

Mutation Testing of Software Using a MIMD Computer

A. Jefferson Offutt
Roy P. Pargas
Scott V. Fichter
Prashant K. Khambekar
Department of Computer Science
Clemson University
Clemson, SC 29634-1906
email: ofut@cs.clemson.edu

Abstract – Mutation testing is a fault-based method for testing software that is computationally expensive. Mothra is an interpreter-based mutation testing system that is centered around an interpreter. This paper presents a parallel implementation of Mothra’s interpreter on a MIMD machine. The parallel interpreter, HyperMothra, is implemented on a sixteen processor Intel iPSC/2 hypercube. Our goal was to demonstrate that the expense of software testing schemes such as mutation can be reduced by using parallel processing, and we demonstrate this by measuring the performance gains of the parallel interpreter over the Mothra interpreter. Results are presented using ten test programs, three different static work distribution schemes, and various numbers of processors. On our test programs, we found that our parallel interpreter achieved almost linear speedup over Mothra’s sequential interpreter. With larger, faster high-performance computers available, mutation testing can be done at significantly less expense.

1992 International Conference on Parallel Processing, Chicago, Illinois, pages II-257--266, August 1992.

Introduction

The demand for complex and reliable software today is high, and this demand will remain high in the future. Reliable software is usually achieved through testing, but structured testing methods are computationally expensive when large, complex software systems are tested. Because of this expense, software testers often rely on informal, less expensive, and less effective testing methods. If more computing power is available to software testers, then the structured testing methods can be more feasibly used for large software systems. Parallel computer architectures, judiciously used, have tremendous potential for reducing the cost of software testing and thereby increasing the reliability of the software. This paper presents the development of and experimental results with HyperMothra, a parallel implementation of the Mothra mutation testing software system. HyperMothra currently runs on a sixteen processor Intel iPSC/2 hy-

In other words, a test set is adequate if it distinguishes the program being tested from a set of incorrect programs. Mutation testing can be regarded as a software analogue of the hardware fault-injection experiment. Mutation testing systems apply a collection of *mutation operators* to the test program, each of which produces a set of executable variations, called *mutants*, of the original program.

Test cases are used to cause the mutants to generate incorrect output. Mutant programs that have been shown to be incorrect by a test case are considered *dead* and are not executed against subsequent test cases. Some mutants are functionally equivalent to the original program and cannot be killed. The *mutation score* of a test set is the percentage of non-equivalent mutants that are killed by the test set. If the total number of mutants is M , the number of dead mutants is D , and the number of equivalent mutants is E , the mutation score is: $MS(P, T) = D/(M - E)$. This mutation score is a close approximation of the relative-adequacy of a set of test data; a test set is *relative adequate* if its score is 100% (all mutants were killed). *The goal of mutation testing is to find test data to kill all mutants.* The assumption is that such test data will provide a strong test of the original program.

Test cases for a test program can be created manually or automatically. The test data is high quality because the data must test for predetermined classes of faults, as represented by the mutations. The twenty-two mutation operators used by Mothra [5] represent more than ten years of refinement through several mutation systems. These operators explicitly require that the test data meet statement and (extended) branch coverage criteria, extremal values criteria, and domain perturbation; the mutation operators also directly model many types of faults. Each mutation operator is said to create a different *mutant type*.

The faults considered by relative adequacy are commonly restricted by two principles, the *competent programmer hypothesis* [1] and the *coupling effect* [3]. The

competent programmer hypothesis states that competent programmers tend to write programs that are “close” to being correct. In other words, a program written by a competent programmer may be incorrect, but it will differ from a correct version by only a few faults. The coupling effect states that a test data set that detects all simple faults in a program is so sensitive that it will also detect more complex faults. In other words, complex faults are coupled to simple faults. The coupling effect cannot be proved, but it has been demonstrated experimentally [11] and supported probabilistically [10]. The coupling effect allows us to focus on simple faults since test data that kills simple faults can also be expected to kill more complicated faults. Figure 1 gives a program statement followed by two mutated versions of that statement.

```

x = y + 1 (original statement)
x = x + 1 (first mutation)
x = y - 1 (second mutation)

```

Figure 1: Two Example Mutations.

The first mutation is created by replacing y in the original statement with x (a *scalar variable replacement* (**svr**) mutation), and the second mutation is created by replacing the $+$ binary operator with $-$ (an *arithmetic operator replacement* (**aor**) mutation).

Parallelism in the Mothra Mutation Testing Process

In any automated mutation testing system there are several important steps that a tester must follow. Because of the large number of mutants that are generated for each program ($O(N^2)$, where N is the number of variable references), it has been considered impractical to separately compile and store each mutant program. Therefore most mutation systems, including Mothra, have been built as interpretive systems. With this approach, instead of creating, compiling, and storing N^2 separate programs, the program is translated once into an intermediate form and each mutant is stored in the form of a short description of the changes to the intermediate code necessary to create the mutant. In Mothra, these descriptions are stored in records called *mutant descriptor records* (**MDRs**). The steps are presented here to expose the inherent parallelism in mutation testing. The testing process continues until the tester has a test set with a satisfactory mutation score or is forced to stop due to time or economic constraints. The major steps of testing with mutation are listed below; Mothra implements most of these steps as separate programs that communicate through files.

