A LOAD SHARING POLICY FOR CPU-INTENSIVE TASKS ON A NETWORK OF INDEPENDENT WORKSTATIONS

By

ANIL FRANCIS THOMAS

Bachelor of Technology

Regional Engineering College

Calicut, India

1988

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1993

A LOAD SHARING POLICY FOR CPU-INTENSIVE TASKS ON A NETWORK OF INDEPENDENT WORKSTATIONS

Thesis Approved:

[illegible]

Thesis Adviser

Blayne E. Mayfield

[illegible]

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGMENTS

I sincerely thank my graduate adviser Dr. Mitchell L. Neilsen for the guidance, help and time he has given me for the completion of my thesis work. His perseverance and hard work inspired me to venture into the advanced aspects of this work. I would like to express my sincere thanks to Dr. David L. Nofziger for the guidance and help he has given me during the entire period of my graduate study. Without the encouragement and help he has given me, the completion of this work would have been impossible. I also sincerely thank Dr. Blayne Mayfield for serving on my committee. His suggestions have helped me to improve the quality of this work.

My special thanks goes to Mr. Dwayne T. Hunter for arranging the computing facilities for me. I thank Dr. Marvin Stone and Dr. Dan Storm for making their machines available to me.

My respectful thanks goes to my parents Prof. S. L. Thomas and Mrs. Thankamma Thomas for all the love and support they have given me in my life. And, last but certainly not least, I thank all other members of my family for the love, encouragement and confidence they have endorsed in me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In a local area network (LAN) of high performance workstations, a large percentage of the CPU cycles are typically wasted. Most of the time, users execute interactive processes that require very little CPU processing. In order to utilize this wasted CPU power, various *load distributing* policies have been proposed over the past several years. These policies can be classified as *load sharing* or *load balancing* policies [SKS92].

- **Load sharing** policies attempt to maximize the rate at which a distributed system performs work by ensuring that no node in the network is idle while processes are waiting to be serviced at some other node in the network. Since there can be waste due to delays involved in the determination of idle nodes, an improvement to this scheme is to make *anticipatory transfers* to nodes with low loads that are expected to become idle soon.

- **Load balancing** policies go a step further, and try to equalize the workload among nodes. Even though load balancing can potentially reduce the mean and standard deviation of task response times, the overhead due to the higher transfer rate can outweigh the potential performance improvement.

Distributed operating systems, such as *Sprite*, *V-Kernel* and *Stealth*, provide a policy for process migration to achieve improved performance. In the Sprite implementation [DO87, DO91], for example, processes can run on other nodes in the network. A remote process running on a machine is preempted and returned to the machine on which it was invoked, when the owner returns to the remote machine. Even though this kind of preemptive load distribution policy enhances the system's performance, it has been shown that a large percentage of the wasted CPU power can be retrieved by using a non-preemptive policy[AC88, PL88].

Moreover, previous research has shown that even while a workstation is being used, the resource utilization is probably very low, wasting the available power. The distributed operating system Stealth was designed to take advantage of this wasted power. Stealth allows foreign processes to run at a lower priority along with local processes. The foreign processes are allocated resources that are left after meeting all local process requirements.

Even without the aid of a distributed operating system, a LAN of independent workstations can retrieve the wasted CPU power by using a load sharing policy. Load sharing is more appropriate than load balancing when the entire network cannot be considered as a single system. A client-server paradigm is ideal in such a network, because minimal resources are wasted on the additional processes and minimal global information is maintained.

1.1 Thesis

Monte Carlo simulations carried out on single processor workstations are ideal candidates for parallel processing. Most of the computation is carried out on blocks of data which are independent of any previous computation. *Chemical Movement through Layered Soils* (*CMLS*) is one such simulation performed on a Sun workstation. The simulation was developed by faculty members in the Agronomy Department at Oklahoma State University. It is an improved version of the work presented by Nofziger [NH86]. The objective of this thesis is to design a system to distribute the work onto different workstations connected by a network, and consequently speed up the computation.

We develop a load distribution policy suitable for Monte Carlo simulations that are executed on Sun workstations connected by a LAN. The scheme involves using a client-server model. Servers run on individual machines, and accept requests for the remote execution of jobs based on the local load. Also, servers communicate with each other to facilitate the execution of local jobs. The decision to accept a remote request is adjusted dynamically based on the overall system load.

The system developed is fault-tolerant. In particular, the system is capable of recovering from remote machine failures. Secondly, the system developed is fairly general and can be adapted to the distributed computation of any other CPU-intensive task. All of the network programming aspects are handled within the server code. Finally, asynchronous communication between the various machines improves the processor utilization of the machines and speeds up the computation.

1.2 Organization

The thesis is divided into the following chapters:

- Chapter 2: A discussion on previous work related to load distribution is presented.

- Chapter 3: A description of the Monte Carlo simulation on a single machine and issues related to implementing a distributed simulation are presented.

- Chapter 4: A detailed discussion of the work and implementation details are given.

- Chapter 5: Performance metrics used to evaluate the simulation and the results obtained are explained.

- Chapter 6: A summary of the thesis and suggestions for future work are presented.

- Appendix A: A guide for using the system is presented.

- Appendix B: The client code is given.
- Appendix C: The *Makefile* for the client code is given.
- Appendix D: The header file for the server code is presented.
- Appendix E: The server code is given.

CHAPTER 2
LITERATURE REVIEW

Availability of excess computing power in a cluster of high performance workstations has been previously studied in detail [ABM^{+}92, LK89, Mut91]. Various methods have been proposed for estimating the load on machines so that excess processing power can be shared by users on other machines. Strategies used in utilizing the excess power in various contexts are investigated here.

First, we summarize work on load distribution in distributed operating systems. The workstations connected to the network are called *nodes* in the system. Performance improvement is achieved by using a load balancing or load sharing policy. Examples of such operating systems include *Sprite*, *V-Kernel* and *Stealth* [DO87, DO91, PC91, SKS92].

In order to achieve better performance, individual processes running on each of the nodes are preemptively or non-preemptively transferred onto other nodes in the system. This requires a global scheduler for the scheduling of the tasks on different nodes in the system. The primary objectives of such a scheduler is to [TL88]:

- minimize performance degradation due to overhead imposed by the scheduler,
- scale well as the number of nodes increase, and
- be fault-tolerant.

Both centralized and distributed schedulers have been implemented with these as the primary objectives. Centralized schedulers are found to scale better than distributed schedulers. A centralized scheduler determines the subset of nodes willing to handle remote jobs. As the number of nodes increases,this reduces the number of messages exchanged and the time needed in negotiating a request. In the case of the distributed scheduler, individual machines have schedulers that keep track of the system state. Hence, as the number of nodes increases, the number of messages exchanged increases, making it less efficient.

However, a centralized scheduler is less fault-tolerant. Fault tolerance can be achieved in a centralized system by having multiple replicas of the scheduler in the network. But this will increase the complexity of the system, since some form of error detection and error recovery procedure will need to be invoked when the node running the scheduler fails. A distributed scheduler does not require any recovery upon the failure of a node. Thus, although the centralized scheduler may scale better, it is less fault-tolerant than a distributed scheduler.

Goswami [GDI93] proposes a scheme which considerably reduces the overhead in querying other machines to find their actual load. A prediction-based load sharing heuristic is proposed. The heuristic predicts the requirements of a process in terms of CPU time, memory and file I/O. Then, the process is assigned to a machine based on these requirements. Such a scheme requires the study of previous requirement patterns of processes. This is implemented in two stages. The first stage is performed offline and the second stage is performed online. By reducing the overhead of online study, higher performance is obtained.

In order to recover the unused power in distributed systems, various concurrent programming languages are available. Examples of these include *SR*, *Concurrent C* and *Ada*. The SR programming language [OACT92], for example, allows a program to be split into subprograms. Each subprogram is executed on a *virtual machine*. Virtual machines can be located on one or more physical machines.

In a network of independent workstations, several client-server computing models were investigated for distributing the load from heavily loaded workstations to lightly loaded ones [AC88, Hag86, LLM88, Mur92, WHH92]. Since the nodes of the network in this case are independent, load balancing schemes are not appropriate in this context. Instead, load sharing schemes are used. Each node shares some of its resources with other nodes as long as its users are not significantly affected.

Some proposed strategies involve allowing remote jobs to be executed on a machine if the machine is completely idle or if its load is below a threshold value. In most of these schemes, only an idle machine becomes a candidate for being a server. Also, when the user returns to the machine, the remote jobs are either run at a lower priority, transferred back to their originating machine, transferred to another idle machine or simply terminated.

The Condor scheduling system [LLM88] , for example, is designed to take advantage of idle workstations in the system. It identifies idle workstations in a system and schedules background jobs on them. These background jobs would have actually been waiting to be executed on another heavily loaded node. The scheduler assures minimum interference to the local processes by a remote process scheduled on the local machine. When an owner returns to the local machine, remote jobs running on it are checkpointed and transfered to another machine. The system ensures that the job will eventually complete and very little work will be performed more than once.

The Condor scheduling system chooses an approach in between the centralized and distributed schedulers. Each node has a local scheduler. A central coordinator running on one of the machines allocates capacity from idle workstations to the local scheduler on workstations with background jobs waiting to be executed.

The observation that even machines that are not idle will under-utilize the resources has prompted some researchers to use some threshold values of load on the machines to decide whether it becomes a candidate for processing remote jobs. In one such scheme, the same threshold value is used to decide whether to accept jobs from other machines or to off-load jobs onto other machines. An improvement to this scheme is the use of two threshold values, a low and high mark [AC88]. If the load on a machine is above the high mark, when a new local job request arrives, the machine tries to execute the job on a remote machine. The low mark serves as a decision parameter to accept remote jobs. When a request for the execution of a remote job arrives at a machine, it checks its load against the low mark. If the load is less than the low mark the job is accepted, otherwise it is rejected.

Almasi [AHM+93] gives a practical implementation of distributed computing in a network of independent workstations. The system is implemented using remote procedure calls (RPC). The performance improvement is not directly proportional to the number of machines participating in the computation, since there will be a considerable increase in the network load with an increase in number of machines.

CHAPTER 3
PROBLEM STATEMENT

Monte Carlo simulations carried out on single processor workstations are ideal candidates for parallel processing. Most of the computation is carried out on blocks of data which are independent of any previous computation. *Chemical Movement through Layered Soils* (*CMLS*) is one such simulation performed on a Sun workstation. The simulation was developed by faculty members in the Agronomy Department at Oklahoma State University. It is an improved version of the work presented by Nofziger [NH86]. The objective of this thesis is to design a system to distribute the work onto different workstations connected by a network, and consequently speed up the computation.

Input to the program is given through an input file. The input file has a general information block, which applies to a group of system data blocks that follow. Certain general information parameters need to be generated, based on this general information, for that group of data. There can be multiple sets of general information and system data group combinations, requiring the generation of the general information parameters, before the computation of each system data group.

The general format of the input file is shown in Figure 3.1. In a typical file, there will be 450 sets of the system data that needs to be computed based on one general information block. The general information parameters are recomputed a number of times, and each time the new set of parameters is applied to the system blocks. The input file specifies the number of times each recomputation needs to be done. Each such iteration is called a *replication* in this thesis. Typically, there will be 500 replications. This means that the general information parameters should be generated 500 times and each time the associated 450 system data blocks should be processed.

In the single machine implementation, the general information parameters are generated first. Then, the data associated with the general information is read in one set at a time, and the necessary computations are performed. The output from each set of input data is written to an output file. When all data associated with a general information block has been processed, general information parameters are generated for the next general information block, if specified, and so on.

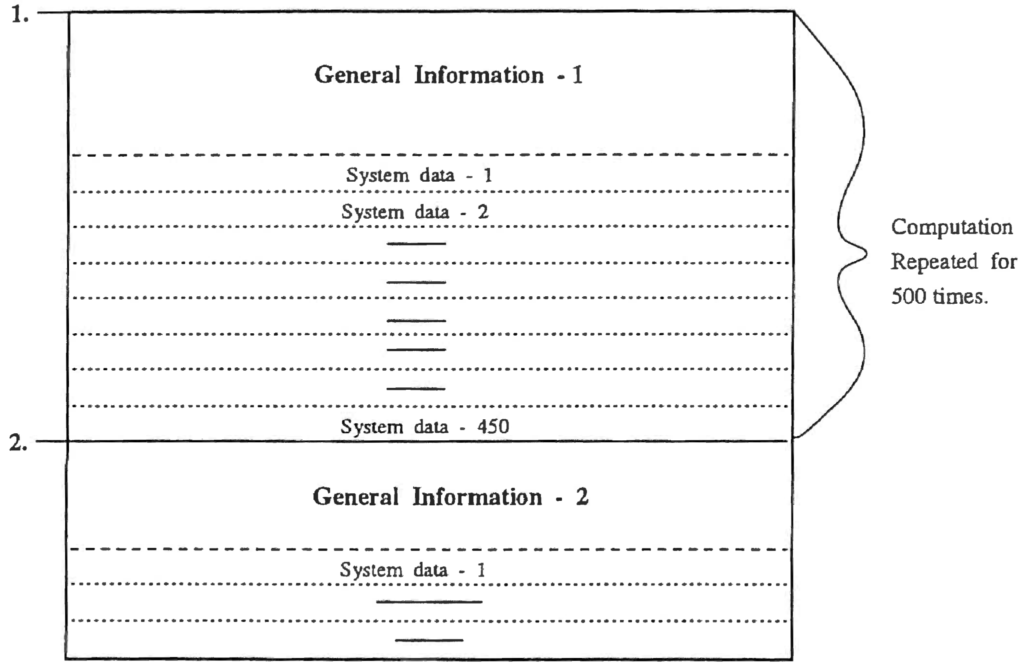

Figure 3.1 Input File

The processing time of general information data and each system data block depends on the speed of the machine on which it is executing. On the fastest machine used here, the general information can be generated in approximately 600 milliseconds. Each system block computation takes 20 milliseconds. On the slowest machine these values are 1600 milliseconds and 80 milliseconds, respectively.

In parallelizing this computation, two approaches are considered. The first method is to compute the general information block on the base machine and to send the generated parameters with each set of the system blocks to the remote machines.

However, this approach imposes heavy network traffic because the ratio of the computation time to communication time is very low. Moreover, in another mode of operation of the program, the general information parameters need to be regenerated before the computation of each system block. This will require the transmission of the general information parameters with each set of the system blocks. This will add to the network load and deteriorate performance.

The second method is to send the general information data to the remote machine so that the parameters can be generated there. The system blocks can be made available at the remote location at the time of execution. In this way, once the general information parameters are generated, one entire replication can be processed at the remote machine. The ratio of the computation time to the communication time is high, and consequently a high performance improvement can be expected. Another advantage of this approach is that, due to the style of the single machine implementation, the system blocks need not be transmitted to the remote machine. Instead, the data file can be made available at the remote end before execution and a pointer to the beginning of the first system block can be passed to the remote machine.

The advantages of the second method prompted us to implement the distributed computation using this method. The implementation details are given in the next chapter.

CHAPTER 4
IMPLEMENTATION

The principal objectives in designing the model were

- Maximum speedup.
- Reliable computation.
- Minimum network overhead.
- Minimum changes to the existing program.

In order to obtain maximum speedup in solving the above problem, it is essential to minimize the communication overhead. Also, delay due to server overload should be avoided. The model presented here is designed with these motives in mind.

4.1 Environment

A local area network (LAN), consisting of 8 Sun[1] workstations running SunOS 4.x, forms the platform for running the experiment. All machines on the network, have there own disk drives, and two machines act as file servers. The workstations are connected by an Ethernet with a capacity of 10Mb/sec. The versions of the operating systems running on the machines are different. Table 4.1 gives the specifications of the machines used in the simulation.

4.2 Implementation

This section describes the implementation of each feature in the developed system. Section 4.2.1 gives a brief outline of the overall system. The following sections describe

[1] Sun and SunOS 4.x are the registered trademarks of Sun Microsystems, Inc.

in detail the implementation of each of the modules.

Model	Operating System	Name of Machine
Sparc 10 model 41	SunOS Release 4.1.3	Biosun
Sparc 10 model 41	SunOS Release 4.1.3	Soil
Sparc 10 model 21	SunOS Release 4.1.3	Wqsun
Sparc 10 model 20	SunOS Release 4.1.3	Sand
IPX	SunOS Release 4.1.3	Zoo
Sparc 2	SunOS Release 4.1.3	Hydsun
IPX	SunOS Release 4.1.3	Neusun
IPC	SunOS Release 4.1.2	Soilwater

Table 4.1 Workstations

4.2.1 Outline

The scheme uses the client-server paradigm [Ste90], with a server running on each of the workstations in the network and the client running on any one workstation. Network communication is handled using TCP/IP. Mutual exclusion and interprocess communications are achieved using the IPC facilities.

Each server is blocked on their respective machines, waiting for either a client on its own machine or a server from another machine to contact it for service. Clients request service from the servers. A client can only contact the server on its own machine. Services from other servers are requested and obtained by the local server.

When the client makes a call to the initialization procedure, it sets up a connection with the server on its own machine and requests service. Then, the server contacts other servers and finds out if any other server is willing to offer service. Each remote server makes this decision based on the load on its machine. If a remote server agrees to accept the request, the client informs the local server which executable code is to be applied to the data. Then, the client passes data to its local server, one set at a time. The local server passes data and the location of the executable code to the remote servers. The output from the servers are collected by the local server and written to the output file specified by the client.

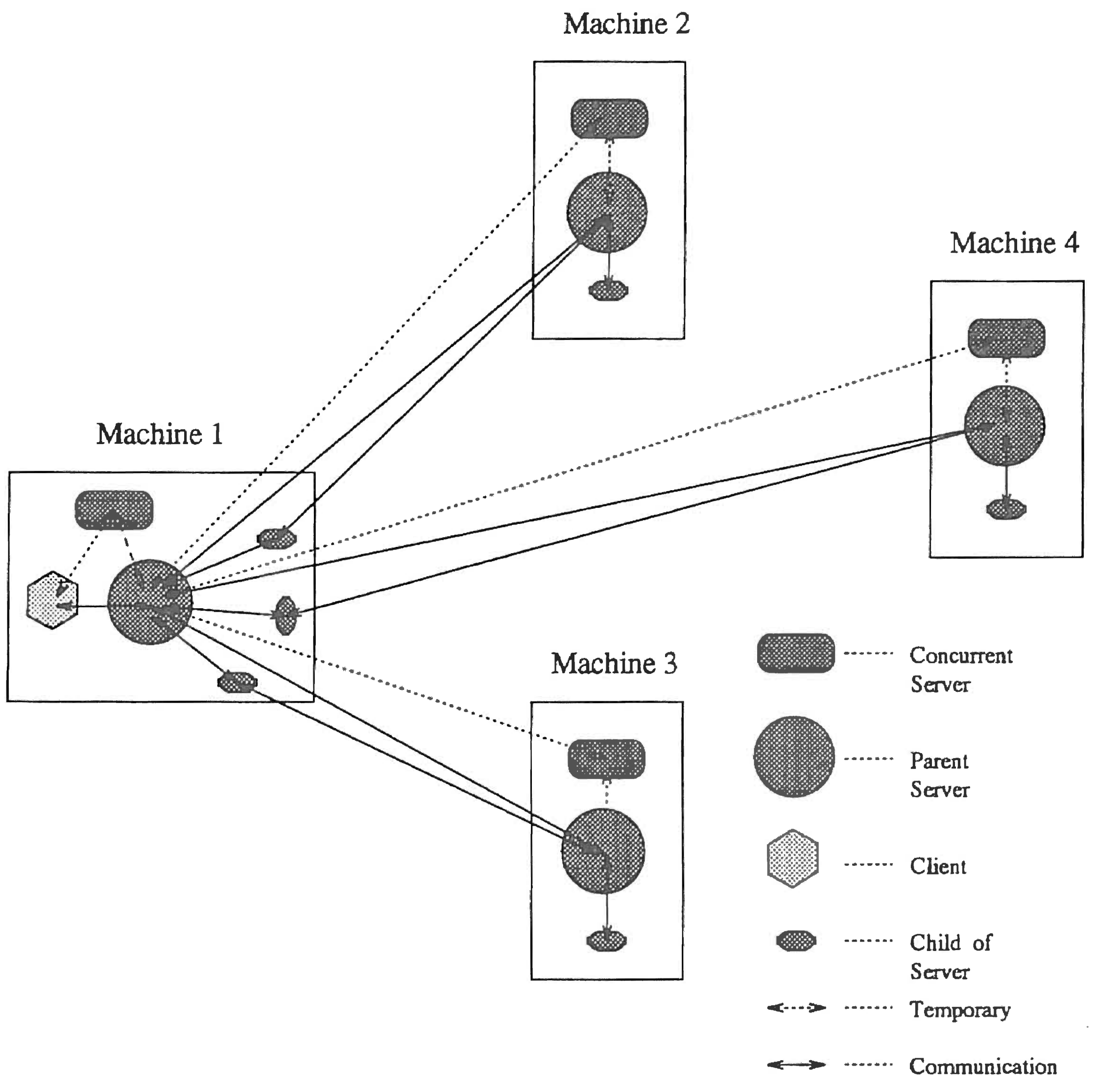

Figure 4.1 Prototype

The dissemination of the data and collection of the output is done asynchronously so that maximum utilization of the servers is obtained. Also, the servers are concurrent. Hence, clients requesting service are not unduly blocked.

