High-Performance Testing: Parallelizing Functional Tests for Computer Systems Using Distributed Graph Exploration

Alexey Demakov, Alexander Kamkin, and Alexander Sortov
Institute for System Programming of the Russian Academy of Sciences (ISPRAS)
Moscow, Russia
{demakov,kamkin,sortov}@ispras.ru

Abstract—**Functional testing of complex hardware and software systems has long been recognized as an immensely computer-intensive task. Consisting of a huge number of interacting components, computer systems are hard to be verified due to the well-known fundamental problem – combinatorial state explosion. One of the ways to overcome the complexity is to use abstract models for generating test sequences and checking design correctness. However, models of really complex systems are complex themselves, which leads to enormously long test sequences (tests are usually targeted at covering all model states reachable from the initial one). In this paper, we suggest a method for high-performance generation and execution of model-based tests based on the distributed exploration of a system's graph model. The key feature of the method is that parallelization is done dynamically and fully transparently for a user.**

Keywords–functional testing; simulation-based verification; combinatorial state explosion; distributed computing; graph exploration; UniTESK technology

I. Introduction

Many researchers are familiar with the term *high-performance computing* designating a branch of knowledge about using supercomputers and computer clusters to solve advanced computation problems [1]. The term is mostly associated with the computing used for scientific research or computational science (modeling of physical processes, materials, structures, and so on). However, when designing complex hardware and software systems (hereinafter referred to as *computer systems* or *systems* for short), especially when analyzing and verifying them, one also needs to solve very computer-intensive problems. Generally, these problems come to exploration of a system's state space and testing some properties of a system.

To denote testing methods concerned with the use of supercomputers and computer clusters, we introduce a term *high-performance testing*. Within the scope of this paper, a particular kind of testing (namely, *functional testing*) is considered. Functional testing checks that a *target system* (*SUT*, *system under test*) correctly implements all the functions stated in the specification not examining the internal structure of the system (source code) [2]. It should be noted that for hardware designs functional testing is usually performed not for a finished product (chip or circuit board), but for a simulation model described in a special *hardware description language* (*HDL*) and is known as *simulation-based verification* or *pre-silicon verification* [3].

Many manufactures of complex hardware and software systems (like microprocessors and operating systems) use a lot of computational resources for functional testing of their designs. The most usable approach to test execution is to run independent tests on separate computers (a computer can execute several tests, but one test cannot be executed on two or more computers in parallel). The approach works well for relatively small test batches, but it, obviously, fails to handle huge tasks. The only known exception is *random-based tests*, where several instances of the same test system are executed concurrently. Trying to diversify tests sequences, different instances of a system use different seeds of the random number generator [4].

When testing complex systems with immense state spaces, one needs very long test sequences to cover different states. Theoretically, random tests can solve the problem, but it can take significant amount of time (first, some states are highly improbable and, second, there is duplication of actions among different instances of a test system). The natural solution is to use *graph* or *automaton models*[1] and generate test sequences by *exploring* (*traversing*) those models [5-7]. However, existing graph-based testing tools do not support parallelization and therefore are almost inapplicable to test really complex computer systems. If a state graph is known, one can compute a set of paths covering all the transitions and then use those paths as goals for the test system's instances. However, a transition relation is often unknown and constructed during state graph exploration [6].

In this paper, we suggest a method for parallel generation and execution of tests based on the *distributed graph exploration*. The main advantage of the method is that parallelization is performed fully automatically and transparently. A test system is developed in the same way as it is done for execution on a single computer. The only parameter has to be set when starting a distributed test system is the number of computers to run the system onto. It should be noted that in the suggested approach a test system's processes are not independent ones – they communicate with each other sharing information on what part of the graph model has been traversed. The method and developed tools extend the

[1] Hereinafter, we do not distinguish between graph and automaton models assuming that graphs describe transition relations (state spaces).

UniTESK technology [8,9] and allow verifying systems with millions of model states.