1. **program submission** – A program P is submitted for testing and is parsed to create an intermediate form ready for interpretation.
2. **test case generation** – A set of test cases T is submitted. Each test case in T contains values

for the input variables of P . T can be created manually by a tester or generated automatically by software such as the automatic test data generator **Godzilla** [4].

3. **original execution** – P is interpreted once for each test case in T by the tool **rosetta**. This produces a set of expected outputs E that has one element for each test case in T . The values in each element of E correspond to the output variables of P . The expected outputs of P can be examined at any point during testing to determine whether or not P performed correctly on T . If any output is incorrect, then P is incorrect and must be modified, forcing testing to be restarted from step 1.
4. **mutation operator selection** – One or more mutation operators are selected to be applied to P . Testers typically use all mutation operators available.
5. **mutant generation** – Each mutation operator selected is applied to P to produce the MDRs that describe the set of mutants M of P . Mothra uses the tool **mutmake** to create these MDRs.
6. **mutant execution** – Each mutant program in M is interpreted by **rosetta** using the input values for each test case in T . This produces a set of mutant outputs E' that has at most $|M| * |T|$ elements. E' is not usually this large, since mutants are not executed against new test cases after being killed. Also, T is usually small compared to M . (Note: A typical test case will kill a large number of mutants, so the number of executions is usually much less than $|M| * |T|$. For the 10 programs in Table 1, the actual number of executions ranged from 10% to 41% of $|M| * |T|$.)
7. **output comparison** – Each element of E' is compared with the element of E generated from the same test data. If an output differs, the mutant is killed. If, after output comparison, some mutant remains alive, either the test data in T is not adequate or the mutant is equivalent to P , and can never be killed. Determination of equivalent mutants is a time-consuming, manual process that is necessary in mutation testing.
8. **result analysis** – The results of testing are analyzed and, if necessary, further testing is done. If all mutants are dead and all mutation operators have been applied to P , then no further testing is necessary. If one or more non-equivalent mutants remain alive, then additional test cases should be added to T and appropriate steps of the process repeated. If, as a result of testing, faults in P are uncovered, then P must be modified and the testing process repeated. Existing test cases can usually be reused for subsequent testing.

The most computation-expensive parts of the mutation process are original execution, mutant execution,

output comparison, and test data generation. Since mutant execution is done once for each mutant and each test case, it is referred to as the *inner loop*. The inner loop includes the interpretation of a mutant on a test case, the comparison of the mutant output with original output, and, if they differ, the killing of the mutant. The inner loop is by far the most computationally expensive part of mutation testing, and therefore the target of our parallelization efforts. The most two obvious ways to parallelize mutant execution on a MIMD machine are by supplying each processor with all test cases and a subset of the mutants, and by supplying each processor with all mutants and a subset of the test cases. Either approach yields a straightforward parallelization that has a moderate amount of communication costs. Since the size of T is usually small relative to the size of M , we have implemented the first approach by parallelizing on the mutants.

Previous Work

Several proposals or attempts have been made to implement mutation testing on high-performance computer systems. Work in parallel mutation testing has been suggested for vector processors, single-instruction-multiple-data (SIMD) machines, and multiple-instruction-multiple-data (MIMD) machines.

Mutant unification was proposed by Mathur and Krauser [8, 9]. They suggest that vectorizable programs be created, each one incorporating several mutants of the same type. Their hope is that a vector processor could then execute the unified mutant programs and achieve a significant speedup over a scalar processor. The proposed strategy has never been implemented, and the authors imply in their papers that only scalar variable replacement (**svr**) type mutants are suitable for unification.

A later paper by Krauser, Mathur, and Rego [6] suggests a strategy for efficient execution of mutants on SIMD machines. As in mutant unification, the authors suggest that mutants of the same type be grouped together and that the groups be handled by different processors in the SIMD system. This strategy also has not been implemented.

Choi and Mathur give a general method for scheduling mutant executions on the nodes of a hypercube [2]. In this strategy, each mutant program is separately compiled on the host processor and the resulting executable programs are scheduled for execution on the node processors. The implementation of the strategy, called PMothra, runs on a 128 processor NCUBE/7 hypercube. Unfortunately, because of the cost of separately compiling each mutant program, PMothra actually ran slower than the single processor, interpretive version of Mothra. In their paper, Choi and Mathur suggested removing the compilation bottleneck from PMothra through a method called *compiler integrated testing*. In this method, the original program is compiled once and the mutant programs are created by making simple code patches to the original executable program. To date, no results have been reported from

this approach.

