MedusaMothra – A Distributed Interpreter for the Mothra Mutation Testing System

Christian Zapf

Department of Computer Science
Clemson University
Clemson, South Carolina 29634-1906

1993

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I
INTRODUCTION

Software testing continues to be an area of active research, since the demand for reliable software is high and will remain high in the future. Reliable software can be achieved through testing, but current testing methods are computationally expensive. Recent developments in the area of distributed computing coupled with ongoing research in software testing provide us with the opportunity to produce testing tools which can be cost effective.

Mutation testing is a technique, originally proposed in 1978 [1], that asks the tester to demonstrate that a program being tested does not contain a finite, well-specified set of faults. The tester does this by finding test cases that cause faulty versions of the program to fail and either get correct output from the tested program (demonstrating its quality) or also cause the tested program to fail (detecting a fault).

Unit level testing techniques, of which mutation is one, hold great promise for improving the quality of software. Unfortunately, the application of these techniques is currently so expensive that we cannot afford to use them.

This paper focuses on and presents two areas of research: first, the development of and experimental results with a distributed implementation of the Mothra mutation testing system called MedusaMothra; and second, the experimentation with a dynamic load balancing technique, which we term *natural load balancing.*

Chapter II introduces the concepts of mutation analysis and shows why this testing method is computationally expensive. Chapter III discusses the Mothra software testing environment. Chapter IV discusses previous work in parallel mutation testing and introduces MedusaMothra. Chapter V looks at the issues and problems involved with load balancing in general and static and dynamic load balancing in particular. Chapter VI gives an overview over the MedusaMothra design issues and highlights some of the implementation details. Chapter VII presents the experimental procedures used, the performance results, and evaluates the natural load balancing model. Chapter VIII provides a concise summary of our results and concludes with suggestions for future work, emphasizing areas that need to be addressed for further improvement.

CHAPTER II

MUTATION ANALYSIS

Mutation testing helps a user create test data by interacting with the user to iteratively strengthen the quality of test data. During mutation testing, faults are introduced into a program by creating many versions of the program, each of which contains one fault. Test data is used to execute these faulty programs with the goal of causing the faulty program to fail. Hence the term mutation; faulty programs are *mutants* of the original, and a mutant is *killed* when a test case causes it to fail. When this happens, the mutant is considered *dead* and no longer needs to remain in the testing process since any faults represented by that mutant would have been detected.

Figure 1 contains a simple Fortran function with three mutated lines (preceded by the Δ symbol).

```
         FUNCTION Min (I,J)
1        Min = I
     Δ   Min = J
2        IF (J .LT. I) Min = J
     Δ   IF (J .GT. I) Min = J
     Δ   IF (J .LT. Min) Min = J
3        RETURN
```

Figure 1. Function Min.

Note that each of the mutated statements represents a separate program. The most recent mutation system, Mothra [2, 3], uses 22 types of mutation operators to test Fortran 77 programs. These operators have been developed and refined over 10 years through several mutation systems. A complete list of the current Mothra mutation operators and their descriptions can be found in Table 1.

The mutation operators are limited to simple changes because of the *coupling effect*, which says that complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect most complex faults [1, 4].

Table 1. Mothra mutation operators.

Mutation Operator	Description
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar variable replacement
DER	`DO` statement end replacement
DSA	`DATA` statement alterations
GLR	`GOTO` label replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	`RETURN` statement replacement
SAN	statement analysis
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement
UOI	unary operator insertion

The mutation testing process begins with the construction of mutants of a test program. The user then adds test cases to the mutation system and checks the output of the program on each test case to see if it is correct. If incorrect, a fault has been found and the program must be modified and the process restarted. If the output is correct, that test case is executed against each live mutant. If the output of a mutant differs from that of the original program over the same test case, it is assumed to be incorrect and the mutant is killed.

After all of the test cases have been executed against all of the live mutants, each of the remaining mutants falls into one of two categories. One, the mutant is functionally *equivalent* to the original program. An equivalent mutant always produces the same output as the original program, so no test case can kill it. Two, the mutant is killable, but the set of test cases is insufficient to kill it. In this case, new test cases need to be created, and the process iterates until the test set is strong enough to satisfy the tester. The *mutation score* for a set of test data is *the percentage of non-equivalent mutants killed by that data.* We call a set of data *mutation-adequate* if its mutation score is 100%. The mutation testing process is depicted graphically in Figure 2. P is the program to be tested and T is the test case set.

The major computational cost of mutation testing is incurred when running the mutant programs against the test cases. Offutt *et al.* [5] have analyzed the number of mutants generated for a program and found them to be roughly proportional to the product of the number of data references times the size of the set of data objects. Typically, this is a large number. For example, 44 mutants are generated for the function `Min` shown in Figure 1. For a typical 30 line subroutine, between 900 and 1000 mutants will be created. Since each mutant must be executed against at least one, and potentially many, test cases, mutation testing requires large amounts of computation.

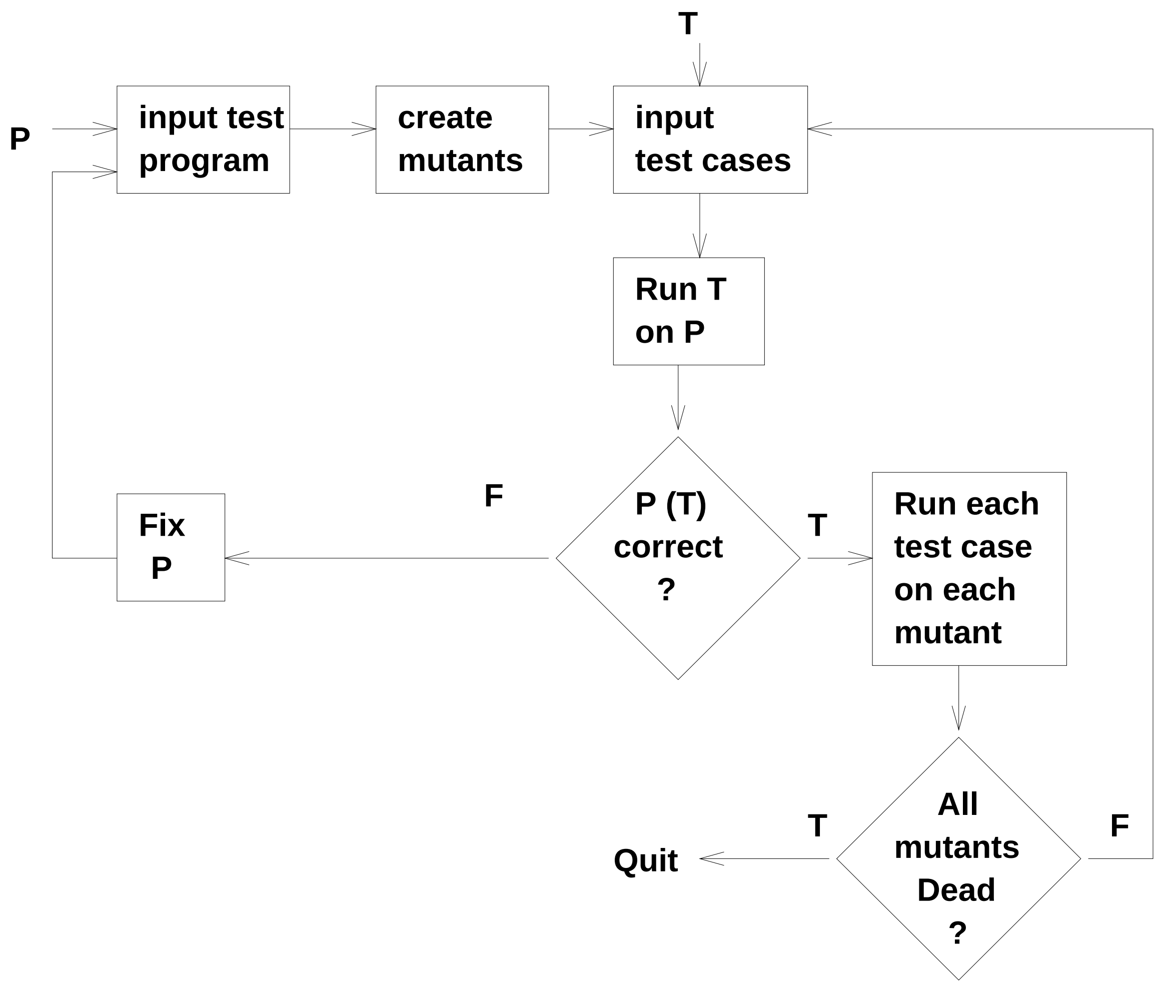

Figure 2. Mutation testing process.

CHAPTER III

THE MOTHRA SOFTWARE TESTING ENVIRONMENT

The Mothra software testing environment was developed at the Georgia Institute of Technology and consists of several tools that allow the user to interactively test a Fortran 77 program [2, 3, 6]. Figure 3 shows the architecture of Mothra and the interactions between the tools [1].

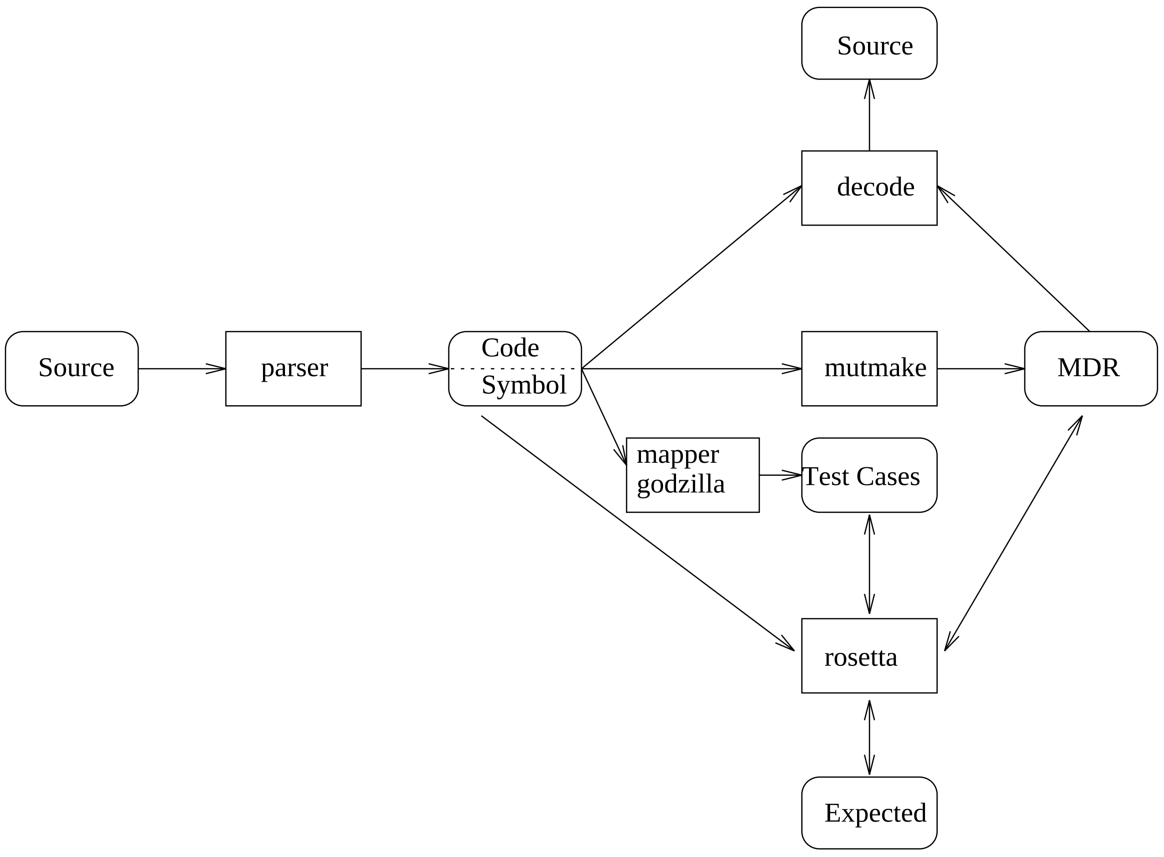

Figure 3. Mothra system architecture.

`Parser` translates a Fortran 77 program into intermediate code and creates symbol table information. `Mutmake` uses the intermediate code and the symbol table information to

[1]Taken from King and Offutt [3]

produce mutant descriptor records (MDRs). `Mapper` allows the user to interactively create and modify test cases. An alternative to `mapper` is `godzilla`, which generates test cases automatically based on path expressions and necessity constraints [7]. `Rosetta` executes intermediate code instructions and can be used in two modes: (1) executing the original program and creating the expected output; or (2) executing the mutant programs on one or more test cases. `Decode` enables the user to view a Fortran 77 program, its intermediate code, mutations, and/or symbol table information. Besides these tools, additional tools allow the user to retrieve status information for a particular testing experiment.

CHAPTER IV

PREVIOUS WORK IN PARALLEL MUTATION TESTING

Several proposals and attempts have been made to implement mutation testing on high-performance machines to speed up the testing process. The target machines for these efforts were vector processors, single-instruction-multiple-data (SIMD) or multiple-instruction-multiple-data (MIMD) machines. This section briefly reviews previous work in parallel mutation testing and discusses two implementations of **HyperMothra** in more detail.

General Work

Mathur and Krauser proposed *mutant unification* [8, 9]. They suggest that vectorizable programs be created, each of which incorporates several mutants of the same type, which are *unified* into one program. Their hope is that a vector processor could then execute the unified mutant programs and achieve a significant speedup over a scalar processor. In a later publication [10], Krauser, Mathur, and Rego expand their concept of *mutant unification* to SIMD machines. Although the simulation results were showing significant speedup, two major problems are still to be overcome in this proposal: (1) the lack of a mutant unifier, and (2) the compilation bottleneck. Neither proposal has been implemented and the authors imply in their papers that both techniques require further research before it can become practical for mutation testing.

The latest efforts to increase the performance of mutation testing have been implemented on multiple-instruction-multiple-data (MIMD) machines. Two main approaches are currently researched: (1) compiling the mutant programs and executing them on node processors; and (2) parallelizing Mothra's interpreter.

Choi, Mathur, and Pattison [11, 12] developed a framework for scheduling mutant executions on the nodes of a hypercube (**PMothra**). Each mutant is separately compiled on a host processor and the resulting executable programs are scheduled for execution on the node processors. Again, the speedup for the tested programs was significant, but the compilation bottleneck makes this system impractical.

