2009 International Conference on Software Testing Verification and Validation

PKorat: Parallel Generation of Structurally Complex Test Inputs

Junaid Haroon Siddiqui
University of Texas at Austin
Austin, TX 78712
jsiddiqui@ece.utexas.edu

Sarfraz Khurshid
University of Texas at Austin
Austin, TX 78712
khurshid@ece.utexas.edu

Abstract

Constraint solving lies at the heart of several specification-based approaches to automated testing. Korat is a previously developed algorithm for solving constraints in Java programs. Given a Java predicate that represents the desired constraints and a bound on the input size, Korat systematically explores the bounded input space of the predicate and enumerates inputs that satisfy the constraint. Korat search is largely sequential: it considers one candidate input in each iteration and it prunes the search space based on the candidates considered.

This paper presents PKorat, a new parallel algorithm that parallelizes the Korat search. PKorat explores the same state space as Korat but considers several candidates in each iteration. These candidates are distributed among parallel workers resulting in an efficient parallel version of Korat. Experimental results using complex structural constraints from a variety of subject programs show significant speedups over the traditional Korat search.

1. Introduction

The increasing availability of cheap commodity (multi-core) processors offers novel opportunities for developing parallel techniques for efficient testing. This paper presents a parallel technique for efficient *test generation*, i.e., the process of creation of test cases to run, which is typically one of the most resource-intensive processes of automated testing. For programs with structurally complex inputs, such as balanced binary search trees or XML documents, test generation can be particularly expensive.

Korat [1, 27] is a framework for automated generation of structurally complex tests. Korat performs *specification-based testing*: given a Java predicate that represents properties of desired inputs, Korat uses a backtracking search to explore the input space of the predicate and enumerate inputs for which the predicate returns true—each enumerated input represents a desired test input. To test a method, Korat uses the method precondition to generate tests and the method postcondition to check correctness. Korat enables *bounded exhaustive testing*: it tests against all nonisomorphic inputs within a given bound on the input size. Bounded exhaustive testing has successfully been used to find bugs in various applications, including a fault-tree analyzer [37], a resource dicovery architecture [17], and an XPath compiler [36].

For efficient search, Korat uses a dynamic analysis of the predicate: Korat search prunes large portions of the input space by monitoring predicate executions on candidate inputs. Korat monitors the field accesses of the predicate and backtracks on the last field accessed. Korat makes a non-deterministic assignment to that field to generate a new candidate input and re-executes the predicate.

While the dynamic analysis enables efficient pruning, it renders the Korat search inherently sequential: without executing the predicate on a candidate input, Korat cannot determine the next candidate input. Moreover, a simple partitioning of the state space and a distributed search does not yield high speed-ups since it is not possible to predict which parts of the state space would be pruned and exploring those parts is simply redundant.

This paper presents PKorat, a parallel test generation framework based on Korat. Our key insight is that even though it is not feasible to *fast-forward* Korat search effectively, the systematic exploration of the input space can still be parallelized by exploring non-deterministic field assignments in parallel. PKorat search backtracks by generating *several* candidate inputs that are explored in parallel. A key aspect of PKorat is that it explores exactly the same number of candidates that Korat does and still enables a scalable parallel implementation. PKorat uses a master-slave configuration to implement its parallel search. When delegating exploration to a slave processor, PKorat provides a small amount of meta-information that prevents different slaves from exploring the same candidates and avoids redundant exploration. Previous work [31] on parallelizing Korat provided techniques that efficiently parallelize test execution, while test generation was mostly sequential (Section 5).

978-0-7695-3601-9/09 $25.00 © 2009 IEEE
DOI 10.1109/ICST.2009.48

While this paper focuses on a parallel search that implements the Korat algorithm, our technique applies to parallelizing other systematic analyses that are based on nondeterministic assignments, e.g., symbolic execution [6, 18, 19, 34]. Our future work will generalize our parallel technique and evaluate its effectiveness in the more general context of symbolic execution (Section 6).

This paper makes the following contributions:

- **Parallel Algorithm for Test Generation:** We present PKorat, a scalable parallel algorithm based on Korat for generation of structurally complex test inputs.
- **Implementation:** Our prototype implementation is in C++ and uses MPI [11]. We ran it on a Linux cluster hosted at Texas Advanced Computing Center [15].
- **Evaluation:** We evaluate our approach on a variety of commonly data structures, including complex structures, such as red-black trees and Java programs [3]. The results show the scalability of PKorat and the significant speedups it provides over Korat.

The rest of this paper is organized as follows. Section 2 presents an example of a binary tree and illustrates how Korat and PKorat explore the input space. Section 3 discusses the PKorat algorithm in detail, and argues its correctness. Section 4 evaluates its performance against Korat on five subject programs. Section 5 discusses related work and Section 6 presents conclusions and future work.

2. Background and Example

To explain our parallel algorithm, we take the example of a binary tree. We first describe the working of Korat on this structure with three nodes. We then explain how PKorat explores the same state space.

