Multi-Core On-The-Fly Saturation

Tom van Dijk[1,2*], Jeroen Meijer[1†], and Jaco van de Pol[3,1]

[1] Formal Methods and Tools, University of Twente, the Netherlands
t.vandijk@utwente.nl
[2] Formal Models and Verification, Johannes Kepler University, Austria
[3] Department of Computer Science, University of Aarhus, Denmark

Abstract. Saturation is an efficient exploration order for computing the set of reachable states symbolically. Attempts to parallelize saturation have so far resulted in limited speedup. We demonstrate for the first time that on-the-fly symbolic saturation can be successfully parallelized at a large scale. To this end, we implemented saturation in Sylvan's multi-core decision diagrams used by the LTSmin model checker.
We report extensive experiments, measuring the speedup of parallel symbolic saturation on a 48-core machine, and compare it with the speedup of parallel symbolic BFS and chaining. We find that the parallel scalability varies from quite modest to excellent. We also compared the speedup of on-the-fly saturation and saturation for pre-learned transition relations. Finally, we compared our implementation of saturation with the existing sequential implementation based on Meddly.
The empirical evaluation uses Petri nets from the model checking contest, but thanks to the architecture of LTSmin, parallel on-the-fly saturation is now available to multiple specification languages.

1 Introduction

Model checking is an exhaustive algorithm to verify that a finite model of a concurrent system satisfies certain temporal properties. The main challenge is to handle the large state space, resulting from the combination of parallel components. Symbolic model checking exploits regularities in the set of reachable states, by storing this set concisely in a decision diagram. In asynchronous systems, transitions have locality, i.e. they affect only a small part of the state vector. This locality is exploited in the saturation strategy, which is probably the most efficient strategy to compute the set of reachable states.

In this paper, we investigate the efficiency and speedup of a new parallel implementation of saturation, aiming at a multi-core, shared-memory implementation. The implementation is carried out in the parallel decision diagram framework Sylvan [16], in the language-independent model checker LTSmin [22]. We empirically evaluate the speedup of parallel saturation on Petri nets from the Model Checking Contest [24], running the algorithm on up to 48 cores.

[*] Supported by FWF, NFN Grant S11408-N23 (RiSE)
[†] Supported by STW SUMBAT Grant 13859

1.1 Related Work

The saturation strategy has been developed and improved by Ciardo et al. We refer to [13] for an extensive description of the algorithm. Saturation derives its efficiency from firing all local transitions that apply at a certain level of the decision diagram, before proceeding to the next higher level. An important step in the development of the saturation algorithm allows on-the-fly generation of the transition relations, without knowing the cardinality of the state variable domains in advance [12]. This is essential to implement saturation in LTSmin, which is based on the PINS interface to discover transitions on-the-fly.

Since saturation obtains its efficiency from a restrictive firing order, it seems inherently sequential. Yet the problem of parallelising saturation has been studied intensively. The first attempt, Saturation NOW [9], used a network of PCs. This version could exploit the collective memory of all PCs, but due to the sequential procedure, no speedup was achieved. By firing local transitions speculatively (but with care to avoid memory waste), some speedup has been achieved [10]. More relevant to our work is the parallelisation of saturation for a shared memory architecture [20]. The authors used CILK to schedule parallel work originating from firing multiple transitions at the same level. They reported some speedup on a dual-core machine, at the expense of a serious memory increase. Their method also required to precompute the transition relation. An improvement of the parallel synchronisation mechanism was provided in [31]. They reported a parallel speedup of 2× on 4 CPUs. Moreover, their implementation supports learning the transition relation on-the-fly. Still, the successful parallelisation of saturation remained widely open, as indicated by Ciardo [14]: “Parallel symbolic state-space exploration is difficult, but what is the alternative?”

For an extensive overview of parallel decision diagrams on various hardware architectures, see [15]. Here we mention some other approaches to parallel symbolic model checking, different from saturation for reachability analysis. First, Grumberg and her team [21] designed a parallel BDD package based on vertical partitioning. Each worker maintains its own sub-BDD. Workers exchange BDD nodes over the network. They reported some speedup on 32 PCs for BDD based model checking under the BFS strategy. The Sylvan [16] multi-core decision diagram package supports symbolic on-the-fly reachability analysis, as well as bisimulation minimisation [17]. Oortwijn [28] experimented with a heterogeneous distributed/multi-core architecture, by porting Sylvan’s architecture to RDMA over MPI, running symbolic reachability on 480 cores spread over 32 PCs and reporting speedups of BFS symbolic reachability up to 50. Finally, we mention some applications of saturation beyond reachability, such as model checking CTL [32] and detecting strongly connected components to detect fair cycles [33].

1.2 Contribution

Here we show that implementing saturation on top of the multi-core decision diagram framework Sylvan [16] yields a considerable speedup in a shared-memory setting of up to 32.5× on 48 cores with pre-learned transition relations, and 52.2× with on-the-fly transition learning.

By design decision, our implementation reuses several features provided by Sylvan, such as: its own fine-grained, work-stealing framework Lace [18], its implementation of both BDDs (Binary Decision Diagrams) and LDDs (a List-implementation of Multiway Decision Diagrams), its concurrent unique table and operations cache, and finally, its parallel operations like set union and relational product. As a consequence, the pseudocode of the algorithm and additional code for saturation is quite small, and orthogonal to other BDD features. To improve orthogonality with the existing decision diagrams, we deviated from the standard presentation of saturation [13]: we never update BDD nodes in situ, and we eliminated the mutual recursion between saturation and the BDD operations for relational product to fire transitions.