The system developed is fault-tolerant; in particular, the system can tolerate the failure of remote servers due to machine failures.

4.2.2 Server

The servers are started up as background process on each of the participating machines. Upon initialization, the server reads from a database containing the names and IP addresses of all known workstations participating in the distributed computation. The database is in the form of a text file. Each line in the database specifies the name and address of a workstation separated by any number of space characters. A line starting with the # character is ignored. Hence, if a remote server is to be eliminated from the computation, its entry can be commented out in the server database in the client's file system. Note that the server reads this database only upon initialization and not for each client request. Hence, if a remote server is to be eliminated from the set of participants, the local server must be restarted after the database is modified.

The server on the client machine can also act as a remote server and service the client's request. This will be useful when the load on the local server is low. This is specified by giving the name and IP address of the local server in the database of servers.

The servers on each machine block and wait for a service request from a local client or a remote server. When a service request arrives, the server forks off a child process to handle the request. Then, the server blocks and waits for another request to arrive. This is called a *concurrent server* and it reduces the delay in serving a new request [Ste90].

The child process (also called a *slave*) checks to see whether the request is from a remote server or a local client. There are two types of remote requests. The first is a request for service. The second is a check by a remote server currently being serviced by the machine. This check is to ensure that the machine is still alive. This step is used as part of the fault tolerance feature of the system.

Server processing local request

A server can receive a request from a local client. When such a request arrives, the server determines which remote servers are willing to process the client's request. A remote server agrees to accept a request based on its local load. If the local server cannot find any remote servers willing to accept the request, it informs the client and the program terminates. If any of the remote servers expresses interest in servicing the request, then the local server informs the client to send data for processing.

When data is sent for processing to different servers, the results cannot be expected back in the same amount of time from all machines. The processing time of a job on a machine will depend on its speed and load. In order to maximize utilization of the servers, it is essential that no remote server has to wait for another slower remote server to finish processing. Hence, the communication between the local server and the remote servers should be asynchronous.

Asynchronous communication is achieved by having a child process created to communicate with each remote server. These child processes read the output from the remote server and write to an output file. When multiple processes are involved in writing to a single file, it is essential to enforce mutual exclusion between the processes to prevent race conditions. This is achieved by using the inter-process communication (IPC) facilities available on most UNIX[2] systems. A *semaphore* is used by the child processes to synchronize their writing to the output file.

Once the server has informed the client to send data, it will create a semaphore, which will be used later by the child processes. Then, the client sends the server the name of the executable code and the output filename. The server creates the output file and informs the current remote server about the location of the executable code. The executable code is maintained by a common file server. Any data files required

[2] UNIX is the registered trademark of UNIX System Laboratories, Inc.

while executing the code are made available through the file server. The need to have a common file server can be avoided by copying the executable code and the data files to the individual machines.

Once the executable code is known to the server, the client passes a data block to the server. The server sends the data block to the remote machine which has agreed to service the request. Next, the server forks off a child process to read the output from the remote server. Then, the server checks if the client has any more data to be processed. If so, it tries to locate another server and sends the executable code name and data to it. If the server cannot locate another machine prepared to accept the request, it will wait until one of the remote servers currently processing the request to become free. The server knows that a remote machine has become free when one of its child processes exits after reading the output from the remote machine. Then, the server queries that remote machine for its willingness to process more data. If the remote machine accepts the request, data is sent to it.

In the single machine version of CMLS, the set of random numbers generated by the end of a replication is needed in processing the next replication. However, in the distributed version, the set of random numbers generated by the end of a replication on a particular machine is used before processing the next replication on that machine. Thus, the set of random numbers at the end of a replication is sent back to the local server. If the same remote server agrees to process another replication, then the same random numbers are sent to it.

Once a remote server starts processing the data, only the child process has further contact with it. Hence, it is necessary to have a means for transferring the random numbers from the child to the parent server. To achieve this, before forking off the child process, the parent process creates a pipe that it will use to read the random numbers from the child process.

When the remote server finishes computing, it sends the random numbers to the child process on the local machine. The child process writes its process id and the random numbers into the pipe. Then, it tries to decrement the semaphore by doing a *down* operation on it. If it succeeds, it will write the output into the output file. If some other process is currently writing into the file, the child process will be blocked. Once a process finishes writing to the file, it releases the semaphore by doing an *up* operation. This will wake up any process currently blocked on the semaphore.

Once the child process completes reading the output, it will do a normal exit. The parent process is set to receive the signal when the child exits. This signal invokes a signal handler. The signal handler checks the pipe to see whether there is anything to read. This measure is required since the child process could have been terminated by the parent, in case of remote machine failure. Section 4.4 explains this situation. The signal handler returns if there is nothing to read from the pipe. Otherwise, the signal handler reads the pipe to get the process id of the child that just exited and the random numbers. The child process id is used to determine which remote machine has completed processing. The set of random numbers are stored in a data structure associated with the remote machine. Then, the signal handler marks the remote machine as idle and returns. Finally, the parent process queries this remote machine and any other machine not currently in use, to service new requests.

This sequence is repeated until all data is processed. When all child processes exit, the parent removes the semaphore from the IPC table and closes the output file. Then, a message is sent to the client regarding the completion of the computation.

Once the parent server is set up, the creation and removal of child processes are dynamic, and based on the availability of remote machines.

Server processing remote request

A request to a remote server can be of two types. The first type is a request for service. The second type is a check by another server currently being serviced by the machine. This check is used to ensure that the machine is still alive. This step is used as part of the fault tolerance feature of the system. When the server receives this request it simply discards it. The local server knows that the remote server is alive because it was able to set up a connection with it. Hence, there is no need to reply to the fault tolerance check.

When the request type is a service request, the server first checks the load on the machine. If the load is below a threshold value, the server will accept the request. Otherwise, if the load is above this value, the server rejects the request. The determination of the load and threshold value is discussed in Section 4.3.

If the decision is to accept the request, then the requesting server will send the executable code name and the set of random numbers to the server. Then, the data on which the computation is to be performed is passed to the server. If the executable code is invoked using the *exec* utility of UNIX, the invoking process will be overlaid by the new code; that is, the server will be destroyed. In order to avoid this, the server first forks off a child process and this process will execute the code.

Even though the data read from the requesting server is available in the child process, when the *exec* call is made, the new process will destroy this data. In order to make the data available to the new process, the inter-process communication facility *shared memory* is used. Before reading the data from the requesting server, the parent server creates a shared memory segment. Then, the data is written directly into this memory. When the child process executes the *exec* call, the new process will acquire this shared memory and use it for processing the data.

The size of the shared memory in this implementation is 4.5 Kbytes. The first 4 Kbytes are used to pass the data and the next 0.5 Kbytes are used for passing the random numbers.

The output from the computation is stored in a temporary file so that the parent server can transmit it back to the requesting server. The name of the temporary file is created by the parent using its process id so that it will be unique. It is passed to the slave process as an argument. The new process writes its output to the temporary file. Once the computation is over, it writes the final set of random numbers into the second part of the shared memory and exits. While the slave process is computing, the parent process is blocked waiting for the slave process to complete. When the slave process exits, the parent process transmits the random numbers to the requesting server. At the receiving side, the child process reads the random numbers and passes them to the parent. Then, the transmitting server reads the output file and passes the contents to the requesting server. Once all data has been transmitted, the parent server removes the shared memory and the temporary output file. Then, it exits.

4.2.3 Client

The client initiates a distributed computing request by making an initialization call. This call sets up a connection with the server on its own machine. Then, the local server contacts other remote servers to find out whether at least one remote server is willing to participate in the computation. If no servers are willing to serve, the client exits. If there is at least one server willing to participate, the client will inform the server of the executable code and the name of the output file. Then, the client will start passing the server one data block at a time, which the server distributes to the remote servers. Once all data blocks are transferred, the client waits for the servers to complete. Upon completion, the output of the computation is available in the output file specified while invoking the client.

Another parameter used while invoking the client is the number of remote replications to be performed per invocation of a remote request. This parameter is passed with the data block to the remote server. The remote server uses this parameter to see how many replications need to be performed. This parameter is useful when the number of remote machines increases. Even though the size of the output that needs

to be transmitted is not different by making one remote invocation with n replications or with n remote invocations, this measure will eliminate n-1 calls across the network to the remote machine. However, since the remote machine will check its load only when accepting a service request, if too many replications are made at the same time, the load balancing may not be properly handled.
After initialization the client, just passes the data to the server. It doesn't know how fast the server will get the work done. The server takes care of all networking aspects.

4.3 Load Metric

When an external request reaches a server, it decides whether to honor the request based on the local load. The load on the machine is determined using the *load average* metric provided by the *uptime* command of UNIX. Uptime gives the exponentially smoothed average number of jobs in the run queue over the previous 1, 5, and 15 minutes. The value given by the uptime command, at the 5 minute level, is used as the metric for accepting a service request. This value is checked against a threshold value to see whether the load is below the threshold value. If the load is below the threshold value, then the request is accepted. Otherwise, it is rejected.

If the load on all of the machines participating in the computation is high, then the threshold value on each of the machines can be adjusted higher. This step will help the local server in finding a suitable remote server. For this reason, the threshold value of a machine is determined based on the global load. The server on the client machine computes the global load from the loads on the individual machines involved in the computation. This global load value is passed to each of the servers so that each can set a threshold value for itself. Effectively, the global load value helps in balancing the loads on the machines involved in the computation.

4.4 Fault Tolerance

Computations involving remote machines are prone to machine failures. When several machines are involved in a computation, it is important to make sure that none of the machines fail during the computation. If a remote machine fails, the server on the client machine will wait forever, expecting the remote machine to return the results at any time. In order to make sure that the remote machines are alive, a fault tolerance scheme is incorporated into the developed system.

The server on the client machine sends *heart beat* messages to remote machines involved in the computation. These messages let the server know whether the remote machines are alive or not. If a remote machine fails, there are two steps the server can take. The normal step is to exit, letting the client know that a remote machine has failed. The second step is to send the data to another machine and get the results computed there. This second option can be installed by defining the keyword FAULT_RECOVERY when compiling the server code.

When the second option is adopted, the server writes a temporary file when data is sent to a remote machine. The name of the temporary file is the same as the name of the remote machine. If a remote machine fails the server terminates the child process created for reading the output from that server. Also, it updates the data structure that holds the information on current remote servers. Then, the data written in the temporary file is sent to another server for recomputation and a new child process is created for reading the output from the new remote server. Since writing temporary files involves additional overhead, the performance will be slightly degraded. Hence, in a reliable network this option does not need to be turned on.

4.5 Random Numbers

The seed to the random number generator is specified in the input file. In the distributed version, since a different seed is needed on each machine, the server on the client machine generates a seed for each machine. Since the set of random numbers used in one replication is required in the next replication, the server on the client machine manages the collection and redistribution of the random numbers to the appropriate machine.

The random number handling is the only feature that is specific to the CMLS project. Other than this feature, the server code is fairly general and can be adapted to distribute the computation of any CPU-intensive task.

CHAPTER 5

PERFORMANCE ANALYSIS

The metrics used for the performance analysis include:

- The speedup achieved by distributing the load to other machines.
- The variation in network load with increasing granularity in the data transmitted.
- The effect of the number of servers involved in the computation on the network load and speedup.

The system was tested with varying numbers of machines using the same input file. The speeds of the machines used were different.

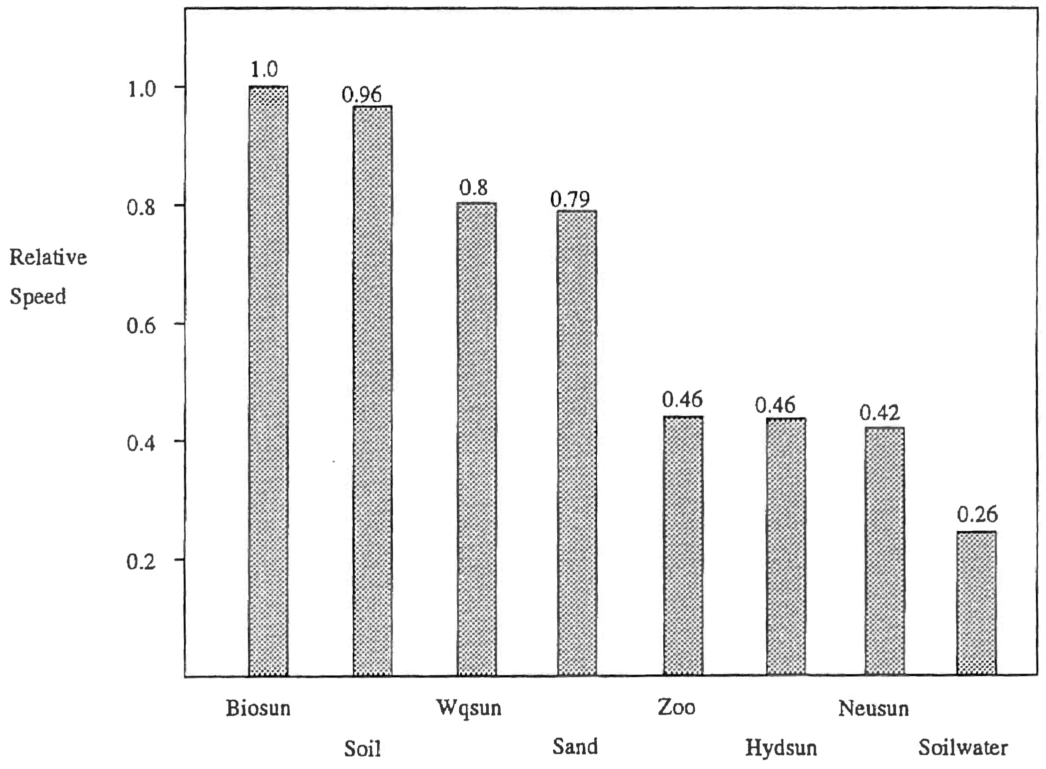

Figure 5.1 Speeds of the Machines

Figure 5.1 shows the comparison between the speeds of the machines. All of the machines are calibrated against the speed of the fastest machine. Machine *biosun* was used as a base for comparing the speeds of other machines. The calibration was done by measuring the time taken to execute the program on that machine alone. Time was measured using the UNIX *time* command.

Since the speeds of the machines used in the study are not uniform, the available power is not an integral multiple of the machines used. When studying the speedup achieved by adding each remote machine to the base machine, this factor has to be considered.

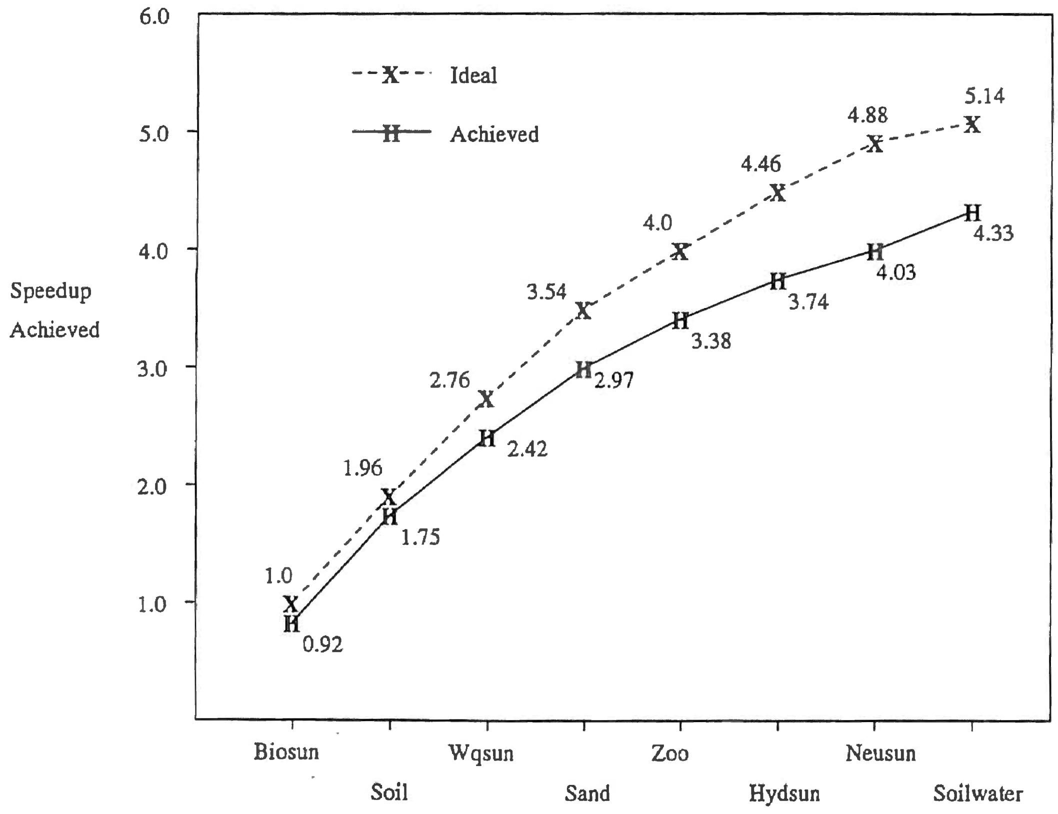

Figure 5.2 Speedup Achieved

In this study the machine *biosun* was used as the base machine to start the client. The speedup achieved by adding each remote machine to the previous machines is given in Figure 5.2. This figure shows that the speedup achieved is not proportional to the available computing power.

Figure 5.3 estimates the speedup that would have been achieved if the speeds of all machines were the same. This normalized speedup chart is obtained by extrapolating the results obtained from the experiment.