2.1. Binary Tree

Consider the C++ definition of binary tree given in Figure 1. The inner class `Node` models actual nodes in the binary tree. Each `Node` has `left` and `right` pointers, pointing to other nodes. The `BinaryTree` class has a `root` pointer pointing to the root of the binary tree and an integer `size`, that caches the total number of reachable nodes.

There are two structural constraints. One is acyclicity along left and right fields. The second is that the number of reachable nodes equals the `size` field. A C++ predicate to verify these constraints is given in Figure 1. Such an imperative predicate is conventionally called repOk [26]. For object oriented languages, these constraints are also called class invariants.

```
class BinaryTree {
  class Node {
   public:
    Node* left;
    Node* right;
  };
  Node* root;
  int size;
 public:
  bool repOk() {
    std::set<Node*> visited;
    std::stack<Node*> worklist;
    if( root ) {
      worklist.push( root );
      visited.insert( root );
    }
    while( !worklist.empty() ) {
      Node* current = worklist.top();
      worklist.pop();
      if( current->left ) {
        if( !visited.insert( current->left ).second )
          return false;
        worklist.push( current->left );
      }
      if( current->right ) {
        if( !visited.insert( current->right ).second )
          return false;
        worklist.push( current->right );
      }
    }
    return visited.size() == size;
  }
};
```

Figure 1. Definition of Binary Tree with its repOk function. This function validates the class invariant consisting of two constraints: acyclicity along all paths and equality of size field and number of reachable fields.

2.2. Test Generation using Korat

To generate test inputs, Korat takes a class declaration with its `repOk` predicate and a set of bounds called *finitization*. Finitization is the specification of bounds on field assignments. For objects, it limits the number of objects created, and for primitive types, it defines the range of valid values. The finitization we use takes one object of class `BinaryTree`, three objects of class `Node`, and a fixed value of 3 for `size` field.

Every possible structure within these bounds can be represented by eight values. The value of `root` and `size` for the lone `BinaryTree` object and the values of `left` and `right` for each of the three `Node` objects. All of these fields except `size` take a value from the set $\{NULL, N_0, N_1, N_2\}$, whereas `size` takes a value from the singleton set $\{3\}$. These sets are called the *domains* of the respective fields. Every candidate structure can equivalently be represented

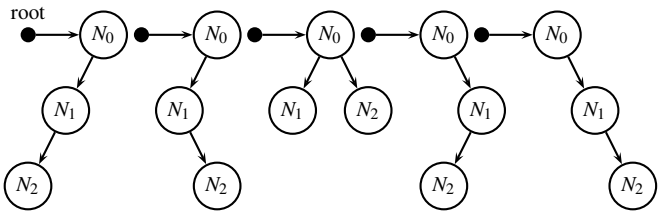

Figure 2. There are only 5 valid non-isomorphic binary trees of 3 nodes. The total number of possible structures is 16,384. Some of them are cyclic, some do not have all three nodes as reachable, while some are isomorphic to the ones shown here.

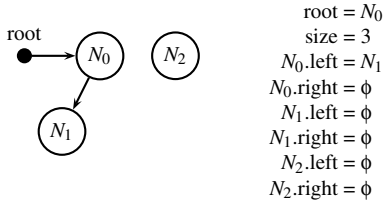

Figure 3. A candidate explored by Korat. It is rejected as size=3 whereas accessible nodes are only 2.

by an 8-tuple consisting of indices into respective domains for each field. It is called a *candidate vector* in Korat.

In our example, the size of state space is $4^7 = 16,384$, whereas only five are valid and non-isomorphic binary trees containing all three nodes as shown in Figure 2. Non-isomorphism means that trees obtained by swapping identities of objects are not reported. Detailed explanation of Korat's isomorphism handling is given elsewhere [30], whereas more details of isomorphism handling in PKorat are discussed in Section 3.3.

Korat explores only 63 candidate vectors out of over 16 thousand possible candidates and still discovers all 5 valid binary tree. Korat starts its search by exploring the candidate vector with all zeroes. It uses the predicate `repOk` to test this input. Korat monitors the execution of `repOk` and finds out which fields were accessed during its execution. For the next candidate, Korat increments the last accessed field to the next value in its domain. If one or more of the fields most recently accessed point to the last valid value in their domain, the field before them is incremented. If every accessed field has reached its maximum, the algorithm terminates. Otherwise it repeats by checking the predicate on this new candidate and by monitoring its execution to find the next candidate.

To avoid isomorphic structures, Korat restricts the domain of non-primitive fields to only one untouched object of that domain. An object is considered touched if another field having the same domain is pointing to that object, and that field is accessed before the current field. This is explained below with the help of an example, whereas algorithmic details, and the completeness and soundness proof of the whole algorithm is given elsewhere [1].