The principle difference between PMothra and HyperMothra is the way that the systems process mutants. In PMothra, each mutant is compiled separately, and the mutant executables are distributed to and executed by the node processors. HyperMothra distributes the MDRs to the node processors, which then apply the changes to the intermediate code and interprets each mutant. This method is directly based on Mothra's interpretive approach, and allows for a more direct comparison of HyperMothra with Mothra than does PMothra. Other differences between PMothra and HyperMothra are that HyperMothra has built-in efficiency improvements over Mothra (described later) and is implemented on an Intel iPSC/2, a second-generation hypercube.

Implementation of HyperMothra

Although Mothra is composed of several distinct programs, the changes required to create HyperMothra were localized within the interpreter, **rosetta**. **rosetta** not only interprets both original and mutant programs, but also saves and compares the outputs and, when necessary, kills mutants. This section summarizes the architecture of the Intel iPSC/2 hypercube, and presents the sequential algorithm used by **rosetta**. We made several modifications to **rosetta** that are independent of the parallelization; these are described first, then the parallel algorithm is presented.

Overview of the Intel iPSC/2 Hypercube

The iPSC/2 hypercube is an Intel 80386/80387 based parallel computer that comes in a variety of configurations [?]. The hypercube used for HyperMothra contains one *host* and sixteen *node* processors. The host node is connected to the disk and other peripheral devices and each node; the nodes have no direct connection to peripheral devices (any external communication must go through the host). The host and nodes each have eight megabytes of main memory. The nodes run the NX/2 operating system that provides capabilities for message passing, memory and process management. The host runs a System V UNIX operating system with extensions for cube and network management. Node-to-node messages go directly from the sender to the receiver via a *direct-connect* module so that non-adjacent nodes can pass messages without interrupting other nodes (this module was not available in older hypercubes).

Mothra's Sequential Interpreter's Algorithm

rosetta has two modes of operation; original execution and mutant execution, corresponding to steps 3 and 6 of Mothra's mutation testing process.

During original execution, the original program is interpreted and the output values are saved for later comparison. During mutant execution, each live mu-

```

1  begin rosetta
2  read intermediate code and symbol table
3  read test case input values
4  if (mutant execution) then
5    read mutant descriptor records (MDRs)
6  endif

7  for (each test case t) do
8    if (mutant execution) then
9      read expected output  $E(P,t)$ 
10   endif
11   if (original execution) then
12     interpret orig intermediate code on t
13     write expected output  $E(P,t)$ 
14   else /* mutant execution */
15     for (each live mutant m) do
16       modify original intermediate code to
17         produce m
18       interpret m on t to produce  $E(m,t)$ 
19       if (abnormal termination) then
20         kill m
21       elseif ( $E(m,t) \neq E(P,t)$ ) then
22         kill m
23       else
24         /* m remains alive */
25       endif
26       restore original program
27     endfor /* end of mutant loop */
28   endif
29 endfor /* end of test case loop */
30 end rosetta

```

Figure 2: Rosetta’s Algorithm.

tant is interpreted and the mutant is killed if its interpretation ends abnormally (e.g., times out due to an infinite loop, an arithmetic error) or if the output it produces differs from the output of the original program on that test case. Mutant interpretation continues until there are no more live mutants or there are no more test cases. Figure 2 gives **rosetta**’s algorithm. On line 17, each mutant *m* is executed on each test case *t*. This step is the computationally expensive step in Mothra, and is the step that we parallelize in HyperMothra.

Non-Parallel Performance Improvements

Before implementing the parallel version of the interpreter, several modifications were made to the sequential version in order to port it to the hypercube. The modifications were to reduce frequent (and unnecessary) file access, remove process forking, and restrict unnecessary freeing and reallocating of memory objects. This resulted in an *enhanced*, but still sequential, version of Mothra’s interpreter.

During mutant execution, **rosetta** writes mutant output to a temporary file in preparation for comparison with the expected output of the current test case. When the mutant interpretation is finished, both output files are read and their contents compared. **rosetta** also repeatedly reads the same expected output file for the current test case for every mutant interpretation and updates the MDR file every time it kills a mutant, again within the inner loop. Lastly, the sequential version of Mothra reads test case values into memory within the test case loop. Lee [7] reported that up to 60% of **rosetta**’s time is spent doing file I/O. When moving **rosetta** to a hypercube environment, these problems became critical because all file accesses from the node processors were processed through the host processor, swamping the system with communication costs.

To solve the file access problems, the following improvements were made to the parallel version of the interpreter: (1) mutant output is saved in memory rather than in a temporary file, (2) expected output is read only once for each test case and thereafter resides in memory, (3) an in-memory comparison of mutant output and expected output is done, (4) mutants are marked killed in memory during interpretation and the MDR file is updated only once at the end of the run, and (5) the call to reread the test case values within the test case loop was removed. (Items 1 and 3 above were first implemented on a sequential version of Mothra by Lee [7].)

Another problem with the sequential version of **rosetta** was that it uses the Unix system call *fork* to create a separate process for each mutant. On the hypercube, this system call is always processed through the host processor, again causing degradation due to communication costs. Therefore, the forking code was removed from the parallel interpreter and code was added to restore the intermediate code after each interpretation.

