and processing power of these machines has also resulted in the development of many large programs and simulations that run on Windows machines. As a result, a PIR tool is needed to capture

these idle CPU cycles and use the cycles to reduce the execution time of these simulations.

1.4 Organization of Thesis

This thesis will discuss the development and evaluation of a Windows based distribution tool REMOTE++, which uses PIR to achieve time parallelization of programs. The remainder of this thesis is organized as follows:

- Section 2 describes existing methods for remote execution including execution in Unix and Windows, GRID computing, operating systems, and the original REMOTE tool.
- Section 3 provides an overview of the goals of PIR.
- Section 4 discusses design and implementation of REMOTE++.
- Section 5 contains an evaluation of the REMOTE++ program.
- Section 6 is a summary and describes future work needed to improve the program.

CHAPTER 2 LITERATURE REVIEW

In this chapter, existing methods and tools for remote distribution and execution of programs are reviewed. Section 2.1 looks at remote shell (`rsh`) and remote execute (`rexec`) commands. Section 2.2 describes some existing tools developed for Unix platforms. Section 2.3 describes an example of existing tools developed for Windows machines. Section 2.4 presents GRID computing and current applications. Section 2.5 describes operating systems designed for remote distribution. Section 2.6 introduces the original REMOTE tool used as a basis for REMOTE++.

2.1 Review of Remote Shell (`rsh`) and Remote Execution (`rexec`)

Unix provides standard commands that facilitate remote execution of a process [25]. This enables the user to take advantage of better processing power from the remote machine or to access software that is not locally available. There are two such commands that are standard to Unix platforms, remote shell (`rsh`) and remote execute (`rexec`). Both of these commands allow the user to execute a single command on another host and receive the results on their local machine. The user supplies remote execute (`rexec`) with the hostname of the remote machine, username, password and the command to execute, which are then passed to the host machine. The remote host verifies the username and

password then allows execution of the command. Remote shell (`rsh`) has a very similar process, but in place of checking a user name and password, `rsh` checks an access list stored on the remote machine, which lists the hostnames of all machines granted access. These commands allow execution of a single process on a single remote machine and then wait for the process to complete. All output and error messages from the remote machine are displayed at the local machine where the command was issued. Several remote distribution programs combine these commands into a script that distributes programs, a few of these programs will be addressed in following sections.

A daemon must be present on the remote machine to enable execution of the `rsh` and `rexec` commands. These daemons (`rshd` and `rexecd`) listen for service requests at the port indicated. When a service request is received, the daemon checks if the port number is in the range of 512 to 1023, then the `rshd` verifies the hostname against the access list or the `rexecd` verifies the username and password. Then the command is executed.

Remote shell and remote execute daemons are standard on Unix platforms. Windows, however, does not include such daemons. Windows does support the `rsh` and `rexec` commands, but only to distribute programs to Unix machines. To enable distribution to Windows machines, daemons are available from independent vendors. A free `rshd` is also available from the author. This remote shell daemon includes a password feature added for increased security. The password feature differs slightly from that of the `rexec` daemon. The `rexec` daemon prompts the user for a password to be entered from the command line after the `rexec` command is sent. The new password

feature in the `rshd` is read from a file on the machine sending the `rsh` request. This enables the `rsh` command to be executed from within a script.

2.2 Review of Remote Execution in Unix

Unix platforms have been used to create remote distribution programs for many years [15] [26]. Unix has been used partially because it enables easy remote execution through the standard `rsh` and `rexec` commands (discussed in section 2.1). Methods for remote distribution in Unix vary greatly. The associated levels of complexity and ease of use also differ greatly between tools.

2.2.1 Xdistribute

Xdistribute [19] is a tool designed to capture the idle CPU cycles of a network of Unix workstations. Xdistribute allows users who require large amounts of processing time to distribute their jobs to remote hosts. However, it is targeted at users who do not have the administrative support required to setup and install a full-fledged process distribution system. Xdistribute achieves this through the use of standard Unix remote shell (`rsh`) and remote copy (`rccp`) commands. It enables programs to be distributed to a list of hosts. If a job is not completed, it will restart the job, from the beginning, on another machine in the list. The type of processes that can be executed using Xdistribute is somewhat limited. The program can only execute monolithic, independent jobs that require no coordination. However, it enables a group of users to execute jobs that are not

able to use other, more complex, systems. Xdistribute enables three types of job distribution: preamble, user, and postamble. Preamble distribution is used to execute a program once on each machine in the list, prior to user jobs. This can be used to perform any customization of the remote machines necessary to allow execution of the user jobs. User distribution executes the job once on one machine in the list. This enables execution of the jobs that users have set up to be executed remotely. Postamble distribution runs each program once on each machine, much like preamble distribution. It can be used to clean up each machine after the user jobs have been completed.

Probably the most significant advantage of Xdistribute is that almost any user, running Unix, can utilize the program. The user is not required to have administrator support nor a central machine to be maintained. Xdistribute is simply executed on the machine of the user who wants to distribute jobs, and it uses standard remote shell commands to achieve execution. However, Xdistribute does require some configuration of the local machine to enable execution to occur. Xdistribute also requires installation of the following software packages to enable execution [20]:

- Tcl 7.4 or higher – used for the graphical user interface
- Tk 4.0 or higher – used for the graphical user interface
- Perl 5.001 (or higher) package with all sub-libraries – required for the process server and process monitor
- Expect package – used for communication with the process server

Another advantage of Xdistribute is that it has a graphical user interface (GUI), which shows the user buffers representing the status of the execution. Figure 1 shows the main screen of the Xdistribute tool. The GUI is simple, but it provides a visual representation of the system to the user. It illustrates the number of jobs are waiting to be

executed, the number that are currently executing, and the number that are completed. The completed jobs are broken down into the ones that encountered an error and those that finished. It also displays the status of the remote machines, including the total number of machines and the number unavailable, idle, busy, and active.

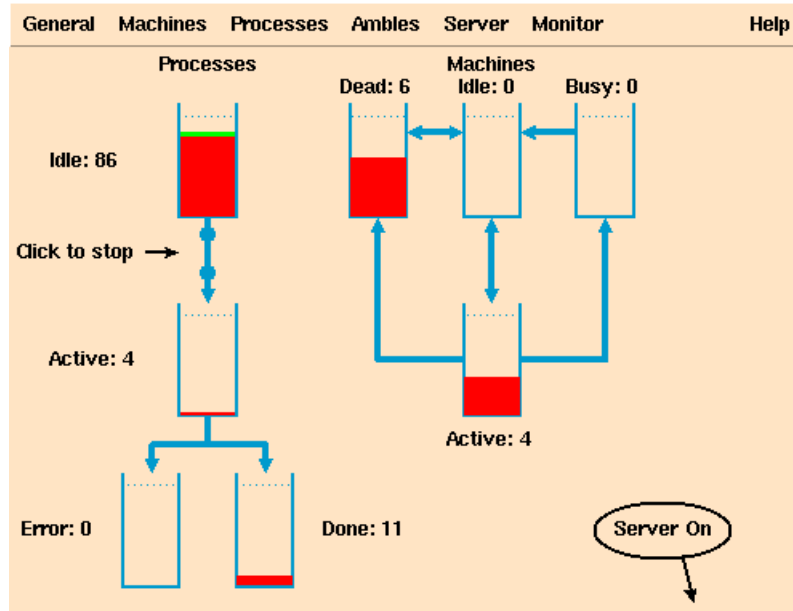


Figure 1: Main screen of Xdistribute tool[20].

2.2.2 Condor

Condor [15] is a scheduling system that utilizes the idle CPU cycles of a network of workstations. Condor is designed on the premise that in a group of workstations the majority of users under utilize the resources of their system. However, a few users have a higher processing demand than can be fulfilled by their workstation. Condor organizes the workstations into a pool and enables the users in need of processing time to execute jobs on the idle machines of other users within the pool. Jobs are started on the idle

machines then are migrated to another workstation when the user returns. Checkpoints are used to migrate the job and start execution at close to the same point as where it was interrupted on the prior machine. These checkpoints save the status of the execution as it progresses. When the owner of the remote machine returns, the execution can be halted and restarted on another machine from the last checkpoint. This ensures that all jobs will eventually be completed with little work duplicated. It also encourages users to join pools and allow their resources to be shared by assuring them that it will not negatively impact their own productivity. Users are also prevented from monopolizing all free CPU cycles in the pool because Condor monitors the usage of the users and applies an Up-Down algorithm designed to ensure fair access to the resources by all users.

Condor is a fairly large program that occupies 20 Mbytes to more than 50 Mbytes of disk space depending on the platform and system where Condor is running. Condor also requires considerable installation, including configuration of global and shared file systems, installation of Condor daemons and libraries, and set-up of console devices and directories. [9]. A central manager must be chosen that will manage the pool of workstations; this machine will be responsible for matching each job with an available machine and migrating the jobs when necessary. The central manager will also receive updates from each machine in the pool about every five minutes. As a result, if this machine encounters problems, no new matches will be made and most Condor tools will stop working. This central manager requires the most configuration, but Condor must also be configured on each machine in the pool.

2.2.3 Akaroa2

Akaroa2 [26] is a project developed by the University of Canterbury in New Zealand. It is a redesigned and improved version of the Akaroa [18] project developed at the same university. Akaroa2, like Akaroa, implements an approach to reduce execution time of quantitative scholastic discrete-event simulations known as Multiple Independent Replication in Parallel (MIRP). MIRP, like other remote distribution schemes, is used to harness the computing power of a network of workstations. The usage of Akaroa2, however, is somewhat limited. It can only be executed on Unix platforms that support Berkeley-style sockets for inter-process communication. The programs Akaroa2 can execute must be sequential simulation programs written in either ANSI C or C++ and be capable of calling library routines written in C++ [1].

Akaroa2 is somewhat limited in its application while still being a very useful tool for reducing the execution time of said simulations. Simulations are generally analyzed sequentially; that is, they are run for a long period of time until the number of collected observances, or sample size, is sufficient to reduce the statistical error of the results below a certain threshold. Multiple Independent Replications in Parallel, or MIRP, employed by Akaroa2, creates multiple instances of the simulation to be executed simultaneously. Akaroa2 automatically collects the observations and calculates their mean value for each parameter being analyzed. Once the required precision is reached for each parameter, the simulations are stopped on all of the machines. Since multiple

instances of the simulation are run in parallel, with each simulation generating results, the time to gather a sufficient sample is reduced.

An improvement in the Akaroa2 project over the original Akaroa tool is the addition of a graphical user interface called Akgui, which enables users to view the list of available hosts, view the list of simulations currently running, and to enter new simulations to be executed.

2.3 Review of Remote Execution in Windows

Traditionally, distributed computing was only available on Unix platforms. Still only a few tools have been developed to capture the abundance of idle CPU cycles available from Windows machines.

2.3.1 Condor for Windows NT

Condor was initially developed for Unix and has been available on Unix platforms since 1988. Condor became available for Windows NT/2000/XP in the version 6.4.3 released in October of 2002. However, it has only been tested on Windows NT systems. Condor for Windows [8] has the majority of the features of the Unix version, discussed in section 2.2.1. However, it does not have the migration capabilities provided by checkpoints that are used when run in Unix. It can suspend execution when the user of a remote machine returns, but must restart execution from the beginning on the next machine, thus possibly executing much of the program twice. Since an interrupted job

will always be restarted from the beginning, unlike execution on Unix machines, jobs are not guaranteed to finish when executed on Windows machines. Also, Condor for Windows does not allow files located on a server to be accessed as it does in Unix. All jobs must be on the local disk. The version 6.4.3 release was the first to enable execution on Windows machines. This release allows the pool of workstations to be a mixture of Unix and Windows NT machines. The default behavior is to require jobs to be executed on a machine running the same operating system as the machine that submitted the job. Only one central manager is necessary, and it can be a Unix or Windows NT machine. Similar to previous versions designed for Unix platforms a central manager must be maintained and each machine in the Condor pool must be configured.

