Bits & Bytes

# DEMONSTRATING RANDOM AND PARALLEL ALGORITHMS WITH SPIN

Mordechai (Moti) Ben-Ari and Fatima Kaloti-Hallak

**Recent versions of the Spin model checker support *search diversity*, where the depth-first search of the state space of computation can be randomized. We show how to use this feature to demonstrate the efficacy of random and parallel algorithms when applied to SAT solving: finding a satisfying interpretation of a formula of propositional logic.**

## 1 Introduction

In [2], the first author presented a survey of model checking, showing that it can be effective for teaching concurrency and nondeterminism. In this article, we extend this claim and demonstrate how the Spin model checker can be easily used to demonstrate the efficacy of random and parallel algorithms. We suggest that the reader review [2], in particular Section 8, before continuing.

The example used is that of a naive SAT solver: a program that searches for a satisfying interpretation for a formula of propositional logic. The SAT problem is central in theoretical computer science because it was the first problem to be proved NP-complete. The SAT problem is also of significant practical importance because many applications can be easily encoded in propositional logic. Although the SAT problem is NP-complete, formulas arising in practice can be efficiently solved using a variety of sophisticated algorithmic and programming optimizations. See [5] for an introduction to SAT solvers.

A model checker such as Spin [1] takes as input a nondeterministic finite automaton that models a nondeterministic (or concurrent) computation and a correctness specification written as assertions (or as a formula in temporal logic). It searches the state space of the execution of the automaton for a counterexample: a state that falsifies an assertion. The search is performed depth-first; since the automaton is nondeterministic, at each node there are several outgoing branches. Normally, the search tries these in a fixed order, but recent versions of Spin support trying the branches in random order. As you will see, the results can be surprising.

## 2 Implementing a Naive SAT Solver in Spin

The input to a SAT solver is a formula in *conjunctive normal form*: a conjunction of disjunctions called clauses. For example:

```
(a V b) & (~a V ~b) & (a V ~c) & (~a V c) &
(c V ~d) & (~c V d) & (b V ~d) & (~b V d)
```

It is straightforward to implement a naive SAT solver for this set of clauses in Promela—the modeling language of Spin—using nondeterministic if-statements to choose an assignment:

```
active proctype sat() {
  bool a, b, c, d, result;

  /* Select an assignment nondeterministically */
  if :: a = true :: a = false fi;
  if :: b = true :: b = false fi;
  if :: c = true :: c = false fi;
  if :: d = true :: d = false fi;

  /* Compute the truth value of the set of clauses */
  result = (a || b) && (!a || !b) && (a || !c) &&
           (!a || c) && (c || !d) && (!c || d) &&
           (b || !d) && (!b || d);

  printf("Result = %d\n", result);
  assert(!result)
}
```

When a verification of this program is performed in Spin, the model checker performs a depth-first search, checking all possible assignments of truth values to the atomic propositions. If a satisfying interpretation is found, the assertion is violated (`result` is true, so `!result` is false) and the verification terminates. Otherwise, the model checker backtracks to continue the search for a counterexample. Since this set of clauses is unsatisfiable, a verification will

terminate successfully after searching the entire state space. See Sidebar 1 that summarizes the relevant concepts.

### SIDEBAR 1

**Efficient *nondeterministic* algorithm (generate and check)**

- Generate an assignment: `a=false, b=true, c=true, d=false`.
- Evaluate the formula (this takes very little time) and check if `result` is true:

```
result =
      (false V true) & ... (~true V false) =
      true & ... & false =
      false
```

- By definition of a nondeterministic algorithm: if *there exists* a computation where `result` is true, then the formula is satisfiable.

**Inefficient *deterministic* implementation (brute-force search)**

- *For all* of the $2^n$ assignments of true or false to the *n* atomic propositions, compute `result`.
- Use a fixed ordering (such as lexicographic ordering) to ensure that you check all possible assignments.
- If the formula is satisfiable, a satisfying assignment will eventually be found, but in the worst case the formula must be evaluated for all $2^n$ assignments.