HyperMothra and Static Load Balancing

A completely different approach to speed up the mutation testing process has been taken by Offutt *et al.* [13]. Using an Intel iPSC/2 hypercube, the Mothra interpreter is divided into a host interpreter and a node interpreter, and the actual mutant interpretation is moved to node processors. In the resulting tool, **HyperMothra**, a host processor distributes mutants to each node processor and the node processors interpret mutants for each test case. Each test case and its expected output is sent to all nodes. Each node interprets all its mutants against each test case and sends the number of live mutants back to the host processor. Once all test cases have been distributed and executed, the nodes send the remaining mutant information back to the host. In this first implementation of HyperMothra, static load balancing was implemented; i.e., the mutants were distributed to the node processors only once and no mutants were moved from one node to another during execution. Three mutant distribution orders were used: (1) as created by the `mutmake` tool; (2) in random order; and (3) by mutant type (Round-Robin).

Experiments with HyperMothra yielded the best speedup when the mutants were distributed by their type order (Round-Robin). Additionally, the communication overhead is relatively high in comparison to the execution cost when testing small programs, but for larger programs, nearly linear speedups were observed.

HyperMothra and Dynamic Load Balancing

Khambekar [14] used this already existing version of HyperMothra and added two dynamic load balancing schemes to the HyperMothra interpreter, *Buddy* and *Hierarchical.* Both schemes allow for the nodes to exchange mutants if one node finishes applying a test case to its assigned mutants while another node is still executing mutants. This process of dynamic balancing continues until all nodes have finished interpreting the current test case and the next test case is issued. *Buddy* combines several nodes which can exchange mutants during dynamic load balancing efforts, whereas *Hierarchical* establishes a hierarchy between the nodes and uses this hierarchy to perform dynamic load balancing. Although parallelization with static load balancing provided very good speedup and consequently high efficiency, dynamic load balancing resulted in an even higher efficiency, with both balancing schemes performing nearly identically.

Discussion

The currently existing approaches to parallelize mutation testing have three problems. First, they were either only partially implemented or not implemented at all; second, some existing implementations have limitations that make them impractical for extensive testing applications; and third, current implementations use special, not necessarily widely available architectures such as vector processors or hypercubes for parallel mutant execution.

The remainder of this paper presents a distributed mutation testing system, **MedusaMothra**. MedusaMothra is fully implemented within the Mothra software testing environment and has the same functionality as the Mothra mutant interpreter `rosetta`. Additionally, MedusaMothra does not require a specialized hardware architecture such as a hypercube for distributed mutant execution and can be executed in the same computing environment as all the existing Mothra tools. MedusaMothra also uses a different dynamic load balancing scheme than dynamic HyperMothra.

CHAPTER V

LOAD BALANCING TAXONOMY

In the latter part of the previous chapter, we described the static and dynamic load balancing techniques used in HyperMothra. Since MedusaMothra employs a specific dynamic load balancing technique, some major issues and problems of load balancing in general are discussed in this chapter. We also show how these issues influenced the design of the two HyperMothra implementations. (In this paper, we will refer to Fichter's implementation of HyperMothra [15] as *static HyperMothra* and to Khambekar's implementation of HyperMothra [14] as *dynamic HyperMothra*.) It has to be mentioned that dynamic HyperMothra was implemented in the context of a much broader study of load balancing on several different architectures, whereas static HyperMothra only focused on one static load balancing scheme for one specific architecture, an Intel iPSC/2 hypercube. As a result, we focus our discussion more on dynamic HyperMothra.

Load balancing is part of the broad problem of resource allocation; its main problem is how to distribute tasks among processors connected by a network to equalize the workload among the processors. (In this paper, all load that needs to be balanced will be referred to as "tasks".) As a result, the execution time of an application will be reduced. These issues need to be addressed when talking about load balancing [14, 16, 17, 18].

Location of Decision. Status information of all processors and execution environments can be collected at one centralized location where a scheduler is used to dispatch tasks to suitable locations for processing; or the decision making process can be physically distributed among the processors that use information stored in many places. Simplicity gives the centralized approach a major advantage, but at the same time, it suffers from several drawbacks such as congestion at the central processor and low reliability in case of failure of the central processor. In the hierarchical implementation of dynamic HyperMothra, processors are grouped together with an assigned group head. Each member of any particular group keeps its own status and each group head maintains condensed load status information on the group as a whole. As a result, the hierarchical implementation of dynamic HyperMothra is mostly distributed with some partial centralizations at the group heads. The decision-making process in the buddy implementation of dynamic HyperMothra is purely distributed.

Load Granularity. The granularity of load can be divided into three levels [14]: process, subtask, and data–item level, each representing, for example, entities such as processes, multiple related data structures, and elements of an array, respectively. The granularity of load in both load balancing schemes of HyperMothra is at the mutant and/or test case level; i.e., mutant and/or test case information is exchanged between processors (data–item level).

Load Estimation/Calculation. Load estimation/calculation should include processor, process, and environment characteristics. The load is calculated as either the number of tasks in the queue of each processor, the amount of data that needs to be processed by each processor, the total time requirement, or the throughput at each processor. Usually, the load on each processor is only estimated; in a few cases it is either exactly known or assumed to be known *a priori*. The hierarchical scheme of dynamic HyperMothra does not specify the type of load calculation; the buddy implementation uses the task queue length for the load calculation. Both load balancing schemes use only local processor load information for load calculation and do not take process and environment characteristics into consideration.

Homogeneity of Processors. It is important to take the capabilities of all processors involved into consideration when selecting or designing a load balancing method. For example, some processors might be limited to integer arithmetic or some others may have added vector or graphics capabilities. Both load balancing schemes of dynamic HyperMothra can be applied to heterogeneous systems. Khambekar notes in his dissertation [14] that a heterogeneous system with processors having different CPU and/or I/O speed can be handled by simple modifications in the load calculation. Heterogeneous systems in which some processors have special resources need to be handled by having inhibitors when the load to be shared requires those resources and the original requestor does not possess them.

The above mentioned issues of load balancing cover topics that are relevant to static and dynamic load balancing. Since HyperMothra is implemented with static and dynamic load balancing, we further identity some issues that are specific to static and dynamic load balancing and discuss how these issues were implemented in both versions of HyperMothra.

Static Load Balancing

In a system that employs static load balancing, decisions about the locations of all tasks are made before the tasks start execution. This *a priori* assignment of tasks to processors is made deterministically or probabilistically. A task is not moved from one location to another after it has been assigned to a processor. Since no system state information has to be maintained, Goscinski [17] identifies the simplicity of static load balancing algorithms as the principal advantage. Static load balancing algorithms, however, fail to adjust to changes in the system load and ultimately can suffer from poor resource utilization.

Static HyperMothra assigns mutants to nodes of the hypercube. If static HyperMothra can claim the requested number of nodes, the mutant execution will take place; otherwise, the user either has to decrease the number of requested nodes or wait until the requested number of nodes becomes available. No system state information is maintained, because once static HyperMothra claims a particular node, it does not have to compete with other processes for CPU time on that particular node; i.e., that node is *owned* by static HyperMothra until mutant execution is finished.

Dynamic Load Balancing

In contrast to static load balancing, dynamic load balancing attempts to equalize system load during execution time. These dynamic load balancing issues need to be addressed.

Transfer Policy. The transfer policy determines at what load level a processor should transfer out or transfer in a task. A transfer can be decided on no load data, on the average or difference of the load among the sender and receiver processors, or on the difference from the average load on the system. As a result, the transfer policy can be “Don’t Care,” a fixed threshold, or average load. Dynamic HyperMothra uses a threshold to determine when to transfer tasks from one processor to another. The load on a particular processor takes on one of two states: *lightly loaded* and *heavily loaded*. It should also be noted that operating system support for dynamic load balancing is not a common feature in most generally available systems. *Process migration* is a facility to dynamically relocate a process from the processor on which it is executing to another processor. In most cases, moving a process from one processor to another requires extensive kernel support which may not be provided (SunOS 4.1.1, for example, which was used during the MedusaMothra development, does not support process migration).

Initiation Type. Some dynamic load balancing schemes off-load tasks or attempt to off-load tasks from heavily loaded systems (**load-driven/source-driven** approach), whereas in the **demand-driven/server-driven** approach, idle processors can initiate the action of balancing by requesting tasks from other processors. The second approach reduces the overhead of initiation from already busy processors and places it on idle processors. In a third scheme, some systems periodically exchange status and/or balance tasks. Both load balancing schemes implemented in dynamic HyperMothra use a demand-driven approach.

Amount of Status Data Exchanged. Since network traffic has a strong impact on the performance of dynamic load balancing algorithms, the amount of exchanged data between processors and the number of data exchanges between processors are important factors for designing dynamic load balancing algorithms. The amount of information exchanged involves a tradeoff between having a high level of detail and minimizing the volume of network traffic. It has been shown [19, 20] that extremely simple scheduling policies provide dramatic performance improvement relative to no load balancing. The performance achieved with simple policies is close to that which can be expected from complex policies. Therefore, the amount of information exchanged should be minimized.

Number of Status Data Exchanges. The number of data exchanges between processors represents a tradeoff between the volume of network traffic and the desire to communicate with every node in the system. A computer should send load information to every remote computer, as this provides the greatest number of potential destinations for migrating tasks. The problem of how often load information should be exchanged is still an open problem [21]. Dynamic HyperMothra uses a broadcast approach to notify other nodes about load changes (lightly-loaded or heavily-loaded). In the hierarchical scheme, a node informs the node above it in the hierarchy about the status change. This new information change is propagated all the way to the top of the hierarchy and therefore available to all the other nodes. In the buddy implementation a node broadcasts a load change to its "buddies"; i.e., the nodes it was initially grouped with.

In this taxonomy, we deliberately chose just to emphasize the main load balancing issues for static and dynamic load balancing that influenced the design of both HyperMothra versions. In the next chapter, we will discuss how these load balancing issues influence the design of MedusaMothra.

CHAPTER VI

MEDUSAMOTHRA DESIGN AND IMPLEMENTATION

Design Goals

Two main goals were set at the beginning of the design of MedusaMothra:

1. *Speed up the mutation testing process.* Ideally, we would like to see *linear speedup.* Linear speedup is the ideal case where if **n** processors are used, a problem is solved **n** times faster than if a single processor is used. In practice, linear speedup can rarely be achieved, primarily because of the communication overhead.

2. *Efficient dynamic load balancing.* Ideally, all client processors would be fully available and we would like to see exact balance on all clients; i.e., each client performs the same amount of work. A hypercube is an example of a system that provides dedicated nodes. In practice, this exact balance cannot be guaranteed because of the multi-user capabilities of the used Sun workstations. Other users can start processes on any particular client and therefore reduce the CPU-time for the mutation testing process on that particular client. The dynamic load balancing scheme should recognize this situation and distribute tasks in a way that minimizes the running time of the mutation testing process.

These two issues were the main motivations for the MedusaMothra implementation The following sections describe how these goals influenced the design of MedusaMothra.

General Design Issues

Several issues were taken into consideration during the initial design phase of MedusaMothra. We examined the existing implementations of HyperMothra to see if the design goals could be met by porting an existing HyperMothra version to Sun workstations. In Chapter V, we mentioned that static HyperMothra deterministically assigns mutants to nodes and does not take any load information into consideration. This strategy works well on an architecture such as the hypercube because once an application "owns" a particular node, no other application or process can claim that node. Since Sun workstations are multi-user workstations and therefore display a much more dynamic behavior in terms of running multiple processes at the same time, the static HyperMothra load balancing approach would have been unable to respond to the dynamic changes in a computing environment comprised of Sun workstations and as a result would suffer from poor resource utilization.

The load balancing of dynamic HyperMothra seemed to be more suited for a computing environment with Sun workstations since dynamic load balancing schemes take the system load into consideration. Both load balancing implementations targeted tightly-coupled multi-processor systems. Since the hypercube, for example, provides dedicated node processors that do not share CPU-time with any other user, both dynamic load balancing schemes initially distribute mutants to the processors. This initial distribution of mutants could result in a performance degradation for both dynamic HyperMothra load balancing schemes in a network of multi-user workstations. In the hierarchical scheme, it is possible that some overloaded workstations are assigned mutants during the initial mutant distribution. The hierarchical load balancing scheme would resolve this situation by redistributing the mutants from the overloaded workstation to some unloaded workstations. Since mutants are assigned to every workstation in the hierarchy, the unloaded workstations have to execute the assigned mutants first before requesting work from another workstation. In the worst case, the overloaded workstations would have executed no mutants and as a result, all their assigned mutants would be distributed to other workstations causing additional communication overhead. In the buddy load balancing scheme, mutants are grouped together in sets. In a network of multi-user workstations, a particular buddy set could be comprised of only overloaded workstations. Since work is balanced only within a particular buddy set, this dynamic load balancing scheme will balance work only on overloaded workstations and will not take any other, possibly unloaded workstations outside its buddy set into consideration. As a result, performance of HyperMothra could be decreased.

Both implementations meet the first goal of speeding up the mutation testing process, but we felt that the second goal, efficient dynamic load balancing in a network of Sun workstations, would not be completely satisfied if we ported an already implemented HyperMothra load balancing scheme to a computing environment comprised of Sun workstations. As a result, we developed a new dynamic load balancing scheme which we term *natural load balancing.*

Natural load balancing is *demand–driven.* Clients request work from a particular server and the server responds to a request with more information. The location of decision is completely determined by the client and therefore purely distributed. The server is not involved at all in the decision–making process, but only responds to the client's requests. As a result, the server does not have to collect status data from any of the clients, which minimizes the network traffic volume and overhead. No status data of any arbitrary client is used to influence the decision on where to distribute more work. Clients only request additional work when they finish a previously assigned task.

A centralized approach was chosen mainly because of its simplicity. In a centralized model, one server accumulates all the necessary information and controls the access of the clients to this information. Figure 4 depicts a possible configuration consisting of one server and four clients.