The implementation is available in the open-source high-performance model checking tool LTSmin [22], with its language-agnostic interface, Partitioned Next-State Interface (PINS) [5, 22, 25]. Here, a specification basically provides a next-state function equipped with dependency information, from which LTSmin can derive locality information. We fully support the flexible method of learning the transition relation on-the-fly during saturation [12]. As a consequence, our contribution extends the tool LTSmin with saturation for various specification languages, like Promela, DVE, Petri nets, mCRL2, and languages supported by the ProB model checker. See section 4 on how to use saturation in LTSmin.

The experiments with saturation in Sylvan are carried out in LTSmin as well. We used Petri nets from the MCC competition. Our experimental design has been carefully set up in order to facilitate fair comparisons. Besides learning the transition relation on-the-fly, we also pre-learned them in order to measure the overhead of learning, and eliminating its effect in comparisons. It is well known that the variable ordering has a large effect on the BDD sizes [29]. Hence, our experiments are based on two of the best static variable orderings known, Sloan [26] and Force [1]. In particular, our experiments measure and compare:

- The performance of our parallel algorithm with one worker, compared to a state-of-the art sequential implementation of saturation in Meddly [4].
- The parallel speedup of our algorithm on 16 cores, and for specific examples up to 48 cores.
- The efficiency and speedup of saturation compared to the BFS and chaining strategies for reachability analysis.
- The effect of choosing Binary Decision Diagrams or List Decision Diagrams.
- The effect of choosing Sloan or Force to compute static variable orders.

2 Preliminaries

This paper proposes an algorithm for decision diagrams to perform the fixed point application of multiple transition relations according to the saturation strategy, combined with on-the-fly transition learning as implemented in LTSmin. We briefly review these concepts in the following.

2.1 Partitioned transition systems

A transition system (TS) is a tuple $(S, \rightarrow, s^0)$, where S is a set of states, $\rightarrow \subseteq S \times S$ is a transition relation and $s^0 \in S$ is the initial state. We define $\rightarrow^*$ to be the reflexive and transitive closure of $\rightarrow$. The set of reachable states is $R = \{s \in S \mid s^0 \rightarrow^* s\}$. The goal of this work is to compute R via a novel multi-core saturation strategy.

In this paper, we evaluate multi-core saturation using Petri nets. Figure 1 shows an example of a (safe) Petri net. We show its initial marking, which is the initial state. A Petri net transition can fire if there is a token in each of its source places. On firing, these tokens are consumed and tokens in each target place are generated. For example, t_1 will produce one token in both p_2 and p_5, if there is a token in p_4. Transition t_6 requires a token in both p_3 and p_1 to fire. The markings of this Petri net form the states of the corresponding TS, so here $|S| = 2^5 = 32$. From the initial marking shown, four more markings are reachable, connected by 10 enabled transition firings. This means $|R| = 5$, and $|\rightarrow| = 10$.

Notice that transitions in Petri nets are quite local; transitions consume from, and produce into relatively few places. The firing of a Petri net transition is called an event and the number of involved places is known as the *degree of event locality*. This notion is easily defined for other asynchronous specification languages and can be computed by a simple control flow graph analysis.

To exploit event locality, saturation requires a disjunctive partitioning of the transition relation $\rightarrow$, giving rise to a Partitioned Transition System (PTS). In a PTS, states are vectors of length N, and $\rightarrow$ is partitioned as a union of M transition groups. A natural way to partition a Petri net is by viewing each transition as a transition group. For Figure 1 this means we have $N = 5$ and $M = 6$. After disjunctive partitioning, each transition group depends on very few entries of the state vector. This allows for efficiently computing the reachable state space for the large class of asynchronous specification languages. LTSmin supports commonly used specification languages, like DVE, mCRL2, Promela, PNML for Petri nets, and languages supported by ProB.

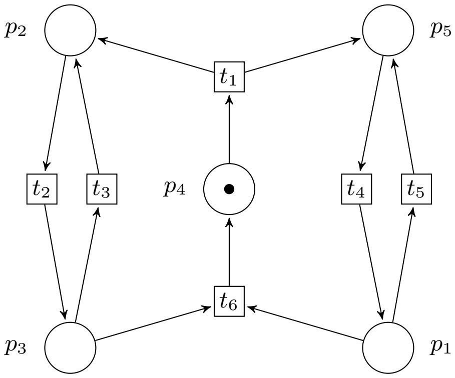
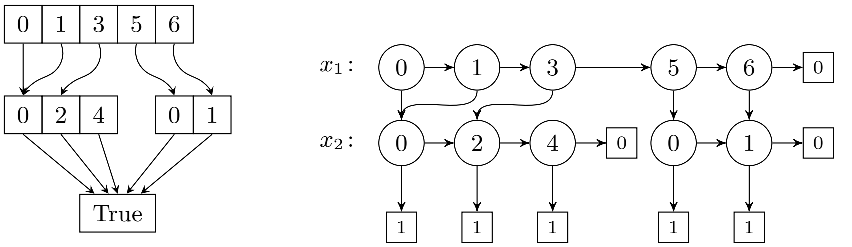


Fig. 1: Example Petri net



(a) LDD as array

(b) Same LDD, internal linked-list representation

,

Fig. 2: LDD for $\{\langle 0,0\rangle,\langle 0,2\rangle,\langle 0,4\rangle,\langle 1,0\rangle,\langle 1,2\rangle,\langle 1,4\rangle,\langle 3,2\rangle,\langle 3,4\rangle,\langle 5,0\rangle,\langle 5,1\rangle,\langle 6,1\rangle\}$.

2.2 Decision diagrams

Binary decision diagrams (BDDs) are a concise and canonical representation of Boolean functions $\mathbb{B}^N \to \mathbb{B}$ [7]. A BDD is a rooted directed acyclic graph with leaves 0 and 1. Each internal node v has a variable label x_i, denoted by var(v), and two outgoing edges labeled 0 and 1, denoted by low(v) and high(v). The efficiency of *reduced, ordered* BDDs is achieved by minimizing the structure with some invariants: The BDD may neither contain *equivalent nodes*, with the same var(v), low(v) and high(v), nor *redundant nodes*, with low(v) $=$ *high*(v). Also, the variables must occur according to a fixed ordering along each path.