The rest of the paper is organized as follows. Section 2 reviews the related work addressing parallelization of functional tests for computer systems. Section 3 describes the suggested method based on distributed graph exploration; it considers a test system's architecture and process synchronization issues. Section 4 describes applications of the approach to functional testing of real-life systems; this section comprises results on parallelization efficiency with respect to the number of computers. Section 5 concludes the paper and outlines directions of our future research and development.

II. Related Work

First of all, it should be noted that we do not know works investigating parallelization of test generation and execution by means of distribution graph exploration. There are lots of papers on application of graph and automaton models to functional testing (see [5-7] for example), which consider algorithms and methods for generating test sequences, but do not touch upon the issues of tests parallelization.

As we have said before, practical approaches to high-performance testing are based on simultaneous execution of independent or random-based tests on a number of computers. In the latter case, an interesting problem arises. One needs to diversify pseudorandom values being generated among all instances of a test system. This problem is usually referred to as *parallel pseudorandom number generation* and is solved by splitting a long-period stream of random numbers into a set of parallel substreams or by parameterization of a generator to produce several independent full-period streams [4].

Our work is based on the UniTESK test system architecture described in [8,9]. The architecture identifies basic components of a test system including *Traverser* (*Test Engine*), *Test Scenario* (*Test Action Iterator*), and *Test Oracle*. Traverser encapsulates an algorithm for exploring a wide class of directed labeled graphs (some of the algorithms are considered in [10,11]). Test Scenario defines a graph in an implicit form. It provides functions for calculating a current node (state) and for iterating the outgoing arcs (transitions). Test Oracle checks whether SUT's behavior relating to arc traversal (test action) is correct.

Distributed graph-based testing (and our approach in particular) has much in common with exploration of indoor environment by a team of mobile robots (agents) [12,13]. Whenever robots have to solve a common task, they need to coordinate their actions to carry out the task efficiently and to avoid interferences between robots [13]. This problem is also relevant for distributed test systems. Multiple processes should be synchronized to avoid duplication of arc traversals. In [13], an effective coordination strategy is suggested to better distribute robots over environment and to avoid redundant work. The strategy we use is much simpler, but it allows achieving almost linear speedup when testing a system on multiple interconnected computers.

In papers [14] and [15], effective breadth-first-search algorithms for exploring very large graphs on advanced multi-core processors are proposed. In the experimental evaluation they have proved that their algorithms running on 4-socket Intel$^{®}$'s Nehalem-EX is 2.4 times faster than a Cray XMT™ with 128 processors [15]. The tests have also shown linear speedups when using multiple processing units [14]. As opposed to these works, trying to produce as portable system as possible we do not use processor-specific optimizations.

Theoretical aspects of the distributed graph exploration can be found in [16,17] and other papers. The first article describes algorithms to compute spanning trees in undirected networks. In [17], a deterministic algorithm that constructs a map of the simple undirected graph by multiple agents, elects a leader among them, and provides a unique labeling of the nodes is suggested.

III. Suggested Method

According to the UniTESK technology, SUT is modeled by a *finite state machine* (*FSM*). A test system explores a state graph of that FSM model, applies test actions corresponding to the graph's arcs and analyses correctness of SUT's behavior. The thing is that graph models of complex systems are of a huge size, and therefore test execution on a single computer takes too much time.

As a way to improve test execution performance we have chosen tests parallelization on computer clusters and have set the task to extend the UniTESK technology by means of distributed test execution on computer clusters. The main requirement we have formulated is that changes in test execution should not affect test development. It means that all tests (including those ones that have been developed earlier) can be executed on a single computer as well as on a number of interconnected computers.

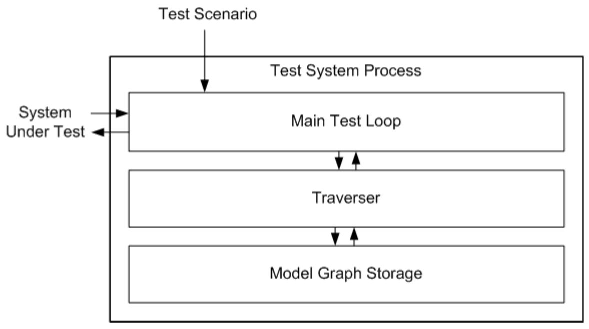

Figure 1. Test system architecture for execution on a single computer

Test system architecture for execution on a single computer is shown on Fig. 1. A library subsystem *Model Graph Storage* manages information about a known part of a model graph. Another library subsystem, *Traverser*, fills up the storage when exploring a model graph. *Main Test Loop* is an active part of the test system. It takes *Test Scenario* (which is an implicit description of a model graph) as an input and uses Traverser to explore the graph by finding paths to uncovered arcs. When traversing the model graph, Main Test Loop applies test actions and analyses SUT's reactions.