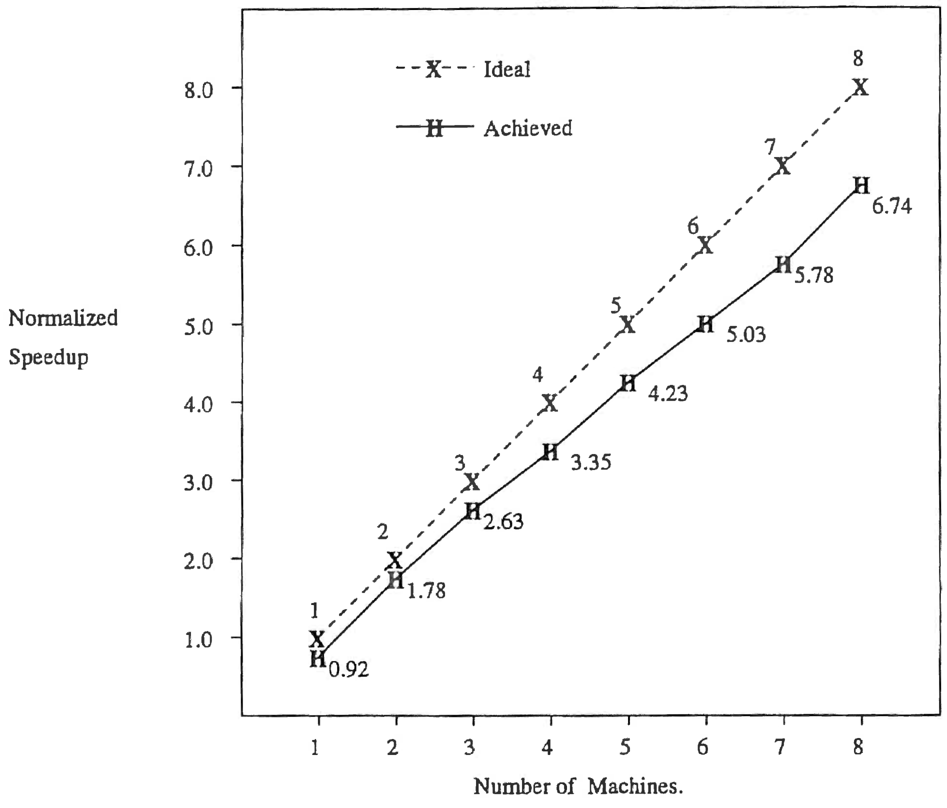

Figure 5.3 Normalized Speedup

As the number of machines increases the speedup achieved will not be linear. This is partly due to the increase in network load and partly due to the overhead of the sequential parts of the program. On the base machine, a single process reads the input file. Also, the output from the machines are collected from different machines and written into the same file. These parts of the program form a bottleneck to the amount of performance gain that can be achieved.

Figure 5.4 shows the network load as the number of machines increases. This graph is obtained from the number of packets sent across the network as the computation was in progress. A packet size of 1500 bytes is assumed in computing the network load. This measurement is the worst case approximation of the actual network load imposed by the distributed computation.

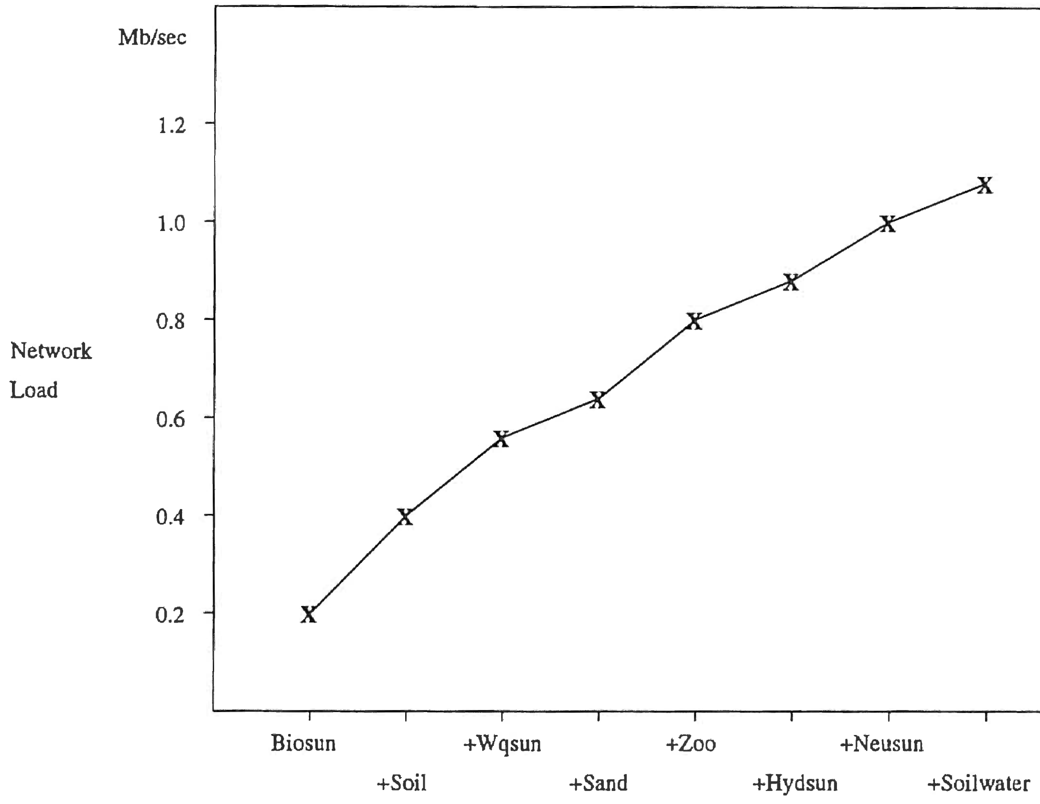

Figure 5.4 Network Load

In this simulation the relative speedup is not significantly affected by the number of machines involved. One reason is that the number of machines used is not large enough to reflect this change. Another reason is that the ratio of the computation time to the communication time is fairly high.

An additional option given to the simulation is to have a variable number of replications computed on the remote machine in one invocation. This feature will help in reducing the network load. Figure 5.5 shows the variation in the network load with different number of replications computed at the remote locations. All eight machines were used in computing this result.

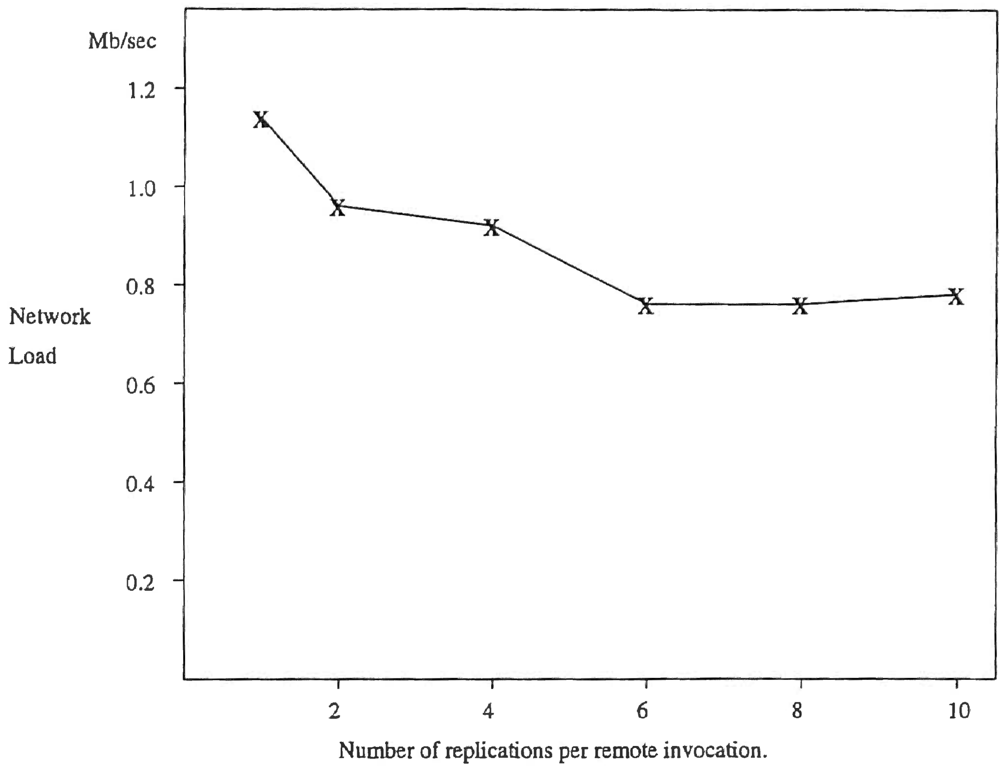

Figure 5.5 Network Load Variation

From the graphs given here it is evident that the system developed is highly efficient in utilizing the available CPU power. Also, since the ratio of the computation time to the communication time is high, the performance can be expected to improve comparably with the addition of more machines.

CHAPTER 6
CONCLUSION

6.1 Summary

A client-server paradigm for the efficient distribution of work on a network of independently owned workstations is presented here. This system utilizes the available CPU power with high efficiency. This is achieved by having asynchronous communication between the local server and the remote servers. The system developed is fault-tolerant and can recover from remote machine failure. The loads imposed on the different machines involved in the computation are balanced by means of a global load parameter. Individual machines adjust their threshold values based on the global load. This measure ensures that unnecessary network traffic due to unsuccessful service requests are avoided.

6.2 Future Work

Load balancing is achieved in the current implementation by checking the load on the machine and comparing it with a threshold value. The computation of the threshold value is based on the global load. However, in a network of machines with varying speeds, this measure alone is inadequate for measuring the current load. The load on a machine is defined as the exponentially smoothed average number of jobs in the run queue over the past 5 minutes. However, during the experiment it was observed that it is the fastest machine that decides not to accept more jobs, instead of the slower machines. Since the faster machines can process jobs better than slower machines, even while being loaded, the load average may not be an adequate measure for deciding whether to accept more work. This aspect could be investigated further.

CPU intensive tasks, such as a Monte Carlo simulations, which run for several hours on a machine should be protected from failure after a significant amount of the computation has been performed. In a distributed implementation, this means that the client which started the computation becomes a single point of failure. In order to avoid the single point of failure, an effective method would be to start up a backup client in parallel on another machine. However, necessary check pointing and synchronization would need to be performed to make this system perform correctly.

As the number of machines involved in the computation increases significantly, on the order of 50, the scheme using heavy weight child processes to achieve asynchronous communication becomes inefficient. Alternate schemes could be investigated. One alternative is to have a single process poll all communication ports to determine which one is ready. Child processes could be forked off to read from ready ports. Furthermore, on Sun workstations with light weight process support, threads could be used as an alternative to heavy weight processes [Bur93a, Bur93b].

BIBLIOGRAPHY

[ABM+92] M. J. Atallah, C. L. Black, D.C. Marinescu, H. J. Siegel, and T. L. Casavant. Models and algorithms for coscheduling compute-intensive tasks on a network of workstations. *Journal of Parallel and Distributed Computing*, 16:319–327, 1992.

[AC88] R. Alonso and L. L. Cova. Sharing jobs among independently owned processors. In *Proc. of 8th International Conference on Dist. Comp. Systems*, pages 282–288. IEEE Computer Society, 1988.

[AHM+93] G. S. Almasi, D. Hale, T. McLuckie, J. Bell, and A Gordon. Parallel distributed seismic migration. *Concurrency: Practice and Experience*, 5:105–131, 1993.

[Bur93a] Sean Burke. Parallel processing on your network, Part I. *Sun Expert*, pages 65–69, May 1993.

[Bur93b] Sean Burke. Parallel processing on your network, Part II. *Sun Expert*, pages 62–65, July 1993.

[DO87] F. Douglis and J. Ousterhout. Process migration in the Sprite operating system. *Proc. of the 7th International Conference of Dist. Comp. Systems*, pages 18–25, 1987.

[DO91] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

[GDI93] K. K. Goswami, M. Devarakonda, and R. K. Iyer. Prediction based dynamic load sharing heuristics. *IEEE Trans. on Parallel and Distributed Systems*, 4:638–648, 1993.

[Hag86] R. Hagmann. Process server: Sharing processing power in a workstation environment. In *Proc. of 7th International Conference on Dist. Comp. Systems*, pages 18–25. IEEE Computer Society, 1986.

[LK89] L.Klienrock and W. Korfhage. Collecting unused processing capacity: An analysis of transient distributed systems. In *Proc. of 9th IEEE Distributed Computing Conference*, pages 482–489. IEEE, 1989.

[LLM88] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - a hunter of idle workstations. In *Proc. of 8th International Conference on Dist. Comp. Systems*, pages 104–111. IEEE Computer Society, 1988.

[Mur92] P. M. Murray. Leveraged computing: A task distribution protocol. In *Proc. of the 12th International Conference on Dist. Comp. Systems*, pages 563–570, 1992.

[Mut91] Matt W. Mutka. An examination of strategies for estimating capacity to share among private worstations. In *Proc. of the 1991 SIGSMALL/PC Symposium on Small Systems*, pages 53–61. ACM Press, 1991.

[NH86] D. L. Nofziger and A. G. Hornsby. A microcomputer-based management tool for chemical movement in soil. *Applied Agricultural Research*, 1(1):50–56, 1986.

[OACT92] R. A. Olsson, G. R. Andrews, M. H. Coffin, and G. M. Townsend. A language for parallel and distributed programming. Technical Report TR 92-09, Department of Computer Science - Univ. of Arizona, 1992.

[PC91] P.Krueger and R. Chawala. The stealth distributed scheduler. In *Proc. of 11th International Conference on Dist. Comp. Systems*, pages 336–343. IEEE Computer Society, 1991.

[PL88] P.Krueger and M. Linvy. A comparison of preemptive and non-preemptive load distributing. In *Proc. of 8th International Conference on Dist. Comp. Systems*, pages 123–130. IEEE Computer Society, 1988.

[SKS92] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *Computer*, 25(12):33–44, 1992.

[Ste90] W. R. Stevens. *UNIX Network Programming.* Englewood Cliffs, N. J. : Prentice Hall, 1990.

[TL88] M. M. Theimer and K. A. Lantz. Finding idle machines in a workstation-based distributed system. In *Proc. of 8th International Conference on Dist. Comp. Systems*, pages 112–122. IEEE Computer Society, 1988.

[WHH92] C. A. Waldspurger, T. Hogg, and B. A. Huberman. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18:103–117, 1992.

APPENDIX A

USING THE SYSTEM

First, the server processes are started on the machines which participate in the computation. The server program includes the files server.c and h_serv.h. This program is compiled using

cc -o server server.c

For installing the fault-recovery feature, the keyword FAULT_RECOVERY should be defined. This is done from the command line by

cc -o server -DFAULT_RECOVERY server.c

For debugging purposes the random numbers passed between the machines can be displayed by defining the keyword RANDOM.

Once the server code is compiled it can be started up on a machine as a background process by

server &

During initialization, the server needs the data file SERVERS.DB to determine which workstations are participating in the distributed computation.
The client code is compiled using the Makefile given in Appendix C. If the keyword DISTRIB_OTHERS is defined, the client program will be compiled for distributed computation. Otherwise, the code will be compiled to run on a single machine.

When the program is compiled to run on a single machine it is invoked with

cmls92b inputfilename screen_output

When the program is compiled for distributed computation, two executables are created and used. The first executable code (*cmls92m*) acts as the client requesting service from the server. The second executable code (*cmls92rem*) is used by the remote machine for processing the data (*slave*).

The source code for both executable files is the same. However, if the keyword REMOTE is defined then the executable code for the slave is created. Otherwise, the code for the local client is created.

Before invoking the client, the slave code and the data files should be made available to the remote machine. This can be done either by using a common file server or by manually copying the slave code and the data files to the remote machines. The input file and the parameter file are the data files needed by the remote machine.

The local client is invoked using

cmls92m inputfilename replications

The argument *replications* specifies the number of replications to be performed per remote invocation.

When the slave code is loaded from a common file server, the following precautions need to be observed.

- The path for the slave code and the data files should be the same for all machines. If the path for the executable code is not same for all machines, then the remote server will not be able to locate the code.

- The servers on the individual machines should not be started up from the same directory. While the remote computation is in progress some temporary files are created. If all servers are started up in the same directory, then the temporary files will be corrupted.

If the slave code and the data files are manually copied onto the remote machine, then it can be copied into the directory in which the server is started up from.

If a server exits due to an error, then the machine on which the server was running should be checked to see if the shared memory segment and the semaphore have been removed from the IPC table. This can be done using the command

ipcs

If there are shared memory segments or semaphores in the table, they can be removed using the command

ipcrm

This step needs to be performed only in case of an error.

APPENDIX B

CLIENT CODE

The client program is given in this section. The Makefile given in Appendix C is used to compile the client program.

The steps performed by the client are:

- Set-up a connection to the local server and request for service.
- Read the result of the local server's inquiry to other machines.
- If there are no remote servers willing to accept the request, then terminate. Otherwise, inform the server the slave code and the output file name.
- Pack each data block into a character array and pass it to the local server.
- If all data blocks are transferred, wait for the server to complete processing.
- When the server completes processing, exit with a message of successful completion.

The source code is given below in C code.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

#define  TRUE  1
#define  FALSE 0
#define  BUFSIZE    512

#ifndef REMOTE
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/stream.h>
#include <sys/param.h>
#include <unistd.h>
```

```
#define  PORT_NUM   12872

#define  SERVER  'S'
#define  CLIENT  'C'

#define  HOST_NAME_LEN    35
#define  ADDR_LEN         35
char     outfilename[81];
char     fopen_mode[5];
int      num_of_remote_sims;
int      screen_output = 0;
char     myself[MAXHOSTNAMELEN];
#else

// Remote (Slave process)
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

/*Random number generator variables */
extern INT RAN1_IFF, RAN1_IX1, RAN1_IX2, RAN1_IX3;
extern float RAN1_R[98];

int      SHMKEY;     // Key for shared memory passed from parent
int      shmid;      // Shared memory id
char     *shmsg;     // Pointer to shared memory

#endif

#define PERMS                      0600
#define MAX_STREAM_SIZE            4096

#define DATA_OVER        "DATA_OVER"       // input file over sysmbol
#define DATA_AVAILABLE   "DATA_AVAILABLE"  // input data not over symbol

#define STR_L_LEN                  21
#define NAME_LEN                   65
#define SYMB_LEN                   11
#define ARRAY_LEN                  10
#define CHEM_NAME_LEN              40
#define LINE                       80
#define DAYS_PER_YEAR              367

extern  "C" void gethostname(char*, int);
```

```
typedef struct crop_info {
  char  name[STR_L_LEN];
  char  coef_file[NAME_LEN];
  float coef[DAYS_PER_YEAR];
  int   earliest_plant_day;
  int   latest_plant_day;
  } CROP_INFO;

typedef struct infiltration {
  int  source;
  char parm_file[NAME_LEN];
  char daily_weather_file[NAME_LEN];
  char daily_rain_file[NAME_LEN];
  int  estimator;
  } INFIL_INFO;

typedef struct evap {
  int   source;
  char  parm_file[NAME_LEN];
  char  daily_weather_file[NAME_LEN];
  char  daily_et_file[NAME_LEN];
  float par1;
  float par2;
  char  pan_file[NAME_LEN];
  int   estimator;
  } ET_INFO;

typedef struct simulation {
  char input_file[NAME_LEN];
  char output_file[NAME_LEN];
  int  begin_day;
  int  begin_year;
  int  end_day;
  int  end_year;
  int  num_sims;
  int  soil_chem_combinations;
  } SIM_INFO;

typedef struct total_water {
  float  et;
  float  pet;
  float  rain;
  float  infil;
  float  irrig;
  float  runoff;
  } H2O;

typedef struct day_depth_amount {
  int   day;
  float depth;
  float amount;
  } DAY_DEPTH_AMT;

typedef struct out_selections {
  DAY_DEPTH_AMT at_depth[ARRAY_LEN];
  int           no_depth;
  DAY_DEPTH_AMT at_time[ARRAY_LEN];
  int           no_time;
  int           amount;
  int           infilET;
  char          depth_units;
  H2O           water;
  } OUT_SEL;

typedef struct periodic_info {
  int   begin_day;
  int   end_day;
  int   period;
  float amount;
  } PERIODIC_INFO;

typedef struct demand_info {
  float critical_depletion;
  float efficiency;
  float amount;
  int   begin_day;
  int   end_day;
  } DEMAND_INFO;

typedef union periodic_demand {
  PERIODIC_INFO   peri;
  DEMAND_INFO     demd;
  char            file[NAME_LEN];
  } PERIODIC_DEMAND;

typedef struct irri_info {
  int             type;
  PERIODIC_DEMAND datum;
  char            amount_units;
  } IRRI_INFO;
```