2.4 Review of GRID Computing

Grid computing [3] [10] is another method of sharing computing resources. Grid computing is an approach to distributed systems that shares resources over a local or wide area network. The specific focus, that underlies Grid computing, is coordinated resource sharing in a multi-institutional environment [12]. It attempts to combine all types of resources, including supercomputers and clusters of machines, from multiple institutions, into a resource more powerful than any single resource. Therefore, grid computing is a very large non-dedicated cluster. A cluster is a type of distributed processing system, which consists of a collection of interconnected computers used as a single computing resource [6]. The use of clusters of computers as a distributed resource is a very cost effective way to create supercomputing power. There are basically two types of clusters based on resource ownership. The first is dedicated clusters which consists of machines

that are reserved solely for use by the cluster system. An example of such a system is Beowulf, discussed in Section 2.5.2. The second are non-dedicated clusters. A Grid is a non-dedicated clusters; these clusters use idle CPU cycles of individually owned machines. The goal of Grid computing is to achieve flexible, secure, large-scale resource sharing. The meaning of the term Grid is an analogy to the electrical power grid. Researchers developing Grid Computing envision it becoming a resource much like electricity that users just “plug into”. Users would be able to access processor and software needs without consideration of where they are coming from, much like one is able to plug into electrical power without considering where and how it is being produced.

2.4.1 NetSolve

NetSolve [3] is a Grid Middleware system that uses Remote Procedure Call (RPC) to harness resources distributed by ownership and geography. It provides an environment that manages hardware and software resources and allocates the services they provide to programs and users [2]. NetSolve has two types of users. The first type users act as clients only, accessing existing NetSolve resources, but not adding resources to the system. The second group of users acts as servers and clients, enabling their computing resources to be used by other NetSolve users, in addition to accessing existing NetSolve resources. Client and server users can access the NetSolve system by downloading the source code and executable from the NetSolve website. The NetSolve system consists of a set of loosely connected computers connected through a local, wide

or global area network. NetSolve incorporates load balancing and scheduling strategies to allocate tasks evenly among servers. It also maintains statistics on the hardware and software capacities and current usage of the servers through its agents and allocates server resources accordingly. NetSolve attempts to assign each request to the resource that will provide the quickest results; if however the assignment fails, NetSolve will try reassigning it to every appropriate server until the request is completed. NetSolve uses standard Internet protocols like TCP/IP sockets. This makes the program available on most variants of Unix and some features are available on Microsoft Windows 95 and higher.

2.4.2 Condor-G

Condor-G [10] is based on the principle that the capacity of computational and storage resources have increased at a rate sometimes even exceeded Moore's Law, yet few researchers or engineers have tried to move beyond the confines of their own institution to utilize the vast computational resources that are available. Condor-G is targeted at utilizing the resources available from different institutions to speed the processing of simulations, large-scale optimization, and image processing, among other computationally intensive tasks. Condor-G combines the advantages of two projects, the Globus Toolkit and the Condor System, to create a tool that can overcome the barriers between institutions. Globus Toolkit [11] is designed to provide security, resource discovery and resource access in a multi-domain environment. Condor, as discussed in section 2.2.1, provides management of processes and harnessing of resources on a pool of

workstations. Combining the benefits of these two methods allows users to harness resources of a multi-domain environment as if they belong to a personal domain.

2.4.3 Grid Computing in Use

Several projects are underway which use Grid computing to harness resources of Internet connected computer to analyze large amounts of data. Berkeley Space Sciences Laboratory's SETI@home project, Search for Extra Terrestrial Intelligence [27], is one such project. It is believed by SETI@home researchers that life-forms on other planets may be sending radio waves into space in an attempt to communicate with life on other planets. SETI@home uses Grid computing to harness the CPU cycles needed to analyze radio waves from space. By analyzing radio waves gathered from space, researchers hope to detect communication from other planets. The SETI@home project divides gathered signals into independent sections, by radio frequency and section of the sky, that can be analyzed separately. These sections, which are small enough to be executed on a home PC, are distributed to individual users to be analyzed [13]. The analysis is run as a screen saver on idle computers. When the user next logs onto the Internet the results are uploaded to the SETI@home site. This uploading is transparent to the user. An example of the SETI@home screen saver is shown in figure 2. The screen saver displays information about the status of the analysis to the user. The Data Analysis on the upper left pane of the screen saver is dynamically updated as the analysis proceeds, supplying the user with a graphical image of the work being done on their machine.



Figure 2: The SETI@home project's screen saver.

Another project, which uses Grid computing to analyze data, is Stanford University's Folding@home project [21]. The Folding@home project is used to analyze protein folding. Proteins are composed of long chains of amino acids. The Human Genome Project has mapped the order of these amino acids; however, the order alone offers limited insight into protein activity. Proteins fold into different shapes that are believed to be related to their function. It is suspected that misfolded proteins contribute to certain illnesses, including cancer and Alzheimer's disease. The Folding@home project is studying the protein's structural properties to understand protein folding. This involves extensive computing resources not available at a single institution. Therefore, much like the SETI@home project, the Folding@home project is distributed to Internet connected computers as a screen saver that analyzes protein folding while the machine is idle. Figure 2 shows the Folding@home screen saver. The image of the protein shown changes as the protein is being analyzed.

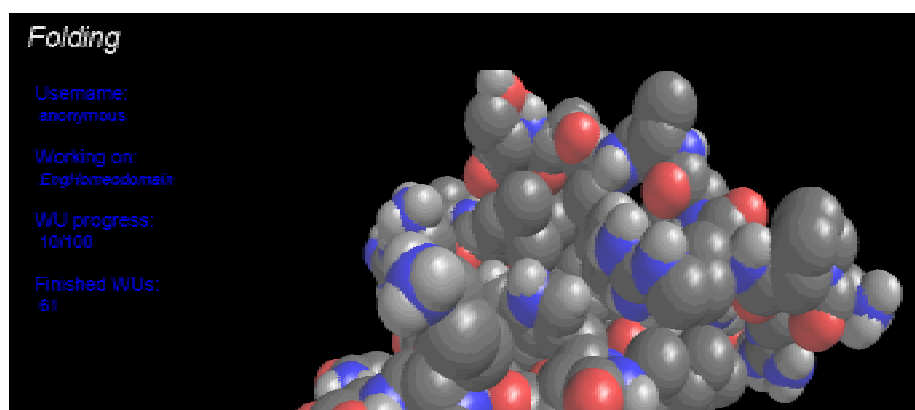


Figure 3: The Folding@home project's screen saver [23].

2.5 Distributed Operating Systems

Distributed operating systems [24] are yet another way of enabling computing resources to be distributed to users based on need. When distributing programs with a distributed operating system, it is the system, not the user that decides where a process should be executed. A distributed operating system combines machines in a way that they are viewed as a whole and not as a set of separate machines. With operating systems like Windows and Unix, a program is executed on the local computer, unless the user gives an explicit command to run it remotely. When a program is executed in a distributed operating system, it is distributed by the system, not the user. The system decides the best place for the program to execute, and it is transparent to the user where the program is actually executed.

2.5.1 Ameoba

Amoeba [24] is an example of a distributed Operating System developed at the Vrije Universiteit in Amsterdam. Amoeba has four design objectives: distribution, parallelism, transparency and performance. Amoeba combines multiple machines in a way that it appears to the user to be a single system. The user does not know which machine in the system is executing their program. Amoeba is a parallel system; that is, a single program can utilize multiple processors. Amoeba, like all operating systems, also strives for efficient performance. It does this by optimizing the communications system. Amoeba, however, is not designed to run on just any system. It is designed for a system on network-connected workstations. These workstations can operate as terminals or contribute to the processor pool. These workstations are connected to a set of main processors and to specialized servers. Amoeba schedules processes to execute on idle processors or processors with the lightest loads. It uses Remote Procedure Calls (RPCs) for communications between threads.

2.5.2 Beowulf

A Beowulf [4] is a cluster of computers configured to enable parallel processing. A Beowulf can be seen as a distributed operating system; however, Beowulf is not a software package but rather a means of achieving parallel processing using a free operating system like Linux on a cluster of machines. In addition to a free operating system there are several pieces of basically standard parallel computation software that are useful, but not essential, for building Beowulfs [22]. They include MPI and PVM, which are software systems that enable development (in Fortran and C) of message-

passing parallel programs that run on a cluster. Many other software programs can also be used with Beowulfs.

A Beowulf system is generally seen as an inexpensive means of developing a supercomputer using a cluster to achieve parallel computing. However, it does not speed the execution of all processes. To reduce the execution time of an existing program it must be split into parallel tasks that communicate using MPI or PVM or other software packages. Even multi-threaded software won't automatically experience a speedup because multi-threaded software assumes shared-memory which is not available in a Beowulf system. A Beowulf can also reduce the overall execution time of programs that need to be run numerous times with different input files, using a shell script or remote distribution tool for distribution.

Unlike other methods of remote distribution discussed, the machines used in a Beowulf cluster are not borrowed they are dedicated to cluster computing. A typical Beowulf consists of a number of identical commercial off the shelf (COTS) computers used as nodes [5]. The nodes are not required to be identical but often the best results are achieved with identical nodes; this is because splitting up applications optimally generally gets a little harder on inconsistent nodes. These nodes are connected by a COTS fast private network, often a fast Ethernet switch. A diagram of a Beowulf cluster is shown in Figure 4(a). The nodes are served and accessed from one or more common head nodes. Typically, a single head node is used. The monitor and keyboard of this head node may be connected to all nodes through monitor/keyboard switch boxes (shown in Figure 4(b)). Monitor/keyboard switches are not essential but can make the system easier to maintain.

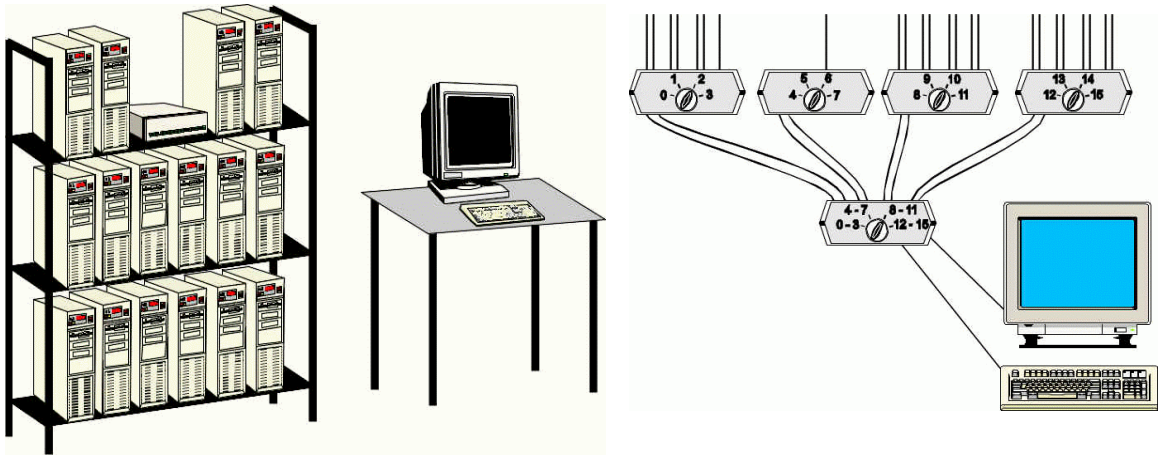


Figure 4. ((a) on left) A Beowulf system setup; showing a rack of identical COTS computers and a single monitor and keyboard. ((b) on right) A COTS monitor/keyboard switch setup, connecting the single monitor and keyboard to the cluster of Beowulf machines through 4-way switches[14].

2.6 Review of Existing REMOTE tool

REMOTE [7] is a tool for automatic remote execution of simulation models and other console mode programs on Windows PCs. The REMOTE++ tool addressed in this thesis is based on this tool. Much like many of the distribution programs created for Unix, REMOTE was designed to harness the CPU cycles of idle machines to speed the execution of large programs. REMOTE uses PIR of programs to achieve this reduction in execution time. Therefore, REMOTE is designed to execute programs that are independent of interaction, and small enough to be executed on a single machine, but require multiple executions. Thus REMOTE is well suited for the simulation programs that it is designed to distribute.

REMOTE assigns jobs from a job list to hosts in a host list. A job in the job list consists of an executable file, an input file, and an output file. All three files are transferred to the assigned remote machine before execution, and the output file is transferred back to the master host at the completion of execution. All communications in REMOTE are implemented using the Winsock interface. As a result of this sockets interface, the REMOTE tool is complex. It also has some known timing-related bugs. For example, if two processes executing on remote machines finish at the same time and both transfer an output file to the master machine simultaneously, only one output file is received by the master host.