Consider the candidate vector $\langle 1,3,2,0,0,0,0,0\rangle$ and the corresponding candidate shown in Figure 3. This is the 27^{th} candidate explored by Korat. It is rejected by `repOk` as there are two accessible nodes whereas `size` is 3. The two fields, N_2.left and N_2.right are never accessed during its execution as there is no way to reach them from the root. The other fields are accessed in the order $\langle root, N_0.left, N_0.right, N_1.left, N_1.right, size\rangle$.

Since the primitive value `size` is already at its maximum, Korat tries to mutate $N_1.right$ to its next value. Korat calculates the maximum index it should try for this field. It observes that from its domain, N_0 and N_1 have been accessed before this particular field is accessed. Using the rule of only one untouched object, Korat allows N_2 as the maximum value of this field. In this case, it is the last value in the domain. But even if there were more than 3 nodes, Korat would still allow objects only up to N_2 at this stage.

As the next value of $N_1.right$ is N_0 which is before the maximum just calculated (N_2) so it is a valid candidate. This change makes the structure cyclic and hence it is rejected by `repOk`. It would point to N_1 in the next iteration and again get rejected due to a self loop. It would then point to N_2 and become the first binary tree shown in Figure 2. Korat proceeds by changing fields according to access order and testing the resulting candidates in this manner.

2.3. PKorat Algorithm

Korat produces only one candidate vector after testing the previous one. PKorat, on the other hand, produces a list of candidate vectors after testing each candidate. Furthermore, all these candidates are produced by Korat as well. The working of PKorat on the same example as in Section 2.2 is given below. The equivalence of candidates explored by Korat and PKorat is discussed in Section 3.4.

Consider that PKorat is working on the candidate given in Figure 3 and the ordered list of accessed fields during `repOk` is $\langle root, N_0.left, N_0.right, N_1.left, N_1.right, size\rangle$.

PKorat, like Korat, finds the maximum value for `size` which in this example is 3. As it already maximum, it is reset to minimum (which is also 3) and maximum value for $N_1.right$ is calculated and found to be N_2. At this point, PKorat differs in that it generates three candidate vectors with the value of $N_1.right$ set to N_0, N_1, and N_2, respectively. It then finds the maximum value of $N_1.left$ and produces

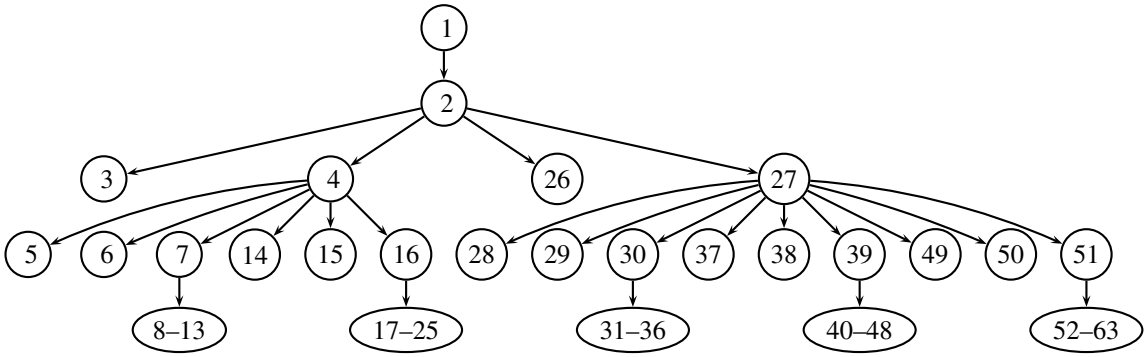

Figure 4. Tree of explored candidates showing Parallel Korat search progress, where the numbers represent the sequential order of their exploration in standard Korat (last level nodes aggregated). Note that given enough processors and ignoring overheads, PKorat can ideally explore this state space in 5 units of time, whereas Korat would take 63. Even a quad-core can complete in 18 units of time giving 3.5X speedup.

three more candidates with value of $N_1.left$ set to N_0, N_1, and N_2, respectively. Another three vectors are produce by setting $N_0.right$ to N_0, N_1, and N_2, respectively. When we observe that $N_0.left$ is not at index 0 we have to stop since the current vector would have been produced in parallel by a similar previous step which catered $N_0.left$ and anything before it. After producing all these nine candidate we can distribute them and test them with a predicate in parallel. Each distributed task consists of evaluating a candidate and distributing the subsequent candidates again.

The motivation behind producing all these candidates is based on the fact that Korat cannot prune a candidate all of whose non-zero fields are accessed. Using the ordered list of accessed fields, we make a candidate by changing some field while keeping everything accessed before this field unchanged, and everything accessed after it, pointing to the first value in their domain. Every non-zero field in this candidate would be accessed, because we assume `repOk` is deterministic and every field accessed before the changed field has the same value. Therefore Korat does not prune it. Furthermore stopping at first non-zero accessed fields protects against the same candidate produced out of two different candidates. Both of these facts are formally proved in Section 3.4.

