Parallel Genetic Algorithms for Sequential Circuit Test Generation

Dilip Krishnaswamy, Michael Hsiao, Elizabeth M. Rudnick, Prithviraj Banerjee, Janak H. Patel and Vikram Saxena

Technical Report No. CPDC-TR-9803-018

March 4, 1998

Center for Parallel and Distributed Computing
Department of Electrical & Computer Engineering
Northwestern University
Technological Institute
2145 Sheridan Road, Evanston IL-60208

Parallel Genetic Algorithms for Sequential Circuit Test Generation

Dilip Krishnaswamy[†] **Michael Hsiao**[†] **Elizabeth Rudnick**[†]
Prithviraj Banerjee[‡] **Janak H. Patel**[†] **Vikram Saxena**[†]

[†] Center for Reliable and High Performance Computing
University of Illinois
1308 W. Main St.
Urbana, IL 61801
{dilip,mhsiao,liz,patel}@crhc.uiuc.edu

[‡] Center for Parallel and Dist. Computing
Northwestern University
2145 Sheridan Rd
Evanston IL 60208
banerjee@ece.nwu.edu

This research was supported in part by the Semiconductor Research Corporation under Contract SRC 96-DP-109 and the Advanced Research Projects Agency under contract DAA-H04-94-G-0273 administered by the Army Research Office. Dilip Krishnaswamy (dkrishna@ichips.intel.com) is presently with Intel Corporation, 1900 Priarie City Road, M/S FM5-108, Folsom CA95630. Michael Hsiao (mhsiao@ece.rutgers.edu) is presently an Assistant Professor at Rutgers, State Univ. of NJ, Dept. of Electrical and Computer Engr., 94 Brett Road, Piscataway, NJ 08854-8058. Vikram Saxena (vikram@synopsys.com) is presently with Synopsys Inc., 700 East Middlefield Road, Mountain View, CA 94043. A preliminary version of this paper has been presented at the International Conference on VLSI Design, Jan 1997. The experimental results in this paper are all new.

Parallel Genetic Algorithms for Sequential Circuit Test Generation

Abstract

In this paper, we propose three new parallel genetic algorithms for simulation-based sequential circuit test generation. The three parallel algorithms are based upon a sequential algorithm that targets a group of faults simultaneously. The first algorithm is PGATEST1, which uses data decomposition to distribute the computation of the fitness of different members of a population among various processors. It produces the same result as the sequential algorithm. The second algorithm, PGATEST2, uses a parallel search strategy where each processor executes the sequential genetic algorithm with a different seed, and uses migration to share information between processors. This parallel algorithm has the potential of obtaining a better quality result than the serial algorithm. The third algorithm, PGATEST3, is a subpopulation based version of PGATEST2, where subpopulations are distributed across processors, and information is migrated from one processor to another.

All implementations were done using the MPI communication library and are therefore portable to many parallel platforms. Experimental results are reported on a wide range of sequential benchmark circuits on an IBM SP-2 multicomputer and a SUN Ultra Enterprise 3000 multiprocessor. All algorithms provide excellent performance in terms of speedup and quality on both shared-memory and distributed-memory parallel platforms.

List of Tables

List of Figures

Parallel Genetic Algorithms for Sequential Circuit Test Generation

1 Introduction

The rapid advances in VLSI technology have enabled the implementation of complex digital circuits and systems. This has resulted in increased complexity of testing such chips to ensure that they work correctly. A long sequence of input test vectors is applied to circuits by a tester. The sequence (also known as the test set) of vectors applied must be able to identify as many faults as possible in the circuit, so that if any fault is present on a chip, the chip can be identified as being defective and then be discarded. This test sequence must have high defect coverage if it is to be effective in identifying defective chips. Furthermore, since the cost of testing VLSI chips is a significant fraction of the overall manufacturing cost, the time required to test a chip should be minimized. It is therefore imperative to have fast and efficient tools for automatic test pattern generation (ATPG) to obtain compact test sets with high defect coverages in a short time.

The objective of automatic test generation is to obtain a set of test vectors that detect defects that might occur in the manufacturing process. The problem of test generation belongs to the class of NP-complete problems, and it is becoming more and more difficult as the complexity of VLSI circuits increases [1]. Parallel implementations can potentially provide significant speedups while retaining good quality results.

In this paper, we investigate three new parallel genetic algorithms for simulation-based sequential circuit test generation. Simulation-based test generators are more capable of handling the constraints of complex design features than deterministic test generators. The three algorithms are based upon a sequential algorithm that targets a group of faults simultaneously. The first algorithm, PGATEST1, uses data decomposition to distribute the computation of the fitness of different members of a population among various processors. It produces the same result as the sequential algorithm. The second algorithm, PGATEST2, uses a parallel search strategy where each processor executes the sequential genetic algorithm with a different seed and uses migration to share information between processors. This parallel algorithm has the potential of obtaining a better quality result than the serial algorithm. The third algorithm, PGATEST3, is a subpopulation-based version of PGATEST2, where subpopulations are distributed across processors and information is migrated from one processor to another. All parallel implementations were done in C++ using the MPI communication library [2], and are therefore portable to many parallel platforms. Results are shown on 6-processor SUN Ultra Enterprise 3000 shared memory multiprocessor and on a 16-processor IBM SP2 distributed memory multicomputer. All algorithms provide excellent performance in terms of speedup and quality on both shared-memory and distributed-memory parallel platforms.

The paper is organized as follows. We begin with a brief description of the problem of test generation in the following section. In Section 3, existing serial and parallel algorithms for test generation are discusssed. In Section 4, we provide a brief description of the serial genetic algorithm for test generation that was used in this paper. We then provide a general discussion on parallel genetic algorithms in Section 5. The proposed parallel genetic algorithms for sequential circuit test generation are presented in Section 6. An analysis of the proposed parallel algorithms is presented in Section 7. The experimental results with the proposed parallel genetic algorithms are discussed in Section 8. Section 9 concludes the paper.

2 Test Generation

The goal of automatic test generation is to obtain a sequence of input test vectors that will detect any defect that might occur in the manufacturing process. However, covering all potential defects would be very difficult and would require a large number of test vectors.

In general, the effect of a defect in a chip is represented by a fault model, which represents a variation in the circuit signals caused by the defect. In general a fault model can only model a subset of potential defects in the chip. In this paper, we will consider gate-level circuits for which the single stuck-at fault model is used most often, with the assumption that many additional defects will be detected by tests generated using this fault model. It is possible that multiple faults may be present in the same chip. However, we assume that there exists only one fault in the circuit. Hence, with the single stuck-at fault model, faults that tie a circuit node to logic one or logic zero for every node in a circuit are considered.

In a typical ATPG tool, faults in a circuit are targeted one-at-a-time using the single stuck-at fault model [3]. This is done by taking the good circuit and modifying it to include a fault by forcing a line in the circuit to a logic 1 or a logic 0. Then the test generator is used to obtain a test for the fault. A test could be a single input vector or a sequence of input vectors. Consider, for example, the node C in the circuit in Figure 1, in which the output of the NOR gate is tied to logic 1 (C s-a-1 i.e. C stuck-at-1). A logic zero must be applied to node C if there is to be a difference between the faulty and fault-free circuits. This is obtained by setting either input A or input B or both to logic 1. Let us assume that we set input A to logic

1 and input B to logic 0 so that node C is at value 0 in the fault-free circuit. This procedure is known as *fault excitation*. Now, it is not enough to control the inputs to the circuits so that there is a different value at the line corresponding to the fault. One needs to observe this difference at the output of the circuit. Hence, a logic 0 must be applied to the input nodes D and E of the 3-input OR gate. Therefore, a vector ABDE $\equiv$ 1000 produces an output 1 in the presence of the fault and an output 0 in absence of the fault, and is hence able to detect the fault. It is therefore called a *test vector* for the fault C s-a-1. It can be seen that the vectors ABDE $\equiv$ 0100 and ABDE $\equiv$ 1100 are also possible test vectors for the fault. There are many other faults that must be considered during the test generation process. Some of these faults are logically equivalent and no test can distinguish between them. For example, the faults C s-a-1, D s-a-1, E s-a-1, and F s-a-1 are all equivalent since a test that detects any one of them detects all of them (as all of them require all inputs of the OR gate to be at logic 0). Therefore, prior to test generation, equivalence classes of faults are created and only one fault from each equivalence class is selected as a candidate fault for which a test needs to be generated. This procedure is known as equivalent fault collapsing [3].