The key idea used for parallel test execution is that Traverser's algorithm remains almost the same if there are additional sources for filling up the model graph storage. It is enough to start test system processes (each containing a test system's instance and SUT) on computer cluster's nodes and provide exchange of information about traversed arcs of the model graph between all of the processes. We have developed a new library subsystem, called *Synchronizer*, that is responsible for information exchange. Modified test system architecture is shown on Fig. 2.

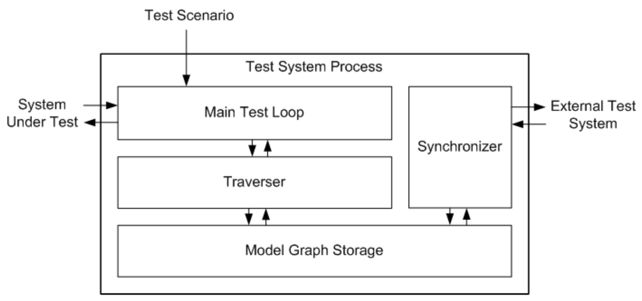

Figure 2. Test system architecture for execution on a computer cluster's node

A model graph's arc traversed by one test system process will be known to all other processes and they will not traverse it by themselves wasting no time to duplicate work that has been already done. Performance improvement is possible only under the following conditions:

- It should be more profitable for a process to receive information about traversed arcs from other processes than traversing them by process itself. In other words, exchange of information about traversed arcs should be significantly faster than local traversing.
- Among information about traversed arcs received from other processes should not be many arcs already known by a process.

Small change in traversal algorithm has been made to satisfy the second condition. In case of single execution, a Traverser's choice of the next untraversed arc was deterministic. For parallel execution it means that starting from the initial node all Traversers will go through the same arc, and further they will also make the same decisions sending each other a lot of useless information and thus violating the second condition. Therefore, the choice of the next untraversed arc has been made dependent on index of a process (all processes are numerated) producing different variations of the traversal order.

A synchronization protocol for exchanging graph information should provide delivery of each new message about a traversed arc to each process and preferably only once (to reduce network traffic). Such protocol has been developed. Synchronizers of different instances of a test system establish one-way point-to-point connections to each other. A topology can be arbitrary while the oriented graph of the test system channels (*communication graph*) is strongly connected (to ensure delivery of messages from any process to any other in the same order and without losses).

Each test system process does synchronization regularly. It is initiated by incoming messages, by new arcs traversed locally or on timeout. The synchronization algorithm is as follows:

1. Synchronizer receives all incoming network messages and asks Traverser for *local update* – a set of new arcs traversed by the process since the time of the latest synchronization. Let $\boldsymbol{R}$ (received) be a set of arcs in the received messages, $\boldsymbol{S}$ (sent) be a set of arcs that have been already sent via the outgoing connections of the process and $\boldsymbol{N}$ (new) be a local update.
2. A set of new arcs received from other processes $\boldsymbol{R}\setminus(\boldsymbol{S}\cup\boldsymbol{N})$ is added to Model Graph Storage.
3. A message containing a set of arcs $(\boldsymbol{R}\cup\boldsymbol{N})\setminus\boldsymbol{S}$ is sent via all outgoing connections.
4. A set of sent arcs is updated: $\boldsymbol{S} := \boldsymbol{S}\cup\boldsymbol{R}\cup\boldsymbol{N}$.

It is obvious that this synchronization protocol guarantees that a message about any traversed arc will be transferred through each connection only once. Therefore, if messages about some traversed arc have been received from every ingoing connection, that arc may be removed from the set $\boldsymbol{S}$, because all other processes already know about it.