```
typedef struct soil_prop {
  float depth;
  float oc;
  float bd;
  float fc;
  float pwp;
  float sat;
  float koc;
  float half_life;
  float kd;
  } SOIL_PROP;

typedef struct soil_info  {
  char      name[NAME_LEN];
  float     curve_no;
  float     root_depth;
  SOIL_PROP prop[LMAX+3];
  int       no_horizons;
  } SOIL_INFO;

typedef struct appl_info {
  int   earliest_day;
  int   latest_day;
  int   day;
  int   year;
  int   date_type;
  float depth;
  float amount;
  } APP_INFO;

typedef struct sys_info {
  char      index[STR_L_LEN];
  SOIL_INFO soil;
  char      chem_name[CHEM_NAME_LEN];
  APP_INFO  appl;
  char      depth_units;
  int       resampling;
  } SYS_INFO;

typedef struct weather {
  int   num_days;
  int   first_year;
  char  station_id[NAME_LEN];
  char  station_name[NAME_LEN];
  float latitude;
  char  hemisphere;
  float longitude;
  char  lon_side;
  float elevation;
  char  elev_units[STR_L_LEN];
  char  temp_units;
  char  infil_units;
  } WTH_INFO;

typedef struct check_info {
  int  need_check;
  char file[NAME_LEN];
  } CHECK_INFO;

typedef struct prop_infor {
float  oc;
float  bd;
float  fc;
float  pwp;
float  sat;
} PROP_INFOR;

typedef struct file_info {
char file[NAME_LEN];
int  no_header;
}  FILE_INFO;

typedef struct resample_soil {
SOIL_PROP property[LMAX];
int       horizons;
} RESAMPLE_SOIL;

/***************************************************************************
*                  ------ Function Prototypes ------                        *
***************************************************************************/
#ifndef  REMOTE
int  call_simulate(SIM_INFO*, SYS_INFO*, INFIL_INFO*, ET_INFO*, IRRI_INFO*,
                   OUT_SEL*, long, CROP_INFO*, int, char*, FILE*, FILE *);
char *convert_to_byte_stream(SIM_INFO*, SYS_INFO*, INFIL_INFO*, ET_INFO*,
                             IRRI_INFO*, OUT_SEL*, long, CROP_INFO*, int,
                             char*, char *);
int   initialize_sock(int *);
void  read_stream(int, char *, int);
void  write_stream(int, char *, int);
char *pack_string(char *, char *, int);
char *pack_long(char *, long);
```

```
#else
void  get_shared_mem();
void  set_random_numbers_back();
void  unpack_msg(char *, SIM_INFO*, SYS_INFO*, INFIL_INFO*, ET_INFO*,
                 IRRI_INFO*, OUT_SEL*, long *, CROP_INFO*, int *, char*,
                 int *, int *);
void  unpack_msg_execute(FILE *);
char *unpack_string(char *, char *, int);
char *unpack_int(char *, int *);
char *unpack_float(char *, float *);
char *unpack_long(char *, long *);
#endif

char *pack_int(char *, int);
char *pack_float(char *, float);
int   simulate(SIM_INFO*, SYS_INFO*, INFIL_INFO*, ET_INFO*, IRRI_INFO*,
             OUT_SEL*, long, CROP_INFO*, int, char*, FILE*, FILE*, int);

#ifndef REMOTE
/***************************************************************************
*                           ------- main -------                           *
*      This routine becomes the main if the keyword 'REMOTE' is not        *
*  defined at compilation.  Hence, this routine is the main routine for    *
*  the client which requests service.                                      *
*      Belongs to:      Client Code.                                       *
***************************************************************************/
void main(int argc, char *argv[])
  {
  FILE *ifp;   /*  file pointer to input file   */
  FILE *out;   /*  file pointer to output file   */

  OUT_SEL     output;
  CROP_INFO   crop;
  INFIL_INFO  infil;
  ET_INFO     et;
  SIM_INFO    sim;
  IRRI_INFO   irri;
  SYS_INFO    sys;
  CHECK_INFO  check;
  FILE_INFO   base[MAX_OUTFILE];
  char        fir_str[STR_L_LEN];
  char        low_fir_str[STR_L_LEN];
  char        row[LINE];
  long int    offset;
  int         find = FALSE;
  int         block = 1;
#ifndef DISTRIB_OTHERS

//  Single machine processing.
    if(argc < 3){
        printf("\n%s%s%s\n\n\n","usage: ",argv[0], " inputfile",
                                                       " screen_output");
        exit(1);
        }

  screen_output = atoi(argv[2]);
#else

//  Distributed computing.
    if(argc < 3){
        printf("\nUsage: %s inputfile Replications\n\n",
                                                              argv[0]);
        printf("Replications - # of replications at the remote m/c per \
                                                      remote call.\n\n");
        exit(1);
        }
    num_of_remote_sims = atoi(argv[2]);
    if(num_of_remote_sims <= 0){
    printf("\nNumber of remote replications should be >0\n\n");
    exit(1);
    }
#endif

1.  Read input file and get the general information parameters.

                          /* call the simulation routine */
2.  call_simulate(&sim,&sys,&infil,&et,&irri,&output,offset,&crop,
                                       block, fir_ str, ifp, out);

3.  If there is another general information block, go to step 1.

4.  Close the input and output files.

  }

/***************************************************************************
*                      ----- call_simulate -----                           *
*     If a keyword 'DISTRIB_OTHERS' is defined, this routine will connect*
*  with the local server and distribute the computation across the network*
*  If this keyword is not defined, then it will repeatedly call the simu- *
*  late routine to do the computation locally.                            *
*      Belongs to:      Client Code.                                         *
*************************************************************************/
int call_simulate(SIM_INFO *sim,     SYS_INFO  *sys,   INFIL_INFO *infil,
                  ET_INFO  *et,      IRRI_INFO *irrig, OUT_SEL    *output,
                  long      offset,  CROP_INFO *crop,  int         block,
                  char     *fir_str, FILE      *in,    FILE       *out)
  {
  int    sock;
  int    sim_number = 0;
  int    num_sims;
  int    NumRemSims;         // Number of remote replications in one call
  int    status;
  char   msg_to_send[MAX_STREAM_SIZE], *msg, *lst;
  char   buff[BUFSIZE];

  if((et->source == GENERATED) && (infil->source == GENERATED))
    {
    num_sims = sim->num_sims;
    }
  else
    {
    num_sims = 1;
    }

  printf("\nSimulation will be performed for %d times\n",num_sims);

#ifdef DISTRIB_OTHERS

// if keyword DISTRIB_OTHERS is defined, distribute across the network.

  initialize_sock(&sock);

  memset(buff, (char)0, sizeof(buff));
  sprintf(buff, "%s", DATA_AVAILABLE);

  memset(msg_to_send, (char)0, sizeof(msg_to_send));

  lst = convert_to_byte_stream(sim,sys,infil,et,irrig,output,offset,
                                  crop, block,fir_str, msg_to_send);

  NumRemSims = num_of_remote_sims;
  if((num_sims - sim_number) < NumRemSims)
    NumRemSims = num_sims - sim_number;

  for(sim_number = 0; sim_number < num_sims; sim_number +=
                                         num_of_remote_sims){
```

```
      if((num_sims - sim_number) < NumRemSims)
         NumRemSims = num_sims - sim_number;

      msg = pack_int(lst, NumRemSims);
      msg = pack_int(msg, sim_number);

      printf("\nSimulation # = %d RemoteSims = %d", sim_number+1,

//             inform availability of data
      write_stream(sock, buff, BUFSIZE);

//             write data to the server
      write_stream(sock, msg_to_send, MAX_STREAM_SIZE);
      }

    memset(buff, (char)0, sizeof(buff));
    sprintf(buff, "%s", DATA_OVER);
    write_stream(sock, buff, BUFSIZE);

    printf("\nClient: Waiting for the server to complete computation.\n");
    read_stream(sock, buff, BUFSIZE);
    sscanf(buff, "%d ", &status);
    printf("\n%s\n\n",buff);
    if(!status)
      return FALSE;
    close(sock);

#else

// Single machine computing

  for (sim_number = 0; sim_number < num_sims; sim_number++) {
      printf("\nSimulation = %d\n", sim_number+1);
      simulate(sim, sys, infil, et, irrig, output, offset, crop,
                                 block, fir_str, in, out, sim_number);
      }

#endif

  return TRUE;
  }

#ifdef DISTRIB_OTHERS
/**************************************************************************
*                  ------ convert_to_byte_stream ------                   *
*     This routine packs the parameters (arguments to the simulate rou-   *
*  tine) into a character array.  This character array is then send to the*
*  remote machine.                                                         *
*      Belongs to:      Client Code.                                       *
*************************************************************************/
char *convert_to_byte_stream(
               SIM_INFO   *sim,     SYS_INFO  *sys,    INFIL_INFO  *infil,
               ET_INFO    *et,      IRRI_INFO *irrig,  OUT_SEL     *output,
               long        offset,  CROP_INFO *crop,   int          block,
               char       *fir_str, char      *msg_to_send)
  {
  int    i;
  char   *msg, str[NAME_LEN];

  msg = msg_to_send;

          // pack SIM_INFO
  msg = pack_string(msg, sim->input_file, NAME_LEN);
  msg = pack_string(msg, sim->output_file, NAME_LEN);
  msg = pack_int(msg, sim->begin_day);
  msg = pack_int(msg, sim->begin_year);
  msg = pack_int(msg, sim->end_day);
  msg = pack_int(msg, sim->end_year);
  msg = pack_int(msg, sim->num_sims);
  msg = pack_int(msg, sim->soil_chem_combinations);

         // pack SIM_INFO
  msg = pack_string(msg, sys->index, STR_L_LEN);
  msg = pack_string(msg, sys->soil.name, NAME_LEN);
  msg = pack_float(msg, sys->soil.curve_no);
  msg = pack_float(msg, sys->soil.root_depth);

  for(i = 0; i < LMAX+3; i++) {
     msg = pack_float(msg, sys->soil.prop[i].depth);
     msg = pack_float(msg, sys->soil.prop[i].oc);
     msg = pack_float(msg, sys->soil.prop[i].bd);
     msg = pack_float(msg, sys->soil.prop[i].fc);
     msg = pack_float(msg, sys->soil.prop[i].pwp);
     msg = pack_float(msg, sys->soil.prop[i].sat);
     msg = pack_float(msg, sys->soil.prop[i].koc);
     msg = pack_float(msg, sys->soil.prop[i].half_life);
     msg = pack_float(msg, sys->soil.prop[i].kd);
     }
  msg = pack_int(msg, sys->soil.no_horizons);
  msg = pack_string(msg, sys->chem_name, CHEM_NAME_LEN);
  msg = pack_int(msg, sys->appl.earliest_day);
  msg = pack_int(msg, sys->appl.latest_day);
msg = pack_int(msg, sys->appl.day);
msg = pack_int(msg, sys->appl.year);
msg = pack_int(msg, sys->appl.date_type);
msg = pack_float(msg, sys->appl.depth);
msg = pack_float(msg, sys->appl.amount);
(*msg) = sys->depth_units;  msg++;
msg = pack_int(msg, sys->resampling);

      // pack INFIL_INFO
msg = pack_int(msg, infil->source);
msg = pack_string(msg, infil->parm_file, NAME_LEN);
msg = pack_string(msg, infil->daily_weather_file, NAME_LEN);
msg = pack_string(msg, infil->daily_rain_file, NAME_LEN);
msg = pack_int(msg, infil->estimator);

      // pack ET_INFO
msg = pack_int(msg, et->source);
msg = pack_string(msg, et->parm_file, NAME_LEN);
msg = pack_string(msg, et->daily_weather_file, NAME_LEN);
msg = pack_string(msg, et->daily_et_file, NAME_LEN);
msg = pack_float(msg, et->par1);
msg = pack_float(msg, et->par2);
msg = pack_string(msg, et->pan_file, NAME_LEN);
msg = pack_int(msg, et->estimator);

      // pack IRRI_INFO
msg = pack_int(msg, irrig->type);
msg = pack_int(msg, irrig->datum.peri.begin_day);
msg = pack_int(msg, irrig->datum.peri.end_day);
msg = pack_int(msg, irrig->datum.peri.period);
msg = pack_float(msg, irrig->datum.peri.amount);
msg = pack_float(msg, irrig->datum.demd.critical_depletion);
msg = pack_float(msg, irrig->datum.demd.efficiency);
msg = pack_float(msg, irrig->datum.demd.amount);
msg = pack_int(msg, irrig->datum.demd.begin_day);
msg = pack_int(msg, irrig->datum.demd.end_day);
msg = pack_string(msg, irrig->datum.file, NAME_LEN);
(*msg) = irrig->amount_units;  msg++;

      // pack OUT_SEL
for(i = 0; i < ARRAY_LEN; i++) {
   msg = pack_int(msg, output->at_depth[i].day);
   msg = pack_float(msg, output->at_depth[i].depth);
   msg = pack_float(msg, output->at_depth[i].amount);
   }
msg = pack_int(msg, output->no_depth);
```

```
  for(i = 0; i < ARRAY_LEN; i++) {
     msg = pack_int(msg, output->at_time[i].day);
     msg = pack_float(msg, output->at_time[i].depth);
     msg = pack_float(msg, output->at_time[i].amount);
     }
  msg = pack_int(msg, output->no_time);
  msg = pack_int(msg, output->amount);
  msg = pack_int(msg, output->infilET);
  (*msg) = output->depth_units;  msg++;
  msg = pack_float(msg, output->water.et);
  msg = pack_float(msg, output->water.pet);
  msg = pack_float(msg, output->water.rain);
  msg = pack_float(msg, output->water.infil);
  msg = pack_float(msg, output->water.irrig);
  msg = pack_float(msg, output->water.runoff);

  msg = pack_long(msg, offset);

        // pack CROP_INFO
  msg = pack_string(msg, crop->name, STR_L_LEN);
  msg = pack_string(msg, crop->coef_file, NAME_LEN);
  for(i = 0; i < DAYS_PER_YEAR; i++)
     msg = pack_float(msg, crop->coef[i]);
  msg = pack_int(msg, crop->earliest_plant_day);
  msg = pack_int(msg, crop->latest_plant_day);

  msg = pack_int(msg, block);

  msg = pack_string(msg, fir_str, STR_L_LEN);

  msg = pack_int(msg, no_header);

  msg = pack_int(msg, change_weather);

  return msg;
  }

/***************************************************************************
*                         ------ pack_string ------                        *
*      This routine packs a string into the character array.               *
*      Belongs to:      Client Code.                                       *
***************************************************************************/
char * pack_string(char *msg, char *str, int strlen)
  {
  int i;
  for(i = 0; i < strlen; i++) {
    (*msg) = (*str);
    msg++;
    str++;
    }
  return msg;
  }

/****************************************************************************
*                          ------ pack_long  ------                        *
*      This routine packs a long value into the character array.           *
*      Belongs to:      Client Code.                                       *
****************************************************************************/
char * pack_long(char *msg, long num)
  {
  int i;
  long tmp;
  char *ptr;

  tmp = num;
  ptr  = (char *)&tmp;
  for(i = 0; i < sizeof(long); i++) {
    (*msg) = (*ptr);
    msg++;
    ptr++;
    }
  return msg;
  }

/****************************************************************************
*                          ------ initialize_sock ------                   *
*      This routine will set up the initial connection with the local      *
*  server, and requests for service.                                       *
*      Belongs to:      Client Code.                                       *
****************************************************************************/
int initialize_sock(int *sd)
  {
  int sock;
  int flag = FALSE;
  char *addr;
  struct sockaddr_in  sock_cli;
  struct hostent   *hp;
  struct in_addr *ptr, in_a;
  char   buff[BUFSIZE];
  char   own_server[MAXHOSTNAMELEN];
```

```
gethostname(own_server, 32);
hp = gethostbyname(own_server);
printf("\nOwn_server = %s ", own_server);
ptr = &in_a;
memcpy(ptr, hp->h_addr, sizeof(struct in_addr));
addr = inet_ntoa(in_a);
printf("\nMyself = %s", addr);
strcpy(myself, addr);

sock_cli.sin_family     = AF_INET;
sock_cli.sin_port       = PORT_NUM;

memcpy(&sock_cli.sin_addr, hp->h_addr, hp->h_length);

if((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
  perror("Client: Error in creating socket ");
  exit(1);
  }

if(connect(sock, (struct sockaddr *)&sock_cli, sizeof(sock_cli)) < 0){
  sprintf(buff,"Connecting to own server - %s", own_server);
  perror(buff);
  close(sock);
  exit(1);
  }

*sd = sock;

memset(buff, (char) 0, BUFSIZE);
sprintf(buff, "%c", CLIENT);
write_stream(sock, buff, BUFSIZE);

read_stream(sock, buff, BUFSIZE);

sscanf(buff, "%d", &flag);
printf("\nClient: Reply from server-> %d\n", flag);
if(flag <= 0) {
  printf("\nServer Cannot Distribute \n");
  close(sock);
  exit(1);
  }

memset(buff, (char) 0, BUFSIZE);
sprintf(buff, "cmls92rem %d %s %s", old_seed, outfilename, fopen_mode);
write_stream(sock, buff, BUFSIZE);
```

```
  return flag;
  }

/*************************************************************************
*                        ------ read_stream ------                       *
*      This routine is used to read from a file discriptor, opened as a  *
*  stream end point.(socket, file etc)                                   *
*      Belongs to:      Client Code.                                     *
*************************************************************************/
void read_stream(int stream, char *buffer, int numbytes)
  {
  int nread = 0;
  char *str;

  memset(buffer, (char)0, numbytes);
  str = buffer;
  while((nread += read(stream, str, (numbytes - nread))) < numbytes)
     str = buffer + nread;
  }

/*************************************************************************
*                        ------ write_stream ------                      *
*      This routine is used to write from a file discriptor, opened as a *
*  stream end point.(socket, file etc)                                   *
*      Belongs to:      Client Code.                                     *
*************************************************************************/
void write_stream(int stream, char *buffer, int numbytes)
  {
  int nwrite = 0;
  char *str;

  str   = buffer;
  while((nwrite +=  write(stream, str, (numbytes - nwrite))) < numbytes)
    str = buffer + nwrite;
  }

#endif
#endif

#ifdef REMOTE
/*************************************************************************
                              ------ main ------                         *
*      If the keyword 'REMOTE' is defined at compilation, this routine   *
*  becomes the main routine. Hence, this routine is the main routine, for *
*  the remote slave code.                                                *
*      Belongs to:      Remote Slave Code                                *
```

```
*************************************************************************/
void main(int argc, char *argv[])
  {
  FILE *fp_out;

  if((fp_out = fopen(argv[2], "wt")) == NULL) {
    fprintf(stderr,"\nREMOTE: Error opening output file - %s\n\n",argv[2]);
    exit(1);
    }

  printf("\nNOW THE REMOTE PROGRAM.........................");
  SHMKEY = atoi(argv[1]);
  get_shared_mem();
  unpack_msg_execute(fp_out);

  set_random_numbers_back();

  printf("\nREMOTE: EXECUTION OVER\n");

  fclose(fp_out);
  }

/*************************************************************************
*                        ------ get_shared_mem ------                    *
*     This routine gets the shared memory segment, created by the remote *
*  server.                                                               *
*     Belongs to:      Remote Slave Code                                 *
*************************************************************************/
void get_shared_mem()
  {

  if(( shmid = shmget(SHMKEY, MAX_STREAM_SIZE+BUFSIZE,  0))  < 0) {
    fprintf(stderr, "\nREMOTE:Error getting shared memory\n\n");
    exit(1);
    }

  if(( shmsg = (char *) shmat(shmid, (char *) 0, 0)) == (char *) -1)  {
     fprintf(stderr, "\nREMOTE: cannot attach shared memory\n\n");
     exit(1);
     }
  }