Multi-valued or multiway decision diagrams (MDDs) generalize BDDs to finite domains ($\mathbb{N}^N \to \mathbb{B}$). Each internal MDD node with variable x_i now has n_i outgoing edges, labeled 0 to $n_i - 1$. We use quasi-reduced MDDs with sparse nodes. In the sparse representation, values with edges to leaf 0 are skipped from MDD nodes, so outgoing edges must be explicitly labeled with remaining domain values. Contrary to BDDs, MDDs are usually “quasi-reduced”, meaning that variables are never skipped. In that case, the variable x_i can be derived from the depth of the MDD, so it is not stored.

A variation of MDDs are list decision diagrams (LDDs) [5, 16], where sparse MDD nodes are represented as a linked list. See Figure 2 for two visual representations of the same LDD. Each LDD node contains a value, a “down” edge for the corresponding child, and a “right” edge pointing to the next element in the list. Each list ends with the leaf 0 and each path from the root downwards ends with the leaf 1. The values in an LDD are strictly ordered, i.e., the values must increase to the “right”.

LDD nodes have the advantage that common suffixes can be shared: The MDD for Figure 2a requires two more nodes, one for $[2, 4]$ and one for $[1]$, because edges can only point to an entire MDD node. LDDs suffer from an increased memory footprint and inferior memory locality, but their memory management is simpler, since each LDD node has a fixed small size.

$$\begin{array}{c|ccccc|} & p_1 & p_2 & p_3 & p_4 & p_5 \\ \hline t_1 & 0 & \mathbf{1} & 0 & \mathbf{1} & \mathbf{1} \\ t_2 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 \\ t_3 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 \\ t_4 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} \\ t_5 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} \\ t_6 & \mathbf{1} & 0 & \mathbf{1} & \mathbf{1} & 0 \end{array}$$

(a) Natural order

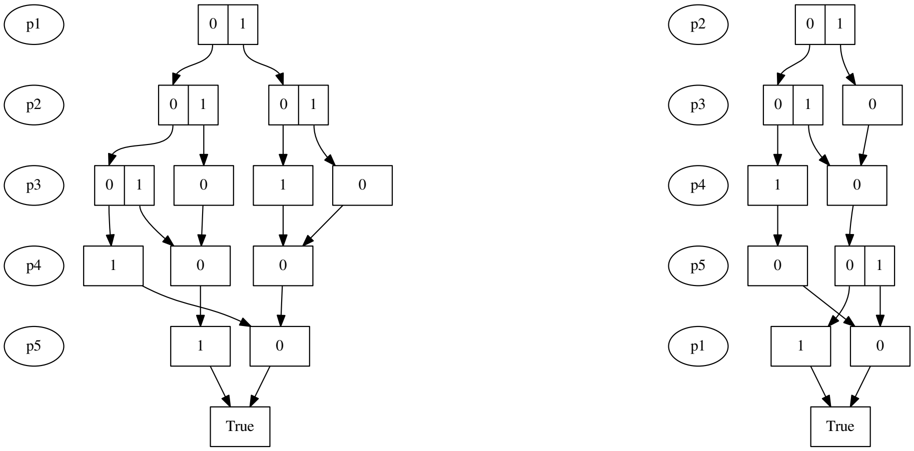
$$\begin{array}{c|ccccc|} & p_2 & p_3 & p_4 & p_5 & p_1 \\ \hline t_2 & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 \\ t_3 & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 \\ t_1 & \mathbf{1} & 0 & \mathbf{1} & \mathbf{1} & 0 \\ t_6 & 0 & \mathbf{1} & \mathbf{1} & 0 & \mathbf{1} \\ t_4 & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} \\ t_5 & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} \end{array}$$

(b) Optimized order

Fig. 3: Dependency matrices of Figure 1.

2.3 Variable orders and event locality

Good variable orders are crucial for efficient operations on decision diagrams. The syntactic variable order from the specification is often inadequate for the saturation algorithm to perform well. Hence, finding a good variable order is necessary. Variable reordering algorithms use heuristics based on event locality. The locality of events can be illustrated with dependency matrices. The size of those matrices is $M \times N$, where M is the number of transition groups, and N is the length of the state vector. The order of columns in dependency matrices determines the order of variables in the DD. Figure 3a shows the natural order on places in Figure 1. A measure of event locality is called *event span* [29]. Lower event span is correlated to a lower number of nodes in decision diagrams. This can be seen in LDDs in Figures 4a and 4b that are ordered according to columns in Figures 3a and 3b respectively.



(a) Natural order

(b) Optimized order

Fig. 4: Reachable states as LDDs with different orders on places

Event span is defined as the sum over all rows of the distance from the leftmost non-zero column to the rightmost non-zero column. The event span of Figure 3a is 22 ($= 4 + 2 + 2 + 5 + 5 + 4$); the event span of Figure 3b is 16, which is better. Optimizing the event span and thus variable order of DDs is NP-complete [6], yet there are heuristic approaches that run in subquadratic time and provide good enough orders. Commonly used algorithms are Noack [27], Force [1] and Sloan [30]. Noack creates a permutation of variables by iteratively minimizing some objective function. The Force algorithm acts as if there are springs in between nonzeros in the dependency matrix, and tries to minimize the average tension among them. Sloan tries to minimize the profile of matrices. In short, profile is the symmetric counterpart to event span. For a more detailed overview of these algorithms see [3]. In our empirical evaluation we use both Sloan and Force, because these have been shown to give the best results [2, 26].