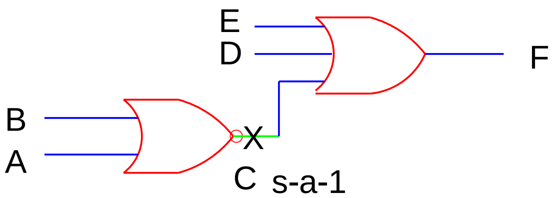

Figure 1: Circuit Showing a Single Stuck-at Fault

For sequential circuits, a sequence of vectors is usually required before a fault can be identified as being detected. A test is said to be generated when the output value of the good circuit differs from the output value of the faulty circuit at one or more of the output nodes. When a test is successfully generated to detect a fault, a fault simulator is used to find all other faults incidentally detected by the test. In a typical fault simulator, the good circuit (fault-free circuit) and the faulty circuits (corresponding to each undetected fault) are simulated for the test vector sequence obtained from the test generator. If the output responses of a faulty circuit differ from those of the good circuit, then the corresponding fault is detected. In the above example, since the vector 1000 is generated as the test vector, this vector is used in fault simulation to drop all faults that it can detect. The circuit is therefore simulated for each faulty circuit with the vector 1000 as input. We can see that the fault A s-a-0 is a fault that produces a value 1 at the output F for the faulty circuit and a value 0 at the output F for the fault-free circuit. This fault is therefore detected in addition to the fault C s-a-1 and is hence dropped from the list of faults that need to be targeted. Thus, in fault simulation, all detected faults are removed from the fault list and need not be targeted by the more computation-intensive test generator which then proceeds to target the next undetected fault. This procedure is iterated until there are no more faults left that can be detected. Usually one can find test vectors to detect most of the faults in the circuit. However, some faults are very difficult to test, and a test generator may not be able to come up with the right sequence to test such faults. Some other faults may be redundant. Hence, there does not exist any vector sequence to test them. Redundancies may exist due to combinational logic redundancies or due to characteristics of the finite-state machines corresponding to the good and faulty circuits. The reader is referred to [4, 5, 3, 6] for discussions related to redundant faults. For purposes of our discussion, we will assume that a redundant fault is one for which no test exists that would detect it. Let us assume that we have $\mathbf{F}$ faults in the fault list after performing equivalent fault collapsing. It should be noted that since we are using stuck-at faults, $\mathbf{F} = \mathbf{O}(\mathbf{L})$, where $\mathbf{L}$ is the number of lines in the circuit. Now, let us assume that $\mathbf{D}$ faults are detected and that $\mathbf{r}$ faults are identified as redundant during the test generation process. Then the fault coverage provided by the set of vectors obtained from the test generator is defined as $\frac{\mathbf{D}}{\mathbf{F}}$, while the fault efficiency is defined as $\frac{\mathbf{D}+\mathbf{r}}{\mathbf{F}}$.

3 Previous Work

In this section, we present existing serial algorithms for test generation followed by a discussion of parallel algorithms for test generation.

3.1 Review of Serial Algorithms for Test Generation

Due to the high complexity of the sequential circuit test generation problem, the automatic generation of high quality tests for highly sequential circuits remains a challenging task. The first automatic test generation algorithm (ATPG) for sequential circuits was proposed by Seshu and Freeman [8]. Since then, many new algorithms have been proposed.

Currently, most sequential circuit test generators use modified versions of deterministic combinational test generation algorithms. On such algorithm is PODEM [9], an enumeration algorithm in which all input patterns are examined as tests for a given fault. In this algorithm, the search for a test continues until the search space is exhausted or a test pattern is found. If no test pattern if found, the fault is considered to be untestable. Hence, the test generation problem is viewed as a search problem in the vector space of primary inputs. All nodes in the circuit, other than the primary input nodes, can be considered to be some function of the primary inputs. Hence, when primary inputs are chosen as nodes in the search space, each node can be assigned independently, and there will not be any conflicts with respect to the assignment of values to primary inputs. However, a given assignment of values to the primary inputs may give rise to a conflict on an internal node in the circuit. Suppose that we have a set of primary input assignments, and another primary input has been assigned a value that causes a conflict. Then one has to try the complement of the current value at the input node to determine if the current assignment will ever be compatible with the goal. If the complementary assignment also conflicts, then the goal cannot be achieved with the existing input assignments. Hence, a large portion of the search space can be pruned off in this manner. The algorithm picks a fault from the list of undetected faults and attempts to generate a test for this target fault. PODEM first tries to identify a value to be assigned to a primary input that would excite the target fault. It does so by *backtracing* from the fault site to the primary inputs. Backtracing is the process of working backward in the circuit from a current goal (e.g., setting the faulty node to a value opposite to the stuck-at value for the fault) to determine an input assignment. In the backtracing step, the handling of components other than simple gates can be difficult. This step is followed by a *forward implication step*, akin to traditional logic simulation, which determines the values of all internal nodes of the circuit due to this input assignment. It is checked whether the simulation reveals any conflicts with the current goal. If there are no conflicts, then the procedure tries to find another input to assign a value to. The process is repeated until a test is found or until the algorithm finds that it cannot backtrack anymore, that is, it has tried all possible input combinations explicitly or implicitly. Thus, the PODEM algorithm will eventually try all the primary input combinations, either explicitly or implicitly, until it finds a test, hits a limit on the number of backtracks, or exhausts all combinations.

While one only needs to generate a single input vector to test a fault in a combinational circuit, one needs to typically generate a sequence of vectors to detect a single stuck-at fault in a sequential circuit. Due to the presence of memory elements, the controllability and observability of nodes in the circuit can become more complicated [3]. These factors cause the complexity of sequential circuit test generation to be much higher than that for combinational circuits [10, 11, 12, 13, 14, 15, 16]. One can treat a sequential circuit as a combinational circuit expanded over many time frames. Each time frame has a present state value and a next state value. The next state value in one time frame corresponds to the present state value in the succeeding time frame. For each target fault, fault excitation is performed. If the fault effects have not propagated to a primary output in the same time frame that the fault is excited, then the fault effects present at the flip-flops are propagated to the primary outputs in succeeding time frames. Propagation of a fault effect to a node is said to have occured when the value assigned to the node differs between the good and the faulty circuits. The excitation of a fault imposes a requirement on the values of the flip-flops corresponding to the state of the circuit. This state is justified using reverse time processing. In this state justification step, the required values at the flip-flops are traced back in time using time frames until the requirement on the state of the flip-flops are simply “dont-care” values. A “dont-care” at a node imples that one can use either a 1 or a 0 as the value at that node. However, the use of such a deterministic approach can result in very large test sequence lengths and it can also take up a long time for execution. Therefore, simulation-based test generators have been proposed as an alternative [8, 17, 18, 19, 22, 21, 20]. In a simulation-based approach, processing occurs only in the forward direction and no backtracing is required. This makes the handling of complex component types to be easier. As a result, the development time is greatly reduced. The earlier work in simulation-based test generation used many heuristics and ad-hoc techniques.

More recently, genetic algorithms [23] have been proposed as a framework for simulation-based test generation in [24, 25, 27, 28, 26, 29, 30, 31, 32, 33, 34]. The serial test generation algorithm used in this paper, GATEST [28, 29], has been developed at the University of Illinois. The GATEST algorithm provides compact test with high fault coverages and it is faster than a deterministic test generator such as HITEC. This paper proposes parallel algorithms based on this serial algorithms. The serial algorithms used and the proposed parallel algorithms will be discussed in detail in subsequent sections.

3.2 Review of Parallel Test Generation

In this section, we discuss several approaches that have been proposed for parallelizing test pattern generation. More detailed descriptions on these approaches can be found in [35, 36].

3.2.1 Fault parallelism

Fault parallelism refers to the generation of test patterns for individual faults in parallel. The complete fault set is divided equally among all processors, each processor generating tests for its subset independent of all the other processors [7, 37, 38, 39, 40, 41, 42, 43, 44, 45]. The main advantage of fault parallelism is that the communication overhead is very low, and it is possible to achieve linear speedups when the number of processors is very small compared to the number of faults. However, faults that are *hard-to-detect* for the uniprocessor algorithm remain undetected even in the parallel implementation. However, this method suffers from a serious drawback as far as the overall test length and run time are concerned. Useless effort may be expended to detect a fault which may be detected with test patterns generated for a fault during fault simulation on another processor.

3.2.2 Heuristic parallelism

Almost all test generation algorithms use some heuristics to guide test generation. Experiments reported in [46] suggested that there is no clear cut advantage of using one heuristic over the other. We can exploit *heuristic parallelism* by letting each processor use a different heuristic to guide the search for the same fault [37].

The main advantages of this method are that the recoding effort is very low, and the communication overheads are low. The main disadvantage of this method is that the parallelism is limited by the number of heuristics available for search (at most five to six). Also, with different heuristics, there is no guarantee that the search spaces are disjoint. This may lead to redundant work. Also, no improvement is possible if a fault remains undetectable for all the heuristics.