## 3 Tseitin Clauses

For this small set of clauses, the verification terminates immediately, but the set is a member of a family of sets of clauses called *Tseitin clauses*, which are built from arbitrary large connected undirected graphs. The details of the construction are beyond the scope of this note; the interested reader is referred to [3, Section 4.5].

The above set of clauses was generated from the complete bipartite graph $K_{2,2}$. A program was written to generate Promela programs for Tseitin clauses corresponding to $K_{n,n}$ for any n and to generate *variants* of the programs, where a random literal is complemented in each variant, thereby making the set of clauses satisfiable.

For the unsatisfiable sets of clauses, the search must try all possible assignments of true or false to the atomic propositions and this demonstrates the *exponential* nature of the naive algorithm for SAT. The verification of the programs for the clauses associated with $K_{2,2}$, $K_{3,3}$ and $K_{4,4}$ terminated immediately, whereas for the clauses associated with $K_{5,5}$ the verification took 48 seconds. An attempt to verify the clauses associated with $K_{6,6}$ terminated for lack of memory.[1]

## 4 SAT solving with Search Diversity

Holzmann, Joshi and Groce [4] showed that time, not memory, is currently the limiting factor is the application of the Spin model checker to large problems. They suggest using *search diversity* and *parallelism*: running many verifications in parallel on a multicore machine, where each run differs in the way the search is conducted. As noted in [4], one need not actually run several verifications in parallel to become convinced that search diversity will work. It is sufficient to run a number of verifications equal to the number of processors that are assumed to be available and to examine the execution times.

We carried out an experiment with search diversity applied to the naive SAT solver. We configured the verifier to search the transitions from each state randomly and ran each verification with several seeds. The unsatisfiable Tseitin clauses give upper bounds on the resources needed, while the satisfiable variants are intended to represent practical problems that have a solution that we are looking for. Eight variants and eight verifications (with different seeds) for each variant were run for Tseitin clauses corresponding to $K_{5,5}$ and $K_{6,6}$.

As noted above, the unsatisfiable Tseitin clauses for $K_{5,5}$ took 48 seconds to verify. Out of the eight satisfiable variants, seven were verified in just a few seconds, but one variant showed more interesting behavior. The verification of this variant using the non-random algorithm took 37 seconds, while the random verifications finished in the following times.

11, 6, 6, 39, 35, 2, 12, 39.

Suppose that you have a computer with four cores running the random algorithm with four different seeds. *At worst*, the running times would be 39, 35, 12, 39, so the program would output an answer in only 12 seconds, a third of the time of the non-random sequential algorithm. With a bit of luck, the answer would be received in 6 or even 2 seconds. With eight cores, of course, the answer would be received in 2 seconds. While these improvements might not seem impressive, a real problem could see its running time reduced from 37 hours (you would probably cancel the run before receiving the result ...) to a reasonable 2 hours!

The clauses for $K_{6,6}$ demonstrate more realistic scenarios. Recall that the unsatisfiable clauses could not be verified, so we are not surprised that the results for the satisfiable variants are not uniformly positive. Nevertheless, they are encouraging in some cases, as shown in Table 1.

The verifications of variants 3, 5 and 7 remain infeasible, but those for variants 2, 4, 6, 8 would terminate on a four-core processor because any choice of four out of the eight seeds includes at

**TABLE 1**

| Variant | Time (– = out of memory) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **1** | – | – | – | – | – | – | 0 | – |
| **2** | 21 | 0 | 0 | – | 21 | 0 | 21 | 21 |
| **3** | – | – | – | – | – | – | – | – |
| **4** | – | 0 | 0 | 0 | 21 | – | 0 | 0 |
| **5** | – | – | – | – | – | – | – | – |
| **6** | 21 | 20 | 0 | – | 0 | 21 | – | – |
| **7** | – | – | – | – | – | – | – | – |
| **8** | 0 | 0 | 0 | 21 | – | 0 | – | – |