2.4 The Saturation strategy

The saturation strategy for reachability analysis, i.e., the transitive closure of transition relations applied to some set of states, was first proposed by Ciardo et al. See for an overview [11, 13]. Saturation was combined with on-the-fly transition learning in [12]. Besides reachability, saturation has also been applied to CTL model checking [32] and in checking fairness constraints with strongly connected components [33].

Saturation is well-studied. The core idea is to always fire enabled transitions at the lower levels in the decision diagram, before proceeding to the next level. This tends to keep the intermediate BDD sizes much smaller than for instance the breadth-first exploration strategy. This is in particular the case for asynchronous systems, where transitions exhibit locality. There is also a major influence from the variable reordering: if the variables involved in a transition are grouped together, then this transition only affects adjacent levels in the decision diagram.

We refer to [13] for a precise description of saturation. Our implementation deviates from the standard presentation in three ways. First, we implemented saturation for LDDs and BDDs, instead of MDDs. Next, we never update nodes in the LDD forest in situ; instead, we always create new nodes. Finally, the standard representation has a mutual recursion between *saturation* and *firing transitions*. Instead, we fire transition using the existing function for relational product, which is called from our saturation algorithm. As a consequence, the extension with saturation becomes more orthogonal to the specific decision diagram implementation. We refer to Section 3 for a detailed description of our algorithm. We show in Section 5 that these design decisions do not introduce computational overhead.

3 Multi-core saturation algorithm

To access the three elements of an LDD node x, Sylvan [16] provides the functions `value`(x), `down`(x), `right`(x). To create or retrieve a unique LDD node using the hash table, Sylvan provides `LookupLDDNode`$(\mathit{value}, \mathit{down}, \mathit{right})$.

Furthermore, Sylvan provides several operations on LDDs that we use to implement reachability algorithms, such as `union`(A, B) to compute the set union $A \cup B$ and `minus`(A, B) to compute the set difference $A \setminus B$. For transition relations, Sylvan provides an operation `relprod`(S, R) to compute the successors of S with transition relation R, and an operation `relprodunion`(S, R) that computes `union`$(S,$ `relprod`$(S, R))$, i.e., computing the successors and adding them to the given set of states, in one operation. All these operations are internally parallelized, as described in [16].

We implement multi-core saturation as in Algorithm 1. We have a transition relation disjunctively partitioned into M relations $R_0 \dots R_{M-1}$. These relations are sorted by the level (depth) of the decision diagram where they are applied, which is the first level touched by the relation. We say that relation R_i is applied at depth d_i. We identify the current next relation with a number k, $0 \leq k \leq M$, where $k = M$ denotes "no next relation". Decision diagram levels are sequentially numbered with 0 for the root level.

The `saturate` algorithm is given the initial set of states S and the initial next transition relation $k = 0$ and the initial decision diagram level $d = 0$. The algorithm is a straightforward implementation of saturation. First we check the easy cases where we reach either the end of an LDD list, where $S = 0$, or the bottom of the decision diagram, where $S = 1$. If there are no more transition relations to apply, then $k = M$ and we can simply return S. When we arrive at line 4, the operation is not trivial and we consult the operation cache.

```
    global: $M$ transition relations $R_0 \dots R_{M-1}$ starting at depths $d_0 \dots d_{M-1}$
 1  def saturate($S$, $k$, $d$):
 2      if $S = 0 \vee S = 1$ : return $S$
 3      if $k = M$ : return $S$
 4      if result $\leftarrow$ cache$[(S, k, d)]$ : return result
 5      if $d = d_k$ :
 6          $k' \leftarrow$ next relation $k < k' < M$ where $d_{k'} \neq d$, or $M$
 7          while $S$ changes :
 8              $S \leftarrow$ saturate($S$, $k'$, $d$)
 9              for $i \in [k, k')$ :  $S \leftarrow$ relprodunion($S$, $R_i$)
10          result $\leftarrow S$
11      else:
12          do in parallel:
13              right $\leftarrow$ saturate(right($S$), $k$, $d$)
14              down $\leftarrow$ saturate(down($S$), $k$, $d + 1$)
15          result $\leftarrow$ LookupLDDNode(value($S$), down, right)
16      cache$[(S, k, d)] \leftarrow$ result
17      return result
```

Algorithm 1: The multi-core saturation algorithm, which, given a set of states S and next transition relation k and current decision diagram depth d, exhaustively applies all transition relations $R_k \dots R_{M-1}$ using the saturation strategy.

If the result of this operation was not already in the cache, then we check whether we have relations at the current level. Since the relations are sorted by the level where they must be applied, we compare the current level d with the level d_k of the next relation k. If we have relations at the current level, then we perform the fixed point computation where we first saturate S for the remaining relations, starting at relation k', which is the first relation that must be applied on a deeper level than d, and then apply the relations of the current level, that is, all R_i where $k \leq i < k'$. If no relations match the current level, then we compute in parallel the results of the suboperations for the LDD of successor "right" and for the LDD of successor "down". After obtaining these sub results, we use `LookupLDDNode` to compute the final result for this LDD node. Finally, we store this result in the operation cache and return it.

The **do in parallel** keyword is implemented with the work-stealing framework Lace [18], which is embedded in Sylvan [16] and offers the primitives `spawn` and `sync` to create subtasks and wait for their completion. The implementation using `spawn` and `sync` of lines 12–14 is as follows.

```
12 spawn(saturate(right($S$), $k$, $d$))
13 down ← saturate(down($S$), $k$, $d+1$)
14 right ← sync()
```

The implementation of multi-core saturation for BDDs is identical, except that we parallelize on the "then" and "else" successors of a BDD node, instead of on the "down" and "right" successors of an LDD node.