REMOTE consists of two programs; one is a `remote.exe` program to be executed on each of the machines in the host list, the other is a `master.exe` program to be executed on the master host. `Remote.exe` is a single program small enough to be distributed via e-mail to each of the remote hosts. No configuration of the remote computers is required. When a user wants to enable a master computer to utilize its processor power, the user simply executes a single command from the console mode (dos prompt). This command specifies which host will be allowed access by supplying the hostname of the master machine. The master machine can then assign a job to that remote host.

REMOTE includes a status feature that can be executed on the master machine to monitor the progress of the jobs. It will display the jobs currently being executed and the percent of the job that has been completed. This status feature can run in a separate window on the master machine; thus allowing the user of the master machine to easily detect any problems with the execution of the jobs.

CHAPTER 3 GOAL OF REMOTE DISTRIBUTION

The basis for remote distribution is the combination of two main factors. First, the need to reduce execution time required to complete large processor intensive experiments and simulations. Second, to maximize the usage of processing resources that are often under used. Remote distribution combines these two goals by harnessing the idle CPU cycles of network connected PCs to speed the execution of large programs. Time based parallelization, as a means of remote distribution, specifically targets experiments and simulations, which require multiple executions to gather results.

3.1 Remote Distribution System:

Figure 5 shows the basic structure of a remote distribution system. The system consists of a single master computer and two or more idle remote machines. The master machine distributes executables and, if applicable, their input and output files to remote PCs. The jobs are executed on the remote machines and, at the completion of execution, the output is on the master machine. Once a job is completed, the master computer can send another job to that machine. The master machine is idle while the jobs are being executed remotely, so it is free to complete other work. The master and remote machines must be network connected throughout the process to enable the transfer of jobs and

output. The remote machines must be idle to allow their resources to be harnessed without burdening the owner of the remote PC.

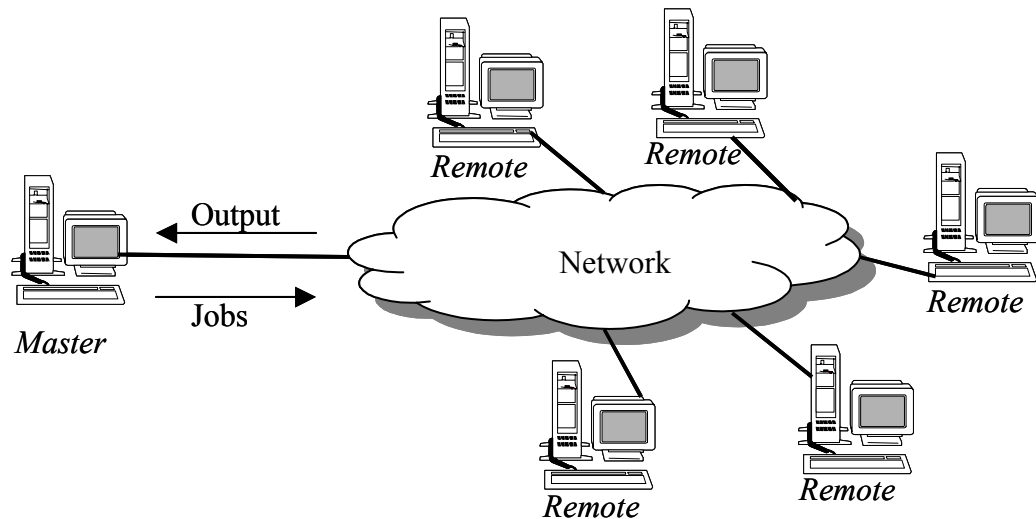


Figure 5: Diagram of Remote distribution system. Master distributes jobs to remote hosts. Remotes execute the jobs. The output is on the Master at the end of execution.

3.2 Requirements of Distribution Tools

Remote distribution tools rely on the idle CPU cycles of machines for execution. This idle CPU time is primarily available during evenings and weekends when users will not be returning to their machines quickly. This allows resource utilization to be maximized; however, failures on these unmonitored machines can result in significant time setbacks. The remote machines are often in locations distant from the master machine. So, resetting machines in the case of a failure or collecting results manually is often not an option. Thus, remote distribution tools must be very stable, independent of user interaction and not prone to failures. In order to ensure successful remote execution, the distribution tool must meet the following requirements:

1. Distribution and execution of programs must be automatic and not require any manual interaction.
2. The remote distribution program must be simple to allow for easy maintenance and modification.
3. Output files must be available on the master PC at the completion of execution of each job.
4. A single process must be distributed to each remote machine at a time.
5. Once a job completes, the next job must be sent to that remote host until all jobs are executed.
6. Each job must be executed only once.
7. The failure of a job to complete must be detected. It must not stop the distribution of the remaining jobs.
8. The failure of a remote host must be detected. The job must be reassigned to another host. Future jobs should not be assigned to the failed host.
9. Status and error messages must be displayed at the master PC to allow diagnosis of run-time errors.
10. A log file should indicate which jobs failed to execute and which hosts were invalid.

3.3 The Jobs:

Remote distribution cannot be used with all processes. The executables must be stand-alone processes. They can read input and write output, but cannot require any

interaction with other processes. Since reduction in time is the main goal of distribution, no modification of the executable should be necessary to enable remote execution. Programs will be executed in parallel; so, the results of one process cannot be required as input into another process. There is overhead involved in distributing executables and receiving results; therefore, the time required to complete execution of each job should be long enough to make this overhead insignificant. Processes that require only a few seconds to execute may take longer to distribute than to execute, and may not experience significant reduction in overall execution time. Simulation programs like the one used in the evaluation in section 5 are ideal for use with remote distribution tools.

CHAPTER 4 REMOTE++ DESIGN AND IMPLEMENTATION

REMOTE++ implements time based parallelization of programs through Parallel Independent Replications; it is, therefore, a remote distribution program designed to meet the requirements presented in chapter 3. The details of this design will be discussed in this chapter. Section 4.1 describes the files that comprise REMOTE++. Section 4.2 describes the structure of REMOTE++. Section 4.3 describes the input and output of REMOTE++. Section 4.4 provides the users' view of REMOTE++.

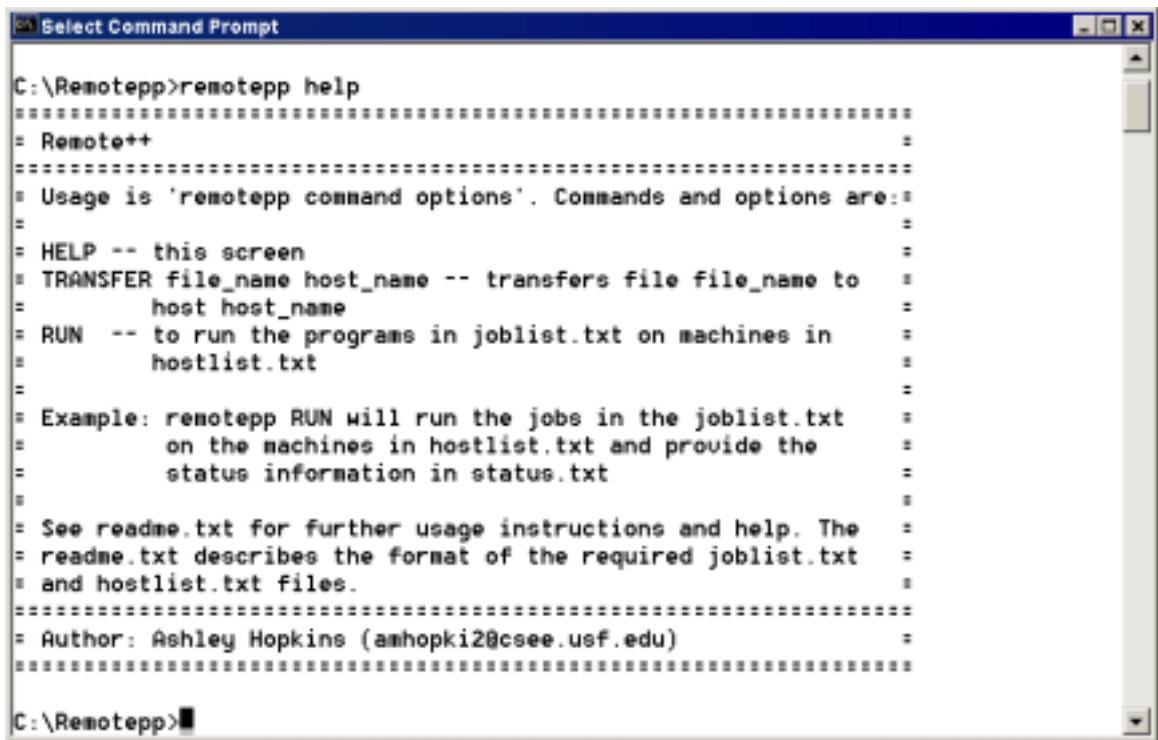
4.1 REMOTE++ Components

The REMOTE++ program is distributed as a package including a `readme.txt` file, an executable, a make file, and the code files from which the executable was built. Appendix A contains the `readme.txt` file and the code for these components. The REMOTE++ executable is built from the following files:

- `remotepp.h` – defines constants and prototypes
- `remotepp.c` – main program which launches the run, transfer and help functions
- `run.c` – runs the jobs in `joblist.txt` on the hosts in `hostlist.txt`
- `transfer.c` – transfers a file from the master to a remote host
- `help.c` – displays the REMOTE++ help screen

REMOTE++ was designed with simplicity as a main goal to allow for easy use, maintenance, and modification of the program; it consists of less than 400 lines of straight C code. REMOTE++ is a command line program. It is launched by executing “remtoepp cmd”, where cmd is run, transfer *filename hostname*, or help. From the main function contained in the remotepp.c file, the arguments are checked and the appropriate function is called.

An invalid argument, no argument, or “help” supplied to the remotepp command will call the help function, contained in the help.c file. The help function then displays the help screen shown in figure 6 below. This screen informs the user of the proper commands and options required to run REMOTE++.

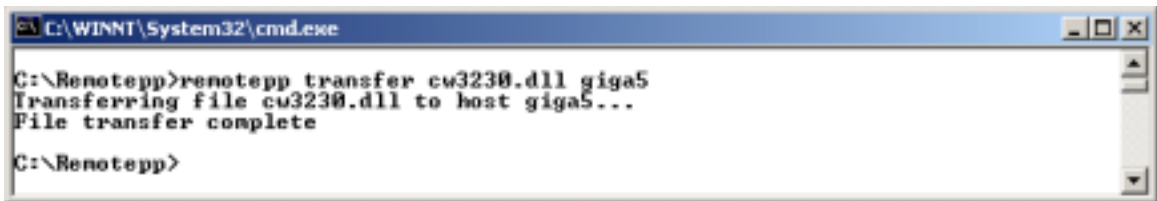


```
C:\Remotepp>remotepp help
=====
= Remote++
=====
= Usage is 'remotepp command options'. Commands and options are:
=
= HELP -- this screen
= TRANSFER file_name host_name -- transfers file file_name to
=         host host_name
= RUN -- to run the programs in joblist.txt on machines in
=         hostlist.txt
=
= Example: remotepp RUN will run the jobs in the joblist.txt
=         on the machines in hostlist.txt and provide the
=         status information in status.txt
=
= See readme.txt for further usage instructions and help. The
= readme.txt describes the format of the required joblist.txt
= and hostlist.txt files.
=====
= Author: Ashley Hopkins (amhopki2@csee.usf.edu)
=====
C:\Remotepp>
```

Figure 6: Help screen for REMOTE++, displayed by remotepp help.