/*************************************************************************
*                  ------ set_random_numbers_back ------                 *
*     The random numbers at the end of the computation is set back in the*
*  shared memory segment, by this routine.                               *
*      Belongs to:      Remote Slave Code                                  *
**************************************************************************/
void set_random_numbers_back()
  {
  int  j;
  char *msg;

  msg  = shmsg+MAX_STREAM_SIZE+sizeof(float);
                 /* sizeof float is left for the parent to fill in the
                    Load value */

  msg = pack_int(msg, RAN1_IFF);
  msg = pack_int(msg, RAN1_IX1);
  msg = pack_int(msg, RAN1_IX2);
  msg = pack_int(msg, RAN1_IX3);
  msg = pack_int(msg, seed);
#ifdef RANDOM
  printf("\n%d %d %d %d\n",RAN1_IFF, RAN1_IX1, RAN1_IX2, RAN1_IX3,seed);
#endif
  for(j=0;j<98;j++){
     msg = pack_float(msg, RAN1_R[j]);
#ifdef RANDOM
     printf(" %.4f", RAN1_R[j]);
     if((j+1)%7==0) printf("\n");
#endif
     }
  return;
  }

/*************************************************************************
*                      ------ unpack_msg_execute ------                  *
*      The character array send to the remote slave is unpacked and the  *
*  routine simulate is called to carry out the computation.              *
*      Belongs to:      Remote Slave Code                                *
*************************************************************************/
void unpack_msg_execute( FILE *fp_out)
  {
  SIM_INFO   sim;                SYS_INFO   sys;       INFIL_INFO  infil;
  ET_INFO    et;                 IRRI_INFO   irrig;    OUT_SEL     output;
  long       offset;             CROP_INFO  crop;
  char       fir_str[STR_L_LEN], *msg;
  FILE       *fp_in;
  int i, block, sim_number,  num_of_sims;

  msg = shmsg;
  unpack_msg(msg, &sim, &sys, &infil, &et, &irrig, &output, &offset, &crop,
                        &block, fir_str, &sim_number, &num_of_sims);

  if((fp_in = fopen(sim.input_file, "rt")) == NULL) {
    fprintf(stderr, "\nError opening input file %s\n\n", sim.input_file);
    exit(1);
    }

  for(i = 0; i< num_of_sims; i++) {
    printf("\nSimulation Number = %d\n", sim_number+1+i);
    simulate(&sim, &sys, &infil, &et, &irrig, &output, offset, &crop,
                            block, fir_str, fp_in, fp_out, sim_number+i);
    }

  fclose(fp_in);
  return;
  }

/***************************************************************************
*                         ------ unpack_msg ------                         *
*      The character array send to the remote slave is unpacked using this*
*  routine.                                                                *
*      Belongs to:      Remote Slave Code                                  *
***************************************************************************/
void unpack_msg(char        *start,    SIM_INFO *sim,     SYS_INFO  *sys,
                INFIL_INFO  *infil,    ET_INFO  *et,      IRRI_INFO *irrig,
                OUT_SEL     *output,   long     *offset,  CROP_INFO *crop,
                int         *block,    char     *fir_str,
                int         *sim_number, int    *num_of_sims)
  {
  int i = 0;
  int loc_num;
  char *msg;
  int j = 0;
  int r1;
  float tmp_fl;

  msg = start;

          // unpack SIM_INFO
  msg = unpack_string(msg, sim->input_file, NAME_LEN);
  msg = unpack_string(msg, sim->output_file, NAME_LEN);
  msg = unpack_int(msg, &(sim->begin_day));
  msg = unpack_int(msg, &(sim->begin_year));
  msg = unpack_int(msg, &(sim->end_day));
  msg = unpack_int(msg, &(sim->end_year));
msg = unpack_int(msg, &(sim->num_sims));
msg = unpack_int(msg, &(sim->soil_chem_combinations));

         // unpack SIM_INFO
msg = unpack_string(msg, sys->index, STR_L_LEN);
msg = unpack_string(msg, sys->soil.name, NAME_LEN);
msg = unpack_float(msg, &(sys->soil.curve_no));
msg = unpack_float(msg, &(sys->soil.root_depth));
for(i = 0; i < LMAX+3; i++) {
   msg = unpack_float(msg, &(sys->soil.prop[i].depth));
   msg = unpack_float(msg, &(sys->soil.prop[i].oc));
   msg = unpack_float(msg, &(sys->soil.prop[i].bd));
   msg = unpack_float(msg, &(sys->soil.prop[i].fc));
   msg = unpack_float(msg, &(sys->soil.prop[i].pwp));
   msg = unpack_float(msg, &(sys->soil.prop[i].sat));
   msg = unpack_float(msg, &(sys->soil.prop[i].koc));
   msg = unpack_float(msg, &(sys->soil.prop[i].half_life));
   msg = unpack_float(msg, &(sys->soil.prop[i].kd));
   }
msg = unpack_int(msg, &(sys->soil.no_horizons));
msg = unpack_string(msg, sys->chem_name, CHEM_NAME_LEN);
msg = unpack_int(msg, &(sys->appl.earliest_day));
msg = unpack_int(msg, &(sys->appl.latest_day));
msg = unpack_int(msg, &(sys->appl.day));
msg = unpack_int(msg, &(sys->appl.year));
msg = unpack_int(msg, &(sys->appl.date_type));
msg = unpack_float(msg, &(sys->appl.depth));
msg = unpack_float(msg, &(sys->appl.amount));
sys->depth_units = *(msg);  msg++;
msg = unpack_int(msg, &(sys->resampling));

       // unpack INFIL_INFO
msg = unpack_int(msg, &(infil->source));
msg = unpack_string(msg, infil->parm_file, NAME_LEN);
msg = unpack_string(msg, infil->daily_weather_file, NAME_LEN);
msg = unpack_string(msg, infil->daily_rain_file, NAME_LEN);
msg = unpack_int(msg, &(infil->estimator));

       // unpack ET_INFO
msg = unpack_int(msg, &(et->source));
msg = unpack_string(msg, et->parm_file, NAME_LEN);
msg = unpack_string(msg, et->daily_weather_file, NAME_LEN);
msg = unpack_string(msg, et->daily_et_file, NAME_LEN);
msg = unpack_float(msg, &(et->par1));
msg = unpack_float(msg, &(et->par2));
msg = unpack_string(msg, et->pan_file, NAME_LEN);
msg = unpack_int(msg, &(et->estimator));

        // unpack IRRI_INFO
msg = unpack_int(msg, &(irrig->type));
msg = unpack_int(msg, &(irrig->datum.peri.begin_day));
msg = unpack_int(msg, &(irrig->datum.peri.end_day));
msg = unpack_int(msg, &(irrig->datum.peri.period));
msg = unpack_float(msg, &(irrig->datum.peri.amount));
msg = unpack_float(msg, &(irrig->datum.demd.critical_depletion));
msg = unpack_float(msg, &(irrig->datum.demd.efficiency));
msg = unpack_float(msg, &(irrig->datum.demd.amount));
msg = unpack_int(msg, &(irrig->datum.demd.begin_day));
msg = unpack_int(msg, &(irrig->datum.demd.end_day));
msg = unpack_string(msg, irrig->datum.file, NAME_LEN);
irrig->amount_units = (*msg);  msg++;

        // unpack OUT_SEL
for(i = 0; i < ARRAY_LEN; i++) {
   msg = unpack_int(msg, &(output->at_depth[i].day));
   msg = unpack_float(msg, &(output->at_depth[i].depth));
   msg = unpack_float(msg, &(output->at_depth[i].amount));
   }
msg = unpack_int(msg, &(output->no_depth));
for(i = 0; i < ARRAY_LEN; i++) {
   msg = unpack_int(msg, &(output->at_time[i].day));
   msg = unpack_float(msg, &(output->at_time[i].depth));
   msg = unpack_float(msg, &(output->at_time[i].amount));
   }
msg = unpack_int(msg, &(output->no_time));
msg = unpack_int(msg, &(output->amount));
msg = unpack_int(msg, &(output->infilET));
output->depth_units = (*msg);  msg++;
msg = unpack_float(msg, &(output->water.et));
msg = unpack_float(msg, &(output->water.pet));
msg = unpack_float(msg, &(output->water.rain));
msg = unpack_float(msg, &(output->water.infil));
msg = unpack_float(msg, &(output->water.irrig));
msg = unpack_float(msg, &(output->water.runoff));

msg = unpack_long(msg, offset);

      // unpack CROP_INFO
msg = unpack_string(msg, crop->name, STR_L_LEN);
msg = unpack_string(msg, crop->coef_file, NAME_LEN);
for(i = 0; i < DAYS_PER_YEAR; i++)
   msg = unpack_float(msg, &(crop->coef[i]));
```

```
  msg = unpack_int(msg, &(crop->earliest_plant_day));
  msg = unpack_int(msg, &(crop->latest_plant_day));

  msg = unpack_int(msg, block);

  msg = unpack_string(msg, fir_str, STR_L_LEN);

  msg = unpack_int(msg, &(no_header));

  msg = unpack_int(msg, &(change_weather));

  msg = unpack_int(msg, &(loc_num));
  (*num_of_sims) = loc_num;

  msg = unpack_int(msg, sim_number);

  msg = start+MAX_STREAM_SIZE;

  msg = unpack_int(msg, &r1);
  RAN1_IFF = r1;
  msg = unpack_int(msg, &r1);
  RAN1_IX1 = r1;
  msg = unpack_int(msg, &r1);
  RAN1_IX2 = r1;
  msg = unpack_int(msg, &r1);
  RAN1_IX3 = r1;
  msg = unpack_int(msg, &r1);
  seed = r1;
  old_seed = seed;
#ifdef RANDOM
  printf("\nIFF %d %d %d %d seed=%d\n", RAN1_IFF, RAN1_IX1, RAN1_IX2,
                                                          RAN1_IX3, seed);
#endif
  for(j=0;j<98;j++){
     msg = unpack_float(msg, &tmp_fl);
     RAN1_R[j] = tmp_fl;
#ifdef RANDOM
     printf(" %.4f", RAN1_R[j]);
     if((j+1)%7==0) printf("\n");
#endif
     }
  return;
  }

/****************************************************************************
*                          ------ unpack_string ------                      *
*      This routine unpacks a string of a given length from the character *
*  array.                                                                 *
*      Belongs to:      Remote Slave Code                                 *
***************************************************************************/
char * unpack_string(char *msg, char *str, int strlen)
  {
  int i;

  for(i = 0; i < strlen; i++) {
    (*str) = (*msg);
    msg++;
    str++;
    }
  return msg;
  }

/**************************************************************************
*                          ------ unpack_int ------                       *
*      This routine unpacks an integer from the character array.          *
*      Belongs to:      Remote Slave Code                                 *
***************************************************************************/
char * unpack_int(char *msg, int *num)
  {
  int tmp, i;
  char *ptr;

  ptr = (char *)&tmp;
  for(i = 0; i < sizeof(int); i++) {
    (*ptr) = (*msg);
    ptr++; msg++;
    }
  (*num) = (int)tmp;
  return msg;
  }

/**************************************************************************
*                         ------ unpack_float ------                      *
*      This routine unpacks a floating point value from the character     *
*  array.                                                                 *
*      Belongs to:      Remote Slave Code                                 *
***************************************************************************/
char * unpack_float(char *msg, float *num)
  {
  int i;
  float tmp;
  char *ptr;
  ptr = (char *)&tmp;
  for(i = 0; i < sizeof(float); i++) {
    (*ptr) = (*msg);
    ptr++; msg++;
    }
  (*num) = (float)tmp;
  return msg;
  }

/***************************************************************************
*                          ------ unpack_long ------                       *
*      This routine unpacks a long integer from the character array.      *
*      Belongs to:      Remote Slave Code                                  *
***************************************************************************/
char * unpack_long(char *msg, long *num)
  {
  int i;
  long tmp;
  char *ptr;

  ptr = (char *)&tmp;
  for(i = 0; i < sizeof(long); i++) {
    (*ptr) = (*msg);
    ptr++; msg++;
    }
  (*num) = (long)tmp;
  return msg;
  }
#endif

/***************************************************************************
*                          ------ pack_int ------                          *
*      This routine packs an integer value into the character array.      *
*      Belongs to:      Both Client and Remote Slave Code                  *
***************************************************************************/
char * pack_int(char *msg, int num)
  {
  int i, tmp;
  char *ptr;

  tmp = num;
  ptr  = (char *)&tmp;
  for(i = 0; i < sizeof(int); i++) {
    (*msg) = (*ptr);
    msg++;
    ptr++;
    }
  return msg;
  }

/**************************************************************************
*                        ------ pack_float ------                         *
*      This routine packs a floating point value into the character array.*
*      Belongs to:      Both Client and Remote Slave Code                 *
**************************************************************************/
char * pack_float(char *msg, float num)
  {
  int i;
  float tmp;
  char *ptr;

  tmp = num;
  ptr  = (char *)&tmp;
  for(i = 0; i < sizeof(float); i++) {
    (*msg) = (*ptr);
    msg++;
    ptr++;
    }
  return msg;
  }

/**************************************************************************
*                          ------ simulate ------                         *
*      This is the routine used for carrying out the computation.  It     *
*  calls several other routines to complete the work.  The results obtai- *
*  ned is written into the otput file.                                    *
*      Belongs to:      Both Client and Remote Slave Code                 *
**************************************************************************/
int simulate(SIM_INFO   *sim,      SYS_INFO   *sys,   INFIL_INFO  *infil,
             ET_INFO    *et,       IRRI_INFO  *irrig, OUT_SEL     *output,
             long         offset,   CROP_INFO  *crop,  int          block,
             char        *fir_str,  FILE       *in,    FILE        *out,
            int         sim_number)
  {
  Call the necessary routines to carry out the computation and write the
  result to the output file.
  }
```

APPENDIX C

MAKEFILE FOR CLIENT

```
.KEEP_STATE:

CCC = /usr/DISK/CPP/SC1.0/CC
all: cmls92b cmls92m cmls92rem

cmls92b: cmls_main.o cmls_sbf.o cmls_lib.o cmls_w.o fungen.o
      $(CCC)  -O -o cmls92b -Bstatic cmls_main.o cmls_sbf.o \
      cmls_lib.o cmls_w.o fungen.o

cmls_main.o: h_client.h cmlsbat.h cmls_main.c
      $(CCC) -target sun4 -c -o cmls_main.o -g cmls_main.c

cmls_sbf.o: cmlsbat.h cmls_sbf.c
      $(CCC) -target sun4 -c -g cmls_sbf.c

cmls_lib.o: cmlsbat.h cmls_lib.c
      $(CCC) -target sun4 -c -g cmls_lib.c

cmls_w.o: cmlsbat.h cmls_w.c
      $(CCC) -target sun4 -c -g cmls_w.c

fungen.o: cmlsbat.h fungen.c
      $(CCC) -target sun4  -c -g fungen.c
#
# The client code
#
cmls92m: cmls_loc_main.o cmls_sbf.o cmls_lib.o cmls_w.o fungen.o \
      cmls_loc.o
      $(CCC)  -O -o cmls92m -Bstatic cmls_loc_main.o cmls_sbf.o \
      cmls_lib.o cmls_w.o fungen.o cmls_loc.o

cmls_loc_main.o: h_client.h cmlsbat.h cmls_main.c
      $(CCC) -target sun4 -c -o cmls_loc_main.o -DDISTRIB_OTHERS \
      cmls_main.c

cmls_loc.o: h_client.h cmlsbat.h cmls_dist.c
      $(CCC) -target sun4 -c -o cmls_loc.o -DDISTRIB_OTHERS cmls_dist.c
#
# The remote (slave) code
#
cmls92rem: cmls_rem_main.o cmls_sbf.o cmls_lib.o cmls_w.o fungen.o \
      cmls_rem.o
      $(CCC)  -O -o  cmls92rem -Bstatic -DREMOTE cmls_rem_main.o \
      cmls_sbf.o cmls_lib.o cmls_w.o fungen.o cmls_rem.o -lm

cmls_rem_main.o: cmlsbat.h cmls_main.c
```

```
      $(CCC) -target sun4 -c -o cmls_rem_main.o -g -DREMOTE cmls_main.c

cmls_rem.o: h_client.h cmlsbat.h cmls_dist.c
      $(CCC) -target sun4 -c -o cmls_rem.o -DREMOTE cmls_dist.c