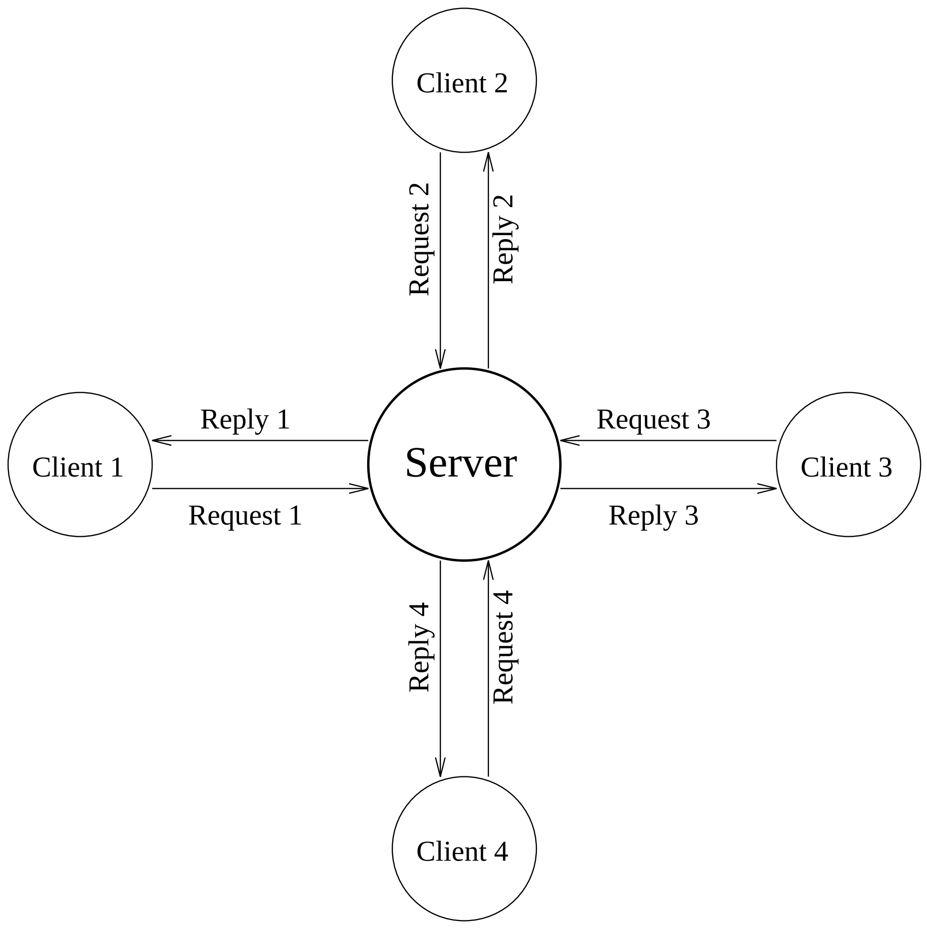

Figure 4. Centralized model.

MedusaMothra employs the following communication scheme between the server and any particular client. After making itself known to the server, a client initially receives some initialization data and a (mutant, test case) pair, specifying the test case and the mutant that needs to be executed. This initialization data is sent once in the beginning from the server to the client, and after a particular client has initialized its data structures, it executes the mutant against the test case and determines the status (alive or killed) of the mutant. The status is sent back to the server. Once the server receives a status from a particular client, a new (mutant, test case) pair is produced and sent to the client. Additionally, the server updates the mutant descriptor file if necessary. The client executes this new mutant against the received test case and determines its status. Figure 5 shows this recurring communication between the server and any arbitrary client. When all mutants have been executed against all test cases, a termination message is sent to all clients that request more work.

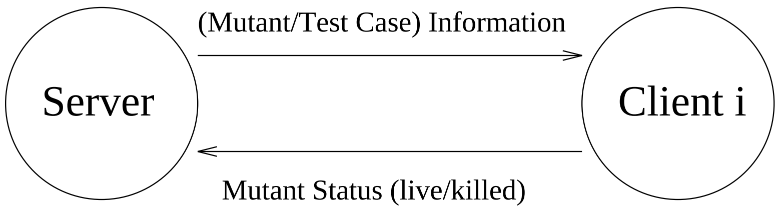

Figure 5. Communication between server and clients.

Figures 6 and 7 show the server and client algorithms. The server has two modes of operation; original and mutant execution. During original execution, the expected output for the original program is created. The expected output for each test case is used later during mutant execution when expected output must be compared with mutant output. No mutant programs are executed during original execution. Experience with Mothra indicates that original output generation can be done efficiently on one workstation. During mutant execution, the server accepts requests from the clients and generates the next (mutant, test case) pair. The (mutant, test case) pairs are indices for the mutant descriptor file and the test case file.

```
begin MedusaMothraServer

   read symbol table information
   read intermediate code
   read test case values

   if (original execution) then
      for (each selected test case) do
         interpret original intermediate code
         write expected output to output files
      endfor
   else
      while (any client connected)
         read request from a client
         generate next mutant and test case pair

         if (no more mutants or test cases) then
            send "Good-Bye" message to client
            mark client as disconnected
         else /* initial request */
            if (first request from client) then
               send initialization information and
               (mutant, test case) pair to client
            else /* result from previous mutant execution */
               send (mutant, test case) pair to client
               if (mutant status == dead) then
                  mark mutant as killed in mutant descriptor file
               endif
            endif
         endif
      endwhile
   endif
end MedusaMothraServer
```

Figure 6. MedusaMothra server algorithm.

```
begin MedusaMothraClient

    write initial request message to server
    read initialization information and
        (mutant, test case) pair from server

    read symbol table information
    read intermediate code
    read test case values

    while ("Good-Bye" message not received from server) do
      generate received mutant
      execute generated mutant against received test case
      if (interpretation ends abnormally ||
        mutant output values are incorrect) then
        mark mutant status as killed
      endif

      send mutant status back to server
      read new (mutant, test case) pair from server
    endwhile

end MedusaMothraClient
```

Figure 7. MedusaMothra client algorithm.

Implementation Details

Two protocol types for ARPA Internet communication are currently supported by SunOS Release 4.1.1, which was used for the Implementation of MedusaMothra. The Transmission Control Protocol/Internet Protocol (TCP/IP) and the User Datagram Protocol (UDP). TCP is the virtual circuit (connection-oriented) protocol of the Internet protocol family and provides reliable, flow-controlled, in-order, two-way transmission of data. The connection establishment is achieved via a procedure called "three-way handshake," and significant state information is kept. UDP is a connectionless protocol, which provides unreliable delivery; acknowledgements are not used to confirm message delivery, and messages may arrive out of order. As a result, messages can be lost, duplicated, or arrive out of order. The major advantage of the connectionless service is the that the protocol is simple, data delivery is fast, and there is no connection management (establishment, release) overhead [17, 22]. However, the data delivery is unreliable which requires an underlying protocol manager to correct the problems introduced by UDP. In contrast, TCP provides reliable service, but because of the overhead created by the establishment and release operations for the virtual circuits, it is less efficient.

MedusaMothra uses TCP/IP for its inter–process communication (IPC); i.e., the communication between the server and the clients. We chose TCP for two reasons. First, initial experiments and studies with both protocol types indicated that using a protocol manager to correct UDP problems can actually thrash the server workstation when many messages are sent. Second, considering the running time (usually at least several minutes) of MedusaMothra experiments, the overhead created by the establishment and release operations for TCP connections can be neglected.

The Network File System (NFS) is used to perform transparent file access over the network. The server and the client need permanent access to the mutant descriptor file, the symbol table file, and the test case value file in order to create the necessary mutants and test cases. Since only the server writes to the mutant descriptor file, any update/timing problems introduced by NFS were also eliminated. We also used NFS to make the server name and the port number at which the server accepts connections from any particular client well-known to the clients. The server name and the port number are stored in a data file called “SERVER” and any client must have access to this file in order to establish a connection with the server. Additionally, the names of all clients are stored in a data file called “CLIENTS”. This file is only read by the server and is used as a security mechanism to prevent unauthorized or unknown clients from connecting and sending messages to the server.

The *select()* system call is used in the MedusaMothra server to multiplex between the virtual circuits with the clients. Each virtual circuit is associated with one specific I/O descriptor. The *select()* system call examines I/O descriptor sets to see if some of the descriptors in a particular I/O descriptor set have data that is ready for reading, writing, or have an exceptional condition pending. Using the select statement ensures that no connections or data transmissions are pending without the server knowing about it. Figure 8 shows the use of the *select()* system call in the MedusaMothra server.

The server listens to a well–known socket which the clients use to establish the initial virtual circuit and adds it to the I/O descriptor set. Once the select() call indicates that some data can be retrieved at any of the I/O descriptors in the set, priority is given to the initial connection requests of any client in an effort to maximize the number of known clients to the server.

User Interface

We use the *rsh* facility to start MedusaMothra clients on any particular client. *Rsh* connects to a specified client and executes a specified command. For MedusaMothra, *rsh* connects to a particular client specified in the “CLIENTS” file and executes the Medusa-Mothra client. A shell script lets the user specify the number of clients. The shell script sequentially reads the “CLIENTS” file and starts the MedusaMothra client on the specified clients. The source code for the shell script can be found in Appendix A. By keeping the information of the server name, the server socket number, and the client names in data files, we enable the user to change the configuration of MedusaMothra easily and without recompiling any of the source code. As mentioned earlier, the “SERVER” file contains the name of the server and the port number it listens to. Port numbers are user specified and can take on any value from 4096 to 65535. Figures 9 and 10 depict a sample “SERVER”

```
          .
          .
          .
create well-known socket for clients to connect to
add well-known socket to descriptor set
listen to well-known socket
          .
          .
          .
while (more test cases) do
  rval = select(descriptor set)
  if (rval == 0) then
    select timed out; no data at any descriptor
  else
    if (connection request on well-known socket) then
      accept new client
      establish virtual circuit
      add new client to descriptor set
    else
      for (each virtual circuit with pending data) do
        read data from client
        send next (mutant, test case) pair
      endfor
    endif
  endif
          .
          .
          .
endwhile
```

Figure 8. Usage of *select()* in the MedusaMothra server.

and “CLIENTS” file. The number 16 at the beginning of the “CLIENTS” file tells the number of entries in the file. This number is used by the server of MedusaMothra. Currently, MedusaMothra is only implemented for Sun 4 workstations. The issues involved in porting MedusaMothra to a heterogeneous computing environment are discussed in Chapter VII.

```
wayne 5284
```

Figure 9. Sample SERVER file.

```
16
oak
palmetto
poplar
juniper
walnut
mahogany
redwood
cedar
cypress
bamboo
magnolia
sequoia
hedge
banyan
birch
teak
```

Figure 10. Sample CLIENTS file.

CHAPTER VII

EXPERIMENTAL EVALUATION OF MEDUSAMOTHRA

To investigate whether the design goals mentioned in Chapter VI were met, several Fortran 77 programs of various sizes were tested. The first part of this chapter discusses the experimental procedure and the results of the experiments that evaluated the speedup of MedusaMothra. The second part describes the results of some studies that were made with respect to the dynamic load balancing responsiveness and effectiveness of MedusaMothra.

Validation of MedusaMothra

To ensure the correctness of MedusaMothra, the original Sun version of Mothra was used as an oracle throughout the development of MedusaMothra. An oracle is a (hypothetical) entity that knows whether the output of a given test case on the test program is correct [23]. Many test programs were processed with the original Mothra version and MedusaMothra. For each program in the experimental test set, MedusaMothra kills the same mutants as the original Mothra for the same test cases.

Speedup Evaluation

Ten Fortran 77 programs that cover a wide range of applications were selected for the experiments. These programs range in size from 10 to 52 executable statements and had from 196 to 2746 mutants. The programs are described in Table 2. Appendix B contains complete source code listings for each program.

All tests were conducted with a Sun 4/75 workstation as the server and Sun 4/25 workstations as clients and a relatively unloaded Ethernet. To ensure a relatively low load on all the workstations and the Ethernet, all tests were run at very early morning hours.

Experimental Procedure

To measure performance data for MedusaMothra, these steps were performed for each Fortran 77 program.

Parse Program: Intermediate code and symbol table information for the program under test were produced by `parser`.

Table 2. Experimental programs.

Program	Description	Statements	Mutants	Test Cases
BUB	Bubble sort on an integer array	11	338	313
DEADLK	Deadlock avoidance program	52	2746	66
EUCLID	Greatest common divisor	11	196	161
FIND	Partitions an array	28	1022	364
INSERT	Insertion sort on an integer array	14	460	193
PAT	Pattern matching	18	394	274
QUAD	Real roots of quadratic equation	10	359	13
SEARCH	Search for an element in an array	18	370	269
TRITYP	Classifies triangle types	28	951	647
WARSHALL	Transitive closure of a matrix	11	305	269

Generate Mutants: All possible mutants were created using `mutmake`.

Define Variable Classes: The class (in, out, in/out, don't care) of each variable was defined using `mapper`.

Generate Test Cases: `Godzilla` was used to generate a set of test cases. Since `Godzilla` failed or created too many test cases for DEADLK and QUAD, test cases were hand–generated for these programs. The hand-generated test cases attempted to kill all mutants.

Create Expected Output: The MedusaMothra interpreter was run in original execution mode to determine and store the output of the original program for each test case.

Interpret Mutants on One Workstation: The wall time for mutant execution was measured for one workstation (Sun 4/25). This step was necessary since speedup is calculated relative to the execution time of one workstation.

Interpret Mutants on Multiple Workstations: MedusaMothra was run in mutant execution mode with a selected number of workstations (2, 4, 8, and 16) and the wall time for the mutant execution was measured.

Speedup Results

To evaluate the performance of MedusaMothra, the wall time was measured and the resulting speedup was calculated for 2, 4, 8, and 16 client workstations. Wall time can be defined in terms of the time the user has to wait until the server of MedusaMothra finishes execution. The wall time was measured using the *time()* C library function. *Speedup* for **n** processors is defined as the execution time on one processor divided by execution time on **n** processors. Formula 1 shows this relationship.

$$Speedup = \frac{Execution\ Time\ on\ One\ Processor}{Execution\ Time\ on\ \mathbf{n}\ Processors}. \qquad (1)$$

The speedup calculations were done with respect to an “intermediate” version of the rosetta interpreter. For his master’s thesis, Lee [24] had removed some inefficiencies from the original rosetta code, including removing the *fork()* system call from the rosetta interpreter. A more detailed description of the inefficiencies in the original rosetta code can also be found in Fichter’s master’s thesis [15]. We added some additional code for debugging purposes and timing measurements and used this “intermediate” version to measure the running time of the experiments for one workstation.