[1] All times are rounded to the nearest second. We used a garden-variety Windows PC like the one a student might use: a low-end Intel i3 processor with two cores and 4 GB of memory. If you use more powerful computers, simply scale up the problem to use $K_{n,n}$ for larger *n*.

least one that terminates in at most 21 seconds. The verification of variant 1 would succeed on an eight-core processor because for one seed out of the eight, the verification terminates immediately. Sidebar 2 expresses random search and parallel search diversity in terms of throwing dice.

### SIDEBAR 2

**Random search diversity**

- Compute result for all the $2^n$ assignments taken in some *random order*.
- If you get "lucky", the random search will find a satisfying assignment faster than a search in a fixed ordering!
- For example, if the fifth assignment out of six is satisfying, then the fixed-order search will perform 5 steps to find it, but if you throw a die to determine where to start the search, you might get lucky and start from 4 or even 5.

**Parallel random search diversity**

- On a multicore computer, compute result *in parallel* for different random orders of the $2^n$ assignments.
- This increases your chance of getting "lucky" quickly.
- For example, if you throw four dice at the same time, the probability of getting a 5 is much higher ($1-(5/6)^4=62\%$) than if you just throw one die (17%).

## 5 From Search Diversity to Parallelism

Even on a two-core processor, parallelism is easy to demonstrate by opening multiple command windows. Consider the variant of the program for the clauses associated with $K_{5,5}$ that took 37 seconds to verify using the standard search method. It was run simultaneously in two windows: one with the seed that led to a 35-second execution and one with the seed that needed only about 12 seconds. Since they ran in parallel on the two cores, the faster verification finished in just over 12 seconds, while the slower verification continued to run. However, when two slow verifications were initiated in parallel with the fast verification, the latter's run time increased to almost 18 seconds, indicating that it was time-sharing the cores with the other programs.

## 6 Conclusion

Random algorithms are rather unintuitive for students brought up to think of algorithms as sequential deterministic procedures, or as a set of deterministic processes in the case of concurrency. Furthermore, while concurrency as the time-shared execution of high-level processes is also familiar, demonstrating speedup from parallelism is more difficult. We have shown how the support for search diversity in Spin makes it very easy to demonstrate both randomness and parallelism for solving a nondeterministic algorithm.

The programs described in this paper are open-source and can be downloaded from: http://code.google.com/p/mlcs/. **Ir**

### References


[1] Ben-Ari, M. *Principles of the Spin Model Checker*. Springer, London, 2008.

[2] Ben-Ari, M. A primer on model checking. *ACM Inroads*, 1(1):40–47, 2010.

[3] Ben-Ari, M. *Mathematical Logic for Computer Science (Third Edition)*. Springer, London, 2012.

[4] Holzmann, G.J., Joshi, R., and Groce, A. Swarm verification techniques. *IEEE Transactions on SoftwareEngineering*, 37(6):845–857, 2011.

[5] Malik, S. and Zhang, L. Boolean satisfiability: From theoretical hardness to practical success. *Communications ACM*, 52(8):76–82, 2009.



**MORDECHAI (MOTI) BEN-ARI**
Department of Science Teaching
Weizmann Institute of Science, Rehovot 76100 Israel
*moti.ben-ari@weizmann.ac.il*

**FATIMA KALOTI-HALLAK**
Department of Science Teaching
Weizmann Institute of Science, Rehovot 76100 Israel
*fatima.hallak@weizmann.ac.il*



**Categories and Subject Descriptors:** D.2.4 [Software/Program Verification (F.3.1)]-Model checking; D.1.3 [Concurrent Programming]-Parallel programming
**General Terms:** Algorithms
**Keywords:** model checking, Spin, random algorithm, multicore

**DOI:** 10.1145/2339055.2339069