To add on-the-fly transition relation learning to this algorithm, we simply modify the loop at line 9 as follows:

```
 9 for $i \in [k, k')$ :
10     learn-transitions($S$, $i$, $d$)
11     $S$ ← relprodunion($S$, $R_i$)
```

The `learn-transitions` function provided by LTSmin updates relation i given a set of states S. The function first restricts S to so-called short states S^i, which is the projection of S on the state variables that are touched by relation i. Then it calls the `next-state` function of the PINS interface for each new short state and it updates R_i with the new transitions.

Updating transition relations from multiple threads is not completely trivial. LTSmin solves this using lock-free programming with the compare-and-swap operation. After collecting all new transitions, LTSmin computes the union with the known transitions and uses compare-and-swap to update the global relation; if this fails, the union is repeated with the new known transitions.

4 Contributed tools

We present several new tools and extensions to existing tools produced in this work. The new tools support experiments and comparisons between various DD formats. The extension to Sylvan and LTSmin provides end-users with multi-core saturation for reachability analysis.

4.1 Tools for experimental purposes

For the empirical evaluation, we need to isolate the reachability analysis of a given LDD (or BDD or MDD). To that end, we implemented three small tools that only compute the set of reachable states, namely `lddmc` for LDDs, `bddmc` for BDDs and `medmc` for MDDs using the library Meddly. These tools are given an input file representing the model, compute the set of reachable states, and report the number of states and the required time to compute all reachable states. Additionally we provide the tools `ldd2bdd` and `ldd2meddly` that convert an LDD file to a BDD file and to an MDD file. The LDD input files are generated using LTSmin (see below). These tools can all be found online[4].

4.2 Tools for on-the-fly multi-core saturation

On-the-fly multi-core saturation is implemented in the LTSmin toolset, which can be found online[5]. The examples in this section are also online[6]. On-the-fly multi-core saturation for Petri nets is available in LTSmin's tool pnml2lts-sym. This tool computes all reachable markings with parallel saturation. The command line to run it on Figure 1 is pnml2lts-sym pnml/example.pnml --saturation=sat. The tool reports: pnml2lts-sym: `state space has 5 states, 16 nodes`. Additionally, it appears the final LDD has 16 nodes.

Here the syntactic variable order of the places in pnml/example.pnml is used. To use a better variable order, the option `-r` is added to the command line. For instance adding -rf runs *Force*, while -rbs runs *Sloan*'s algorithm (as implemented in the well-known Boost library). Running pnml2lts-sym pnml/example.pnml --saturation=sat -rf reports that the final LDD has only 12 nodes.

The naming convention of LTSmin's binaries follows the Partitioned Next-State Interface (PINS) architecture [5, 22, 25]. PINS forms a bridge between several language front-ends and algorithmic back-ends. Consequently, besides pnml2lts-sym, LTSmin also provides {pnml,dve,prom}2lts-{dist,mc,sym} and several other combinations. These binaries generate the state space for the languages PNML, DVE and Promela, by means of distributed explicit-state, multi-core explicit-state and multi-core symbolic algorithms, respectively. Additionally, LTSmin supports checking for deadlocks and invariants, and verifying LTL properties and μ-calculus formulas. In this work we focus on state space generation with the symbolic back-end only.

We now demonstrate multi-core saturation for Promela models. Consider the file Promela/garp_1b2a.prm which is an implementation of the GARP protocol [23]. To compute the reachable state space with the proposed algorithm and Force order, run: prom2lts-sym --saturation=sat Promela/garp_1b2a.prm -rf. On a consumer laptop with 8 hardware threads, LTSmin reports 385,000,995,634 reachable states within 1 minute. To run the example with a single worker, run

[4] `https://github.com/trolando/sylvan`

[5] `https://github.com/utwente-fmt/ltsmin`

[6] `https://github.com/trolando/ParallelSaturationExperiments`

prom2lts-sym –saturation=sat Promela/garp_1b2a.prm -rf --lace-workers=1. On the same laptop, the algorithm runs in 4 minutes with 1 worker. We thus have a speedup of 4× with 8 workers for symbolic saturation on a Promela model.

5 Empirical Evaluation

Our goal with the empirical study is five-fold. *First*, we compare our parallel implementation with only 1 core to the purely sequential implementation of the MDD library Meddly [4], in order to determine whether our implementation is competitive with the state-of-the-art. *Second*, we study parallel scalability up to 16 cores for all models and up to 48 cores with a small selection of models. *Third*, we compare parallel saturation with LDDs to parallel saturation with ordinary BDDs, to see if we get similar results with BDDs. *Fourth*, we compare parallel saturation without on-the-fly transition learning to on-the-fly parallel saturation, to see the effects of on-the-fly transition learning on the performance of the algorithm. *Fifth*, we compare parallel saturation with other reachability strategies, namely chaining and BFS, to confirm whether saturation is indeed a better strategy than chaining and BFS.

To perform this evaluation, we use the P/T Petri net benchmarks obtained from the Model Checking Contest 2016 [24]. These are 491 models in total, stored in PNML files. We use parallel on-the-fly saturation (in LTSmin) with a generous timeout of 1 hour to obtain LDD files of the models, using the Force variable ordering and using the Sloan variable ordering. In total, 413 of potentially 982 LDD files were generated. These LDD files simply store the list decision diagrams of the initial states and of all transition relations. We convert the LDD files to BDD files (binary decision diagrams) with an optimal number of binary variables. We also convert the LDD files to MDD files for the experiments using Meddly. This ensures that all solvers have *the same input model with the same variable order*.