clean:
      rm cmls92b cmls92rem cmls_rem_main.o cmls_sbf.o cmls_lib.o \
      cmls_w.o fungen.o cmls92m cmls_loc_main.o cmls_loc.o cmls_rem.o\
      cmls92b cmls_main.o
```

APPENDIX D

HEADER FILE FOR SERVER

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/stream.h>
#include <sys/param.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <errno.h>
#include <fcntl.h>
#include <math.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

#define TRUE  1
#define FALSE 0

#define  WAIT_QLEN        10
#define  PORT_NUM         12872

#define  BUFSIZE          512
#define  MAX_STREAM_SIZE  4096

#define  SERVER           'S'
#define  CLIENT           'C'
#define  FAULT_TOLERANCE  'F'

#define  HOST_NAME_LEN    MAXHOSTNAMELEN
#define  ADDR_LEN         MAXHOSTNAMELEN

#define  MAX_SERVERS      10

#define  SERVERS_DB  "SERVERS.DB"

int      Num_Of_Servers,  Num_Of_Curr_Servers;
char     myself[MAXHOSTNAMELEN];

#define  PERMS   0600   /* permissions for shared memory */
```

```
key_t    SHMKEY;        /* basevalue for shmem key */
key_t    SEMKEY;        /* basevalue for semaphore key */

int   shmid;            /* shared memory identifier */
char  *shmsg;           /* shared memory */
int   semid;            /* semaphore identifier */

int   fd_pipe[2];       /* pipe used for parent child communication */
char  filename[81];     /* temperory filename created - remote end */
int   fd_out;           /* fd of the output file - local end */

int   exited_child_pid = -1;  /* pid of the local second child */
float LOW_LOAD = 1.0;         /* Global load level */
int   client_sock;            /* socket of client process */

#define DATA_OVER        "DATA_OVER"
#define DATA_AVAILABLE   "DATA_AVAILABLE"

#define ALARM_INTERVAL    10      /* Interval for checking other
                                     servers are alive or not */
#define WAIT_INTERVAL     50      /* Sleep interval before checking
                                     another machines availability,
                                     if no child processes are there
                                     to exit */

struct servs {
   char   srv_name[HOST_NAME_LEN];
   char   srv_addr[ADDR_LEN];
                                  /* Application Specific */
   float load_mark;  /* Current load at 5 minute interval */
   int   Ran1_ix1;
   int   Ran1_ix2;
   int   Ran1_ix3;
   int   Ran1_iff;
   int   seed;
   float  Ran1_r[98];
   }servers[MAX_SERVERS];

struct sock_info{
   int    sock;
   char   srv_name[HOST_NAME_LEN];
   int    childpid;
   char   filename[HOST_NAME_LEN];
   }SOCK_INFO[MAX_SERVERS];

extern char*sys_errlist[];
```

```
              /* The following code is for debugging the CMLS data */
#ifdef DEBUG
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define  COMMENT                              0
#define  OUTPUT                               1
#define  CROP                                 2
#define  INFILTRATION                         3
#define  ET                                   4
#define  IRRIGATION                           5
#define  BEGINSIMULATION                      6
#define  ENDSIMULATION                        7
#define  NUMBEROFSIMULATIONS                  8
#define  PLANTINGDATE                         9
#define  SEED                                 10
#define  OUTPUTFILE                           11
#define  LENGTHUNITS                          12
#define  MODE                                 13

#define  TRAVELTIME                           0
#define  DEPTH                                1
#define  AMOUNT                               2

#define  CALENDAR                             0
#define  RELATIVE                             1

#define  NONE                                 0
#define  ACTUAL                               1
#define  PERIODIC                             2
#define  DEMAND                               3

#define  SOILNAME                             1
#define  CURVENOROOTDEPTH                     2
#define  SOILPROPERTY                         3
#define  CHEMICAL                             4
#define  CHEMICALPROPERTY                     5
```

```
#define  RESAMPLE                          6

#define  HISTORICAL                        0
#define  GENERATED                         2
#define  PAN                               3

#define  SCS_BLANEYCRIDDLE                 0
#define  FAO_BLANEYCRIDDLE                 1

#define  GEN_BANK                          1
#define  OUTPUT_BANK                       2
#define  IRRI_BANK                         3
#define  SYS_BANK                          4
#define  ESTI_BANK                         5
#define  INFIL_BANK                        6
#define  ET_BANK                           7
#define  DATE_BANK                         8

#define  START                             0
#define  PROPERTY                          1
#define  BETWEEN                           2

#define TRUE                      1
#define FALSE                     0
#define BLANK                     ""

#define MAXDAYS                   14615
#define MAXYEARS                  40
#define LMAX                      21
#define MAX_SOIL                  26
#define MAX_OUTFILE               15
#define MMPERINCH                 25.4
#define MIN_FLOAT                 1.0E-37

#define STR_L_LEN                 21
#define NAME_LEN                  65
#define SYMB_LEN                  11
#define ARRAY_LEN                 10
#define CHEM_NAME_LEN             40
#define LINE                      80
#define DAYS_PER_YEAR             367

typedef struct crop_info {
  char  name[STR_L_LEN];
  char  coef_file[NAME_LEN];
  float coef[DAYS_PER_YEAR];
  int   earliest_plant_day;
  int   latest_plant_day;
  } CROP_INFO;

typedef struct infiltration {
  int  source;
  char parm_file[NAME_LEN];
  char daily_weather_file[NAME_LEN];
  char daily_rain_file[NAME_LEN];
  int  estimator;
  } INFIL_INFO;

typedef struct evap {
  int   source;
  char  parm_file[NAME_LEN];
  char  daily_weather_file[NAME_LEN];
  char  daily_et_file[NAME_LEN];
  float par1;
  float par2;
  char  pan_file[NAME_LEN];
  int   estimator;
  } ET_INFO;

typedef struct simulation {
  char input_file[NAME_LEN];
  char output_file[NAME_LEN];
  int  begin_day;
  int  begin_year;
  int  end_day;
  int  end_year;
  int  num_sims;
  int  soil_chem_combinations;
  } SIM_INFO;

typedef struct total_water {
  float  et;
  float  pet;
  float  rain;
  float  infil;
  float  irrig;
  float  runoff;
  } H2O;

typedef struct day_depth_amount {
  int   day;
  float depth;
  float amount;
  } DAY_DEPTH_AMT;

typedef struct out_selections {
  DAY_DEPTH_AMT at_depth[ARRAY_LEN];
  int           no_depth;
  DAY_DEPTH_AMT at_time[ARRAY_LEN];
  int           no_time;
  int           amount;
  int           infilET;
  char          depth_units;
  H2O           water;
  } OUT_SEL;

typedef struct periodic_info {
  int   begin_day;
  int   end_day;
  int   period;
  float amount;
  } PERIODIC_INFO;

typedef struct demand_info {
  float critical_depletion;
  float efficiency;
  float amount;
  int   begin_day;
  int   end_day;
  } DEMAND_INFO;

typedef union periodic_demand {
  PERIODIC_INFO   peri;
  DEMAND_INFO     demd;
  char            file[NAME_LEN];
  } PERIODIC_DEMAND;

typedef struct irri_info {
  int             type;
  PERIODIC_DEMAND datum;
  char            amount_units;
  } IRRI_INFO;

typedef struct soil_prop {
  float depth;
  float oc;
  float bd;
  float fc;
  float pwp;
  float sat;
  float koc;
  float half_life;
  float kd;
  } SOIL_PROP;

typedef struct soil_info  {
  char      name[NAME_LEN];
  float     curve_no;
  float     root_depth;
  SOIL_PROP prop[LMAX+3];
  int       no_horizons;
  } SOIL_INFO;

typedef struct appl_info {
  int   earliest_day;
  int   latest_day;
  int   day;
  int   year;
  int   date_type;
  float depth;
  float amount;
  } APP_INFO;

typedef struct sys_info {
  char      index[STR_L_LEN];
  SOIL_INFO soil;
  char      chem_name[CHEM_NAME_LEN];
  APP_INFO  appl;
  char      depth_units;
  int       resampling;
  } SYS_INFO;

typedef struct weather {
  int   num_days;
  int   first_year;
  char  station_id[NAME_LEN];
  char  station_name[NAME_LEN];
  float latitude;
  char  hemisphere;
  float longitude;
  char  lon_side;
  float elevation;
  char  elev_units[STR_L_LEN];
  char  temp_units;
  char  infil_units;
  } WTH_INFO;

typedef struct check_info {
  int  need_check;
  char file[NAME_LEN];
  } CHECK_INFO;

typedef struct prop_infor {
float  oc;
float  bd;
float  fc;
float  pwp;
float  sat;
} PROP_INFOR;

typedef struct file_info {
char file[NAME_LEN];
int  no_header;
}  FILE_INFO;

typedef struct resample_soil {
SOIL_PROP property[LMAX];
int       horizons;
} RESAMPLE_SOIL;

#endif
```