Maximal time required for all processes to receive a message about an arc traversed by some process is proportional to a diameter of the communication graph. The polar variations are "all-to-all" and "ring" topologies. Choosing optimal topology of communication graphs is beyond the scope of this paper. However, we have experimented with some topologies; our results are described in the next section.

Parallelization considerably speedups test execution, but there are huge tests taking much time even in case of parallel execution. A capability to temporarily interrupt test execution makes testing more convenient. This facility is implemented by logging all outcoming network messages by one or more processes of a test system. When execution continues after interruption, all processes start from the initial state as usual. At the same time, those processes that have logs resend the logged messages to the other processes. It is obviously much faster than test reexecution. Moreover, it increases reliability of the system.

Further specialization of test system processes is possible. Some processes may be targeted at resource-intensive tasks, for example, they may be intended for searching a path in a graph from a given node to a node that has untraversed outgoing arcs. It lowers CPU and memory requirements for other processes – they may store not a whole graph but only neighborhoods of their current nodes. A light-weight process works independently from the bigger ones when there are untraversed arcs in the current state, but if it fails to find a path by itself, it delegates the task to one of the dedicated processes.

Another promising optimization is using different graph exploration strategies with central coordination. For example, if there are two arcs going out from the initial node – the first one leads to a big loop with only one arc from each node, while the second one leads to the root of a tree with many arcs from each node. With our current strategy about half of processes will go via the first arc through the big loop without any profit. With central coordination it is possible to pause all processes except two and decide where to direct them after the first two explore the neighbor nodes. Such optimizations have much sense if it is known that a model graph has a specific structure.

IV. Experimental Results

The approach described above has been applied to parallel functional testing of various hardware designs. Depending on the design complexity and testing purposes, model graphs include from thousands to millions of nodes and up to several millions of arcs. Test execution has been performed on 1-150 computers (Intel® Core™2 Quad Q9400, 2.66GHz; 4GB RAM) running the Linux operating system and networked via Ethernet. Table I shows the tests classification depending on the number of arcs in a model graph and indicates the amount of resources required for test execution (number of computers and time).

TABLE I. COMPLEXITY-BASED TESTS CLASSIFICATION

Test complexity	**Number of arcs in a model graph**	**Number of computers**	**Execution time, min.**
Simple tests	< 10000	1	< 30
Medium tests	10000 - 100000	1-10	< 30
Complex tests	100000 - 1000000	10-100	< 30
Huge tests	> 1000000	> 50	> 60

The analytical estimation of the method effectiveness is difficult, because many factors should be taken into account (message passing time, communication topology, and others). We have conducted a number of experiments and have measured the parallelization efficiency $\boldsymbol{K(n)=T(1)/(n \cdot T(n))}$, where $\boldsymbol{T(n)}$ is time of test execution on $\boldsymbol{n}$ computers.

The experiments show that if a communication topology is chosen correctly, the parallelization efficiency exceeds 0.8. There should be however a few comments. First, experiments have been performed on multi-core microprocessors enabling Synchronizer not to take computational resources from Traverser. Second, we have tried two topologies for different numbers of computers. ("ring" for 8 or less computers and "two-dimensional torus" for 9 or more computers). These two options are enough for 100-150 computers, but for effective parallelization on a larger number of machines it might require other topologies ("three-dimensional torus", "hypercube", etc.).

TABLE II. PARALLELIZATION OF A MEDIUM TEST

Number of computers	**Topology**	**Execution time, min.**	**Parallelization efficiency**
1	—	95.2	1
9	Ring	11.8	0.9
9	Torus 3×3	10.9	0.97
16	Ring	6.7	0.89
16	Torus 4×4	6.2	0.96
25	Ring	4.4	0.87
25	Torus 5×5	4.0	0.95

Tables II and III show the results of tests execution. The first table shows execution time and parallelization efficiency for a test of medium complexity (18 227 nodes and 109 362 arcs). The second table corresponds to a complex test (84 561 nodes and 338 244 arcs).