3.2.3 Search space parallelism

A way to view the test generation process is as a search in a search space of primary inputs that can be each assigned 0/1 values in different time frames. Depth-first search is typically performed on that large search space, with backtracks on input assignments when one encounters conflicts. *Search space parallelism* refers to evaluating the functions associated with several decision points in parallel [47, 48, 41, 44, 49, 42].

The advantage of search-space parallelism is that the processors search different parts of the search space and therefore do not perform redundant work. Parallel processing is used to speed up the test generation for a particular fault. This is especially important when we have *hard-to-detect* faults. An additional advantage of this scheme is that the communication overhead can be kept low, and a large number of processors can be utilized efficiently. Superlinear speedups may also be possible, but the disadvantage is that coding is difficult.

3.2.4 Functional parallelism

Functional parallelism can be exploited in algorithms that use divide-and-conquer methods to solve certain problems. In such methods, a task is divided into a series of subtasks, all of which have to be completed and the results merged [49, 50].

The disadvantage of the technique is that the subtasks can be executed in parallel if they do not access or modify any common variables. If they have common variables, execution of the functional-parallel subtasks becomes complicated due to variable binding conflicts. As a consequence, the speedups achievable using the functional parallel approach to test generation are limited.

3.2.5 Circuit parallelism

In all the parallel approaches discussed so far, each processor keeps a copy of the entire circuit. For extremely large circuits, the memory of each processor may not be enough to store the entire circuit. In a *Circuit Parallel* approach, each processor keeps a partition of the circuit and performs various operations (such as backtracing, justification and implication) on its own subcircuit to satisfy various test generation objectives [51, 52, 53]. With this approach, it is extremely hard to achieve efficient speedups.

In this paper, we propose new parallel genetic algorithms in our approach to parallel test generation. The first of the proposed algorithms is similar to the one proposed in is similar to the one used in GATTO* [54]. We will compare these algorithms when we present in detail the various approaches to using parallel genetic algorithms for sequential circuit test generation in Section 5. This concludes our discussion on related work in the areas of serial and parallel algorithms for test generation.

4 Serial Genetic Algorithm For Test Generation (GATEST)

We now discuss the serial implementation of the GA-based sequential circuit test generator, GATEST [28], used in this paper in detail. The serial algorithm GATEST [28] is a simulation-based approach to test generation. In a simulation-based approach, test generation proceeds in the forward direction. Input vectors or sequences of input vectors are simulated, and any sequence found that can detect one or more faults is added to the test set. A genetic algorithm is used in GATEST to direct the search in the hope of finding such vector sequences in a fast and efficient manner. The serial implementation of the test generator uses a GA akin to the simple GA proposed by Goldberg [23] and illustrated in Figure 2. The GA contains a population of individuals which evolves based on the genetic principles of selection, crossover and mutation [23]. Each individual represents a sequence of test vectors which may be added to the test set if it can detect one or more faults that have not been detected during the course of execution. Random vectors are picked for the initial population for the GA. The fitness of each individual in the population is measured using a sequential circuit fault simulator. When a new population evolves from an old one, individuals with high fitness values are preserved from one generation to the next. Pairs of individuals are selected with a higher probability of selection assigned to individuals with higher fitness values. These pairs of individuals are crossed to obtain new pairs of individuals for the new population. Tournament selection without replacement is used, where two individuals are removed from the population, and the better individual is selected from the two. The two individuals are set aside and are not replaced until all other individuals have also been removed for further selection operations.

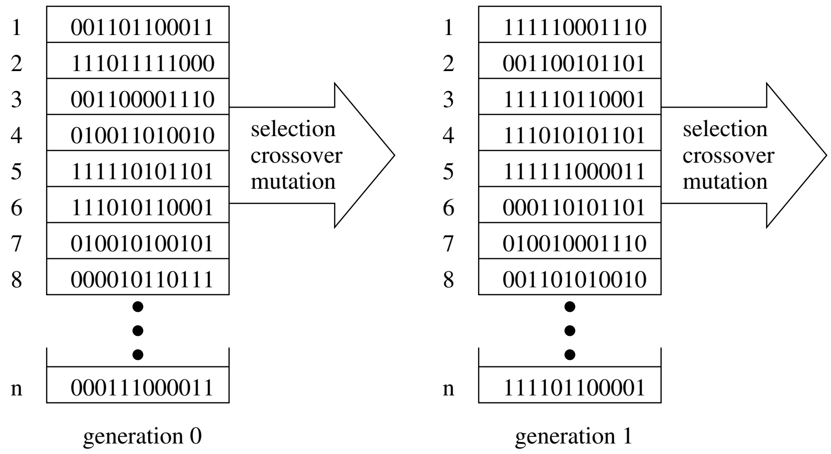

Figure 2: A Simple Genetic Algorithm (GA)

Crossover of every pair of selected individuals is always performed; hence, the crossover probability is 1. A very low probability of mutation of 0.01 is also used which means that, on the average, one in 100 bits are randomly flipped in the bit vector sequences representing the new individuals. The GA population evolves until an individual is found that can meet the desired goal, such as detecting one or more faults, or until marginal or no improvement is observed in the fitness values, or until a fixed maximum number of eight generations is reached.

4.1 Serial Implementation of GATEST

Processing in the sequential GA-based test generator GATEST is divided into three stages. The three stages of the test generator target different goals, and their corresponding fitness functions are different. In each of the three stages, the GA targets all faults in the fault list in groups of 31 faults, traversing the fault list several times until little or no more improvement is made. Between consecutive GA runs, useful vectors from the best sequence are added to the test set, and the two best sequences from the previous GA run are used to seed the population in the next GA run. The remaining individuals in the population are initialized with random test sequences. The test generation algorithm is shown below in Figure 3.

The GA population evolves over several generations until no more improvements in fitness values are obtained or the maximum of eight generations is reached. The population size is a function of the string length, and the string length is equal to the number of primary inputs in the circuit multiplied by

```
Algorithm: GATEST

While there is improvement in this stage
    While there is improvement in the GA
        For all undetected faults, in groups of 31 faults
            target-faults = next 31 undetected faults
            best-sequence = GA-test-generate (target-faults)
            fault-simulate (best-sequence)
            seed the next GA population (best 2 sequences)
        End For
        compute improvement
    End While
    compute overall improvement
End While
```

Figure 3: GATEST Algorithm.

the test sequence length. A multiple of the sequential depth of the circuit is used as the test sequence length, where the sequential depth is defined as the minimum number of flip-flops in a path between the primary inputs and the farthest gate. The test sequence length is set equal to the sequential depth in the first stage, two times the sequential depth in the second stage, and four times the sequential depth in the third stage. Also, the population size is doubled in the third stage to further expand the search space.

For the purpose of evaluating the fitness of an individual, a fault simulation using a small fault sample of 31 faults is performed using the individual as the test vector sequence. The reason 31 faults are chosen is to maximize the execution speed. Fault simulation on the entire fault list can be costly, and previous work has shown that small fault samples give good quality results as well [28]. The good circuit evaluation requires only one bit position out of the 32 available positions in a computer word, and the remaining positions can be used for faulty circuit simulations. Thus, groups of 31 faults are used as fault samples in the current work.

Figure 4 shows how faults are grouped for test generation. Initially the first 31 undetected faults are used as the fault group targeted for test generation. All individuals in the GA population target the same group of 31 faults, and the individuals evolve in successive generations until no more improvements in fitness values are obtained. Then a full-scale fault simulation is performed, using the entire list of undetected faults and the best test sequence found, and all detected faults are dropped from the fault list. An improved version of PROOFS [7] was used as the fault simulator. For the next GA run, the next group of 31 undetected faults is obtained, starting at the position in the fault list where the previous fault