HyperMothra’s Parallel Interpreter’s Algorithm

On the Intel iPSC/2 hypercube a host program and a node program are typically developed by partitioning the sequential algorithm. During execution of a parallel application, the host program runs on the host processor and a copy of the node program runs independently on each of the node processors. Typically, the host program and the node program have separate responsibilities. We have divided **rosetta**’s algorithm

using a *producer-consumer* model, where the host processor serves as a producer of mutants, and the node processors serve as consumers of mutants.

Mutant interpretation and output comparison are delegated to the node processors, making each node processor responsible for a portion of the total number of mutants. In Mothra, the mutant interpretation and output comparison tasks are included within the loop over all live mutants (see Figure 2). Thus, the general strategy for converting Mothra to HyperMothra was to place a mutant loop structure in the node program and to implement the rest of the algorithm of Figure 2 in the host program. The result is a parallel application in which the host processor distributes mutants to each node processor and the node processors interpret mutants for each test case. Figures 3 and 4 give algorithms for HyperMothra’s host program and node program.

The host algorithm has two execution modes, *original* and *mutant*. In original mode, nodes are not used, and the host processor computes and saves the expected output from the original program. In mutant mode, the host node begins by sending the startup information to the nodes. For each test case, the test case values and the expected output is sent to the nodes, then once a node interprets all its mutants on the test case, a count of how many mutants remain alive is sent back. If all the mutants on all the nodes are dead, then the algorithm takes an early exit and does not send more test cases to the nodes, otherwise, the next test case is sent. When all the test cases have been processed, the nodes send the updated MDRs back to the host.

There is recurring communication between the host and a node; the host sends test case information which includes original output and a list of statements referenced in the original program, and the nodes send their live mutant counts which indicate the number of mutants still alive at the nodes.

Work Distribution Strategies

Although parallelizing mutation by mutants is a natural way to divide up the work, it does not necessarily guarantee an even distribution of work among the processors. Some mutants are killed by a test case that is executed early in a mutation run and are not executed against most of the test cases. At the other extreme, since equivalent mutants are not killed by any test case, they must be executed against *all* test cases. Thus, there is a wide variability in the amount of execution time required for individual mutants. To achieve maximal speedup, we would like to distribute mutants to nodes so that each node performs the same amount of execution. Unfortunately, we have no way of knowing ahead of time how much execution will be required for a mutant, or how many test cases will have to be run against it. Thus we cannot determine the optimum distribution of mutants.

A dynamic load balancing scheme that shifts mu-

```

1  begin HyperMothra_host
   /* also done by each node processor */
2  read intermediate code and symbol table
3  read test case input values

4  if (original execution) then
5    for (each test case  $t$ ) do
6      interpret orig intermediate code on  $t$ 
7      write expected output  $E(P, t)$ 
8    endfor /* end of test case loop */
9  else /* mutant execution */
10 load node program onto node processors
11 send initial experiment info to nodes
12 read mutant descriptor records ( $MDRs$ )
13 distribute  $MDRs$  to nodes
14 for (each test case  $t$ ) do
15   send  $t$  to nodes
16   read expected output  $E(P, t)$ 
17   send  $E(P, t)$  to nodes

   /* nodes now interpret mutants */
18 receive node live mutant counts
19 if (all mutants are dead) then
20   exit /* early exit from loop */
21 endif
22 endfor /* end of test case loop */
23 send "quit" message to nodes
24 receive modified  $MDRs$  from each node
25 merge and sort  $MDRs$ 
26 write  $MDRs$ 
27 endif
28 end HyperMothra_host

```

Figure 3: HyperMothra Host Program Algorithm.

tants from a loaded processor node to a node that has finished all its mutants and test cases is possible, but because of the communication costs that would be involved, we decided to experiment with static distribution strategies first. Three static distribution strategies are described below that attempt to balance the amount of work done by the processors. Each strategy distributes approximately the same number of mutants to each node. The distribution strategies are implemented on line 13 in Figure 3. Results from each of the three strategies are presented in the next section.

Distribution of MDRs in Original Order

Mothra generates MDRs first by mutation operator, then by program line number. Thus, for each mutation operator the user selects, the **mutmake** tool scans each statement in the program, creating all mutants for that mutant type. The first mutant distribution strategy divides the original MDR list among the node processors. If there are n processors and $|M|$ mutants,

```

1  begin HyperMothra_node
2    receive initial experiment info from host
3    read intermediate code and symbol table
4    read test case input values

5    while (host "quit" message not recvd) do
6      receive  $t$  from host
7      receive  $E(P, t)$  from host
8      if ( $MDRs$  not received from host) then
9        receive  $MDR$  /* done only once */
10     endif
11     for (each live mutant  $m$ ) do
12       modify original intermediate code to
13         produce  $m$ 
14       interpret  $m$  on  $t$  to produce  $E(m, t)$ 

15     if (abnormal termination) then
16       mark  $m$  as killed
17     elseif ( $E(m, t) \neq E(P, t)$ ) then
18       mark  $m$  as killed
19     else
20       /* mutant remains alive */
21     endif
22     restore original program
23   endfor /* end of mutant loop */

24   send live mutant count to host
25 endwhile /* end of test case loop */
26 send modified  $MDRs$  to host
27 end HyperMothra_node

```

Figure 4: HyperMothra Node Program Algorithm.

the first processor gets the first $|M|/n$ mutants, the second processor gets the next $|M|/n$ mutants, and so on. If the MDR list contains dead mutants, they are not distributed to the nodes.