See Table 1 for the list of solving methods. As described in Sec. 4, we implement the tools `lddmc`, `bddmc` and `medmc` to isolate reachability computation for the purposes of this comparison, using respectively the LDDs and BDDs of Sylvan and the MDDs of Meddly. The on-the-fly parallel saturation using LDDs is performed

Method	**Tool**	**Description**	**Input**	**Parallel**	**OTF**
otf-ldd-sat	`pnml2lts-sym`	saturation	PNML	✓	✓
ldd-sat	`lddmc`	saturation	LDD	✓	
ldd-chaining	`lddmc`	chaining	LDD	✓	
ldd-bfs	`lddmc`	BFS	LDD	✓	
bdd-sat	`bddmc`	saturation	BDD	✓	
mdd-sat	`medmc`	saturation in Meddly	MDD		

Table 1: The six solving methods that we use in the empirical evaluation. Five methods are parallelized and one method is on-the-fly.

Method	**Number of solved models with # workers**					
	1	2	4	8	16	Any
otf-ldd-sat	387	397	399	404	407	408
ldd-sat	388	393	399	402	402	404
ldd-chaining	351	354	360	367	371	371
ldd-bfs	325	331	347	360	362	362
bdd-sat	395	396	401	402	403	405
mdd-sat	375	–	–	–	–	375

Table 2: Number of benchmarks (out of 413) solved within 20 minutes with each method with the given number of workers.

with the `pnml2lts-sym` tool of LTSMIN. We use the command line `pnml2lts-sym ORDER --lace-workers=WORKERS --saturation=sat FILE`, where `ORDER` is `-rf` for Force and `-rbs` for Sloan and `WORKERS` is a number from the set $\{1, 2, 4, 8, 16\}$.

All experimental scripts, input files and log files are available online[6]. The experiments are performed on a cluster of Dell PowerEdge M610 servers with two Xeon E5520 processors and 24 GB internal memory each. The tools are compiled with gcc 5.4.0 on Ubuntu 16.04. The experiments for up to 48 cores are performed on a single computer with 4 AMD Opteron 6168 processors with 12 cores each and 128 GB internal memory.

When reporting on parallel executions, we use *the number of workers* for how many hardware threads (cores) were used.

Overview After running all experiments, we obtain the results for 413 models in total, of which 196 models with the Force variable ordering and 217 models with the Sloan variable ordering. In the remainder of this section, we study these 413 benchmarks. See Table 2, which shows the number of models for which each method could compute the set of reachable states within 20 minutes.

To correctly compare all runtimes, we restrict the set of models to those where all methods finish within 20 minutes with any number of workers. We retain in total 301 models where no solver hit the timeout. See Table 3 for the cumulative times for each method and number of workers and the parallel speedup. Notice that this is the speedup for the *entire* set of 301 models and not for individual models.

Comparing LDD saturation with Meddly's saturation We evaluate how ldd-sat with just 1 worker compares to the sequential saturation of Meddly. The goal is not to directly measure whether there is a parallel overhead from using parallelism in Sylvan, as the algorithm in `lddmc` is fundamentally different because it uses LDDs instead of MDDs and the algorithm does not in-place saturate nodes, as also explained in Sec. 3. The low parallel overheads of Sylvan are already demonstrated elsewhere [15, 16, 18]. Rather, the goal is to see how our version of saturation compares to the state-of-the-art.

Method	Order	Total time (sec) with # workers					Total speedup			
		1	2	4	8	16	2	4	8	16
otf-ldd-sat	Sloan	1850	1546	698	398	313	1.2	2.7	4.6	5.9
ldd-sat	Sloan	932	609	311	194	151	1.5	3.0	4.8	6.2
ldd-chaining	Sloan	4156	3019	1916	1121	863	1.4	2.2	3.7	4.8
ldd-bfs	Sloan	9030	5585	2990	1652	1219	1.6	3.0	5.5	7.4
bdd-sat	Sloan	708	419	212	139	115	1.7	3.3	5.1	6.1
mdd-sat	Sloan	572	–	–	–	–	–	–	–	–
otf-ldd-sat	Force	2704	1162	712	401	343	2.3	3.8	6.8	7.9
ldd-sat	Force	856	602	348	216	180	1.4	2.5	4.0	4.7
ldd-chaining	Force	3149	2560	1835	1160	1024	1.2	1.7	2.7	3.1
ldd-bfs	Force	4696	2951	1556	859	633	1.6	3.0	5.5	7.4
bdd-sat	Force	1041	733	384	253	206	1.4	2.7	4.1	5.1
mdd-sat	Force	1738	–	–	–	–	–	–	–	–

Table 3: Cumulative time and parallel speedups for each method-#workers combination on the models where all methods solved the model in time. These are 301 models in total: 151 models with Force, 150 models with Sloan.

Table 2 shows that Meddly's implementation (mdd-sat) and our implementation (ldd-sat 1) are quite similar in the number of solved models. Meddly solves 375 benchmarks and our implementation solves 388 within 20 minutes.

See Table 3 for a comparison of runtimes. Meddly solves the 150 models with Sloan almost 2× as fast as our implementation in Sylvan, but is slower than our implementation for the 151 models with Force. We observe for individual models that the difference between the two solvers is within an order of magnitude for most models, although there are some exceptions. Our implementation quickly overtakes Meddly with additional workers.

Parallel Scalability As shown in Table 3, using 16 workers, we obtain a modest parallel speedup for saturation of 6.2× (with Sloan) and 4.7× (with Force). On individual models, the differences are large. The average speedup of the individual benchmarks is only 1.8× with 16 workers, but there are many slowdowns for models that take less than a second with 1 worker. We take an arbitrary selection of models with a high parallel speedup and run these on a dedicated 48-core machine. Table 4 shows that even up to 48 cores, parallel speedup keeps improving. We even see a speedup of 52.2×. For this superlinear speedup we have two possible explanations. One is that there is some nondeterminism inherent in any parallel computation; another is already noted in [20] and is related to the "chaining" in saturation, see further [20].

Comparing LDD saturation with BDD saturation As Table 3 shows, the ldd-sat and bdd-sat method have a similar performance and similar parallel speedups.