To transfer a file to the remote machine, the argument “transfer” will be used, this will call the transfer function, contained in the transfer.c file. The proper syntax is

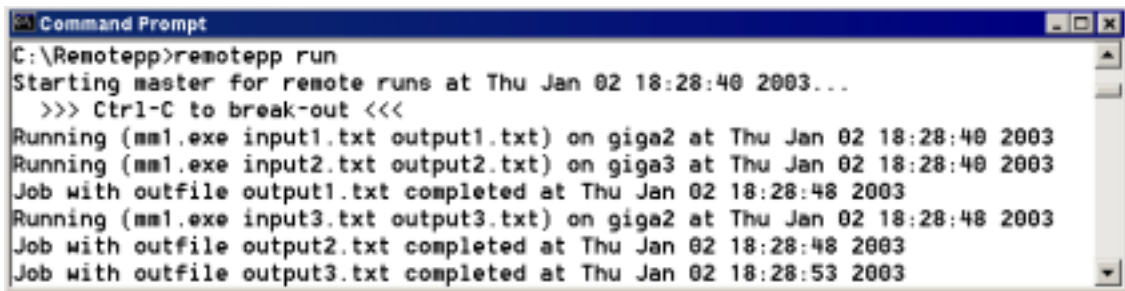
“remotepp transfer *filename hostname*”, where *filename* is the name of a file on the master machine to be transferred to the remote machine named *hostname*. The transfer function uses a simple remote copy (rcp) command to transfer the file. This command is used if supplemental files are needed on the remote machine to enable execution of the jobs in the job list. For example, the transfer command could be used to transfer a .dll file required to execute a simulation. Using transfer in place of a standard rcp command will ensure that the file is transferred to the proper directory, where the REMOTE++ program can then access it. Figure 7 shows a sample execution of the transfer command.



```
C:\WINNT\System32\cmd.exe
C:\Renotepp>remotepp transfer cw3230.dll giga5
Transferring file cw3230.dll to host giga5...
File transfer complete
C:\Renotepp>
```

Figure 7: Sample execution of transfer command. Transfer of file cw3230.dll to remote host giga5.csee.usf.edu.

The run function, contained in run.c, is called when the “run” argument is supplied to the remotepp command. This function is the focus of the REMOTE++ program, which enables remote distribution of the jobs in the joblist.txt. The run function uses threads to execute a series of rcp and rsh commands that enable remote execution of these jobs. The run function is described further in the next section. A sample execution of the run command is shown in figure 8.



```
Command Prompt
C:\Renotepp>remotepp run
Starting master for remote runs at Thu Jan 02 18:28:40 2003...
>>> Ctrl-C to break-out <<<
Running (mm1.exe input1.txt output1.txt) on giga2 at Thu Jan 02 18:28:40 2003
Running (mm1.exe input2.txt output2.txt) on giga3 at Thu Jan 02 18:28:40 2003
Job with outfile output1.txt completed at Thu Jan 02 18:28:48 2003
Running (mm1.exe input3.txt output3.txt) on giga2 at Thu Jan 02 18:28:48 2003
Job with outfile output2.txt completed at Thu Jan 02 18:28:48 2003
Job with outfile output3.txt completed at Thu Jan 02 18:28:53 2003
```

Figure 8: Sample execution of run command. Run of mm1.exe on remote hosts giga2.csee.usf.edu and giga3.csee.usf.edu.

4.2 Remote Distribution Structure in REMOTE++

REMOTE++'s run function is a script that runs standard remote shell (rsh) and remote copy (rcp) commands within Windows threads to enable Parallel Independent Replications of processes. It is within these threads that remote distribution is achieved. The flowchart in Figure 9 illustrates the structure of the run function, and the flowchart in Figure 10 shows the structure of the run thread, which is started from the run function for each job in the job list.

The first step taken by REMOTE++ in running the jobs is to verify each job in the `joblist.txt` file. The input file, output file, and executable file are opened to verify their existence. If all three files are valid the job will be placed in the job queue. The job will also be assigned a host if there is a host in the `hostlist.txt` that has not been assigned to another job. The host will then be assigned to the host queue. If all hosts have been assigned to a job, the host field for the remaining jobs will be left empty and assigned later. In the case where one or more of the files are not valid the job will be marked invalid and will not be executed. If there are unassigned hosts remaining in `hostlist.txt` after all valid jobs have been assigned to the queue, the hosts will be added to the host queue; these hosts will only be used if one of the assigned hosts is invalid.

The jobs in the queue that have been assigned a host are ready to be run. For each of these jobs, a run thread will be started to allow execution of the job. If there are jobs that have not been assigned a host, they will wait for a job to complete and that host to become available. As jobs complete, the host will be reassigned to another job. That job

will be ready to run, and a thread will be started. Once all jobs in the job list have completed, the run function will end.

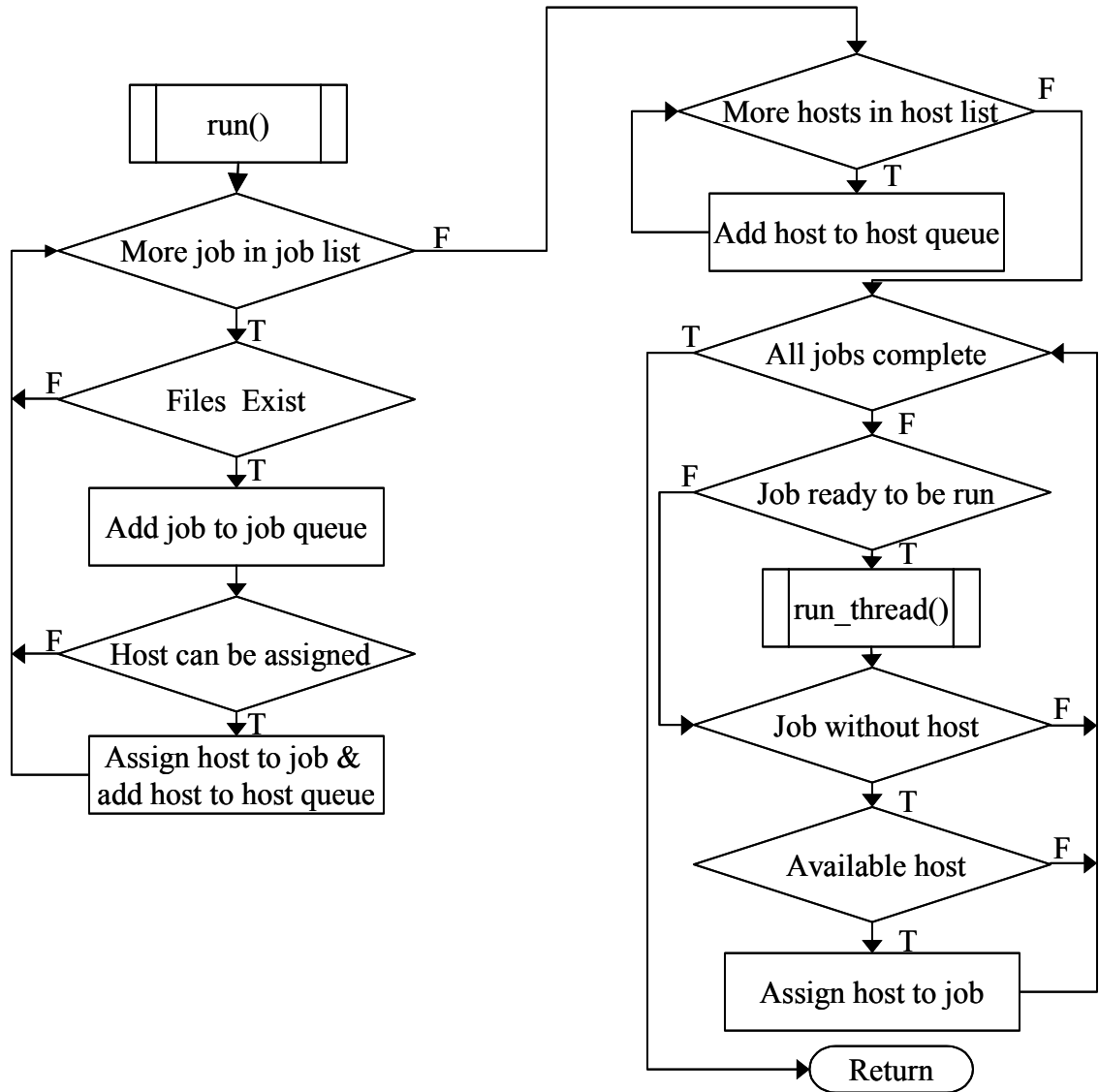


Figure 9: Flowchart of run function.

Figure 10 diagrams the run thread of REMOTE++. One thread will be started from the run function for each valid job in joblist.txt. The threads will be used to execute the jobs remotely in parallel. First, the executable will be remote copied (with an rcp command) to the assigned host. If the rcp is successful, then the host is valid, and the execution of the job will continue. Otherwise, the host will be marked invalid, the thread

will end, and the job will be placed back in the queue to be reassigned to the next available host. If the host is valid, the input/output method of the job will be determined. If the input/output method is “std”, then a remote shell (rsh) command is executed, and the job is run on the remote host with the input and output redirected from the master PC. If the input/output method is “file”, the input and output files are copied (with rcp commands) to the remote machine, and the job is then executed with an rsh command. After execution, the output file must then be remote copied (rcp) back to the master machine. Before the thread ends, the host is marked available for future jobs.

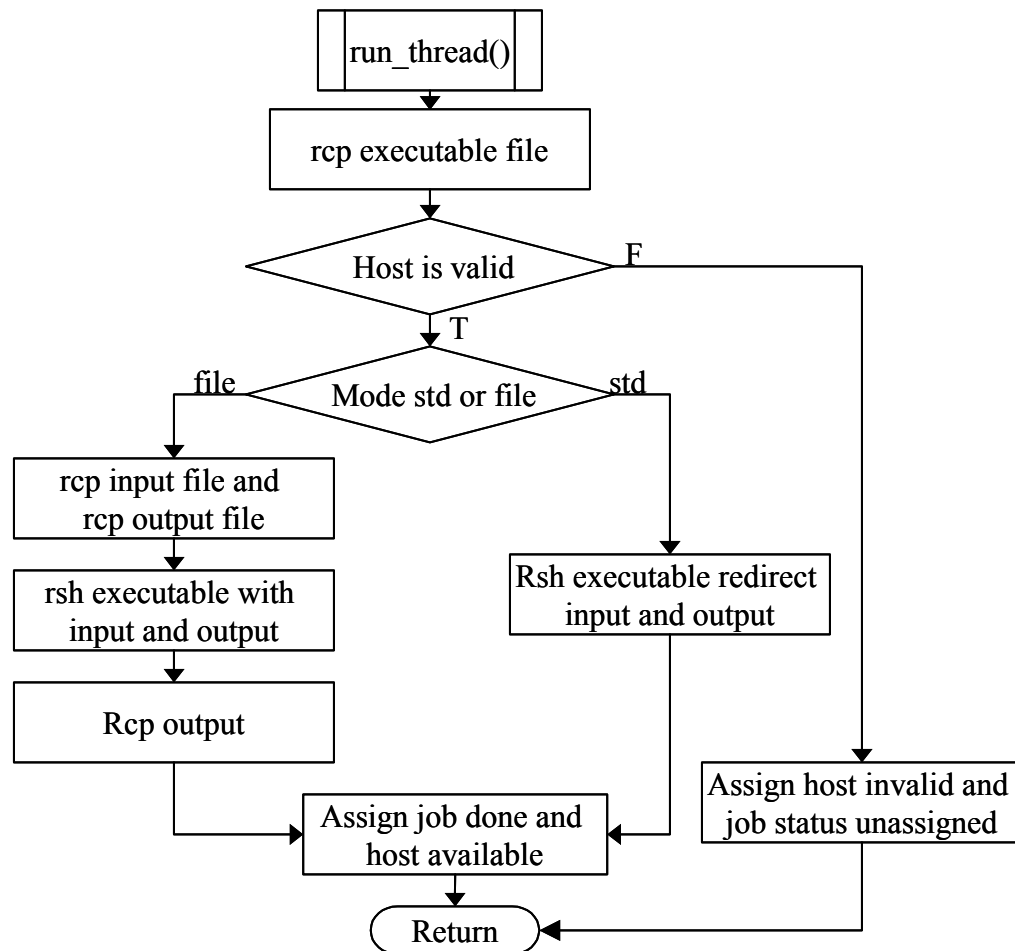


Figure 10: Flowchart of run thread.