Distribution of MDRs in Random Order

The second strategy for assigning MDRs to nodes distributes the MDRs randomly (uniformly) among the processors. To ensure repeatability of tests for this strategy, the number of MDRs for a program is used as the seed for the randomizer.

Distribution of MDRs by Mutant Type

The third mutant distribution strategy is based on the observation that certain types of mutants are generally harder to kill than others. For example, a large percentage of absolute value mutants (**abs**) turn out to be equivalent in many experiments. Thus a processor with a large number of these "hard to kill" mutants may require longer to process its workload than other processors. The third distribution strategy assigns each processor approximately the same number

of mutants of each type by using a *round-robin* distribution algorithm. If there are $|M_i|$ mutants of type i , then each processor gets $|M_i|/n$ mutants of type 1, $|M_2|/n$ mutants of type 2, and so on. Each processor receives the same number of live mutants (plus or minus one), and the mutation operators are as evenly distributed as possible.

Experimentation With HyperMothra

To determine the performance of HyperMothra and the effectiveness of the mutant distribution strategies, several programs of various sizes were tested using the parallel interpreter. This section presents the experimental methods, data, and results for HyperMothra. HyperMothra was compared against Mothra on the basis of speedup and efficiency for a variety of cube sizes and for all three mutant distribution strategies.

Experimental Programs

These experiments were performed with ten Fortran subroutines. Table 1 lists each selected program unit along with a brief description, the number of executable Fortran lines, the number of mutants generated by **mutmake**, the number of test cases we used, and the actual number of mutants interpreted (bounded by $|M| * |T|$). The complete source code listings for each program is given in the technical report [12].

In our experiments, we generated all MDRs for the programs using all twenty-two Mothra operators, generated test data automatically using the automatic test data generator **Godzilla** [4] for eight programs, and 100 test cases were hand-generated BIG and BG2. The original program was executed on the host processor, and mutants were executed sequentially first on the host, then on one node, and then in parallel on multiple nodes. In order to measure speedup only on the basis of the parallelization, we compute speedup relative to execution of mutants on one node. Finally, we executed mutants with all combinations of numbers of nodes (2, 4, 8, and 16), and different work distribution strategies (original MDR order, random MDR order, and mutation operator order). This involved 12 separate invocations of the parallel interpreter for each program.

Performance Measurements

Speedup for n processors is defined as execution time on one processor divided by execution time on n processors. Speedup indicates how much execution time is saved, which is heavily influenced by processor utilization. A common measure indicating processor utilization for parallel algorithms is efficiency. *Efficiency* for n processors is defined as the speedup for n processors divided by n . Acceptable speedup and high efficiency are the major performance goals for HyperMothra.

A factor that affects both speedup and efficiency is the variability in the number of mutants interpreted

Program	Description	Statements	Mutants	Test Cases	Mutants Interpreted
MAX	Index of the maximum element	5	111	106	1184
EUC	Greatest common divisor (Euclid's)	12	271	175	4681
BS	Binary search on an integer array	17	273	314	11455
BUB	Bubble sort on an integer array	11	294	231	11622
PAT	Search for a pattern	18	753	79	14491
TT	Classifies triangle types	24	951	719	69982
QCK	Non-recursive integer quicksort	31	1175	581	74856
FND	Partitions an array	30	1187	580	106408
BIG	A large (useless) program	46	5292	100	217360
BG2	A larger (useless) program	81	20690	100	806768

Table 1: HyperMothra Test Programs.

by each of n node processors. The processor with the greatest execution time ultimately determines the maximum speedup and efficiency achievable by the group of n processors. Since our three work distribution strategies assign mutants to processors in different ways, a variability of work measurement will help explain the observed performance for each strategy. The method used to measure the variability of mutants interpreted is as follows:

1. Let n be the number of node processors.
2. Let m_i be the number of mutants interpreted by node i .
3. Let m_{tot} be the total number of mutants interpreted by all nodes.
4. Let $m_{avg} = m_{tot}/n$, the average number of mutants interpreted by all nodes.
5. Let $m_{error} = \sum_{i=1}^n |m_i - m_{avg}|/n$. m_{error} represents the average number of mutant interpretations a given node differs from m_{avg} .
6. Let $m_{var} = m_{error}/m_{tot}$. m_{var} is scaled this way to prevent the number of mutants (m_{tot}) from skewing the variability measure.