Model (with ldd-sat)	**Order**	**Time (sec)**			**Speedup**	
		1	24	48	24	48
Dekker-PT-015	Sloan	77.3	4.7	2.4	16.3	32.5
PhilosophersDyn-PT-10	Force	273.8	16.8	12.4	16.3	22.1
Angiogenesis-PT-10	Sloan	333.2	28.5	16.5	11.7	20.2
SwimmingPool-PT-02	Force	25.0	2.1	1.4	11.6	17.8
BridgeAndVehicles-PT-V20P10N20	Force	1035.8	101.8	60.7	10.2	17.1
Model (with otf-ldd-sat)						
Dekker-PT-015	Sloan	174.5	7.4	3.3	23.6	52.2
SwimmingPool-PT-07	Sloan	1008.0	69.2	42.0	14.6	24.0
SmallOperatingSystem-PT-MT0256DC0064	Sloan	957.3	52.9	40.0	18.1	23.9
Kanban-PT-0050	Sloan	940.6	78.7	48.9	11.9	19.2
TCPcondis-PT-10	Force	68.4	5.7	3.8	11.9	17.8

Table 4: Parallel speedup for a selection of benchmarks on the 48-core machine (only top 5 shown)

On-the-fly LDD saturation Comparing the performance of offline saturation with on-the-fly saturation, we observe the same scalability with the Sloan variable order, but on-the-fly saturation requires roughly 2× as much time. With the Force variable order, on-the-fly saturation is slower but has a higher parallel speedup of 7.9×.

Comparing saturation, chaining and BFS We also compare the saturation algorithm with other popular strategies to compute the set of reachable states, namely standard (parallelized) BFS and chaining, given in Figure 5. As Table 2 and Table 3 show, chaining is significantly faster than BFS and saturation is again significantly faster than chaining. In terms of parallel scalability, we see that parallelized BFS scales better than the others, because it can already parallelize in the main loop by computing successors for all relations in parallel, which chaining and saturation cannot do. For the entire set of benchmarks, saturation is the superior method, however there are individual differences and for some models, saturation is not the fastest method.

6 Conclusion

We presented a multi-core implementation of saturation for the efficient computation of the set of reachable states. Based on Sylvan's multi-core decision diagram framework, the design of the saturation algorithm is mostly orthogonal to the type of decision diagram. We showed the implementation for BDDs and LDDs; the translation relation can be learned on-the-fly. The functionality is accessible through the LTSmin high-performance model checker. This makes parallel saturation available for a whole collection of asynchronous specification

```
global: $N$ transition relations $R_0 \dots R_{M-1}$
def bfs($S$):
    $U \leftarrow S$
    while $U \neq \emptyset$ :
        $U \leftarrow$ par-next($U$, 0, $M$)
        $U \leftarrow$ minus($U$, $S$)
        $S \leftarrow$ union($U$, $S$)
    return $S$
def par-next($S$, $i$, $k$):
    if $k = 1$ : return relprod($S$, $R_i$)
    do in parallel:
        left $\leftarrow$ par-next($S$, $i$, $k/2$)
        right $\leftarrow$ par-next($S$, $i + k/2$, $k - k/2$)
    return union(left, right)
```

```
def chaining($S$):
    $U \leftarrow S$
    while $U \neq \emptyset$ :
        for $i \in [0, M)$ :
            $U \leftarrow$ relprodunion($U$, $R_i$)
        $U \leftarrow$ minus($U$, $S$)
        $S \leftarrow$ union($U$, $S$)
    return $S$
```

Fig. 5: Algorithms bfs and chaining implement the Parallel BFS and Chaining strategies for reachability.

languages. We demonstrated multi-core saturation for Promela and for Petri nets in PNML representation.

We carried out extensive experiments on a benchmark of Petri nets from the Model Checking Contest. The total speedup of on-the-fly saturation is 5.9× on 16 cores with the Sloan variable ordering and 7.9× with the Force variable ordering. However, there are many small models (computed in less than a second) in this benchmark. For some larger models we showed an impressive 52× speedup on a 48-core machine. From our measurements, we further conclude that the efficiency and parallel speedup for the BDD variant is just as good as the speedup for LDDs. We compared efficiency and speedup of saturation versus other popular exploration strategies, BFS and chaining. As expected, saturation is significantly faster than chaining, which is faster than BFS; this trend is maintained in the parallel setting. Our measurements show that the variable ordering (Sloan versus Force), and the model representation (pre-computed transition relations versus learned on-the-fly) do have an impact on efficiency and speedup. Parallel speedup should not come at the price of reduced efficiency. To this end, we compared our parallel saturation algorithm for one worker to saturation in Meddly. Meddly solves fewer models within the timeout, but is slightly faster in other cases, but parallel saturation quickly overtakes Meddly with multiple workers.

Future work could include the study of parallel saturation on exciting new BDD types, like tagged BDDs and chained BDDs [8, 19]. The results on tagged BDDs showed a significant speedup compared to ordinary BDDs on experiments in LTSmin with the BEEM benchmark database. Another direction would be to investigate the efficiency and speedup of parallel saturation in other applications, like CTL model checking, SCC decomposition, and bisimulation reduction.