4.3 REMOTE++ Input and Output

There are two methods of input and output recognized by REMOTE++, these are “classic” and “new”. The “classic” or “file” mode uses input and output from files as was done with the original REMOTE tool. The “new” or “std” mode is an addition in REMOTE++, which allows execution of programs which read input from standard input and write output to standard output. To enable processes that read from standard input to be executed independently of interaction with a user, all input must be written into a file in the sequence that it will be read. Redirection can then be used for input into and output from these processes. Different command sequences are used within REMOTE++ to support execution of these two file methods.

Figure 11, below, shows a sample command sequence for each method. These command sequences are executed by the REMOTE++ program for each valid job in the `joblist.txt` file. These commands sequences are not executed by the user. If the “file” method (Figure 11(a)) is used, the executable file (`mm1.exe`), input file (`input1.txt`), and output file (`output1.txt`) are transferred to the `temp` directory of the remote machine (host name `giga1.csee.usf.edu`). This is done with a series of three binary file copies using `rcp` commands with the `-b` option, each of which transfers one file. It is necessary to copy all three files to the remote machine because they must be located on that machine to allow execution of programs that read directly from files. The job is then executed, using an `rsh` command. After execution, the output file is transferred back to master (to the `remotep` directory) with an `rcp` command. This is done to ensure that the output file is on the master machine after each job completes.

When the “std” method is used the command sequence in Figure 11(b) is executed by REMOTE++. In the “std” method only the executable (mm1.exe) is copied to the remote machine (giga1.csee.usf.edu), with a single rcp command. The input (input1.txt) and output (output1.txt) files remain on the master machine. Redirection of input and output from the master PC is then used in the rsh command to read from and write to these files. Since the output file remains on the master machine, no transfer of files is necessary at end of execution. The “file” method has more overhead in transfers (four rcp commands) than does the “std” method (one rcp command), which uses files stored on the master machine. This can result in a greater reduction in overall execution time of the experiment if the file method is “std” than if the file method is “file”.

- (a)

```
rcp -b /Remotepp/mm1.exe giga1.csee.usf.edu:/temp/  
rcp -b /Remotepp/input1.txt giga1.csee.usf.edu:/temp/  
rcp -b /Remotepp/output1.txt giga1.csee.usf.edu:/temp/  
rsh giga1.csee.usf.edu c:/temp/mm1.exe  
rcp -b giga1.csee.usf.edu:/temp/output1.txt /Remote/
```
- (b)

```
rcp -b /Remotepp/mm1.exe giga1.csee.usf.edu:/temp/  
rsh giga1.csee.usf.edu c:/temp/mm1.exe <input1.txt >output1.txt
```

Figure 11: Sample rsh and rcp command sequence for “file” method (a) and “std” method (b)

4.4 User View of REMOTE++

REMOTE++ was designed with minimizing the load on the user as a key consideration. Therefore very little configuration is needed to enable the REMOTE++ program to run. Each remote host must be a network-connected machine running Windows 9x or higher. There must be an rsh and rcp daemon running on each remote

host. A free rsh/rcp daemon is available from the author and Silviu C. Marghescu [16] or one can be purchased from a vendor. These rsh/rcp daemons require no configuration to enable execution. However, a .rhosts file with the host name of the master machine should be setup to provide security. A c:/temp/ directory must also exist on each remote host. Each executable, input and output file copied to the remote host will be saved in the c:/temp/ directory to allow easy execution and “clean up” of files on the remote machines. Files transferred with the transfer command will also be stored in the c:/temp/ directory to ensure availability to the jobs executed by the run command.

The configuration of the master machine is also minimal. The user must create three files: joblist.txt, hostlist.txt, and status.txt. The joblist.txt file contains the list of jobs. As seen in the sample joblist.txt file shown in figure 12(a), each job must consist of an input/output method keyword (“file” or “std”), followed by the name of the executable file, the name of the input file, and the name of the output file (mm1.exe, input.txt, and output1.txt respectively in the example). The hostlist.txt file must contain a list of the hostnames of the remote hosts. A sample hostlist.txt file is shown in Figure 12(b). The status.txt file can initially be empty. It will contain information about the success or failure of each job and each host at the completion of the execution. A c:/remotepp/ directory must also exist on the master machine to contain all files used by REMOTE++. The REMOTE++ executable, joblist.txt, hostlist.txt, and status.txt must be contained in this directory. Additionally, all executable, input, and output files listed in the joblist.txt should be located in the c:/remotepp/ directory.

REMOTE++ is executed from the Window’s console. To execute the jobs in the joblist.txt on the hosts in the hostlist.txt, the command remotepp run must be executed

at the command line on the master machine. The progress of the job list, and any error messages will be seen on the master machine. At the completion of execution, the `status.txt` file contains the status of each job and the validity of each remote host. Figure 12(c) is a sample `status.txt` file that would be created after running the jobs in Figure 12(a) on the hosts in Figure 12(b). For each job, the file method is listed, followed by existence of each file (executable, input, and output). This information can be used to determine which jobs were completed. Following the jobs in `status.txt` is a list of the validity of each host. A host is considered valid if jobs could be executed on it and invalid if it was not available to execute jobs.

<p>(a) file mm1.exe input1.txt output1.txt file mm1.exe input2.txt output2.txt file mm1.exe input3.txt output3.txt</p>	<p>(c) Mode is classic. Executable file mm1.exe found Input file input1.txt found Output file output1.txt found</p>
<p>(b) giga2.csee.usf.edu giga3.csee.usf.edu</p>	<p>Mode is classic. Executable file mm1.exe found Input file input.txt found Output file output.txt found</p>
	<p>Mode is classic. Input file input.txt found Output file output.txt found</p>
	<p>giga2.csee.usf.edu is a valid host giga3.csee.usf.edu is a valid host</p>

Figure 12: (a) A sample `joblist.txt` file (b) A sample `hostlist.txt` file (c) A sample `status.txt` file with results from REMOTE++ execution with `joblist.txt` from (a) and `hostlist.txt` from (b) if all jobs and hosts were valid.

CHAPTER 5 EVALUATION OF REMOTE++

The performance of queueing systems is often studied using simulation methods. Queue simulations are regularly used to gather statistical information. To calculate such statistical information queue simulations must be executed numerous times with varying input. This makes queueing simulations ideal for Parallel Independent Replications, which will allow the simulations to be executed with multiple parameters on a pool of machines. Therefore, a queue simulation was utilized to evaluate the REMOTE++ tool.

5.1 Overview of Queues

A queue is a line or sequence of customers waiting to receive service. A customer is a broad term, which could vary from customers at a bank waiting in line for a teller, to computer programs in a job queue where they are spooled and wait for the computer to process them. The following features determine the behavior of a queue [17]:

- a. The distribution of time between arriving customers.
- b. The distribution of time to service a customer.
- c. The number of servers available to service the customers.
- d. The capacity of the queue; this is the number of customers that can be in the queue.
- e. The number of customers that are available to enter the queue (population size).

Queues are named according to the above features. The naming convention is $a/b/c/d/e$ where a , b , c , d , and e correspond to the features above. The distribution time (features a and b) can be Markovian (M), deterministic (D), r -stage Erlangian (E_r), k -stage hyperexponential (H_k) or general (G). The number of servers, the queue capacity, and the customer population (features c through e) are each represented by a number value. Commonly both the queue capacity and population size are infinite; when this is the case features d and e are often dropped.

The order in which the customers are served is determined by the queuing discipline or rules that determine where customers are inserted into and served from the queue. There are numerous queuing disciplines but several are very standard. One common queuing discipline is first come first serve (FCFS), also called first in first out (FIFO). This rule requires that the first customer to enter the queue is the first customer serviced. The last come first served (LCFS) queuing discipline, also called last in first out (LIFO) is another popular discipline. The last customer to join the queue with a LCFS discipline is the first customer serviced. Queuing disciplines can also be random and priority. In a priority queue the customers are identified by some feature, which determines the priority of that customer. Many other queuing disciplines are also possible.

5.2 M/M/1 Queuing System

An M/M/1 queue is one of the most commonly and heavily used models of a processing system. The queuing discipline of an M/M/1 queue is First In First Out

(FIFO). A diagram of an M/M/1 queue is shown in Figure 13. From the naming convention discussed in the previous section, it is clear that an M/M/1 queue (also denoted M/M/1/ ∞/∞) has the following features:

1. Markovian (exponentially distributed) inter-arrival of customers into the queue.
2. Markovian (exponentially distributed) service times.
3. A single server.
4. An unlimited queue capacity.
5. An infinite customer population

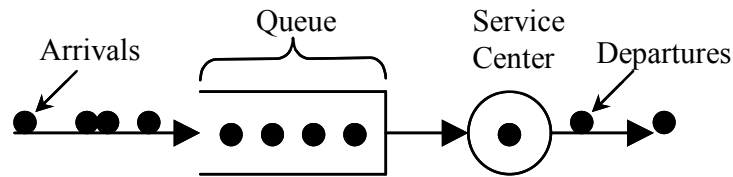


Figure 13: M/M/1 queue.

There are several variables that control simulation of an M/M/1 queue. The length of an M/M/1 queue is the number of jobs waiting to be serviced. At any time this length (L) can be determined by the arrival rate of items into the queue (λ) and the wait time in the queue (W). This relationship is $L = \lambda W$. The performance of an M/M/1 queue system is measured by its utilization. The utilization (ρ) is the fraction of the time that the system is busy. The utilization of the queue is the ratio of the arrival rate of items into the queue and the service rate of items leaving the queue. The queue length can be simply calculated from the utilization of the queue by $L = \rho/(1 - \rho)$. As the utilization of the queue increases and approaches 100%, the length of the queue approaches infinity.

5.3 Evaluation of REMOTE++

A relationship exists between the utilization of an M/M/1 queue, the length of that queue, and the simulation time. To determine this relationship, the M/M/1 simulation must be executed numerous times with utilizations approaching 100% and the related simulation times calculated. No output from one simulation is needed for any other simulation to complete. Therefore, each execution is independent. REMOTE++ was used to determine the relationship between the M/M/1 queue utilization and the simulation run time for mean queue length within a percent of the theoretical length. REMOTE++ was used to distribute simulations with increasing utilization to a pool of five remote machines. Three of these were Pentium III 866MHz machines and two were Pentium III 700 MHz machines. The M/M/1 simulation reads the target utilization and the desired margin of error (i.e., from the known theoretical length of the queue as calculated by $L = \rho/(1 - \rho)$) as input from a file. A 10% margin of error was used for this evaluation. Utilization was started at 1% and increased to 99.5%. Evaluation of each utilization was executed ten times using different seeds, and the results were then averaged to gather statistical results.

Using REMOTE++ to run these simulations on this pool of machines took about 20 minutes. When executed on a single 866 Mhz machine, the simulation took approximately 50 minutes. Therefore, it executed about two and a half times faster on five machines than it did on one machine. Since the simulations are independent, the overall increase was expected to be about five times faster when executed on five machines (about 10 minutes). The plot shown in figure 14, diagrams the execution times

of this experiment. The difference in actual and projected execution time is due to overhead in distributing the simulations. No overhead was considered in the predicted time; however, there is about seven seconds of overhead in executing each job in the job list, which has input/output method “file”. Since many of the simulations in the job list had low target utilizations, they took only seconds to execute. Therefore, this seven seconds of overhead was significant in the time needed to complete these simulations. If each of the jobs in the job list took longer to execute, it is projected that the overhead would be offset by the execution time. The speed up in this case would have been closer to five times.

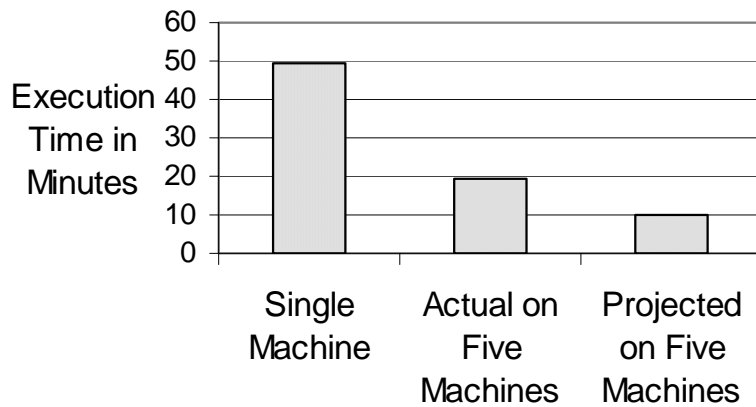


Figure 14: Execution times of M/M/1 simulation. Column 1 is actual execution time when executed one 866 MHz Pentium III computer. Column 2 is the actual execution time when executed on five machines (three 866MHz and two 700 MHz Pentium III computers). Column 3 is the projected execution time on five machines based on a 5 times speed up.