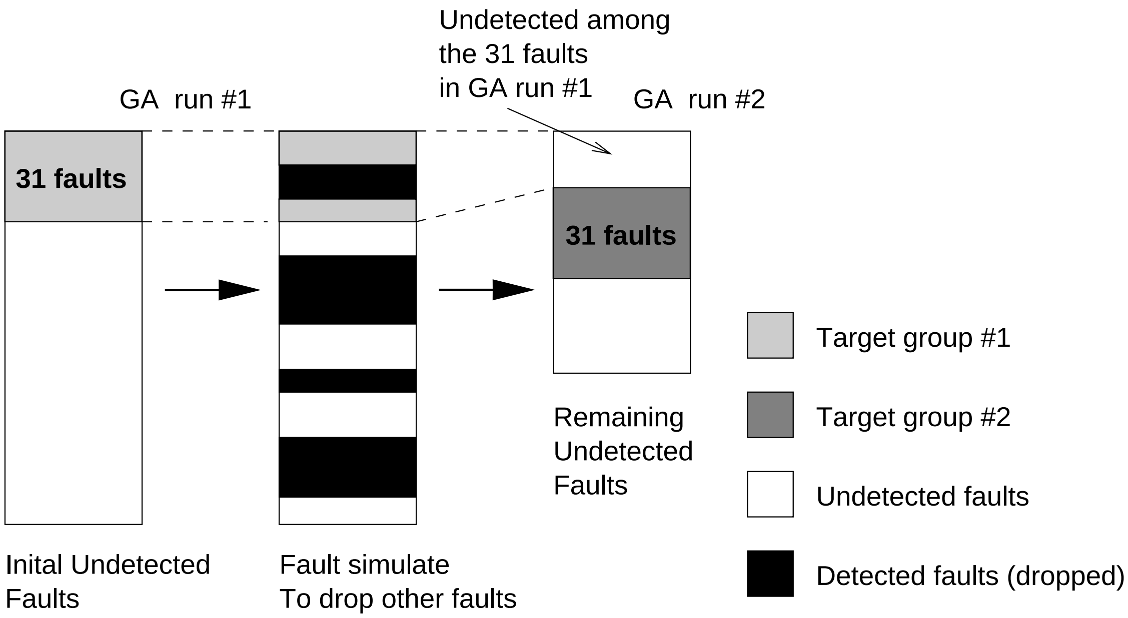

Figure 4: Fault Grouping of 31 Faults

group left off. Only the good portion of the best test sequence is added to the test set to keep the test set compact.

4.2 Redundant Fault Identification

The simulation-based test generation approaches as used in GATEST and STRATEGATE cannot identify redundant faults. One needs to exhaust the search space to identify all redundant faults. Hence, although a high fault coverage may be achieved using a GA-based approach, the fault efficiency of the test generator is no better than the fault coverage. Hence, once the GA has finished generating test vectors, one should run a redundancy-identifying algorithm to identify the redundant faults in the list of undetected faults that remain. This will improve the efficiency of the overall test generation process. A deterministic test generator such as HITEC [15] can be used to accomplish this. HITEC targets each fault in a fault list one at a time. Usually, most redundant faults can be identified in a very short time compared to the actual test generation process. Therefore, by assigning a short time-out of 1 sec per fault, one can identify most of the redundant faults among the faults that remain in the fault list after the GA has finished test generation. In principle, one could have used any program which would accomplish the task of redundancy identification.

Hence, the overall approach that we have adopted is to use a simulation-based approach using genetic algorithms to get good fault coverage and small test sizes, followed by a short redundancy identification step to improve the efficiency of the test generator.

5 Parallel Genetic Algorithms

Genetic algorithms can be parallelized in various different ways. Good overviews for parallel GAs can be found in [56, 57, 58, 59]. We shall now discuss some of the algorithms below in the context of our application.

One can implement a parallel GA using *parallelization by data decomposition*. In this model, a single population exists, and evaluation of the individuals (and sometimes the application of the genetic operators) is completed in parallel. The evaluation can be parallelized by assigning a subset of individuals to each of the processors. A second approach is to use a *migration-based GA*. This model of parallelization introduces a new migration operator that is used to send individuals from one subpopulation to another. The migration operator can be defined in various ways. Classically, one tends to migrate individuals to "neighboring" processors. This approach is not very pertinent in multicomputers today, since the diameter of the network interconnecting the processors is reasonably small, and the distance between processors can be assumed to be constant, to a reasonable approximation. We have therefore introduced a randomized migration operator in two of our algorithms so that a processor may communicate with any arbitrary processor, to disperse information faster among the processors. This migration operator introduces a new dimension of parallel search into the algorithm and opens up exciting possibilities. It is possible that the parallel GA will terminate faster with the exchange of information between processors. It is also possible that one might be able to outperform the sequential GA in obtaining a better quality solution. A third approach to parallelization is to use a *subpopulation-based GA*. Here, the population is divided into a few subpopulations, which are relatively isolated from each other. A number of studies have been made on approaches which use subpopulation-based GAs, both with and without migration [60, 61, 62]. Using subpopulations without migration is equivalent to performing simultaneous runs of the sequential algorithm on the subpopulations. Since there is no exchange of information, this approach is not very interesting. The execution time and results are the same as those for the execution of the sequential GA on a subpopulation. Hence, we shall only consider subpopulation-based GA's with migration. This introduces two dimensions of parallelism at once. The first dimension is that of migration of information between processors. The second dimension is that now each processor only works on a subpopulation; and hence, it has to evaluate the fitness of fewer individuals. One could get superlinear speedups with this approach due to the two dimensions of parallelism, but the quality of the solution may degrade if population sizes become too small. We now proceed to a detailed description of the proposed parallel genetic algorithms for sequential circuit test generation.

6 Parallel Algorithms for GATEST

Previous parallel approaches to test generation have focussed on parallelizing deterministic algorithms for testing, as was described briefly earlier in Section 3.2. A review of parallel genetic algorithms can be found in [57]. In GATTO* [54], a parallel genetic algorithm was presented which is similar to PGATEST1, and the differences between these two algorithms will be highlighted in the discussion that follows. Figure 5 shows a graphical outline of the approaches used in each of the algorithms for 4 processors.

6.1 PGATEST1: Parallelization using data decompostion

In this approach, we parallelize the loop which evaluates the fitness of all individuals in a given population, as illustrated in Figure 5(a). Each processor maintains its own copy of the entire population. The work of evaluating the fitness is equally and statically distributed over the processors. The fault list is the same on all processors. Hence, if we have N individuals and P processors, we assign $\frac{N}{P}$ individuals to each processor. The fitness values computed by the respective processors are communicated to a single processor, which collects this information and broadcasts it to all processors. Each processor now has the information regarding the fitness values of all individuals and can evolve the next generation and compute the best individual in the population for the current generation. After processing for the last generation is completed, the best individual is added to the test set if it detects one or more faults out of the current set of 31 faults targeted. Each processor generates its own test set, and the test sets for all processors are identical, since the random number generators for all processors are initialized with the same seed. An exhaustive fault simulation is then done by all processors using this best-fit individual.

The final results in terms of the number of faults detected and the test set generated are identical to those obtained on a uniprocessor when this algorithm is used. Therefore, there is no degradation in the quality of the solution obtained through parallelization. This approach is similar to the one used in GATTO* [54]. One major difference is that, while we target 31 faults at a time using a fault parallel approach, only one fault is targeted at a time in GATTO*. Secondly, the same fitness function is used at all times in GATTO*, which may slow down execution. Thirdly, GATTO* required an additional central processor, which is not required in our implementation.

6.2 PGATEST2: Parallel migration based genetic search

In this algorithm, each processor starts with a different random seed and therefore will have different initial populations to begin with. Each processor executes the serial GATEST algorithm primarily. However, processors interact with each other by communicating the fittest individuals to one another, at an interval determined by the epoch for the parallel GA. In our implementation, a highly fit individual is communicated to a random processor whenever it is available. Hence, the epoch value for our implementation is 1. On any processor, when new individuals are introduced into the GA population on that processor, a nonblocking probe is used to determine if there are any pending messages from other processors containing good individuals. All pending messages are read, and these individuals are inserted into the population on that processor. After the probe, the execution continues as in the serial genetic algorithm.

The hope is that with $migration$ of information between processors which are exploring different parts of the search space, there will be an improvement in the overall solution (the number of faults detected will be greater, the overall test set size will be smaller, etc.). This algorithm is illustrated in Figure 5(b). We always assign the same number of individuals to each processor that are assigned in the sequential algorithm, irrespective of the number of processors. Hence, each processor is assigned the same workload as in the sequential case. Each processor is now following a different search path. There is now a possibility that the processors will finish faster due to the migration of information between them. This is indeed the case in our implementation as processors converge to their results faster with the added advantage that the quality of the result is improved in certain cases.

Each processor maintains an independent fault list and proceeds completely independently, generating its own test set. Periodically and asynchronously, each processor transmits the individual with the best fitness to a random processor. A queue of messages is maintained on each processor to receive migrating individuals from other processors. These individuals are absorbed into the local subpopulation as it evolves. This gives rise to the possibility of obtaining P different solutions to the problem using a randomized parallel genetic search.

6.3 PGATEST3: Subpopulation-based GA with migration

This approach is similar to the previous approach, except that each processor starts with a population of $M = \frac{N}{S}$ individuals, where S is a positive integer greater than 1. This algorithm is illustrated in Figure 5(c). One can expect speedups for two reasons. Each processor works on a subpopulation; therefore, the population size being used is smaller, and each processor individually has less work to do. Also, due to migration of fit individuals from one processor to another, each processor can detect faults faster than if they were to run independent GAs with a reduced population size.