In general, higher values of m_{var} indicate poor mutant distribution, and values of m_{var} close to zero indicate good mutant distribution. A comparison of m_{var} for each work distribution strategy will indicate the relative effectiveness of each strategy.

Experimental Results

This section presents performance results from HyperMothra. First, a comparison between HyperMothra and the original host program is given. The selected indicators of speedup, efficiency, and variability of mutants interpreted are then presented separately, and observations are made concerning HyperMothra's performance in each area. Also, we identify

the work distribution strategy yielding the best results and suggest reasons for its superiority.

Non-parallel performance improvements. To consider the modifications that improved the execution speed of the non-parallel version of the interpreter separately from the parallel modifications, we first computed the speedup of our enhanced interpreter over Mothra's original interpreter by running the original interpreter on the hypercube's host processor and the enhanced version on just one node processor. In all cases, the speedup is between 1.6 and 1.9. Later speedups are calculated based on the enhanced interpreter.

Speedup. In the HyperMothra experiments, speedup was calculated relative to one node processor for every combination of the 10 test programs, 3 distribution strategies, and cube sizes of 2, 4, 8, and 16. Figure 5 gives speedup results for 16 nodes. As might be expected, we achieved more speedup with larger programs, where the communication overhead and setup costs were less critical. In fact, for the two largest programs, speedup was nearly linear.

Table 2 shows the speed values for all 4 node sizes. Although the speedups increase as the number of processors increase, the speedups with fewer processors are closer to linear. This is because the communication costs are higher when there are more nodes. The *efficiency*, shown in the technical report [13], varied from 0.4 for small programs to 0.9 for large ones on 2 processors, and varied from 0.0 for small programs to 0.9 for large ones on 16 processors.

It is also apparent from both Figure 5 and Table 2 that work distribution strategies 2 and 3 (random MDR and mutant type order) give much better speedup than strategy 1 and that 3 is marginally better than 2. As expected, distributing mutants according to their original order causes some processors to be overworked with hard-to-kill mutants. This is discussed next under variability.

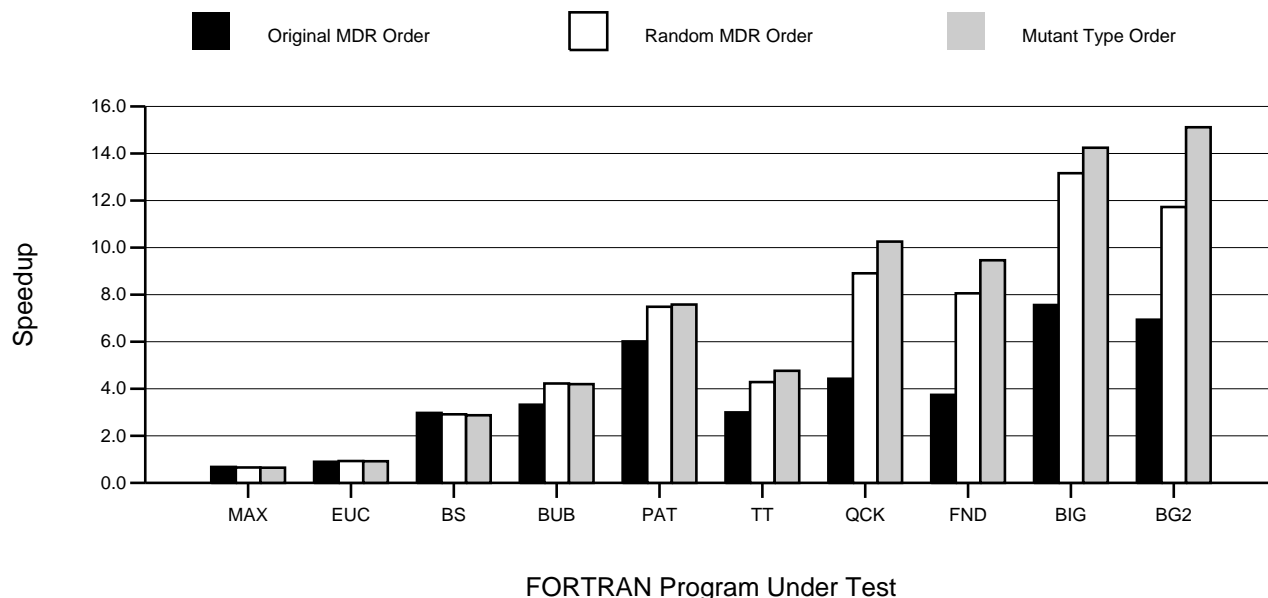


Figure 5: Speedup Achieved by HyperMothra Using 16 Processors.