The plot shown in figure 15 conveys the results gathered in the M/M/1 queue simulation run with the REMOTE++ program. It illustrates the relationship between the target utilization and the simulation time for the M/M/1 queue for a target utilization between 90% and 99.5% and a margin of error of 10%.

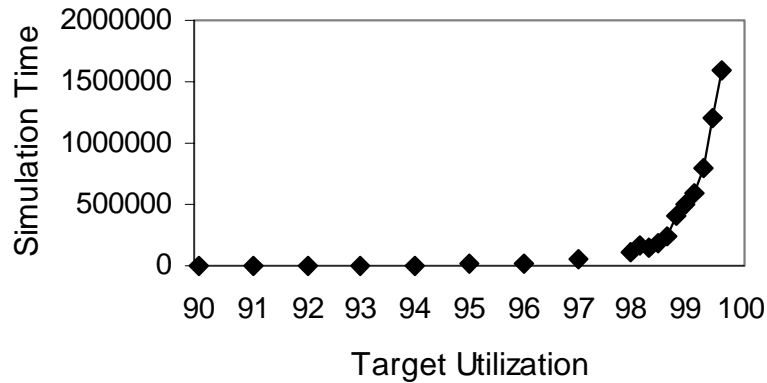


Figure 15: Simulations time versus target utilizations for an M/M/1 queue with margin of error of 10.0%.

Figure 16 shows the results illustrated in Figure 14 plotted with an order six polynomial trend line. The growth of the simulation time nearly mirrors that of the trend line at utilizations approaching 100%. The statistical confidence or R^2 relationship for this trend line is 0.9922 on a scale from zero to one. This is a reasonably close approximation of the trend in the results. However at utilizations very close to 100%, the simulation time grows slightly faster than the trend line. From these results one can conclude that as the target utilization approaches 100% the simulation time of the M/M/1 queue grows at a rate slightly faster than order six polynomial growth.

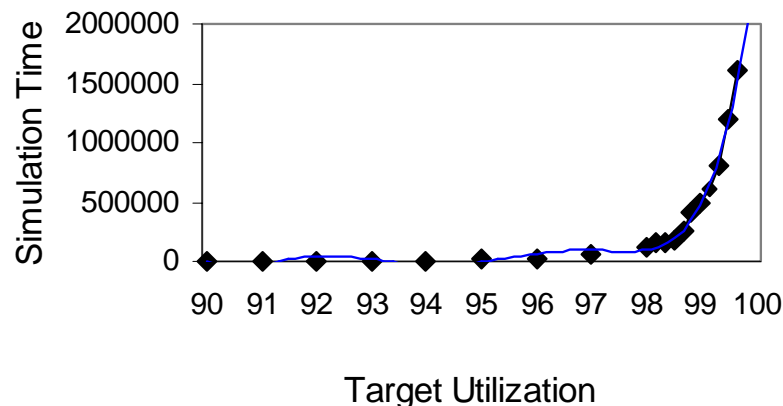


Figure 16: Order 6 polynomial trend line (blue) with the results of M/M/1 queue simulation (black) from in Figure 14. The simulation time of the M/M/1 queue grows at a rate slightly faster than order 6 polynomial growth.

CHAPTER 6: SUMMARY AND FUTURE WORK

Simulation programs require significant time to execute on a single machine. At the same time, many computer resources sit idle, wasting CPU cycles. The REMOTE tool was designed to harness these non-dedicated idle CPU cycles of Windows PCs to enable a reduction in the execution time of simulations. This is achieved through a process distribution and remote execution system. This system consists of a single master machine, which controls job distribution, and a group of idle remote machines, which execute the jobs. At the completion of execution, the results are on the master machine. REMOTE++ improves upon this tool. REMOTE++ eliminates the complex Winsock interface used by REMOTE and replaces it with standard remote shell (`rsh`) and remote copy (`rcp`) commands. These standard commands provide a straightforward, efficient way to communicate between the master and the remote machines. The use of these commands also reduces the program from approximately 1100 to 400 lines of standard C code; this reduction makes REMOTE++ easier to maintain and modify. REMOTE++ also expands the types of programs that can be executed without modification. The original REMOTE tool supported execution of programs that read from and write to files. REMOTE++ supports the execution of these programs, but also enables execution of programs that read from standard input and write to standard output. Only a single program (`remotep.exe`) is required to be executed on the master

machine. This program distributes and executes the jobs from the job list to the hosts in the host list. The remote machines need only run an rsh/rcp daemon to enable transfer and execution of programs. No REMOTE++ program need be run on the remote machines. REMOTE++ will skip a job if any portion of it is invalid (including executable, input or output). Jobs assigned to an invalid host will be reassigned to the next available host and no future jobs will be assigned to the invalid host. The `status.txt` log will list the validity of each file and host at the completion of execution. The REMOTE++ program, an rsh/rcp daemon, and the M/M/1 queue simulation model used to evaluate the program are freely available from the author as open source with no restrictions on use.

There are several known problems with REMOTE++ that are the subject of future work. There is a free rsh/rcp daemon available from Silviu C. Marghescu [16]. This free rsh/rcp daemon has been debugged to enable it to be used with REMOTE++. However, it does not enable redirection of input and output; therefore, only the “classic” or “file” mode can be used to execute programs. This limits distribution to programs that read from and write to files. The rsh/rcp daemons available from independent vendors allow redirection and hence support the “new” or “std” mode of input and output; however, they cost approximately \$40.00, which may limit use of the REMOTE++ program. A reliable, free daemon which supports redirection, therefore, needs to be developed to maximize the use of this program. REMOTE++ also requires that the user list whether each executable reads from and write to files or standard input and output (by declaring the input/output method as “file” or “std”). It also requires that both input and output be done with the same method. Future improvements on REMOTE++ should

enable automatic detection of the proper method to be used and allow any combination of methods to be used. This will further expand the set of programs that can be executed using REMOTE++. Currently, the REMOTE++ program relies on the rsh/rcp daemon to provide security through access lists or a password. Security features should be implemented as part of the REMOTE++ program that are not dependant on the rsh/rcp daemon and do not require configurations on the remote PCs. Finally, the progress indication available in the original REMOTE tool should be implemented in REMOTE++. The current methods notify the user of which jobs are currently executing and which jobs have completed, but do not provide a percent complete as was available from REMOTE. This functionality would make it easier for the user of the master machine to detect a problem with execution of a job.

REFERENCES

- [1] Akaroa 2 - Architecture and Implementation
http://www.cosc.canterbury.ac.nz/research/RG/net_sim/simulation_group/akaroa/architecture.html
- [2] Arnold, D., Agrawal, S., Blackford, S., Dongarra, J., Miller, M., Vadhiyar, S. User's Guide to NetSolve Version 1.4, (1995-2001)
<http://icl.cs.utk.edu/netsolve/documents/ug/nug.pdf>
- [3] Arnold, D.C., and Dongarra, J. The Netsolve environment: progressing towards the seamless grid. *International Workshop on Parallel Processing*, (Proceedings 2000), pages 199-206.
- [4] Becker D. Beowulf.Org The Beowulf Cluster Site <http://www.beowulf.org/>
- [5] Brown, R. Maximizing Beowulf Performance
http://www.phy.duke.edu/brahma/als_talk.pdf
- [6] Buyya R. High Performance Cluster Computing (Architecture, Systems, and Applications) <http://www.cs.mu.oz.au/~raj/cluster/>
- [7] Christensen, K.J. REMOTE: A Tool for Automatic Remote Execution of CISM Simulation Models. *Proceedings 35th Annual Simulation Symposium*, (2002), pages 134-142.
- [8] Condor for Microsoft Windows 4.0.
http://www.cs.wisc.edu/condor/manual/v6.1/5_Condor_Microsoft.html
- [9] Drakos, N., Moore, R. Condor Version 6.4.7 Manual. (2003)
<http://www.cs.wisc.edu/condor/manual/v6.4/>
- [10] Frey, J., Tannenbaum, T. Livny, M., Foster, I., and Tuecke, S. Condor-G: a computation management agent for multi-institutional grids. *Proceedings 10th IEEE*

- International Symposium on High Performance Distributed Computing*, (2001), pages 55-63.
- [11] Foster I., Kesselman C. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*,(1997) pages115-128.
- [12] Foster, I.; Kesselman, C.; Tuecke, S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, Volume 15, Issue 3 (2001) pages 200-222.
- [13] Korpela, E.; Werthimer, D.; Anderson, D.; Cobb, J.; Leboisky, M. SETI@home-Massively Distributed Computing for SETI. *Computing in Science & Engineering* Volume: 3, Issue: 1, (January/February 2001) pages 78-83.
- [14] Lindheim, J. Building a Beowulf System
<http://www.cacr.caltech.edu/research/beowulf/tutorial/beosoft/>
- [15] Litzxkow, M., Livny, M., and Mutka, M. Condor – A Hunter of Idle Workstations. *Proceedings of the 8th International Conference on Distributed Computing Systems*, (June 1988), pages104-111.
- [16] Marghescu, Silviu C. RSH daemon Version (1.6) for Windows NT/95
<http://home.us.net/~silviu/>
- [17] Molloy, M. Fundamentals of Performance Modeling Macmillan 1989.
- [18] Mota, E., Wolisz, A., and Pawlikowski, K. Comparing overlapping batch means and standardized time series under Multiple Replications in Parallel. Simulation and Modeling. Enablers for a Better Quality of Life. *14th European Simulation Multiconference* (2000), 43-48.
- [19] Petty K. F., McKeown N. Xdistribute: A Process Distribution System. University of California, Berkeley, Electronics Research Lab Research Report #UCB/ERL M96/67, (November, 1996). <http://www.path.berkeley.edu/~pettyk/pubs.html>
- [20] Petty, K. Introduction to Xdistribute. (1996)
<http://www.path.berkeley.edu/~pettyk/Xdist/>
- [21] Schreiner, K. Distributed Projects Tackle Protein Mystery. *Computing in Science & Engineering* Volume: 3, Issue: 1, (January/February 2001) pages 13-16
- [22] Sitaker K. Beowulf Mailing List FAQ, Version 2
<http://www.canonical.org/~kragen/beowulf-faq.txt>

- [23] Sezen, T. Folding@Home: An Opportunity for Teachers and Students to Participate in Scientific Research. <http://folding.stanford.edu/education/>
- [24] Tanenmaum, Andrew, and Sharp, Gregory. The Amoeba Distributed Operating System. www.cs.vu.nl/pub/amoeba/Intro.pdf
- [25] Unix man pages: rsh and rexec commands <http://www.csee.usf.edu/cgi-bin/man-cgi>
- [26] Yau, V., and Pawlikowski, K. AKAROA: A package for automating generation and process control of parallel stochastic simulation. *Australian Computer Science Communications*, Volume 15, Issue 1, Part A, (1993), pages 71-82.
- [27] Young, E., and Cliff, P. Distributed Computing – the SETI@home Project. *Ariadne* Issue 27, (March 2001), <http://www.ariadne.ac.uk/issue27/seti/>

APPENDICES

Appendix A: REMOTE++ Code

readme.txt

Readme file for REMOTE++ tool AMH (readme.txt - 3/18/03)

This is the readme.txt file for the REMOTE++ tool. The REMOTE++ tool runs on a "master" PC and distributes programs, exe files, to "remote" PCs. It does this in two modes the "classic" mode and "new" mode. The "classic" mode is used with programs that are written to read input from a file and write output to a file. This mode is called "classic" because it was used in the original REMOTE tool developed by Dr. Kenneth Christensen at the University of South Florida. The "new" mode is used with programs that are written to read input from standard input and write output to standard output. In the "new" mode only the executable file is copied to the "remote" PC and the input and output files can remain on the "master" PC. For the "classic" mode of REMOTE++ the executable file, input file and output file must all be copied to the "remote" PC and the output files are then copied back to the "master" PC after completion.