Figure 4 shows how the 63 candidates explored by Korat are explored by PKorat. The numbers represent the order in which Korat explores them sequentially. Note that the example candidate 27 produces nine candidates as discussed above.

Given enough processors and ignoring overheads, the parallel algorithm can complete test generation in 5 units of time instead of 63 units. A quad-core can ideally complete the task in 18 units of time giving 3.5X speedup. The actual speedup is lower due to communication overheads.

3. Algorithm

PKorat uses a master slave configuration [23], where processor 0 is always designated as the master processor. It contains a work queue of candidate vectors. Every other processor acts as slave and contacts the master for a candidate vector to process. After processing it send a list of candidate vectors back to the master, and the master enqueues them.

3.1. Master Algorithm

Algorithm for master process is given in Figure 5. The master processor maintains a list of free slaves (`slaveQ`) and a list of pending candidate vectors (`workQ`). It initializes work queue with an all zero candidate vector (`ZeroV`). It then loops while receiving and processing slave requests.

There are two types of requests. A QUEUE request and a DEQUEUE request. The former is used to add candidate vectors to the work queue, while the later is used to fetch a candidate vector. If a candidate vector is not readily available, the slave is remembered in slave queue. Whenever candidate vectors become available, they are sent to these free slaves instead of being queued in work queue. The algorithm terminates when every slave is in slave queue. At this stage the work queue must also be empty. As soon as it happens, it sends all slaves an EXIT message.

3.2. Slave Algorithm

Slave processors are responsible for processing a given candidate vector and producing zero or more candidate vectors to be processed next. Whereas the master processor is

```
function MASTER
    workQ ← CREATEQUEUE()
    slaveQ ← CREATEQUEUE()
    QUEUE(workQ, ZeroV)
    while SIZE(slaveQ) ≠ SlaveCount do
        (slave, cmd, candidateV) ← RECV(any)
        if cmd = QUEUE then
            if EMPTY(slaveQ) then
                QUEUE(workQ, candidateV)
            else
                slave′ ← DEQUEUE(slaveQ)
                SEND(slave′, WORK, candidateV)
            end if
        else if cmd = DEQUEUE then
            if EMPTY(workQ) then
                QUEUE(slaveQ, slave)
            else
                candidateV ← DEQUEUE(workQ)
                SEND(slave, WORK, candidateV)
            end if
        end if
    end while
    for all s ← slaveQ do
        SEND(s, EXIT)
    end for
end function
```

Figure 5. Algorithm for master processor in PKorat. It keeps a queue of waiting slaves and a queue of candidate vectors. On a dequeue request, a candidate vector is sent if available, otherwise the slave goes in waiting queue. On a queue request, the candidate vector is sent to waiting slaves, or stored in candidate queue if none are waiting. The algorithm terminates when every slave is in waiting queue.

```
function SLAVE
    SEND(master, DEQUEUE)
    (cmd, candV) ← RECV(master)
    while cmd = WORK do
        candidate ← BUILDCANDIDATE(candV)
        (pred, accV) ← REPOK(candidate)
        if pred then VALIDCANDIDATE(candidate)
        end if
        while SIZE(accV)>0 ∧ TOP(accV)=0 do
            field ← POP(accV)
            for i ← 1, NONISOMAX(candV, accV, field)
                do
                candV[field] ← i
                SEND(master, QUEUE, candV)
            end for
            candV[field] ← 0
        end while
        SEND(master, DEQUEUE)
        (cmd, candV) ← RECV(master)
    end while
end function
```

Figure 6. Algorithm for slave processors in PKorat. It builds a C++ object structure using BuildCandidate. Then tests it using repOk. Then for all accessed fields up to a field pointing to a non-zero index, it generates candidates for all non-zero indices of that field up to the maximum index given by NonIsoMax. NonIsoMax is given in Figure 7

only concerned with seamless communication, novel contributions of this paper manifest in slave processing. The algorithm is given in Figure 6.

Every slave sends the master processor a DEQUEUE request, receives a candidate vector (`candV`), processes it, possibly sends back a QUEUE request, and then repeats the whole thing. It terminates when a DEQUEUE request results in an EXIT response from the master.

To process a candidate vector, the slave first converts the candidate vector to an actual candidate C++ data structure using BUILDCANDIDATE. This function not given here, is identical to the one used in Korat. It works by assigning all fields of all objects involved according to domain indices stored in the candidate vector. Figure 3 shows the assignments for the example discussed in Section 2.2 and Section 2.3.

The predicate REPOK is then used to validate the given candidate. VALIDCANDIDATE is invoked if this candidate is valid. This function, also not given here, is a placeholder for *any* processing of valid candidates. For example, it can count the number of valid candidates, store the candidate in a file for later use, or even invoke the actual program for which these tests are generated. The last two options are similar in nature to offline and online test execution in previous work on parallelizing Korat [31]. However, as discussed before, our solution does not share its limitations.