Variability of mutants interpreted. We defined variability (m_{var}) as a measure of how well mutant interpretations are distributed among the processors. Variability affects both speedup and efficiency. In an ideal static work distribution scheme, each processor would perform the same number of mutant interpretations. When the number of interpretations done by some node deviates greatly from the average, distribution of mutants is poor and the value of m_{var} will be high. On the other hand, when each node performs approximately the same number of interpretations, distribution is good and the value of m_{var} will be close to zero. Figure 6 gives the variability of work for each of the 3 work distribution strategies using a 16 processor cube (graphs for node sizes 2, 4, and 8 are given in the technical report [12]).

In every instance, strategy 1 exhibits much higher value variability than strategy 2 or 3, and strategy 3 is somewhat better than strategy 2 in most cases. Unlike speedup and efficiency, there is little correlation between the program size and the variability of mutants interpreted. This is true because the variability measure is independent of communication overhead. These results suggest that strategies 2 and 3 distribute the hard-to-kill mutants to the nodes much more effectively than strategy 1 does.

Conclusions and Future Work

In this paper, we have demonstrated that high-performance architectures can be utilized to do unit-level software testing faster and less expensively than with traditional architectures. We have demonstrated this with an algorithm for parallelizing one unit testing strategy, mutation testing, and implementing mutation on a MIMD machine, specifically, a sixteen processor Intel iPSC/2 hypercube. With this implementation, we were able to see speedup results that approached linear, indicating that MIMD architectures work well for parallelizing mutation testing.

We also found that with small programs that had relatively few mutants, speedup was not as good as with larger programs. This is because with small programs the communication costs are high relative to the execution costs. From this, we conclude that software testing should be done on sequential machines when programs are small, and moved to parallel machines only when the size of the programs warrant the added expense. We also got better results with larger cube sizes, indicating that our algorithm scales up to larger machines fairly easily. Although we were limited to 16 nodes on our hypercube, we would expect this scalability to continue to be true with larger machines as well as more powerful processors.

We also implemented three different static work distribution strategies by distributing different subsets of mutants to the nodes. It is clear from the variability

NODES	DISTRIBUTION	MAX	EUC	BS	BUB	PAT	TT	QCK	FND	BIG	BG2
2	Original	1.6	1.1	1.2	1.4	0.9	1.6	1.3	1.2	1.8	1.8
	Random	0.9	1.2	1.8	1.6	1.9	1.8	1.7	1.9	1.9	1.9
	Mutant type	0.9	1.3	1.7	1.7	1.9	1.8	1.8	1.9	1.9	1.9
4	Original	2.1	1.3	1.3	1.9	0.9	2.7	1.7	1.5	2.4	3.1
	Random	0.9	1.3	3.0	2.9	3.4	2.7	2.9	3.4	3.7	3.5
	Mutant type	0.9	1.3	3.0	2.9	3.6	3.1	3.4	3.4	3.8	3.9
8	Original	2.7	2.2	1.1	2.1	0.8	3.7	3.0	1.9	4.3	4.2
	Random	0.8	1.1	3.5	4.1	5.1	3.3	5.0	5.0	6.8	6.4
	Mutant type	0.7	1.1	3.3	4.4	5.9	4.5	6.0	6.1	7.5	7.7
16	Original	2.9	3.3	0.8	3.7	0.6	6.0	4.4	2.9	7.5	6.9
	Random	0.6	0.9	2.9	4.2	7.4	4.2	8.9	8.0	13.1	11.7
	Mutant type	0.6	0.9	2.8	4.1	7.5	10.2	4.7	9.4	14.2	15.1

Table 2: Speedup Data for HyperMothra.

results in Figure 6 that our strategy 3, mutant type order, allowed for very good usage of processor time. Again, this was most evident for the larger programs in our test suite.

As discussed earlier, several other proposals for parallelizing mutation have been made. To date, most proposals have not been implemented, and the one implementation has several severe limitations that makes it impractical for extensive testing applications (the compilation bottleneck, in particular), and was not directly comparable to Mothra, making it hard to compare with sequential systems. HyperMothra is as easy and practical to use as Mothra, and was close enough in design to be directly comparable, thus the speedup figures given reflect what we would expect if a MIMD machine was used for mutation testing in a production environment.

One problem with HyperMothra is that communication overhead is fairly high. This is true because the host parallel algorithm broadcasts one test case at a time to all nodes, and some small programs do not require much time for mutant interpretation. To decrease communication overhead in all cases, test cases could be sent to the nodes in blocks. Since few large messages (n test cases at a time) are processed more efficiently than many small messages (one test case at a time) in hypercube systems, this offers great potential to improve the performance of HyperMothra.

Another problem in HyperMothra is that nodes often sit idle waiting for the slowest node to finish interpreting mutants. If nodes were allowed to request work from the host rather than wait for the host to send the next test case, then the idle time could be significantly reduced. This *demand-driven* strategy seeks

to restrict overhead to the time necessary for communication of test case information. Given the low variability results in Figure 6, it is not clear that this improvement would have a large impact on speedup.