The following files are used:

```
remotepp.c - - - - main program
remotepp.h - - - - header file
remotepp.mak - - - - make file (for bcc32 -- make -fremotepp) for remotepp.exe
help.c - - - - - help function
run.c - - - - - send programs to remote functions
transfer.c - - - - transfer file to remote functions
```

Other files:

```
csim.ide - - - - - CSIM18 M/M/1 IDE file for Borland CSIM18 build of mml.c
mml.c - - - - - CSIM18 M/M/1 simulation source code
mml.exe - - - - - CSIM18 M/M/1 simulation (see header of mml.c)
cs3230.dll - - - - Needed for Borland C++ build CSIM18 simulations
mmlin.txt - - - - - An input file for mml.exe (see header of mml.c)
hello.c - - - - - Hello world source code
hello.exe - - - - - Hello world executable (see header of hello.c)
helloin.txt - - - - An input file for hello.exe (see header of hello.c)
readme.txt - - - - this file
```

SETUP INSTRUCTIONS:

The REMOTE programs allows for a set of console-mode programs to be distributed to multiple PCs for remote execution. One PC is the "master" (and runs remotepp.exe) and all other PCs are the "remotes" contained in hostlist.txt. All PC's are assumed to run Windows 9X or higher and be running a rsh daemon. (I suggest the Denicomp version available at www.denicomp.com as this was the daemon used in testing the program.)

The master PC contains a joblist.txt that lists the jobs to be run on the hosts listed in hostlist.txt. The joblist.txt contains two types of jobs "classic" and "new".

In the "classic" mode the program is assumed to be designed to read input from a file and write output to a file. "Classic" programs are programs that open files to read from and write to.

A "classic" job in joblist.txt would look like this:

```
file mml.exe input.txt output1.txt
```

In the "new" mode programs are assumed to be designed to read their input from standard input and write their output to standard output. Redirection is used by REMOTE++ to supply the input from and output to the files in the joblist.

A "new" job in joblist.txt would look like this:

```
std hello.exe input2.txt output2.txt
```

Appendix A (Continued):

In both the "classic" and "new" modes the arguments are the same. The first argument specifies the mode. The keyword "file" specifies the mode is "classic" and the keyword "std" specifies the mode is "new". The second argument is the executable file. The third argument is the input file containing the input to be used by the executable. The fourth argument is the output file. It is on the master PC upon completion of remote execution of the executable file and contains the output from the execution.

The joblist.txt file contains a list of "classic" and "new" jobs to be executed remotely.

An example joblist.txt is:

```
file mml.exe input1.txt output1.txt
file mml.exe input2.txt output2.txt
std hello.exe input3.txt output3.txt
file mml.exe input4.txt output4.txt
std hello.exe input5.txt output5.txt
```

The hostlist.txt file contains the hostnames of each remote machine that will be used to execute the jobs from the joblist.txt.

An example hostlist.txt is:

```
giga1.csee.usf.edu
giga2.csee.usf.edu
giga3.csee.usf.edu
```

At the completion of execution the status of the jobs and hosts will be in the status.txt file on the master machine.

An example status.txt is:

```
Mode is classic.
Executable file mml.exe found.
Input file input1.txt found.
Output file output1.txt found.
Job was run.

Mode is classic.
Executable file mml.exe could not be found.
Input file input2.txt found.
Output file output2.txt found.
Job was not run.

Mode is new.
Executable file hello.exe found.
Input file input3.txt found.
Output file output3.txt found.
Job was run.

giga1 is a valid host.
giga2 is a valid host.
giga3 is a valid host.

There are 2 valid jobs.
```

The status.txt file will contain an entry for each job in the joblist.txt and each host in the hostlist.txt. Each job entry specifies what the mode is, and if the executable, input, and output files were found. If all three files were valid then the job is run if not the job is skipped and marked "Job was not run". Following the job entries is an entry for each host specifying if it was valid. At the end of the status.txt is a count of the total number of valid jobs.

All files on the master machine used by REMOTE++ must reside in one common directory c:/Remotepp. This includes the joblist.txt, hostlist.txt, status.txt, executable, input and output files. A directory c:/temp must also exist on each of the remote PCs. This directory will contain all files copied to the remote machine to be used by REMOTE++.

Appendix A (Continued):

EXECUTION INSTRUCTIONS:

To run the jobs in the joblist.txt on the machines in the hostlist.txt, open a command window (a console or "DOS" window) and enter:

```
remotepp run
```

The progress of distribution will be shown on the screen. When each job is started and when each job completes a message including the time will be displayed to standard output on the master machine. All error messages will also be displayed at the standard output on the master machine.

To transfer a file file_name from the master to a remote PC host_name enter:

```
remotepp transfer file_name host_name
```

A message will be displayed if the transfer was successful.

To view the help screen for REMOTE++ enter:

```
remotepp help
```

Use the example programs hello.c and mml.c to run a sample execution using mmlin.txt and helloin.txt for input.

Appendix A (Continued):

remotepp.h

```
//===== file = remotepp.h ==
//= Include file for REMOTE++ =
//=====
//= Notes: 1) This file contains includes, constants, and function =
//=          prototypes for remotepp.c =
//=====
//= Contains:
//-----
//= History: AMH (11/01/02) - Genesis (from master.h and common.h by KJC) =
//=====

//----- Includes -----

#include <stdio.h>           // Needed for I/O functions
#include <string.h>         // Needed for string functions
#include <ctype.h>          // Needed for toupper()
#include <conio.h>          // Needed for kbhit()
#include <time.h>           // Needed for time stuff
#include <windows.h>        // Needed for windows stuff
#include <fcntl.h>          // Needed for file I/O constants
#include <sys\stat.h>       // Needed for file I/O constants
#include <io.h>             // Needed for open(), close(), and eof()
#include <process.h>        // Needed for thread stuff
#include <stddef.h>         // Needed for _threadid

//-----Conditional definitions-----

#if defined (MAIN)
    #define EXTERN
#endif
#if defined (EXTERNAL)
    #define EXTERN extern
#endif

//----- Constants -----

#define FALSE                0    // FALSE = 0
#define TRUE                 1    // TRUE = 1
#define JOB_LIST             "joblist.txt" // Required job list
#define HOST_LIST            "hostlist.txt" // Required host list
#define STATUS               "status.txt" // Logs the status of the jobs and hosts
#define MAX_HOSTS            250 // Maximum number of hosts in host list
#define MAX_JOBS             250 // Maximum number of jobs in job list
#define CLASSIC              0    // Mode is classic - from file
#define NEW                  1    // Mode is new - from std input/output
#define UNASSIGNED           0    // Job is not yet assigned
#define ASSIGNED             1    // Job has been assigned
#define RUNNING              2    // Job is running
#define DONE                 3    // Job is done
#define AVAILABLE            0    // Host is available
#define BUSY                 1    // Host is busy
#define INVALID              2    // Host is not valid

//----- External variables -----

EXTERN struct {
    char exe_file[100];      // *** executable file name
    char in_file[100];      // *** input file name
    char out_file[100];     // *** output file name
    char host_name[100];    // *** host name
    int  status;            // *** job status
    int  mode;              // *** mode of the input/output
} Run_list[MAX_HOSTS];    // Run list structure

EXTERN int  Num_jobs;      // Number of jobs in the job list
EXTERN int  Num_hosts;    // Number of hosts in the host list
```

Appendix A (Continued):

```
EXTERN char   Time[100];           // Time value string

EXTERN struct {
    char host_name[80];           // *** host name
    int  status;                 // *** host status
} Hosts[MAX_HOSTS];             // Hosts structure

//-----Prototypes-----

void help(void);                // Help function
int  transfer(char *option1, char *option2); // Transfer function
int  run(void);                 // Run function to run the jobs
int  get_time(char *time_val);  // Used in Run to display time
void run_thread(void *thread_arg); // Used in Run to thread the jobs
```

Appendix A (Continued):

remotepp.c

```
//===== file = remotepp.c =
//= Main function and get_time function for REMOTE++ =
//=====
//= Contains: main() - Launches appropriate functions based on user commands =
//=          get_time()- returns the current date and time =
//=====
//= History: AMH (11/01/02) - Genesis (from master.c by KJC) =
//=====
#define MAIN
//----Includes-----
#include "remotepp.h"
//=====
//= Main
//=====

void main(int argc, char *argv[])
{
    char          command[80];          // Command
    char          option1[80];         // Option #1 for a command
    char          option2[80];         // Option #2 for a command
    int           i;                   // Loop counter

    //Parse the command and call the appropriate function
    if(argc==1)
        help();
    else
    {
        // Convert command in argv[1] to all upper case
        for (i=0; i<((int) strlen(argv[1]) + 1); i++)
            command[i] = toupper(argv[1][i]);

        // Parse the command and any options
        if ((strcmp(command, "HELP") == 0) || (strcmp(command, "?") == 0))
            help();

        else if (strcmp(command, "TRANSFER") == 0)
        {
            strcpy(option1, argv[2]);
            strcpy(option2, argv[3]);
            transfer(option1, option2);
        }
        else if (strcmp(command, "RUN") == 0)
            run();

        else
            help();
    }
}

//=====
//= Get current date and time function =
//= - Assumes that time_val is allocated in caller =
//=====

int get_time(char *time_val)
{
    time_t        timer;               // Timer variable
    struct tm     *tblock;             // Time stucture

    // Get the time, format it, and remove ending '\n' character
    timer = time(NULL);
    tblock = localtime(&timer);
    strcpy(time_val, asctime(tblock));
    time_val[strlen(time_val) - 1] = '\0';

    return(1);
}
```

Appendix A (Continued):

help.c

```
//===== file = help.c =====
//= Help function for REMOTE++ =
//=====
//= Contains: help() - Help for Remote++ program =
//-----
//= History: AMH (11/18/02) - Genesis (from rhelp.c by KJC) =
//=====
#define EXTERNAL //This is an external file(used in remotep.h)

//----Includes-----
#include "remotep.h"

//=====
//= Help function =
//=====
void help(void)
{
    printf("=====\n");
    printf("= Remote++ =\n");
    printf("=====\n");
    printf("= Usage is 'remotep command options' =\n");
    printf("= =\n");
    printf("= Commands and options are: =\n");
    printf("= HELP -- this screen =\n");
    printf("= TRANSFER file_name host_name -- transfers file file_name to =\n");
    printf("= host host_name =\n");
    printf("= RUN -- to run the programs in joblist.txt on machines in =\n");
    printf("= hostlist.txt =\n");
    printf("= =\n");
    printf("= Example: remotep RUN will run the jobs in the joblist.txt =\n");
    printf("= on the machines in hostlist.txt and provide the =\n");
    printf("= status information in status.txt =\n");
    printf("= =\n");
    printf("= See readme.txt for further usage instructions and help. The =\n");
    printf("= readme.txt describes the format of the required joblist.txt =\n");
    printf("= and hostlist.txt files. =\n");
    printf("=====\n");
    printf("= Author: Ashley Hopkins (amhopki2@csee.usf.edu) =\n");
    printf("=====\n");
}
```


Appendix A (Continued):

transfer.c

```
//===== file = transfer.c =====
//= Transfer function for REMOTE++ =
//=====
//= Notes: 1) Assumes file to be transferred is in c:/remotep =
//=          2) Assumes c:/temp exists on remote host =
//=          3) Transfers file in binary, to transfer as ASCII change -b to -a =
//=              in: strcpy(rcp_string, "rcp -a /Remotep/"); =
//=====
//= Contains: transfer() - transfers files to remote hosts =
//=====
//= History: AMH (12/02/02) - Genesis (from mxfer.c by KJC) =
//=====
#define EXTERNAL // This is an external file (used in remotep.h)

//----- Include files -----
#include "remotep.h"

//=====
//= Function to transfer files to remote hosts =
//=====
int transfer (char *file_name, char *host_name)
{
    int          retcode;          // Return code
    char         rcp_string[100];  // Rcp string to use in System()

    // Output banner
    printf("Transferring file %s to host %s... \n", file_name, host_name);

    //form rcp string to copy the file to the remote host
    strcpy(rcp_string, "rcp -b /Remotep/");
    strcat(rcp_string, file_name);
    strcat(rcp_string, " ");
    strcat(rcp_string, host_name);
    strcat(rcp_string, ":/temp/");

    // Send the file
    retcode = system(rcp_string);
    if (retcode < 0)
    {
        printf("*** ERROR in transfer() - rcp failed \n");
        return(-1);
    }

    // Output completion message
    printf("File transfer complete \n");

    return(1);
}
```