Lastly, candidate vectors to be explored next are found. The algorithm works using the accessed fields stack (`accV`) returned by REPOK. This contains all fields accessed by REPOK in making its decision, in the order of first access. The slave pops fields one by one off this stack until a non-zero field is found. For each popped field, it produces candidate vectors for all non-zero valid values of this field. The maximum valid value is given by NONISOMAX and is discussed in Section 3.3. It stops at a non-zero field to avoid produc-

```
function NONISOMAX(candV, accV, field)
    m ← MAXDOMAININDEX(field)
    if NONPRIMITIVE(field) then
        touched ← 0;
        for all i ∈ accessedV do
            if SAMEDOMAIN(i, field) then
                touched ← MAX(candV[i], candV[field])
            end if
        end for
        m ← MIN(m, touched+1)
    end if
    NonIsoMax ← m
end function
```

Figure 7. Helper function for slave processors in PKorat. It returns the maximum valid index for primitive fields. For non-primitive fields it ensures that no more than one untouched object is tried. An object is touched if a previously accessed field with the same domain points to it.

ing the same candidates as some other slave would do. This is discussed and proved in Section 3.4.

Actual implementation aggregates candidate vectors to be sent back to master in fewer messages. It also uses double buffering at client so that a candidate vector can be received and ready to be processed while the previous one is being processed. These are considered as implementation optimizations and excluded from the algorithm description here, to emphasize key contributions of this work.

3.3. Non-isomorphism

Isomorphic candidates are candidates that only differ in the identities of their objects. By swapping all members of any two objects and then swapping all references to these objects, an isomorphic copy of the original structure can be formed. For example, isomorphic copies of trees in Figure 2 can be formed by using N_1 as the root, and N_0 in place of N_1. Most programs do not make decisions based on object identities (e.g. their memory address) and therefore testing isomorphic copies is a waste of time.

Our algorithm for avoiding isomorphic copies is identical to Korat [30]. A simplified version assuming that *NULL* is always tested for object fields and assuming no polymorphic fields (fields that take objects of more than one type) is given in Figure 7. For non-primitive fields the algorithm checks all previously accessed fields having the same domain, and finds the maximum index accessed (`touched`). If any untouched objects are left, it allows one of them to be tried.

Even though the basic algorithm for isomorphism avoidance is identical to Korat [1], there is an inherent efficiency in PKorat. Korat needs to repeat this procedure to find the valid maximum in every iteration. It optimizes this by using caching, so that the maximum for a given field, with the same values of fields accessed before it, is calculated only once. Our parallel algorithm however generates all such candidate vectors at the same time. Therefore it only calculates the valid maximum once for each field with the same values accessed before it and no caching is needed either.

3.4. Completeness and Soundness

We prove completeness and soundness of PKorat by proving equivalence with Korat for a deterministic `repOk` function. Completeness and soundness of Korat has been proven elsewhere [1]. We prove equivalence by showing that PKorat explores a candidate if and *only* if Korat explores it.

Proof. PKorat pops fields from accessed field stack and stops at the first field set to a non-zero index. This is because PKorat tries all non-zero indices at the same time. Thus, the time this field was set to the current non-zero value, all other mutations would have been tried as well. Similarly, mutation of previous fields would have either been produced at the same time or would have been produced at some previous invocation by applying the argument recursively. Therefore PKorat produces the immediate next candidate (according to Korat sequence) either at the same time or has produced it previously. Therefore every candidate produced by Korat is produced by PKorat as well. This proves the *if* direction.

A candidate is pruned out by Korat only if there is another explored candidate with the same values for all fields accessed by `repOk`. This explored candidate has all non-accessed fields set to zero index. This is because Korat generates a new candidate by mutating only the last accessed field. For a deterministic `repOk`, this implies that *exact* same fields will be accessed again and possibly some more. When Korat backtracks, fields up to the backtracked field must be accessed and possibly some more. Thus all fields that are not certain to be accessed are at zero index. Therefore, if all non-accessed fields of a candidate are at zero index, it cannot be pruned out by Korat. Since PKorat only produces such candidates, we can infer that they are produced by Korat as well. This proves the *only if* direction and completes our proof. □

4. Evaluation

To evaluate the efficiency of PKorat, we implemented it in C++ using Message Passing Interface (MPI) [11] library. We also implemented the serial Korat algorithm in

Subject	Proc	Time (s)	Speedup
Singly Linked List (500 nodes)	Serial	1387	1.0X
	4p	523	2.6X
	16p	108	12.8X
	64p	28	49.5X
Binary Tree (13 nodes)	Serial	480	1.0X
	4p	250	1.9X
	8p	120	4.0X
	16p	105	4.6X
Red Black Tree (10 nodes)	Serial	347	1.0X
	4p	136	2.6X
	8p	82	4.2X
	16p	77	4.5X
Directed Acyclic Graph (7 nodes)	Serial	1163	1.0X
	4p	544	2.1X
	8p	318	3.7X
	16p	284	4.1X
Java Class Hierarchies (8 nodes)	Serial	149	1.0X
	4p	71	2.1X
	8p	45	3.3X
	16p	39	3.8X

Table 1. Results of Korat and PKorat for a number of different data structures. For each parallel run, there is one master and the rest are slaves. When the subject data structure has many nodes, the parallel versions give the most benefit, up to 49.5X on 64 nodes. Quad-core performance in all cases is exceptional, considering that only three processors are actually working on candidates it is between 1.9–2.6X.