APPENDIX E

SERVER CODE

```
#include "h_serv.h"

int         check_used_servers();
void        daemon_start();
static int  DOWN();
void        fault_tolerance();
int         find_a_remote_server();
void        find_new_low_mark();
void        fir_child_exit();
float       get_load();
void        get_shared_mem();
void        get_random_number_set();
void        initialize_server();
int         local_load_ok();
void        process_request_with_all_servers();
void        read_output();
void        read_server_names();
void        read_stream();
void        resend_data();
void        run_for_ever();
void        sec_child_exit_loc();
void        sec_child_exit_rem();
static int  semcall();
static int  seminit();
static void semkill();
void        serv_external();
void        serv_internal();
void        serv_request();
void        set_new_random_numbers();
void        set_reader_of_output();
static void UP();
void        write_stream();
void        write_temp_file();

char *      pack_float();
char *      pack_int();
char *      unpack_float();
char *      unpack_int();

#ifdef DEBUG
  void   call_unpacker();
  char * unpack_string();
  char * unpack_long();
  void   unpack_msg();
#endif
```

```
#ifdef FAULT_RECOVERY
int  fault_servers = 0;
char fault_files[MAX_SERVERS][HOST_NAME_LEN];
#endif

int send_seed;
/***************************************************************************
*                            ------ main ------                            *
*     This routine calls the initialization routine and the goes into the *
*  infinite wait routine.                                                  *
*     Executed by:  Local & Remote Server                                  *
***************************************************************************/
main(argc, argv)
int argc;
char *argv[];
   {
   int sd;

/* daemon_start();  */
     /*  If the above line is uncommented, the server will run as a daemon.
         Then the server should be started with the command line "server"
         without an &.  */

   if(argc > 1)
      LOW_LOAD = atof(argv[1]);

   printf("\nLOW_LOAD = %.2f\n", LOW_LOAD);

   read_server_names();

   initialize_server(&sd);

   signal(SIGCHLD, fir_child_exit);

   run_for_ever(sd);
   }

/***************************************************************************
*               ++++----++++ GENERAL ROUTINES ++++----++++                 *
*                                                                          *
*                          1. daemon_start()                               *
*                          2. initialize_server()                          *
*                          3. pack_float()                                 *
*                          4. read_server_names()                          *
*                          5. read_stream()                                *
*                          6. run_for_ever()                               *
*                         7. serv_request()                              *
*                         8. write_stream()                              *
*************************************************************************/
/*************************************************************************
*                       ------ daemon_start ------                       *
*     This routine makes the server a daemon.                            *
*     Executed by:  Local & Remote Server                                *
*************************************************************************/
void daemon_start()
   {
   int  pid, fd;
                     /* if not started by init process then detach */
   if(getppid() != 1){
      if((pid = fork()) < 0){
         perror("Forking first child");
         exit(1);
         }
                     /* Parent exits */
   if(pid > 0)
      exit(0);

   if(setsid() == -1) {
      perror("Changing process group");
      exit(1);
      }
   }
                     /* Close any open files */
   for(fd = 0; fd < NOFILE; fd++)
      close(fd);
   errno = 0;
   umask(0);
   }

/*************************************************************************
*                      ------ initialize_server ------                   *
*     The server makes a socket call and binds its address.              *
*     Executed by:  Local & Remote Server                                *
*************************************************************************/
void initialize_server(sd)
int *sd;
   {
   int  i, j, sock;
   int  addrlen;
   char *addr;
   struct sockaddr_in sock_serv;
   struct hostent *hp;
for(i = 0; i < MAX_SERVERS; i++) {
   SOCK_INFO[i].sock      = -1;
   strcpy(SOCK_INFO[i].srv_name, "");
   SOCK_INFO[i].childpid  = -1;
   strcpy(SOCK_INFO[i].filename, "");
   servers[i].load_mark = LOW_LOAD;
   servers[i].Ran1_iff = 0;
   servers[i].Ran1_ix1 = 0;
   servers[i].Ran1_ix2 = 0;
   servers[i].Ran1_ix3 = 0;
   servers[i].seed = 0;
   for(j=0; j<98;j++) servers[i].Ran1_r[j] = 0.0;
   }

Num_Of_Curr_Servers       = 0;

sock_serv.sin_family      = AF_INET;
sock_serv.sin_addr.s_addr = INADDR_ANY;
sock_serv.sin_port        = PORT_NUM;

if((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
   perror("Server: Error in creating socket");
   exit(1);
   }

if(bind(sock, &sock_serv, sizeof(sock_serv)) < 0) {
   perror("Server: Error in binding the socket");
   exit(1);
   }

addrlen = sizeof(sock_serv);
if(getsockname(sock, &sock_serv, &addrlen) < 0) {
   perror("Server: Error in getting port number");
   exit(1);
   }

gethostname(myself, 32);
hp = gethostbyname(myself);
addr  =  inet_ntoa(hp->h_addr);
strcpy(myself, addr);

listen(sock, WAIT_QLEN);
*sd = sock;
}
```

```
/***************************************************************************
*                          ------ pack_float ------                        *
*     This routine is used to pack a floating point value in a string.    *
*     Executed by:  Local & Remote Server                                  *
***************************************************************************/
char * pack_float(msg, num)
char *msg; float num;
   {
   int i;
   float tmp;
   char *ptr;

   tmp = num;
   ptr  = (char *)&tmp;
   for(i = 0; i < sizeof(float); i++) {
      (*msg) = (*ptr);
      msg++;
      ptr++;
      }
   return msg;
   }

/***************************************************************************
*                     ------ read_server_names ------                      *
*     During initialization, server reads the name of all other servers.  *
*     Executed by:  Local & Remote Server                                  *
***************************************************************************/
void read_server_names()
   {
   int i = 0;
   char tmp_buf[81];
   FILE *fp;

   if((fp = fopen(SERVERS_DB, "rt")) == NULL) {
      perror("Opening the servers database");
      exit(1);
      }

   while(fgets(tmp_buf, 80, fp) != NULL) {
      if(i > MAX_SERVERS) {
         perror("Not enough size for reading in servers database");
         exit(1);
         }

      if((strlen(tmp_buf) > 10) && tmp_buf[0] != '#') {
         sscanf(tmp_buf,"%s %s", servers[i].srv_name, servers[i].srv_addr);
         i++;
         }
      }

   Num_Of_Servers  = i;
   printf("\n No of records read = %d\n",i);
   }

/***************************************************************************
*                          ------ read_stream ------                       *
*     Used to read from a stream file descriptor (socket, file etc).      *
*     Executed by:  Local & Remote Server                                  *
***************************************************************************/
void read_stream(stream, buffer, numbytes)
int  stream, numbytes;
char *buffer;
   {
   int nread = 0;
   char *str;

   memset(buffer, (char)0, numbytes);
   str = buffer;
   while((nread += read(stream, str, (numbytes - nread))) < numbytes)
      str = buffer + nread;
   }

/***************************************************************************
*                          ------ run_for_ever ------                      *
*     The concurrent server waits for ever for a connection request.      *
*     Executed by:  Local & Remote Server                                  *
***************************************************************************/
void  run_for_ever(sock)
int sock;
   {
   int  pid;
   int  status;
   int  new_sock;

   while(TRUE) {
      while((new_sock = accept(sock, 0, 0)) < 0) {
         if(errno == EINTR)
            errno = 0;
         else {
            perror("Error accepting a new connection: SERVER");
            exit(1);
            }
         }

      pid = fork();
      switch(pid) {
         case -1:
            perror("Error forking process");
            exit(1);

         case  0:
            close (sock);
            serv_request(new_sock);
            exit(0);

         default:
            close (new_sock);
            break;
         }
      }
   }

/***************************************************************************
*                        ------ serv_request ------                        *
*     The child server processes the service request.                      *
*     Executed by:  Local & Remote Server                                  *
***************************************************************************/
void serv_request(new_sock)
int new_sock;
   {
   char type, buff[BUFSIZE], *str;

   read_stream(new_sock, buff, BUFSIZE);
   sscanf(buff, "%c", &type);

   if(type == SERVER) {
      sscanf(buff, "%c %f", &type, &LOW_LOAD);
      printf("\nLOW_LOAD = %.2f\n", LOW_LOAD);
      serv_external(new_sock);
      }
   else
      if(type == CLIENT)
         serv_internal(new_sock);
      else
         if(type == FAULT_TOLERANCE)
            printf("Fault Tolerance connection request\n");
         else {
            printf("\nUnrecognized connection request\n");
         close(new_sock);
         exit(1);
         }

   close(new_sock);
   }

/**************************************************************************
*                       ------ write_stream ------                        *
*     Used to write to a stream file descriptor (socket, file etc).       *
*     Executed by:  Local & Remote Server                                 *
**************************************************************************/
void write_stream(stream, buffer, numbytes)
int  stream, numbytes;
char *buffer;
   {
   int nwrite = 0;
   char *str;

   str   = buffer;
   while((nwrite +=  write(stream, str, (numbytes - nwrite))) < numbytes)
      str = buffer + nwrite;
   }

/**************************************************************************
*         ++++----++++  CLIENT END (LOCAL) ROUTINES  ++++----++++         *
*                                                                         *
*                    1. check_used_servers()                              *
*                    2. DOWN()                                            *
*                    3. fault_tolerance()                                 *
*                    4. find_a_remote_server()                            *
*                    5. find_new_low_mark()                               *
*                    6. get_random_number_set()                           *
*                    7. pack_int()                                        *
*                    8. process_request_with_all_servers()                *
*                    9. read_output()                                     *
*                   10. sec_child_exit_loc()                              *
*                   11. semcall()                                         *
*                   12. seminit()                                         *
*                   13. semkill()                                         *
*                   14. serv_internal()                                   *
*                   15. set_new_random_numbers()                          *
*                   16. set_reader_of_output()                            *
*                   17. UP()                                              *
**************************************************************************/
/**************************************************************************
```

```
*                       ------ check_used_servers ------                      *
*     Find if the server is currently being used.                             *
*     Executed by:  Local Server                                              *
******************************************************************************/
int check_used_servers(indx)
int indx;
   {
   int i;

   for(i = 0; (i < Num_Of_Curr_Servers) &&
      strcmp(SOCK_INFO[i].srv_name, servers[indx].srv_name) != 0; i++);
   if(i >= Num_Of_Curr_Servers)
      return FALSE;

   return TRUE;
   }

/*****************************************************************************
*                       ------ DOWN ------                                    *
*     Acquire semaphore by a Down operation.                                  *
*     Executed by:  Local Server                                              *
******************************************************************************/
static int DOWN(sid)
int sid;
   {
   return (semcall(sid, -1));
   }

/*****************************************************************************
*                         ------ fault_tolerance ------                       *
*     Check whether current remote servers are alive. If one of them is       *
*  not alive, either terminate(default) or invoke fault_recovery(if keyw-     *
*  ord FAULT_RECOVERY defined at compilation).                                *
*     Executed by:  Local Server                                              *
******************************************************************************/
void fault_tolerance()
   {
   int i, j;
   int tmp_sock;
   char buff[BUFSIZE];
   struct sockaddr_in sock_serv;
   struct hostent *hp;

   memset(buff, (char)0, sizeof(buff));
   sprintf(buff, "%c", FAULT_TOLERANCE);
```

```
    sock_serv.sin_family  =  AF_INET;
    sock_serv.sin_port    =  PORT_NUM;

    for(i = 0; i < Num_Of_Curr_Servers; i++)
       printf("\nCurrent Server - %s", SOCK_INFO[i].srv_name);

    for(i = 0; i < Num_Of_Curr_Servers; i++) {
       if((tmp_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
          perror("Server: Error in creating socket - 3");
          exit(1);
          }

        hp = gethostbyname(SOCK_INFO[i].srv_name);

        memcpy(&sock_serv.sin_addr, hp->h_addr, hp->h_length);
        if(connect(tmp_sock, &sock_serv, sizeof(sock_serv)) < 0) {
           printf("\nREMOTE SERVER DEAD %s\n", SOCK_INFO[i].srv_name);
           kill(SOCK_INFO[i].childpid, SIGKILL);
#ifdef FAULT_RECOVERY
           strcpy(fault_files[fault_servers], SOCK_INFO[i].filename);
           fault_servers++;

               /* move all records 1 place left */
           for(j = i; j < (Num_Of_Curr_Servers - 1); j++) {
              strcpy(SOCK_INFO[j].srv_name, SOCK_INFO[j+1].srv_name);
              SOCK_INFO[j].sock      = SOCK_INFO[j+1].sock;
              SOCK_INFO[j].childpid  = SOCK_INFO[j+1].childpid;
              strcpy(SOCK_INFO[j].filename, SOCK_INFO[j+1].filename);
              }
           strcpy(SOCK_INFO[j].srv_name, "");
           SOCK_INFO[j].sock      = -1;
           SOCK_INFO[j].childpid  = -1;
           strcpy(SOCK_INFO[j].filename, "");

           Num_Of_Curr_Servers--;
           i--;
#else
           exit(1);
#endif
           }
        else
           write_stream(tmp_sock, buff, BUFSIZE);

        close(tmp_sock);
       }
```

```
   alarm(ALARM_INTERVAL);
   }

/***************************************************************************
*                   ------ find_a_remote_server ------                     *
*     Locate a server that is willing to accept the job.                   *
*     Executed by:  Local Server                                           *
***************************************************************************/
int find_a_remote_server(sock, rem_serv)
int *sock;
char *rem_serv;
   {
   int i;
   int tmp_sock;
   int success = FALSE;
   char *str, buff[BUFSIZE];
   struct sockaddr_in sock_serv;
   struct hostent *hp;

   sock_serv.sin_family  =  AF_INET;
   sock_serv.sin_port    =  PORT_NUM;

   for(i = 0; (i < Num_Of_Servers) && !success; i++) {
      if(!check_used_servers(i)) {
         printf("\nFind: Checking %s Num_Curr=%d Num_SER= %d\n",
             servers[i].srv_name, Num_Of_Curr_Servers, Num_Of_Servers);

         if((tmp_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
            perror("Server: Error in creating socket - 2");
            exit(1);
            }
         hp = gethostbyname(servers[i].srv_name);
         memcpy(&sock_serv.sin_addr, hp->h_addr, hp->h_length);
         if(connect(tmp_sock, &sock_serv, sizeof(sock_serv)) < 0) {
            printf("SKIPPING %s\n", servers[i].srv_name);
            close(tmp_sock);
            }
         else {
            find_new_low_mark();
            memset(buff, (char)0, sizeof(buff));
            sprintf(buff, "%c %f", SERVER, LOW_LOAD);

            write_stream(tmp_sock, buff, BUFSIZE);
                /* Wait for reply */
            read_stream(tmp_sock, buff, BUFSIZE);
                /**  Success decided by acceptance of other machine  **/
            printf("\nReceived from other server: %s\n", buff);
            sscanf(buff, "%d", &success);
            if(success){
               strcpy(rem_serv, servers[i].srv_name);
               (*sock) = tmp_sock;
               }
            else {
               sscanf(buff, "%d %f", &success, &(servers[i].load_mark));
               close(tmp_sock);
               }
            }
         }
      }

   if(success)
      return TRUE;
   else
      return FALSE;
   }

/***************************************************************************
*                     ------ find_new_low_mark ------                       *
*     Find the new global load.                                             *
*     Executed by:  Local Server                                            *
***************************************************************************/
void find_new_low_mark()
   {
   int i;
   float tmp_load = 0.0;

   for(i=0; i< Num_Of_Servers; i++)
      tmp_load += servers[i].load_mark;

   LOW_LOAD = tmp_load/Num_Of_Servers + 1.0;
   }

/***************************************************************************
*                    ------ get_random_number_set ------                    *
*     Get the random number set associated with a remote machine.           *
*     Executed by:  Local Server                                            *
***************************************************************************/
void get_random_number_set(buff, rem_serv)
char *buff, *rem_serv;
   {
   int i, j;
   int r1;
   char *msg;

   for(i = 0; i < MAX_SERVERS &&
                        strcmp(servers[i].srv_name, rem_serv)!=0; i++);

   if(!(servers[i].Ran1_iff)) {
      r1 = send_seed;
      send_seed *= (float)rand()/(float)rand();
      }
   else
      r1 = servers[i].seed;

#ifdef RANDOM
   printf("\nIFF=%d %d %d %d SD=%d\n",servers[i].Ran1_iff,
             servers[i].Ran1_ix1, servers[i].Ran1_ix2, servers[i].Ran1_ix3,
             servers[i].seed);
#endif

   memset(buff, (char)0, BUFSIZE);
   msg = buff;
   msg = pack_int(msg, servers[i].Ran1_iff);
   msg = pack_int(msg, servers[i].Ran1_ix1);
   msg = pack_int(msg, servers[i].Ran1_ix2);
   msg = pack_int(msg, servers[i].Ran1_ix3);
   msg = pack_int(msg, r1);
   for(j=0;j<98;j++){
      msg = pack_float(msg,servers[i].Ran1_r[j]);

#ifdef RANDOM
      printf(" %.4f",servers[i].Ran1_r[j]);
      if((j+1)%7==0) printf("\n");
#endif
      }
   }

/***************************************************************************
*                            ------ pack_int ------                        *
*     Pack an integer into a character string.                             *
*     Executed by:  Local Server                                           *
***************************************************************************/
char * pack_int(msg, num)
char *msg; int num;
   {
   int i;
   int tmp;
   char *ptr;
   tmp = num;
   ptr  = (char *)&tmp;
   for(i = 0; i < sizeof(int); i++) {
      (*msg) = (*ptr);
      msg++;
      ptr++;
      }
   return msg;
   }

/*************************************************************************
*        ------ process_request_with_all_servers ------                  *
*     This routine is the main routine that processes a local request.   *
*     Executed by:  Local Server                                         *
*************************************************************************/
void process_request_with_all_servers(new_sock, sock, remote, filename)
int  new_sock, sock;
char *remote;
   {
   int   i = 0, j, status;
   int   data_available;
   int   mask;
   int   open_flag;
   char  executable[81];
   char  outfilename[81];
   char  fopen_mode[5];
   char  buff1[BUFSIZE], buff2[BUFSIZE];
   char  msg[MAX_STREAM_SIZE], *str;
   char  rem_serv[ADDR_LEN];

   strcpy(rem_serv, remote);

   SEMKEY = (key_t)getpid();
   if((semid = seminit(SEMKEY, 1)) < 0){
      printf("\nError in creating the semaphore with Key = %d\n", SEMKEY);
      exit(1);
      }

   if(pipe(fd_pipe) < 0){
      printf("\nError in creating the pipe with \n");
      exit(1);
      }
            /* read executable file name */
   read_stream(new_sock, buff1, BUFSIZE);
   sscanf(buff1,"%s %d %s %s",executable, &send_seed, outfilename,
                                                            fopen_mode);
   if(strcmp(fopen_mode, "wt") == 0)
      open_flag = (O_WRONLY | O_CREAT | O_TRUNC);
   else
      open_flag = (O_WRONLY | O_CREAT | O_APPEND);


   if((fd_out = open(outfilename, open_flag, PERMS)) == -1){
      printf("\nSERVER: Error opening the temp output file - %s\n\n",
                                                              filename);
      exit(1);
      }

   printf("\nSERVER: EXEC = %s SEND_SEED= %d OUTFILE = %s\n", executable,
                                                 send_seed, outfilename);
            /* read if further data available */
   read_stream(new_sock, msg, BUFSIZE);

   if(strcmp(msg, DATA_AVAILABLE) == 0){
      data_available = TRUE;

      read_stream(new_sock, msg, MAX_STREAM_SIZE);

         /* write executable file to remote machine */
      write_stream(sock, buff1, BUFSIZE);
         /* write data */
      write_stream(sock, msg, MAX_STREAM_SIZE);

#ifdef FAULT_RECOVERY
      write_temp_file(msg, rem_serv);
#endif

      get_random_number_set(buff2, rem_serv);
         /* write random_numbers */
      write_stream(sock, buff2, BUFSIZE);

      set_reader_of_output(sock, rem_serv);
      }
   else
      data_available = FALSE;

   close(sock);

   alarm(ALARM_INTERVAL);

               /* Rest of the data is processed below */
   while(data_available){
           /* read if further data available */
      read_stream(new_sock, msg, BUFSIZE);

      if(strcmp(msg, DATA_AVAILABLE) == 0){

         data_available = TRUE;
         read_stream(new_sock, msg, MAX_STREAM_SIZE);

          /* If no servers available, wait for one to become free */

wait_c:  while(!find_a_remote_server(&sock, rem_serv)) {
            printf("\nWaiting for a server to become available\n");
            if((exited_child_pid = wait(&status)) < 0)
              if(errno == EINTR) {
                 printf("\nINTERRUPTED:%s\n", sys_errlist[errno]);
                 errno = 0;
                 }
              else{
                perror("\nError while waiting for child to exit.");
                sleep(WAIT_INTERVAL);
                }
            printf("\nWait returned\n");
            }

#ifdef FAULT_RECOVERY
         if(fault_servers){
           mask = sigmask(SIGALRM);
           mask = sigblock(mask);
           resend_data(sock, buff1, rem_serv);
           sigsetmask(mask);
           goto wait_c;
           }
#endif
            /* write executable file to remote machine */
         write_stream(sock, buff1, BUFSIZE);

         write_stream(sock, msg, MAX_STREAM_SIZE);

#ifdef FAULT_RECOVERY
         write_temp_file(msg, rem_serv);
#endif

         get_random_number_set(buff2, rem_serv);
            /* write random_numbers */
         write_stream(sock, buff2, BUFSIZE);
         set_reader_of_output(sock, rem_serv);

         close(sock);
         }
      else
         data_available = FALSE;
      }

   j = Num_Of_Curr_Servers;
   for(i = 0; i < j; i++){
   printf("\nIn for loop\n");
      if((exited_child_pid = wait(&status)) < 0) {
         if(errno == EINTR) {
            printf("\nINTERRUPTED:%s\n", sys_errlist[errno]);
            errno = 0;
            }
         else{
            perror("\nError while waiting for child to exit.\n\n");
            sleep(WAIT_INTERVAL);
            }
         }

#ifdef FAULT_RECOVERY
      if(fault_servers){
        mask = sigmask(SIGALRM);
        mask = sigblock(mask);
        while(!find_a_remote_server(&sock, rem_serv))
           sleep(WAIT_INTERVAL);
        resend_data(sock, buff1, rem_serv);
        sigsetmask(mask);
        i--;
        }
#endif
      }

   close(fd_out);
   semkill(semid);

   sprintf(buff1,"%d COMPUTATION SUCCESSFULLY COMPLETED", TRUE);
                  /* signal the client of the completion of work */
   write_stream(new_sock, buff1, BUFSIZE);
   }

/****************************************************************************
*                      ------ read_output ------                            *
*     The child process, reads the output from a remote server.          *
*     Executed by:  Local Server                                         *
**************************************************************************/
void read_output(sock)
int sock;
   {
   int   send_bytes;
   int   first = TRUE;
   int   pid;
   char  buff1[MAX_STREAM_SIZE], buff2[BUFSIZE], *msg;

      /* put the pid value of the process so that the parent can read it */
   memset(buff1, (char)0, MAX_STREAM_SIZE);
   pid = getpid();
   msg = buff1;
   msg = pack_int(msg, pid);

        /* append the low_load & Random number data from the server */
   read_stream(sock, msg, BUFSIZE);

   memcpy(buff2, buff1, BUFSIZE);
        /* write pid +low_load &  random number data to the parent */
        /* Note that the low_load & random number data should be 4bytes
           lesser than BUFSIZE, since the pid is also added */
   DOWN(semid);

   while(TRUE) {

      read_stream(sock, buff1, BUFSIZE);

      sscanf(buff1, "%d", &send_bytes);
      if(first){
        if(send_bytes == 0){
          printf("\nCHILD READ OUTPUT: NO DATA SEND FROM REMOTE SERVER\n");
          msg = pack_int(buff2, -1);
          write_stream(fd_pipe[1], buff2, BUFSIZE);
          UP(semid);
          exit(1);
          }

        write_stream(fd_pipe[1], buff2, BUFSIZE);
        first = FALSE;
        }

      if(send_bytes == 0)
         break;
      read_stream(sock, buff1, MAX_STREAM_SIZE);

      write_stream(fd_out, buff1, send_bytes);
      }

   UP(semid);
   close(sock);
   }

#ifdef FAULT_RECOVERY
/***************************************************************************
*                        ------ resend_data ------                          *
*     If the remote machine fails, resend the data to another machine.     *
*     Executed by:  Local Server                                           *
***************************************************************************/
void resend_data(sock, buff1, rem_serv)
int  sock;
char *buff1, *rem_serv;
   {
   int   i, fd;
   char  buff2[MAX_STREAM_SIZE];

   printf("\nRESENDING DATA File = %s\n", fault_files[0]);
   write_stream(sock, buff1, BUFSIZE);

   if((fd = open(fault_files[0], O_RDONLY)) == -1){
      printf("\nError opening the data file %s\n\n", fault_files[0]);
      exit(1);
      }

   read_stream(fd, buff2, MAX_STREAM_SIZE);

   write_stream(sock, buff2, MAX_STREAM_SIZE);

   write_temp_file(buff2, rem_serv);

   get_random_number_set(buff2, rem_serv);
            /* write random_numbers */
   write_stream(sock, buff2, BUFSIZE);

   set_reader_of_output(sock, rem_serv);

   close(sock);
   close(fd);
   for(i = 0; i < (fault_servers-1); i++)
```

```
     strcpy(fault_files[i], fault_files[i+1]);
   strcpy(fault_files[i], "");
   fault_servers--;

   printf("\nRESENDING DATA - OVER\n");
   }

/***************************************************************************
*                        ------ write_temp_file ------                    *
*     If FAULT_RECOVERY is defined, then write the data to a temp file.   *
*     Executed by:  Local Server                                          *
***************************************************************************/
void write_temp_file(msg, rem_serv)
char  *msg, *rem_serv;
   {
   int fd;

   if((fd = open(rem_serv, O_WRONLY | O_CREAT | O_TRUNC, PERMS)) == -1){