Appendix A (Continued):

run.c

```
//===== file = run.c =====
//= Run function for REMOTE++ =
//=====
//= Notes: 1) Input files must be different for all jobs in joblist =
//=         2) Assumes c:/temp exists on remote machines =
//=         3) Assumes c:/Remotep.exe exists and it contains remotep.exe and =
//=            files to be copied to the remote machines =
//=         4) Hostlist.txt, Joblist.txt, and Status.txt must exist =
//=         5) If only 1 host and it is invalid will get stuck =
//=         6) An unused host is always "valid" in the status file =
//=====
//= Contains: run() - Runs programs in joblist on remote hosts in hostlist =
//=====
//= History: AMH (11/01/02) - Genesis (from mrun.c by KJC) =
//=====
#define EXTERNAL

//----Include files-----
#include "remotep.h"

//=====
//= Function to run programs on remote hosts =
//=====
int run(void)
{
    FILE          *st1;                // File stream #1 for JOB_LIST
    FILE          *st2;                // File stream #2 for HOST_LIST
    FILE          *st3;                // File stream #3 for STATUS
    FILE          *st4;                // File stream #4 for exe_file
    FILE          *st5;                // File stream #5 for in_file
    FILE          *st6;                // File stream #6 for out_file
    char          run_mode[10];        // Mode of the input & output
    char          host_name[100];      // Remote host name
    char          exe_file[100];       // Executable file name
    char          in_file[100];        // Input file name
    char          out_file[100];       // Output file name
    int           i, j;                // Loop counters
    int           done_flag;            // Done flag for job list
    int           assign_flag;         // Assignment flag
    int           job_number;          // number of job being processed
    HANDLE        handle_array[MAX_JOBS]; // array of handles
    int           file_error;          // Error in opening file

    // Output banner
    get_time(Time);
    printf("Starting master for remote runs at %s... \n", Time);
    printf(" >>> Ctrl-C to break-out <<< \n");

    // Open job_list
    st1 = fopen(JOB_LIST, "r");
    if (st1 == NULL)
    {
        printf("*** ERROR in run() - unable to open job list %s \n", JOB_LIST);
        return(-1);
    }

    // Open host_list
    st2 = fopen(HOST_LIST, "r");
    if (st2 == NULL)
    {
        printf("*** ERROR in run() - unable to open host list %s \n", HOST_LIST);
        return(-1);
    }
}
```

Appendix A (Continued):

```
//Open the STATUS file
st3 = fopen(STATUS, "w");
if (st3 ==NULL)
{
    printf("*** ERROR in run() - unable to open status file %s \n", STATUS);
    return(-1);
}

// Build the run list
Num_jobs = 0;
Num_hosts = 0;

while(TRUE)
{
    //initially set error to 0
    file_error=0;

    //scan in job to be executed
    fscanf(st1, "%s %s %s %s", run_mode, exe_file, in_file, out_file);
    if (feof(st1))
        break;

    //Set the Mode
    if((strcmp(run_mode, "file")==0)|| (strcmp(run_mode, "File")==0)
        ||(strcmp(run_mode, "FILE")==0))
    {
        Run_list[Num_jobs].mode = CLASSIC;
        fprintf(st3, "Mode is classic. \n");
    }
    else if ((strcmp(run_mode, "std")==0)|| (strcmp(run_mode, "Std")==0)
        ||(strcmp(run_mode, "STD")==0))
    {
        Run_list[Num_jobs].mode = NEW;
        fprintf(st3, "Mode is new. \n");
    }
    else
    {
        fprintf(st3, "Mode %s is not valid. \n", run_mode);
        file_error=1;
    }

    // Determine if the executable exists
    st4 = fopen(exe_file, "r");
    if (st4 == NULL)
    {
        file_error=1;
        fprintf(st3, "Executable file %s could not be found. \n", exe_file);
    }
    else fprintf(st3, "Executable file %s found. \n", exe_file);
    fclose(st4);

    // Determine if input file exists
    st5 = fopen(in_file, "r");
    if (st5 == NULL)
    {
        file_error=1;
        fprintf(st3, "Input file %s could not be found. \n", in_file);
    }
    else fprintf(st3, "Input file %s found. \n", in_file);
    fclose(st5);

    // Determine if output file exists
    st6 = fopen(out_file, "r");
    if (st6 == NULL)
    {
        file_error=1;
        fprintf(st3, "Output file %s could not be found. \n", out_file);
    }
}
```

Appendix A (Continued):

```
else fprintf(st3, "Output file %s found. \n", out_file);

//print if job was run
if (file_error==0)
    fprintf(st3, "Job was run\n");
else
    fprintf(st3, "Job was not run\n");

fprintf(st3, "\n");

// Print dummy message to output file
fprintf(st6, "Dummy");
fclose(st6);

//Set up runlist if all three files exist
if(file_error==0)
{
    strcpy(Run_list[Num_jobs].exe_file, exe_file);
    strcpy(Run_list[Num_jobs].in_file, in_file);
    strcpy(Run_list[Num_jobs].out_file, out_file);

    fscanf(st2, "%s", host_name);
    if (!feof(st2))
    {
        strcpy(Hosts[Num_hosts].host_name, host_name);
        Num_hosts++;
    }

    if (feof(st2))
        strcpy(host_name, "");
    strcpy(Run_list[Num_jobs].host_name, host_name);

    if (strcmp(host_name, "") != 0)
    {
        Run_list[Num_jobs].status = ASSIGNED;
        Hosts[Num_hosts-1].status = BUSY;
    }
    else
        Run_list[Num_jobs].status = UNASSIGNED;

    Num_jobs++;
}
}

//Copy the remainder of the hosts into the Hosts array
fscanf(st2, "%s", host_name);
while(!feof(st2))
{
    strcpy(Hosts[Num_hosts].host_name, host_name);
    Hosts[Num_hosts].status = AVAILABLE;
    Num_hosts++;
    fscanf(st2, "%s", host_name);
}

// Close JOB_LIST and HOST_LIST
fclose(st1);
fclose(st2);

// Run the jobs
done_flag = FALSE;
while(done_flag == FALSE)
{
    // Test if an assignment can be made
    assign_flag = FALSE;
```

Appendix A (Continued):

```
for (i=0; i<Num_jobs; i++)
{
    if (((strcmp(Run_list[i].host_name, "") != 0) &&
        (Run_list[i].status == ASSIGNED)))
    {
        job_number=i;
        Run_list[i].status = RUNNING;
        assign_flag = TRUE;
    }

    // Make an assignment if it can be done
    if (assign_flag == TRUE)
    {
        // Output a job assigned banner
        get_time(Time);
        printf("Running (%s %s %s) on %s at %s \n", Run_list[job_number].exe_file,
            Run_list[job_number].in_file, Run_list[job_number].out_file,
            Run_list[job_number].host_name, Time);

        // Lanch the thread for the job
        handle_array[job_number] = (HANDLE)_beginthread(run_thread, 4096,
            (void*)job_number );
    }
    assign_flag=FALSE;
}
// Test if all jobs done
done_flag = TRUE;
for (i=0; i<Num_jobs; i++)
{
    if (Run_list[i].status != DONE)
    {
        done_flag = FALSE;

        //Assign a host if needed and available
        if(strcmp(Run_list[i].host_name, "")==0)
        {
            for(j=0; j<Num_hosts; j++)
                if(Hosts[j].status==AVAILABLE)
                {
                    strcpy(Run_list[i].host_name, Hosts[j].host_name);
                    Hosts[j].status = BUSY;
                    Run_list[i].status = ASSIGNED;
                    break;
                }
        }
    }
}
}

//Print host status to status.txt
for(i=0; i<Num_hosts; i++)
{
    if(Hosts[i].status == INVALID)
        fprintf(st3,"%s is not a valid host.\n", Hosts[i].host_name);
    else
        fprintf(st3,"%s is a valid host.\n", Hosts[i].host_name);
}
fprintf(st3, "\nThere are %d valid jobs.\n", Num_jobs);
fclose(st3);

return(1);
}
```

Appendix A (Continued):

```

//=====
//= Thread code for run() =
//=====
void run_thread(void *thread_arg)
{
    int          current_job;          // number of the current job
    char         rcp_exe[100];         // rcp command to copy exe file
    char         rcp_in[100];         // rcp command to copy infile
    char         rcp_out[100];        // rcp command to copy outfile
    char         rshstring[100];      // rsh command to execute exe file
    int          i, j;                // Loop counters
    int          retcode;              // Return code

    //set thread_arg to struct1
    current_job= (int) thread_arg;

    //form rcp string to copy files to remote host
    strcpy(rcp_exe, "rcp -b /Remotepp/");
    strcat(rcp_exe, Run_list[current_job].exe_file);
    strcat(rcp_exe, " ");
    strcat(rcp_exe, Run_list[current_job].host_name);
    strcat(rcp_exe, " :/temp");

    // Copy the executable file to the remote host
    retcode = system(rcp_exe);

    // Check if host in hostlist is valid
    if (retcode > 0)
    {
        for(i=0; i<Num_hosts; i++)
        {
            if(strcmp(Run_list[current_job].host_name, Hosts[i].host_name)==0)
            {
                Hosts[i].status = INVALID;
                break;
            }
        }
        strcpy(Run_list[current_job].host_name, "");
        Run_list[current_job].status = UNASSIGNED;
        _endthread();
    }

    // Set up and send job if mode is New
    if(Run_list[current_job].mode == NEW)
    {
        //form rsh string to execute on remote host
        strcpy(rshstring, "rsh ");
        strcat(rshstring, Run_list[current_job].host_name);
        strcat(rshstring, " c:/temp/");
        strcat(rshstring, Run_list[current_job].exe_file);
        strcat(rshstring, " <");
        strcat(rshstring, Run_list[current_job].in_file);
        strcat(rshstring, " >");
        strcat(rshstring, Run_list[current_job].out_file);

        // run rsh with redirection of input and output
        system(rshstring);
    }

    // Set up and send job if mode is Classic
    else
    {
        //form rcp string to copy input file to remote host
        strcpy(rcp_in, "rcp -b /Remotepp/");
        strcat(rcp_in, Run_list[current_job].in_file);
        strcat(rcp_in, " ");
        strcat(rcp_in, Run_list[current_job].host_name);
        strcat(rcp_in, " :/temp/infile");
    }
}

```

Appendix A (Continued):

```
//rcp the input file
system(rcp_in);

//form rcp string to copy output file to remote host
strcpy(rcp_out, "rcp -b /Remotep/");
strcat(rcp_out, Run_list[current_job].out_file);
strcat(rcp_out, " ");
strcat(rcp_out, Run_list[current_job].host_name);
strcat(rcp_out, ":/temp/outfile");

//rcp the output file to remote
system(rcp_out);

//form the rsh string
strcpy(rshstring, "rsh ");
strcat(rshstring, Run_list[current_job].host_name);
strcat(rshstring, " c:/temp/");
strcat(rshstring, Run_list[current_job].exe_file);

// run rsh with input and output from file
system(rshstring);

//form rcp string to copy output file back from remote host
strcpy(rcp_out, "rcp -b ");
strcat(rcp_out, Run_list[current_job].host_name);
strcat(rcp_out, ":/temp/outfile");
strcat(rcp_out, " ");
strcat(rcp_out, "/Remotep/");
strcat(rcp_out, Run_list[current_job].out_file);

//rcp the output file back from remote
system(rcp_out);
}

//Output a done message
get_time(Time);
printf("Job with outfile %s completed at %s \n",
    Run_list[current_job].out_file, Time);

// Set status to DONE
Run_list[current_job].status = DONE;

//Set host available
for(j=0; j<Num_hosts; j++)
{
    if(strcmp(Run_list[current_job].host_name, Hosts[j].host_name)==0)
    {
        Hosts[j].status=AVAILABLE;
        break;
    }
}
_endthread();
}
```