Figures 11 and 12 show the wall time results for the tested Fortran 77 programs for 2, 4, 8, and 16 clients. Notice that the wall execution time for the computationally more expensive programs such as DEADLK and WARSHALL is decreased by a much larger factor than for computationally inexpensive programs such as EUCLID and QUAD. The wall time for WARSHALL, for example, was reduced from 8047 seconds for one workstation to 596 seconds for 16 client workstations. Figures 13 and 14 depict the speedup (relative to one workstation) for all test programs using 2, 4, 8, and 16 clients. We have also plotted the line which represents *linear speedup* for comparison purposes (denoted **x**). Note that corresponding to the wall execution times, the computationally more expensive programs yield better speedup than the computationally less expensive programs.

These observations about MedusaMothra speedup can be explained by communication overhead cost. If little computation is necessary for a problem, or if more clients are cooperating to solve a problem, then communication overhead accounts for a more significant part of the total execution time. Additionally, the amount of communication increases as the number of clients increases. The results for QUAD illustrate this behavior very well. QUAD is a small, computationally very inexpensive program with a small number of test cases. As a result, we had only limited speedup for 2, 4, and 8 clients; 1.38, 2.42, and 3.62, respectively. For 16 clients, the speedup actually decreased to 2.42 and yielded the same wall time as for four clients. Only 13 clients out of the 16 clients were able to connect

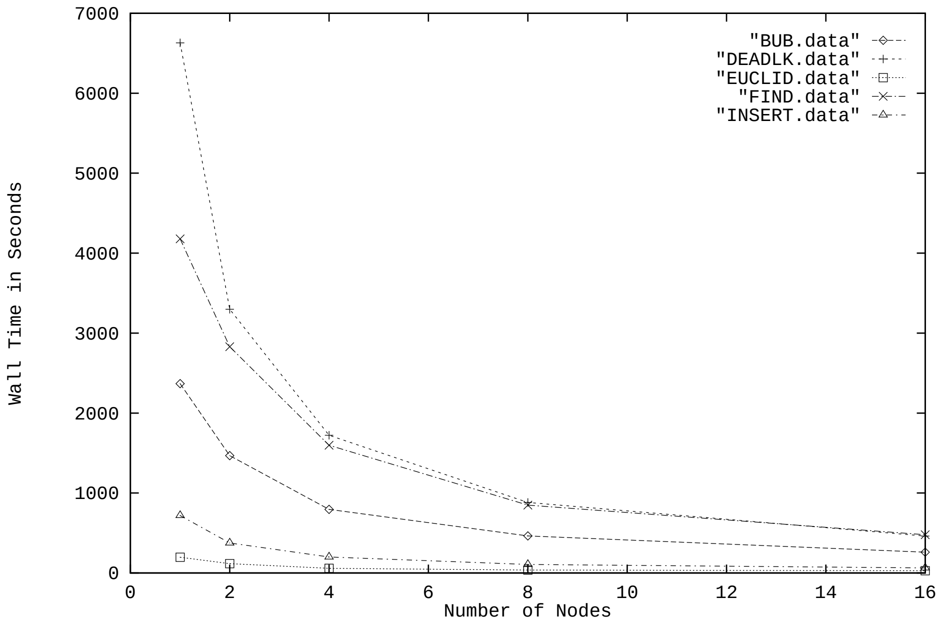

Figure 11. Wall time (Part 1).

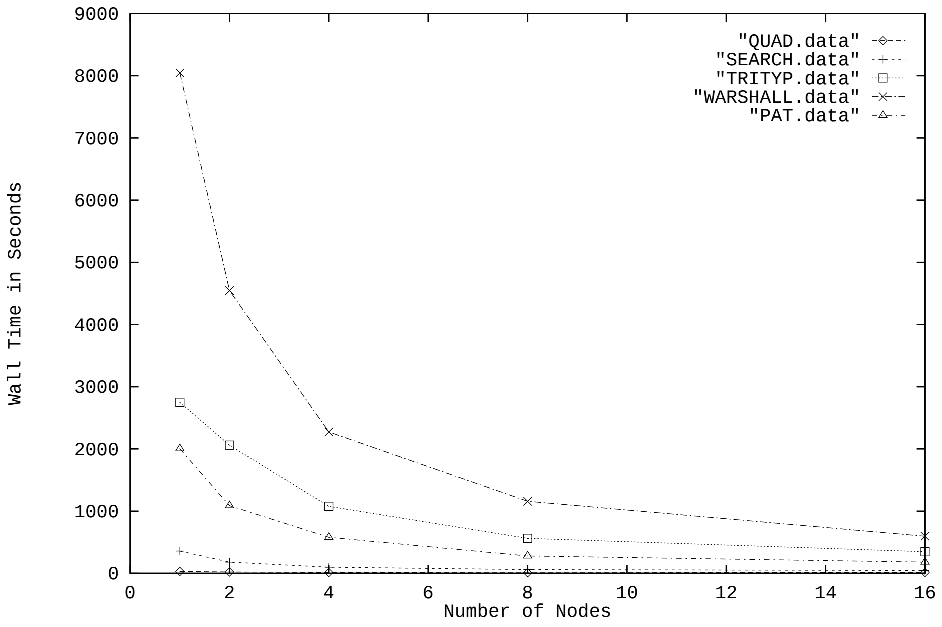

Figure 12. Wall time (Part 2).

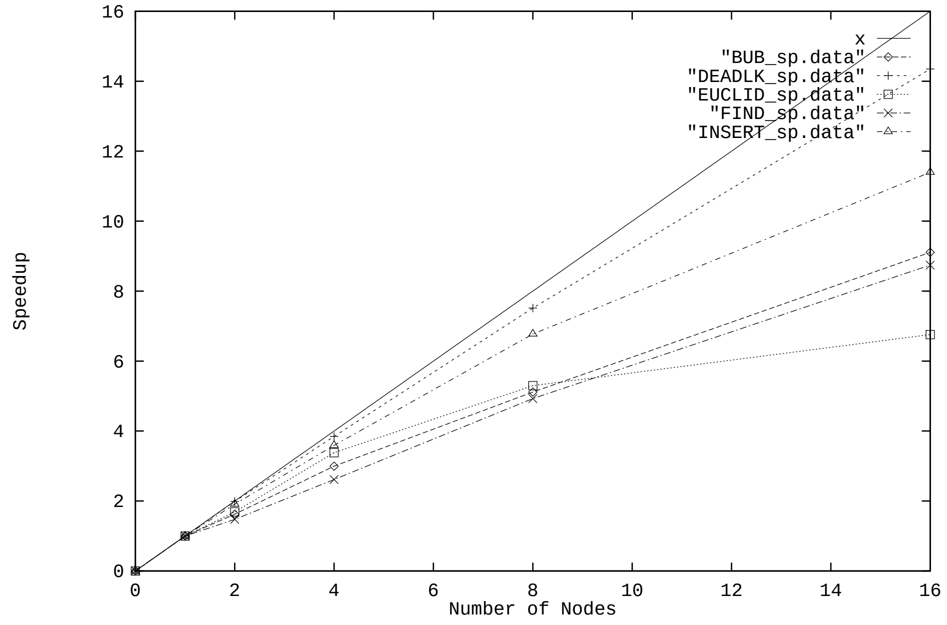

Figure 13. Speedup (Part 1).

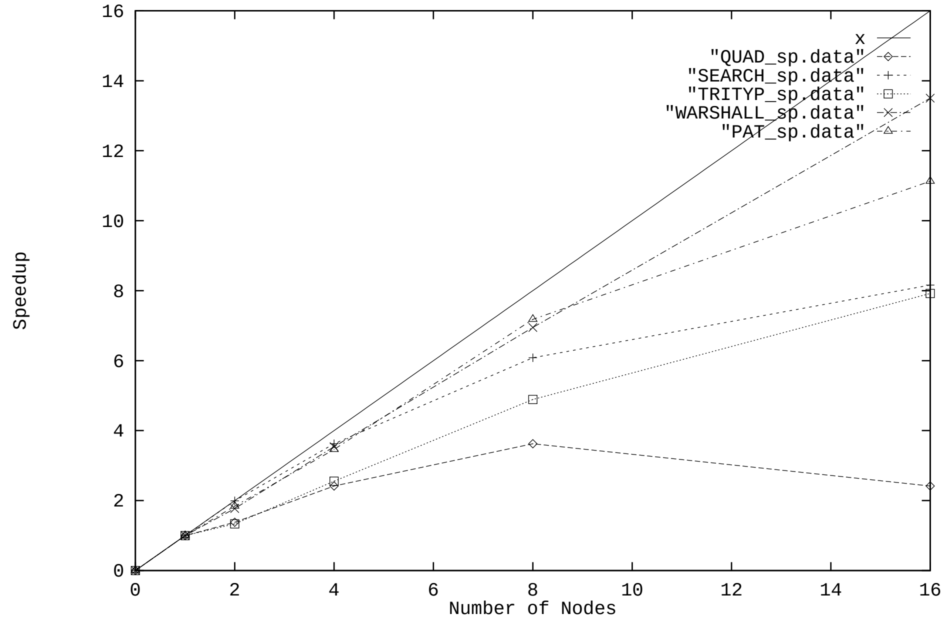

Figure 14. Speedup (Part 2).

to the server and only eleven actually received mutant and test case information. This behavior is a result of the computationally inexpensive characteristic of QUAD, and the size and the quality of the test case set. The test cases for QUAD were hand–generated and attempted to kill all mutants. All other programs yielded increased speedup when the number of clients was increased.

A detailed listing of all experimental data produced for this paper can be found in Appendix C. For each Fortran 77 program, the number of clients, the wall execution time, and if two or more clients were used, the distribution of (mutant, test case) pairs among the clients are listed.

Dynamic Load Balancing Studies

In Chapter VI, we introduced the concept of *natural load balancing.* No status data is used in the natural load balancing scheme to influence the decision on where to distribute more work. Clients only request more work if they have finished a previously assigned task. In this section, we investigate whether the natural load balancing scheme satisfies our design goals for efficient dynamic load balancing and present the results of this investigation.

Experimental Procedure

According to our load balancing design goals, we investigated whether the natural load balancing scheme distributed tasks equally over a network of relatively unloaded workstations. Additionally, we examined how the natural load balancing scheme made adjustments in its task distribution to the clients when two workstations were overloaded with two CPU-bound processes. We selected two Fortran 77 programs, TRITYP and WARSHALL, for these experiments. Both had relatively long wall execution times for eight clients, but TRITYP is computationally cheap whereas WARSHALL is one of the most computationally expensive programs of the programs tested. These steps were performed for both programs.

Interpret Mutants with Eight Clients on a Relatively Unloaded System Medusa-Mothra was run in mutant execution mode and the wall time, the (mutant, test case) distribution, and the load average of all clients over the wall time were measured.

Interpret Mutants with Eight Clients on a Loaded System Again, MedusaMothra was run in mutant execution mode, but this time, two clients were overloaded with two CPU-bound programs on each client. The wall time, the (mutant, test case) distribution, and the load average of all clients over the wall time were measured.

The load average of the clients was measured using the *rup* command. Among other information, *rup* returns the average number of jobs in the run queue over the last 1, 5, and 15 minutes of a remote machine. We used the average number of jobs in the run queue over the last one minute as a measure of how the average load changed when two clients were overloaded. A shell script was used to update all the necessary data of all clients in a circular fashion. In the following two sections, we will refer to "updates" when talking about timing dependencies. These updates are obtained by the previously mentioned shell script and do not correspond to seconds but only to the number of times each client was queried for the average load using the *rup* command. The source code for the CPU-bound program and for the update shell script can be found in Appendix D.

Trityp Results

Figure 15 shows the load average of the eight clients on a relatively unloaded system. Except for two short instances which are unrelated to the TRITYP mutant execution, the average load stays between 0.6 and 0.9. When two clients were overloaded with the CPU-bound programs, the load average for the two overloaded clients increased to around 2.5, but the load average for the other six clients stayed about the same as for the unloaded system. Figure 16 depicts the load averages for the two overloaded clients and the six other clients. These graphs indicate that the natural load balancing scheme does in fact balance tasks equally on all clients. Even for the case with two overloaded clients and six relatively unloaded clients, tasks were distributed equally with respect to the load averages on the overloaded and unloaded clients.

In order to compare the load average of the unloaded and overloaded systems, we calculated the mean average load for the eight unloaded clients of the first test run and for the six unloaded clients of the second test run. The results are show in Figure 17. Notice that although the difference of the average loads is small, the wall execution time for the second test run with the two overloaded clients is slightly longer. To be exact, with the two overloaded clients, the wall time for TRITYP was 610 seconds instead of 562 seconds for the unloaded system. Considering the fact that 25% of the computing capacity was overloaded, this 8.54% increase in wall execution time seems to be minor.

To evaluate whether a redistribution of (mutant, test case) pairs took place, we looked at the number of pairs received and executed by each client. Figures 18 and 19 display graphically the distribution of (mutant, test case) pairs among the eight clients. Although one might expect that the two overloaded clients (in this case *oak* and *palmetto*) actually executed fewer (mutant, test case) pairs, this is not the case for TRITYP. An explanation for this behavior is the fact that TRITYP is computationally inexpensive and was not affected by the overloading of two clients. The only difference between the two distributions is that for the overloaded case, the distribution is not quite as constant. In the unloaded case, the difference between the maximum and minimum number of (mutant, test case) pairs was 235 pairs, whereas in the overloaded case, this difference was 424 pairs.