C++ so that it can be run on the same machines and meaningful performance comparison can be made. We have run the experiments on a Linux cluster provided by Texas Advanced Computing Center [15]. The cluster consists of 1,300 nodes, with 2 dual core processors per node, for a total of 5,200 cores. The serial tests were run on a single core of this machine while the parallel runs used more cores.

Our test subjects include standard data structures such as a singly linked list, a binary tree (as given in Section 2), and a red-black tree. We also test directed acyclic graphs (DAG) to compare performance with previous work and to discuss issues with isomorphic graphs. We then give a novel application for automated generation of all valid Java class hierarchies with a given number of classes and interfaces.

We run our experiments with various number of processors to show the scalability of our algorithm. For each experimental run, we give the speedup from serial version and the clock time for the execution. We discuss specific details and experimental observations in the following Sections.

4.1. Singly Linked Lists, Binary Trees, and Red-black trees

We test PKorat and compare it to Korat for singly linked lists, binary trees, and red-black trees. These standard structures have been commonly used to evaluate the performance of algorithms for generating structurally complex test inputs. This experiment shows the benefits of using parallel processing for generating these structures.

For singly linked lists, we were able to explore 500 node lists in a reasonable time. Using PKorat for such big structures proved the most beneficial. We were able to get 49.5X speedup with 64 processors. Due to complexity of state space search for binary tree and red-black tree, Korat is able to search 13 node for the former and 10 nodes for the later within a 30 minute limited we posed. They showed scalability up to 16 processors. If we had more nodes like singly linked lists, the speedup would be better. However the clock time to run the experiment would also be much more.

Note that communication costs rise with more nodes. That is because when processors are allocated from a big cluster, they are not always close together and there is a single master to communicate to. Therefore, even though PKorat cannot deteriorate in performance with more processors, it does deteriorate in practice after an optimal number of processors due to increased average node to node communication time.

4.2. Directed Acyclic Graphs (DAG)

Directed Acyclic Graphs (DAG) can be represented as shown in Figure 8a. To verify the acyclicity constraint, we can do a depth first search from each starting node and find if any node can be reached twice. This scheme, however, leads to a number of equivalent inputs, i.e. structures that are different at the concrete level but represent the same DAG. Detailed discussion of this topic can be found in previous work [31], which also discusses an optimized `repOk`. This optimized `repOk` is more restrictive and would even reject some valid graphs. The end result is fewer generated graphs without missing any unique graph. The algorithm is based on descendent counting and ordering.

We use a very simple and less compute intensive approach that results in even fewer generated graphs. We write a `repOk` that only accepts topologically sorted graphs. Furthermore, nodes in any array are ordered in increasing order of some identifier. It is easy to prove that this approach generates all graphs since every DAG can be topologically sorted. But equivalent graphs are still produced as topological sorting of a DAG is not unique. For a DAG of six nodes, a `repOk` that accepts every valid graph explores 16,216,503 structures and declares 1,336,729 as valid. The optimized `repOk` [31] explores 2,628,140 structures with 21,430 valid

instances. Our simple topological sort based `repOk` explores merely 517,743 structures but validates 32,768 of them. Thus generation is much faster but testing would be slower due to more test cases generated than previous work. For comparison, note that the actual number of non-isomorphic graphs of size 6 is only 5,984 [31].

Results show that we can more than double the speed on a quad-core. Note that only three processors are actually evaluating candidates on a quad-core, so the speedup on 2.1 of a maximum possible of 3 is actually significant. Performance keeps on improving for more processors until speedup crosses 4. After that, enough parallelism is not generated to exploit additional processors. Rise in average communication costs actually decreases speedup for more processors.

4.3. Java Class Hierarchies

We use Korat to generate all valid Java programs with a given number of classes and interfaces. We then show the performance gains with our parallel algorithm. Automated generation of valid programs can be very useful in testing of compilers and associated tools. We define the three classes in Figure 8b for this experiment. A JavaProgram class whose lone instance represents the program to be generated. JavaClass and JavaInterface classes representing classes and interfaces, respectively. Finitization puts bounds on the number of classes and interfaces. The `repOk` function checks that the classes form a valid class hierarchy. One of the results produced is shown in Figure 8c.