TABLE III. PARALLELIZATION OF A COMPLEX TEST

Number of computers	**Topology**	**Execution time, min.**	**Parallelization efficiency**
1	—	803.3	1
81	Ring	12.2	0.81
81	Torus 9×9	11.4	0.87
100	Ring	10.2	0.79
100	Torus 10×10	9.5	0.85

V. CONCLUSION

In the paper, the extension of the UniTESK technology by means of parallel test execution on computer clusters is described. An important feature of the suggested method is that parallelization is done dynamically without using static information on a SUT's transition relation (structure of a model graph). From an engineer's point of view, it is not more difficult to work with a distributed test system than to use a single-computer one (additional input data are number of computers to run the system onto and, optionally, a network topology). The approach significantly speeds up test execution shrinking bug detection time and accelerating the design process in whole. In the future we are planning to make the tools even more flexible. For instance, we are going to support dynamic reconfiguration of a test system's topology (e.g. run-time changing of a number of computers executing the test system) and to support computer systems with shared memory (in this case, more efficient implementation of the synchronizers is possible as well as graph storage sharing).

REFERENCES

[1] http://en.wikipedia.org/wiki/High-performance_computing.

[2] B. Beizer. Black-Box Testing: Techniques for Functional Testing of Software and Systems. John Wiley & Sons, 1995.

[3] W. Lam. Hardware Design Verification: Simulation and Formal Method-Based Approaches. Prentice Hall, 2005.

[4] M. Mascagni. Parallel Pseudorandom Number Generation, SIAM News, 32(5), 1999, pp. 1-6.

[5] C. Turner, D. Robson. The State-Based Testing of Object-Oriented Programs. Conference on Software Maintenance, 1993, pp. 302-310.

[6] I. Bourdonov, A. Kossatchev, V. Kuliamin. Application of Finite Automatons for Program Testing. Programming and Computer Software, 26(2), 2000, pp. 61-73.

[7] G. Friedman, A. Hartman, K. Nagin, T. Shiran. Projected State Machine Coverage for Software Testing. International Symposium on Software Testing and Analysis, 2002, pp. 134-143.

[8] I. Bourdonov, A. Kossatchev, V. Kuliamin, A. Petrenko. UniTesK Test Suite Architecture. International Symposium of Formal Methods Europe, 2002, pp. 77-88.

[9] V. Kuliamin, A. Petrenko, A. Kossatchev, I. Bourdonov. The UniTesK Approach to Designing Test Suites. Programming and Computing Software, 29(6), 2003, pp. 310-322.

[10] I. Bourdonov, A. Kossatchev, V. Kuliamin. Irredundant Algorithms for Traversing Directed Graphs: The Deterministic Case. Programming and Computer Software, 29(5), 2003, pp. 245-258.

[11] I. Bourdonov, A. Kossatchev, V. Kuliamin. Irredundant Algorithms for Traversing Directed Graphs: The Nondeterministic Case, Programming and Computing Software, 30(1), 2004, pp. 2-17.

[12] M. Gossage, A. Peng New, C.K. Cheng. Frontier-Graph Exploration for Multi-robot Systems in an Unknown Indoor Environment. Distributed Autonomous Robotic Systems 7, 2006, pp. 51-60.

[13] C. Stachniss, O.M. Mozos, W. Burgard. Efficient Exploration of Unknown Indoor Environments Annals of Mathematics and Artificial Intelligence, 52(2-4), 2008, pp. 205-227.

[14] O. Villa, D.P. Scarpazza, F. Petrini, J.F. Peinador. Challenges in Mapping Graph Exploration Algorithms on Advanced Multi-core Processors. International Parallel and Distributed Processing Symposium, 2007, pp. 1-10.

[15] V. Agarwal, F. Petrini, D. Pasetto, D. Bader. Scalable Graph Exploration on Multi-Core Processors. International Conference for High Performance Computing, Networking, Storage and Analysis, 2010, pp. 1-11.

[16] G. Tel. Distributed Graph Exploration. 1997.

[17] S. Das, P. Flocchini, A. Nayak, N. Santoro. Distributed Exploration of an Unknown Graph. Structural Information and Communication Complexity, 2005, pp. 99-114.