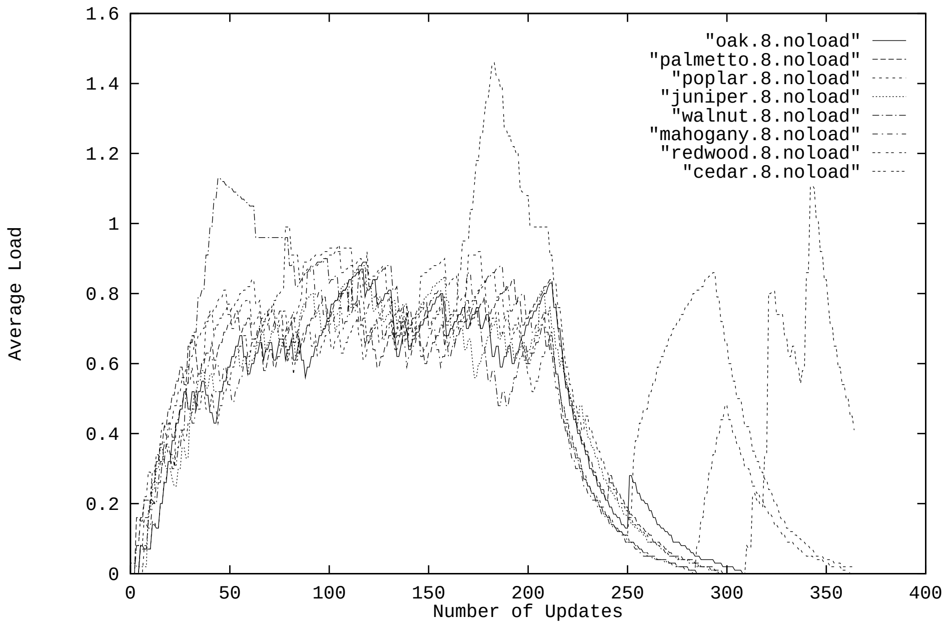

Figure 15. Average load, unloaded – Trityp.

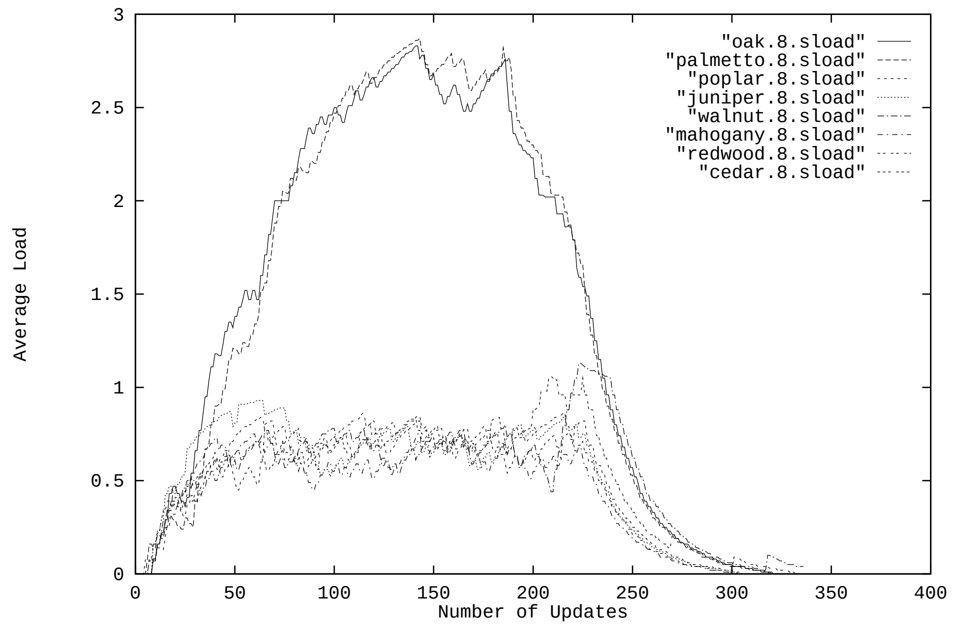

Figure 16. Average load, overloaded – Trityp.

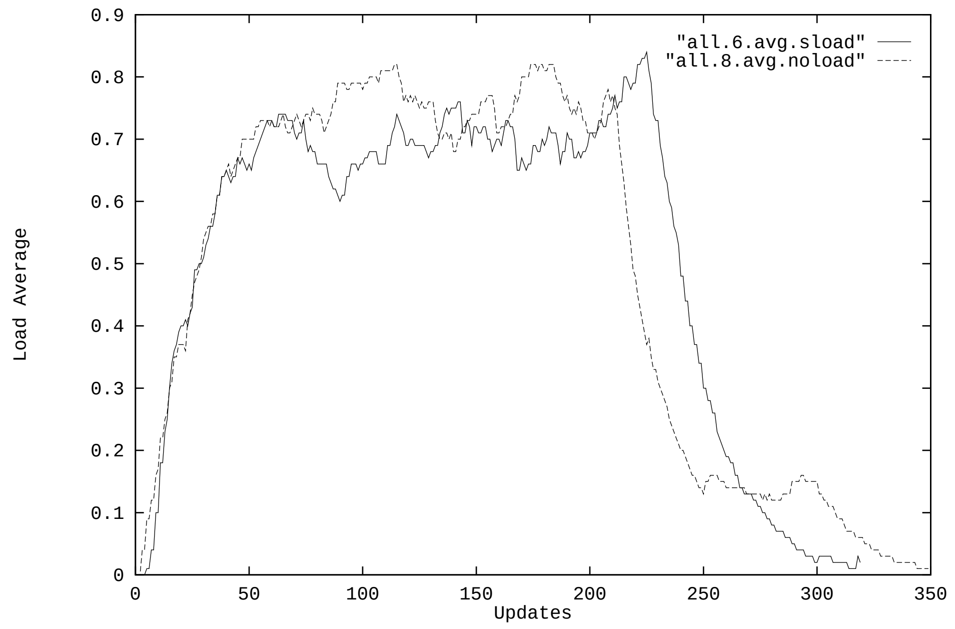

Figure 17. Comparison unloaded/overloaded – Trityp.

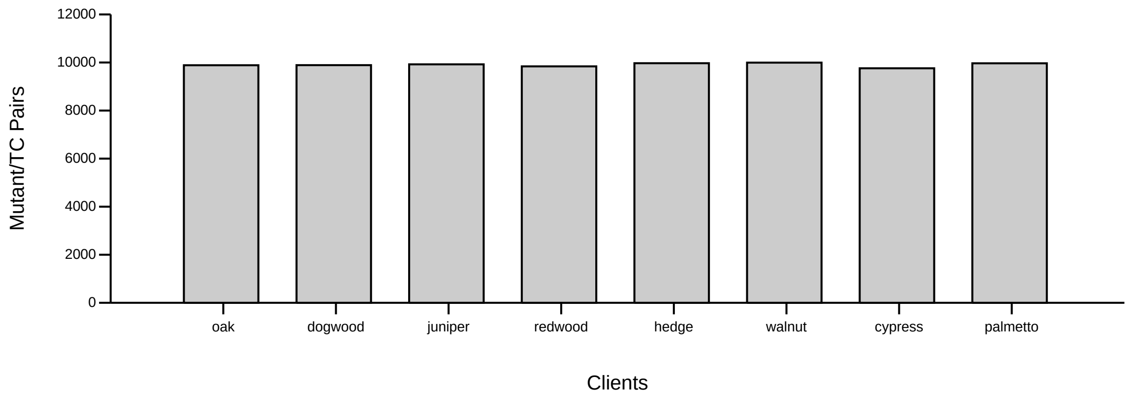

Figure 18. (Mutant/Test Case) distribution, unloaded – Trityp.

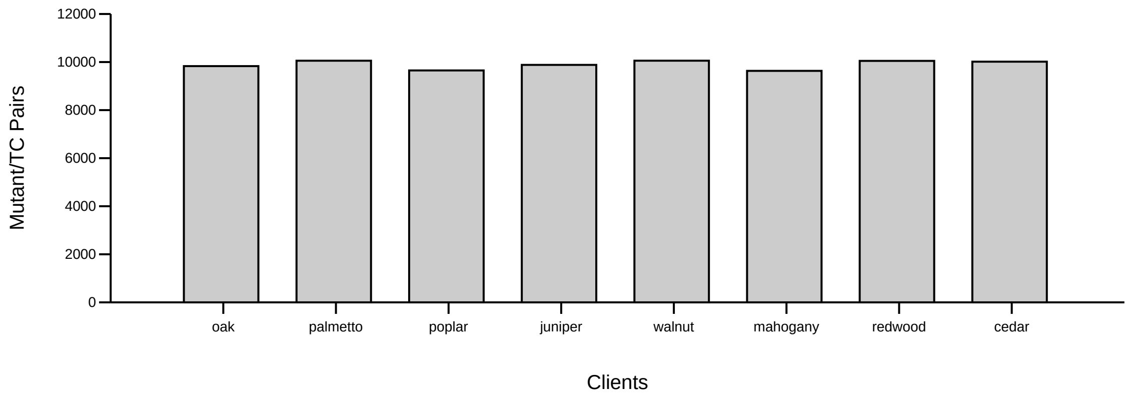

Figure 19. (Mutant/Test Case) distribution, overloaded – Trityp.

Warshall Results

Since TRITYP, with its computationally inexpensive characteristic did not produce any redistribution among the clients used, we expected WARSHALL to do so. Figure 20 shows the load average for WARSHALL of the eight clients on a relatively unloaded system. The load average is not quite as steady as for TRITYP. This can be explained by the nature of the WARSHALL program. WARSHALL consists primarily of three loops and the mutations performed on these loops can result in infinite or long-running executions. When two clients were overloaded with the CPU-bound processes, the load average for the two overloaded clients displayed the same behavior as for the TRITYP experiment; i.e., tasks were distributed equally to all clients.

Figure 21 depicts the load averages for the two overloaded clients and the six unloaded clients. Again, we calculated the mean load average for the eight unloaded clients of the first test run and for the six unloaded clients of the second test run. The results are show in Figure 22. This time, the average load was slightly higher for the six unloaded clients of the second test run and the wall execution time increased by 15.85% from 1180 seconds to 1367 seconds.

To evaluate whether a redistribution of (mutant, test case) pairs took place for WARSHALL, we looked at the number of pairs received and executed by each client. Figures 23 and 24 display graphically the distribution of (mutant, test case) pairs among the eight clients. Notice that for the unloaded case, each client received approximately 8900 (mutant, test case) pairs. In the overloaded case, the two overloaded clients (*oak* and *palmetto*) only processed approximately 4400 (mutant, test case) pairs and the other six unloaded clients processes approximately 10400 clients each.

These results for TRITYP and WARSHALL indicate that although no status information of the clients is used to make decisions about the location of a task, a good distribution of the tasks takes place. Tasks are distributed equally to all clients in a relatively unloaded system for both programs. In the case of overloading two clients, a task redistribution took place to equalize the unavailable CPU-time on the overloaded clients for WARSHALL. For TRITYP, no redistribution took place, because the two CPU-bound processes had almost no influence on the very short execution time of TRITYP. Since no additional network traffic is introduced by collection status information of clients, the **natural load balancing** model works well for MedusaMothra. The experimental data produced for the dynamic load balancing experiments can be found in Appendix E. The number of clients, the wall execution time, and the distribution of (mutant, test case) pairs among the clients are listen for the experiments with two overloaded clients. The data for the experiments on a relatively unloaded system can be found in Appendix C.

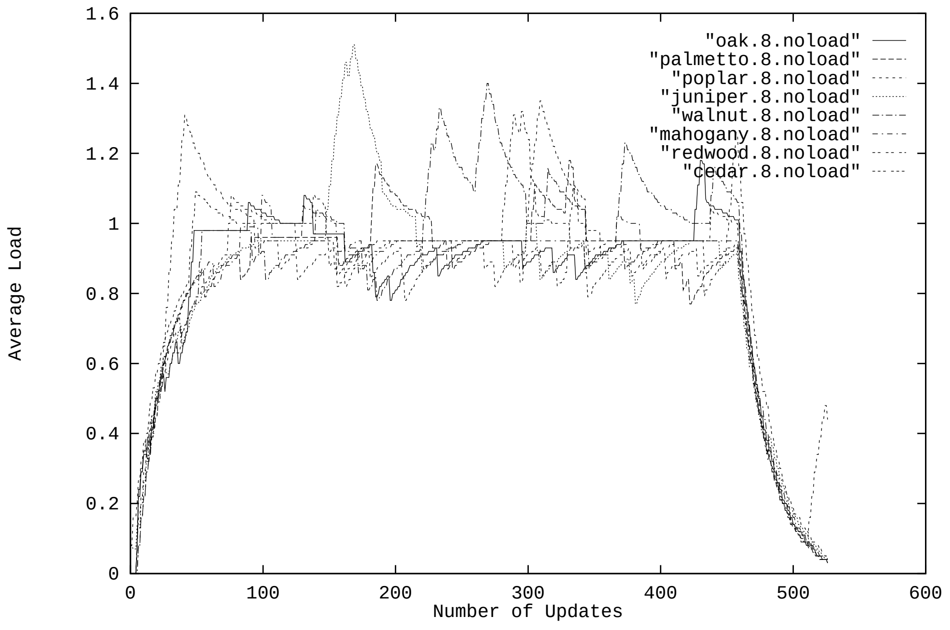

Figure 20. Average load, unloaded – Warshall.

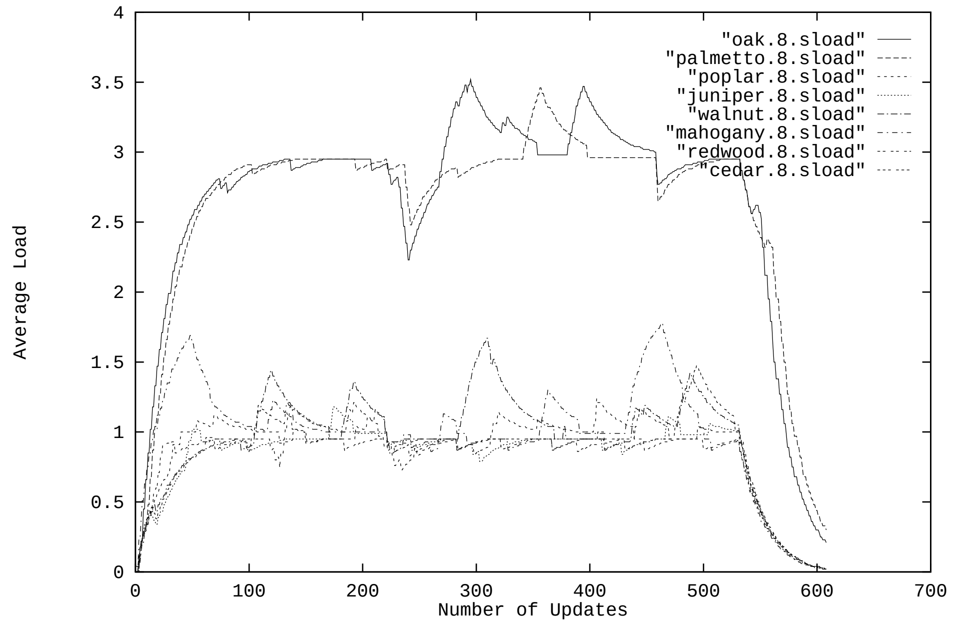

Figure 21. Average load, overloaded – Warshall.