This problem is similar to directed acyclic graphs with some additional checks. These additional checks allow exploring a slightly larger state space in less time. We can see that quad-core performance is still similar at more than twice the speed. While the maximum speedup attained is around 4. For more nodes, the maximum speedup is more, and less for less nodes. Thus the bigger the problem space, the bigger the potential speedup.

5. Related Work

There is a vast amount of research on using constraints in software testing. Much of the classic work focused on constraints on inputs with primitive types [2, 7, 10, 20, 21, 33]. Korat [1, 27, 28, 30] and TestEra [29] are among the first frameworks that handle structurally complex inputs. Korat uses imperative predicates. TestEra uses declarative constraints written in the first-order logic Alloy [12]. While Korat and TestEra both generate the same test inputs, the parallel search of PKorat does not directly apply to TestEra, which is based on the Alloy tool-set, which uses off-the-shelf propositional satisfiability (SAT) solvers. A parallel implementation of TestEra requires a parallel SAT solver.

```
class DAG {
  class Node {
    std::vector<Node*> children;
  };
  std::vector<Node*> nodes;
};                              (a)

class JavaClass {
  JavaClass* extends;
  std::vector<JavaInterface*> implements;
};
class JavaInterface {
  std::vector<JavaInterface*> extends;
};
class JavaProgram {
  std::vector<JavaClass*> classes;
  std::vector<JavaInterface*> interfaces;
};                              (b)

public interface I1 {}
public interface I2 {}
public class C1 {}
public class C2 extends C1 implements I1, I2 {}
                                (c)
```

Figure 8. Definition of Classes for Directed Acyclic Graphs (a), for Java Program Generation (b) and one generated Java program (c)

Parallel testing using Korat has previously been explored in the context of test generation as well as test execution [31]. The focus of this paper is a new technique for parallel test generation. PKorat can directly use the previous algorithms for parallel test execution. For test generation, PKorat significantly improves on the previous work, which considered two strategies to parallelize test generation. The first strategy, SEQ-OFF/ON, executes Korat sequentially once to determine an optimal partitioning of the input space such that in a subsequent execution of Korat for the same input space and predicate, each slave explores the same number of candidate inputs. PKorat differs from SEQ-OFF/ON by not requiring an initial sequential run and can in fact be used to optimize SEQ-OFF/ON. The second strategy PAR-OFF/ON uses randomization to fast-forward Korat search on one machine to "guess" equidistant candidate vectors. However, experimental results show little speedup and low efficiency for PAR-OFF/ON. For example, for generating DAGs with 7 nodes, PAR-OFF provides 1.41X speed-up on average for 16 workers, and 8.08X speed-up on average for 1024 workers.

Korat implements a state-less search, in the spirit of VeriSoft [5], where backtracking is achieved through re-execution. The backtracking engine only stores the sequence of choices on the current path but not the entire program state at each choice point. The parallel search of PKorat applies to other analyses implemented using state-less search. To illustrate, consider symbolic execution imple-

mented using the Java PathFinder (JPF) model checker [18], which uses JPF with state matching turned off. The delegation of work to slaves for symbolic execution would be identical to the PKorat technique, although a key challenge in scaling this technique for symbolic execution is to minimize communication among the processors—*path conditions* in symbolic execution can grow significantly and communicating them is likely to be expensive. Similarly, in principle, the parallel search of PKorat applies to combined symbolic/concrete execution [6, 34] and enables PKorat to perform parallel white-box testing. We plan to develop these applications of our parallel search technique in future work.

Parallel search algorithms in general have long been studied [8, 13, 16]. Only recently, however, they have been used for searching state spaces in the area of model checking and program testing. Stern and Dill's parallel Murϕ [35] is an example of a parallel model checker. It keeps the set of visited states shared between parallel workers so that the same parts of the state space are not searched by multiple workers. Keeping this set synchronized between the workers results in expensive communication and thus the algorithm does not scale well.

A similar technique was used by Lerda and Visser [38] to parallelize the Java PathFinder model checker [25]. Parallel version of the SPIN model checker [9] was produced by Lerda and Sisto [24]. More work has been done in load balancing and reducing worker communication in these algorithms [14, 22, 32].

Parallel Randomized State Space Search for JPF by Dwyer et al. [4] takes a different approach with workers exploring randomly different parts of the state space. This often speeds up time to find first error with no worker communication. However when no errors are present, every worker has to explore every state. PKorat differs in that no two workers explore the same state.

6. Conclusions and Future Work

This paper presented PKorat, a highly scalable parallel algorithm for generation of structurally complex test inputs. The algorithm is based upon Korat and makes some key observations in its working to propose a parallel version. Korat is a sequential algorithm that generates all structures within some size bounds that satisfy structural constraints. These constraints are given in the form of an imperative predicate. PKorat explores the same space as Korat, and uses the same imperative predicate, but generates a number of candidate structures at every step. These candidates are distributed by a master processor among parallel slaves resulting in a an efficient parallel version of Korat.