One possible disadvantage of this approach is that, for a circuit with a relatively small population size (corresponding to a circuit with small number of primary inputs and small sequential depth), the algorithm is not very scalable. If the population size for the GA becomes too small, one can expect a degradation in the results obtained. But if the population size is large, as is the case for large circuits, then this should not be problem. Therefore, one should be careful in choosing the value of the scaling factor S to avoid affecting the result in order to obtain the benefit of a subpopulation approach coupled with migration.

6.4 FPARTEST: A fault partitioned approach

One could also have a traditional fault partitioned approach for parallel test generation which we will call FPARTEST. In this algorithm, the faults are statically partitioned across the processors, for the purpose of test generation, by cyclically distributing the faults across the processors. Each processor attempts to generate tests, using the GATEST algorithm, for faults allocated to its partition. A list of all undetected faults on all processors is also maintained on each processor. If a test sequence for a fault is found on one processor, fault simulation on the entire list of undetected faults is performed, and all detected faults after fault simulation are broadcast to all processors. Every processor drops any faults thus found from its fault list. The overall sequence length is obtained by concatenating the sequences obtained on each processor.

7 Analysis of Parallel Algorithms for GATEST

We now present a brief analysis of the proposed parallel algorithms for GATEST to provide an insight into these algorithms.

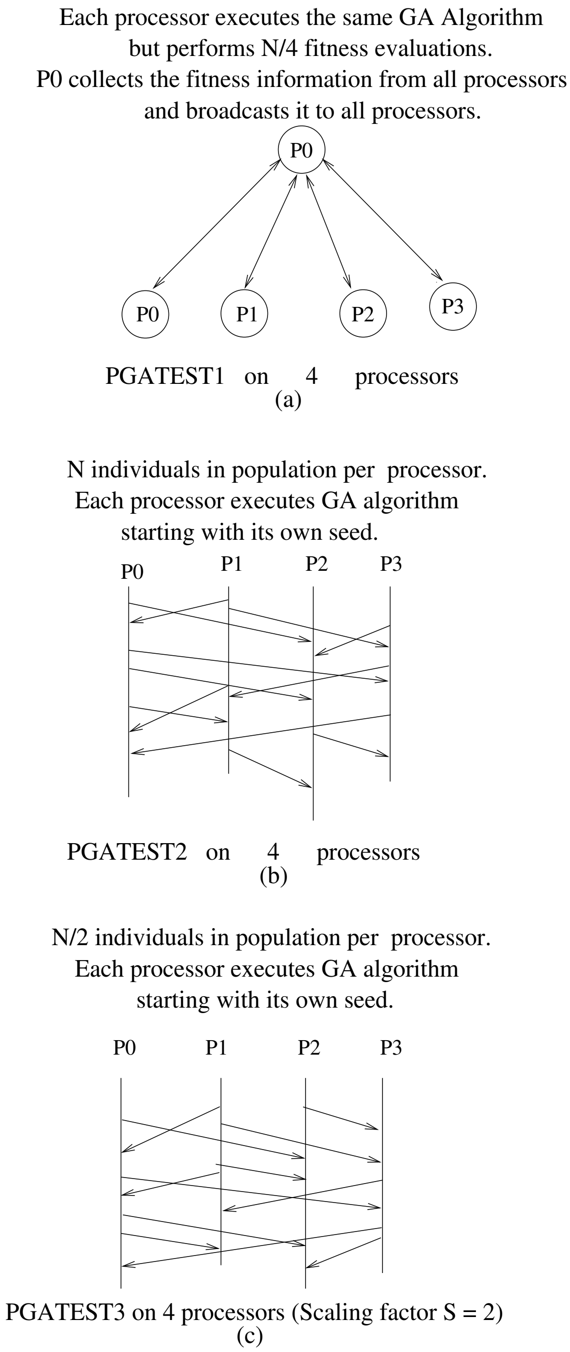

Figure 5: Comparison of Three Proposed Parallel Algorithms

7.1 Analysis of PGATEST1

Let us assume that we have a population size of N in any generation and P processors. If $S(i)$ is the sequence length of an individual i, and LS is the cost of logic simulation per time frame, then the actual cost of evaluating the fitness of an individual i is given approximately by $S(i) * LS$, and the cost of exhaustive fault simulation $Y(i)$ for individual i is given approximately by $S(i) * \lceil \frac{RF(m)}{31} \rceil * LS$, where $RF(m)$ is the number of faults remaining in the fault list at step m of the simulation process. Assume that we take g generations on the average to find an individual which can be used for the exhaustive fault simulation. Also, assume that we have k individuals eventually forming the entire test sequence (i.e., the overall simulation process has k steps). Let us assume for the rest of the analysis that the cost of evaluating the fitness of an individual is X seconds, independent of i. It must be noted that this assumption is only approximate, since the fitness evaluation of an individual depends on the activity of the flip-flops/gates in the circuit and on the sequential depth. Let us also assume that the exhaustive fault simulation using the entire test sequence with k individuals takes Y seconds. Then the sequential execution time on a uniprocessor is

$$T_1(N) = kgNX + Y.$$

If the cost of communication involved is ignored, the parallel execution time on P processors for algorithm PGATEST1 would be approximately given by

$$T_p(N) = kg\frac{N}{P}X + Y.$$

Let us assume that α is the startup cost in seconds associated with sending a message and that β is the reciprocal of the bandwidth of the communication network in seconds per word on a distributed memory multicomputer. Let us assume that an integer takes b bytes of memory (b is usually 4 or 8). Since each processor works on $\frac{N}{p}$ individuals, it has to communicate the integer corresponding to the index of each individual in the population and the fitness of each individual to processor 0. The cost for this communication can be approximated by $P(\alpha + \beta 2b(\frac{N}{P}))$. The communication of the best fit individual to processor 0 occurs after g generations. After this, processor 0 collects the fitness and index of each individual that it receives and broadcasts it to all processors. The cost for this is given approximately by $log(P)(\alpha + \beta 2bN)$.

Hence the overall execution parallel execution time, including the communication costs, is given by

$$T_p(N) = k(g\frac{N}{P}X + \alpha(P + log(P)) + \beta 2bN(1 + log(P)) + Y.$$

The above model is a bit simplistic in that it has not taken into account the increase in the population size that occur during the successive stages of GA evolution in the algorithm and assumes that any issues related to the sequential depth of the circuit are buried in the terms X and Y. Effectively, the parallelism available depends on how large the term $kgNX$ is, compared to the term Y and to the communication costs involved.

7.2 Analysis of PGATEST2

To capture the gain obtained through parallel search on P processors using migration, we will introduce a function $\kappa(P)$, which effectively reduces the average number of generations required from g to $\frac{g}{\kappa(P)}$. Then the execution time on P processors for the algorithm PGATEST2 is given by

$$T_p(N) = k\frac{g}{\kappa(P)}NX + Y.$$

We have ignored the asynchronous communication costs for migration of individuals between processors in the above formula. The reason for this is because the algorithm is asynchronous, and the only communication overhead is mainly in packing while sending a message, in probing for a message, and in unpacking it while receiving any message, with no synchronization costs or waiting costs. There is certainly some overhead involved in doing this, but one can ignore this overhead in comparison to the computation costs involved. It should also be noted that there will be a different solution available on each processor; hence, the values of k and Y will differ from one processor to another. In our implementation, we take the maximum of the execution times over all processors to be the total parallel execution time, and we pick the best among the P solutions available.

7.3 Analysis of PGATEST3

Since the population assigned to each processor is now $\frac{N}{S}$, and since we have migration, the parallel execution time on P processors for the algorithm PGATEST3 is given by

$$T_p(N, S) = k \frac{g}{\kappa(P)} \frac{N}{S} X + Y.$$

We can see that the parallel execution time reduces due to migration and due to the smaller population size on each processor.

7.4 Analysis of FPARTEST

If all faults are equally difficult to test, a parallel algorithm such as FPARTEST may provide linear speedups. However, in general, some faults are hard-to-detect while others may be easy-to-detect and it is difficult to predict prior to test generation regarding the degree of difficulty involved in testing each fault. Hence, static fault partitioning can create load imbalance, which is circuit-dependent, across the processors resulting in lower speedups. The results will show that a fault partitioned approach in general can only provide small or moderate speedups due to this load imbalance.

It should be noted that the parallel genetic algorithms discussed above are non-deterministic in nature. It is difficult to map the parameters assumed in the above analysis to the experimental results that will follow due to this non-determinism involved. Therefore we will not make an attempt to obtain such a mapping between the theoretical analysis and the experimental results. The analysis was presented to give a deeper insight into each of the proposed algorithms. We will demonstrate nevertheless that the experimental results validate the trends predicted by the above analysis.