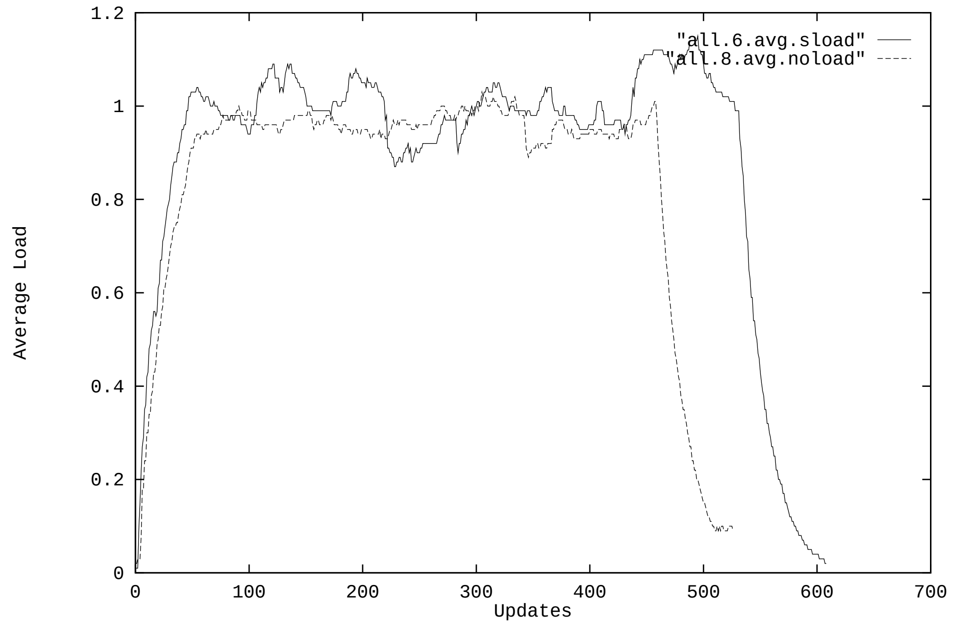

Figure 22. Comparison unloaded/overloaded – Warshall.

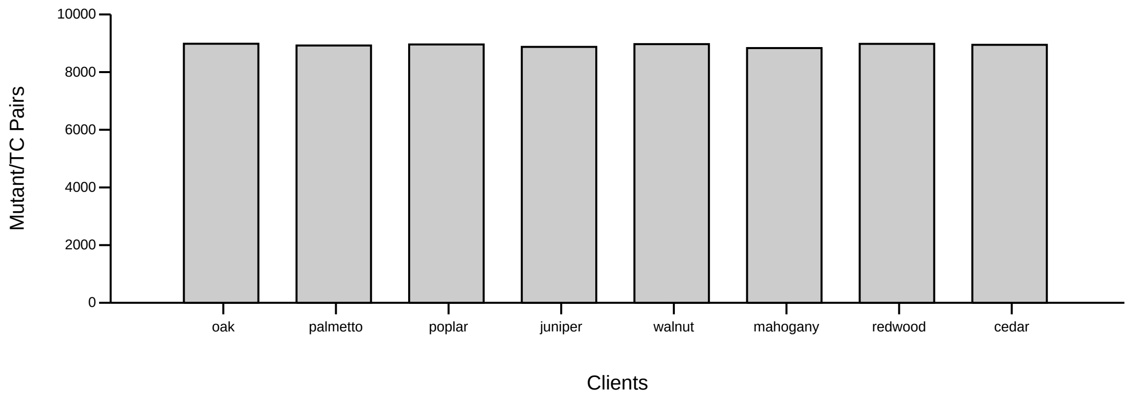

Figure 23. (Mutant/Test Case) distribution, unloaded – Warshall.

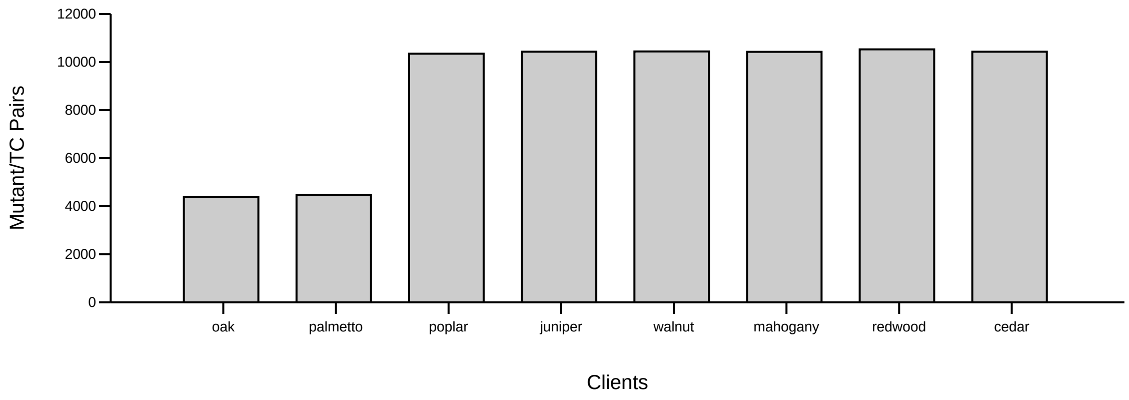

Figure 24. (Mutant/Test Case) distribution, overloaded – Warshall.

CHAPTER VIII
CONCLUSIONS AND FUTURE WORK

We have shown that the time spent on unit software testing can be significantly reduced by distributing the work over several clients. MedusaMothra achieves a speedup of 14.35 for DEADLK and 13.50 for WARSHALL when sixteen clients were used. These numbers are very close to linear speedup. As a result, the mutant execution time was reduced from hours to minutes when comparing the mutant execution time of the original Mothra to the mutant execution time of MedusaMothra. MedusaMothra can theoretically be expanded to handle 256 clients, which makes it even more applicable for larger test programs. For computationally inexpensive programs, the number of clients for MedusaMothra should be chosen carefully, because the communication overhead can actually increase the wall execution time. For example, the QUAD program had better performance results with eight clients than with 16 clients. In general, good speedup can be obtained when multiple clients are used.

In the future, several additions can improve MedusaMothra. Currently, MedusaMothra does not support heterogeneous architectures. Since distributed computing over heterogeneous architectures is one of the more active research areas at the moment, an attempt was made to keep MedusaMothra as transparent as possible with respect to the addition of other architectures. The data which is sent between the server and the clients could be encoded and decoded using External Data Representation (XDR) [25] routines. The use of XDR routines allows programmers to describe arbitrary data structures in a machine-independent fashion.

A further step would be the use of a tool such as Parallel Virtual Machine (PVM) for the communication between the server and the clients. PVM is a software system that enables a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational resource. It transparently handles message routing, data conversion for incompatible architectures, and other tasks that are necessary for operation in a heterogeneous network environment [26]. During the initial design phase of MedusaMothra, we considered using PVM but opted to implement MedusaMothra with Unix sockets using TCP/IP, mainly because PVM was not then considered a standard.

The only problem besides porting either the MedusaMothra server or client to a different architecture would be file access. MedusaMothra currently provides access to data files via NFS, which allows transparent file access for the server and all the clients. If MedusaMothra were distributed over a network of heterogeneous architectures, the necessary data files would have to be distributed to all the target architectures before starting the mutant execution.

Since the natural load balancing scheme performs well for MedusaMothra, it would be interesting to see the performance results in terms of speedup for a more complex dynamic load balancing scheme. Additionally, the possibility of executing more than one MedusaMothra client on a particular workstation introduces some interesting load balancing problems such as moving a MedusaMothra client to another workstation. Although we mentioned in Chapter V that SunOS does not support process migration, it should be possible to migrate a mutant during execution from one client to another, because mutants are interpreted and not compiled. One major drawback for "mutant migration" would be the possibly large amount of state information that had to be kept in order to migrate a mutant from one client to another during mutant execution.

Other important issues for long-running distributed applications are checkpointing and restart facilities. Both facilities are not fully implemented in MedusaMothra, but we did introduce the necessary status data structures into MedusaMothra to keep track of which client executes which (mutant, test case) pair in the server. Adding a timeout logic in the server for the case of a client failure would ensure that no data is lost. Synchronizing the status data structures with the mutant descriptor file would make it possible to implement an effective check pointing and restart mechanism.

Several other attempts have been made to speed up the mutation testing process. Weak mutation testing [24] and selective mutation testing [5] have shown very promising results in terms of mutant execution time reduction and adequacy. Combining any of these mutation techniques with MedusaMothra would certainly result in even smaller execution times and therefore making structured testing methods such as mutation testing more cost effective and usable.

APPENDICES

Appendix A

MedusaMothra Startup Shell Script

```
#! /bin/csh
# script to start up medusa environment with $1 clients
# clients are specified in the CLIENTS file
# this script must be run on the server
# assumes all mutants were created, i.e. mutmask = 3fffff
# IMPORTANT:
# MEDUSA_PATH defines the path of the server and client binaries.
# Assumes that the SERVER and CLIENTS file are in the directory of the
# experiment

set MEDUSA_PATH = /u2/czapf/research/test

# check arguments
if ($#argv != 2) then
    echo ""
    echo "Usage: stm <Number of Clients> <Experiment Name>"
    echo ""
    exit
endif

# clean up
/bin/rm -f *.out >& /dev/null

# start medusa server
echo "Starting Medusa Server"
echo "medusa -m3fffff -g "$PWD"/"$2
$MEDUSA_PATH/medusa -m3fffff -g $PWD/$2 >& server.out &

# make sure server is up and running
sleep 2

# start clients
set clients = `nawk '{ if (NR > 1) print $1 }' CLIENTS`

@ counter = 0
foreach s ($clients)
    echo starting client on $s
    echo "rsh "$s "$MEDUSA_PATH/medusa_client $PWD &"
    rsh $s "$MEDUSA_PATH/medusa_client $PWD >& $PWD/$s.out"  &
    @ counter++
    if ($counter >= $1) break;
end
```

Appendix B

Experimental Fortran 77 Program Listings

BUB

```
C-------------------------------------------------------------------------
C---  BUB Source Code
C-------------------------------------------------------------------------

      SUBROUTINE BUBBLE (A)
      INTEGER A (5)

C     Sort A() using the bubble sort technique.

      INTEGER N, ITMP

      N = 5
      DO 200 J = N-1, 1, -1
         DO 100 I = 1, J, 1
            IF (A (I).LE.A (I+1)) GOTO 100
               ITMP = A (I)
               A (I) = A (I+1)
               A (I+1) = ITMP
100   CONTINUE
200   CONTINUE
      RETURN
      END
```

DEADLK

```
C-----------------------------------------------------------------------------
C---  DEADLK Source Code
C-----------------------------------------------------------------------------

      SUBROUTINE DEADLK(PCNT,RCNT,ALLOC,AVAIL,RQSTED,DLOCKV,STUCK)

C This subroutine checks to see if deadlock has occurred.  It attempts
C to simulate the completion of all processes not currently blocked,
C then see if all blocked processes can complete.  If not, we are
C deadlocked, and the routine returns a list of deadlocked processes.

C Constants
C ------------

      PARAMETER ( MAXP=10, MAXR=10 )

C MAXP - maximum number of processes that may be specified
C MAXR - maximum number of resources that may be specified

C Parameters
C ----------

      INTEGER PCNT, RCNT, AVAIL(0:MAXP-1), RQSTED(0:MAXP-1,0:MAXR-1)
      INTEGER ALLOC(0:MAXP-1,0:MAXR-1), DLOCKV(0:MAXR-1)
      LOGICAL STUCK

C PCNT   - number of user processes in use
C RCNT   - number of resource classes in the system
C AVAIL  - array; for each resource class, number still available
C ALLOC  - matrix; for each process, number of resources of each class
C          currently allocated to it.  ALLOC + CLAIMS = max total claim.
C RQSTED - matrix; for each process, number of resources it has requested
C          in each class, but not received.
C          Used only for blocked processes.
C DLOCKV - list of deadlocked processes, if any
C STUCK  - true if we are deadlocked, false otherwise

C Local variables
C ---------------

      INTEGER LAVAIL(0:MAXP-1), LALLOC(0:MAXP-1,0:MAXR-1)
      INTEGER LRQSTD(0:MAXP-1,0:MAXR-1), I, J, DLVN
      LOGICAL CHANGE

C LALLOC - Local copy of the ALLOC matrix; for each process, the number
C          of resources of each class currently allocated to it.
C LAVAIL - Local copy of the AVAIL vector; for each resource class, the
C          number of such resources still available.
C LRQSTD - Local copy of the RQSTED matrix; for each blocked process,
C          the number of resources of each class it still wants
C DLVN   - counts entries in the DLOCKV
C I,J,K  - Loop indexes
C CHANGE - Lets us traverse the claims list until changes occur
```

```
C---------------------------------------------------------------
C Begin code

C Make local copies of the available vector, and the allocated and
C requested matrices.

       DO 20 I = 0, RCNT-1
          DO 10 J = 0, PCNT-1
             LALLOC(I,J) = ALLOC(I,J)
             LRQSTD(I,J) = RQSTED(I,J)
10        CONTINUE
          LAVAIL(I) = AVAIL(I)
20     CONTINUE

C Repeat the following loop until no changes occur (no unblocked
C processes are released).  Set change false initially.  The occurence
C of a change sets it true.  The loop is post-tested and returns to 30
C if change is true.

30     CHANGE = .FALSE.

C Simulate completion of all nonblocked processes having allocated
C resources.  For each process...

       DO 200 I = 0, PCNT-1

C If it has requested resources it is blocked so start on next process.

          DO 50 J = 0, RCNT-1
             IF (LRQSTD(I,J) .GT. 0) GOTO 200
50        CONTINUE

C It has no requested resources, release any it has.

          DO 70 J = 0, RCNT-1
             LAVAIL(J) = LAVAIL(J) + LALLOC(I,J)
             LALLOC(I,J) = 0
70        CONTINUE

200    CONTINUE

C Now, see if we can unblock any blocked processes.  If so, a change
C has occurred.  For each process...

       DO 300 I = 0, PCNT-1

C If it has a request current, it is blocked.  If we check all without
C finding one, goto consider the next process.

          DO 250 J = 0, RCNT-1
             IF (LRQSTD(I,J) .GT. 0) GOTO 270
250       CONTINUE
          GOTO 300
```

```
C The Ith process is blocked.  See if we can unblock it.  Check all
C resources it has requested.  If we find a request that exceeds the
C number available goto consider the next process.

270       DO 280 J = 0, RCNT-1
             IF (LRQSTD(I,J) .GT. LAVAIL(J)) GOTO 300
280       CONTINUE

C Getting here, we know we can unblock process I.  Do so, by simulating
C it getting its requested resources.  Its completion will be simulated
C on the next pass through the main loop.

          CHANGE = .TRUE.
          DO 290 J = 0, RCNT-1
             LALLOC(I,J) = LALLOC(I,J) + LRQSTD(I,J)
             LRQSTD(I,J) = 0
290       CONTINUE

300    CONTINUE

C Return to 30 if changes occurred, to repeat the loop.

       IF (CHANGE .EQ. .TRUE.) GOTO 30

C Now if any processes with outstanding requests remain, we are deadlocked.
C As we search for them we put their IDs into DLOCKV to pass back.
C First initialize DLOCKV to null by placing -1 in all its locations.

400    DO 450 I = 0, PCNT-1
          DLOCKV(I) = -1
450    CONTINUE
       DLVN = 0
       DO 500 I = 0, PCNT-1
          DO 480 J = 0, RCNT-1
             IF (LRQSTD(I,J) .GT. 0) THEN
                DLOCKV(DLVN) = I
                DLVN = DLVN + 1
                GOTO 500
             ENDIF
480       CONTINUE
500    CONTINUE

C If DLVN .gt. 0 we placed a process in the dlockv, so we are stuck.
C Else we are not.

       IF (DLVN .GT. 0) THEN
          STUCK = .TRUE.
       ELSE
          STUCK = .FALSE.
       ENDIF

       RETURN
       END
```