Experimental results show high scalability and efficiency of our algorithm. We have compared the performance of Korat and our parallel algorithm on five different complex structures including red black tress, directed acyclic graphs, and on the problem of automatic generation of Java programs. For structures with a large number of nodes, we attained up to 49.5X speedup on 64 processors. For structures with very few nodes, efficiency drops when increasing processors as there is not enough parallelism to exploit. In all cases, the quad-core performance ranged between 1.9X–2.6X. It is noteworthy, since a dedicated master means that only three slaves were actually generating candidates.

Our future work is to apply the same parallelizing technique to Symbolic Execution. We treat exploring symbolic execution for a given path expression as a work item and new path expressions deduced as the result of current execution as child work items. These child work items are distributed among parallel workers via a central queue. Initial results are encouraging. We are formalizing our approach and performing extensive evaluations to validate it.

Acknowledgements

We thank Darko Marinov, Eddy Reyes, and Bassem Elkarablieh for extensive discussions on parallel analyses. This work was funded in part by the Fulbright Program, the National Science Foundation (awards #CCF-0702680 and #IIS-0438967), and the Naval Undersea Warfare Center.

References

[1] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 123–133, New York, NY, USA, 2002. ACM.

[2] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, 1976.

[3] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 185–194, New York, NY, USA, 2007. ACM.

[4] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.

[5] P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL'97: Proceedings of the Annual ACM Symposium on the Principles of Programming Languages*, Paris, France, Jan. 1997.

[6] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[7] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. *SIGSOFT Softw. Eng. Notes*, 23(2):53–62, 1998.

[8] A. Grama and V. Kumar. State of the art in parallel search techniques for discrete optimization problems. *IEEE Trans. on Knowl. and Data Eng.*, 11(1):28–35, 1999.

[9] G. J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[10] J. C. Huang. An approach to program testing. *ACM Comput. Surv.*, 7(3):113–128, 1975.

[11] Message passing interface (mpi) library. `http://www-unix.mcs.anl.gov/mpi`.

[12] D. Jackson. *Software Abstractions: Logic, Language and Analysis*. The MIT Press, Cambridge, MA, 2006.

[13] V. K. Janakiram, D. P. Agrawal, and R. Mehrotra. A randomized parallel backtracking algorithm. *IEEE Trans. Comput.*, 37(12):1665–1676, 1988.

[14] M. D. Jones and J. Sorber. Parallel search for ltl violations. *Int. J. Softw. Tools Technol. Transf.*, 7(1):31–42, 2005.

[15] Texas advanced computing center (tacc). `http://www.tacc.utexas.edu`.

[16] R. M. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *J. ACM*, 40(3):765–789, 1993.

[17] S. Khurshid and D. Marinov. Testera: Specification-based testing of java programs using sat. *Automated Software Engg.*, 11(4):403–434, 2004.

[18] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS'03: Proceedings of the Conference on Tools and Algorithms for Construction and Analysis of Systems*, Warsaw, Poland, April 2003.

[19] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

[20] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[21] B. Korel. Automated test data generation for programs with procedures. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 209–215, New York, NY, USA, 1996. ACM.

[22] R. Kumar, D. Eric, G. Mercer, D. Quinn, O. Snell, D. Bryan, S. Morse, B. Y. University, B. Y. University, W. Embley, G. R. Bryce, and A. Dean. Load balancing parallel explicit state model checking. *Electr. Notes Theor. Comput. Sci*, 128, 2005.

[23] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing: design and analysis of algorithms*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[24] F. Lerda and R. Sisto. Distributed-memory model checking with spin. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 22–39, London, UK, 1999. Springer-Verlag.

[25] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 80–102, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[26] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[27] D. Marinov. *Automatic testing of software with structurally complex inputs*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2005.

[28] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical report, MIT Computer Science and Artificial Intelligence Laboratory Report MIT -LCS-TR-921, 2003.

[29] D. Marinov and S. Khurshid. Testera: A novel framework for automated testing of java programs. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 22, Washington, DC, USA, 2001. IEEE Computer Society.

[30] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. Korat: A tool for generating structurally complex test inputs. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 771–774, Washington, DC, USA, 2007. IEEE Computer Society.

[31] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov. Parallel test generation and execution with korat. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 135–144, New York, NY, USA, 2007. ACM.

[32] R. Palmer and G. Gopalakrishnan. A distributed partial order reduction algorithm. In *FORTE '02: Proceedings of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems*, page 370, London, UK, 2002. Springer-Verlag.

[33] C. V. Ramamoorthy, S. B. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Trans. Softw. Eng.*, 2(4):293–300, 1976.

[34] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2005.

[35] U. Stern and D. L. Dill. Parallelizing the murphi verifier. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 256–278, London, UK, 1997. Springer-Verlag.

[36] K. Stobie. Model based test generation and abstract state machine language, 2003. `http://www.sasqag.org/pastmeetings/asml.ppt`.

[37] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 133–142, New York, NY, USA, 2004. ACM.

[38] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.