8 Experimental Results on Parallel Algorithms for GATEST

The algorithms PGATEST1, PGATEST2, and PGATEST3 were implemented using the MPI communication library [2]. As a comparison, a parallel fault partitioned approach called FPARTEST was also implemented. All implementations are portable to a wide variety of shared-memory and distributed-memory parallel platforms. These algorithms were first developed [64] using the ProperCAD II library [65], which is built around the Actor paradigm [66] of concurrent object-oriented computing, and provides a portable, parallel object-oriented platform for the development of parallel algorithms for VLSI-

CAD applications. The experimental results presented here obtained with implementations developed using the the MPI communication library [2] provide a more detailed understanding of the various proposed parallel algorithms compared to the earlier results in [64]. Results are shown on a 6 processor SUN E3000 shared memory multiprocessor with 1024 MB of memory and on a 16 processor IBM SP2 distributed memory multicomputer with 256 MB of memory per node.

Each of the parallel genetic algorithms on completion was followed by a phase of PHITEC, a parallel version of HITEC [15], executed in parallel, to drop any redundant faults among the faults left undetected by the genetic algorithm. PHITEC was implemented in parallel using the MPI communication library and in this algorithm, the undetected faults are statically partitioned across the processors and a short time-out of 1 second is assigned per fault. Many redundant faults are found in parallel using this approach.

All three parallel genetic algorithms execute in the same fashion on a single processor. The circuits s1196, s1238, s1494, s1488 and s1423 were taken from the ISCAS89 benchmark suite [55]. A collapsed fault list for each circuit was taken as the initial fault list. All execution times are in seconds. In the tables, "Fault Det" refers to the number of faults detected while "Seq Len" refers to the length of the overall test vector sequence generated. It must be noted that this sequence length is obtained by truncating the final sequence if necessary. This is done by dropping the last few vectors in the final sequence which do not contribute to any new fault detections, if such a set of vectors exist.

The algorithm PGATEST1 was compared with a traditional fault partitioned approach which we will call FPARTEST. In the algorithm FPARTEST, the faults are statically partitioned across the processors, for the purpose of test generation, by cyclically distributing the faults across the processors. Each processor attempts to generate tests, using the GATEST algorithm, for faults allocated to its partition. A list of all undetected faults on all processors is also maintained on each processor. If a test sequence for a fault is found on one processor, fault simulation on the entire list of undetected faults is performed, and all detected faults after fault simulation are broadcast to all processors. Every processor drops any faults thus found from its fault list. The overall sequence length is obtained by concatenating the sequences obtained on each processor. The results are shown in Table 1. One can see that the speedups are low with a static fault partitioning approach such as FPARTEST. The fault coverages are also low, and the sequence lengths are high compared to PGATEST1. From the results, we see that PGATEST1 has certainly provided better results than a traditional fault partitioned approach to parallel test generation.

Table 2 shows the results of execution of PGATEST1 followed by PHITEC on a single processor of the SUN E3000 shared-memory multiprocessor. Table 3 shows the performance of PGATEST1 followed by PHITEC on all 6 processors of the SUN E3000 shared-memory multiprocessor. One can see that the results are the same as in the uniprocessor case with a significant reduction in the execution time for all circuits. One can also see that most of the execution time is taken up by the test generation step PGATEST1, and that the redundancy identification step with PHITEC takes a smaller fraction of the execution time usually. However, if a large number of faults are left undetected during the test generation phase (PGATEST1), then the redundancy identification phase (PHITEC) may also take up a substantial yet small amount of time. Table 4 shows the speedups on 6 processors of the shared-memory multiprocessor for the test generation phase (PGATEST1), for the redundancy identification phase (PHITEC) and for the overall approach (PGATEST1-PHITEC).

We will now focus on the three parallel genetic algorithms proposed and compare results for these algorithms on a 16 processor IBM SP2, and we shall disregard the redundancy identification phase for further discussions.

Table 5 shows the performance of the parallel algorithm PGATEST1 on 16 processors of the IBM SP2. Results are shown for 1, 8 and 16 processors. It can be seen that the algorithm scales better with increase in the size of the circuit used. From Table 5, we can once again observe that for the algorithm PGATEST1, the results remain unchanged from the uniprocessor run and that execution times have been reduced significantly.

Table 6 shows the performance of PGATEST2 on 16 processors of the IBM SP2. The reduction in execution time obtained with the algorithm PGATEST2 is purely due to the migration of fit individuals from one processor to another. The algorithm PGATEST2 tends to take longer to execute than PGATEST1 but provides better or matching results with respect to the quality in terms of the fault coverage obtained, when compared to PGATEST1.

Two experiments were performed with the algorithm PGATEST3 using 16 processors of the IBM SP2. In the first experiment the population size was scaled down by a term equal to the number of processors. In the second experiment, the population size was scaled down by 8 which was equal to half the number of processors. The algorithm PGATEST3, whose performance can be seen in Table 7, provides excellent execution times, but the test set sizes are larger than the previous algorithms. It should be noted that with the first experiment with PGATEST3 there is a degradation in the quality of the result obtained in terms of the fault coverage. This can be attributed to the fact that the population size was reduced to a very small number which drove the algorithm to a local minimum very early. The results for the second experiment with PGATEST3, where the population size was scaled by a factor of 8, provide excellent results in terms of lower execution times and in terms of fault coverage. This is due to the fact that we

Table 1: Comparison of PGATEST1 with FPARTEST on 6 Processors of the SUN E3000

Circuit	Faults	PGATEST1			FPARTEST		
		Faults Det	Seq Len	Speedup	Faults Det	SeqLen	Speedup
s1196	1242	1233	423	5.8	1224	654	2.3
s1238	1355	1278	426	5.1	1265	847	2.5
s1494	1506	1377	276	4.9	1358	588	1.9
s1488	1486	1314	261	4.8	1282	476	2.7
am2910	2391	2185	600	5.0	2170	862	2.4
mult16	1708	1649	180	2.6	1594	448	2.5
s1423	1515	1238	1108	4.6	1219	1462	2.4
pcont2	11300	6829	221	4.2	6724	869	1.9

Table 2: PGATEST1-PHITEC: Uniprocessor Run on the SUN Ultra Enterprise 3000

Circuit	Faults	PGATEST1		PHITEC		Total	Test	Fault	Fault
		Faults Det	Time	Faults Red	Time	Time	Seq Len	Coverage	Efficiency
s1196	1242	1233	116	3	0.07	116	423	0.992	0.995
s1238	1355	1278	231	72	1.1	232	426	0.943	0.996
s1494	1506	1377	151	27	81	232	276	0.914	0.932
s1488	1486	1314	201	14	156	357	261	0.884	0.893
am2910	2391	2185	709	168	42	751	600	0.914	0.984
mult16	1708	1649	378	12	61	439	180	0.965	0.972
s1423	1515	1238	8811	11	279	9090	1108	0.817	0.824
pcont2	11300	6829	10497	2765	2024	12521	221	0.604	0.849

have taken proper advantage of migration and the scaling of the population size without compromising on the quality of the results in terms of the fault coverage.

Table 8 shows the speedups obtained on 16 processors with the different parallel genetic algorithms. It can be seen that the highest speedups were obtained when the population size was scaled down by a factor of 16. The speedups were superlinear occassionally since the population size was low and migration was used, but the quality of the result has been compromised with this approach.

One common feature of all the migration based approaches presented in Tables 6 and 7 is that the test set sizes seem to be much larger than those obtained with PGATEST1. However, the fault coverage is the most important metric in test generation. Hence, the fact that better quality results in terms of fault coverage, and in addition better speedups, are obtained using migration based approaches offsets the fact that the test set size has increases.

From the above discussion of the results, it can be seen that the best results are obtained when the population size is scaled by a moderate factor and when migration is employed.