      printf("\nFAULT_TOLERANCE: Error opening the temp file - %s\n\n",
                                                            rem_serv);
      exit(1);
      }
   write_stream(fd, msg, MAX_STREAM_SIZE);
   close(fd);
   }
#endif



/***************************************************************************
*                      ------ sec_child_exit_loc ------                   *
*     Signal handler, when the second child reading output, exits.        *
*     Executed by:  Local Server                                          *
***************************************************************************/
void sec_child_exit_loc()
   {
   int     i;
   char    buff[BUFSIZE], *str;
   int     maxfd;
   fd_set  fdvar;
   static struct timeval timeout;

   maxfd  = fd_pipe[0] + 1;
   FD_ZERO(&fdvar);
   FD_SET(fd_pipe[0], &fdvar);
   timeout.tv_sec  = 0L;
   timeout.tv_usec = 0L;
   select(maxfd, &fdvar, (fd_set *) 0, (fd_set *) 0, &timeout);

   if(FD_ISSET(fd_pipe[0], &fdvar)) {
      read_stream(fd_pipe[0], buff, BUFSIZE);
      set_new_random_numbers(buff, &exited_child_pid);
      if(exited_child_pid < 0) {
        printf("\nSERVER ERROR: Child could not read any output from \
                                                  remote server!");
        printf("\nPossible reason: Executable file path not correct!\n");
        sprintf(buff,"%d SERVER ERROR: Possible reason: Executable\
                file location not correct!", FALSE);
        write_stream(client_sock, buff, BUFSIZE);
        shutdown(client_sock, 0);
        exit(1);
        }
      }
   else{
      printf("\nCould not read child pid\n");
      return;
      }
                   /* locate the child pid in the list */
   for(i = 0; (i < Num_Of_Curr_Servers) &&
                      SOCK_INFO[i].childpid != exited_child_pid; i++);

   printf("\nSERVER %s is FREE Exited_Child_Pid = %d \n",
                             SOCK_INFO[i].srv_name, exited_child_pid);

        /* move all records 1 place left */
   if(i < Num_Of_Curr_Servers){
#ifdef FAULT_RECOVERY
      signal(SIGCHLD, SIG_IGN);
      sprintf(buff,"rm %s", SOCK_INFO[i].filename);
      system(buff);
#endif
      for( ; i < (Num_Of_Curr_Servers - 1); i++) {
         strcpy(SOCK_INFO[i].srv_name, SOCK_INFO[i+1].srv_name);
         SOCK_INFO[i].sock      = SOCK_INFO[i+1].sock;
         SOCK_INFO[i].childpid  = SOCK_INFO[i+1].childpid;
         strcpy(SOCK_INFO[i].filename, SOCK_INFO[i+1].filename);
         }
      strcpy(SOCK_INFO[i].srv_name, "");
      SOCK_INFO[i].sock      = -1;
      SOCK_INFO[i].childpid  = -1;
      strcpy(SOCK_INFO[i].filename, "");
      Num_Of_Curr_Servers--;
      }
   signal(SIGCHLD, sec_child_exit_loc);
   }

/**************************************************************************
*                       ------ semcall ------                            *
*     Semaphore operation.                                               *
*     Executed by:  Local Server                                         *
**************************************************************************/
static int semcall(sid, op)
int sid;
int op;
   {
   struct sembuf sb;

   sb.sem_num = 0;
   sb.sem_op = op;
   sb.sem_flg = 0;
   return (semop(sid, &sb, 1));
   }

/**************************************************************************
*                       ------ seminit ------                            *
*     Semaphore initialization.                                          *
*     Executed by:  Local Server                                         *
**************************************************************************/
static int seminit (key, initval)
key_t key;
int initval;
   {
   int sid;
   union semun semun;

       /* Get the semaphore id based on the key */
   if((sid = semget (key, 1, PERMS | IPC_CREAT)) == -1)
      perror("semget");
   else {
       /* Got the id, now set its initial value */
      printf("Semaphore id (sid) = %d\n", sid);
      semun.val = initval;
      if(semctl(sid, 0, SETVAL, semun) == -1)
         perror("semctl");
      }
   return(sid);
   }
```

```
/*************************************************************************
*                        ------ semkill ------                           *
*     Remove the semaphore from the IPC table.                           *
*     Executed by:  Local Server                                         *
*************************************************************************/
static void semkill (sid)
int sid;
   {
   if(semctl(sid,0,IPC_RMID,0) == -1)
      perror("semctl (kill)");
   printf("Semaphore with value of sid = %d is killed \n",sid);
   }

/*************************************************************************
*                        ------ serv_internal ------                     *
*     Serve an internal request.                                         *
*     Executed by:  Local Server                                         *
*************************************************************************/
void serv_internal(new_sock)
int  new_sock;
   {
   int  i;
   int  success = FALSE;
   int  sock;
   char buff[BUFSIZE], *str;
   struct sockaddr_in sock_serv;
   struct hostent *hp;

   client_sock = new_sock;
   printf("\nServer: %s - Internal Connection.\n", myself);
   signal(SIGALRM, fault_tolerance);

   sock_serv.sin_family      = AF_INET;
   sock_serv.sin_port        = PORT_NUM;

   for(i = 0; (i < Num_Of_Servers) && !success; i++) {
         if((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
            perror("Server: Error in creating socket - 2");
            exit(1);
            }

         hp = gethostbyname(servers[i].srv_name);
         memcpy(&sock_serv.sin_addr, hp->h_addr, hp->h_length);
         if(connect(sock, &sock_serv, sizeof(sock_serv)) < 0) {
            printf("SKIPPING %s\n", servers[i].srv_name);
            close(sock);
            }
         else {
            find_new_low_mark();
            memset(buff, (char)0, sizeof(buff));
            sprintf(buff, "%c %f", SERVER, LOW_LOAD);

            write_stream(sock, buff, BUFSIZE);
                /* Wait for reply */
            read_stream(sock, buff, BUFSIZE);

                /*  Success decided by acceptance of other machine  */
            printf("\nReceived from other server: %s\n", buff);
            sscanf(buff, "%d", &success);
            if(!success){
               sscanf(buff, "%d %f", &success, &(servers[i].load_mark));
               close(sock);
               }
            }
      }

   memset(buff, (char)0, sizeof(buff));
   sprintf(buff, "%d", success);

   write_stream(new_sock, buff, BUFSIZE);

   if(success)
      process_request_with_all_servers(new_sock, sock,
                                              servers[i-1].srv_name);
   }

/***************************************************************************
*               ------ set_new_random_numbers ------                      *
*     This routine sets back the random numbers from a remote machine.    *
*     Executed by:  Local Server                                          *
***************************************************************************/
void set_new_random_numbers(buff, child_pid)
char *buff;
int  *child_pid;
   {
   int    i, j;
   int    tmp_int;
   float  tmp_fl;
   char   *msg;

   msg = buff;
   msg = unpack_int(msg, &tmp_int);
   (*child_pid) = tmp_int;
   if((*child_pid) < 0)
      return;                  /* Error at remote server */

           /* locate the server name that was servicing this request */
   for(i = 0; (i < Num_Of_Curr_Servers) &&
                             SOCK_INFO[i].childpid != (*child_pid); i++);

   if(i < Num_Of_Curr_Servers) {
      for(j = 0; (j < Num_Of_Servers) &&
          strcmp(servers[j].srv_name, SOCK_INFO[i].srv_name) != 0; j++);

      if(j < Num_Of_Servers) {
         msg = unpack_float(msg, &tmp_fl);
         servers[j].load_mark = tmp_fl;
         msg = unpack_int(msg, &tmp_int);
         servers[j].Ran1_iff = tmp_int;
         msg = unpack_int(msg, &tmp_int);
         servers[j].Ran1_ix1 = tmp_int;
         msg = unpack_int(msg, &tmp_int);
         servers[j].Ran1_ix2 = tmp_int;
         msg = unpack_int(msg, &tmp_int);
         servers[j].Ran1_ix3 = tmp_int;
         msg = unpack_int(msg, &tmp_int);
         servers[j].seed     = tmp_int;

#ifdef RANDOM
         printf("\nPID %d %.2f %d %d %d %d %d\n",(*child_pid),
            servers[j].load_mark, servers[j].Ran1_iff, servers[j].Ran1_ix1,
            servers[j].Ran1_ix2, servers[j].Ran1_ix3, servers[j].seed);
#endif
         for(i=0;i<98;i++){
            msg = unpack_float(msg, &tmp_fl);
            servers[j].Ran1_r[i] = tmp_fl;
#ifdef RANDOM
            printf(" %.4f", servers[j].Ran1_r[i]);
            if((i+1)%7==0) printf("\n");
#endif
            }
         }
      }
   }

/***************************************************************************
*                   ------ set_reader_of_output ------                     *
*     The parent server sets a child to read the output from a rem server.*
*     Executed by:  Local Server                                          *
***************************************************************************/
void set_reader_of_output(sock, rem_serv)
int sock;
char *rem_serv;
   {
   int  pid;

   signal(SIGCHLD, sec_child_exit_loc);

   pid = fork();
   switch(pid){
      case  -1:
         perror("Forking the second child for reading output\n");
         exit(1);

      case   0:
         read_output(sock);
         exit(0);

      default:
         break;
      }

      /* parent */
   strcpy(SOCK_INFO[Num_Of_Curr_Servers].srv_name, rem_serv);
   SOCK_INFO[Num_Of_Curr_Servers].sock      = sock;
   SOCK_INFO[Num_Of_Curr_Servers].childpid  = pid;
   strcpy(SOCK_INFO[Num_Of_Curr_Servers].filename, rem_serv);
   Num_Of_Curr_Servers++;
   }

/***************************************************************************
*                             ------ UP ------                            *
*     Release a semaphore by doing an UP operation.                       *
*     Executed by:  Local Server                                          *
***************************************************************************/
static void UP(sid)
int sid;
   {
   if(semcall(sid, 1)== -1)
      perror("Semop");
   }

/***************************************************************************
```

```
*         ++++----++++  SERVER END (REMOTE) ROUTINES  ++++----++++       *
*                                                                        *
*                       1. fir_child_exit()                              *
*                       2. get_load()                                    *
*                       3. get_shared_mem()                              *
*                       4. local_load_ok()                               *
*                       5. sec_child_exit_rem()                          *
*                       6. serv_external()                               *
*                       7. unpack_int()                                  *
*                       8. unpack_float()                                *
*************************************************************************/
/*************************************************************************
*                       ------ fir_child_exit ------                     *
*     Signal handler for the first child's exit.                         *
*     Executed by:  Remote Server                                        *
*************************************************************************/
void fir_child_exit()
   {
   int statusp, ret;
   struct rusage rusage;

   signal(SIGCHLD, fir_child_exit);

   if((ret = wait3(&statusp, WNOHANG, &rusage)) == 0)
      printf("\n\nNO PROCESS TO REPORT STATUS\n\n");
   if(ret == -1)
      printf("\nRETURN VALUE = -1\n\n");
   }

/*************************************************************************
*                        ------ get_load ------                          *
*     Find the current load.                                             *
*     Executed by:  Remote Server                                        *
*************************************************************************/
float get_load()
   {
   FILE  *fp;
   char  buff[BUFSIZE], str[15], *ptr;
   float load_1, load_5, load_15;

   if((fp = popen("uptime", "r")) == NULL) {
      printf("\nError opening with popen!\n");
      exit(1);
      }

   fgets(buff, BUFSIZE, fp);
   pclose(fp);
          /* locate the "load average:" string in buff" */
   if((ptr = strstr(buff, "average:")) == NULL){
      printf("\nError in strstr()\n");
      exit(1);
      }
   sscanf(ptr, "%s %f, %f, %f", str, &load_1, &load_5, &load_15);

   return load_5;
   }

/***************************************************************************
*                      ------ get_shared_mem ------                        *
*     Create the shared memory segment and attach it.                      *
*     Executed by:  Remote Server                                          *
***************************************************************************/
void get_shared_mem()
   {
   SHMKEY  = (key_t)getpid();

   if((shmid = shmget(SHMKEY, MAX_STREAM_SIZE+BUFSIZE, PERMS|IPC_CREAT))
                                                                    < 0){
      fprintf(stderr, "\nError getting shared memory\n\n");
      exit(1);
      }

   if((shmsg = (char *)shmat(shmid, (char *) 0, 0)) == (char *) -1){
      fprintf(stderr, "\nServer cannot attach shared memory\n\n");
      exit(1);
      }
   }

/***************************************************************************
*                      ------ local_load_ok ------                         *
*     Check the local load.                                                *
*     Executed by:  Remote Server                                          *
***************************************************************************/
int local_load_ok()
   {
   if(get_load() < LOW_LOAD)
      return TRUE;
   else
      return FALSE;
   }
```

```
/****************************************************************************
*                   ------ sec_child_exit_rem ------                        *
*     Signal handler for second childs exit.                                *
*     Executed by:  Remote Server                                           *
****************************************************************************/
void sec_child_exit_rem(sock)
int sock;
   {
   int   fd;
   int   nread;
   float tmp_load;
   char  buff[BUFSIZE], *str;

           /* send the low_load & Random number DATA to the other server */
   tmp_load = get_load();

   pack_float(shmsg+MAX_STREAM_SIZE, tmp_load);
             /* space is reserved for load in the string */

   write_stream(sock, shmsg+MAX_STREAM_SIZE, BUFSIZE);

   if((fd = open(filename, O_RDONLY)) == -1){
      printf("\nSERVER: Opening the temporary (Rem)file- %s\n\n",filename);
      memset(buff, (char)0, BUFSIZE);
      sprintf(buff, "0");
      write_stream(sock, buff, BUFSIZE);
      exit(1);
      }

   while(TRUE) {
      memset(shmsg, (char)0, MAX_STREAM_SIZE);
      str = shmsg;
      nread = read(fd, str, MAX_STREAM_SIZE);
/*    printf("\nNREAD FROM FILE = %d",nread); */
      if(nread < 0) {
         printf("\nServer: Error reading from the output file\n");
         memset(buff, (char)0, BUFSIZE);
         sprintf(buff, "0");
         write_stream(sock, buff, BUFSIZE);
         exit(1);
         }

      memset(buff, (char)0, BUFSIZE);
      sprintf(buff, "%d", nread);
      write_stream(sock, buff, BUFSIZE);
      if(nread == 0)
```

```
         break;
      write_stream(sock, shmsg, MAX_STREAM_SIZE);
      }

   if(shmdt(shmsg) < 0) {
      fprintf(stderr, "\n\nServer can't detach memory\n");
      exit(1);
      }

   if(shmctl(shmid, IPC_RMID, (struct shmid_ds *) 0) < 0) {
      fprintf(stderr, "\nServer can't remove shared memory\n");
      exit(1);
      }

   close(fd);
   sprintf(buff,"rm %s", filename);
   system(buff);
   return;
   }

/***************************************************************************
*                        ------ serv_external ------                       *
*     Service an external request.                                         *
*     Executed by:  Remote Server                                          *
***************************************************************************/
void serv_external(sock)
int sock;
   {
   int   pid, status;
   char  buff[BUFSIZE];
   char  executable[81];
   char  loc_filename[81];
   char  *user;
   char  shmkey_str[10];
   int   ACCEPTANCE;

   printf("\nServer: %s - External connection.\n", myself);

   signal(SIGCHLD, SIG_DFL);
   memset(buff, (char)0, sizeof(buff));
                    /*   Acceptance decided by LOAD   */
   ACCEPTANCE = local_load_ok();
   sprintf(buff, "%d", ACCEPTANCE);
   write_stream(sock, buff, BUFSIZE);

   if(!ACCEPTANCE)
```

```
      return;           /*   Return if not accepting request   */

   get_shared_mem();
   read_stream(sock, buff, BUFSIZE);

   sscanf(buff,"%s",executable);
   printf("\nSERVER : EXEC = %s\n", executable);
   read_stream(sock, shmsg, MAX_STREAM_SIZE);

                 /* read the Random number data into shared memory */
   read_stream(sock, shmsg+MAX_STREAM_SIZE, BUFSIZE);

#ifdef DEBUG
   call_unpacker(shmsg);
#endif

   user = getenv("USER");
   sprintf(filename, "F%d.%s.tmp",SHMKEY, user);

   pid = fork();
   switch(pid) {
      case -1:
         perror("Error forking process");
         exit(1);

      case  0:
         close(sock);
         sprintf(shmkey_str, "%d",(int)SHMKEY);
                 /* the shared memory key send to child */
         strcpy(loc_filename, filename);
         execlp(executable, executable, shmkey_str,loc_filename,(char *)0);
         perror("Error in EXEC\n");
         exit(1);

      default:
         break;
      }

   pid = wait(&status);

       /* The rest will be executed only when the child exits */
   printf("\nOutPut in %s\n", filename);
   printf("\n*********************************************************\n");
   sec_child_exit_rem(sock);
   }
```

```
/**************************************************************************
*                      ------ unpack_int ------                           *
*     Unpacks an integer from the character array.                        *
*     Executed by:  Remote Server                                         *
**************************************************************************/
char * unpack_int(msg, num)
char *msg; int *num;
   {
   int tmp, i;
   char *ptr;

   ptr = (char *)&tmp;
   for(i = 0; i < sizeof(int); i++) {
      (*ptr) = (*msg);
      ptr++; msg++;
      }
   (*num) = (int)tmp;
   return msg;
   }

/**************************************************************************
*                      ------ unpack_float ------                         *
*     Unpacks a floating point value from the character array.            *
*     Executed by:  Remote Server                                         *
**************************************************************************/
char * unpack_float(msg, num)
char *msg; float *num;
   {
   int i;
   float tmp;
   char *ptr;

   ptr = (char *)&tmp;
   for(i = 0; i < sizeof(float); i++) {
      (*ptr) = (*msg);
      ptr++; msg++;
      }
   (*num) = (float)tmp;
   return msg;
   }

#ifdef DEBUG
/**************************************************************************
*            ++++----++++  TESTING THE PACKED MESSAGE  ++++----++++       *
*                                                                         *
*                             1. call_unpacker                            *
*                              2. unpack_string                         *
*                              3. unpack_long                           *
*                              4. unpack_msg                            *
*************************************************************************/
/*************************************************************************
*                     ------ call_unpacker ------                        *
*       Call the unpaking routine.                                       *
*************************************************************************/
void call_unpacker(msg)
char *msg;
   {
   SIM_INFO    sim;          SYS_INFO    sys;
   INFIL_INFO  infil;        ET_INFO     et;
   IRRI_INFO   irrig;        OUT_SEL     output;
   long        offset;       CROP_INFO   crop;
   int         block;        char        fir_str[STR_L_LEN];
   FILE        *fp_in;       int         sim_number;

   unpack_msg(msg, &sim, &sys, &infil, &et, &irrig, &output, &offset,
                                 &crop, &block , fir_str, &sim_number);
   }

/*************************************************************************
*                     ------ unpack_string ------                        *
*       Unpacks a string from the character string.                      *
*************************************************************************/
char * unpack_string(msg, str, strlen)
char *msg; char *str; int strlen;
   {
   int i;

   for(i = 0; i < strlen; i++) {
      (*str) = (*msg);
      msg++;
      str++;
      }
   return msg;
   }

/*************************************************************************
*                     ------ unpack_long ------                          *
*       Unpacks a long integer from the character string.                *
*************************************************************************/
char * unpack_long(msg, num)
char *msg; long *num;
   {
```

```
   int i;
   long tmp;
   char *ptr;

   ptr = (char *)&tmp;
   for(i = 0; i < sizeof(long); i++) {
      (*ptr) = (*msg);
      ptr++; msg++;
      }
   (*num) = (long)tmp;
   return msg;
   }

/**************************************************************************
*                      ------ unpack_msg ------                           *
*       Unpacks the character array.                                      *
**************************************************************************/
void unpack_msg(msg, sim, sys, infil, et, irrig, output, offset, crop,
                block, fir_str, sim_number)
          char        *msg;     SIM_INFO  *sim;      SYS_INFO   *sys;
          INFIL_INFO  *infil;   ET_INFO   *et;       IRRI_INFO  *irrig;
          OUT_SEL     *output;  long      *offset;   CROP_INFO  *crop;
          int         *block;   char      *fir_str;  int        *sim_number;
   {
   int i = 0;
   int loc_seed, seed, old_seed, no_header, change_weather;

                           /* unpack SIM_INFO */
   msg = unpack_string(msg, sim->input_file, NAME_LEN);
   msg = unpack_string(msg, sim->output_file, NAME_LEN);
   printf("\nSim: %s  %s ", sim->input_file, sim->output_file);
   msg = unpack_int(msg, &(sim->begin_day));
   msg = unpack_int(msg, &(sim->begin_year));
   msg = unpack_int(msg, &(sim->end_day));
   msg = unpack_int(msg, &(sim->end_year));
   msg = unpack_int(msg, &(sim->num_sims));
   msg = unpack_int(msg, &(sim->soil_chem_combinations));

   printf("\nSim : %s  %s  %d %d %d %d %d %d\n",sim->input_file,
          sim->output_file, sim->begin_day, sim->begin_year, sim->end_day,
          sim->end_year, sim->num_sims, sim->soil_chem_combinations);

                           /* unpack SIM_INFO */
   msg = unpack_string(msg, sys->index, STR_L_LEN);
   msg = unpack_string(msg, sys->soil.name, NAME_LEN);
   msg = unpack_float(msg, &(sys->soil.curve_no));
msg = unpack_float(msg, &(sys->soil.root_depth));
for(i = 0; i < LMAX+3; i++) {
   msg = unpack_float(msg, &(sys->soil.prop[i].depth));
   msg = unpack_float(msg, &(sys->soil.prop[i].oc));
   msg = unpack_float(msg, &(sys->soil.prop[i].bd));
   msg = unpack_float(msg, &(sys->soil.prop[i].fc));
   msg = unpack_float(msg, &(sys->soil.prop[i].pwp));
   msg = unpack_float(msg, &(sys->soil.prop[i].sat));
   msg = unpack_float(msg, &(sys->soil.prop[i].koc));
   msg = unpack_float(msg, &(sys->soil.prop[i].half_life));
   msg = unpack_float(msg, &(sys->soil.prop[i].kd));
   }
msg = unpack_int(msg, &(sys->soil.no_horizons));
msg = unpack_string(msg, sys->chem_name, CHEM_NAME_LEN);
msg = unpack_int(msg, &(sys->appl.earliest_day));
msg = unpack_int(msg, &(sys->appl.latest_day));
msg = unpack_int(msg, &(sys->appl.day));
msg = unpack_int(msg, &(sys->appl.year));
msg = unpack_int(msg, &(sys->appl.date_type));
msg = unpack_float(msg, &(sys->appl.depth));
msg = unpack_float(msg, &(sys->appl.amount));
sys->depth_units = *(msg);  msg++;
msg = unpack_int(msg, &(sys->resampling));
printf("\nsys->depth = %.2f %.2f\n", sys->appl.depth,sys->appl.amount);

                      /* unpack INFIL_INFO */
msg = unpack_int(msg, &(infil->source));
msg = unpack_string(msg, infil->parm_file, NAME_LEN);
msg = unpack_string(msg, infil->daily_weather_file, NAME_LEN);
msg = unpack_string(msg, infil->daily_rain_file, NAME_LEN);
msg = unpack_int(msg, &(infil->estimator));

                      /* unpack ET_INF */
msg = unpack_int(msg, &(et->source));
msg = unpack_string(msg, et->parm_file, NAME_LEN);
msg = unpack_string(msg, et->daily_weather_file, NAME_LEN);
msg = unpack_string(msg, et->daily_et_file, NAME_LEN);
msg = unpack_float(msg, &(et->par1));
msg = unpack_float(msg, &(et->par2));
msg = unpack_string(msg, et->pan_file, NAME_LEN);
msg = unpack_int(msg, &(et->estimator));

                      /* unpack IRRI_INFO */
msg = unpack_int(msg, &(irrig->type));
msg = unpack_int(msg, &(irrig->datum.peri.begin_day));
msg = unpack_int(msg, &(irrig->datum.peri.end_day));
msg = unpack_int(msg, &(irrig->datum.peri.period));
msg = unpack_float(msg, &(irrig->datum.peri.amount));
msg = unpack_float(msg, &(irrig->datum.demd.critical_depletion));
msg = unpack_float(msg, &(irrig->datum.demd.efficiency));
msg = unpack_float(msg, &(irrig->datum.demd.amount));
msg = unpack_int(msg, &(irrig->datum.demd.begin_day));
msg = unpack_int(msg, &(irrig->datum.demd.end_day));
msg = unpack_string(msg, irrig->datum.file, NAME_LEN);
irrig->amount_units = (*msg);  msg++;

                      /* unpack OUT_SEL */
for(i = 0; i < ARRAY_LEN; i++) {
   msg = unpack_int(msg, &(output->at_depth[i].day));
   msg = unpack_float(msg, &(output->at_depth[i].depth));
   msg = unpack_float(msg, &(output->at_depth[i].amount));
   }
msg = unpack_int(msg, &(output->no_depth));
for(i = 0; i < ARRAY_LEN; i++) {
   msg = unpack_int(msg, &(output->at_time[i].day));
   msg = unpack_float(msg, &(output->at_time[i].depth));
   msg = unpack_float(msg, &(output->at_time[i].amount));
   }
msg = unpack_int(msg, &(output->no_time));
msg = unpack_int(msg, &(output->amount));
msg = unpack_int(msg, &(output->infilET));
output->depth_units = (*msg);  msg++;
msg = unpack_float(msg, &(output->water.et));
msg = unpack_float(msg, &(output->water.pet));
msg = unpack_float(msg, &(output->water.rain));
msg = unpack_float(msg, &(output->water.infil));
msg = unpack_float(msg, &(output->water.irrig));
msg = unpack_float(msg, &(output->water.runoff));

msg = unpack_long(msg, offset);
printf("\nOffset = %ld\n", *offset);

                      /* unpack CROP_INFO  */
msg = unpack_string(msg, crop->name, STR_L_LEN);
msg = unpack_string(msg, crop->coef_file, NAME_LEN);
for(i = 0; i < DAYS_PER_YEAR; i++) {
   msg = unpack_float(msg, &(crop->coef[i]));
   printf("%.2f ", crop->coef[i]);
   if((i+1)%8 == 0)
   printf("\n");
   }
   msg = unpack_int(msg, &(crop->earliest_plant_day));
   msg = unpack_int(msg, &(crop->latest_plant_day));
   printf("\ncrop=%s %s %d \n", crop->name, crop->coef_file,
                                            crop->latest_plant_day);

   msg = unpack_int(msg, block);
   msg = unpack_string(msg, fir_str, STR_L_LEN);
   msg = unpack_int(msg, &(no_header));
   msg = unpack_int(msg, &(change_weather));
   printf("\nFir_str=%s %d %d \n", fir_str, no_header, change_weather);

   msg = unpack_int(msg, &(loc_seed));
   printf("\nSeed = %d\n", loc_seed);
   seed = loc_seed;
   old_seed = seed;
   msg = unpack_int(msg, sim_number);
   return;
   }

#endif
```

VITA

Anil Francis Thomas

Candidate for the Degree of

Master of Science

Thesis: A LOAD SHARING POLICY FOR CPU-INTENSIVE TASKS ON A NETWORK OF INDEPENDENT WORKSTATIONS

Major Field: Computer Science

Biographical Data:

Personal Data: Born in Changanacherry, Kerala, India, on March 25, 1967, the son of S. L. Thomas and Thankamma Thomas.

Education: Graduated from S. B. High School, Changanacherry, India, in 1982; graduated from S. B. College, Changanacherry, 1984, received Bachelor of Technology in Mechanical Engineering from Regional Engineering College, Calicut, India in December 1988. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in December 1993.

Experience: Computer Programmer, Agronomy Department, Oklahoma State University, May, 1992 to December 1993. Senior Engineer, BPL-Sanyo Ltd, Bangalore, India, February, 1989 to December 1991.