Bibliography

[1] Aloul, F.A., Markov, I.L., Sakallah, K.A.: FORCE: a fast and easy-to-implement variable-ordering heuristic. In: VLSI 2003. pp. 116–119. ACM (2003)

[2] Amparore, E.G., Beccuti, M., Donatelli, S.: Gradient-based variable ordering of decision diagrams for systems with structural units. In: ATVA. Lecture Notes in Computer Science, vol. 10482, pp. 184–200. Springer (2017)

[3] Amparore, E.G., Donatelli, S., Beccuti, M., Garbi, G., Miner, A.S.: Decision diagrams for Petri nets: Which variable ordering? In: PNSE @ Petri Nets. CEUR Workshop Proceedings, vol. 1846, pp. 31–50. CEUR-WS.org (2017)

[4] Babar, J., Miner, A.S.: Meddly: Multi-terminal and edge-valued decision diagram library. In: QEST. pp. 195–196. IEEE Computer Society (2010)

[5] Blom, S., van de Pol, J., Weber, M.: LTSmin: Distributed and Symbolic Reachability. In: CAV. LNCS, vol. 6174, pp. 354–359. Springer (2010)

[6] Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Trans. Computers 45(9), 993–1002 (1996)

[7] Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. Computers C-35(8), 677–691 (8 1986)

[8] Bryant, R.E.: Chain reduction for binary and zero-suppressed decision diagrams. In: TACAS (1). Lecture Notes in Computer Science, vol. 10805, pp. 81–98. Springer (2018)

[9] Chung, M., Ciardo, G.: Saturation NOW. In: QEST. pp. 272–281. IEEE Computer Society (2004)

[10] Chung, M., Ciardo, G.: Speculative image computation for distributed symbolic reachability analysis. J. Log. Comput. 21(1), 63–83 (2011)

[11] Ciardo, G., Lüttgen, G., Siminiceanu, R.: Saturation: An efficient iteration strategy for symbolic state-space generation. In: TACAS. Lecture Notes in Computer Science, vol. 2031, pp. 328–342. Springer (2001)

[12] Ciardo, G., Marmorstein, R.M., Siminiceanu, R.: Saturation unbound. In: TACAS. Lecture Notes in Computer Science, vol. 2619, pp. 379–393. Springer (2003)

[13] Ciardo, G., Marmorstein, R.M., Siminiceanu, R.: The saturation algorithm for symbolic state-space exploration. STTT 8(1), 4–25 (2006)

[14] Ciardo, G., Zhao, Y., Jin, X.: Parallel symbolic state-space exploration is difficult, but what is the alternative? In: PDMC. EPTCS, vol. 14, pp. 1–17 (2009)

[15] van Dijk, T.: Sylvan: Multi-core Decision Diagrams. Ph.D. thesis, University of Twente (July 2016)

[16] van Dijk, T., van de Pol, J.: Sylvan: multi-core framework for decision diagrams. STTT 19(6), 675–696 (2017)

[17] van Dijk, T., van de Pol, J.: Multi-core symbolic bisimulation minimisation. STTT 20(2), 157–177 (2018)

[18] van Dijk, T., van de Pol, J.C.: Lace: Non-blocking split deque for work-stealing. In: Euro-Par 2014: Parallel Processing Workshops Part II. pp. 206–217 (2014)

[19] van Dijk, T., Wille, R., Meolic, R.: Tagged BDDs: Combining reduction rules from different decision diagram types. In: FMCAD. pp. 108–115. IEEE (2017)

[20] Ezekiel, J., Lüttgen, G., Ciardo, G.: Parallelising symbolic state-space generators. In: CAV. Lecture Notes in Computer Science, vol. 4590, pp. 268–280. Springer (2007)

[21] Heyman, T., Geist, D., Grumberg, O., Schuster, A.: A scalable parallel algorithm for reachability analysis of very large circuits. Formal Methods in System Design 21(3), 317–338 (2002)

[22] Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: High-Performance Language-Independent Model Checking. In: TACAS 2015. LNCS, vol. 9035, pp. 692–707. Springer (2015)

[23] Konnov, I., Letichevsky, O.: Model checking GARP protocol using Spin and VRS. In: IW on Automata, Algorithms, and Information Techn. (2010)

[24] Kordon, F., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Chiardo, G., Hamez, A., Jezequel, L., Miner, A., Meijer, J., Paviot-Adet, E., Racordon, D., Rodriguez, C., Rohr, C., Srba, J., Thierry-Mieg, Y., Trịnh, G., Wolf, K.: Complete Results for the 2016 Edition of the Model Checking Contest. http://mcc.lip6.fr/2016/results.php (June 2016)

[25] Meijer, J., Kant, G., Blom, S., van de Pol, J.: Read, Write and Copy Dependencies for Symbolic Model Checking. In: HVC. pp. 204–219 (2014)

[26] Meijer, J., van de Pol, J.: Bandwidth and wavefront reduction for static variable ordering in symbolic reachability analysis. In: NFM. Lecture Notes in Computer Science, vol. 9690, pp. 255–271. Springer (2016)

[27] Noack, A.: A ZBDD package for efficient model checking of Petri nets. Forschungsbericht, Branderburgische Technische Uinversität Cottbus (1999)

[28] Oortwijn, W., van Dijk, T., van de Pol, J.: Distributed binary decision diagrams for symbolic reachability. In: 24th ACM SIGSOFT Intl. SPIN Symposium on Model Checking of Software. pp. 21–30 (2017)

[29] Siminiceanu, R., Ciardo, G.: New metrics for static variable ordering in decision diagrams. In: TACAS. Lecture Notes in Computer Science, vol. 3920, pp. 90–104. Springer (2006)

[30] Sloan, S.W.: A FORTRAN program for profile and wavefront reduction. International Journal for Numerical Methods in Engineering 28(11), 2651–2679 (1989)

[31] Vörös, A., Szabó, T., Jámbor, A., Darvas, D., Horváth, Á., Bartha, T.: Parallel saturation based model checking. In: ISPDC. pp. 94–101. IEEE Computer Society (2011)

[32] Zhao, Y., Ciardo, G.: Symbolic CTL model checking of asynchronous systems using constrained saturation. In: ATVA. LNCS, vol. 5799, pp. 368–381. Springer (2009)

[33] Zhao, Y., Ciardo, G.: Symbolic computation of strongly connected components and fair cycles using saturation. ISSE 7(2), 141–150 (2011)