Table 3: PGATEST1-PHITEC: Six Processor Run on the SUN Ultra Enterprise 3000

		PGATEST1		PHITEC		Total	Test	Fault	Fault
Circuit	Faults	Faults Det	Time	Faults Red	Time	Time	Seq Len	Coverage	Efficiency
s1196	1242	1233	20	3	0.02	20	423	0.992	0.995
s1238	1355	1278	45	72	0.3	45	426	0.943	0.996
s1494	1506	1377	31	27	17	48	276	0.914	0.932
s1488	1486	1314	42	14	28	70	261	0.884	0.893
am2910	2391	2185	143	168	8	151	600	0.914	0.984
mult16	1708	1649	147	12	14	161	180	0.965	0.972
s1423	1515	1238	1931	11	50	1981	1108	0.817	0.824
pcont2	11300	6829	2526	2765	348	2874	221	0.604	0.849

Table 4: Speedups on 6 Processors of the SUN Ultra Enterprise 3000

Circuit	PGATEST1	PHITEC	PGATEST1-PHITEC
s1196	5.8	3.5	5.8
s1238	5.1	3.7	5.1
s1494	4.9	4.8	4.8
s1488	4.8	5.6	5.1
am2910	5.0	5.3	5.0
mult16	2.6	4.4	2.7
s1423	4.6	5.6	4.6
pcont2	4.2	5.8	4.4

Table 5: PGATEST1 on the IBM SP2

		One Processor			8 Processors			16 Processors		
Circuit	Faults	Faults Det	Time	Seq Len	Faults Det	Time	Seq Len	Faults Det	Time	Seq Len
s1196	1242	1233	210	423	1233	54	423	1233	38	423
s1238	1355	1278	421	426	1278	103	426	1278	81	426
s1494	1506	1377	293	276	1377	63	276	1377	45	276
s1488	1486	1314	384	261	1314	84	261	1314	63	261
am2910	2391	2185	1319	600	2185	297	600	2185	231	600
mult16	1708	1649	735	180	1649	174	180	1649	145	180
s1423	1515	1238	17396	1108	1238	4974	1108	1238	2568	1108
pcont2	11300	6829	20385	221	6829	2819	221	6829	1358	221

Table 6: PGATEST2 on 16 Processors of the IBM SP2

Circuit	Faults	One Processor			16 Processors		
		Faults Det	Time	Seq Len	Faults Det	Time	Seq Len
s1196	1242	1233	210	423	1234	70	583
s1238	1355	1278	421	426	1279	117	996
s1494	1506	1377	293	276	1406	56	310
s1488	1486	1314	384	261	1389	135	242
am2910	2391	2185	1319	600	2188	619	664
mult16	1708	1649	735	180	1649	212	265
s1423	1515	1238	17396	1108	1238	6827	1206
pcont2	11300	6829	20385	221	6829	9746	341

Table 7: PGATEST3 on 16 Processors of the IBM SP2

Circuit	Faults	One Processor			16 Processors (S=16)			16 Processors (S=8)		
		Faults Det	Time	Seq Len	Faults Det	Time	Seq Len	Faults Det	Time	Seq Len
s1196	1242	1233	210	423	1214	32	1231	1233	34	639
s1238	1355	1278	421	426	1271	49	998	1278	56	793
s1494	1506	1377	293	276	1392	21	245	1399	30	275
s1488	1486	1314	384	261	1361	20	212	1370	28	224
am2910	2391	2185	1319	600	2140	82	749	2185	121	692
mult16	1708	1649	735	180	1640	43	220	1648	97	308
s1423	1515	1238	17396	1108	1219	449	1013	1238	1260	1162
pcont2	11300	6829	20385	221	6807	542	374	6829	1283	328

Table 8: Speedups on 16 Processors of the IBM SP2

Circuit	PGATEST1	PGATEST2	PGATEST3	PGATEST3
			S=16	S=8
s1196	5.5	3.0	6.6	6.17
s1238	5.2	3.6	8.6	7.5
s1494	6.5	5.2	13.9	9.8
s1488	6.1	2.8	19.2	13.7
am2910	5.7	2.1	16.1	10.9
mult16	5.1	3.5	17.1	7.6
s1423	6.8	2.5	38.7	13.8
pcont2	15.0	2.1	37.6	15.8

9 Summary

In this paper, we have presented three parallel genetic algorithms, PGATEST1, PGATEST2, and PGATEST3, for simulation-based sequential circuit test generation. PGATEST1 is a good algorithm in general, as it provides significant speedups without degradation in the quality of the results. It exploits the parallelism available corresponding to the evaluation of fitness values of the individuals in a population. PGATEST2 exploits the search parallelism available through parallel randomized genetic search with migration of information. This algorithm attempts to improve the quality of the results, but it tends to have limited scalability. This means that migration of information can provide improvement in the quality of a result and in performance but only to a limited degree. PGATEST3 provides a dual degree of parallelism by using a subpopulation-based GA approach to reduce the workload among the processors and by exploiting the benefits of the randomized migration strategy used. However, there is a likelihood of degradation in the quality of the result for this algorithm for a large number of processors if the population size is scaled by a very high factor. One must therefore ensure that the population size is scaled reasonably to get the best results, and that one is not too ambitious when scaling the population down in a subpopulation-based approach with migration.

We have therefore proposed and successfully demonstrated the usefulness of various parallel genetic algorithms for the highly computationally intensive practical problem of automatic test pattern generation for sequential circuits. It is hoped that the techniques used in the proposed parallel algorithms can be successfully extended to many other practical and computationally expensive tasks.

References

[1] T. E. Marchok, Aiman El-Maleh, W. Maly, and J. Rajski, “Complexity of sequential ATPG,” in *Proc. European Design and Test Conf.*, March 1995, pp. 252–261.

[2] W. Gropp, E. Lusk, and A. Skellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface.* Cambridge, MA: MIT Press, 1994.

[3] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*. New York, NY: Computer Science Press, 1990.

[4] I. Pomeranz and S. M. Reddy, “Classification of faults in synchronous sequential circuits,” *IEEE Transactions on Computers*, vol. 42, no. 9, pp. 1066–1077, 1993.

[5] K.-T. Cheng and H.-K. T. Ma, “On the over-specification problem in sequential ATPG algorithms,” *IEEE Transactions on Computer-Aided Design*, vol. 12, no. 10, 1599–1604, 1993.

[6] Irith Pomeranz, Sudhakar M. Reddy, and Janak H. Patel, “Theory and practice of sequential machine testing and testability,” in *Proc of the International Symposium on Fault Tolerant Computing*, June 1993, pp. 330-337.

[7] T. M. Niermann, W. -T. Cheng, and J. H. Patel, “PROOFS: A fast, memory-efficient sequential circuit fault simulator,” *IEEE Trans. Computer-Aided Design*, vol. 11, no. 2, pp. 198–207, Feb. 1992.

[8] S. Seshu and D. N. Freeman, “The diagnosis of asynchronous sequential switching systems,” *IRE Trans. Electronic Computing*, vol. 11, pp. 459–465, Aug. 1962.

[9] P. Goel, “An implicit enumeration algorithm to generate tests for combinational logic circuits,” *IEEE Trans. Computers*, vol. C-30, no. 3, pp. 215–222, March 1981.

[10] R. Marlett, “An effective test generation system for sequential circuits,” in *Proc. Design Automation Conf.*, June 1986, pp. 250–256.

[11] W. -T. Cheng, “The BACK algorithm for sequential test generation,” in *Proc. Int. Conf. Computer Design*, Oct. 1988, pp. 66–69.

[12] H. -K. T. Ma, S. Devadas, A. R. Newton, and A. Sangiovanni-Vincentelli, “Test generation for sequential circuits,” *IEEE Trans. Computer-Aided Design*, vol. 7, no. 10, pp. 1081–1093, Oct. 1988.

[13] M. H. Schulz and E. Auth, “Essential: An effection self-learning test pattern generation algorithm for sequential circuits,” in *Proc. Int. Test Conf.*, Aug. 1989, pp. 28–37.

[14] A. Ghosh, S. Devadas, and A. R. Newton, “Test generation for highly sequential circuits,” in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1989, pp. 362–365.

[15] T. M. Niermann and J. H. Patel, “HITEC: A test generation package for sequential circuits,” in *Proc. European Conf. Design Automation (EDAC)*, Feb. 1991, pp. 214–218.

[16] D. H. Lee and S. M. Reddy, “A new test generation method for sequential circuits,” in *Proc. Int. Conf. Computer-Aided Design*, Nov, 1991, pp. 446–449.

[17] M. A. Breuer, “A random and an algorithmic technique for fault detection test generation for sequential circuits,” *IEEE Trans. Computers*, vol. 20, no. 11, pp. 1364–1370, Nov. 1971.

[18] H. D. Schnurmann, E. Lindbloom, and R. G. Carpenter, “The weighted random test-pattern generator,” *IEEE Trans. Computers*, vol. 24, no. 7, pp. 695–700, July 1975.

[19] R. Lisanke, F. Brglez, A. J. Degeus, and D. Gregory, “Testability-driven random test-pattern generation,” *IEEE Trans. Computer-Aided Design*, vol. 6, no. 6, pp. 1082–1087, Nov. 1987.

[20] V. D. Agrawal, K. T. Cheng, and P. Agrawal, “A directed search method for test generation using a concurrent simulator,” *IEEE Trans. Computer-Aided Design*, vol. 8, no. 2, pp. 131–138, Feb. 1989.

[21] T. J. Snethen, “Simulator-oriented fault test generator,” in *Proc. Design Automation Conf.*, 1977, pp. 88–93.