EUCLID

```
C----------------------------------------------------------------------------
C---  EUCLID Source Code
C----------------------------------------------------------------------------

      INTEGER FUNCTION Euclid (A, B)
C
C     Euclid's GCD algorithm.
C= A in
C= B in
C
      INTEGER A, B
      INTEGER Div, Rem
      ASSERT (A .GT. 0 .AND. B .GT. 0)

      Rem = 1
C     WHILE (Rem .GT. 0) DO
10    CONTINUE
      IF (Rem .LE. 0) GOTO 20
         Div = A/B
         Rem = A - Div*B
         A = B
         B = Rem
C     ENDWHILE
      GOTO 10
20    CONTINUE

      Euclid = A
      END
```

FIND

```
C-----------------------------------------------------------------------------
C---  FIND Source Code
C-----------------------------------------------------------------------------

       SUBROUTINE FIND (A, N, F)
       INTEGER A (10), N, F

C      F is index into A().  After execution, all elements to the left of
C      A(F) are less than or equal to A(F) and all elements to the right of
C      A(F) are greater than or equal to A(F).
C      Only the first N elements are considered.
C      From DeMillo, Lipton, and Sayward [DeMi78], repeated from Hoare's
C      paper [Hoar70].

       INTEGER M, NS, R, I, J, W

       ASSERT (F.GE.1.AND.F.LE.N.AND.N.GE.1.AND.N.LE.10)
       M = 1
       NS = N
10     IF (M.GE.NS) GOTO 1000
       R = A (F)
       I = M
       J = NS
20     IF (I.GT.J) GOTO 60
30     IF (A(I).GE.R) GOTO 40
       I = I + 1
       GOTO 30
40     IF (R.GE.A(J)) GOTO 50
       J = J - 1
       GOTO 40
50     IF (I.GT.J) GOTO 20

       W = A (I)
       A (I) = A (J)
       A (J) = W
       I = I + 1
       J = J - 1
       GOTO 20

60     IF (F.GT.J) GOTO 70
       NS = J
       GOTO 10
70     IF (I.GT.F) GOTO 1000
       M = I
       GOTO 10
1000   RETURN
       END
```

INSERT

```
C---------------------------------------------------------------------------
C---  INSERT Source Code
C---------------------------------------------------------------------------

        SUBROUTINE INSERT (L,N)
INTEGER  L(N),N
INTEGER  KEY,I,J
C=  L inout
C=  N in

      ASSERT (N.EQ.10)
J=2
1       IF (J.GT.N) GOTO 99
KEY=L(J)
I=J-1
5 IF (I.LE.0) GOTO 15
   IF (L(I).LE.KEY) GOTO 15
      L(I+1) = L(I)
      I=I-1
GOTO 5
15 L(I+1) = KEY
J=J+1
GOTO 1
99 RETURN
END
```

PAT

```
C---------------------------------------------------------------------------
C---  PAT Source Code
C---------------------------------------------------------------------------

       SUBROUTINE PAT (P, S, CODE)
       INTEGER P(9), S(3), CODE, I, J, K
       I=0
2      CODE=0
       J=0
3      K=I+J
       IF (P(K).EQ.S(J+1)) GOTO 1
       I=I+1
       IF (I.GT.7) RETURN
       GOTO 2
1      J=J+1
       IF (J.GT.2) GOTO 4
       GOTO 3
4      CODE=1
       RETURN
       END
```

QUAD

```
C---------------------------------------------------------------------------
C---  QUAD Source Code
C---------------------------------------------------------------------------

C
C     JEFF OFFUTT
C     12-02-89
C
C     This program accepts three constants CoeffA, CoeffB, and CoeffC
C     that represent a quadratic equation.  The roots are computed and
C     printed if they exist.
C
      SUBROUTINE ROOTS (CoeffA, CoeffB, CoeffC, Root1, Root2)
C     INPUT VARIABLES ...
      REAL CoeffA, CoeffB, CoeffC
C     OUTPUT VARIABLES ...
      REAL Root1, Root2
C     Internal VARIABLES ...
      REAL Disc
      ASSERT (CoeffA .NE. 0.0)

      Disc = CoeffB*CoeffB - (4*CoeffA*CoeffC)
      IF (DISC .GE. 0.0) THEN
         Root1 = ((-CoeffB) + SQRT(Disc)) / (2*CoeffA)
         Root2 = ((-CoeffB) - SQRT(Disc)) / (2*CoeffA)
      ELSE
         Root1 = 0.0
         Root2 = 0.0
      ENDIF
      STOP
      END
```

SEARCH

```
C-----------------------------------------------------------------------------
C---  SEARCH Source Code
C-----------------------------------------------------------------------------

       INTEGER FUNCTION SEARCH(A,E)

C Search array A for element E, return the index to E or the index
C immediately preceding where E should be placed.  Uses Binary Search.

       INTEGER A(8), E, J, K, I

       IF (E .LT. A(1)) THEN
         SEARCH = 0
       ELSE
         I = 1
         J = 8
10       IF (I .LT. J) THEN
            K = (I + J + 1) / 2
            IF (E .LT. A(K)) THEN
               J = K - 1
            ELSE
               I = K
            ENDIF
            GOTO 10
         ENDIF

         SEARCH = I

       ENDIF
       RETURN
       END
```

TRITYP

```
C---------------------------------------------------------------------------
C---  TRITYP Source Code
C---------------------------------------------------------------------------

       INTEGER FUNCTION TRIANG(I,J,K)
       INTEGER I,J,K

C      MATCH IS OUTPUT FROM THE ROUTINE:
C         TRIANG = 1 IF TRIANGLE IS SCALENE
C         TRIANG = 2 IF TRIANGLE IS ISOSCELES
C         TRIANG = 3 IF TRIANGLE IS EQUILATERAL
C         TRIANG = 4 IF NOT A TRIANGLE

C      After a quick confirmation that it's a legal
C      triangle, detect any sides of equal length

       IF (I.LE.0.OR.J.LE.0.OR.K.LE.0) THEN
          TRIANG=4
          RETURN
       ENDIF
       TRIANG=0
       IF (I.EQ.J) TRIANG=TRIANG+1
       IF (I.EQ.K) TRIANG=TRIANG+2
       IF (J.EQ.K) TRIANG=TRIANG+3
       IF (TRIANG.EQ.0) THEN

C      Confirm it's a legal triangle before declaring
C      it to be scalene

          IF (I+J.LE.K.OR.J+K.LE.I.OR.I+K.LE.J) THEN
     TRIANG = 4
  ELSE
             TRIANG = 1
          ENDIF
          RETURN
       ENDIF

C      Confirm it's a legal triangle before declaring
C      it to be isosceles or equilateral

       IF (TRIANG.GT.3) THEN
          TRIANG = 3
       ELSE IF (TRIANG.EQ.1.AND.I+J.GT.K) THEN
  TRIANG = 2
       ELSE IF (TRIANG.EQ.2.AND.I+K.GT.J) THEN
  TRIANG = 2
       ELSE IF (TRIANG.EQ.3.AND.J+K.GT.I) THEN
  TRIANG = 2
       ELSE
  TRIANG = 4
       ENDIF

       END
```

```
C---------------------------------------------------------------------------
C---  WARSHALL Source Code
C---------------------------------------------------------------------------

      SUBROUTINE WARSHALL (A)
      INTEGER A (5,5)
C= A inout

      ASSERT (A.LE.1 .AND. A.GE.0)

C     Calculate the transitive closure of A using Warshall's algorithm.

      INTEGER I, J, K

  DO 100 K = 1, 5
DO 200 I = 1, 5
  DO 300 J = 1, 5
IF (A(I, J) .EQ. 0) THEN
  A(I, J) = A(I, K) * A(K, J)
            ENDIF
300   CONTINUE
200   CONTINUE
100   CONTINUE
      RETURN
      END
```

Appendix C

Experimental Data

BUB

```
Experiment Summary: /home/students/czapf/research/test/BUB
----------------------------------------------------------
Number of Clients: 1
Running Time: 2386

Experiment Summary: /home/students/czapf/research/test/BUB
----------------------------------------------------------
Number of Clients: 4
Running Time:        796 seconds

Client   (M/TC) Pairs
---------------------
banyan     20032
oak        19197
poplar     20632
dogwood    20033

Experiment Summary: /home/students/czapf/research/test/BUB
----------------------------------------------------------
Number of Clients: 8
Running Time:        463 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
-----------------------------------------------
oak        10254         hedge       9860
palmetto    9795         poplar      9941
juniper    10093         walnut     10084
mahogany   10009         redwood     9855

Experiment Summary: /home/students/czapf/research/test/BUB
----------------------------------------------------------
Number of Clients: 16
Running Time:        260 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
-----------------------------------------------
oak         5051         hedge        4917
palmetto    4712         poplar       5341
juniper     5087         walnut       5311
mahogany    5204         redwood      4968
cedar       4800         cypress      4960
bamboo      5056         magnolia     4786
sequoia     4909         banyan       4896
dogwood     5030         aspen        4852
```

DEADLK

```
Experiment Summary: /home/students/czapf/research/test/D
--------------------------------------------------------
Number of Clients: 1
Running Time: 6629

Experiment Summary: /home/students/czapf/research/test/D
--------------------------------------------------------
Number of Clients: 2
Running Time:       3299 seconds

Client   (M/TC) Pairs
---------------------
banyan     13748
oak        13882

Experiment Summary: /home/students/czapf/research/test/D
--------------------------------------------------------
Number of Clients: 4
Running Time:       1723 seconds

Client   (M/TC) Pairs
---------------------
banyan      6975
oak         6683
poplar      7527
dogwood     6447

Experiment Summary: /home/students/czapf/research/test/D
--------------------------------------------------------
Number of Clients: 8
Running Time:        882 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
-----------------------------------------------
oak         3539         hedge       3178
palmetto    3520         poplar      3434
juniper     3826         walnut      3388
mahogany    3606         redwood     3478

Experiment Summary: /home/students/czapf/research/test/D
--------------------------------------------------------
Number of Clients: 16
Running Time:        462 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
-----------------------------------------------
oak         1918         hedge       1518
palmetto    1812         poplar      1855
juniper     1923         walnut      1627
mahogany    1655         redwood     1634
cedar       1625         cypress     1858
bamboo      1816         magnolia    1649
sequoia     1565         banyan      1783
dogwood     1806         aspen       1808
```

EUCLID

```
Experiment Summary: /home/students/czapf/research/test/E
--------------------------------------------------------
Number of Clients: 1
Running Time: 196

Experiment Summary: /home/students/czapf/research/test/E
--------------------------------------------------------
Number of Clients: 2
Running Time:        116 seconds

Client   (M/TC) Pairs
---------------------
banyan      2011
oak         2026

Experiment Summary: /home/students/czapf/research/test/E
--------------------------------------------------------
Number of Clients: 4
Running Time:         58 seconds

Client   (M/TC) Pairs
---------------------
banyan      1018
oak         1011
poplar       999
dogwood     1009

Experiment Summary: /home/students/czapf/research/test/E
--------------------------------------------------------
Number of Clients: 8
Running Time:         37 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
-----------------------------------------------
banyan       557         oak          538
poplar       551         dogwood      507
juniper      469         redwood      482
maple        497         hedge        436

Experiment Summary: /home/students/czapf/research/test/E
--------------------------------------------------------
Number of Clients: 16
Running Time:         29 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
-----------------------------------------------
oak          282         hedge        308
palmetto     282         poplar       272
juniper      275         walnut       292
mahogany     225         redwood      254
cedar        242         cypress      260
bamboo       158         magnolia     247
sequoia      232         banyan       194
dogwood      266         aspen        249
```

FIND

```
Experiment Summary: /home/students/czapf/research/test/F
--------------------------------------------------------
Number of Clients: 1
Running Time: 4179

Experiment Summary: /home/students/czapf/research/test/F
--------------------------------------------------------
Number of Clients: 2
Running Time:       2830 seconds

Client   (M/TC) Pairs
---------------------
banyan     69419
oak        69265

Experiment Summary: /home/students/czapf/research/test/F
--------------------------------------------------------
Number of Clients: 4
Running Time:       1598 seconds

Client   (M/TC) Pairs
---------------------
banyan     35800
oak        36677
poplar     33827
dogwood    32338

Experiment Summary: /home/students/czapf/research/test/F
--------------------------------------------------------
Number of Clients: 8
Running Time:        848 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
-----------------------------------------------
oak        17798         hedge      16610
palmetto   17356         poplar     17543
juniper    17597         walnut     17267
mahogany   17443         redwood    17045

Experiment Summary: /home/students/czapf/research/test/F
--------------------------------------------------------
Number of Clients: 16
Running Time:        478 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
-----------------------------------------------
oak         8721         hedge       9169
palmetto    8191         poplar      8479
juniper     8571         walnut      8187
mahogany    8581         redwood     8390
cedar       8306         cypress     9062
bamboo      8460         magnolia    8992
sequoia     8807         banyan      9022
dogwood     8965         aspen       8762
```