Another possible improvement to HyperMothra is the introduction of *dynamic load balancing* to the system. Currently, mutants are assigned to processors before they are interpreted, and there is no way to redistribute mutants during interpretation if one or more processors become overworked. With dynamic load balancing, mutants can be reassigned during interpretation so that each processor performs approximately the same amount of work. There are several effective dynamic load balancing schemes available, and some of these can be implemented in HyperMothra. The question remains, however, whether or not the performance gains (if any) due to dynamic load balancing justify its increased complexity and communication overhead. We are currently exploring dynamic load balancing schemes for HyperMothra [?].

Finally, more computing resources can be used to make mutation testing for large software systems even more beneficial. Mothra’s design is easily parallelized on any MIMD computer. The current implementation of HyperMothra runs on a small, second-generation hypercube. Intel’s third-generation hypercube systems contain i860 processors that are at least eight times faster than the 80386/80387 processors of their iPSC/2 systems. An i860 implementation of HyperMothra using 64 or 128 nodes could be one to two orders of magnitude faster than HyperMothra on the sixteen node iPSC/2 system. Such performance improvements might persuade software testers to use structured testing methods like mutation testing to test their software.

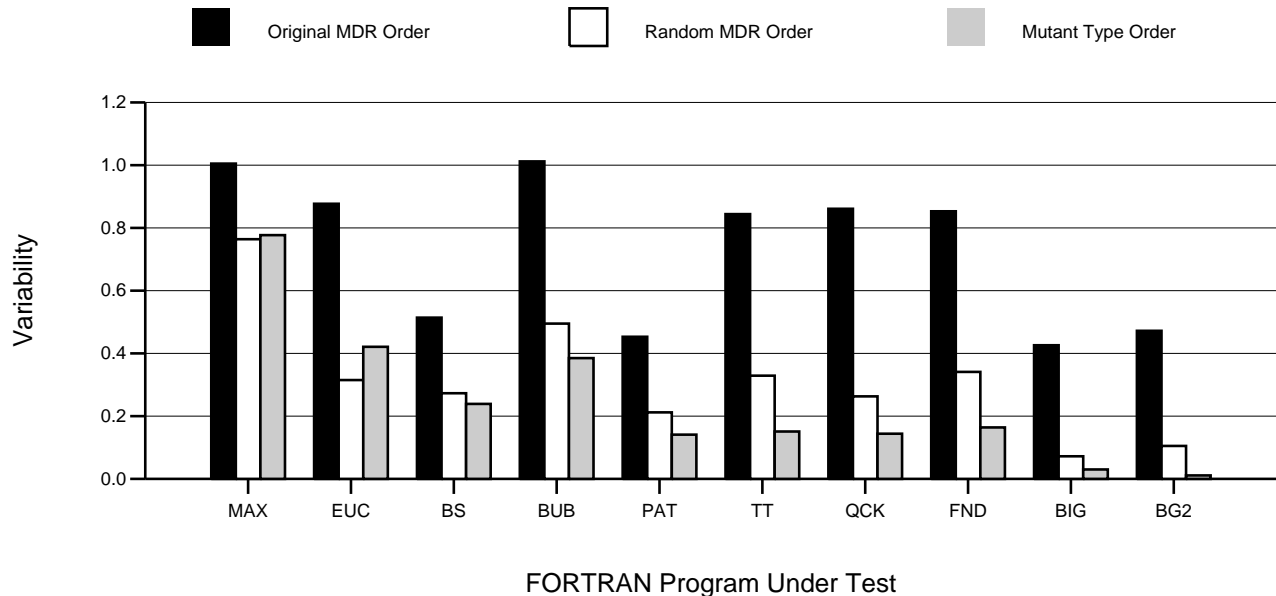


Figure 6: Variability of Work Using 16 Processors.

References

- [1] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, September 1979.
- [2] B. J. 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.
- [3] 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.
- [4] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.
- [5] 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.
- [6] E. W. Krauser, A. P. Mathur, and V. Rego. High performance testing on SIMD machines. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 171–177, Banff Alberta, July 1988. IEEE Computer Society Press.
- [7] S. Lee. Weak vs. strong: An empirical comparison of mutation variants. Master’s thesis, Department of Computer Science, Clemson University, Clemson SC, 1991.
- [8] Aditya P. Mathur and E. W. Krauser. Modeling mutation on a vector processor. In *Proceedings of the 10th International Conference on Software Engineering*, pages 154–161, Singapore, April 1988. IEEE Computer Society Press.
- [9] Aditya 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] L. J. Morell. Theoretical insights into fault-based testing. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 45–62, Banff Alberta, July 1988. IEEE Computer Society Press.
- [11] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3–18, January 1992.
- [12] A. J. Offutt and S. V. Fichter. A parallel interpreter for the Mothra mutation testing system. Technical report 92-100, Department of Computer Science, Clemson University, Clemson SC, 1992.

- [13] A. J. Offutt, Roy P. Pargas, Scott V. Fichter, and Prashant K. Khambekar. Mutation testing of software using a parallel computer. Technical report 92-105, Department of Computer Science, Clemson University, Clemson SC, 1992.