[22] H.-J. Wunderlich, “Multiple distributions for biased random test patterns,” *IEEE Trans. Computer-Aided Design*, vol. 9, no. 6, pp. 584–593, June 1990.

[23] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning,* Reading, MA: Addison-Wesley, 1989.

[24] D. G. Saab, Y. G. Saab, and J. A. Abraham, “CRIS: A test cultivation program for sequential VLSI circuits,” in *Proc. Int. Conf. Computer-Aided Design*, 1992, pp. 216-219.

[25] M. Srinivas and L. M. Patnaik, "A simulation-based test generation scheme using genetic algorithms," in *Proc. Int. Conf. VLSI Design*, 1993, pp. 132-135.

[26] P. Prinetto, M. Rebaudengo, and M. Sonza Reorda, "An automatic test pattern generator for large sequential circuits based on genetic algorithms," in *Proc. Int. Test Conf.*, Oct. 1994, pp. 240–249.

[27] E. M. Rudnick, J. G. Holm, D. G. Saab, and J. H. Patel, "Application of simple genetic algorithms to sequential circuit test generation," in *Proc. European Design and Test Conf.*, Feb. 1994, pp. 40–45.

[28] E. M. Rudnick, J. H. Patel, G. S. Greenstein, and T. M. Niermann, "Sequential circuit test generation in a genetic algorithm framework," in *Proc. Design Automation Conf.*, June 1994, pp. 698–704.

[29] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, "Alternating strategies for sequential circuit ATPG," in *Proc. European Design and Test Conf.*, 1996, pp. 368-374.

[30] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, "Automatic test generation using genetically-engineered distinguishing sequences," in *Proc. VLSI Test Symp.,* 1996, pp. 216-223.

[31] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, "Automatic test generation using genetically-engineered distinguishing sequences," in *Proc. VLSI Test Symp.,* 1996, pp. 216–223.

[32] F. Corno, P. Prinetto, M. Rebaudengo, and M. Sonza Reorda, "Comparing topological, symbolic and GA-based ATPGs: an experimental study," in *Proc. Int. Test Conf.*, 1996, pp. 39-47.

[33] I. Pomeranz and S. M. Reddy, "On improving genetic optimization based test generation," in *Proc. European Design and Test Conf.,* 1997, pp. 506–511.

[34] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, "Sequential circuit test generation using dynamic state traversal," in *Proc. European Design and Test Conf.*, 1997, pp. 22-28.

[35] P. Banerjee, *Parallel Algorithms for VLSI Computer-aided Design Applications*, Prentice Hall, Englewoods-Cliffs, NJ, 1994, pp. 477-590.

[36] R. Klenke, R. D. Williams and J. Aylor, "Parallel Processing Techniques for Automatic Test Pattern Generation," *IEEE Computer* pp. 71-84, Jan. 1992.

[37] S. Chandra and J. H. Patel, "Test generation in a parallel processing environment," in *Digest of Papers, Int. Conf. Computer Design*, Oct. 1988, pp. 11-14.

[38] S. Patil, P. Banerjee and J. Patel, "Parallel test generation for sequential circuits on general purpose multiprocessors," in *Proc. Design Automation Conference*, June 1991, pp. 155-159.

[39] P. Agrawal, V. D. Agrawal, and J. Villoldo, "Sequention circuit test generation on a distributed system," in *Proc. Design Automation Conf.*, 1993, pp. 107-111.

[40] H. Fujiwara and T. Inoue, "Optimal granularity of test generation in a distributed system," *IEEE Trans. on Computer-Aided Design* , pp. 885-892, Aug. 1990.

[41] B. Ramkumar and P. Banerjee, "Portable parallel test generation for sequential circuits," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1992, pp. 220-223.

[42] S. Parkes, P. Banerjee and J. Patel, "ProperHITEC: A portable, parallel, object-oriented approach to sequential test generation," in *Proc. Design Automation Conf.*, June 1994, pp. 717-721.

[43] P. Krauss, "Parallel automatic test pattern generation using large workstation networks,", University of Munich, Dept. of Electrical Engineering, Munich, Germany, Tech. Report, March 1993.

[44] S. Arvindam, V. Kumar, V. N. Rao, and V. Singh, "Automatic test pattern generation on parallel processors," Computer Science Dept., University of Minnesota, Minneapolis, MN, Tech. Report TR-90-29, May 1994.

[45] S. Venkatraman, S. Seth, and P. Agrawal, "Parallel test generation using boolean satisfiability," University of Nebraska, Dept. of Electrical Engineering, Lincoln, NB, Tech. Report CSE 92-023, Dec. 1992.

[46] S. T. Patel and J. H. Patel, "Effectiveness of heuristic measures for automatic test pattern generation," in *Proc. Design Automation Conf.*, June 1986, pp. 547-552.

[47] J. S. Conery, "The AND/OR process model for parallel interpretation of logic programs," Department of Computer Science, University of California, Irvine, CA, Tech. Report UCI-CS-83-204, June 1983.

[48] S. Patil, P. Banerjee, "A parallel branch and bound approach to test generation," *IEEE Trans. Computer-Aided Design of Circuits and Systems*, vol. 9, no. 3, pp. 313-322, March 1990.

[49] A. Motohara, K. Nishimura, H. Fujiwara, and I. Shirakawa, "A parallel scheme for test pattern generation," in *Proc. Int. Conf. Computer-Aided Design*, 1986, pp. 156-159.

[50] S. Patil, "Parallel algorithms for test generation and fault simulation," Coordinated Science Laboratory, University of Illinois, Urbana, IL, Tech. Report, Sept. 1990.

[51] R. H. Bell, R. H. Klenke, J. Aylor, and R. D. Williams, "Results of a topologically partitioned parallel automatic test pattern generation system on a distributed memory multiprocessor," in *Proc. Application Specific Integrated Circuits Conf.*, Sept. 1992.

[52] G. A. Kramer, "Employing massive parallelism in digital ATPG algorithms," in *Proc. Int. Test Conf.*, 1993, pp. 108-114.

[53] T. Chan, "Distributed automatic test pattern generation," in *Proc. Application Specific Integrated Circuits Conf.*, Sept. 1992.

[54] F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, and E. Veiluva, "A portable ATPG tool for parallel and distributed systems," in *Proc. VLSI Test Symp.*, 1995, pp. 29-34.

[55] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *Proc. Int. Symp. Circuits and Systems*, May 1989, pp. 1929–1934.

[56] J. Grefenstette, "Parallel adaptive algorithms for function optimization," Vanderbilt University, Nashville, TN, Tech. Report CS-81-19, 1981.

[57] P. Adamidis, "Review of parallel genetic algorithms bibiliography," Aristotle University of Thessaloniki, Thessaloniki, Greece, Tech. Report, 1994.

[58] V. Gordon, and D. Whitley, "Serial and parallel genetic algorithms as function optimizers," in *Proc. Fifth Int. Conf. Genetic Algorithms*, pp. 177-183, San Mateo, CA:Morgan Kaufamann Publishers, 1993.

[59] S. C. Lin, W. Punch, and E. Goodman, "Coarse-grain parallel genetic algorithms: Categorization and new approach," *Sixth IEEE Symp. Parallel and Distributed Processing*, Los Alamitos, CA, Oct. 1994.

[60] H. Muhlenbein, M. Schomish, J. Born, "The parallel genetic algorithm as a function optimizer," in *Proc. Fourth Int. Conf. Genetic Algorithms*, 1991, pp. 271-278.

[61] R. Tanese, “Distributed genetic algorithms,” in *Proc. Third Int. Conf. Genetic Algorithms*, 1989, pp. 398-405.

[62] Harik, G., Cantz-Paz, E., Goldberg, D., and Miller, B., “The Gambler’s Ruin Problem, Genetic Algortihms, and the Sizing of Populations,” in *Proc. of the IEEE Int. Conference on Evolutionary Computation*, 1997, pp. 7-12.

[63] E. M. Rudnick, “Simulation-based Techniques for Sequential Circuits,” Ph.D. dissertation, University of Illinois at Urbana-Champaign, Technical Report CRHC-94-14, Aug. 1994.

[64] D. Krishnaswamy, M. S. Hsiao, V. Saxena, P. Banerjee, E. M. Rudnick and J. H. Patel, “Parallel genetic algorithms for simulation-based sequential circuit test generation,” in *Proc 10th Intl. Conf. VLSI Design*, 1997, pp. 475-481.

[65] S. Parkes, J. A. Chandy, and P. Banerjee, “A Library-based approach to portable, parallel, object-oriented programming: Interface, implementation and application,” in *Proc. Supercomputing’94*, 1994, pp. 69-78.

[66] G. A. Agha, *Actors: A model of concurrent computation in distributed systems*. Cambridge MA: The MIT Press, 1986.