INSERT

```
Experiment Summary: /home/students/czapf/research/test/IN
----------------------------------------------------------
Number of Clients: 1
Running Time: 718

Experiment Summary: /home/students/czapf/research/test/IN
----------------------------------------------------------
Number of Clients: 2
Running Time:        376 seconds

Client   (M/TC) Pairs
---------------------
banyan      4646
oak         4676

Experiment Summary: /home/students/czapf/research/test/IN
----------------------------------------------------------
Number of Clients: 4
Running Time:        200 seconds

Client   (M/TC) Pairs
---------------------
banyan      2271
oak         2350
poplar      2364
dogwood     2337

Experiment Summary: /home/students/czapf/research/test/IN
----------------------------------------------------------
Number of Clients: 8
Running Time:        106 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
-----------------------------------------------
oak         1206         hedge       1154
poplar      1172         juniper     1161
walnut      1191         mahogany    1157
redwood     1172         palmetto    1109

Experiment Summary: /home/students/czapf/research/test/IN
----------------------------------------------------------
Number of Clients: 16
Running Time:         63 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
-----------------------------------------------
oak          650         hedge        623
palmetto     506         poplar       613
juniper      621         walnut       603
mahogany     550         redwood      601
cedar        583         cypress      575
bamboo       511         magnolia     599
sequoia      587         banyan       578
dogwood      583         aspen        539
```

PAT

```
Experiment Summary: /home/students/czapf/research/test/PAT
----------------------------------------------------------
Number of Clients: 1
Running Time: 2003

Experiment Summary: /home/students/czapf/research/test/PAT
----------------------------------------------------------
Number of Clients: 2
Running Time:       1087 seconds

Client   (M/TC) Pairs
---------------------
banyan     20688
oak        21027

Experiment Summary: /home/students/czapf/research/test/PAT
----------------------------------------------------------
Number of Clients: 4
Running Time:        579 seconds

Client   (M/TC) Pairs
---------------------
banyan     10243
oak        10472
dogwood    10447
juniper    10553

Experiment Summary: /home/students/czapf/research/test/PAT
----------------------------------------------------------
Number of Clients: 8
Running Time:        297 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
-----------------------------------------------
oak         5282         hedge       5206
poplar      5255         juniper     5258
walnut      5226         mahogany    5273
redwood     5183         palmetto    5032

Experiment Summary: /home/students/czapf/research/test/PAT
----------------------------------------------------------
Number of Clients: 16
Running Time:        180 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
-----------------------------------------------
oak         2737         hedge       2712
palmetto    2467         poplar      2664
juniper     2740         walnut      2672
mahogany    2709         redwood     2516
cedar       2503         cypress     2654
bamboo      2567         magnolia    2426
sequoia     2615         banyan      2614
dogwood     2532         aspen       2587
```

QUAD

```
Experiment Summary: /home/students/czapf/research/test/Q
--------------------------------------------------------
Number of Clients: 1
Running Time: 29

Experiment Summary: /home/students/czapf/research/test/Q
--------------------------------------------------------
Number of Clients: 2
Running Time:         21 seconds

Client   (M/TC) Pairs
---------------------
banyan       364
oak          363

Experiment Summary: /home/students/czapf/research/test/Q
--------------------------------------------------------
Number of Clients: 4
Running Time:         12 seconds

Client   (M/TC) Pairs
---------------------
banyan       181
oak          181
poplar       191
dogwood      174

Experiment Summary: /home/students/czapf/research/test/Q
--------------------------------------------------------
Number of Clients: 8
Running Time:          8 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
---------------------------------------------
oak          113         hedge          87
palmetto      84         poplar         95
juniper       78         walnut         91
mahogany      91         redwood        88

Experiment Summary: /home/students/czapf/research/test/Q
--------------------------------------------------------
Number of Clients: 13
Running Time:         12 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
---------------------------------------------
banyan       116         oak           137
poplar        96         dogwood        23
juniper       43         redwood        66
maple          1         hedge          88
walnut        81         escher          0
birch          8         newton          6
cypress        0
```

SEARCH

```
Experiment Summary: /home/students/czapf/research/test/S
--------------------------------------------------------
Number of Clients: 1
Running Time: 359

Experiment Summary: /home/students/czapf/research/test/S
--------------------------------------------------------
Number of Clients: 2
Running Time:        180 seconds

Client   (M/TC) Pairs
---------------------
banyan      3342
oak         3269

Experiment Summary: /home/students/czapf/research/test/S
--------------------------------------------------------
Number of Clients: 4
Running Time:         99 seconds

Client   (M/TC) Pairs
---------------------
banyan      1677
oak         1612
poplar      1686
dogwood     1636

Experiment Summary: /home/students/czapf/research/test/S
--------------------------------------------------------
Number of Clients: 8
Running Time:         59 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
-----------------------------------------------
oak          849         hedge        868
palmetto     811         poplar       827
juniper      776         walnut       818
mahogany     832         redwood      856

Experiment Summary: /home/students/czapf/research/test/S
--------------------------------------------------------
Number of Clients: 16
Running Time:         44 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
-----------------------------------------------
oak          428         hedge        430
poplar       353         juniper      445
walnut       413         mahogany     443
redwood      414         cedar        461
cypress      455         bamboo       490
magnolia     400         sequoia      363
banyan       400         dogwood      471
aspen        333         palmetto     356
```

TRITYP

```
Experiment Summary: /home/students/czapf/research/test/T
--------------------------------------------------------
Number of Clients: 1
Running Time: 2748

Experiment Summary: /home/students/czapf/research/test/T
--------------------------------------------------------
Number of Clients: 2
Running Time:       2061 seconds

Client   (M/TC) Pairs
---------------------
banyan     39780
oak        39491

Experiment Summary: /home/students/czapf/research/test/T
--------------------------------------------------------
Number of Clients: 4
Running Time:       1056 seconds

Client   (M/TC) Pairs
---------------------
oak        19885
hedge      19894
palmetto   19785
poplar     19707

Experiment Summary: /home/students/czapf/research/test/T
--------------------------------------------------------
Number of Clients: 8
Running Time:        562 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
-----------------------------------------------
oak         9886         dogwood     9889
juniper     9923         redwood     9839
hedge       9970         walnut      9994
cypress     9759         palmetto    9967

Experiment Summary: /home/students/czapf/research/test/T
--------------------------------------------------------
Number of Clients: 16
Running Time:        347 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
-----------------------------------------------
banyan      5269         oak         5102
palmetto    5089         poplar      5279
dogwood     4943         juniper     5017
teak        5354         walnut      5047
aspen       4725         mahogany    5036
redwood     5347         spruce      4895
cedar       5292         cypress     2469
maple       4850         willow      5134
```

WARSHALL

```
Experiment Summary: /home/students/czapf/research/test/W
--------------------------------------------------------
Number of Clients: 1
Running Time: 8047

Experiment Summary: /home/students/czapf/research/test/W
--------------------------------------------------------
Number of Clients: 2
Running Time:       4544 seconds

Client   (M/TC) Pairs
---------------------
banyan     35478
oak        35988

Experiment Summary: /home/students/czapf/research/test/W
--------------------------------------------------------
Number of Clients: 4
Running Time:       2273 seconds

Client   (M/TC) Pairs
---------------------
banyan     17905
oak        17939
poplar     17819
dogwood    17803

Experiment Summary: /home/students/czapf/research/test/W
--------------------------------------------------------
Number of Clients: 8
Running Time:       1180 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
-----------------------------------------------
oak         8983        palmetto    8923
poplar      8958        juniper     8874
walnut      8970        mahogany    8833
redwood     8979        cedar       8946

Experiment Summary: /home/students/czapf/research/test/W
--------------------------------------------------------
Number of Clients: 16
Running Time:        596 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
-----------------------------------------------
oak         4487        hedge       4490
palmetto    4517        poplar      4498
juniper     4500        walnut      4539
mahogany    4431        redwood     4455
cedar       4464        cypress     4393
bamboo      4411        magnolia    4410
sequoia     4470        banyan      4449
dogwood     4481        aspen       4471
```

Appendix D

Utilities for Dynamic Load Balancing Studies

CPU-Bound Program

```
/*---------------------------------------------------------------------*/
/* crunch.c                                                            */
/*---------------------------------------------------------------------*/

/* This program is your basic no I/O number cruncher */
/* Written by M. Westall for CpSc 824 */

#include <math.h>

main()
{
   long start;
   long end;
   double x;
   int i, j;

   start = time(0);
   for (i = 0; i < 60; i++)
      for (j = 0; j < 750000; j++)
         x += j * i / 0.3;

   end = time(0);

   printf("%d %d %d \n", start, end, end-start);
}
```

Update Shell Script

```
#! /bin/csh
# script to keep track of load averages on $1 clients
# clients are specified in the CLIENTS file

if ($#argv != 2) then
    echo ""
    echo "Usage: ruptimer <Number of Clients> <# updates>"
    echo ""
    exit
endif

/bin/rm -f *.time

set clients = `nawk '{ if (NR > 1) print $1 }' CLIENTS`

@ counter = 0
foreach s ($clients)
    touch $s.time
    @ counter++
    if ($counter >= $1) break;
end

@ timevar = 0
while (1)
    @ counter = 0
    foreach s ($clients)
        echo -n $timevar >> $s.time
        rup $s | nawk '{ printf " %2.2f\n",$8 }' >> $s.time
        @ counter++
        if ($counter >= $1) break;
    end
    @ timevar++
    if ($timevar > $2) break;
end
```

Appendix E

Experimental Data for Dynamic Load Balancing Studies

TRITYP

```
Experiment Summary: /home/students/czapf/research/test/T
--------------------------------------------------------
Number of Clients: 8
Running Time:        610 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
-----------------------------------------------
oak         9831        palmetto   10056
poplar      9651        juniper     9882
walnut     10057        mahogany    9633
redwood    10048        cedar      10016
```

WARSHALL

```
Experiment Summary: /home/students/czapf/research/test/W
--------------------------------------------------------
Number of Clients: 8
Running Time:       1367 seconds

Client   (M/TC) Pairs    Client    (M/TC) Pairs
-----------------------------------------------
oak         4385        palmetto    4475
poplar     10348        juniper    10432
walnut     10442        mahogany   10425
redwood    10528        cedar      10431
```

LITERATURE CITED

1. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

2. R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Banff Alberta, July 1988. IEEE Computer Society Press.

3. K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software–Practice and Experience*, 21(7):685–718, July 1991.

4. A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3–18, January 1992.

5. A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *Proceedings of the 15th International Conference on Software Engineering*, pages 100–108, Baltimore MD, May 1993. IEEE Computer Society Press.

6. B. J. Choi, A. P. Mathur, R. A. DeMillo, E. W. Krauser, R. J. Martin, A. J. Offutt, and E. H. Spafford. The Mothra tool set. In *Proceedings of the 22nd Hawaii International Conference on Syst em Sciences*, pages 275–284, Kailua-Kona HI, January 1989.

7. R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

8. A. P. Mathur and E. W. Krauser. Modeling mutation on a vector processor. In *Proceedings of the 10th International Conference on Software En gineering*, pages 154–161, Singapore, April 1988. IEEE Computer Society.

9. A. P. Mathur and E. W. Krauser. Mutant unification for improved vectorization. Technical report SERC-TR-14-P, Software Engineering Research Center, Purdue University, West Lafayette IN, April 1988.

10. E. W. Krauser, A. P. Mathur, and V. J. Rego. High performance software testing on SIMD machines. *IEEE Transactions on Software Engineering*, 17(5):403–23, May 1991.

11. B. Choi and A. P. Mathur. Use of fifth generation computers for high performance reliable software testing. Technical report SERC-TR-72-P, Software Engineering Research Center, Purdue University, West Lafayette IN, April 1990.

12. B. Choi, A. P. Mathur, and B. Pattison. PMothra: Scheduling Mutants for Execution on a Hypercube. *ACM SIGSOFT Software Engineering Notes*, 14(8):58–65, December 1989. In *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, Key West FL, December 13–15.

13. A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar. Mutation Testing of Software Using a MIMD Computer. In *Proceedings of the 1992 International Conference on Parallel Processing*, pages II–257–266, St. Charles IL, August 17–21 1992.

14. P. Khambekar. *Dynamic Load Balancing on Distributed Memory Computer Systems.* PhD thesis, Clemson University, Clemson SC, 1992.

15. S. Fichter. Hypermothra – a parallel interpreter for the mothra mutation testing system. Master's thesis, Department of Computer Science, Clemson University, Clemson SC, 1991.

16. T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, SE-14(2):141–154, February 1988.

17. A. Goscinski. *Distributed Operation Systems – The Logical Design*. Addison-Wesley Publishing Company, Reading MA, 1991.

18. Y. Wang and R. J. T. Morris. Load sharing in distributed systems. *IEEE Transactions on Computers*, C-34(3):204–217, March 1985.

19. D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver–initiated and sender–initiated adaptive load sharing. *Performance Evaluation*, 6(1):53–68, 1986.

20. K. Ramamritham, J. A. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Transactions on Computers*, C-38(8):1110–1123, 1989.

21. I. S. Fogg. Distributed scheduling algorithms: A survey. Technical report, Department of Computer Science, University of Queensland, St. Lucia, Australia, 1990.

22. A. S. Tanenbaum. *Computer Networks*. Prentice Hall, Inc., Englewood Cliffs NJ, second edition, 1988.

23. G. Myers. *The Art of Software Testing*. John Wiley and Sons, New York NY, 1979.

24. S. Lee. Weak vs. strong: An empirical comparison of mutation variants. Master's thesis, Department of Computer Science, Clemson University, Clemson SC, 1991.

25. Sun Microsystems, Mountain View CA. *External Data Representation: Sun Technical Notes*, March 1990.

26. A. Beguelin, J. J. Dongarra, G. A. Geist, W. Jiang, R. Manchek, K. Moore, and V. S. Sunderam. *PVM 3.0 User's Guide and Reference Manual*. Oak Ridge TN, February 1993.